Chapter One

Introduction

Web APIs are just code somebody else wrote, that lives on a server controlled by somebody else. The acronym API itself stands for "Application Program Interface", and that's what it says on the tin: an interface for 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 its method names and arguments to be an API.

Essentially it's all the same concept, but when talking about Web APIs we're talking about the utilization of network protocols such as HTTP, AMQP, etc., and instead of command-line arguments or method parameters, we're working with URLs, query string parameters, or chunks of data like JSON.

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 give you data for a single-page JavaScript application, handle payment information so you don't have to worry about storing credit card details, post a Facebook status 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.

For a long time my job was designing and building APIs, Then it was coaching and advising other teams how to build APIs. In more recent years, it has shifted towards helping different teams building different software interact with each others systems. These are sometimes architectural concerns, but a lot of it comes down to clients calling other systems and knowing what to do with the responses.

Essentially the role has been making HTTP interactions reliable, warning people of potential problems like: don’t assume everything is going to be JSON, normalizing state management into the API instead of having loads of clients doing subtly different guesswork, how to configure timeouts to stop half the company’s applications crashing, and improving performance with smart usage of HTTP caching. This advice seemed like it should also go into a book, because I’ve learned a substantial amount from my time working at WeWork.

This book aims to help you interact with an API that is already built. It will explain things from the users perspective, and if you do happen to be an API developer, this might give you some insight into how users will be interacting with your API. Everyone wins. That said, if you want help building fantastic web APIs, this might not be first book you should read, and instead you should consider picking up a copy of *Build APIs You Won’t Hate*. This is for the frontend folks, and those building systems that talk to other systems.

Terminology

APIs are built by all sorts of folks; maybe another company, some large like Google or Facebook, startups, governments, or charity organizations. They could also be coworkers on another team, or another department, on another floor (or continent). Maybe you built an API because you’re a full-stack developer who can do frontend and backend! This whole time we’re going to refer to “them” as the guardians of these Web API(s), and assume we have little control over the API.

Those writing code that interact with APIs are usually called “API Clients” or “user agents”, and that is usually a frontend application (browser app, iOS app, Android app, etc.), or it can actually be another backend application. Backend applications talking to other Web APIs share a lot of the same properties as a frontend application talking to another API, so this book is aimed at all of you.

Web API, microservice, service, etc, they’re all terms that mean subtly different things, that are often used interchangeably by some. For the duration of this book, we’re just going to say API. A micro service, or a service, will probably have an API, otherwise it’s not doing anything. So when you see the generic term API, you know we're talking about a web API, which might be part of a service, micro service, a gateway to multiple microservices, or a giant monolithic application with some API endpoints jammed into it. It's all the same as far as API interaction is concerned.

Do we really need to read a book?

Integrating an API is meant to be easy, and backend engineers will often just throw that phrase around without really understanding what they're asking from people.

If you've never integrated with an API then there's a bit of a learning curve. Beyond that, there are a lot of things to learn, a lot of acronyms and jargon, a lot of conflicting advice on StackOverflow, and a lot of people spouting really bad advice. There are different types of API (REST, RPC, GraphQL, SOAP), different types of transportation layer, lots of different errors to understand, and a tricky balancing act to make your applications both performant and up-to-date with this data that lives somewhere else. Many people don't think about HTTP timeouts or the effects an accidental 10s hang could have on downstream systems. What happens if the user gets on a subway half way through a transaction… Even something as simple as an unexpected validation rule coming back from the API can leave user stuck in a tween-state, with a blank screen and no way to progress. AGGH!

Some of this would be easier if API developers documenting things extensively, but most of the time you're lucky to get an out-of-date Word document called API Documentation-v2-33-January-18-Final.doc. Sometimes you might have to guess a contract, and that's no easy feat.

Some of these problems are just lessons learned over time, but in the mean time your application can be suffering all sorts of bug reports, user complaints, server issues, and who knows what other sort of production issues. If it's not as dire as that, there are at least other sources of confusion that'll get you spinning your wheels trying to figure out what to do next.

Why Integrate with an API?

Throughout the 2000s, most of us web developers were building frontend and the backend in the same application. The frontend was just whatever HTML the backend decided to spit out. Frontend developers would design the templates (HTML + data tags), and the backend would decide which HTML template to show, and what values to shove into those tags. Go Smarty.

It was a simple time, but there was a lot wrong with it. Sharing data between apps was awful, and the amount of iframe trickery was scary. One time I was working on an integration where a financial services company set up a deal to provide the stocks and shares information for the MSN Money UK homepage. We generated full HTML on our site, (possibly generated from a CMS that was not expecting this sort of thing), they read the whole thing, regex replaced some special tags added just for them, cached it somehow, and shoved their own CSS on it…

It was around this time I started blogging about APIs. 🤔

Thankfully we've mostly escaped that swamp, and the vast majority of companies are building APIs first. On the desktop it's all about single-page JavaScript applications, with Angular/Ember/React using data binding to pull down and push back data from a data source, and mobiles often work very similarly, regardless of if the apps are HTML based or native. These frontend applications cannot do all that much by themselves, so they need to hook up to APIs to do *something*. That something might be as simple as persisting data to the server, enabling communications with other users, charging somebodies credit card, etc.

External APIs

These days most architectures involve at least one external API (an API built by another company), and that trend is only going to continue.

Think about older video games like Rollercoaster Tycoon. They were developed singlehandedly, by one software developer who could do a bunch of things. Over time, the expectations for video games have shot up so much that no one person could ever create the next bestselling video game singlehandedly. Now there are whole teams of people who work on physics engines, some folks figure out how to make realistic hair, and everyone focuses on building their specific modules.

Most smaller game studios will skip building their own physics engines and simply pay the licensing fee to use an existing software, because they don't have the human-power to build their own, and it's probably cheaper. APIs are very much the same sort of idea. Startups don't have time to build their own SMS messaging service, so they just use Twilio. It's fantastic, it only takes a few minutes to get started, and it means that startup can focus on their application, which is going to make the world a better place through drone-based, underwear folding in-the-home.

If something awfully complex pops up like geocoding, most folks will just shove Google Maps API in there, or Mapbox API, or OpenStreet Map API, because generating a standard interface for wildly different datasets around the globe is a mess, and why waste time on that when you've got underpants to fold!

