Build APIs You Won't Hate 2:

This Time It's Serious

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. APIs, Services, and Microservices

Before we get too stuck into things we should probably get some terminology agreed upon, otherwise we are throwing around these terms interchangeably like the majority of developers, and talking about different things.

1.1. API

The acronym API stands for "Application Programming Interface", and that's what it says on the tin: an interface for programmatically interacting with an application. This is a rather generic term in the world of computer science and programming, as a Linux command-line utility would consider its command names and options to be an API, whilst a Java library would also consider the method names and arguments to be an API.

Essentially it's all the same concept, but this book will be talking about Web APIs. Frustratingly there are a few different meanings of the term Web API too. Some folks will use this term to describe things like various JavaScipt libraries baked into browsers, like the Local Storage API, Audio API, etc.

Not that. This book will be talking about the utilization of network protocols such as HTTP, AMQP, etc., combined with URLs, and chunks of data (often JSON), to make an API which operates over the web.

The goal of a Web API is to provide other applications with access to specific subset of functionality and data that this application owns. Some APIs are public (anyone can get the weather from api.weather.gov), some are private (it's probably tough to get access to JPMorgan Chase's Enterprise Customers API), and some are a mixture.

APIs can provide data for a single-page JavaScript application, handle payment information to avoid clients needing to worry about storing credit card details, post Facebook statuses on a users timeline, or share the same data across a myriad of different devices; Watch that Netflix show on your Xbox, it's all good, APIs have you covered.

APIs are built by all sorts of folks; maybe another company, some large like Google or Facebook, startups, governments, or charity organizations. The API you build might be for another company, another department within your company, or a backdoor for the US government.

1.2. Service

A "service" is another very overloaded term in computer science, and programming in general. A lot of developers are used to MVC (Model, View, Controller) and a service is just "any code that didn't fit into a model, view, or controller".

When folks talk about services in API-land, they are usually talking about "Service-oriented Architecture".

> Service-oriented architecture (SOA) is a style of software design where services are provided to the other components by application components, through a communication protocol over a network. The basic principles of service-oriented architecture are independent of vendors, products and technologies. A service is a discrete unit of functionality that can be accessed remotely and acted upon and updated independently, such as retrieving a credit card statement online.
>
> A service has four properties according to one of many definitions of SOA:
>
> - It logically represents a business activity with a specified outcome.
> - It is self-contained.
> - It is a black box for its consumers.
> - It may consist of other underlying services.
>
> SOA was first termed Service-Based Architecture in 1998 by a team developing integrated foundational management services and then business process-type services based upon units of work …
>
> — Wikipedia

Service-oriented architecture is essentially the idea of breaking various modules out of your application, possibly putting them on different servers, with a network interface or socket facilitating communication.

For example, instead of having one application that handles user directories, billing, messaging, shipping, inventories, etc., you have multiple applications which handle one of those things independently.

When talking about services you will often hear the term "monolith" pop up. Some people use monolith to describe a mega application that includes the frontend and the backend, including all the presentation logic, business logic, etc. Others just mean that there is a giant API, or a few giant APIs…

Essentially the term monolith is pretty subjective and mostly just means an application which does too many things (or the developer calling it a monolith thinks it does too many things). For example a service which was intended to handle "users" functionality like profiles, but ended up handling authentication, permissions, groups, messaging, friends, mail routing, etc. will probably be called a monolith by somebody.

There are a few benefits to SoA over creating one single giant application, so long as your team is qualified in handling the complications that come with it.

One commonly cited benefit is the ability to write services in all sorts of different languages and frameworks, using the network and standardized protocols as glue. Instead of using the same

language for everything, teams can use the right tool for the job, selecting a programming language based on its strengths and weaknesses when applied to the task at hand.

WARNING	One thing to be wary of here is that some developers get a little giddy when told they can "use whatever language you like", finally giving CoolNewLanguage™ a try, even though the are not very experienced with it. Not only will that make it hard to maintain for that developer, it might also make it hard to find other developers who can help maintain it. If management are not careful, a company just getting into SoA can have 20 services in 20 different languages, and this increases the danger of "Bus Factor" more than a New York City bike commute.

Another benefit is reducing the effects of certain compliance restrictions. Financial standards like PCI might impose a lot of ruling around applications that store credit cards, so if the entire application has access to this data directly the whole application will need to be PCI compliant. Splitting out the billing logic to its own service potentially means that only the billing service has to worry about being compliant.

Other compliance standards like SOX can impose strict rules on how deployments happen, multi-step review processes with multiple stakeholders signing off, rigorous access control, etc. I've worked at companies where SOX really slowed down deployments, meaning that even when code is finished and confirmed working, it might take a week or two to get it deployed. Other teams were happily cranking along, deploying multiple times a day, because the infection of SOX had not spread to their services.

The positives are fairly clear, but an often overlooked problem with service-oriented architecture is the complexity of managing multiple services, and not just the complications of handling "more servers". Managing two services is not twice as hard as managing a single application, it is infinitely more complex, and you need people who know how to handle these issues.

With a monolith if the code is local, unless your tests were garbage and the code you are trying to call does not actually exist, there is zero problem with calling it. SoA means going over the network to another server, and that server might have changed its IP or domain name. It could be out of capacity, or have crashed entirely. Maybe the team who manages it deployed some changes and didn't tell you, and the test suite for your API is not aware of their changes either…

The pros usually outweigh the cons so long as there is a good devops culture, or dedicated sysops people, but SoA is not a merry world of unicorns as some would have you believe.

Anyway, these services need to interact with each other, and their clients need to interact with them. The interaction could happen in a bunch of different ways over the network, but these days it is usually over HTTP, AMQP, or some other messaging protocol. The interaction is by definition a form of API.

An API and a service are not different things. A service will always have some form of API, but not all APIs are a service.

Some APIs could have a more descriptive name than service: like function, or database.

Throughout the book we will use the terms service, function and database accordingly. Attempts will be made to disambiguate if it gets confusing.

1.3. Microservice

Microservices as a term was coined around 2011. It is meant to be an off-shoot of SoA, with more emphasis on isolation and autonomy, like the UNIX philosophy of single responsibility. Like any popular term, over time there has been semantic diffusion of the original meaning, and now there is literally meanings, common meanings, etc.

Some folks define microservice by some objective metric, like number of endpoints or methods - which is essentially confusing them with functions. Others consider the number of conceptual resources.

It is understandable, as hearing service and microservice makes you think one is meant to be "smaller", but size in this context is not counted by the surface area of the interface, or even the size of the entire dependency chart.

> In short, the microservice architectural style [1] is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.
>
> — Martin Fowler and James Lewis

Microservices are meant to be autonomous, so avoiding have a shitload of dependencies will certainly help with that autonomy, but not guarantee it. Dependencies come in two flavours, much like a Brexit: hard and soft. Hard dependencies will cause clients to crash and burn if the dependency is not working as expected, and soft dependencies mean code can continue to work in a degraded way.

> If you switch one of the microservices off and anything else breaks, you don’t really have a microservice architecture, you just have a distributed monolith!
>
> — Domain Modeling Made Functional, Scott Wlaschin

I’ve worked in that architecture. The Rooms Booking service goes down, the customer-facing social network crashes, the messaging system goes down, all of a sudden the user application is down and sparks come flying out of terminals throughout the entire building like the Starship Enterprise is under attack. Tools like "service mesh", "service discovery", "circuit breakers", "distributed tracing", and more exist to help with these problems, and we will talk about that.

Without these things, a microservice architecture is likely to be a distributed monolith, which has all of the downsides of a regular monolith and a whole lot more complications added on top thanks to the joys of networking.

As microservices are meant to be a small part of the full picture, it is pretty common to use them internally to a team/department/some sort of context, then have larger APIs act as an "aggregate" for these services. That also will be written about later.

Seeing as microservices are meant to be services done right, there is not much need to talk about them as different things. We will just talk about services, monoliths, and when we tell horror stories of octopus orgy-like intertwined architectures we will talk about "distributed monoliths".

Chapter 2. 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.

2.1. 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.

2.2. 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 thinks 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.

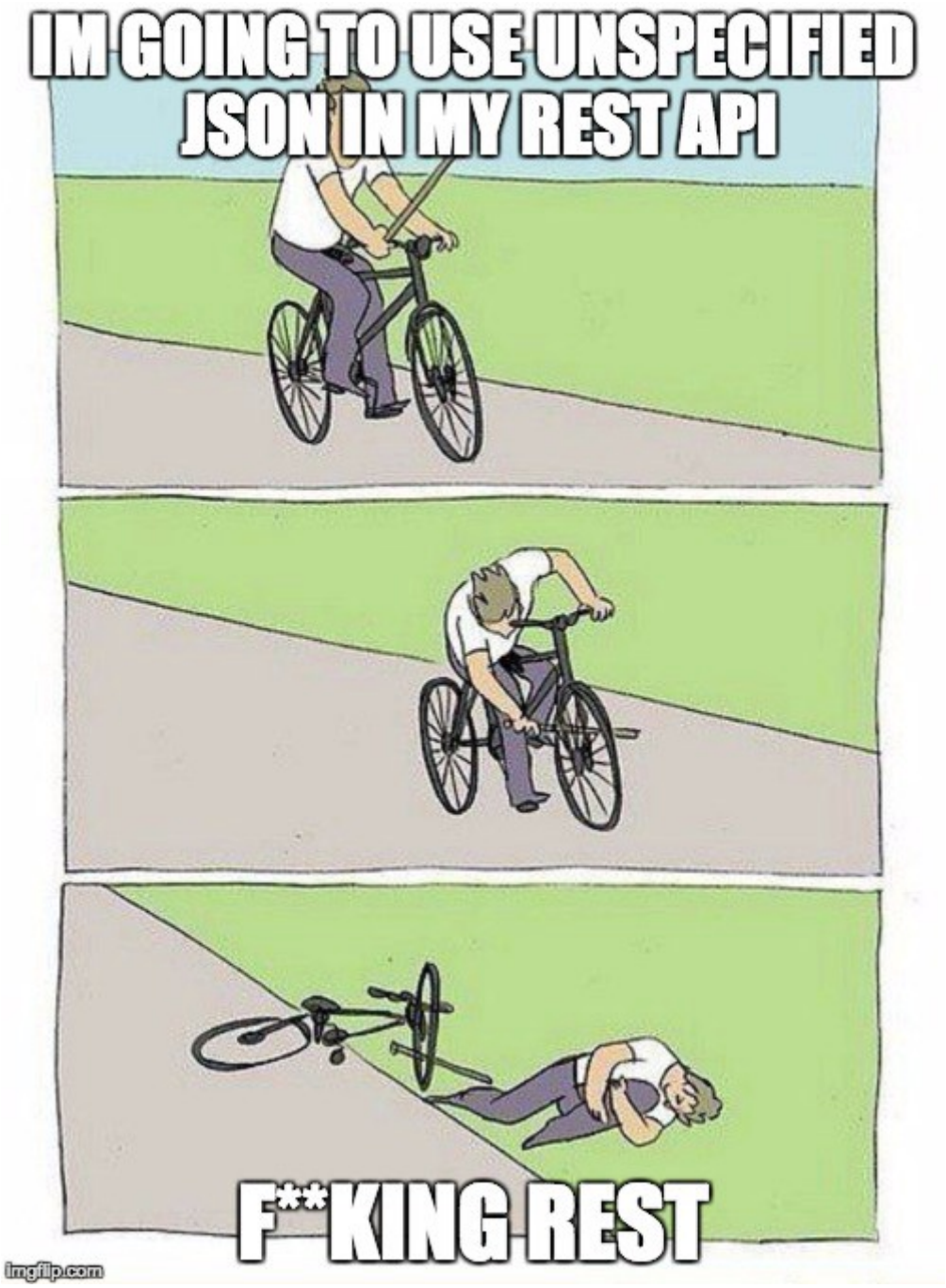


Figure 1. It's pretty common to blame REST for not forcing a type system onto everyone involved in the API, but they're absolutely useable. Don't blame a paradigm for implementation level decisions.

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.

2.3. 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.

2.3.1. 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!

2.4. 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 3. Input and Output

Most of the whole idea of an API is "input" and "output", and in HTTP an input comes in the form of "requests", and output comes in the form of "responses".

There's a "method" involved, and a URL (Uniform Resource Locator) also known as an "endpoint".

3.1. Requests

Making a GET request to the `/places` URL with some query string parameters.

```
GET /places?lat=40.759211&lon=-73.984638 HTTP/1.1
Host: api.example.org
```

Here `GET` is the method, and the part of the URL usually referred to as the "endpoint" is `/places`, and the query string is `lat=40.759211&lon=-73.984638`. The HTTP version in use is HTTP/1.1, and the host name is defined to complete the URL. This is how a request to fetch data from `http://api.example.org/places` looks, and is almost exactly what your browser does when you go to any website.

```
POST /places HTTP/1.1
Host: api.example.org
Content-Type: application/json

{ "some_property" : "some_value" }
```

Here we make a POST request with an "HTTP body". The `Content-Type` header points out we are sending JSON and the blank line above the JSON separates the "HTTP headers" from the "HTTP body". HTTP really is amazingly simple. This is all you need to do for anything, and you can do all of this with an HTTP client in whatever programming language you feel like using this week:

Using PHP and the Guzzle HTTP library to make an HTTP Request

```
use Guzzle\Http\Client;

$headers = [
    'Content-Type' => 'application/json',
];
$payload = [
    'user_id' => 2
];

// Create a client and provide a base URL
$client = new Client('http://api.example.org');

$req = $client->post('/moments/1/gift', $headers, json_encode($payload));
```

Using Python and the Requests HTTP library to make an HTTP Request

```
import json
import requests

headers = {
    'Content-Type': 'application/json',
}
payload = {
    'user_id': 2
}
req = requests.post(
  'http://api.example.org/moments/1/gift',
  data=json.dumps(payload),
  headers=headers
)
```

It's all the same in any language. Define your headers, define the body in an appropriate format, and send it on its way.

