
Build APIs You Won't Hate 2:
This Time It's Serious
200 OK
{
"errors": {
PHILIP STURGEON

Build APIs You Won’t Hate

Philip Sturgeon

Table of Contents

About the Author

You can contact Philip Sturgeon at phil@apisyouwonthate.com.

Some sort of bio in here. Bikes bikes bikes. Turtles.

Introduction

This book is my second effort at writing down everything I know about APIs. *Build APIs You Wont Hate* was essentially every complicated design I had to navigate when building an API for a mobile-centric startup that was basically a combination of Foursquare, Groupon, and Instragram. I had to think about naming endpoints, writing documentation for designers, shaping the response body JSON, was curious about which status codes to use, etc.

Then after the initial ebook was released in 2014 I went on to work at a carpooling company that had a few APIs, with folks spread out all over the world. That helped me understand the need for HATEOAS (explained later) a whole lot more, as we were inferring state in all sorts of weird ways and getting it wrong a lot. The folks on that team also helped me learn loads about various forms of integration testing, load testing, standards like JSON:API, and a whole lot more.

The original book was just my attempt to help readers navigate complex decisions, decide between alternative approaches, outline the potential pitfalls of various strategies, etc., but these days when I read things I wrote back then it feels dated. Even though I don't feel like much of it was outright incorrect, I do disagree with many of the conclusions I came to back then. Modern-day Phil writing this in 2019 has a lot more information to go on.

Beyond the gaps in my knowledge, so much has changed in API land between 2014 and 2019. For a while a lot of these changes were incremental and I could just update the book a little for the ebook readers, or expand things with a blog post. Now I feel like the time has come to start from scratch and write a new book, because having people read a whole book then 35+ articles just felt a little silly.

Back in 2014, a lot of folks (myself included) were acting like REST was the only way to do things, and older systems like SOAP (or anything RPC-based) were bad and silly. These days RPC in the form of gRPC or GraphQL is all the rage, and some people are acting like REST is bad and silly. Comically at that time most of us were building "RESTish" APIs anyway due to lack of understanding in the benefits of HATEOAS, and REST without HATEOAS is basically RPC with pretty URLs. When 99% of the API developers out there discuss REST vs RPC it's a fairly moot point, because they're talking about essentially the same thing until they learn the strengths and weakness of both paradigms correctly. This book will help unfurl this mess.

Another huge change in HTTP-land is HTTP/2 becoming considerably more available, and viable as a choice. Back when I was writing *Build APIs You Wont Hate*, nothing much supported HTTP/2. This made it feel more like more of an academic concept, which we could look forward to, but not actually use yet. Now most web servers, several languages, and a whole bunch of frameworks support it, either fully or mostly. The only thing holding API developers back from more usage is education and momentum. HTTP/2 changes pretty much everything about API design, so we need to look at it a lot more closely. It fundamentally changes how you design everything, as the sort of things you find yourself trying to solve in HTTP/1.1 at the application level are handled for you so much more seamlessly in HTTP/2.

In the past a lot of the API specification tools being used for HTTP-based APIs were… fairly bad. Folks were just using them for documentation (if that), and often using annotation-based approaches, because the tooling out there to help design specifications didn't exist. Nobody wants

to write a bunch of JSON/YAML by hand, so we just skipped it all, and maybe wrote some docs later, or maybe just didn't bother. Now API Specification tooling has grown so much its almost unrecognizable.

Previously I'd recommend people think about their resources, which helps guide which endpoints we need, then jump right into creating serializers to output data, then we start writing the code to populate those serializers. These days I'd flip that entirely, and start with API specifications first, then use those specifications to build mock servers to get client feedback early, then use those specifications as contracts for contract testing on API responses, and a whole lot more.

So, the times have changed, and the approach for this book will feel a bit different to readers of the last. For example, as well as learning about REST properly (which few books have done in a way easy enough for the majority of developers to understand), we're going to learn about the other two popular technologies: GraphQL and gRPC. Many chapters will be broken down into sections which explain how certain things work for each of these technologies, so you can understand and compare the differences, or skip sections if you are not interested in using those technologies.

Another big difference will be a lack of sample application code. The previous book was basically "How to build RESTish APIs with Laravel" with a lot of theory and occasional code samples for other languages. If you are looking for a "How to build gRPC APIs with Go" or "How to build GraphQL APIs with Erlang" or any other language or technology specific guide, then this book is not going to be for you. We are going to cover REST, gRPC and GraphQL equally, we're going to use examples from various languages, and there will not be a "build along with us as you code" or example application bundled with the book, as they get outdated too fast and keeping book code up to date for a decade is not enjoyable or realistic.

So, settle in, and lets learn a whole bunch of stuff about building APIs!

Part One: Theory

Chapter 1. Understanding Different Paradigms

If APIs are the interface for some sort of service, it should not come as a surprise that not all the interfaces in the world are going to have the same needs and requirements. Most of them are a case of "execute a command", "interact with data", or a combination of both.

There are three distinct types of Web API which handles these common needs, and these different types are known as paradigms. To oversimplify things a bit, it's reasonably fair to say that all APIs conform to one of these paradigms:

- RPC (Remote Procedure Call)
- REST (Representational State Transfer)
- Query Languages

These paradigms are general approaches to building APIs, not a specific tool or specification. They are merely an idea or set of ideas, not a tangible thing. For that, we need to look at implementations.

Implementations are something you can actually download, install, and use to build an API, that conforms to whatever rules the implementors picked, from whichever paradigm (or paradigms) they were interested in at the time.

Specifications (or standards, recommendations, etc.) are often drafted up by various working groups, to help implementations share functionality in the same way. An API and a client in different languages can work together perfectly if they're all following the specification correctly. The difference between a standard and a specification is usually just down to who made it. When a working group like the IETF or W3C create something, it's a standard, but when Google knock something out on their own, it's a specification.

For example:

- SOAP is a W3C recommendation, following the RPC paradigm, with implementations like gSOAP
- gRPC is an implementation, following the RPC paradigm, which has no standard defined by any working group, but authors Google Inc. did document the protocol in a specification
- REST is a paradigm, which has never been turned into a specification, and has no official implementations, but building a REST API is usually just a case of picking appropriate standards and tooling

Making direct comparisons between any of these things is tough because it should be possible to see that paradigms, implementations and specifications are all rather different. We'll be covering all the paradigms, and we will be looking at how things work in a few of the more popular implementations throughout the book.

First, let us try and figure out the main conceptual differences between the paradigms.

1.1. Remote Procedure Call (RPC)

RPC is the earliest, simplest form of API interaction. It is about executing a block of code on another server, and when implemented in HyperText Transfer Protocol (HTTP) or Advanced Message Queuing Protocol (AMQP) it can become a Web API. There is a method and some arguments, and that is pretty much it. Think of it like calling a function in JavaScript, taking a method name and arguments.

For example:

```
POST /sayHello HTTP/1.1
HOST: api.example.com
Content-Type: application/json

{"name": "Racey McRacerson"}
```

In JavaScript, the equivalent concept is defining a function, and calling it elsewhere:

```
/* Signature */
function sayHello(name) {
  // ...
}

/* Usage */
sayHello("Racey McRacerson");
```

The idea is the same. An API is built by defining public methods; then, the methods are called with arguments. RPC is just a bunch of functions, but in the context of an HTTP API, that entails putting the method in the URL and the arguments in the query string or body.

When used for CRUD ("Create, Read, Update, Delete"), RPC is just a case of sending up and down data fields. This can be fine, but it also means the client is in charge of pretty much everything. The client must know which methods to hit, and at what time, in order to construct its own workflow out of a bevy of methods, with only human interaction or written interactions to help the client application developer figure out what to do and in which order. That might be a positive, or a negative, depending on what sort of relationship you want your clients and servers to have.

RPC is without doubt the most prominent paradigm used in API land, possibly because it feels so natural to many programmers. Calling a function locally and calling a function remotely feel so similar, that it just clicks with a lot of developers.

That said, there are a few other things to figure out, like should the method name go in the URL, or should it be passed in the body. Should it be entirely POST or a combination of GET and POST? Should we use metadata to describe what the payload data is? To answer questions like this, RPC has a whole bunch of specifications, all of which have concrete implementations:

Older popular RPC standards

- XML-RPC
- JSON-RPC
- SOAP (Simple Object Access Protocol)

XML-RPC and JSON-RPC are not used all that much other than by a minority of entrenched fanatics, but SOAP is still kicking around for a lot of financial services and corporate systems like Salesforce.

XML-RPC was problematic, because ensuring data types of XML payloads is tough. In XML, a lot of things are just strings, which JSON does improve, but has trouble differentiating different data formats like integers and decimals.

You need to layer metadata on top in order to describe things such as which fields correspond to which data types. This became part of the basis for SOAP, which used XML Schema and a WSDL (Web Services Description Language) to explain what went where and what it contained.

This metadata is essentially what most science teachers drill into you from a young age: "label your units!". The sort of thing that stops people paying $100 for something that should have been $1 but was just marked as `price: 100` which was meant to be cents…

All three of these specifications had implementations created by various people, mostly as open-source projects. Occasionally the folks who put the standard together created an official implementation in their favourite language, and the community built their own in other languages they wanted to use.

A modern RPC specification is gRPC, which can easily be considered modern SOAP. It uses a data format called Protobuf, which requires a schema file as well as the data instance, much like the WSDL in SOAP. This Protobuf file is shared with both the client and the server, and then messages can be verified and passed between the server and client in binary, which leads to smaller messages than passing around big chunks of JSON.

gRPC focuses on making single RPC interactions, and it aims to achieve this as quickly as possible, thanks to the aforementioned binary benefits, and its other huge benefit: HTTP/2. All of the gRPC implementations are HTTP/2 by default, and usually handle this with their own built-in web server to make sure HTTP/2 works the whole way through the transaction.

1.2. Representational State Transfer (REST)

REST is a network paradigm, published by Roy Fielding in a dissertation in 2000. REST is all about a client-server relationship, where server-side data are made available through representations of data in simple formats. This format is usually JSON or XML but could be anything (including Protobuf).

These representations portray data from various sources as simple "resources", or "collections" of resources, which are then potentially modifiable with actions and relationships being made discoverable via a concept known as HATEOAS (Hypermedia as the Engine of Application State). That's a rough acronym that confuses folks, so many API people just use the term "Hypermedia Controls" to mean the same thing.

Hypermedia controls are fundamental to REST. It is merely the concept of providing "next available

actions", which could be related data, or more often it's actions available for that resource in its current state, like having a "pay" option for an invoice that has not yet been paid.

These actions are just links, but the idea is the client knows that an invoice is payable by the presence of a "pay" link, and if that link is not there it should not show that option to the end user.

```
{
  "data": {
    "type": "invoice",
    "id": "093b941d",
    "attributes": {
      "created_at": "2017-06-15 12:31:01Z",
      "sent_at": "2017-06-15 12:34:29Z",
      "paid_at": null,
      "status": "published"
    }
  },
  "links": {
    "pay": "https://api.acme.com/invoices/093b941d/payment_attempts"
  }
}
```

This is quite different to RPC. Imagine the two approaches were humans answering the phones for a doctors office:

Client: Hi, I would like to speak to Dr Watson, is he there?

RPC: No. **click**

Client calls back

Client: I found his calendar and luckily I know how to interact with the Google Calander API. I have checked his availability, and it looks like he is off for the day. I would like to visit another doctor, and it looks like Dr Jones is available at 3pm, can I see her then?

RPC: Yes.

The burden of knowing what to do is entirely on the client, and this can lead to "fat clients" (i.e: the client contains a lot of business logic). It needs to know all the data, come to the appropriate conclusion itself, then has to figure out what to do next.