Breaking Down the Monolith

Over the last few years, the rise of the service (or micro service) has meant you'll often be working with more than one API. External APIs might be replaced with internal APIs as your teams grow and ownership becomes more important, and larger applications can start to become an unmanageable non-performant mess. This really makes sense for companies with larger engineering teams, who want to avoid stepping on each others toes with changes they make. The more services at play, the more complex everything gets, but a mature, and educated engineering department with a strong devops culture can eventually learn to manage this.

A lot of teams who are just getting started with building services kinda forget about this integration part, and forget their systems are going to be used by a myriad of others. As companies grow, other developers want to start hooking into these data sources, and often the original developers have moved on. You end up with all these random things talking to other random things and the architectural diagrams (if kept up to date) would look like an octopus orgy.

These services need to talk to each other intelligently, handle various types of error, have realistic timeouts, know when to retry, and most importantly identify themselves, so you don't end up with a stampeding herd; with no idea which client is causing it.

These services also need frontends, or have one mega dashboard that handles the UI for many of them.

One of these scenarios is where you probably come in, so with the why covered, let's look at how.

Chapter Two

Transportation Layers

To send anything to or fetch anything from an API, we're going to need a "transport layer" to do it. Transport layers are conceptually about sending things about over the network, and there's a bunch of different levels with a bunch of different protocols that all sit on top of each other. Whole books are written about transportation, network protocols, and all the fun that goes with it, but we're going to skim past some of the stuff that's not relevant. If you don't care about inside the sausage, just skip this chapter.

Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) provide two means of sending things around. TCP ensures a packet of data was received, and UDP is a fire and forget approach that's a little quicker due to its blasé approach.

One implementation of TCP is TCP/IP, also known as the Internet Protocol Suite. This is a whole bunch of protocols that facilitate the modern Internet as we know it, so yay for that.

TCP/IP facilitates the Hypertext Transportation Protocol (HTTP), and *that* is what we care about. HTTP is an absolutely fantastic resource for API interaction, as it covers so many fantastic features, that would be just awful to implement from scratch.

Ever growing and improving, there have been three versions over the years. They're all huge improvements, that reflect the evolving requirements of the web, and the needs of those building software for it. HTTP/1 was pretty standard and expected early-web stuff: a static HTML page to load of a few links to Word documents or something, HTTP/1.1 added a bunch of amazing things to make multiple connections work far quicker (all that CSS/JS/AJAX/etc), and HTTP/2 changes pretty much everything, turns conventional wisdom on its head, and gives us powerful functionality like multiplexing, amongst many other things.

Learning about HTTP and utilizing its features to make fantastic applications can be tough. HTTP is like snowboarding: you can learn the basics in a very short amount of time, get cocky, and go smashing off at high speed, until you inevitably break three ribs… At 12 years old, HTTP seemed rather clear, and I thought I had a grasp on how it worked. Almost two decades later I'm still learning about features, headers, and options, that were either added back then, or by more recent RFCs.

These are some of the learnings the book will be covering, as HTTP gets a lot of very undue flak for being "bulky" or "slow", which mostly come down to us using it incredibly poorly.

This is not just a book about HTTP however. There are a few other protocols that are commonly used for building more "real-time" APIs: WebSockets which sit on top of a single TCP connection, and Advanced Message Queuing Protocol (AMQP). Often these two technologies will be paired with an HTTP API, providing event-driven interactions for an API that is otherwise mostly just sitting there waiting for you to ask it (or tell) it stuff.

Whilst there are technically others, depending on the sort of work you are doing, HTTP APIs are probably going to be what you bump into 80-90% of the time, with WebSockets or AMQP popping up now and then. As such, a hefty chunk of the book will be covering HTTP APIs.

HTTP Basics

To avoid getting all academic and theoretical, learning by doing might be the way to go with understanding HTTP.

Making your first HTTP query is easy, you just need a HTTP client. Most programming languages have one built in, but before we get into all of that, we can just use the command line. If you have curl available you can do this, otherwise a quick `brew install curl` will help.

```
curl -X GET http://example.com
<!doctype html>
<html>
<head>
    <title>Example Domain</title>
    <meta charset="utf-8" />
    ...
```

Right there, we made a request using the GET method (of course -x means "method"), to the URL http://example.com, and we got a HTML response. If this was some JSON, and if we were doing it in a programming language, we would be off to a reasonable start!

Most programming languages come with a HTTP client by default, but often they are hard to work with and have an interface uglier than a bulldog chewing a wasp. For example, the backend language PHP has a curl extension, which is quite time consuming to do anything with, and therefore most folks use Guzzle (which actually wraps curl).

A popular HTTP client for JavaScript is Axios, which provides promise-based HTTP interactions. Their README example shows the most basic of API interactions, and it looks a bit like this:

```
axios.get('https://api.example.org/companies/12345')
  .then(function (response) {
    console.log(response);
  })
  .catch(function (error) {
    console.log(error);
  });
```

Quite simply, we are trying to GET the information for whichever company has the identifier 12345. If it works, the then() is called, otherwise catch() is called.

Identifiers (IDs) are used to reference companies in a unique way. Human readable names might not work as two companies in difference industries might trade under the same name, so an ID helps keep them unique.

Asking this question of the API is known as the "HTTP Request", and the success or fail will be determined by the APIs answer, the "HTTP Response".

The HTTP client is responsible for providing a useful interface between your programming language, and whatever low-level networking library is in place to make the actual requests. At the network level, HTTP is a plain-text protocol that has a request message and response message. A request has method like GET, POST, PUT, PATCH, DELETE, OPTIONS (and a few more), which indicate the sort of request you are making. There's then a host, and a URI, and you need to specify which HTTP version you're talking about.

The Axios example above would produce a HTTP request like this:

```
GET /companies/12345 HTTP/1.1
Host: api.example.org
```

The response may well be something successful, and we might get JSON back to play with:

```
HTTP/1.1 200 OK
Date: Mon, 18 Dec 2018 12:30:00 GMT
Content-Type: application/json
Connection: Closed

{"id":12345,"name":"Patagonia","description":"Expensive outdoor
     clothing saving the world through taking a decent moral stance”}
```

It could also be a total failure:

```
HTTP/1.1 401 Unauthorized
Date: Mon, 18 Dec 2018 12:30:00 GMT
Content-Type: application/json
Connection: Closed

{"error":"Get off my land!"}
```

Notice here the 401 Unauthorized message on the first line. Thanks to the conventions set up, any HTTP client is going to know that's an error, and Axios will trigger the catch() block for you. Not all HTTP clients are that well set up by default, and some ask their users to pay a bit more attention to detail. Instead of being spoon fed the success or fail, you'll need to programatically be aware of the "status code" in the HTTP response.