There are a bunch of HTTP methods you might run into:

- GET
- POST
- PUT
- PATCH
- DELETE
- HEAD
- OPTIONS
- TRACE

These methods might look familiar from your favourite web application frameworks routing layer, as pretty much every framework offers HTTP method based routing.

Rails example of using HTTP-based routes.

```
get 'photos', to: 'photos#index'
post 'photos', to: 'photos#create'
get 'photos/:id', to: 'photos#show'
put 'photos/:id', to: 'photos#update_full'
delete 'photos/:id', to: 'photos#index'
```

Express (NodeJS) example of using HTTP-based routes.

```
app.get('/photos', (req, res) => { ... })
app.post('/photos', (req, res) => { ... })
```

That said some frameworks like Rails (Ruby) and Laravel (PHP) heavily market their automagical routing conventions to abstract away from manually declaring these routes. These two frameworks and probably others do have command line utilities for printing out the routes they are generating, which are `$ rails routes` and `$ php artisan route:list` respectively.

3.1.1. HTTP Methods

Over the years a lot of people have tried to suggest that HTTP methods do not matter, and anyone who suggests using the right one for the right job is just doing a "Well Actually…" and should spend their time working on something more useful. If you are not using HTTP to its full extent, that stance might be partially true, but when you start to leverage the full power of HTTP that opinion is totally false.

For example, many tools will know that they can cache a GET request, but know that caching a POST would be spooky and problematic. They will know they can retry a PUT or a DELETE, but better not retry a PATCH. These conventions allow a lot of tools that do not have direct knowledge of each other to work together.

Let's learn a bit more about these HTTP methods.

GET

Just fetch some stuff from the server. This is the best example of an "idempotent" action, and this is a word which pops up a bit in API land.

Idempotent means you can do this thing over and over again, and it happening two or more times won't result in different outcomes. If you GET the thing, but the request fails or times out, and you GET it again, the end result is that you got it. Nothing was deleted, or removed, or changed in any lasting way, so if this thing is got a bunch of times it is the same as being got once.

NOTE	GET requests can technically have a body, but some servers and tools will start acting a bit wonky, meaning people generally avoid creating GETs with bodies.

POST

Often used purely for "creates" but can be used for any non-idempotent action.

Creating something is a good example of a non-idempotent action, because if you send two requests to create a thing, you now have two things. You could also be triggering the sending of an email, paying an invoice, etc. It does not have to be a create, but it will result in that action being executed every time you make a request.

PUT

Often incorrectly associated with being an "edit" action, PUT can actually do a whole lot more than that. PUT is designed to be an idempotent way to send data to a server, where the request contains the entirety of the resource. Whatever it says should go, so if the attempt fails for any reason and the client retries, there will be no negative consequence as it's said the same thing twice.

An example of this would be an image upload. An API might have the ability to upload an image for

a user, which is probably a profile image. A request with `PUT /users/jane/image` and a body of the image contents (or a JSON payload with a URL) could then provide the image. It does not matter if the user already had an image or not, if the request is a success they will have one. If the upload fails that is fine, another request can be made, and it will be overridden.

NOTE	Some folks get a bit concerned about this being a "create or update" action, but their concern comes from a misplaced sense that HTTP verbs correspond to a specific CRUD action. If somebody complains at you about this, politely point them at this section.

PUT is not always appropriate, and can lead to race conditions if not used cautiously.

Race Conditions

Think about a resource represented with JSON, that has two properties: `property1` and `property2`. After getting the initial value of the resource with a GET request, two different HTTP clients make requests (Request A and Request B) to update the value of just one property via a PUT. Both `property1` and `property2` are `false` in the original response of the GET request.

Request A

Updating `property1` to be true.

```
PUT /foos/123

{ "property1": true, "property2": false }
```

Request B

Updating `property2` to be true.

```
PUT /foos/123

{ "property1": false, "property2": true }
```

Both properties started at `false`, and both clients were only trying to update one property, but little do they know they are clobbering the results and essentially reverting the updates from other clients. Instead of ending up with both values being `true`, the API will just hold whatever the most recent request was, which is going to be `"property1": false` and `"property2": true`.

Some folks consider this to be a feature, but others consider it a bug because if they only want to update one property, why do they need to send everything? People in the second camp decide to just send the relevant properties they want to change, which is a flagrant misuse of how PUT is supposed to work and can lead to confusion with tools that expect PUT to contain an entire resource, not just partial changes.

For partial changes, there is another method.

PATCH

Patch is a more recent addition to HTTP, with its RFC being finalized in 2010.

> The existing HTTP PUT method only allows a complete replacement of a document. This proposal adds a new HTTP method, PATCH, to modify an existing HTTP resource.

So if PUT is for when a client has all the answers and wants to give that exactly to the server, PATCH is for when the client only wants to update certain parts of the resource.

Some folks have never heard of the conflict scenario above, and recommend PATCH because it is essentially a performance improvement. Technically they are right: sending less stuff over the wire is quicker than sending more stuff.

How exactly PATCH works can vary depending on which data format you're using. If it's JSON then there are two popular approaches: JSON Patch and JSON Merge Patch.

JSON Merge Patch is what most people will want to use for general APIs, and it is simple to use. From the RFC:

Given the following example JSON document:

```
{
  "title": "Goodbye!",
  "author" : {
    "givenName" : "John",
    "familyName" : "Doe"
  },
  "tags":[ "example", "sample" ],
  "content": "This will be unchanged"
}
```

A user agent wishing to change the value of the "title" member from "Goodbye!" to the value "Hello!", add a new "phoneNumber" member, remove the "familyName" member from the "author" object, and replace the "tags" array so that it doesn't include the word "sample" would send the following request:

```
PATCH /my/resource HTTP/1.1
Host: example.org
Content-Type: application/merge-patch+json
```

```
{
  "title": "Hello!",
  "phoneNumber": "+01-123-456-7890",
  "author": {
    "familyName": null
  },
  "tags": [ "example" ]
}
```

The resulting JSON document would be:

```
{
  "title": "Hello!",
  "author" : {
    "givenName" : "John"
  },
  "tags": [ "example" ],
  "content": "This will be unchanged",
  "phoneNumber": "+01-123-456-7890"
}
```

— Internet Engineering Task Force (IETF), https://tools.ietf.org/html/rfc7396

HEAD

Pretty much exactly the same as GET in every way, but HEAD responses must not contain a body. This is great for checks to see if something exists (by inspecting the status code), and if it does the client does not have to wait for the entire response body to be generated then sent down the wire.

DELETE

Guess what this does?!

DELETE requests *can* contain a body, but generally do not. They are considered idempotent like PUT, because if you are asking to delete something, and you accidentally delete it twice, then the response both times should be "Yes this is deleted".

Some APIs do not implement it that way so a second attempt to delete the same thing will get a 404. That is a bit of a shame as it means clients can get a "You cannot delete this" message when they did in fact delete it… plan accordingly.

3.2. Responses

Much the same as an HTTP request, your HTTP response is going to end up as plain text (unless you're using SSL, but hang on, we aren't there yet).

Example HTTP response containing a JSON body

```
HTTP/1.1 200 OK
Server: nginx
Content-Type: application/json

{
  "user":{
    "id":1,
    "name":"Theron Weissnat",
    "bio":"Occaecati excepturi magni odio distinctio dolores.",
    "picture_url":"https://cdn.example.org/foo.png",
    "created_at":"2013-11-22 16:37:57"
  }
}
```

First you might notice the `200 OK`, which is an HTTP status code that says things worked as expected. No issues here buddy. Then there is the `Content-Type`, which just says the data is JSON.

3.2.1. HTTP Status Codes

A status code is a category of success or failure, with specific codes being provided for a range of situations, that are essentially metadata supplementing the body returned from the API.

Back in the early 2000s when AJAX was first a thing, it was far too common for people to ignore everything other than the body, and return some XML or JSON saying:

```
{ "success": true }
```

These days it's far more common to utilize HTTP properly, and give the response a status code as defined in the RFC have a number from `200` to `599` — with plenty of gaps in between — and each has a message and a definition. Most server-side languages, frameworks, etc., default to `200 OK`.

Status codes are grouped into a few different categories, with the first number being an identifier of the category of thing that happened.

3.2.2. 2XX is all about success

Whatever your application tried to do was successful, up to the point that the response was sent. A `200 OK` means you got your answer, a `201 Created` means the thing was created, and a `202 Accepted` is similar but does not say anything about the actual result, it only indicates that a request was accepted and is being processed asynchronously. It could still go wrong, but at the time of responding it was all looking good so far.

3.2.3. 3XX is all about redirection

These are all about sending the calling application somewhere else for the actual resource. The best known of these are the `303 See Other` and the `301 Moved Permanently`, which are used a lot on the web to redirect a browser to another URL. Usually a redirect will be combined with a `Location` header to point to the new location of the content.

3.2.4. 4XX is all about client errors

Indicate to your clients that they did something wrong. They might have forgotten to send authentication details, provided invalid data, requested a resource that no longer exists, or done something else wrong which needs fixing.

3.2.5. 5XX is all about service errors

With these status codes, the API, or some network component like a load balancer, web server, application server, etc. is indicating that something went wrong on their side. For example, a database connection failed, or another service was down. Typically, a client application can retry the request. The server can even specify when the client should retry, using a `Retry-After` HTTP header.

3.2.6. Common Status Codes

Arguments between developers will continue for the rest of time over the exact appropriate code to use in any given situation, but these are the most important status codes to look out for in an API:

- 200 - Generic everything is OK
- 201 - Created something OK
- 202 - Accepted but is being processed async (for a video means encoding, for an image means resizing, etc.)

- 400 - Bad Request (should really be for invalid syntax, but some folks use for validation)
- 401 - Unauthorized (no current user and there should be)
- 403 - The current user is forbidden from accessing this data
- 404 - That URL is not a valid route, or the item resource does not exist
- 405 - Method Not Allowed (your framework will probably do this for you)
- 409 - Conflict (Maybe somebody else just changed some of this data, or status cannot change from e.g: "published" to "draft")
- 410 - Gone - Data has been deleted, deactivated, suspended, etc.
- 415 - The request had a `Content-Type` which the server does not know how to handle
- 429 - Rate Limited, which means take a breather, sleep a bit, try again
- 500 - Something unexpected happened, and it is the API's fault
- 503 - API is not here right now, please try again later

You might spot others popping up from time to time, so check on http.cats (or iana.org for a more formal list) when you see one that's not familiar.

3.3. Mime Types

HTTP APIs can work with all sorts of data. Whilst SOAP may have been restricted to XML, REST and GraphQL can work with any response types. gRPC is kinda stuck with Protobuf.

An API can support almost unlimited options, but of course building support for every content type ever would be a rather laborious job. There are a few we can cut out early on.

3.3.1. Ditch Form Data

"Form Data" uses the `application/x-www-form-urlencoded` mime type, and mostly only seems to be used by PHP developers. Luckily most other folks ignore this wholeheartedly.

One issue with form data is similar to how XML suffers a lack of obvious data types. For example, to handle a boolean a client has to send `1` or `0`, because sending `property=true` would be a literal true on the server side: `string("true")`.

Form data doesn't really have data types, just awkward strings.

```
POST /example HTTP/1.1
Host: api.example.org
Authorization: Bearer adsjakbsd
Content-Type: application/x-www-form-urlencoded

propertyString=something&propertyTrue=1&propertyFalse=0&propertyEmpty=
```

Data types are important, so let's not just throw them out the window for the sake of "easy access to our data", especially as most web application frameworks have something like `$request→body→foo` to easily get to the foo property.

WARNING	Rails is awful at this. If you have a `?foo=a` query string parameter, and you also send `{ "foo": "b" }` in the HTTP body, then `params[:foo]` will be set to `"b"` as the latter overrides the former. Code that you build, including any generic frameworks/tooling that you release, should avoid conflating query strings and body properties at all costs. They're different things and this nonsense causes confusion, especially when you realize that `params[:action]` means "controller method name" and actually overrides whatever is in `?action=` *without* anything being in the body...

Instead of form data, use a nice JSON object.

```
POST /checkins HTTP/1.1
Host: api.example.org
Authorization: Bearer adsjakbsd
Content-Type: application/json

{
    "checkin": {
        "place_id": 1,
        "message": "This is a bunch of text.",
        "with_friends": [1, 2, 3, 4, 5]
    }
}
```

This is a perfectly valid HTTP body for a checkin. You know what they are saying. You know who the current user is from their auth token. You know who they are with, and some API developers like having it wrapped up in a single `checkin` key for making it clear the client should be sending a checkin object, not some other object.

That same request using form data is a mess.

The alternative to a nice JSON object when using form data.

```
POST /checkins HTTP/1.1
Host: api.example.org
Content-Type: application/x-www-form-urlencoded

checkin[place_id]=1&checkin[message]=This is a bunch of
text.&checkin[with_friends][]=1&checkin[with_friends][]=2&checkin[with_friends][]=3&ch
eckin[with_friends][]=4&checkin[with_friends][]=5
```

This makes me upset *and* angry. Do not do it in your API.