REST, however, presents you with the next available options:

Client: Hi, I would like to speak to Dr Watson, is he there?

REST: Doctor Watson is not currently in the office, he'll be back tomorrow, but you have a few options. If it's not urgent you could leave a message and I'll get it to him tomorrow, or I can book you with another doctor, would you like to hear who is available today?

Client: Yes, please let me know who is there!

REST: Doctors Smith and Jones, here are links to their profiles.

Client: Ok, Doctor Jones looks like my sort of Doctor, I would like to see them, let's make that appointment.

REST: Appointment created, here's a link to the appointment details.

REST provided all of the relevant information with the response, and the client was able to pick through the options to resolve the situation.

None of this is magic, no client is going to know exactly what to do without being trained, but the client of a REST API can be told to follow the `"alternative_doctors": "https://api.example.com/available_doctors?available_at=2017-01-01 03:00:00 GMT"` link. That is far less of a burden on the client than expecting it to check the calendar itself, seek for availability, etc.

This centralization of state into the server has benefits for systems with multiple different clients who offer similar workflows. Instead of distributing all the logic, checking data fields, showing lists of "Actions", etc. around various clients—who might come to different conclusions—REST keeps it all in one place.

This book will get more in-depth on hypermedia controls later. There are a few other important things to understand about REST APIs first:

- REST must be stateless: not persisting sessions between requests
- Responses should declare cacheablility: helps your API scale if clients respect the rules
- REST focuses on uniformity: if you're using HTTP you should utilize HTTP features whenever possible, instead of inventing conventions

These constraints of REST when applied to HTTP APIs can help the API last for decades, which is a whole lot more complex without these concepts. What does that mean? Well, REST is often described as a series of layers of abstraction on top of RPC, with all relevant instructions related to the handling of that message being baked into the message itself, to avoid having to tell a human about specific ways to handle things. As things change throughout the entire ecosystem, a well trained REST API and client should be able to handle those changes seamlessly, because the REST API is describing itself well and the client is listening. This loosens the coupling found in other paradigms, where a lot of that is baked into the client itself.

Some folks look at all this and do not understand why REST requires "all the extra faffing about". There are many who just do not quite get the point of any of it, and consider RPC to be the almighty. To them, it is all about executing the remote code as fast possible, but REST (which can still absolutely be performant) focuses far more on longevity and reduced client-coupling instead. Knowing when to have a fat client and when to have a skinny client is a powerful decision making process to have in your arsenal, so definitely do not be one of those people who things it should always be A, or always be B.

Another interesting thing about REST is that it does not require the use of schema metadata (like WSDL or similar), but does allow it. In fact, REST has no opinions either way: it does not explicitly demand it, nor disallow it. The metadata is something many API developers hated about SOAP, and

from 2015 to current day, it has become more and more of a growing trend once again, thanks to gRPC and GraphQL including and requiring type systems in their implementations.

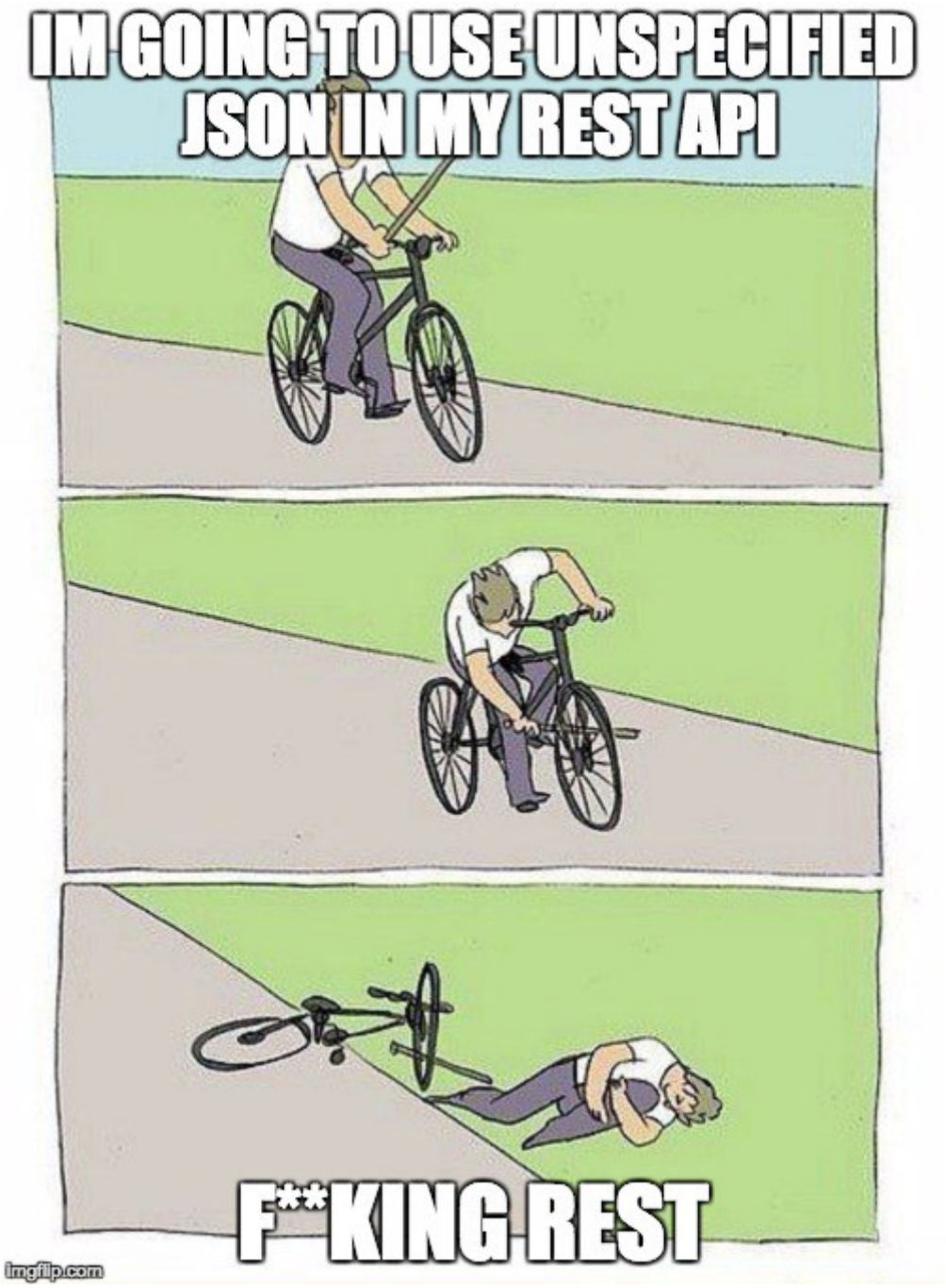


Figure 1. It's pretty common to blame REST for not forcing a type system onto everyone involved in the API, but they're absolutely useable. Don't blame a paradigm for implementation level decisions.

The HTTP community (building REST or whatever) has a few type systems available for optional use, the main one these days being: JSON Schema. JSON Schema is inspired by XML Schema—but not functionally identical—and is one of the most important things to happen to HTTP APIs in years, and will be discussed a lot throughout the book.

Unfortunately, REST became a marketing buzzword for most of 2006–2014. It became a metric of quality that developers would aspire to, fail to understand, then label as REST anyway. Most systems saying they are REST are little more than RPC with HTTP verbs and pretty URLs. As such, you might not get cacheability provided, it might have a bunch of wacky conventions, and there might not be any links for you to use to discover next available actions. These APIs are jovially

called REST*ish* by people aware of the difference.

REST has no specification which is what leads to some of this confusion, nor does it have concrete implementations. That said, there are two large popular API standards designed to standardize certain aspects of REST APIs that chose to use them:

- JSON:API
- OData

If the API advertises itself as using one of these, you will be able to find a whole bunch of tooling that will work out of the box with this API, meaning you can get going quicker. Otherwise you will have to go at it yourself with a common HTTP client, which is fine with a little bit of elbow grease.

This book will look more at these two formats and others, as they are hugely important for avoiding bikeshedding over the implementation of trivial features and already solved problems.

1.3. Query Languages

Query languages are designed to give huge flexibility to the client, to make very specific requests, beyond a few simple arguments. Imagine a client asking for an a RPC endpoint to create a very specific report, like asking for a list of companies with unpaid invoices in the last 3 months. You would end up with `POST /getCompaniesByUnpaidRecently(">= 3 months")` or something very specific.

A good query language lets the client treat the API like a data store, and do whatever it wants -within its permissions.

There are more query languages out there than there are amateur food bloggers at a NYC restaurant opening, but only some of them specifically aim to solve things for Web APIs.

For example, you could probably take some standard SQL, pipe it over an HTTP endpoint `POST /sql` and call it an API, but you probably don't want to do that for a few thousand reasons.

1.3.1. SPARQL (2008)

First released in 2008 and finally making it to be a W3C Recommendation in 2013, SPARQL sets out to handle some rather complex queries.

```
SELECT ?human ?humanLabel
WHERE
{
    ?human wdt:P31 wd:Q5 .       #find humans
    ?human rdf:type wdno:P40 .   #with at least one P40 (child) statement defined to
be "no value"
    SERVICE wikibase:label { bd:serviceParam wikibase:language "[AUTO_LANGUAGE],en" }
}
```

Another one popped up in 2008 called FIQL, which is a little easier to parse by human and by computer.

```
title==foo*;(updated=lt=-P1D,title==*bar)
```

Here is an example looking for a title beginning with "foo", and which has either been updated in the last day, or has a title ending with "bar". That's a powerful query for such a simple syntax.

You could easily imagine shoving this into a query string:

```
/games?filter= + urlencode("title==foo*;(updated=lt=-P1D,title==*bar)")
```

This made it to an IETF Draft, but never became a final standard.

There were other query languages floating around too, but none of those attempts to create query languages ever really made it into the mainstream. Their usage seemed mostly restricted to academic purposes, with folks in universities, libraries, etc., all finding interesting uses, but there are no popular content management systems built around them, and we certainly didn't see the major tech players, and hot new startups, or anyone really building out things using these query languages.

There was one exception to that, and Facebook actually did have a lesser-known query language based API called FQL (not FIQL). It was their own implementation, they never open-sourced any of it, and despite being a bit weird to work with it was really useful for edge cases that a normal API might not quite be able to answer. You could make a query like "Get me an avatar for all of my friends who live in the UK" or "What is the surname of everyone I know who has a Pet" or any other arbitrary query that popped into your head.

```
GET /fql?q=SELECT uid2 FROM friend WHERE uid1=me()&access_token=abc123
```

Facebook got a bit fed up with having a RESTish approach to get data, and then having the FQL approach for more targeted queries as well, as they both require different code. As such, they ended up creating a middle-ground between "endpoint-based APIs" (a term they use to group REST/RESTish) and FQL. This middle-ground solution was known as GraphQL, which was released publicly as a specification with a few official implementations in 2015.

GraphQL is essentially a RPC-based query language system, where the client is required to ask for specific resources, and also list the specific fields they are interested in receiving back. The GraphQL API will then return only those fields in the response.

Figure 2. GraphQL Request (left) and the corresponding response (right).

Any sort of RPC action which is intended to cause modifications is done with a "Mutation". SO to handle creates, updates, deletes, etc. you would create a mutation.

Figure 3. Definition of a mutation (top left), the mutation request (bottom left), and the response (right).

Facebook chose to ignore most of the conventions of the transportation layer so they could focus on building their own conventioins. So although it is often used with HTTP, endpoints are gone,

resources are gone, HTTP methods are gone, and most implementations are a single `POST /graphql` endpoint. This means that resources declaring their own cacheability is gone,and the concept of the uniform interface (as REST defines it) is obliterated.