HTTP Methods

We have made a GET request, because they are nice an easy. There are plenty more methods out there, which all have their own specific meaning and uses. Some APIs will use more than others, but it's important to learn what is what.

- GET - Fetching things, shouldn't cause anything to change on the API other than maybe some metrics
- POST - Most commonly used for "creating" things in a collection, but many APIs will use this for anything that changes state
- PUT - An idempotent method that overrides whatever was there with whatever you send. Maybe nothing was there before, meaning PUT actually facilitates create *and* replace
- PATCH - Update just a few fields instead of everything, to avoid two clients race-condition clobbering data sent from the other
- DELETE - Guess
- HEAD - Like a GET, but only return the headers

We'll ignore the others for now.

If the API you are talking to calls itself a "REST API", it's likely to use all of those methods. If it calls itself "RPC", it might only use GET and POST. If it's GraphQL, it's all going to happen over POST.

Confused? I know. More on all of that later.

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 `507` – 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.

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, but keep in mind that a 202 Accepted 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.

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. Some folks use a `Location` header to point to the content, so if you see a 3xx check for that.

4xx is all about client errors

With these status codes, APIs indicate that the client has done something invalid and needs to fix the request before resending it.

5xx is all about service errors

With these status codes, the API is indicating that something went wrong in 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.

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 - 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 APIs 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.

HTTP Headers

Headers have been mentioned a few times, and they're another great feature for HTTP.

HTTP headers are meta-data about the request or response, and control all sorts of things, like the Content Type (is this JSON or XML), or cache controls (how long should this data be cached for), etc.

For example, some APIs accept "form data", as well as JSON. It's important to understand which is being sent by default, and which the API wants.

Sending form data might look like this:

```
var querystring = require('querystring');
var instance = axios.create({
  baseURL: 'https://api.example.com/',
  headers: {'Content-Type': 'application/x-www-form-urlencoded'}
});
instance.post('/hello', querystring.stringify({someParam: 'Some
      value'}));
```

Sending the same data as JSON might look a little more like this:

```
var instance = axios.create({
  baseURL: 'https://api.example.com/',
  headers: {'Content-Type': 'application/json'}
});
instance.post('/hello', JSON.stringify({someParam: 'Some value'}));
```

Notice the only real difference here is that we have changed the Content-Type, and changed how we generate the string. HTTP APIs are very flexible in this way.

Some APIs will let you request data in a format relevant to your needs: CSV, YAML, or other more complex binary formats, which we'll get into later. You simply need to supply the media type, and if the API has it you'll get it back.

```
var instance = axios.create({
  baseURL: 'https://api.example.com/',
  headers: {'Accept': 'application/csv'}
});
instance.get('/reports/123');
```

If you try requesting a media type that the API has not defined, you will probably end up with a 406 Not Acceptable response.

Headers can do a whole lot more than just switch content types, but we will look at relevant headers in relevant content as we go.

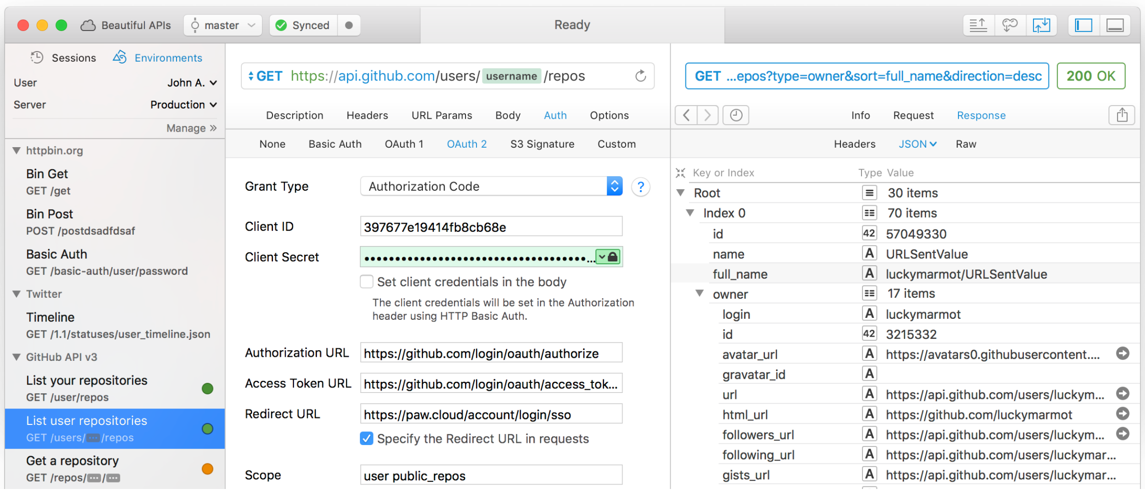
Playing around with HTTP

Mucking about with a HTTP client in a console, like `node -i` or `ruby console` will get you quite a long way, and you can get a feel for how your programing language HTTP client of choice is going to work. It can be a little fiddly working that way to get started though, and when often you will find that a HTTP client with a GUI will be more helpful.

There are plenty of HTTP GUI applications out there, but the three biggest are:

- Postman
- Paw
- Insomnia

These GUIs are very impressive, and can help you build complex requests without having to try to write a bunch of JSON by hand every time. Often attempting to integrate with a new API via one of these tools first is a good way to make sure things work as you expect, then it can be integrated in code.



User	John A.
Server	Production

httpbin.org
- Bin Get — GET /get
- Bin Post — POST /postdsadfdsaf
- Basic Auth — GET /basic-auth/user/password

Twitter
- Timeline — GET /1.1/statuses/user_timeline.json

GitHub API v3
- List your repositories — GET /user/repos
- List user repositories — GET /users/[redacted]/repos
- Get a repository — GET /repos/[redacted]/[redacted]

GET https://api.github.com/users/ username /repos

Grant Type	Authorization Code
Client ID	397677e19414fb8cb68e
Client Secret	••••••••••••••••••••••••••••••••••••
	☐ Set client credentials in the body The client credentials will be set in the Authorization header using HTTP Basic Auth.
Authorization URL	https://github.com/login/oauth/authorize
Access Token URL	https://github.com/login/oauth/access_tok...
Redirect URL	https://paw.cloud/account/login/sso
	☑ Specify the Redirect URL in requests
Scope	user public_repos

GET ...epos?type=owner&sort=full_name&direction=desc 200 OK

Key or Index	Type	Value
Root	≡	30 items
Index 0	≡	70 items
id	42	57049330
name	A	URLSentValue
full_name	A	luckymarmot/URLSentValue
owner	≡	17 items
login	A	luckymarmot
id	42	3215332
avatar_url	A	https://avatars0.githubusercontent....
gravatar_id	A	
url	A	https://api.github.com/users/luckym...
html_url	A	https://github.com/luckymarmot
followers_url	A	https://api.github.com/users/luckym...
following_url	A	https://api.github.com/users/luckymar...
gists_url	A	https://api.github.com/users/luckymar...