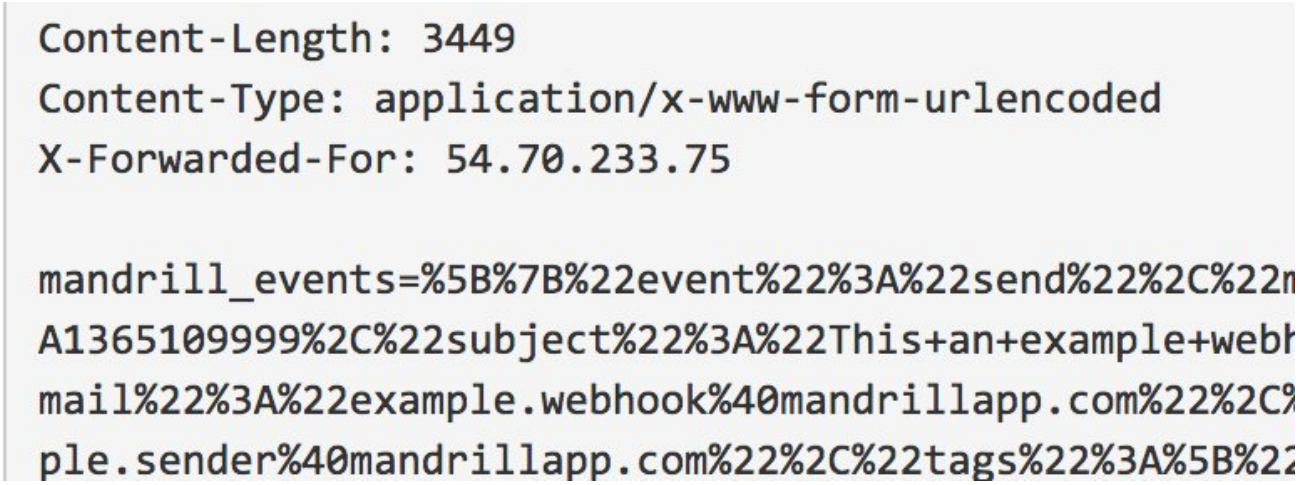


Figure 4. Mandrill API is having a rough time.

Finally, do not try to be clever by mixing JSON with form data:

This is nonsense. Do not do it.

```
POST /checkins HTTP/1.1
Host: api.example.org
Content-Type: application/x-www-form-urlencoded

json="{
    \"checkin\": {
        \"place_id\": 1,
        \"message\": \"This is a bunch of text.\",
        \"with_friends\": [1, 2, 3, 4, 5]
    }
}"
```

This actually happens surprisingly often in the wild.

JSON Data example:

```
{
    "ticket_group": {
        "name": "summer",
        "event_ticket_ids": "{\"28744198672\": [\"56472627\", \"56472628\"]}"
    }
}
```

3.3.2. Why many prefer JSON to XML

Any modern API you interact with will support JSON, or there is some fancy binary format being used. Sometimes APIs will support XML too, especially if the API is maintained by an older financial services company. XML generally takes more memory to convey the same amount of data as JSON, as its similarity to HTML means it requires both a start and an end tag containing the same name.

Beyond purely the size of the data being stored, XML is rather terrible with handling different data

types. That might not worry dynamic language developers all that much, but look at this:

An example of a bunch of different data types in JSON.

```
{
  "place": {
    "id" : 1,
    "name": "This is a bunch of text.",
    "is_true": false,
    "maybe": null,
    "empty_string": ""
  }
}
```

The same example but in XML.

```
<places>
    <place>
        <id>1</id>,
        <name>This is a bunch of text.</name>
        <is_true>0</is_true>
        <maybe />
        <empty_string />
    </place>
</places>
```

Basically, in XML, *everything* is considered a string, meaning integers, booleans, and nulls can be confused. Both `maybe` and `empty_string` have the same value, because there is no way to denote a null value either. Gross.

Work out which content type(s) you actually need, and *stick to that*. The Flickr API used to support `lolcat` as a joke, and that was probably the result of a hack project in which the development team were only paid with cold pizza. JSON is fine.

CSV can be pretty handy as an export format too, especially if your API is offering data for any sort of "Reports".

Whatever you decide to offer, make it very clear in your documentation what formats are supported. HTTP clients can request a specific `Content-Type` be used for the response by placing it in the `Accept` header on the request, and if that type is not available you can return a 406 Not Acceptable response, which is one of many errors an API can return when a request goes wrong.


406
Not Acceptable

Chapter 4. Success or Failure

Dealing with the Happy Path™ in an API is pretty easy: When a client asks for a resource, show them the resource. When they trigger a procedure, let them know if it was triggered ok, and maybe if it completed without a problem.

What to do when something doesn't go according to plan? Well, that can be tricky.

4.1. Errors

HTTP status codes are part of the picture, they can define a category of issue, but they are never going to explain the whole story.

Two examples from a carpooling application which had a "simulated savings" endpoint, to let folks know how much they might save picking up a passenger on their daily commute:

This error let the client know the coordinates were too close together, meaning it is not even worth driving let alone trying to pick anyone else up.

```
HTTP/1.1 400 Bad Request

{
  "errors" : [{
    "code"   : 20002,
    "title"  : "There are no savings for this user."
  }]
}
```

This carpool driver is trying to create a trip from Colombia to Paris.

```
HTTP/1.1 400 Bad Request

{
  "errors" : [{
    "code"   : 20010,
    "title"  : "Invalid geopoints for possible trip."
  }]
}
```

This is often touted as a failing of the HTTP status code concept, but it was never intended to cover every single possible application specific error message. Think of HTTP status codes like an exception. In Ruby you might get a `ArgumentError` or `LoadError` exception which gives you a pretty good hint as to what the issue is, but there is also data specific to the instance of that failure that helps with fixing the situation.

Programming languages do not just give you the exception name, they give you instance information too.

```
> require "nonsense"
LoadError (cannot load such file -- nonsense)
```

Errors in HTTP APIs are pretty similar to exceptions: they can tell the client what is going on, and combine a bunch of useful metadata to help both the client and the server solve problems. This is often in the response body, using JSON or whatever data format the API generally uses.

4.1.1. Error Objects

A well designed API error will have at the very least:

- A 4xx or 5xx status code depending on the situation
- A human readable short summary: `Cannot checkout with an empty shopping cart`
- A human readable message: `It looks like you have tried to check out but there is nothing in your…`
- An application-specific error code relating to the problem: `ERRCARTEMPTY`
- Links to a documentation page or knowledge base where a client or user of the client can figure out what to do next

This will help humans and machines to figure out what is happening. Missing out the error code means clients need to implement substring matching, which is awful for everyone, and turns contents of the error message into part of the agree contract. Imagine a text-change breaking integration with multiple unknown clients!

This used to happen with Facebook and their rather bad Graph API, where any issue with an access token would return type: `OAuthException`, regardless of the type of issue. If it was an expired token which needed a refresh, or if it was just total nonsense, you would get the same type, and a different string.

```
{
  "error": {
    "type": "OAuthException",
    "message": "Session has expired at unix time 1385243766. The current unix time is
1385848532."
  }
}
```

Without getting too much into Authentication at this point, there are times where the client would want to take different actions for different errors. For example, when an access token was previously good but expires, the client wants to suggest the user try logging in again, or reconnecting their Facebook account. When the token is just nonsense (a totally invalid token) then a different action needs to be taken.

These days Facebook have a far more robust error object in their Graph API, with error codes and even "sub-codes", so the client developer has enough information to react programmatically to

various errors.

An improved version of that error message, with an error code and a link

```
{
  "error": {
    "message": "Message describing the error",
    "type": "OAuthException",
    "code": 190,
    "error_subcode": 460,
    "error_user_title": "A title",
    "error_user_msg": "A message",
    "fbtrace_id": "EJplcsCHuLu"
  }
}
```

They explain the structure of the error object in their documentation.

> - **message:** A human-readable description of the error.
> - **code:** An error code. Common values are listed below, along with common recovery tactics.
> - **error_subcode:** Additional information about the error. Common values are listed below.
> - **error_user_msg:** The message to display to the user. The language of the message is based on the locale of the API request.
> - **error_user_title:** The title of the dialog, if shown. The language of the message is based on the locale of the API request.
> - **fbtrace_id:** Internal support identifier. When reporting a bug related to a Graph API call, include the fbtrace_id to help us find log data for debugging.
>
> — Facebook GraphAPI Documentation, https://developers.facebook.com/docs/graph-api/using-graph-api/error-handling

4.1.2. Know Your Audience

Making errors be useful for client users (not just client developers) can be a powerful thing. Clients can build their interface around the expectation that a link in an error will help their users out, without needing to know specifically what the actual error is.

Whenever possible try to avoid creating an API error that you would not want to show to a user. Often a client will create a filter that checks for certain errors to do something, and anything left can be thrown up as a generic error box with the message in it.

Clients doing this help future proof their application. For example, if a new validation rule pops up

they might not have their UI code written to check for that, but an ugly alert box can pop up with instructions to the user and maybe that is better than the application just being completely unusable.

Another useful thing to do is put a link for more information.

Add a href/link/url property to your error object.

```
{
  "error": {
    ...
    "href": "http://example.org/docs/errors/#ERR-01234"
  }
}
```

In some instances maybe this more information link points to a blog post or some documentation which explains that the user should update their application, or take some other action to resolve the situation, or email somebody, or reset their password.

4.1.3. The Trouble with Custom Error Formats

Everyone starts off building APIs with their own error format. It usually starts off as just a string.

```
{
  "error": "A thing went really wrong"
}
```

Then somebody points out it would be nice to have application codes, and new versions of the API (or some different APIs built in the same architecture) start using a slightly modified format.

```
{
  "error": {
    "code": "100110",
    "message": "A thing went really wrong"
  }
}
```

Guess what happens when a client is expecting the first example of a single string, but ends up getting that second example of an object?

Figure 5. A wild [object Object] appears on Gelato - a discontinued API design and analytics platform acquimerged into Kong.

These errors happened at WeWork all the time, because every API had a totally different error format. I remember writing a bunch of code that would check for various properties, if error is a string, if error is an object, if error is an object containing foo, if error is an object containing bar....

4.1.4. Standard Error Formats

There are two common standards out there for API errors which you should consider using for your next API, or maybe even consider adding to your existing APIs.

Problem Details for HTTP APIs

Problem Details for HTTP APIs (RFC 7807) is a brilliant standard from Mark Nottingham, Eric Wilde, and Akamai, released through the IETF.

> This document defines a "problem detail" as a way to carry machine-readable details of errors in a HTTP response to avoid the need to define new error response formats for HTTP APIs.
>
> — Internet Engineering Task Force (IETF), https://tools.ietf.org/html/rfc7807

The goal of this RFC is to give a standard structure for errors in HTTP APIs that use JSON (`application/problem+json`) or XML (`application/problem+xml`).

```
HTTP/1.1 403 Forbidden
Content-Type: application/problem+json
Content-Language: en

{
  "type": "https://example.com/probs/out-of-credit",
  "title": "You do not have enough credit.",
  "detail": "Your current balance is 30, but that costs 50.",
  "instance": "/account/12345/msgs/abc",
  "balance": 30,
  "accounts": ["/account/12345", "/account/67890"]
}
```

This example from the RFC shows the user was forbidden from taking that action, because the balance did not have enough credit. 403 would not have conveyed that (it could have meant the user was banned, or all sorts of things) but there is text, and there is a type, which is just an error code in the form of a URL.

> Note that this requires each of the sub-problems to be similar enough to use the same HTTP status code. If they do not, the 207 (Multi- Status) [RFC4918] code could be used to encapsulate multiple status messages.
>
> A problem details object can have the following members:
>
> - "type" (string) - A URI reference [RFC3986] that identifies the problem type. This specification encourages that, when dereferenced, it provide human-readable documentation for the problem type (e.g., using HTML [W3C.REC-html5-20141028]). When this member is not present, its value is assumed to be "about:blank".
> - "title" (string) - A short, human-readable summary of the problem type. It SHOULD NOT change from occurrence to occurrence of the problem, except for purposes of localization (e.g., using proactive content negotiation; see [RFC7231], Section 3.4).
> - "status" (number) - The HTTP status code ([RFC7231], Section 6) generated by the origin server for this occurrence of the problem.
> - "detail" (string) - A human-readable explanation specific to this occurrence of the problem.
> - "instance" (string) - A URI reference that identifies the specific occurrence of the problem. It may or may not yield further information if dereferenced.
>
> — Internet Engineering Task Force (IETF), https://tools.ietf.org/html/rfc7807

Remembering all of this might be a little tricky, and asking every API developer to go read and memorize an RFC might not be particularly successful. As with most things, there are implementations that can be slotted into place for languages and web application frameworks to make the whole thing easier.

- **PHP:** zendframework/zend-problem-details
- **Java:** problem & problem-spring-web
- **Node:** https://www.npmjs.com/package/problem-json
- **Python:** https://github.com/cbornet/python-httpproblem

JSON:API

[JSON:API](http://jsonapi.org/format/#errors) is a standard for a lot more than just errors, it attempts to help with a lot of design choices for HTTP APIs, outlining the general format of requests and responses in JSON when working with HTTP APIs. In general it labels itself an anti-bikeshedding tool, and this is pretty accurate. HTTP API developers often feel like there are infinite possibilities, which can lead to a lot of discussions and arguments, so using implementations like

JSON:API can get folks on the same page. It will be covered in a few sections of this book, but for now we can just look at the errors section.

The following is an excerpt from the JSON:API standard at time of writing.

> An error object MAY have the following members:
>
> - **"id"** - A unique identifier for this particular occurrence of the problem.
> - **"href"** - A URI that MAY yield further details about this particular occurrence of the problem.
> - **"status"** - The HTTP status code applicable to this problem, expressed as a string value.
> - **"code"** - An application-specific error code, expressed as a string value.
> - **"title"** - A short, human-readable summary of the problem. It SHOULD NOT change from occurrence to occurrence of the problem, except for purposes of localization.
> - **"detail"** - A human-readable explanation specific to this occurrence of the problem.
> - **"links"** - Associated resources, which can be dereferenced from the request document.
> - **"path"** - The relative path to the relevant attribute within the associated resource(s). Only appropriate for problems that apply to a single resource or type of resource.
>
> Additional members MAY be specified within error objects.
>
> — JSON:API, https://jsonapi.org/format/#errors

Pretty familiar stuff here! Whilst RFC 7807 has an interface that suggests one error object be returned with multiple problems provided using extra properties, JSON:API errors are an array of error objects.

```
HTTP/1.1 422 Unprocessable Entity
Content-Type: application/vnd.api+json

{
  "errors": [
    {
      "source": { "pointer": "/data/attributes/firstName" },
      "title": "Invalid Attribute",
      "detail": "First name must contain at least three characters."
    },
    {
      "source": { "pointer": "/data/attributes/firstName" },
      "title": "Invalid Attribute",
      "detail": "First name must contain an emoji."
    }
  ]
}
```

That "pointer" is a JSON Pointer (RFC 6901), and can be used to point to the specific location in the HTTP request body that failed.