All of this has the advertised benefit of making GraphQL portable enough that it could fit into AMQP, or any other transportation protocol which is something REST could theoretically do but nobody ever bothers with due to the amount of crowbarring it would take.

GraphQL has many fantastic features and benefits, which are all bundled in one package, with a nice marketing site. If you are trying to learn how to make calls to a GraphQL API, the Learn GraphQL documentation will help, and their site has a bunch of other resources.

Seeing as GraphQL was built by Facebook, who had previously built a REST*ish* API, they're familiar with various REST/HTTP API concepts. Many of those existing concepts were used as inspiration for GraphQL functionality, or carbon copied straight into GraphQL.

The main selling point of GraphQL is that it defaults to providing the very smallest response from an API, as you are requesting only the specific bits of data that you want, which minimizes the Content Download portion of the HTTP request.

It also reduces the number of HTTP requests necessary to retrieve data for multiple resources, known as the "HTTP N+1 Problem" that has been a problem for API developers through the lifetime of HTTP/1.1. Basically a lot of RPC APIs - and poorly designed REST APIs - would give you a list of resources in the first request, but then to get further information clients would need to make another request for each resource. This means for a list with 10 resources, the client would need to make 11 (1+10) requests to fetch everything it needed. This has been the bane of HTTP/1.1 developers for years, but GraphQL has provided one consistent solution to this via fetching multiple resources in a single request, very similar to how JSON:API and OData have done in the past.

GraphQL is a strong and relatively simple solution to the majority of issues that Web API developers ran into in a HTTP/1.1 world, with clients who do not care about Hypermedia Controls. Their optimizations and the way they've built their own conventions inside HTTP mean they're kinda stuck unable to leverage HTTP/2 fully, and comically when folks design their APIs with HTTP/2 in mind, most of what GraphQL is aiming to do actually hurts the clients, and makes things slower. A lot of folks see it as REST 2.0, but that is probably down to the marketing hype machine, and not down to education.

Let's learn some stuff about GraphQL, and see when its useful, and when its not!

1.4. I'm Lost!

Fair comment. This has been a whirlwind tour of a whole bunch of different topics which will be covered more in depth later. If you are nodding and smiling already then you are going to enjoy this book as we delve deeper, and if you have no idea what's going on then you are in the right place too.

We will cover a lot of specifics about the technical and functional aspects of these various API paradigms and implementations, helping you to figure out when to uses which, and how to build the best possible API whichever of them you chose for a certain use case.

Chapter 2. API Contracts

The "I" in "API" means interface, and so writing a contract for that interface is the act of writing down what that interface is going to look like, to avoid folks having to guess, or go and hunt for some documentation somewhere.

Let's use the fantastic PokeAPI as an example.


```
{
  "id": 12,
  "name": "butterfree",
  "base_experience": 178,
  "height": 11,
  "is_default": true,
  "order": 16,
  "weight": 320,
  "abilities": [{
    "is_hidden": true,
    "slot": 3
  }]
}
```


If we look just at the data, we can deduce that sure, ID is probably an auto-incrementing identifier, the name seems to be a lower-cased English string, but after that it starts getting a bit odd.

Height and weight, I guess there is no imperial and metric in the game just units, but what is order all about? The order that pokemon is sat in my current party? That Ash bumped into in the cartoon? In order of cuteness? What?!

An API contract can help answer most of the questions and more for anyone wanting to understand how an API is meant to work.

- Which resources or methods are available and how do you interact with them?
- Which HTTP headers are required for that endpoint, for things like authentication?
- Which properties are available in the request and response?
- What are those various properties data types?
- Which of those properties are required or optional?
- What other validation rules can be applied to those properties?

> A contract is an agreement between two or more parties, especially one that is written and enforceable by law.
>
> — The American Heritage Dictionary of the English Language

Thinking about writing an API contract might sound a bit odd if you just think about an API as sharing some data or as a convenient way to trigger commands on another server, but people need

to know what that data is, or what arguments they should send to those commands.

The enforceable bit is important too. We can use those contracts in our test suite to confirm that our actual APIs are following the rules of the contract.

NOTE	Some people use the term "API Specifications" to describe all this, but the word "specification" is used to mean a lot of different things. We just used it in the last chapter for something totally different, so let's stick to API Contracts to avoid confusion.

Written down exactly from an early point (and agreed upon) means that there are far fewer surprises throughout the API life-cycle, from planning, through development, and into usage by clients.

Having a good contract means that API developers can be confident that:

1. The interface is doing what they intend
2. The interface is going to be useful for client developers
3. The interface is understood well by client developers
4. The interface is not changing accidentally when code changes

This confidence will save everyone a lot of time, money and frustration. Having no contract in place leads to slower rollout of the initial version, loads more time spent testing subsequent deployments, and wasted developer time having loads of meetings to explain things that could have been written down and clear to everyone already.

2.1. What forms can an API Contract take?

Some folks might think "That sounds like documentation", others might think "That sounds like tests". The answer is both, and more.

An API contract can exist in several forms, some more useful than others.

2.1.1. The Dreaded Verbal Contract

A common case is the verbal contract, where the API developers discuss it with the frontend or other client application developers as they go. The backend developers write their code, the client developers write theirs, and as various endpoints or properties are conceived, they are explained somehow: literally explained out loud over the conference room table, shouted over a hail of nerf gun pellets and ping pong, or DMed over Slack chats.

```
Fred: Hey Sarah, there's a new "fudge" field and it can be "blah" or "whatever".

Sarah: Great! Thanks I'll chuck that in now.
```

This might seem good enough as the API works, the app runs fine, the project/product managers are happy, and the company may even be making money off of this whole thing. Sadly this method has

a bunch of downsides.

The primary issue is that communication is hard. You never really know if somebody understood what you meant even if they say they did.

Beyond that, not having things written down means people can forget. If another client team also needs the information and they ask another client team, they are getting a copy of a copy which might be slightly wrong.

Another outcome is the original author(s) cannot remember the exact rules, leading to time wasted checking the code. An even worse scenario is that the original author(s) are not reachable for some reason. Maybe the whole team of developers who worked on that API are all on a flight together from New York to Uruguay, and for that entire 9 hour flight nobody can get any answers about how the API is supposed to work, so they're stuck not being able to fix a production issue… They might also have quit.

When folks rely on a verbal contract, two common things happen:

1. Developers waste time writing a new endpoint out of fear of using the old one
2. Developers waste time trying to guess the contract, and might guess wrong

2.1.2. Shoddy Human Contracts

When folks first start thinking about contracts, it often takes the form of rudimentary API Reference Documentation. This might be dumped into a Google Doc, shoved in a wiki, or written up in finger blistering HTML. These formats are usually fairly painstaking to create, because you are focusing on formatting *and* content, shuffling things around whenever you remember that you should list headers too, and copying and pasting examples of JSON which might change over time. API developers would not settle for writing software like this, but apparently when it comes to documentation it's accepted as the norm.

This whole slow, manual approach is probably why many people forgo writing contracts this way, and opt for verbal contracts instead. It probably would not be a huge jump to say the "startup mindset" (and "agile") are one of a few potential driving factors in this "just get on with it" approach. There are many technical leaders inexperienced in this area telling their team "We'll write lovely docs later when we've got more time!", without realizing their lack of clarity is slowing down initial development, client integrations, increasing testing efforts, and causing bizarre production issues and throughout their ecosystem.

Anyway, aside from this medium of API contracts being time consuming, it is also impossible to enforce. After you have spent the time writing up a Google Doc/HTML/Wiki, the only output of that is going to be… a Google Doc/HTML/Wiki and maybe a PDF if you want to go wild. You can't exactly jam that Google Doc into an API test suite and have it confirm that the API is conforming to whatever is written in there.

There is another way.

2.1.3. Description Languages

A description language can be text based or look a bit like a programming language. This format lets you describe an API in a reusable way, which means you can do a whole bunch of stuff:

- Documentation
- Client-side validation
- Server-side validation
- Client-library Generation (SDKs)
- UI Generation
- Server/Application generation
- Mock servers
- Contract testing
- Automated Postman/Paw Collections

An early example of that would be SOAP, which used something called a WSDL, something discussed in the previous chapter.

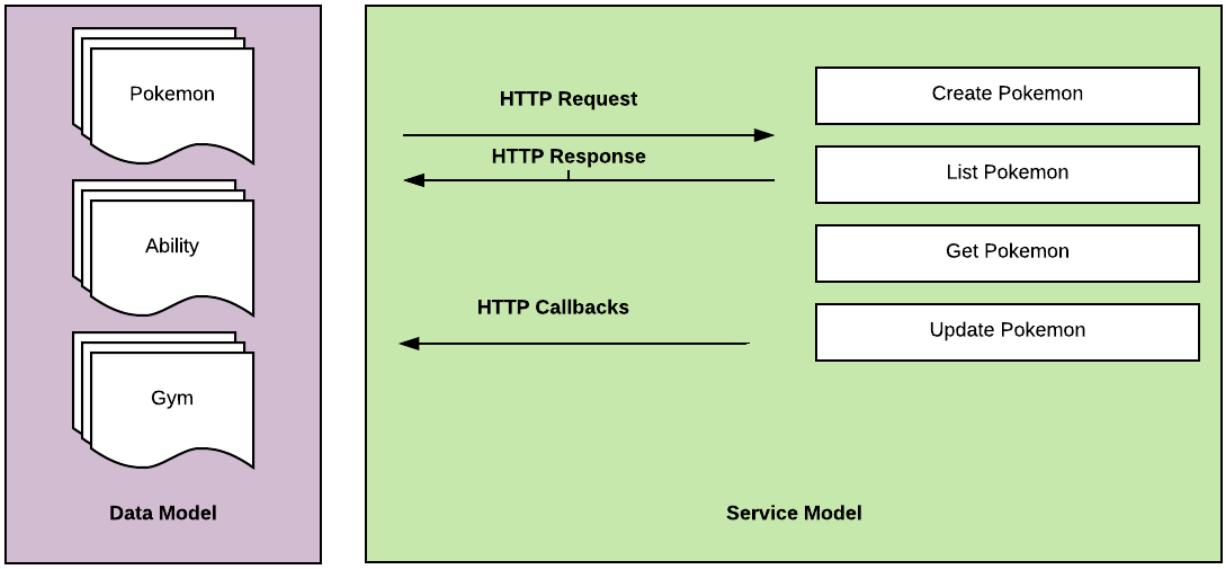
> The Web Services Description Language is an XML-based interface definition language that is used for describing the functionality offered by a web service. The acronym is also used for any specific WSDL description of a web service, which provides a machine-readable description of how the service can be called, what parameters it expects, and what data structures it returns. Therefore, its purpose is roughly similar to that of a type signature in a programming language.
>
> — Wikipedia

WSDLs were only used for SOAP, and not many paradigms or implementations seemed to bother with this sort of description language for a long time. Luckily that has all changed over the last few years.

Before we look at how various description languages are used today, lets learn another bit of theory.

2.2. Service Model & Data Model

Two terms that pop up from time to time are "service model" and "data model". These are two ways to specify which part of the API contract you are talking about.



The service model is the language used to describe things like the URL, HTTP method, headers like content type, authentication strategy, etc. It is used to explain all the things outside of the message, the stuff peripheral to the actual data. In a programming language this would be class names, method names, possible exceptions, but would not cover anything about the arguments: type hints, possible validations, etc.

The data model is used to describe the contents of the message, which is probably what you find in the HTTP body. The word "schema" is often associated with the data model too. They are often just two different terms describing the same concept, but often the term "schema" is used to represent the actual technical file containing the description language, and the term data model is more theoretical.