Screenshot of Paw from paw.rest

Paw is probably the most feature complete, but Postman has been around the longest, and as such has a huge community of shared collections. Download Postman, grab one of those collections, and play around with it.

See the headers, response codes, and data being returned. Twilio is a fantastic API and has a great collection, so maybe start there.

Chapter Three

SDKs

Assuming you've been tasked with integrating with an API, the first thing to do is look for a Software Development Kit (SDK). An SDK is basically a fancy term for some code that has been written to make integration with a third party API easier in a specific programming language. Often they are released officially by the company in charge of the API, sometimes they are written by third-parties and released as open-source.

The use of an SDK defers a lot of the responsibility to the API developers, as most of the HTTP-level code you would need to write to interact with their API is done by them. Theoretically, they should seamlessly take care of authentication, data formats, HTTP errors will be turned into exceptions, validation will be returned in a useful format, etc. If that is the case for the specific API you are attempting to integrate with, then you might be in luck, and able to skip a lot of this book.

For example, we have not covered how Authentication works yet, and with a good SDK you would not need to know, or care. You simply grab the authentication token and put it into the code:

```
var stripe = require("stripe")(
  "sk_test_BQokikJOvBiI2HlWgH4olfQ2"
);
```

A good SDK will not only obfuscate how things work at a transport layer, but a bad one might just get in the way and make it harder to work with than just approaching the API directly. Much like any library/package/plugin, if it cannot do everything you require, and has little room for extension, sometimes it is easier to skip it and do things yourself.

Ensuring SDK Quality

To use an SDK for the API you are trying to integrate, there needs to be an SDK for the programming language you are using, and it needs to be well built. If there is no official SDK, you can search around and to find an unofficial one.

If you find one, often that can still suffer from not being particularly good, especially as abandonment is a big problem in open-source. If the SDK is outdated, it's potentially useless, and might be missing methods for the resources and endpoints your application might require.

To ensure the SDK is of good quality, check for a few things.

1) Does it have unit tests?
2) Does it have documentation?
3) Is it on GitHub?
4) Does it have loads of outstanding issues that are months old?
5) Does it have outstanding pull requests with no feedback?

For examples of great SDKs, again, check out Twilio, who do a fantastic job at building SDKs. They have libraries/packages built for C#, Java, Node, PHP, Python, Ruby, and JavaScript!

Another example is Stripe, who document each SDK and its usage separately, with code examples that show how to carry out certain tasks in those languages.

```
var stripe = require("stripe")(
  "sk_test_BQokikJOvBiI2HlWgH4olfQ2"
);
stripe.disputes.update(
  "dp_1AfYgX2eZvKYlo2CXQAAqDv7",
  {
    evidence: {
      customer_name: 'Elijah Williams',
      product_description: 'Comfortable cotton t-shirt',
      shipping_documentation: 'file_1BC7qO2eZvKYlo2CREONzM9U'
    }
  },
  function(err, dispute) {
    // asynchronously called
  }
);
```

Another than the evidence tag taking a JavaScript object that matches the JSON payload to be sent, there is not much HTTP about this cod, and that’s fantastic if you want to just eat the sausage and not learn how it is made.

Sadly, as mentioned, there is often no SDK, or no good ones, so lets keep trucking and learning about making these calls yourself.

Chapter Four

RPC, REST and GraphQL

Before getting too much further, understanding the differences between these difference concepts is going to be vital. Every API in the world is following some sort of paradigm, whether it knows it or not. They will fall under RPC, REST, or a "query language."

Even if you are confident you understand the difference, do yourself a favor and read them anyway. About 99% of people get this wrong, so you can get be in the top 1% with a quick read.

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 HTTP or 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, we would do the same by defining a function, and later we'd call 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, RPC is just a case of sending up and down data fields, which is fine, but one downside is that the client is entirely in charge of pretty much everything. The client must know which methods (endpoints) to hit at what time, in order to construct its own workflow out of otherwise incredibly naive and non-descriptive endpoints.

RPC is merely a concept, but that concept has a lot of specifications, all of which have concrete implementations:

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

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 Web Services Description Language (WSDL) 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… It is also worth pointing out if your "distance" field is metric or imperial, to avoid bad math crashing your billion dollar satellite into Mars.

A modern RPC implementation is gRPC, which can easily be considered modern (and drastically better) SOAP. It uses a data format called Protocol Buffers (or Protobuff for short), which requires a schema as well as the data instance, much like the WSDL in SOAP. GRPC focuses on making single interactions as quick as possible, thanks to HTTP/2, and the fact that Protobuff packs down smaller than JSON. Fear not, JSON is available for as an option too.

Representational State Transfer (REST)

REST is a network paradigm described 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.

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 hypermedia controls (HATEOAS).