This is great for client developers who have a UI. They probably already have some logic which maps their form inputs to request data, so if they use that pointer they can trace the error back to a form input, and show custom validation errors even if they had not built that validation into their frontend.

NOTE	Clients love copying validation rules into their applications and that leads to all sorts of problems. We will look at how you can avoid that with JSON Schema later.

There are a lot of implementations for JSON:API. To be frank, some are better than others, by which I mean some are amazing and some are truly terrible. Check a few out.

4.1.5. Should You Use a Standard?

RFC 7807 was only released as a final RFC in 2016, and JSON:API is also fairly recent in the grand schema of the Internet. As such there are not many popular APIs using them. This is a common stalemate scenario where people do not implement standards until they see buy-in from a majority of the API community, or wait for a large company to champion it, and seeing as everyone is waiting for everyone else to go first nobody does anything. The end result of this stalemate is that most people roll their own solutions, making a standard less popular, and the vicious cycle continues.

Many large companies are able to ignore these standards because they can create their own effective internal standards, and have enough people around with enough experience to avoid a lot of the common problems around.

Smaller teams that are not in this privileged position, can benefit from differing to standards written by people who have more context on the task at hand. If you are Facebook then certainly

roll your own error format, but if you are not then RFC 7807 will point you in the right direction, and implementations make it easy.

4.1.6. 200 OK and Error Code

HTTP 4XX or 5XX codes alert the client, monitoring systems, caching systems, and all sorts of other network components that something bad happened.

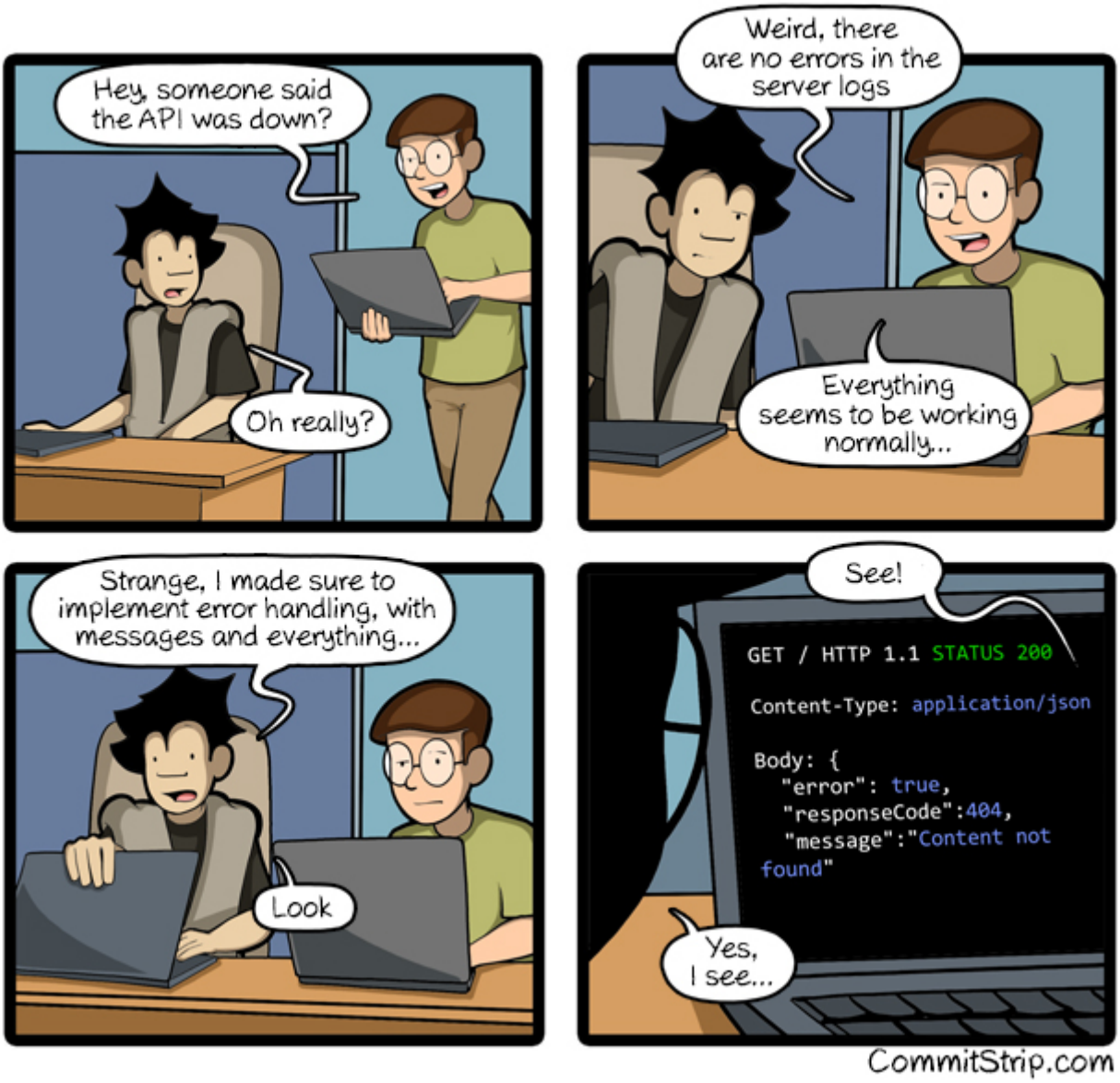


Figure 6. The folks over at CommitStrip.com know what's up.

If you return an HTTP status code of 200 with an error code, then Chuck Norris will roundhouse your door in, destroy your computer, instantly 35-pass wipe your backups, cancel your Dropbox account, and block you from GitHub.

4.1.7. Non-Existent, Gone, or Hiding?

404 is drastically overused in APIs. People use it for "never existed", "no longer exists", "you can't view it" and "deactivated", which is way too vague. That can be split up into 403, 404 and 410 but this is still vague.

If you get a 403, this could be because the requesting user is not in the correct group to see the requested content. Should the client suggest you upgrade your account somehow? Are you not friends with the user whose content you are trying to view? Should the client suggest you add them as a friend?

A 410 on a resource could be due to a user deleting that entire piece of content, or it could be down to the user deleting their entire account.

4.1.8. GraphQL

GraphQL has an error object format defined, so no choice needs to go into selecting one. It has a message and a location, the location being useful for GraphIQL and other visual query tools to help show which line the error was on.

```
{
  "errors": [
    {
      "message": "Field \"name\" must not have a selection since type \"String!\" has
no subfields.",
      "locations": [
        {
          "line": 4,
          "column": 10
        }
      ]
    }
  ]
}
```

There is also a `path` property made available in some error responses:

```
"path": [
  "name"
],
```

At first this may appear to be similar to the JSON:API pointer (JSON Pointer) approach, but is actually considerably more complex.

> If an error can be associated to a particular field in the GraphQL result, it must contain an entry with the key `path` that details the path of the response field which experienced the error. This allows clients to identify whether a `null` result is intentional or caused by a runtime error.
>
> This field should be a list of path segments starting at the root of the response and ending with the field associated with the error. Path segments that represent fields should be strings, and path segments that represent list

indices should be 0-indexed integers. If the error happens in an aliased field, the path to the error should use the aliased name, since it represents a path in the response, not in the query.

For example, if fetching one of the friends' names fails in the following query:

```
{
  hero(episode: $episode) {
    name
    heroFriends: friends {
      id
      name
    }
  }
}
```

The response might look like:

```
{
  "errors": [
    {
      "message": "Name for character with ID 1002 could not be fetched.",
      "locations": [ { "line": 6, "column": 7 } ],
      "path": [ "hero", "heroFriends", 1, "name" ]
    }
  ],
  "data": {
    "hero": {
      "name": "R2-D2",
      "heroFriends": [
        {
          "id": "1000",
          "name": "Luke Skywalker"
        },
        {
          "id": "1002",
          "name": null
        },
        {
          "id": "1003",
          "name": "Leia Organa"
        }
      ]
    }
  }
}
```

> — Lee Byron, https://github.com/graphql/graphql-spec

As you might have noticed here, GraphQL has an interesting spin on errors. With most HTTP APIs you are either trying to do something and succeed, or you fail, and it is usually rather binary.

NOTE	An exception to that rule might be trying to fetch a collection of things, searching, etc. and getting an empty result, but that is not an error, that is a fetch returning an empty result.

GraphQL has a different take, and it tries to provide as much data back even when a request contained incorrectness. It usually seems more like GraphQL considers errors to be merely warnings, which is why you can have data and also have errors, and that not be an issue.

When trying to work out where people should put their own errors, there are a lot of disparate instructions. Some folks saying things like:

> If the viewer should see the error, include the error as a field in the response payload. For example, if someone uses an expired invitation token and you want to tell them the token expired, your server shouldn't throw an error during resolution. It should return its normal payload that includes the error field. It can be as simple as a string or as complicated as you desire:
>
> ```
> return {
>   error: {
>     id: '123',
>     type: 'expiredToken',
>     subType: 'expiredInvitationToken',
>     message: 'The invitation has expired, please request a new one',
>     title: 'Expired invitation',
>     helpText: 'https://yoursite.co/expired-invitation-token',
>     language: 'en-US'
>   }
> }
> ```
>
> — Matt Krick, https://itnext.io/the-definitive-guide-to-handling-graphql-errors-e0c58b52b5e1

This is back to creating custom error formats, despite GraphQL having one bundled… Not sure. Accepting feedback on best practices if anyone finds them before this book goes to print.

4.1.9. gRPC

gRPC does not care about how you do errors, do what you want. The official documentation for gRPC Core has written down some pre-defined error codes, but you can invent your own too.

The official documentation pushes readers towards http://avi.im/grpc-errors/, which is a convenient set of SDKs for most of the programming languages gRPC is implemented in. The code helps API

developers use the status codes defined in gRPC Core, and add their own text too.

Chapter 5. 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.

5.1. 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.

5.1.1. 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

5.1.2. 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.

5.1.3. 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.

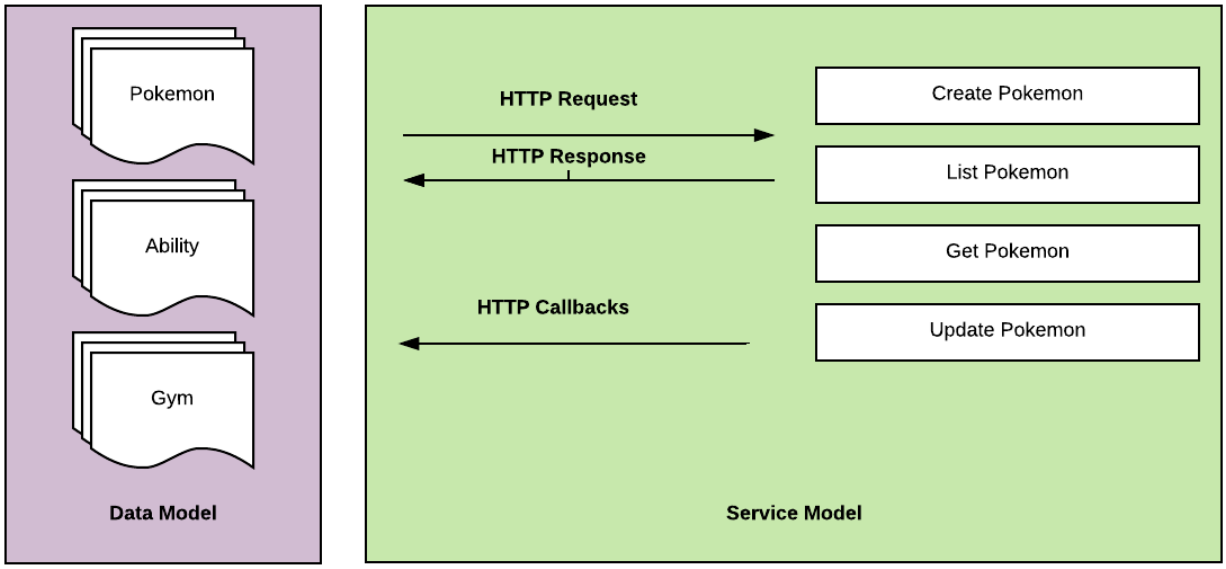
> The Web Services Description Language is an XML-based interface definition language that is used for describing the functionality offered by a web service. The acronym is also used for any specific WSDL description of a web service, which provides a machine-readable description of how the service can 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.

5.2. 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.

5.3. 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.

5.3.1. 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 are 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.

5.4. 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].

5.4.1. 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.

5.5. 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.

Chapter 6. Design Theory

Coming soon.

Chapter 7. Validation

Coming soon.

Chapter 8. Hypermedia Controls (HATEOAS)

Coming soon.

Chapter 9. HTTP/1, HTTP/2 and HTTP/3

Coming soon.

Chapter 10. Asynchronous Paradigms

So far the discussion of these paradigms has probably seemed rather similar, they are all following a request and response structure. Getting a response from a request is one way to do things, but there are lots of situations where a client might expect more than one response, or a response with more information later as more work happens on the server side.