With this in mind, lets go look at how contracts are written for endpoint-based APIs (REST, RESTish, some RPC), GraphQL and gRPC.

2.3. Introduction to API Description Languages

Any generic HTTP API can use the same description languages, but the modern implementations which the conventions of HTTP to roll their own approach require their own specific description languages.

Here we use the term "HTTP APIs" to group REST, RESTish, and regular RPC when it's just interacting over HTTP (GET, POST, etc) without some other set of conventions like gRPC getting involved.

2.3.1. HTTP APIs: OpenAPI & JSON Schema

In the HTTP API world there were a few such as API Blueprint, RAML, and OpenAPI (at the time called Swagger), but for years the tooling was a bit lacking, and mostly only allowed for outputting as documentation.

OpenAPI v3.0 popped in 2015 up which solved a lot of problems with OpenAPI v2.0, and beat the

heck out of the other description formats. It took a few years for tooling to catch up, but by 2018 pretty much everything supported OpenAPI v3.0, and this description format settled as the mainstream favourite.

```
The OpenAPI Specification (OAS) defines a standard, programming
language-agnostic interface description for REST APIs, which allows both humans
and computers to discover and understand the capabilities of a service without
requiring access to source code, additional documentation, or inspection of
network traffic.
```

An overly simplified example of OpenAPI describing an API which lists collections and resources of hats.

```yaml
openapi: 3.0.2
info:
  title: Cat on the Hat API
  version: 1.0.0
  description: The API for selling hats with pictures of cats.
servers:
  - url: "https://hats.example.com"
    description: Production server
  - url: "https://hats-staging.example.com"
    description: Staging server

paths:
  /hats:
    get:
      description: Returns all hats from the system that the user has access to
      responses:
        '200':
          description: A list of hats.
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/hats'

components:
  schemas:
    hats:
      type: array
      items:
        $ref: "#/components/schemas/hat"

    hat:
      type: object
      properties:
        id:
          type: string
          format: uuid
        name:
          type: string
          enum:
            - bowler
            - top
            - fedora
```

OpenAPI is a YAML or JSON based descriptive language which covers endpoints, headers, requests and responses, allows for examples in different mime types, outlines errors, and even lets developers write in potential values, validation rules, etc.

Another popular language is JSON Schema, which parts of OpenAPI are based on. The two are

mostly compatible, and are both used for slightly different but complimentary things.

OpenAPI can describe both service and data model, and JSON Schema mainly only defines the data model. In the example above, everything under `paths` is describing the service model, then everything under `components.schemas` is describing the data model. The schema keywords that OpenAPI v3.0 uses are based on JSON Schema, and there is a bit of a tangent we should look into here about compatibility.

> OpenAPI v3.0 schema objects are a subset/superset/sideset implementation of *JSON Schema draft 05*. Most JSON Schema keywords are available and work as expected, a few extra OpenAPI-only keywords were added, but some JSON Schema keywords are not supported. There is also the tricky situation where JSON Schema has continued to progress quickly since draft 5 (draft 8 is almost complete at time of writing).
>
> This can cause confusion for new developers, but interoperability amongst standards is always a tricky one. Thankfully, future versions of OpenAPI (probably v3.1) aim to solve this, so no need to get too stuck in the weeds here. For those who want to learn more, this first article fully explains the situation, and this second article explains workarounds and longer term solutions.

OpenAPI contract files are usually static. They're usually written down along with the source code, then sometimes deployed to a file hosting service like S3 for folks to use. Some managers want to treat these like business secrets and hide them under lock and key, which makes absolutely no damned sense as they are meta-data only. Most "hackers" could probably figure out that you keep your list of companies under `GET /companies`, so just don't make that a publicly available endpoint and you're gonna be ok. PayPal, Microsoft, and other companies make their OpenAPI contracts available to anyone who wants to download them, and this approach can help folks integrate with your APIs.

You can imagine an OpenAPI file growing to be rather unwieldy once its got 50+ endpoints and more complex examples, but have no fear you can spread things around in multiple files to make it a lot more DRY (Don't Repeat Yourself) and useful. The first thing to go is usually the `components.schemas` definitions, which can be moved to their own files. Once these are split into their own files, an extra step can be taken to turn them into proper JSON Schema files. Once they are split out they can be referenced in a HTTP response header.

```
Link: <http://example.com/schemas/hat.json#>; rel="describedby"
```

When a client sees this they can use it for all sorts of things - like form generation and client-side validation - all without needing to figure out how to distribute the files to them ahead of time.

One more note on OpenAPI and its old name Swagger. You still see the word Swagger floating around a lot. SmartBear, the original authors of the "Swagger" API description language, still use the word Swagger in a lot of their tooling because they have the brand recognition. The description language itself was renamed to OpenAPI and handed over to the OpenAPI Initiative.

Since 2015, anyone calling it Swagger is out of date, and the fact that folks keep using the word Swagger in 2019 is still a huge source of confusion. If you look for "Swagger tools" you will only find those from SmartBear, or really really out of date ones. Call it OpenAPI, search for OpenAPI, and we don't need to keep saying OpenAPI/Swagger like they yare two alternative but equally valid things.

OpenAPI and JSON Schema are a fantastic pair, and we will show how to combine the two throughout the book.

GraphQL Schemas

GraphQL as an implementation comes bundled with GraphQL Schemas. GraphQL does not really have a service model, as it does not need one.

Seeing as most interactions operate under a single HTTP endpoint like `POST /graphql`, there is no real need to bother writing a contract around that in great detail. It would just be mentioned in passing as an implementation detail, and the majority of the effort would go into describing the data model.

NOTE	Some folks might have different endpoints for different use-cases, but this is rarely spotted in the wild.

All the GraphQL documentation examples are Star Wars. Sure, it's obviously inferior to Stargate SG-1, but let's reuse their examples for simplicity:

An example of GraphQL schemas in the GraphQL Schema Language, implementing interfaces and sharing properties across different types.

```
interface Character {
  id: ID!
  name: String!
  friends: [Character]
  appearsIn: [Episode]!
}

type Human implements Character {
  id: ID!
  name: String!
  friends: [Character]
  appearsIn: [Episode]!
  starships: [Starship]
  totalCredits: Int
}

type Droid implements Character {
  id: ID!
  name: String!
  friends: [Character]
  appearsIn: [Episode]!
  primaryFunction: String
}
```

The syntax in this example is using the GraphQL Schema Language (a.k.a IDL), but these files can be written in whatever programming language the API is built in: JavaScript, PHP, Go, whatever.

Writing them in Go for example would make them a little tricky to interact with for other languages, like trying to give these types to a JavaScript web-app. If the client really needs them, a lot of the language-specific implementations offer a way to "Dump" them to the IDL, which can then be read by the client with the right tooling.

If dumping and distributing isn't a viable workflow, introspection can be used! This is basically the process of quering the GraphQL API for information about the schema, just like how `Link` is used to provide the client with the JSON Schema in other HTTP APIs.

GraphQL schema does not support validation rules defined in the contract - beyond required/optional/null like OpenAPI and JSON Schema, but there are some extensions floating around which can help. More on all of that later.

gRPC: Protobuf

gRPC uses another Google tool for its API contract: Protobuf. Protobuf is basically a serializer for data going over the wire. Much like GraphQL and its schemas, Protobuf is integral to gRPC. Instead of schemas they call them "Message Types", but it's all the same sort of idea.

Instead of writing them in whatever language the API is written in (like GraphQL), a new `.proto` file

is written using Protocol Buffer Language Syntax. This C-family/Java style language exists solely for writing these files. It might be a bit of a pain to figure out a brand new syntax, but it has the benefit of being fairly portable as you can read them in multiple languages. Finding a JavaScript, Ruby, Python, Go, etc. tool that can read a `.proto` file is a whole lot easier than trying to get Python to read something written in - for example - JavaScript.

Rarely are `.proto` files made available over a URL, they are usually bundled and distributed with client code. Then usually things are kept backwards compatible until the clients have upgraded whatever client code brought the `.proto` files their way.

2.4. When Are Contracts Written, And By Who?

At what stage these contracts are written, and by who, is very much up to the culture of the organization. In some organizations the culture is "We dont bother at all" and I've been there. I spent two years helping teams fix the issues that came up from being vague about this stuff, and witnessed a lot of my friends and colleagues waste time (and the companies time) guessing contracts.

When I first got to that company, the culture relating to API contracts was:

> That's the thing that Phil keeps going on about, I think? Just ignore him. We've got unstable, untested, undocumented APIs full of problems to try and sort out, and they're being misused by clients. No time for any of that contract-first planning, or contract-later nonsense.

Two years later and that culture had changed substantially, to the point where most older APIs had contracts written down, and new ones invariably had contracts written before the work was started. Don't make me come to your office and shout at you for two years, just start planning your APIs properly now.

Who should create and maintain contracts? Everyone.

When they should be created? As early as possible.

If one person is tasked with developing an API, then that is the one person who should be writing the contracts as a planning stage, before they start working on the actual API code.

If a whole team is tasked with developing an API, then that team should split up the planning work between them.

The planning process involved getting out a whiteboard, getting a few of your clients in a room so you can listen to their needs (instead of just dictating to them), get somebody who knows a bit more about systems architecture than the average developer does (everyone thinks they're an expert), and hash out some ideas.

When those ideas start to solidify, start writing things down, and turn those notes into API contracts. When they're ready, get them into a GitHub pull request, or some other collaborative place, and folks on your team can start to review them.

Sometimes you would even see full service and data models written up and attached to the JIRA tasks! [1: JIRA is a 'popular' piece of project management software from Atlassian].

2.4.1. Contract-First vs Contract-Later

Whenever the topic of contract-first API development comes up, somebody will say "Damn, that sounds pretty good, but we already have an API written, and we didn't write down the contract!"

Fear not. Some more strict languages like Go and Java have annotation-based systems which can allow you to sprinkle some syntax around your applications to generate some API contracts. This approach does not work so well for dynamic programming languages like Ruby because anything can be anything and you end up having to write so much in manually that you might as well just be writing proper API contracts.

There are a few tools out there which will help you create contracts by reading your JSON requests and responses.

2.5. Summary

The terms "API Contract" and "API Specifications" often mean the same thing depending on the context, but the term specification is used to describe a lot of other things in this space too. If you are talking about planning, building, or documenting an API and somebody mentions "API specifications", then they are talking about this stuff.

The terms "schema", and "data model", are usually closely related. In the terms of an HTTP API, the data model describes the body of the HTTP message, and the technical document or actual file which provides that description is often referred to as a schema.

Ask a bunch of different people and you will get a lot of difference answers, but these terms will be used throughout the book just so there is one standard way of talking.

Writing down contracts might seem like a lot of work, but these days it should no longer be considered as an optional step. Flinging around arbitrary JSON and hoping people and other applications are all using it properly over time is just reckless, selfish, and actually makes work considerably more mundane. Seeing as API contracts are not just for creating documentation, writing the contract down with a decent description language increases productivity throughout the life-cycle of the API. Then reference documentation appears for free as one of many outputs of the description language using documentation generators.

Later chapters will cover exactly how contracts can get involved at various points, like how we can use the contracts to get feedback, and specific tools we can use to collaborate on this stuff.

This introduction will most likely have left you with questions, and they will be answered throughout the rest of the book.

Part Two: Planning & Design

Chapter 3. Introduction

When I was younger and dumber, the planning stage for APIs was pretty much "think of a list of endpoints then start coding". This is wildly insufficient. As the API Contracts chapter explained, contracts are incredibly important throughout the life-cycle of an API.

The information provided in *Part One: Theory* is going to come in useful for this section, as we go through the actual planning of an API. During this process we will plan an API properly, instead of doing the cowboy nonsense I used to get up to.