Hypermedia is fundamental to REST, and is essentially just the concept of providing "next available actions", which could be related data, or in the example of an "Invoice" resource, it might be a link to a "Payment Attempts" collection so that the client can attempt paying the invoice. 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": "2017-06-16 09:05:00Z",
      "status": "published"
    }
  },
  "links": {
    "pay": "https://api.acme.com/invoices/093b941d/payment_attempts"
  }
}
```

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

Client: Hi, I would like to speak to Dr Watson, is he there? If so, what time can I see him and what location.

RPC: No. *click*

Calls back

Client: I checked his calendar, 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. 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. Of course REST would needed to know to look for `"status: unavailable"` and follow the `other_doctors` link to `https://api.doc.io/available_doctors?available_at=2017-01-01 03:00:00 GMT`, but that is far less of a burden on the client than forcing it to check the calendar itself, ensure it's getting timezones right when checking for availability for that time, 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.

Other than hypermedia (the most powerful yet most ignored part) there are a few other requirements for a system to be a REST API:

- 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

The goal of these constraints is to make the REST architecture help APIs last for decades, which is almost impossible to do without these concepts.

REST also does not require the use of schema metadata, which many API developers hated in SOAP. For a long time nobody was building REST APIs with schema, but these days it is far more common thanks to JSON Schema (inspired by XML Schema but not functionally identical). This optional layer is something we will talk about a lot, as it can provide some incredibly functionality like client-side validation, that was defined by the backend!

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, so 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 jokingly called RESTish by people aware of the difference.

On the flip side, a REST API can be used in an RPC fashion if you as the client developer chose to ignore the links. It is not advisable of course, but it is possible.

A huge source of confusion for people with REST is that they do not understand "all the extra faffing about" such as hypermedia controls and HTTP caching. They do not see the point, and many 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. I like to think of REST as a state machine operating over a network.

REST can theoretically work in any transportation protocol that provides it the ability to fulfill the constraints, but no transportation protocol other than HTTP quite has the functionality to do so. To fit REST into AMQP, you would need to define hypermedia controls somehow (potentially an array of messages you could call next), a standard for declaring cacheability of the AMQP messages, etc., and create a lot of tooling that does not exist to implement that cacheing. Basically REST is too powerful for other existing transportation protocols, so it is generally implemented in HTTP.

REST has no specification which is what leads to some of this confusion, nor does REST have any concrete implementations. That said, there are two large popular specifications which provide a whole lot of standardization for REST APIs that chose to use them:

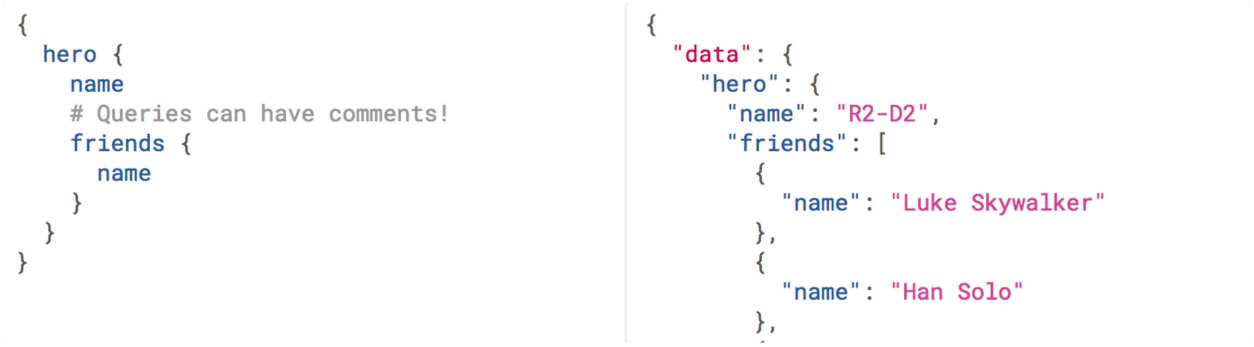
- OData
- JSON-API

If the API advertises itself as using these, you are off to a good start. These are more than just standardized shapes for the JSON, they have guides on pagination, metadata, manipulating relationships between existing items, etc. Find an OData client or a JSON-API client in programming language to save yourself some work. Otherwise go at it yourself with a plain-old HTTP client and you should be ok with a little bit of elbow grease.

GraphQL

Listing GraphQL as a direct comparison to these other two concepts is a little odd, as GraphQL is essentially RPC, with a lot of good ideas from the REST/HTTP community tacked in. Still, it is one of the fastest growing API ecosystems out there, mostly due to some of the confusion outlined above.

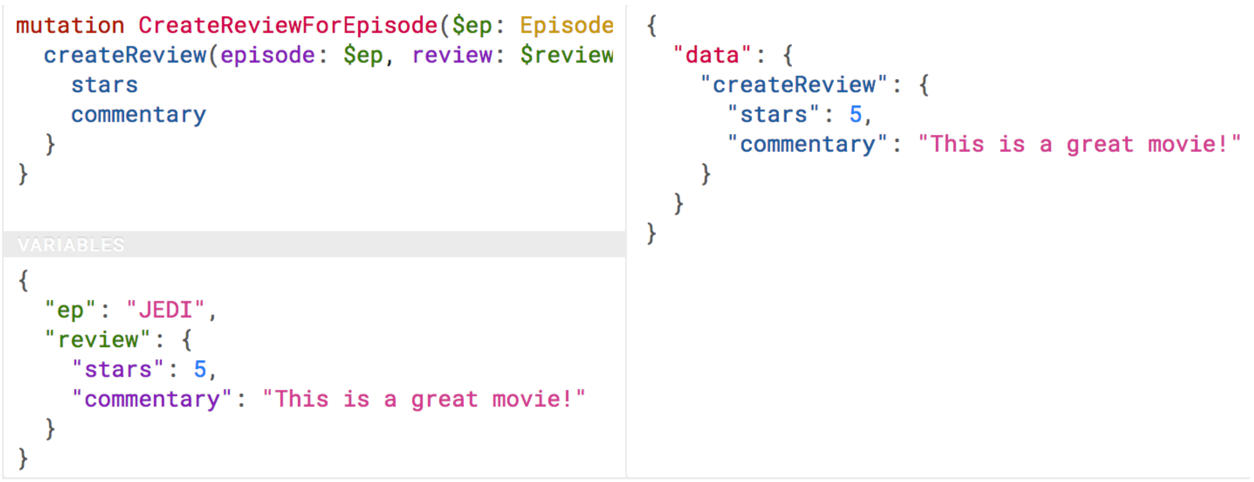
GraphQL is basically RPC with a default procedure providing a query language, a little like SQL - if that is something you are familiar with. You ask for specific resources and specific fields, and it will return that data in the response.



```
{
  hero {
    name
    # Queries can have comments!
    friends {
      name
    }
  }
}
```

```
{
  "data": {
    "hero": {
      "name": "R2-D2",
      "friends": [
        {
          "name": "Luke Skywalker"
        },
        {
          "name": "Han Solo"
        },
```

It has Mutations for creates, updates, deletes, etc. and again they are exactly RPC.



```
mutation CreateReviewForEpisode($ep: Episode
  createReview(episode: $ep, review: $review
    stars
    commentary
  }
}
```

```
{
  "ep": "JEDI",
  "review": {
    "stars": 5,
    "commentary": "This is a great movie!"
  }
}
```

```
{
  "data": {
    "createReview": {
      "stars": 5,
      "commentary": "This is a great movie!"
    }
  }
}
```

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 RESTish 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. Sadly a few of the most powerful REST concepts were completely ignored.

The backstory to GraphQL, is an interesting one. Facebook has experimented with various different approaches to sharing all their data between apps over the years; remember FQL? Executing SQL-like syntax over a GET endpoint was a bit odd.