10.1. Client-Server "Real-Time" Models

Let's have a quick run through of some of the common models for doing asynchronous stuff. Even thought you might not want to use some of these, it is good to know what they are, even if that is only so you can recommend not doing it when somebody else brings them up in a planning meeting.

10.1.1. Short Polling

The first approach that often gets mentioned is short polling, which is the client-server equivalent of a child in the backseat of the car asking "are we there yet? are we there yet? are we there yet?". The client is making the same GET request to the same resource every few seconds, waiting for some sort of change in the answer.

```
00:00:00 C-> Is the cake ready?
00:00:01 S-> No, wait.
00:00:01 C-> Is the cake ready?
00:00:02 S-> No, wait.
00:00:02 C-> Is the cake ready?
00:00:03 S-> Yeah. Have some lad.
00:00:03 C-> Is the other cake ready? ...
```

The only benefit of short polling is that it is *very* easy to implement. The client can just write a while loop that makes a request, checks for a response, then sleeps for a few seconds before making another request.

The main downside with short polling is that it's a very chatty approach, which makes it awful to scale. The more clients you have making these requests the more load your API has to handle, and it can grow exponentially. Conditional network caching can help a bit here, but even then it sucks the battery life on mobile devices.

Generally short polling is rubbish, and you should try to avoid it.

10.1.2. Long Polling

A slight tweak on the concept of short polling, long polling maintains an open connection with the server until the answer comes in.

```
00:00:00 C-> Is the cake ready?
00:00:03 S-> Yeah. Have some lad.
00:00:03 C-> Is the other cake ready?
```

See here that instead of answering immediately, the server waited 3 seconds before responding with a single update.

Generally folks recommend long polling when the client is only waiting for one answer, so if they are ordering a single cake and it might be ready soon, then great, let that thread focus on replying about the one cake, and when that cake is doing the connection is closed. If the client wants multiple status updates, or is interested in multiple cakes, this might not be the way to go.

Personal experiences tell me this sort of blocking activity can lead to quite a mess when you have more than a smattering of users. Blocking an entire web thread for 3 seconds can certainly be a problem when you have a lot of clients asking about a lot of cakes.

For example, in Rails land this is a pretty common server config:

```
workers Integer(ENV['WEB_CONCURRENCY'] || 2)
threads_count = Integer(ENV['RAILS_MAX_THREADS'] || 5)
threads threads_count, threads_count

rackup      DefaultRackup
port        ENV['PORT']     || 3000
environment ENV['RACK_ENV'] || 'development'
```

If there was only one dyno running (a.k.a instance, pod, whatever) then 2 * 5 = 10 and thats only 10 cake orders happening at the same time and you've run out of threads for other things, like accepting payments, or registering a user.

Even having a slightly bulkier setup, with more dynos, and more processes, and more threads, its simply a numbers game where you are throwing more money at the situation.

10.1.3. Web Hooks

Both these polling approaches are essentially a pull model, so lets look at a push model: Web Hooks. Clients register a "callback URL" through a GUI or API, and they may also specify specific types of information they are interested in receiving. When relevant events happen on the server, it will fire a payload at that URL, which usually takes the form of a HTTP POST request.

Examples of Web Hooks in popular APIs.

- Slack API > Incoming Webhooks
- Stripe API > Webhooks

A few years ago my friend and I made a website that sold silly programmer joke t-shirts. We created the whole thing over beers, and we made some whopping mistakes. Our payment acceptance sequence looked like the synchronous one above, as we didn't care about speed at that early stage.

The biggest mistake was that when the payment API gave us a positive response to the `POST` request, we immediately considered the payment good and marked the shirts for dispatch.

It turns out there was a web hook we were meant to listen out for, and this would confirm if the payment had been accepted, or not… Seeing as we totally ignored that, there were a few t-shirts that went out for free! Oops.

This was simple enough, we added a `/callback` endpoint, registered it with the payment gateway, and they would fire updates at it a few seconds or minutes after a payment attempt was made. We simply marked orders as pending in the mean time, made a order status page, and they would get an email later. We could have added 30 second short polling on the status page if we cared, or just had the page refresh, but it was midnight and the bartender had just opened the whiskey.

Web hooks are often thought of as a bit of a dark art, because they happen outside of the normal request/response stuff that API developers are used to. It can be hard to see them happen, tricky to confirm they are sending what you think, and used to be tricky to document.

These days there are loads of awesome tools around to help. OpenAPI v3.0 added support for Callbacks.

NOTE	It makes me sad that this is called callbacks when it should be called web hooks, and it's on the swagger.io website when the specification has been renamed to OpenAPI. Ahh well.

Tools to help with this stuff:

- PipeDream - Formerly known as RequestBin, create an endpoint and register that as the callback URL, and see what traffic is coming in. Then maybe copy that to use in a test suite
- ngrok - Create a tunnel to your locally running application and register your local app as the callback URL

That ngrok example might have made you notice an issue with web hooks: you need some sort of DNS or known IP address to fire the payload at. If the client is written in PHP, Java, Node, Ruby, etc. and running behind a web server then yeah, it will probably have one of those, but if the client is a JavaScript application running in the users browser then that isn't going to work.

For that we need another approach, and that is…

10.1.4. WebSocket

WebSocket is a whole other protocol to HTTP, that operates over TCP directly. Instead of a client asking the server over and over again, or making a blocking connection to the web server, the client establishes a connection to the server, and then the two parties can send messages back and forth in real-time, and the server doesn't need to know the IP address of the client because the client initiated the connection. Cool!

```
Both protocols are located at layer 7 in the OSI model and depend on TCP at layer 4.
Although
they are different, RFC 6455 states that WebSocket "is designed to work over
HTTP ports 80 and 443 as well as to support HTTP proxies and intermediaries"
thus making it compatible with the HTTP protocol. To achieve compatibility, the
WebSocket handshake uses the HTTP Upgrade header to change from the HTTP
protocol to the WebSocket protocol.
```

Basically whatever client and server side code you are using will do the same thing: the server provides a WebSocket URL to the client, and they connect, and messages can then go either way. Just like HTTP has `http://` and `https://`, there is `ws://` and `wss://`.

> const socket = new WebSocket('ws://localhost:8080/cakes/abc123');
>
> socket.addEventListener('open', function (event) { socket.send('Hello Server!'); });
>
> socket.addEventListener('message', function (event) { console.log('update RE that cake ', event.data); });
>
> — javascript

The fact that HTTP and WS are quite similar in many ways, and that they are compatible in general, make it really easy to add WebSocket's into your HTTP API by using a web server that has it built in. MDN maintain a list of WebSocket tools for folks interested in giving it a try, including web servers that support them.

You can punt figuring out how to run a WebSocket server entirely, but using hosted solutions like Pusher and PubNub.

10.1.5. Message Brokers & Job Queues

One important component here is the server being able to "do stuff later", instead of the client twiddling their thumbs waiting for updates. How exactly do we do that? Generally the idea is to push an event into a job queue, and then some sort of worker process will pick up that message and "work on the job".

Sometimes this is done with tools based on AMQP or MQTT, two protocols designed for "Message Brokering" in general. That is a whole topic by itself, and job queues are just one of many possible use cases that message brokers can cover. These tools run their own server which acts as a "broker", and they send the messages off to "subscribers" who can get things done.

A more simplistic approach for job queues it to use tooling like Sidekiq (a popular Ruby tool for handling job queues), or the more recent polyglot version Faktory. These sorts of solutions are popular because they just run off of Redis as the queue, and often run in the same codebase just on different threads. This makes the setup a bit less confusing for many.

Whatever message broker / job worker / do it later approach you use, you need to let the client

know that work is being done in the background. For a HTTP API this is usually done with a `201 Created`, or `202 Accepted` response code, which also lets the client know they should look for some sort of link to get their updates.

10.2. Real World Examples

Alrighty, that was a lot of text, even for the Theory part of the book, so let's get some visuals in here.

Let's start accepting payments with a HTTP API, where a user is buying something something from your website. Once they make a payment, and they want to know if it has been accepted or not, so they know if they are getting their hilarious t-shirt.

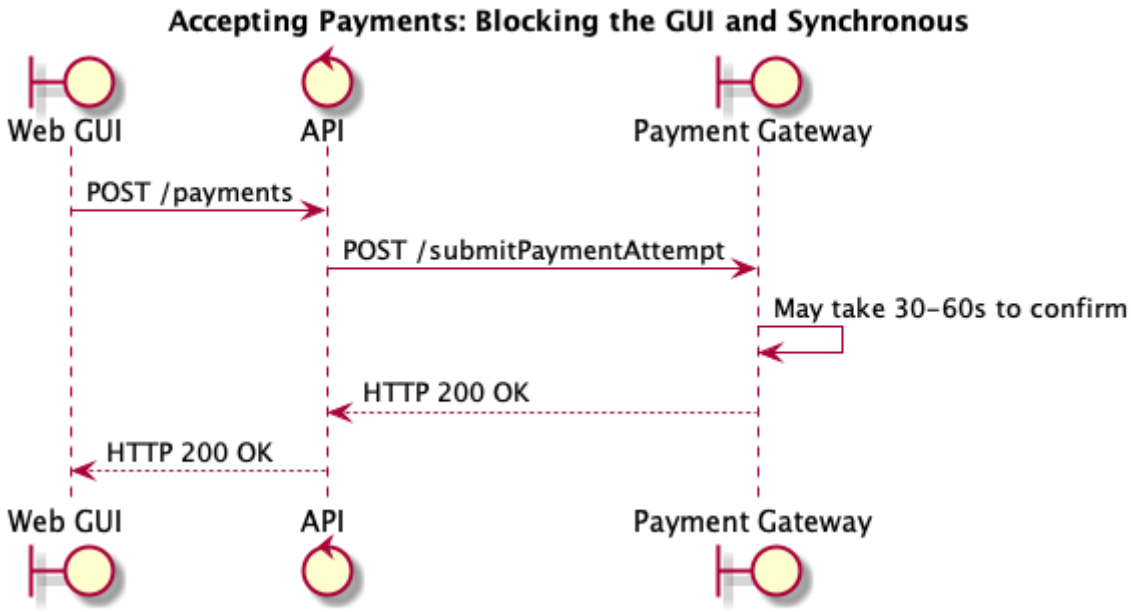


Figure 7. A completely synchronous approach to handling a payment from a GUI, with your API accepting the information and sending it off to an external API acting as a payment gateway.

If the world was a perfect place, this would be fine. All messages would somehow move faster than the speed of light, all servers would alway be responding perfectly, and no messages would be lost in transit, meaning the user gets a nice, quick, consistent experience and the user interface does not leave them guessing.

Sadly none of this is true, but people seem to design their data flows like it might be. The above flow will leave the user sat there twiddling their thumbs for however long it takes that API gateway to respond.

There is one issue here that the user is twiddling their thumbs for 30 seconds, but a bigger issue is that the API server is twiddling its thumbs for 30 seconds. You realize that the payment gateway is too slow to use synchronous requests to talk to them, so lets shift that to a worker.

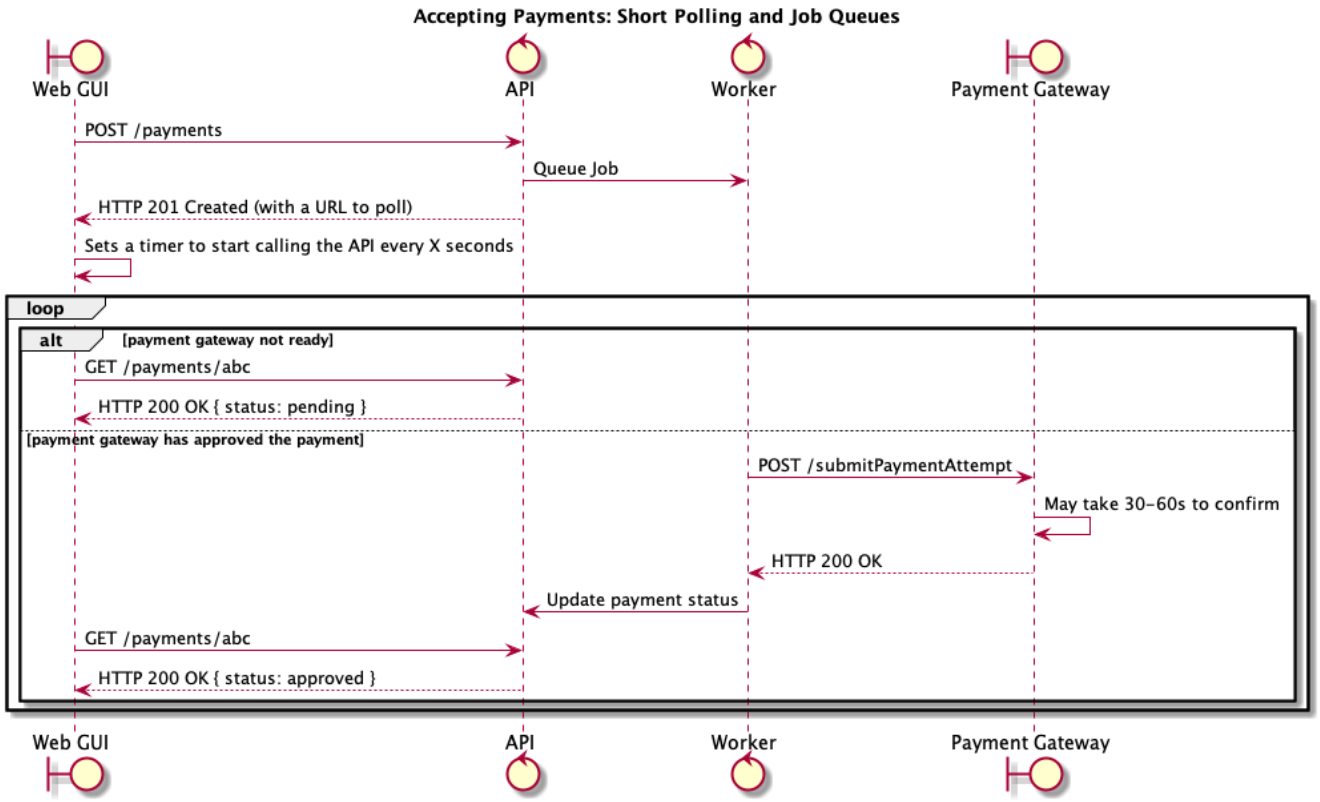


Figure 8. Letting the API do a bit less work up front, shifting the waiting work to a job queue, and letting the GUI short poll for updates.