The planning and design phase of an API is intended to do two things:

1. Make developers think really hard about the requirements for their API
2. Gather feedback from client application developers
3. Use the feedback to improve the planning before the development phase begins

3.1. Time is Money

Poorly planned APIs waste a lot of time. Developers cost money, so paying extra money for developers to rewriting a bunch of code they wrote just recently due to some oversight is a cost most companies would like to avoid. I've seen APIs go from v1 to v2 in the first few months, because the developers handed over a finished product which might have been ok for one client, but caused some issues for another client.

If that client had been involved earlier on then a solution could have been found to satisfy the needs of both clients, and the v1 could have been used for both. Now we've got two separate versions of the API being used at the same time, with developers maintaining two code paths, and at some point the original client using the original version of the API is going to have to rewrite their code to use the newer version of the API. What a daft mess.

3.2. Gather Feedback Early

If clients had more insight into the planning of the API, they have a much better chance of catching out issues earlier on. We can give them fake API implementations to work with (these are called "Mocks" or "Mock Servers"), and when these are based on API contracts the mocks are generated with minimal effort from API developers.

Making a few tweaks to the API contract files is much easier than rewriting a prototype and all of the tests that go with it, and is infinitely easier than deprecating and removing code that has already made it to production.

This is much closer to the Agile workflow, instead of the Waterfall model. Waterfall is junk for API development.

So, you are sold that API contracts are the way to go, you've picked your paradigm (and possibly an implementation too), and now you want to start writing down what the actual API is going to look like.

You might not want to start typing special YAML/JSON/Protobuf blindly into an empty text file. This has been used as an argument against API contracts for a very long time, but it has never been true. We can use text-based or visual editors, which have special knowledge of these description formats, to speed up the development of our contracts!

Chapter 4. Editors

Editors exist in various forms for various API description languages, and they can drastically ease the creation of contracts when starting from scratch. These can take a few forms.

4.1. Text or GUI

Some editors are GUI-based (Graphical User Interface) and let you design things with clicking buttons and working with forms.

Others are text-based and provide you with a real-time preview panel whilst you type, making it slightly easier than working with plain-old YAML/JSON.

4.2. Prototyping, Walled Gardens, & Local File Editors

Beyond the "text-based" or "GUI" distinction, there are other characteristics which will effect where the contracts live and how you interact with them.

Some editors are intended for rapid prototyping only, helping you to create an initial design and then push you to export the contracts as text to copy-and-paste, or download as files. From that point on, you're on your own, and have to find some other way to keep your contract files up-to-date.

Others are hosted solutions which keep a hold of your contract files, and offer limited options for syncing to whatever pre-determined locations they chose to support. Maybe something specific like GitLab, or generic git repository, or they publish them to a URL accessible over HTTP, or something else.

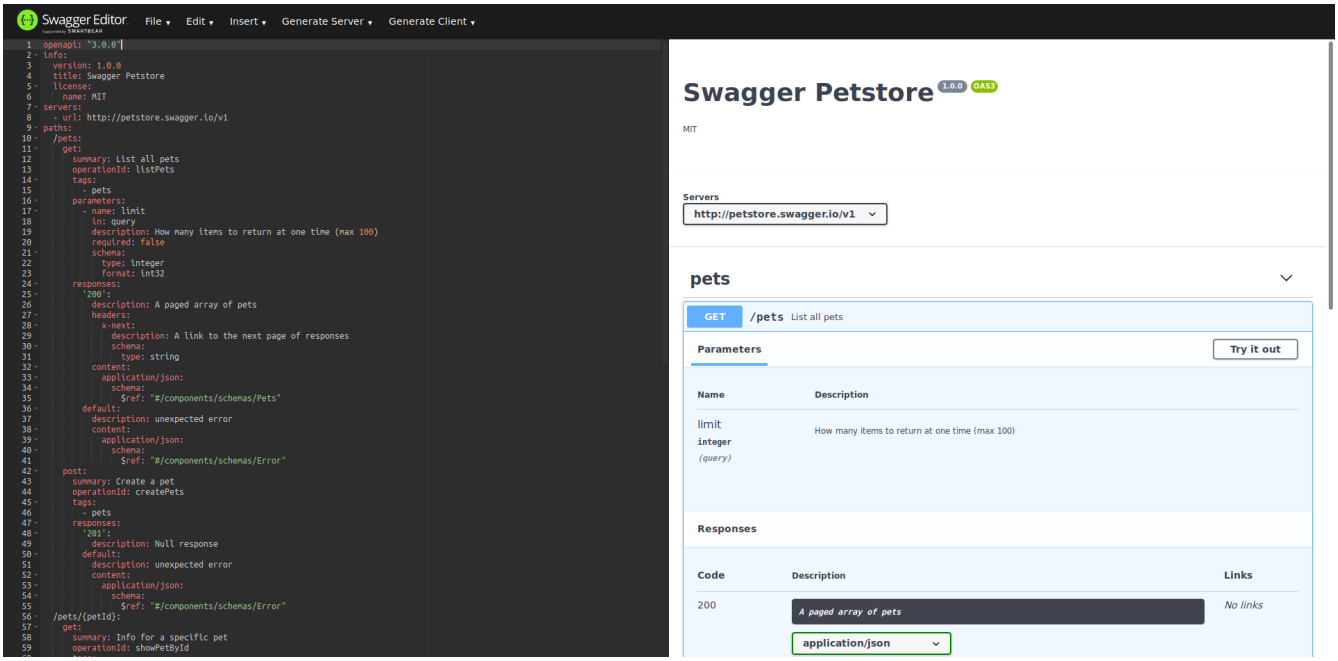
Other editors can take the form of desktop applications, which can be used throughout the life-cycle, loading local files and allowing you to edit them and commit them back to the repository (or wherever they live) at will.

4.3. Standard HTTP Editors

4.3.1. Swagger Editor

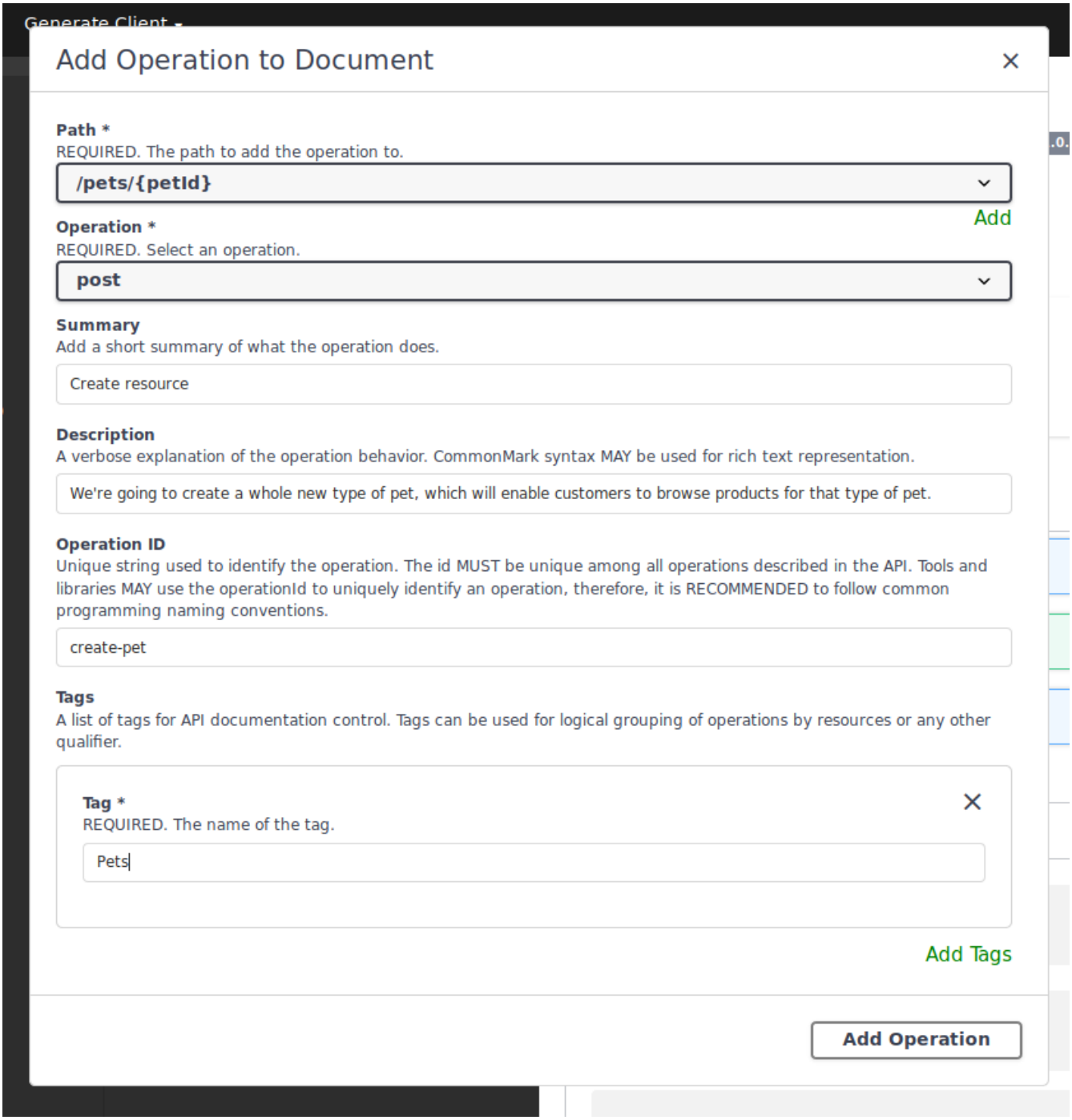
Swagger Editor is a very simple text-based editor from SmartBear. You can find it hosted on swagger.com, and it can be downloaded and run locally for free.

NOTE	Remember, SmartBear call their suite of tools "Swagger" because they own the copyright and they don't feel like re-branding, but the description language is officially called OpenAPI, not Swagger. Nobody else should be using the word Swagger, or calling their tooling Swagger, or anything else.



Mostly it is just a text-editor, with the left panel being a YAML/JSON editor, and the right panel is Swagger UI, the documentation tool from SmartBear.

It can be nice to write specifications on the left and see their rendering as documentation on the right, but it would possibly be more useful if the right contained forms for managing the specification beyond modifying text, but the right panel is purely read-only. There are a few forms for adding new operations, tags, servers, etc. but they do not cover everything.



This simple editor helps in a few cases, like:

- You have never written any OpenAPI and want to try playing around with real-time feedback
- You are getting validation errors from some local tool which seem weird, so you bundle the multiple files into one file and paste it up here to see what it says

There is not much else you will want to use Swagger Editor for. If you are splitting files up using `$ref` (which you pretty much always want to do), then this will not work as it only supports the one text box.

For ongoing editing and multi-file support you will need something a bit more powerful.

4.3.2. SwaggerHub

SwaggerHub is another tool from Smartbear, and it is basically the hosted version of Swagger Editor which has some integrations with the rest of the Swagger tool-suite.