```
GET /fql?
     q=SELECT%2Buid2%2BFROM%2Bfriend%2BWHERE%2Buid1%3Dme()&access_toke
     n=...
```

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, GraphQL was created as a middle-ground between endpoint-based APIs and FQL, the latter being an approach most API developers would never consider—or want.

In the end, they developed this RPC-style query language system, to ignore most of the transportation layer, meaning they had full control over the concepts. Endpoints are gone, resources declaring their own cacheability is gone, the concept of the uniform interface (as REST defines it) is obliterated, which has the supposed benefit of making GraphQL so incredibly simple it could fit into AMQP or any other transportation protocol.

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, but thankfully was solved quite nicely in HTTP/2.

Sadly despite being a rather nice package, GraphQL through Mutations force the responsibility onto clients to know everything. The only difference between RPC and GraphQL is the ability to request which fields you get on a successful mutation.

Client: Hi, I would like to see Dr Watson, and if he is there, what time can I see him and what location?

RPC: No. *click*

Calls back

Client: I checked his calendar, 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? If so, what time can I see him and what location.

RPC: Yes. 2pm at the Soho Office

That is a handy feature as it saves a tiny bit of network bandwidth, but again the client was forced to figure out its own workflow instead of being presented with a plethora of potential next actions. This sort of thing makes GraphQL very nice for fetching custom reports, gathering statistics, etc, somewhat passable at CRUD, but not particularly good for much else. Despite that it is regularly used for APIs that would be better suited as REST, so expect to bump into it now and then.

When you do bump into GraphQL, you will not be able to use your regular HTTP clients, and most of things you do with any other HTTP API will not work here, with GraphQL instead requiring its own special client tools.

Skinny/Fat Clients

There is one more piece of theory we need to get our heads around to really understand the differences between these API paradigms/concepts. In couples dancing there is usually one person leading, and another person following. The leader will be in charge of signaling when moves should be made, and the follower follows their hints. In REST the idea is that the server leads. In all forms of RPC (including GraphQL), the client leads.

We already spoke about REST “normalizing state”, which is the concept of removing the guesswork and decision making on things that the server is authoritative on. The client application in a REST API becomes a thin UI, which handles user interactions, forms, validations, rendering, animations, but very little business logic. That is the theory anyway. A lot of RESTish/RPC/GraphQL APIs force the clients to figure out things they should not be trying to figure out, as their main interest is trying to send as little data as possible.

Inferring State from Arbitrary Fields

At the day job, an upstream API decided originally suggested that clients should check for `location_uuid == nil` to infer the membership type as an “Anywhere member” (a member who can access any of our coworking specs. One day they changed the logic, suggesting clients check for location_uuid == “some-special-uuid”. This was not communicated to the developers in charge of various systems as nobody remembered which of the 50+ systems were making the check. One outcome of that was to confuse the permissions API, which started denying things it should not have been denying. Who even knows what else broke and what else was fixed by various teams working on various systems. Applications need to be investigated, fixed, and updated.

Inferring meaning from arbitrary fields can lead to production issues, angry users, and wasted developer time debugging bizarre things. The preferred approach in many of these situations would be to have the current state normalized on the server with a simple field like "type" which could have a few options like "anywhere", "other", and the API can figure out when to display that. If the API you are working with asks you to infer state of some random fields, politely let them know that you don't want their laziness to break your client application. Request they provide their data in a useful generic fashion, so the clients don't need to guess everything.

Available Actions

Working out the current state is bad enough, but working out what to do "next" when you have a record can be really tough. For example, a ReactJS application at work had loads of awful helper functions to work out which "actions" to show next to a list of users in a company (through the joining resource known as seats).