Alright, better. Our client is a bit chatty and is generating a lot of overheard, but hey they are getting relatively real time updates. Maybe this is good enough, but maybe we want to improve this even more with a WebSocket.

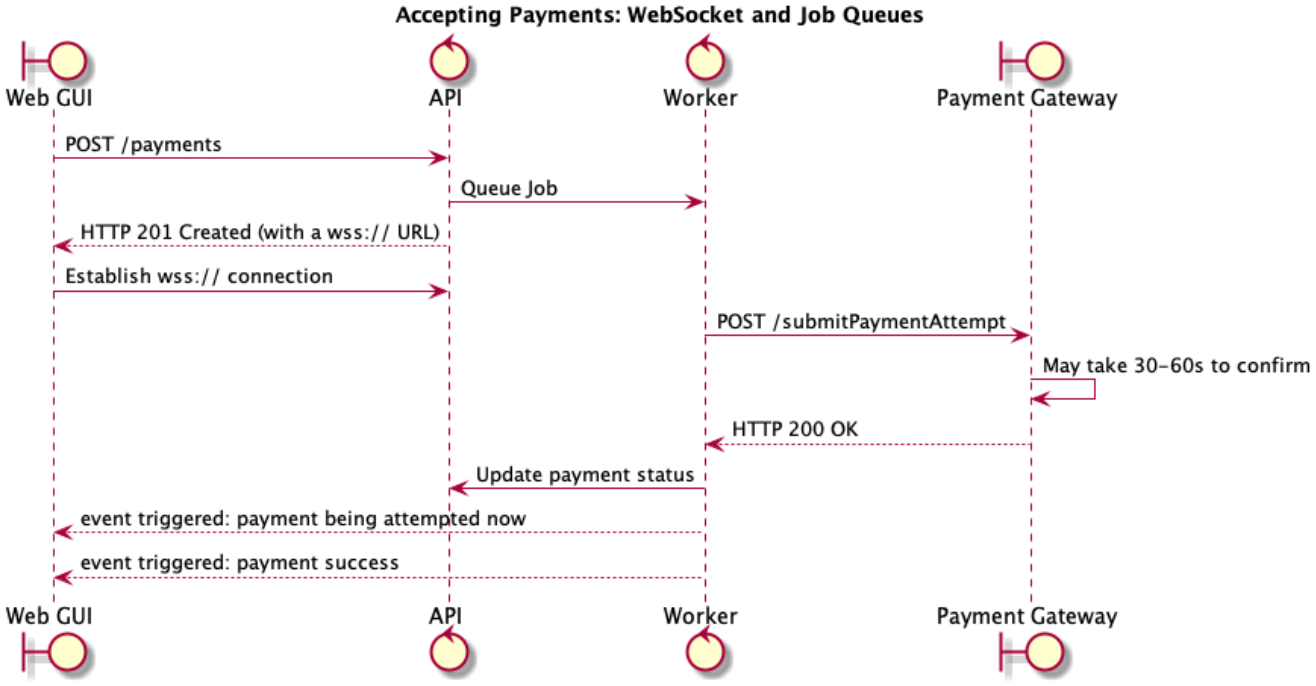


Figure 9. Removing the short polling from the client using a WebSocket.

Now the client is getting real time updates, and not just "is it done or not" but actual progress on various things happening. Also the background worker is able to talk directly to the client, instead of just updating the state in the API and hoping the client checks for a change.

Again this might be just fine, but the last remaining bottleneck is the interaction with this slow third party API. Even though the slowness is in the background worker, maybe the worker is backing up a bunch, due to high demand. During Black Friday this might mean your workers are so backed up they're taking 20 minutes to get to updating, because most of them are spending all the CPU cycles just waiting.

Maybe the company providing this payment gateway realized that 30-60 sync requests are daft, and implemented Web Hooks on their end. Let's update our approach to support that.

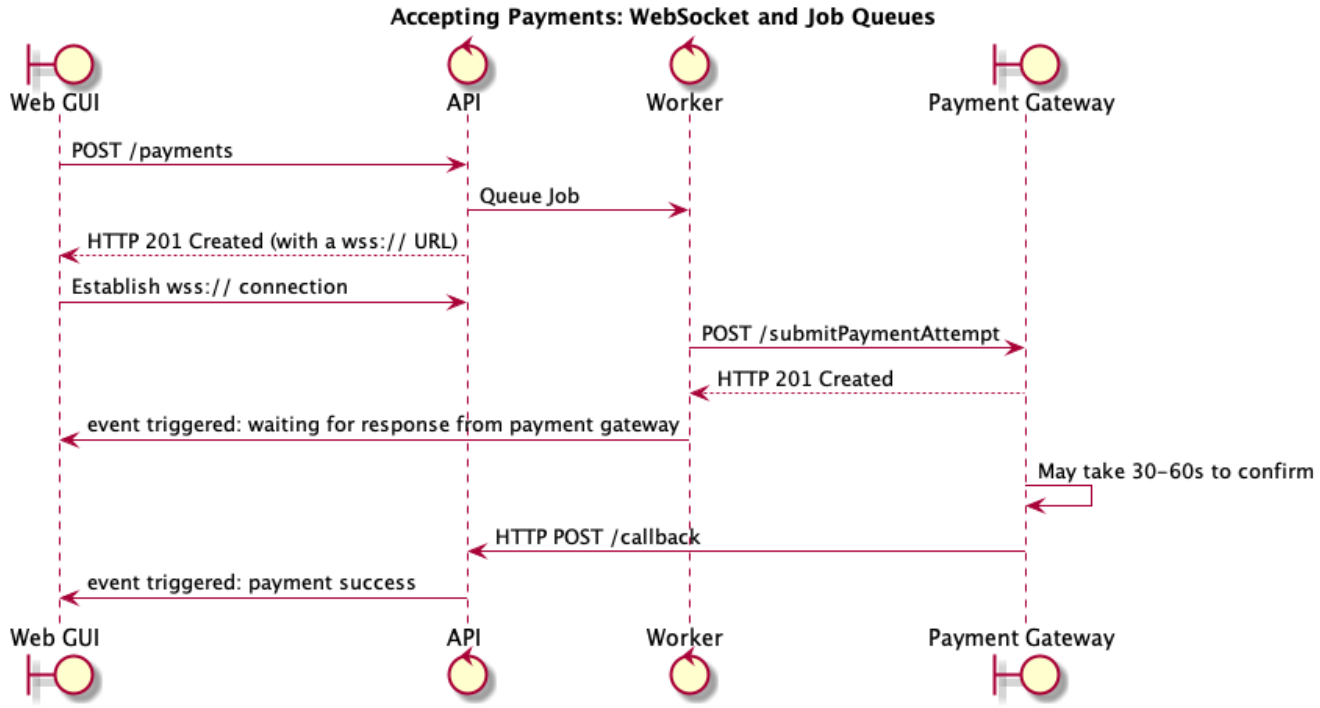


Figure 10. Removing the short polling from the client using a WebSocket.

Some notable differences here, mostly that the worker is just responsible for passing on the job to the payment gateway. Seeing as the API has registered a callback URL, the payment gateway can fire a HTTP POST payload right at our API, and the API can push the message into the WebSocket to update the client.

Why would you still want to use a worker if the third-party payment gateway is doing things asynchronously itself? Well, if that third party gateway goes down, or is acting slowly, your API will also fail, or perform slowly. Chucking a background worker in there adds a bit of resilience to your system, especially as most background workers have automatic retry logic for failed jobs.

10.3. Asynchronous REST APIs

Some people think that polling is inherently part of how some APIs (like REST) are meant to work, but that is not really the case. The REST dissertation does talk about REST being a pull model, and it is, but it can absolutely be supplanted with push too.

```
The interaction method of sending representations of resources to consuming components
has some parallels with event-based integration (EBI) styles. The key difference is
that EBI styles are push-based. The component containing the state (equivalent to an
origin server in REST) issues an event whenever the state changes, whether or not any
component is actually interested in or listening for such an event. In the REST style,
consuming components usually pull representations. Although this is less efficient
when viewed as a single client wishing to monitor a single resource, the scale of the
Web makes an unregulated push model infeasible.
```

You can use any of these methods in your REST API, or any other HTTP API. As mentioned before, using HATEOAS you can return links in your API data (or headers), and those links can be to whatever protocol you like. Give the client a link to a `wss://` and they'll know to connect to a WebSocket, or provide a HTTP link and they can poll it. Or both! Choice is fun.

10.4. Asynchronous GraphQL APIs

GraphQL has a special `subscription` keyword equal to `query` and `mutation`. There is no information on how these work on the GraphQL.org website at time of writing, but third-party vendors explain the situation fairly well.

Basically GraphQL subscriptions add real-time functionality to GraphQL *somehow*, and that somehow is usually implemented via WebSockets. Apollo have some docs on how they suggest implementing GraphQL subscriptions in their documentation, but if you are using something other than Apollo you will have to work that one out for yourself.

10.5. Further Reading

There are a lot of implementations, super-sets and alternatives out there. Here are some pieces of technology that you should know about, that are not being covered in the book at this point.

- **Mercure:** https://mercure.rocks/
- **Kafka:** https://kafka.apache.org/

Part Two: Planning & Design

Chapter 11. 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

11.1. 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.

11.2. 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 12. 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.

12.1. 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.

12.2. 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.

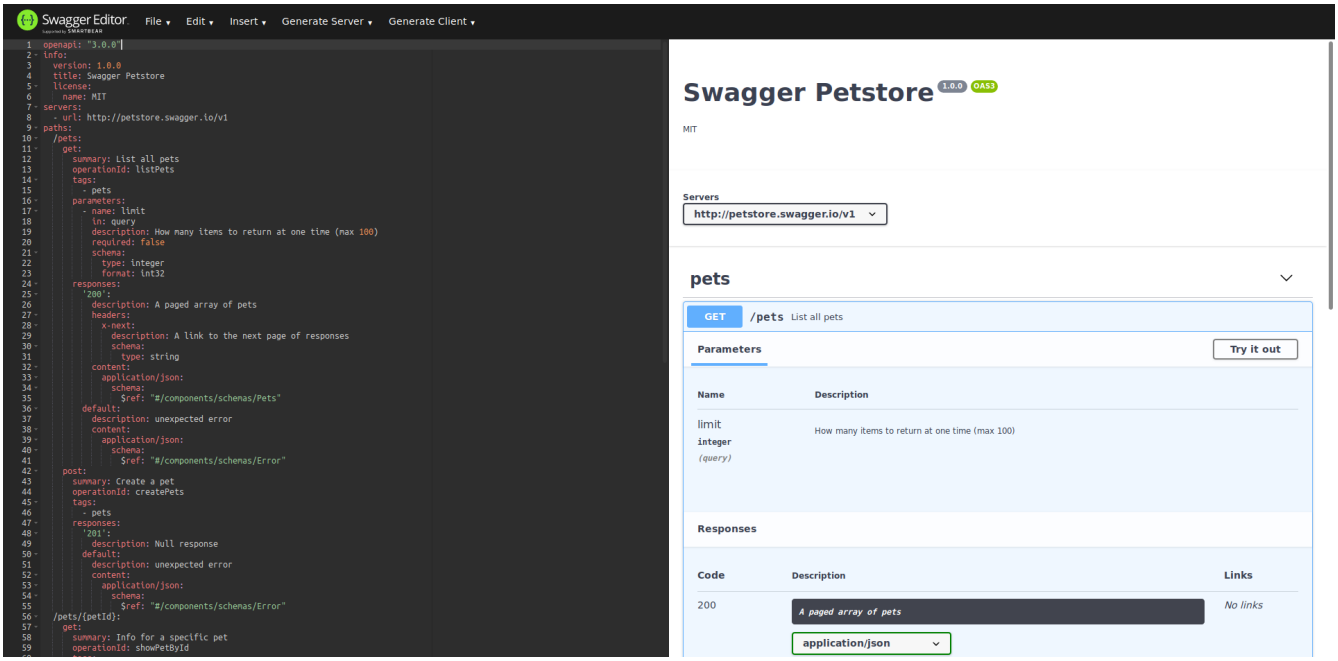
Other editors can take the form of desktop applications, which can be used throughout the life-cycle, loading local files and allowing you to edit them and commit them back to the repository (or wherever they live) at will.