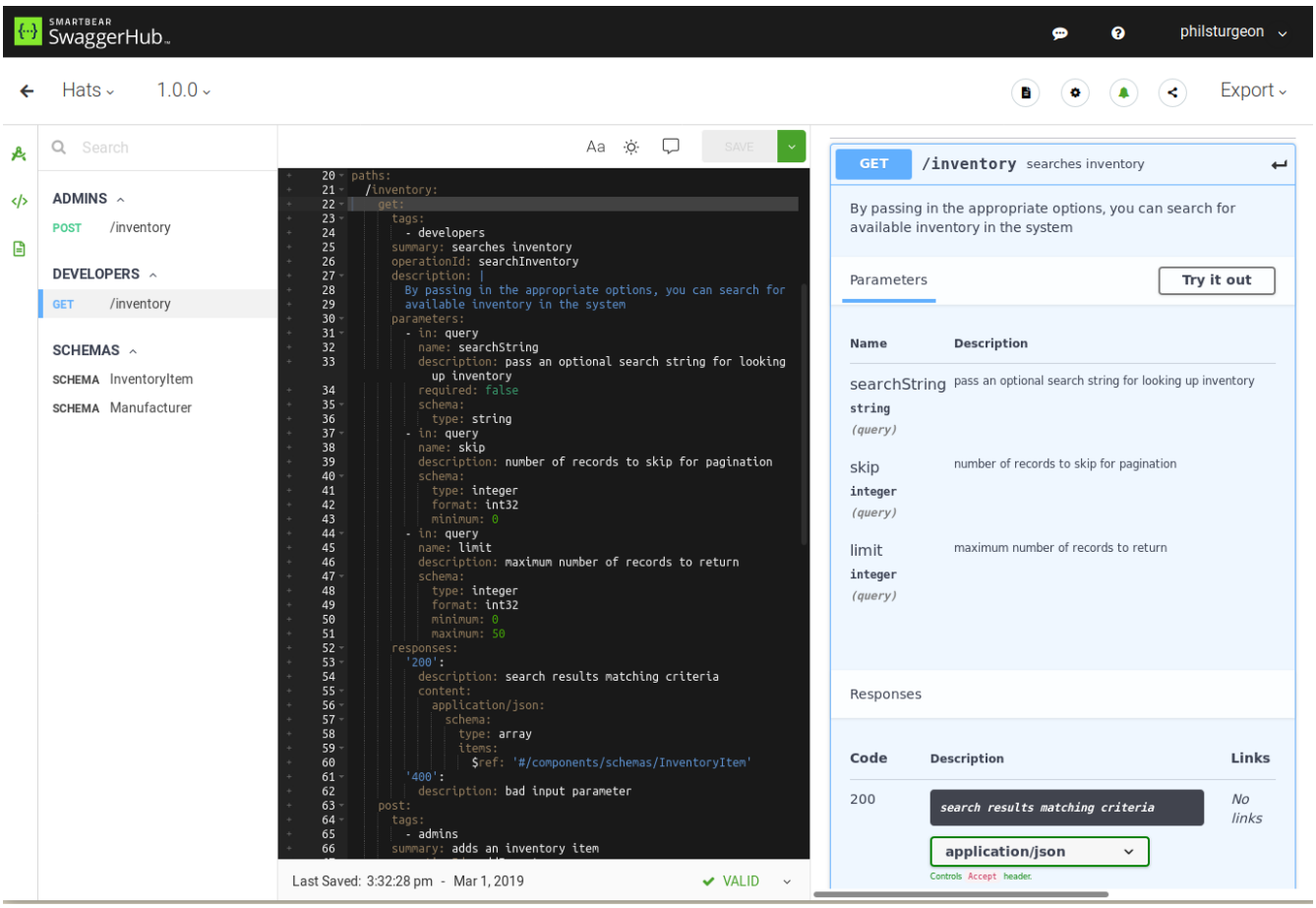


Figure 4. In this demo of SwaggerHub you can see Swagger Editor in full swing, with a few other options floating around the outside.

SwaggerHub includes a mock server, as most of these editors do. It will help you converting from OpenAPI v2.0 to v3.0 which is a nice touch. The integration with Swagger Inspector - a lovely tool which can help build OpenAPI specifications off of HTTP interactions… more on that later.

The collaboration options here are pretty handy. You can invite users or teams to collaborate on an API, and they have a permissions system which seems pretty closely modelled on Google Docs.

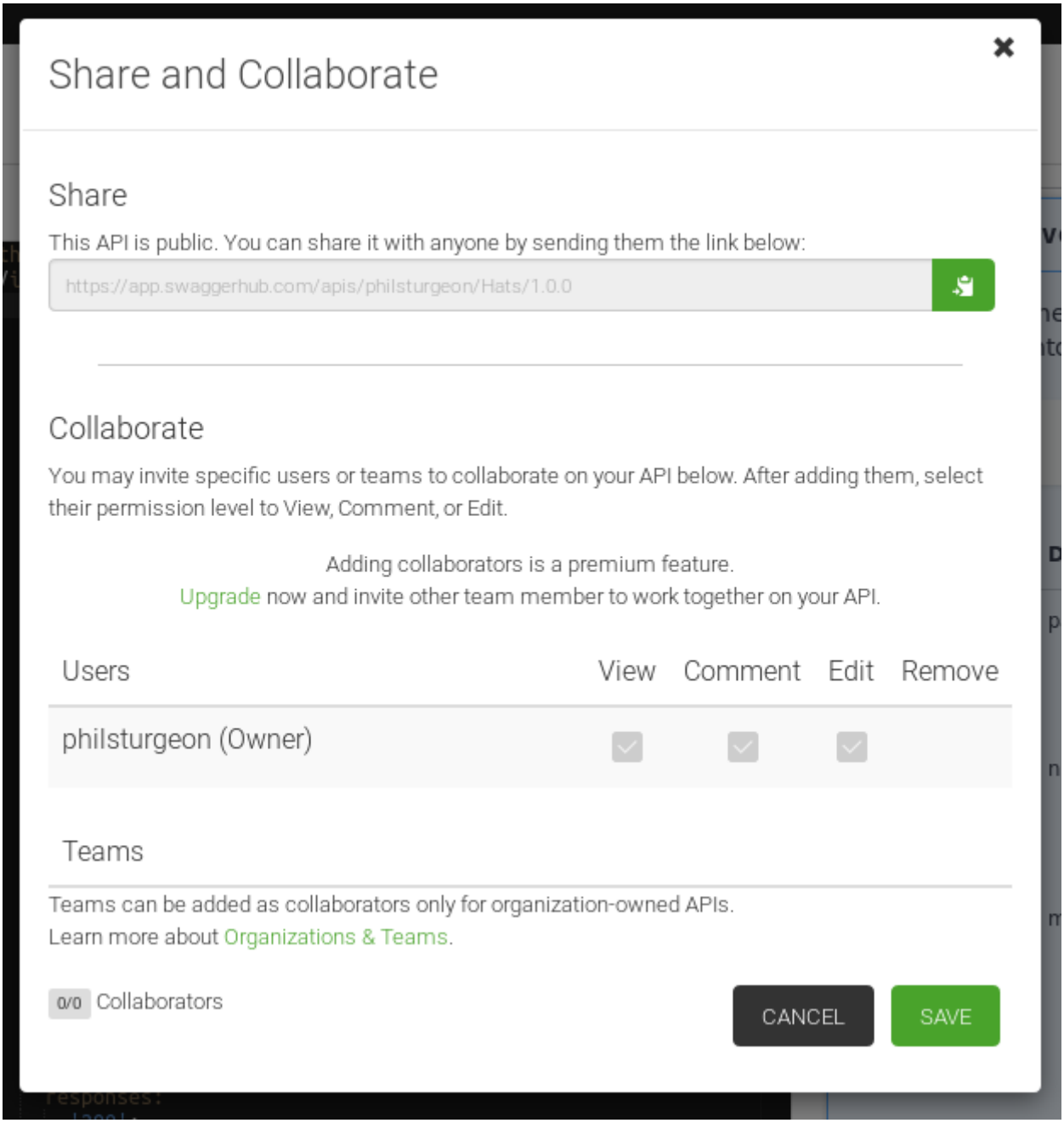


Figure 5. Get collaborating with other folks without having to email YAML files around!

If working with multiple files is possible in SwaggerHub, I have certainly never figured it out. For this reason I have steered clear, but for teams just starting out it might be worth playing with.

The pricing is user-based, and does not seem to support "monthly active users" like slack. Many companies only have 1 or 2 folks who care about API contracts (which is sad and slowly improving over time) but at a company with ~100 engineers (and a better culture), this could get rather expensive. Trying to convince your boss that a YAML text-editor with collaboration might be a tough sell, especially when there are more powerful alternatives, or free open-source offerings.

4.3.3. Stoplight

Stoplight is a GUI editor, with forms and a much more "Wizard"-like approach. You can ignore the YAML representation entirely and work with their forms and model editors.

Right now they only support OpenAPI v2.0 via the GUI but the text mode supports OpenAPI v3.0.

They have all sorts of amazing features. Stoplight has the usual collaboration and mocking, and other features like linting based on style guides, the ability to share models throughout the organization (to avoid having 20 different "user" or "location" representations) and has desktop applications for Windows, Mac and Linux.

I started working for Stoplight right around the time I started writing this second book, so to avoid complex bias I will share some words from Kin "API Evangelist" Lane:

> I feel like Stoplight has the potential to shift the landscape pretty significantly, something I haven't seen any API service provider do in a while.
>
> — Kin Lane, API Evangelist

More to come on this one. They will most likely have a new suite of tools out before this book goes to the printer.

Stoplight costs more than SwaggerHub, but is likely to be an easier sell due to the fact it does a whole lot more. Non-technical users can get in there to play around, and it adds a lot more value beyond being a YAML/JSON text-editor.

4.3.4. Free/Open-Source Tools

Bootstrapping a project and have no budget for fancy tooling? Not a problem.

Check out OpenAPI.tools for an up-to-date list, but there are a few GUI editors, and a few plugins for popular software like Visual Studio Code, Atom, etc. which can ease the struggle of writing the files by hand.

- VS Code: openapi-lint
- Atom: linter-swagger
- Jetbrains IDE: Senya

Then there is ApiBldr, a free hosted GUI editor which can help build and then download contracts.

4.4. GraphQL Editors

GraphQL Editor is a GUI editor, with text on the left where you define various types, and a visual representation of everything as nodes on the right. You can click on those nodes, modify properties, and set criteria like required, null, etc.

It can be installed locally via npm, then invoked as a React component.

If you don't know what any of those things mean then fair enough, there is a hosted version: GraphQL Editor Cloud.