```
IsNotPrimaryOfAnotherCompany(user, company_uuid) {
  return _(user.data.result.seats).select(s => {
    return s.company.uuid != company_uuid &&
    s.status == "active" &&
    s.kind == "Licensee" &&
    s.company.status == "active";
  }).isEmpty();
},
```

The users and company resources have status, and seats has a kind, so this is somewhat ok, but the client is still working really hard for this seemingly complex, but potentially simple functionality. Business logic states that users should not be primary members of more than one company at the time, so a rule was made in the user management interface to hide the "Make Primary Member" option if they were. That seems reasonable enough, but when the iOS native codebase forgot to add that `s.status == "active";` condition, the two devices were showing different actions. We fixed it on iOS, and a few months later another rule was added to the JavaScript code, causing an unexpected difference that regression tests did not catch…

The "skinny server, fat client" approach is forcing the client to download all the seats, and the user information, and the company, just to spin through all the records and establish if the user can or cannot see a “Make Primary Member” button…

The most basic thing to do for RPC would be to make a “can_be_promoted” boolean field, which would contain all of the logic internally. The API is a far better at handling this, it’ll do it more performantly, and it will be able to change its own ruling internally whilst still communicating the same meaning. Try requesting that from the API developers, and who knows you might get lucky. The REST approach would be to normalize hypermedia controls (HATEOAS), to also normalize that state up into the server. If the API was using a format like Siren, you might see a link nicknamed ”promote” for employees, and ”demote” for primary members. The client application could then easily be trained to recognize the promote/demote links and show the appropriate buttons. Siren would also handily show the metadata (URI, method, fields, etc.) required for the client to make the change.

You Cannot Always Be a Skinny Client

Neither approach is definitely good or bad in their entirety, it’s simply the case that skinny clients have fewer changes of misrepresenting the state if they are given the answers explicitly. Whenever clients try to copy the business logic rules, the chance of different clients inferring things differently increases. Different codebases will guess things differently, some will notice changes and others won’t, and the need to deploy multiple clients in unison becomes a lot more common. It also generally leads to having a broken user experience across different devices.

Letting the server take control of such things can be a little scary, and it also depends on the relationship the client has with the API, the team developing it, and the intent of the client application. If the API is a third-party, letting them lead could cause some unexpected results. If the API was designed without the knowledge of the clients requirements, the API might not do a very good job of leading.

But What About...

There is a lot more to it than that, and the terms will come up plenty over the course of the book. Some chapters will only apply to one of these paradigms, but most chapters will cover general concepts that apply to one or more.

Chapter Five

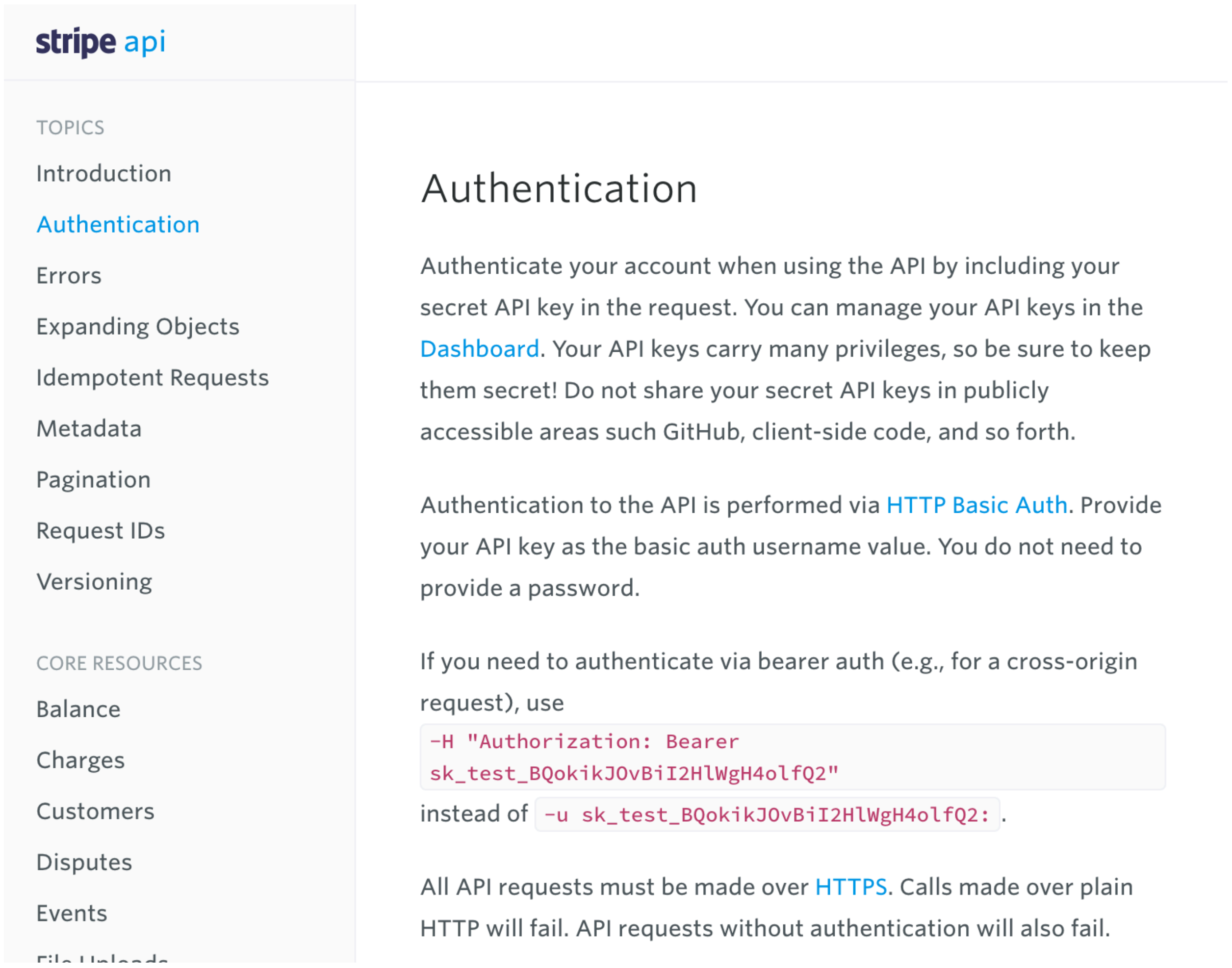
Authentication

Generally the idea of Authentication is to require that clients provide some sort of unique information, with the goal of proving the request is coming from a user or application that actually exists, and is allowed to make that request. The unique information is usually referred to as "credentials" and these will be given to you by developers on the team in charge of the API. If the API belongs to another company you can often sign up for "API Access" or "API Keys" on their website somewhere.

Authentication primarily allows APIs to restrict access to various endpoints, but it also offers APIs the chance to track users, give endpoints user context (GET /feed should only return the _current users_ feed), filter data, or even throttle and deactivate accounts that overdo it.

Some are completely public with no keys at all. Completely open APIs with no keys seem to be getting rarer over time, as APIs get more important, do far more, and are increasingly dynamic.

To get the specifics for a particular API, their documentation is a fantastic place to start. Usually API documentation will contain an "Authentication" section in their docs, and this will usually be towards the start.



Authentication

Authenticate your account when using the API by including your secret API key in the request. You can manage your API keys in the Dashboard. Your API keys carry many privileges, so be sure to keep them secret! Do not share your secret API keys in publicly accessible areas such GitHub, client-side code, and so forth.

Authentication to the API is performed via HTTP Basic Auth. Provide your API key as the basic auth username value. You do not need to provide a password.

If you need to authenticate via bearer auth (e.g., for a cross-origin request), use `-H "Authorization: Bearer sk_test_BQokikJOvBiI2HlWgH4olfQ2"` instead of `-u sk_test_BQokikJOvBiI2HlWgH4olfQ2:`.

All API requests must be made over HTTPS. Calls made over plain HTTP will fail. API requests without authentication will also fail.

Screenshot from Stripe: Authentication

Here Stripe are making it quite clear where you can find your API keys, and talk about two approaches to sending them.

What confuses many folks when faced with HTTP Authentication, is that most of the time the authentication happens on every single HTTP request.

That is right: Every. Single. Request.

This sounds bizarre at first, as we are so often used to the concept of "Users logs in, then they have a session and can do logged in stuff until they log out or the session expires" being the one paradigm for how users login. That paradigm certainly exists for some APIs, but its rare, and for plenty of good reasons.

To cut a long story short, servers are happier when they do not need to remember who a specific user is. For example, if the API actually has multiple application servers all running behind load balancers, a client could hit a different server with each request they make. If cookies were used, then the user would be logged out every time the load balancer routed them to a different application server, unless it went to the trouble of maintaining "sticky sessions" to keep the user hitting the same server. That is all well and good until that application server is replaced by another instance on deployment… should users really be logged out due to a deployment?

For this, and many other reasons (like not having to worry about logouts), HTTP Authentication tokens are passed on every single request made.

Stripe have already mentioned HTTP Basic Auth and Bearer, which are two very similar approaches to authentication in HTTP. There are quite a damn lot of authentication strategies you could bump into, and whilst you do not need to know them all, you will probably want to read the appropriate section when an API you are looking at uses it:

- HTTP Basic
- HTTP Digest Auth
- Bearer
- OAuth 1.0a
- OAuth 2.0
- JWTs

TODO Explain how to work with them all. Ugh it's gonna take a while.

Chapter Six

Common Pitfalls

Getting the happy path coded up is super easy. Usually the happy path is something like "I expect a quick response, with valid JSON, in the format my code expects it" often feels about as naive as thinking "I can just ride my bike to work in NYC without two taxi drivers, a truck, a whole family of tourists, cops on horses, a dog walker, and a raccoon, all trying to kill me."

Connection Failures

Connection failures can happen for all sorts of reasons:

- Mobile user gets on the subway
- Laptop user switches wifi networks which has a captive portal login
- API deployment went bad
- AWS is down again
- Cockroaches made a nest in the server the API is hosted on

Expect the unexpected. Make your HTTP client throw exceptions whenever possible, so you can catch different scenarios easily. Failed to connect? Try it again, and if it fails a few times maybe show something to the user explaining that their Internet is down.

Make sure no single part of any client application *requires* a connection to leave that state. Often I have seen client applications submit a form, hide the form they just submitted, fail to make the connection, and as they were expecting a positive or negative JSON response in a specific structure, in order to dictate showing the form again or progressing, they end up with a blank screen.

Timeouts are also a concern, but more on those later.

Rate Limiting

Another common situation to run into is rate limiting: the API telling you to calm down a bit, and slow down how many requests are being made in a certain timeframe. The most basic rate limiting strategy is often "clients can only send X requests per second."

Many APIs implement rate limiting to ensure relative stability when unexpected things happen. If for some reason one client causes a spike in traffic, the API has to continue running smoothly for other users instead of crashing. A misbehaving (or malicious script) could be hogging resources, or the API systems could be struggling and they need to cut down the rate limit for "lower priority" traffic. Sometimes it is just because the company providing the API has grown beyond their wildest dreams, and want to charge money for increasing the rate limit for high capacity users.

Often the rate limit will be associated to an "API key" or "access token" (see Chapter Five - Authentication), and our friends over at Nordic APIs very nicely explain some other rate limiting strategies:

> Server rate limits are a good choice as well. By setting rates on specific servers, developers can make sure that common use servers, such as those used to log in, can handle a lot more requests than specialized or seldom used servers, such as data conversion devices.
>
> Finally, the API developer can implement regional data limits, which limit calls by region. This is especially useful when implementing behavior-based limiting; for instance, a developer would expect the number of requests during midnight in North America to be lower than the baseline daytime rate, and any behavior contrary to this without a good reason would suggest questionable activity. By limiting the region for a period of time, this can be prevented. — **Nordic APIs**

All fair reasons, but for the client it can be a little pesky.

Throttling Your API Calls

There are a lot of ways to go about throttling your API calls, and it very much depends on where the calls are being made from. One of the hardest things to limit are API calls to a third party being made directly to the client. For example, if your iOS/web/etc clients are making Google Map API calls directly from the application, there is very little you can do to throttle that. You're just gonna have to pay for the appropriate usage tier for how many users you have.

Other setups can be a little easier. If the rate limited API is being spoken to via some sort of backend process, and you control how many of those processes there are, you can limit often that function is called in the backend code.

For example, if you are hitting an API that allows only 20 requests per second, you could have 1 process that allows 20 requests per second to pass through. If this process is handling things synchronously that might not quite work out, and you might need to have something like 4 processes handling 5 requests per second each, but you get the idea.

If this process was being implemented in NodeJS, you could use Bottleneck.