12.3. Standard HTTP Editors

12.3.1. Swagger Editor

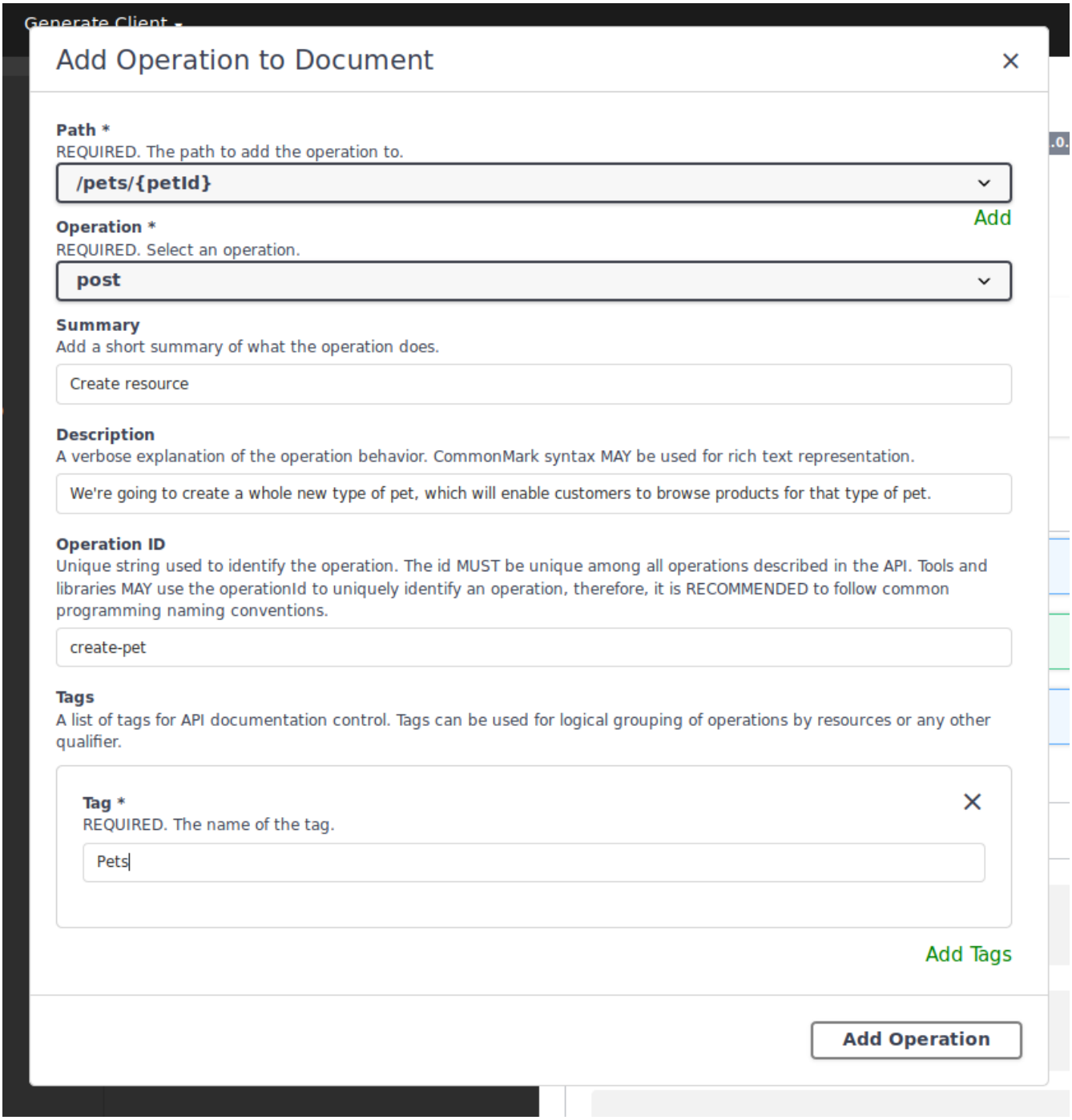
Swagger Editor is a very simple text-based editor from SmartBear. You can find it hosted on swagger.com, and it can be downloaded and run locally for free.

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.

12.3.2. 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.

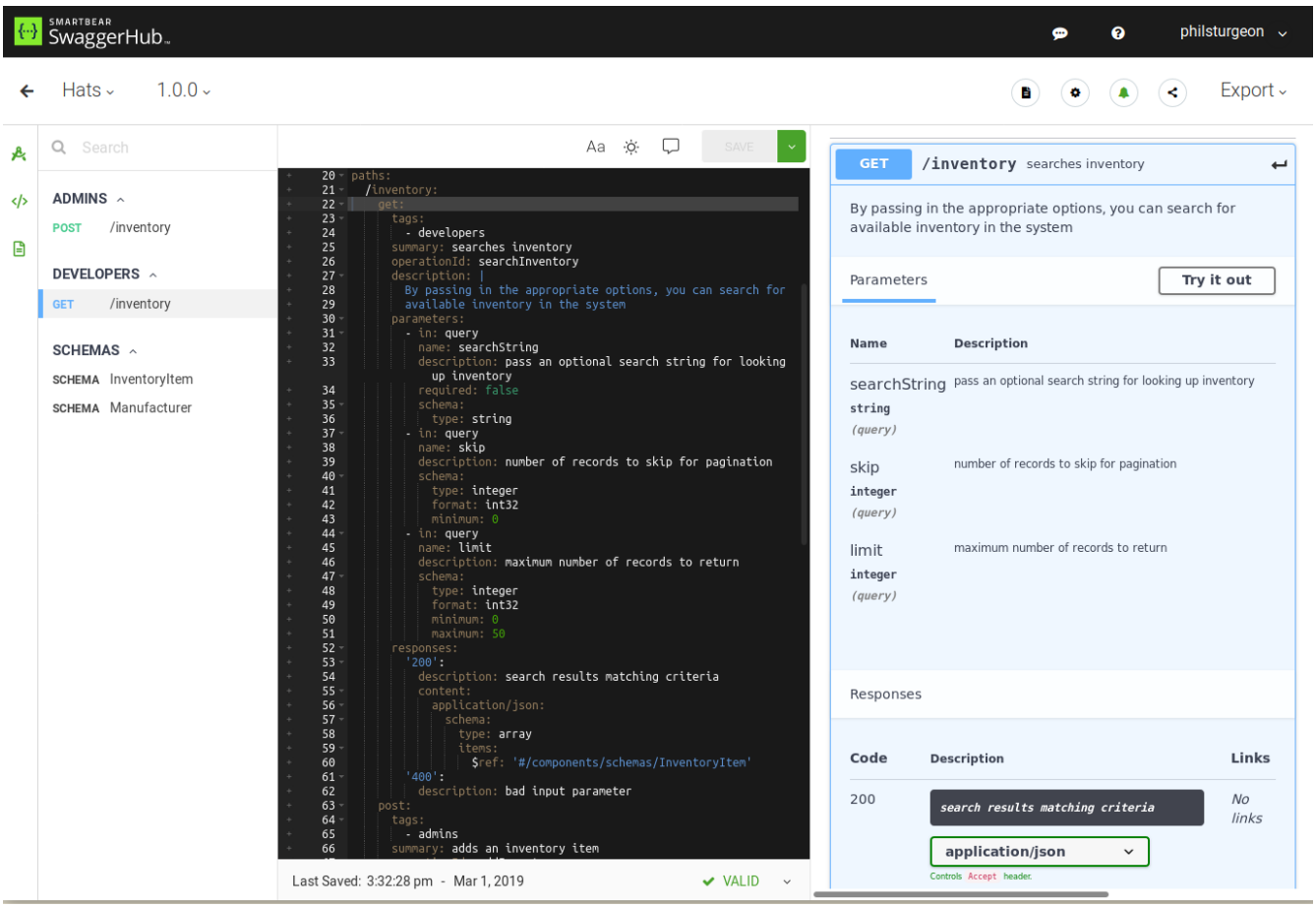


Figure 11. In this demo of SwaggerHub you can see Swagger Editor in full swing, with a few other options floating around the outside.

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.

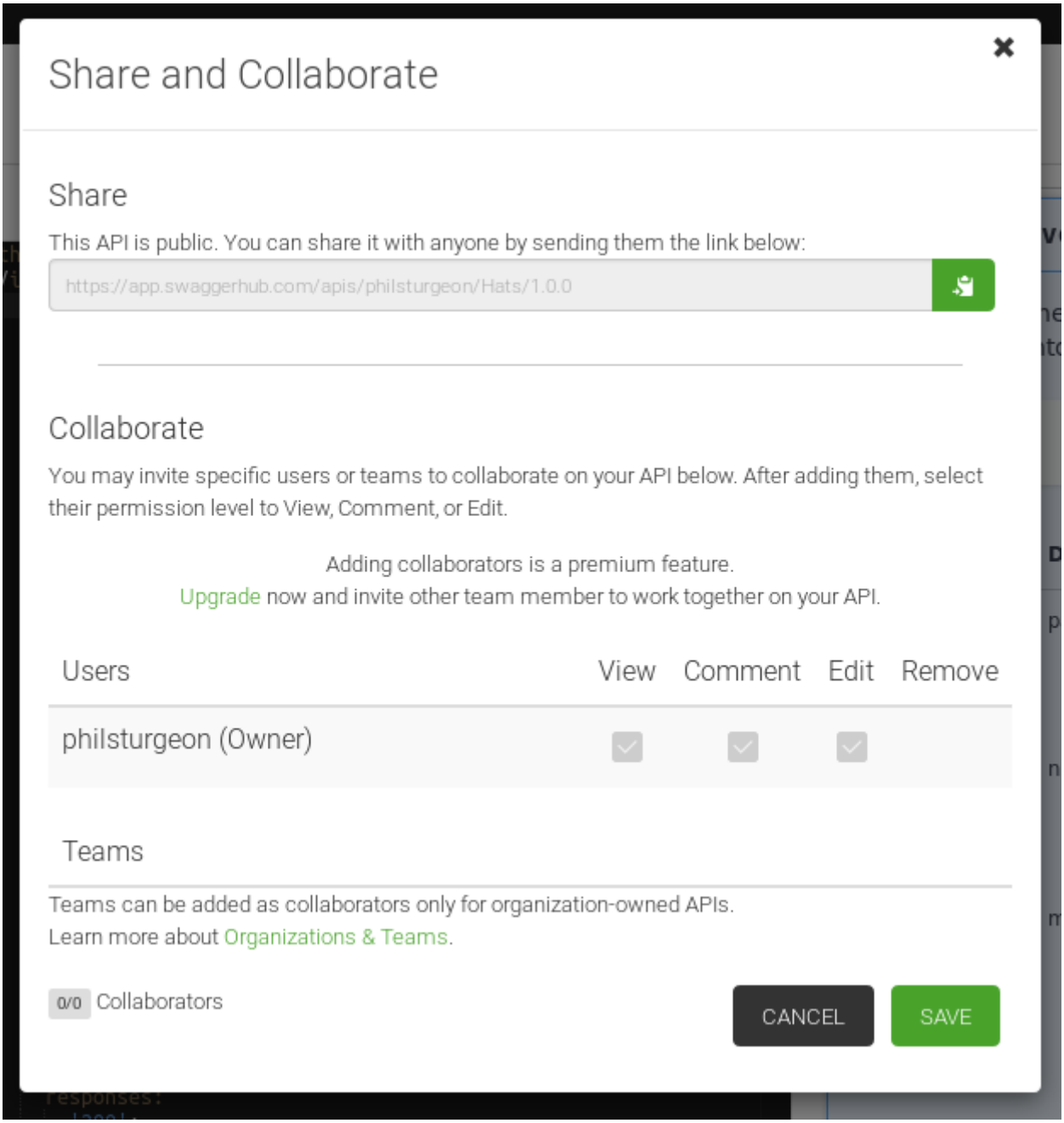


Figure 12. Get collaborating with other folks without having to email YAML files around!

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.

12.3.3. 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.

12.3.4. 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.

12.4. 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.

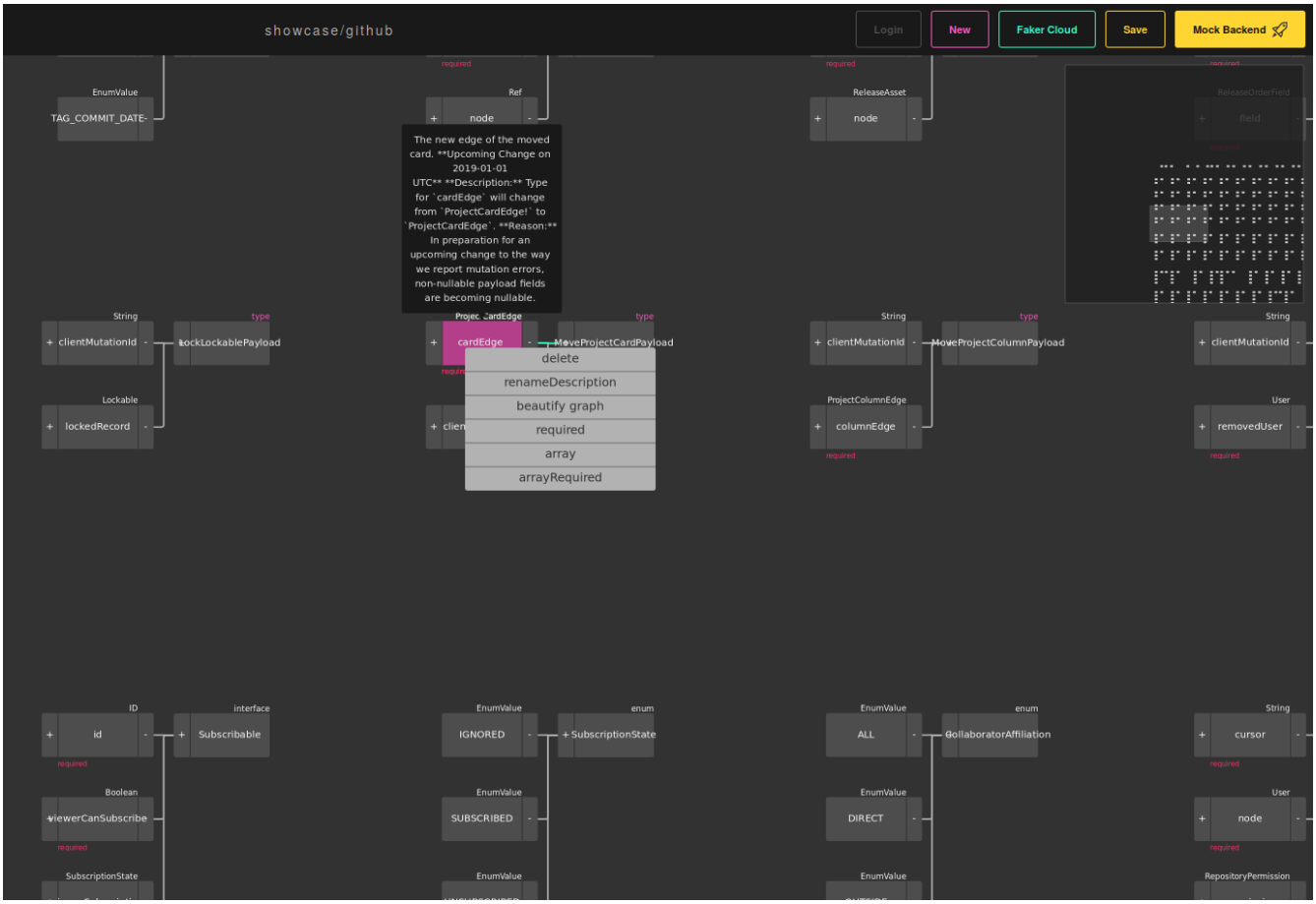


Figure 13. A screenshot of the "Github example" on graphqleditor.com

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.