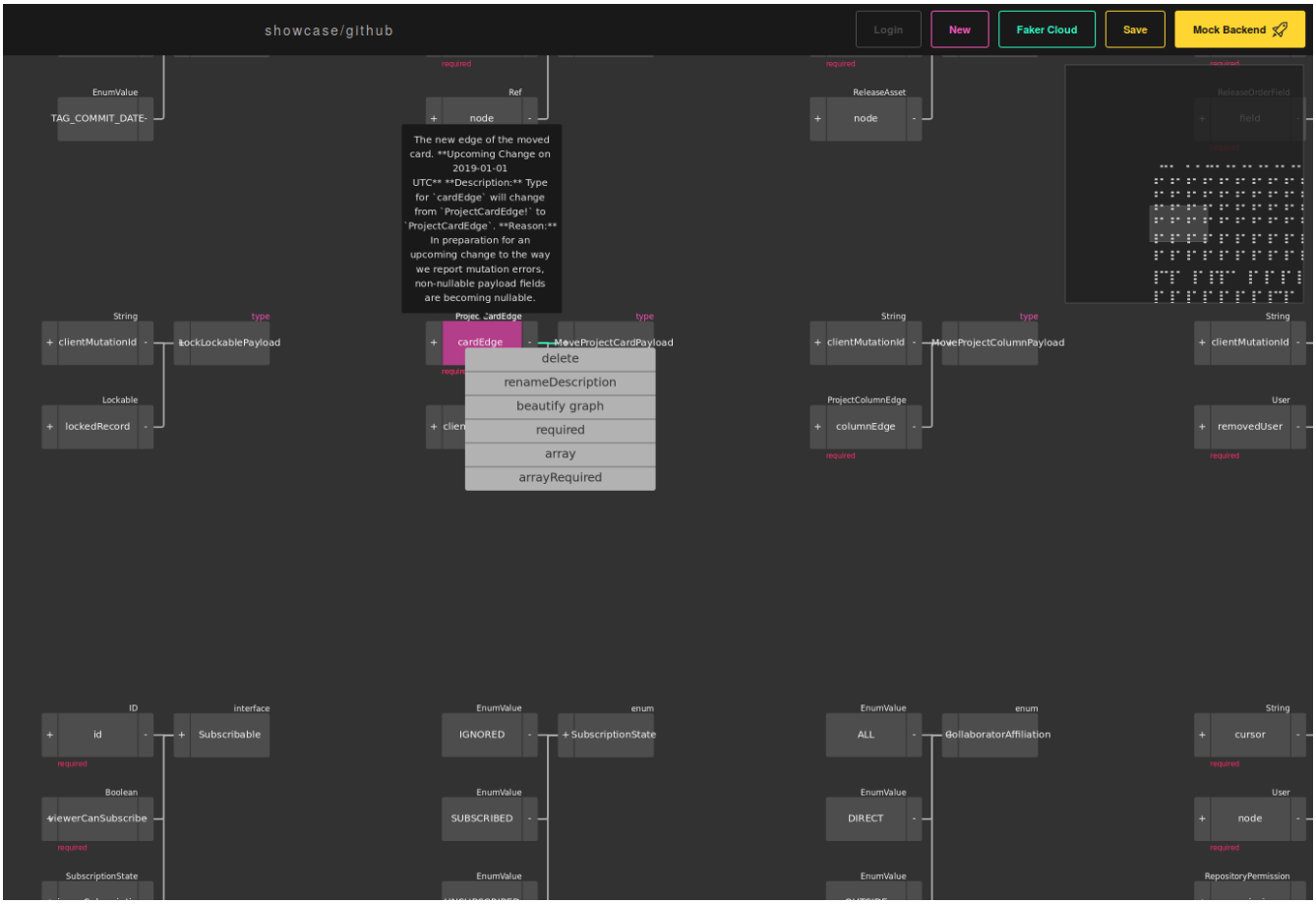


Figure 6. A screenshot of the "Github example" on graphqleditor.com

As with many editors it comes with an option to provide a mock server for the schemas you've just written up. The hosted version also has the ability to save projects.

4.5. Protobuf Editors

The Protobuf syntax is incredibly simple, and there is not much going on in the way of functionality or logic in these files. Maybe this is why there are not many GUI editors around, as you just don't need them. Protobuf Editor is one I found lurking on SourceForge.

There are plenty of plugins ready to add syntax highlighting, linting, auto-complete, etc. to your IDE or code editor of choice.

- VS Code: vscode-proto3
- Atom: language-protobuf
- Jetbrains IDE: Protbuf Support

4.6. Maybe You Don't Use an Editor

This can be a personal choice. Some folks love having their contracts live up in the cloud so they can be easily collaborated on, some want to keep them in the repository so they can discuss things in GitHub pull-requests for their collaboration, and as such want editors which can work with local files.

Whatever you do, when you start out I recommend using an editor to get the ball rolling. Then later on if you want to tweak things by hand, that's probably ok. It will be a while before you work out your exact workflow for contracts (if nobody has worked that out for you), so it can make sense to just get stuck in with whatever hosted editor, then you can probably export things later and cancel your account if you hate it.

4.7. Next

Either way, if you are currently working on an API now, put this book down for a little bit, and get to work on creating some API contracts. It could be nonsense, or it could be a real project, but the next chapter will start by assuming you have written up a bunch of API contracts.

Consult the documentation, tutorials, videos, etc. for the specific API description language in question if there is any confusion that comes up. Hopefully the editors will have your back, but if they don't, there is always Google.

Chapter 5. Mocking

Mocking in relation to APIs is a really simple idea, and that is to create a fake API before building the real one. The basic concepts were mentioned in the introduction, but primarily this is about getting feedback early, allowing you to tweak and change your way to a solution which is hopefully well suited for your clients. Folks scoff at this and say "You can never get things perfect", but you can absolutely cut out a lot back and forth and dodge a few daft mistakes by putting in some time first.

The term "mock" for a lot of developers will have unit-testing connotations. In unit-testing, a mock is a fake implementation of a class or function, which accepts the same arguments as the real thing. It might return something pretty similar to expected output, and different test cases might even modify those returns to see how the code under test works.

This is almost exactly the concept here, just at a HTTP level instead. This is done using a "mock server", which will respond to the expected endpoints, error for non-existent endpoints, often even provide realistic validation errors if a client sends it a nonsense request.

5.1. The Most Basic Mock Server EVER

There are a lot of ways to create a mock server, but the most simple mock server around is JSON Server. This thing is very handy for quickly getting a HTTP server spitting out some JSON that you control:

Woah what a surprise, another NPM tool!

```
$ npm install -g json-server
```

Then you create a `db.json` file with some data in there:

```
{
  "posts": [
    { "id": 1, "title": "json-server", "author": "typicode" }
  ],
  "comments": [
    { "id": 1, "body": "some comment", "postId": 1 }
  ],
  "profile": { "name": "typicode" }
}
```

Now, get that server started via the CLI.

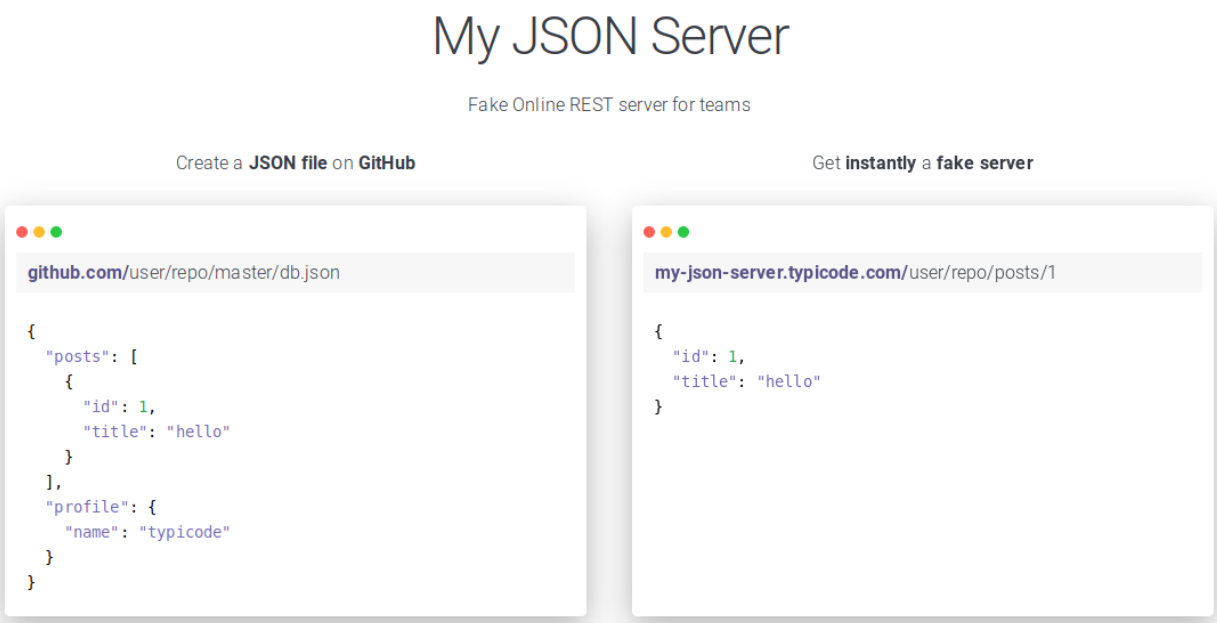
```
$ json-server --watch db.json
```

When browsing to `http://localhost:3000/posts/1` you will see the following JSON:

```
{ "id": 1, "title": "json-server", "author": "typicode" }
```

Boom. A mock server.

You can host that server somewhere simple (shove it on Heroku, Amazon EC2, etc.), or you can use My JSON Server - the hosted version from the same author.



Now, at this point you are probably thinking: "That's lovely and all, but maintaining that one JSON file is going to be a freaking nightmare!"

Yup. Adding a single property to a resource means you are going to have to go through each record in `db.json`. Beyond that, formatting the file can turn into a pain in the ass, conflicts will be rife as the file grows and different developers add different properties, etc. It's good for quick and dirty demos, but is probably not something you will want to use in anger.

How about instead of manually maintaining `db.json`, a tool could be pointed at API contracts, and a mock server was automagically created?! Then it could figure out its own examples based on the examples/default values supplied in the contract, and if the description language has validation rules it could even apply those too!

Thankfully loads of folks in the API ecosystem thought about this, and there is a wide selection of tooling.

5.2. Hosted Editors Love Mock Servers

Most of the hosted editors covered so far have mock servers built in, and often it is either a case of enabling it, or it is already enabled and you just need to find the URL to the server instance.

5.3. Standard HTTP Mocking

5.3.1. Hosted Mock Servers

SwaggerHub

Stoplight

5.3.2. Free/Open-Source Tools

Prism

API Sprout

As always check OpenAPI.Tools for the latest offering of mocking tools.

5.4. GraphQL Mocking

GraphQL Editor Cloud

Apollo - GraphQL Tools

5.5. gRPC Mocking

gRPC has two mocking tools which require a bit more setup. They seem to be more useful for unit/integration testing than helping at a planning stage. We will look at testing in later sections.

5.6. Interacting with this Mock

Interacting with a mock server should be incredibly similar to how you would interact with any API. That is the point, after-all.

Find the instance of the mock server, then point your HTTP/GraphQL/gRPC client at it.

5.7. Next

Hopefully you have managed to get a mock server running for your API contracts, and can interact with it.

Chapter 6. Reference Documentation

So, you handed out the mock server to the various API client teams, and thought they would get straight to work trying to integrate it. Sadly your inbox is now rammed full of questions from those client developers asking how the heck they use the API.

REST devs: You don't need reference docs, the message is self documenting through the rules of HTTP and HATEOAS explains interactions!

GraphQL devs: You don't need reference docs, you have GraphiQL!

gRPC devs: You don't need reference docs, you have .proto files!

Guess what? If you want that sweet-sweet feedback, you'll need to help your clients implement the API, and the best way to avoid inboxmageddon is to distribute "API Reference Documentation".

This is usually an overview of methods/resources available in your API, and looks a lot like the sort of documentation you'd expect to see for a class or function in any programming language. These should always be accompanied by guides and tutorials at a later stage, but early on you will want to provide at least the reference documentation to folks.

Thankfully you have chosen to write down how it all works with contracts, so this will be easy. All we need to do is turn those contract files (probably `.json`, `.yaml`, `.proto`, .etc) into some human-readable documentation for them to point their faces at. Usually HTML to be thrown on a web server, or questionably PDF files for emailing around.

6.1. OpenAPI

OpenAPI has quite a few options for creating reference documentation from source files.

6.1.1. Swagger UI

This tool is hard to recommend due to it's very dated appearance.