12.5. 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

12.6. 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.

12.7. 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 13. 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.

13.1. 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.

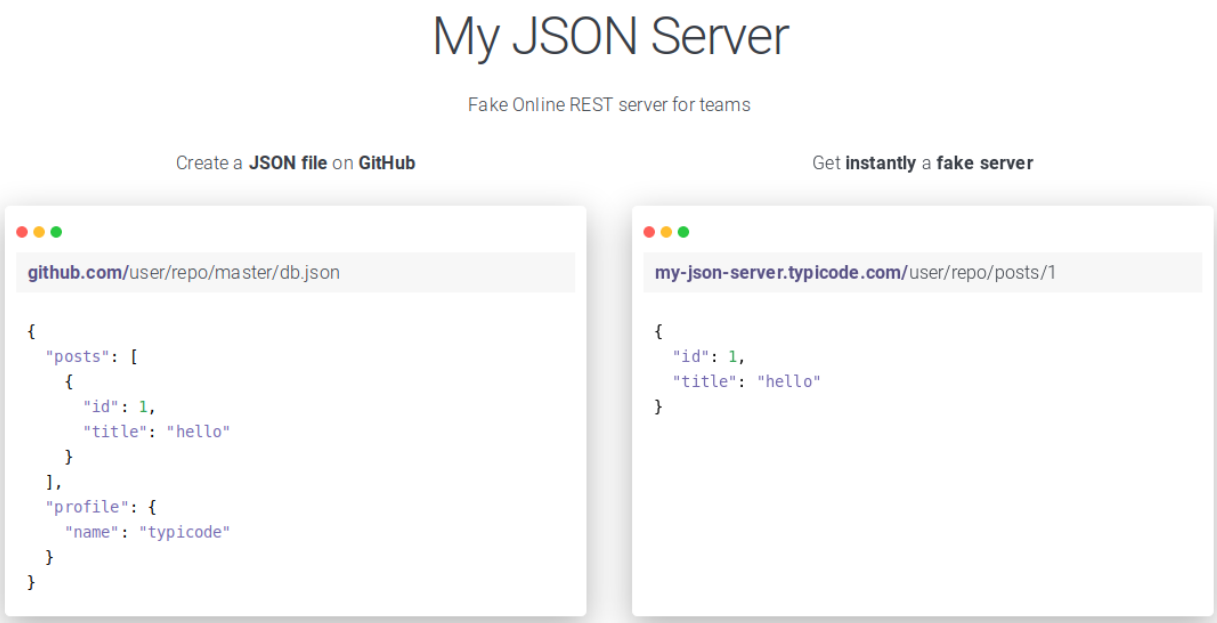
```
$ json-server --watch db.json
```

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.

13.2. 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.

13.3. Standard HTTP Mocking

13.3.1. Hosted Mock Servers

SwaggerHub

Stoplight

13.3.2. Free/Open-Source Tools

Prism

API Sprout

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

13.4. GraphQL Mocking

GraphQL Editor Cloud

Apollo - GraphQL Tools

13.5. 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.

13.6. 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.

13.7. Next

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

Chapter 14. 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.

14.1. OpenAPI

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

14.1.1. Swagger UI

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

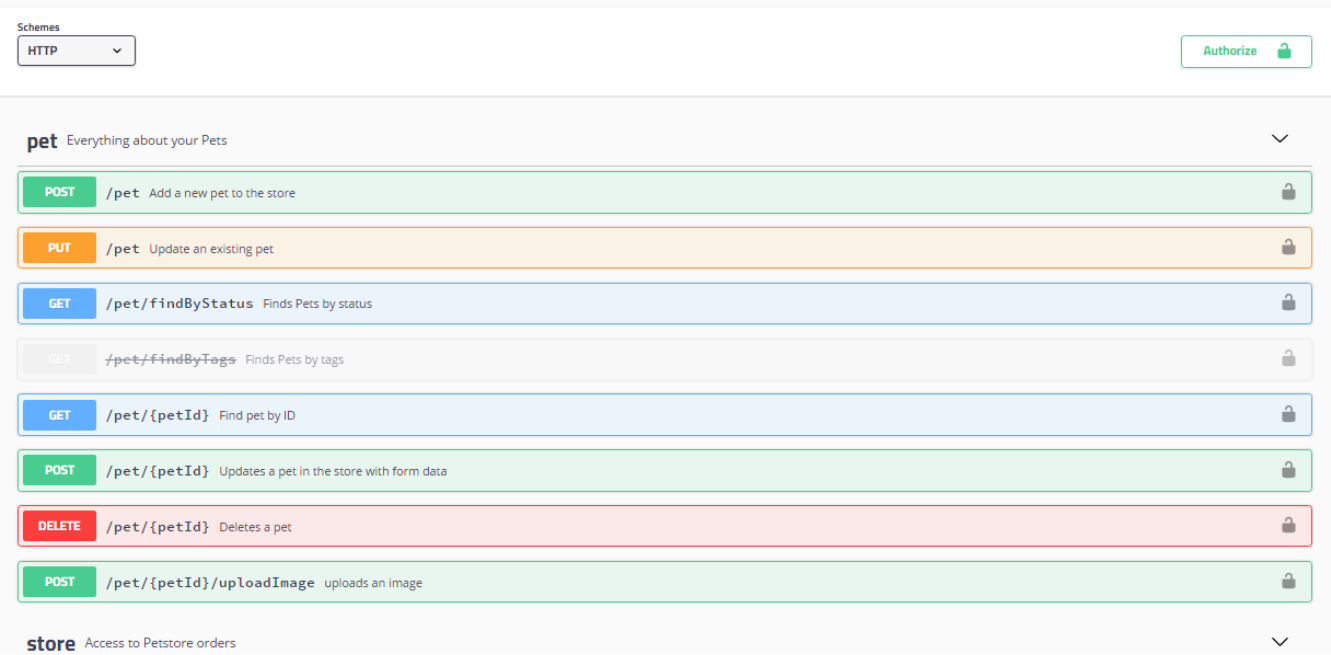


Figure 14. A screenshot of Swagger UI, which looks like it was designed by not just a developer, but one who was a big fan of RPC.

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.

14.1.2. 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.

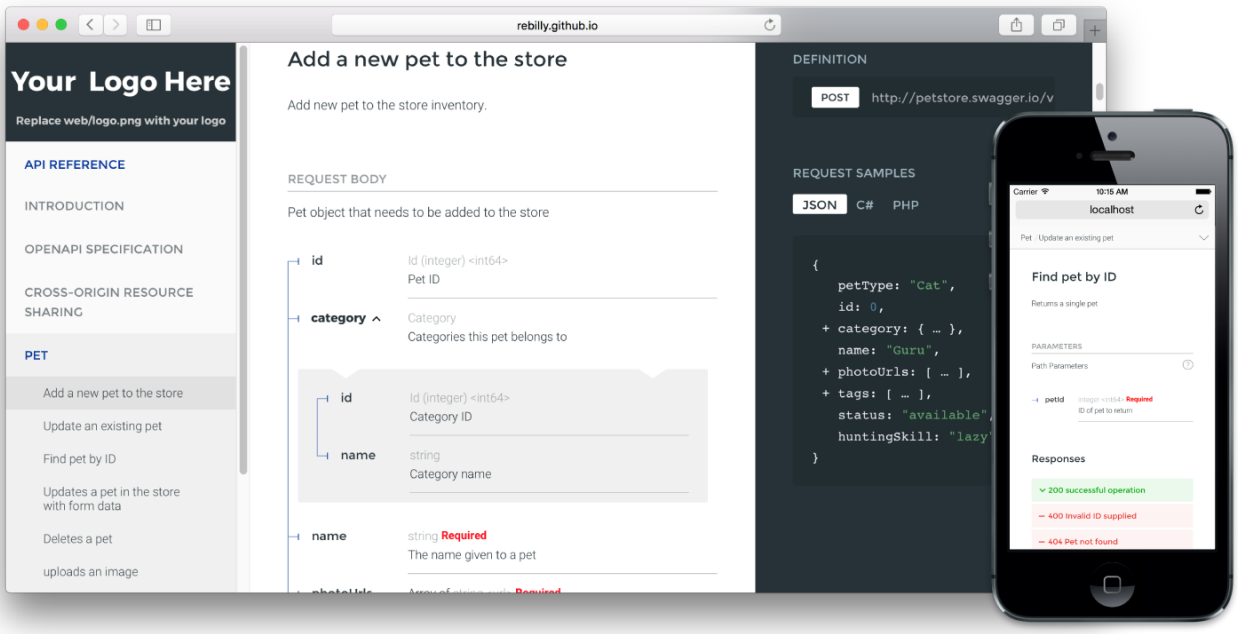


Figure 15. Fancy/modern looking three-col API documentation using ReDoc.

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.

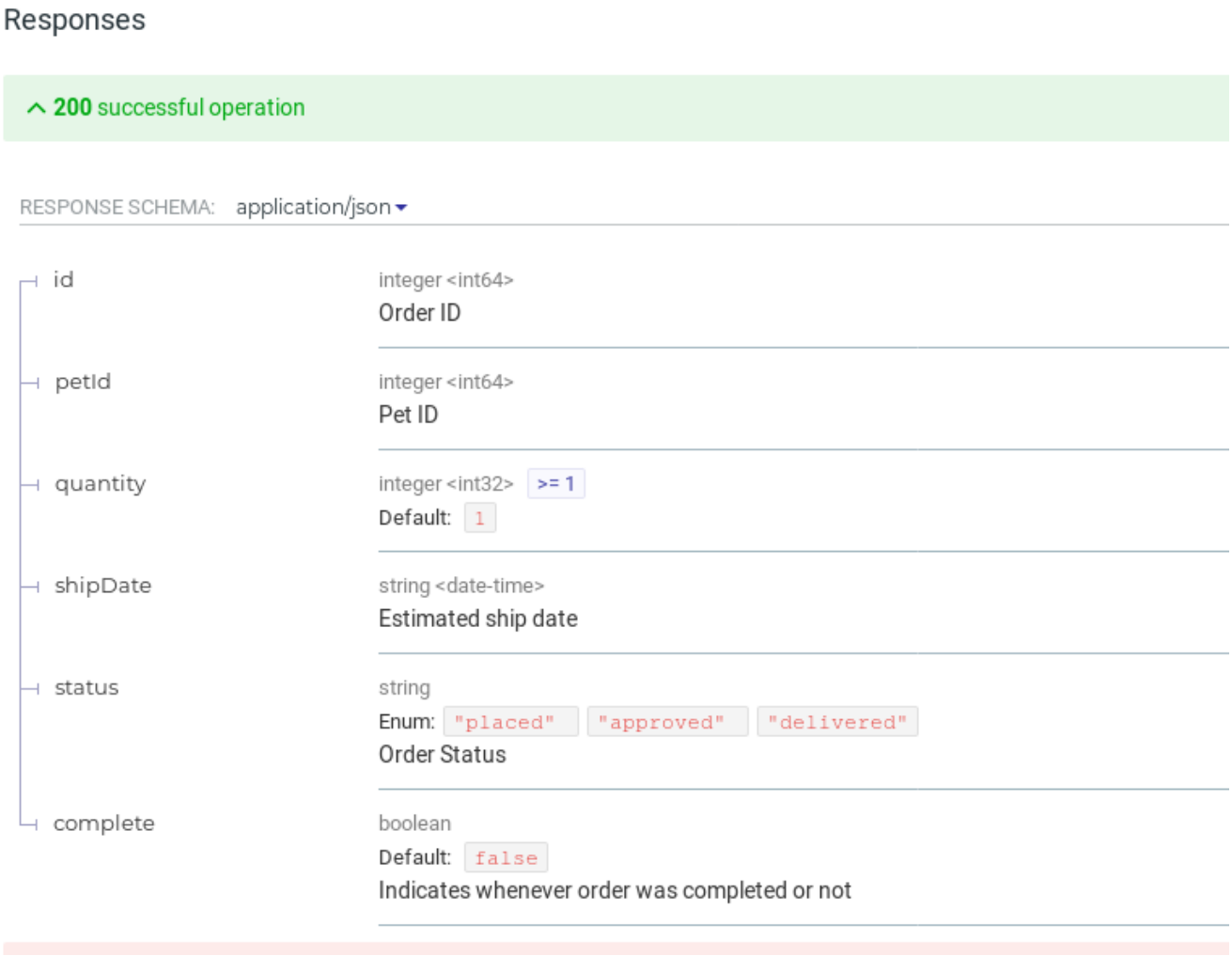


Figure 16. Down arrows let you expand nested objects to avoid initial clutter.

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
Server started: http://127.0.0.1:8080
    Watching /Users/phil/src/pokeapi for changes...
```

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.

14.2. 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.

14.3. 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.