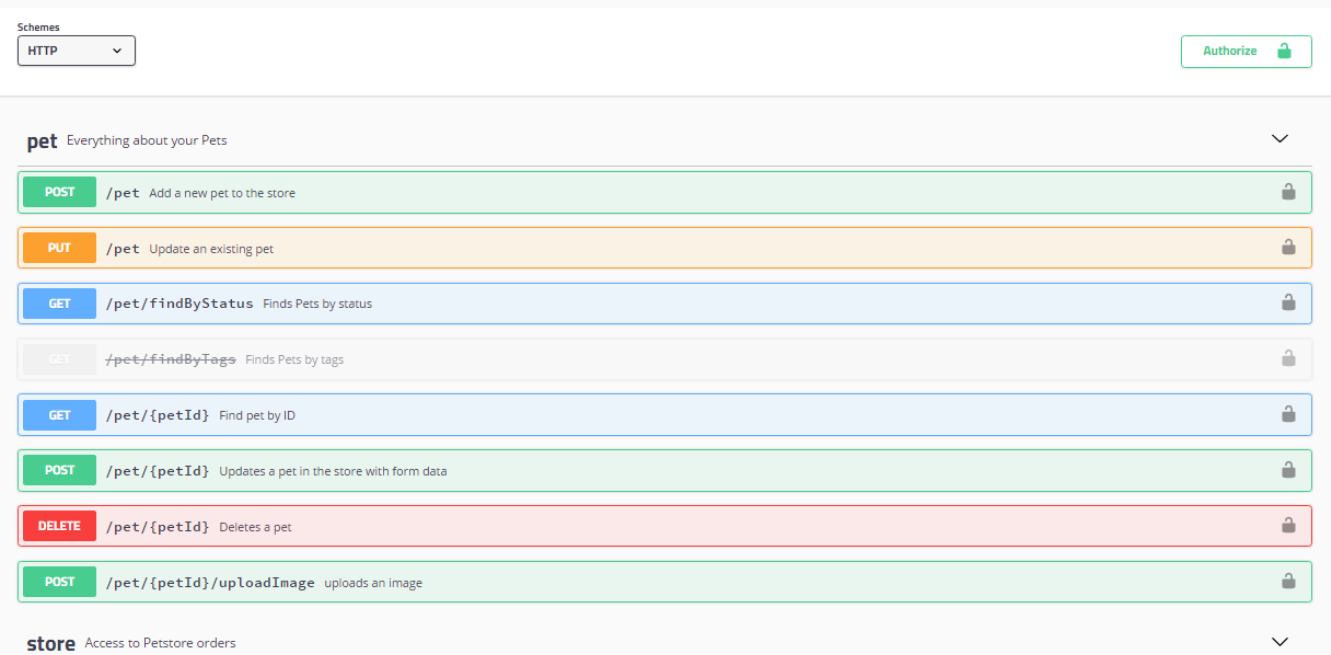


Figure 7. A screenshot of Swagger UI, which looks like it was designed by not just a developer, but one who was a big fan of RPC.

Swagger UI used to be the only option, and it was a major contributor to me completely ignoring Swagger for years. It not only looks pretty rough, but it hides the data model under a few unclear options, meaning the most important data can hardly be seen.

That said, it has an "API console" built in. More on those later.

6.1.2. ReDoc

By far my favourite of the lot, ReDoc looks absolutely stunning. You can add a logo with the `x-logo` vendor extension, tweak colors, and you get that awesome three-column style popularized by docs like the Stripe API Documentation.

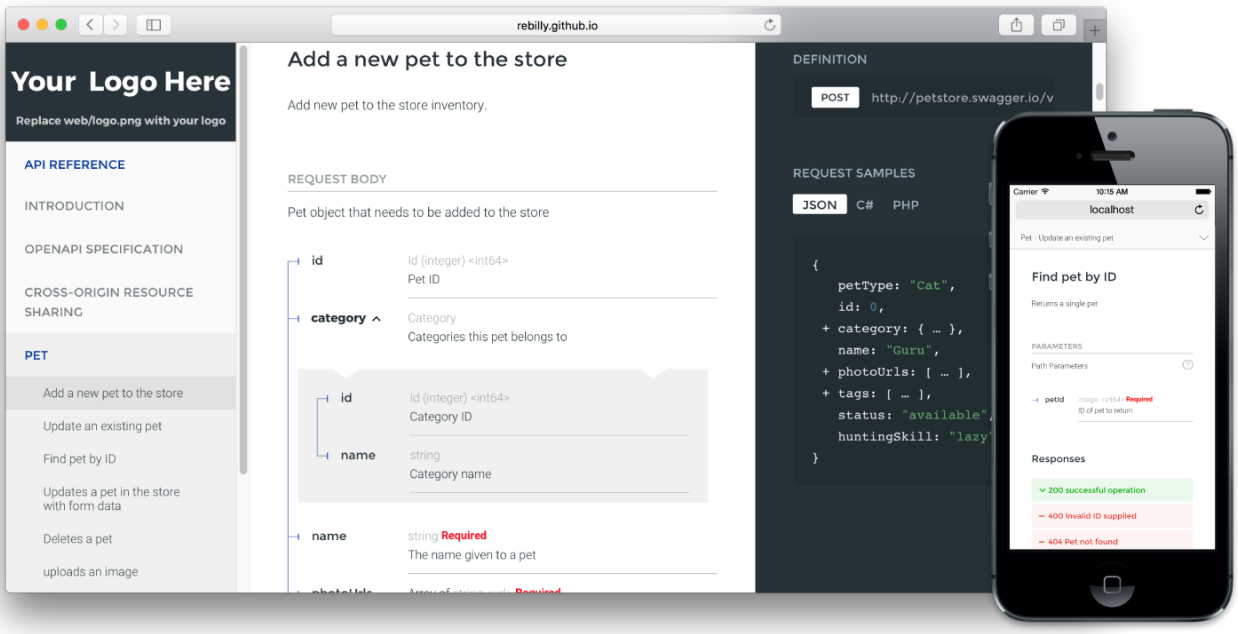


Figure 8. Fancy/modern looking three-col API documentation using ReDoc.

Beyond the looks, ReDoc focuses a lot of elevating the data model (a.k.a schema) to the documentation viewer. It outlines data types, possible enum values, and all sorts of other handy stuff.

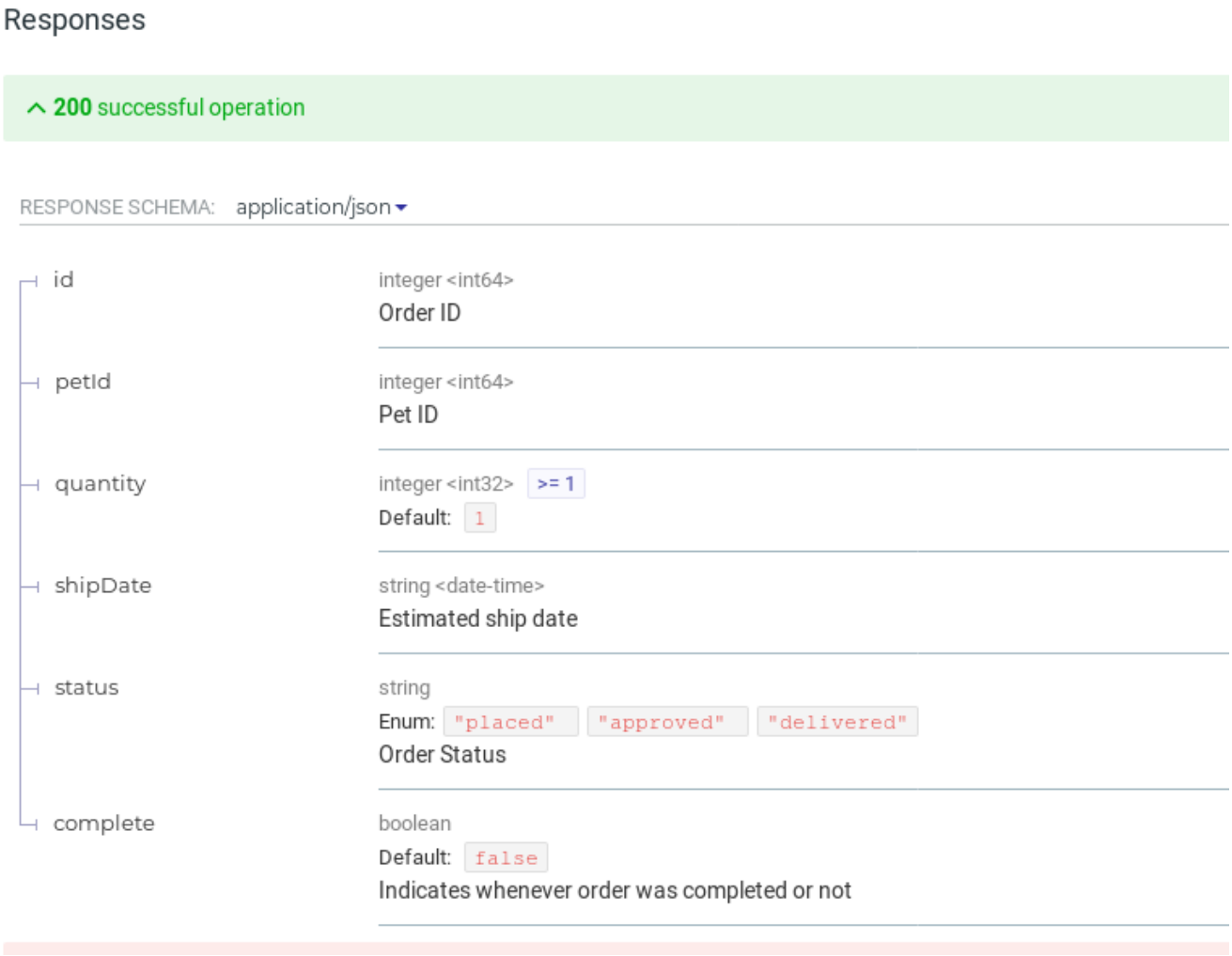


Figure 9. Down arrows let you expand nested objects to avoid initial clutter.

ReDoc can be used as a React command or embedded in HTML, but the most common approach to use it is via the command line. Look at their documentation for instructions, but one way is to install as a node module.

```
$ npm install -g redoc-cli
```

To see how the documentation looks locally, you can run a local web server using the ReDoc CLI tool.

```
$ redoc-cli serve openapi.yaml --watch
TODO update example when ubuntu NodeJS stops cocking about
```

This will run a local HTTP server, and you can load up in your local browser. The `--watch` switch means changes to the local files will be automatically detected, and should show up when you refresh the browser.

When somebody else asks for a look, you can ask the documentation generator to create HTML for sharing around.

```
$ redoc-cli bundle openapi.yaml
```

This will create a single HTML file with embedded CSS and JavaScript, which means there are no dependencies other than the one file it creates. You can chuck this up on Amazon S3 or wherever you like to host your static files, then people can take a look and give you feedback.

ReDoc will also have an API console at some point, but it is not complete at time of writing. This will turn a reference documentation tool into a fully fledged, API client, letting your users trial interactions from the browser.

6.2. GraphQL

During the early stages of GraphQL planning and development, you don't really need to create reference documentation in the classic sense. There is an official tool which is essentially an API console with documentation build in.

This tool is GraphiQL, and it is basically an IDE which can run in various environments.

Although you are in the planning stage and might not be sure what language you are going to build this thing in, if you are a big fan of NodeJS and are likely to use ExpressJS, then you are in luck: express-graphql has Graphiql built in. There are some other installation instructions on the repository README.

There are also a bunch of Chrome extensions floating around which you can use instead of running a local instance of GraphiQL, one of which is called ChromeiQL.

Using these tools you can interact with your fake API, and when you start to develop the real one you can interact with that too.

6.3. gRPC

https://github.com/pseudomuto/protoc-gen-doc [protoc-gen-doc] is a documentation generator plugin for the Google Protocol Buffers compiler (protoc). The plugin can generate HTML, JSON, DocBook and Markdown documentation from comments in your `.proto` files.

Part Three: Testing & Monitoring

Part Four: Management

Part Five: Further Theory