```
const Bottleneck = require("bottleneck");

// Never more than 5 requests running at a time.
// Wait at least 1000ms between each request.
const limiter = new Bottleneck({
  maxConcurrent: 5,
  minTime: 1000
});

const fetchPokemon = id => {
  return pokedex.getPokemon(id);
};

limiter.schedule(fetchPokemon, id).then(result => {
  /* ... */
})
```

Ruby users who are already using tools like Sidekiq can add plugins like Sidekiq::Throttled, or pay for Sidekiq Enterprise, to get rate limiting functionality. Worth every penny in my books.

Every language will have some sort of throttling, job queue limiting, etc. tooling, but you will need to go a step further. Doing your best to avoid hitting rate limits is a good start, but nothing is perfect, and the API might lower its limits for some reason.

Am I Being Rate Limited?

The appropriate HTTP status code for rate limiting has been argued over about as much as tabs vs spaces, but there is a clear winner now; RFC 6585 defines it as 429, so APIs should be using 429.

Twitter's API existed for a few years before this standard, and they chose "420 - Enhance Your Calm". They've dropped this and moved over to 429, but some others copied them at the time, and might not have updated since. You cannot rule out bumping into a copycat API, still using that outdated unofficial status.

Google also got a little "creative" with their status code utilization. For a long time were using 403 for their rate limiting, but I have no idea if they are still doing that. GitHub v3 (a RESTful API that was replaced with a GraphQL, but is still floating around) *is* still using 403:

```
HTTP/1.1 403 Forbidden
X-RateLimit-Limit: 60
X-RateLimit-Remaining: 0
X-RateLimit-Reset: 1377013266
{
   "message": "API rate limit exceeded for xxx.xxx.xxx.xxx. (But
      here's the good news: Authenticated requests get a higher rate
      limit. Check out the documentation for more details.)",
   "documentation_url": "https://developer.github.com/v3/#rate-
     limiting"
}
```

Getting a 429 (or a 420) is a clear indication that a rate limit has been hit, and a 403 combined with an error code, or maybe some HTTP headers can also be a thing to check for. Either way, when you're sure it's a rate limit error, you can move onto the next step: figuring out how long to wait before trying again.

Proprietary Headers

Github here are using some proprietary headers, all beginning with `X-RateLimit-`. These are not at all standard (you can tell by the `X-`), and could be very different from whatever API you are working with.

Successful requests with Github here will show how many requests are remaining, so maybe keep an eye on those and try to avoid making requests if the remaining amount on the last response was 0.

```
curl -i https://api.github.com/users/octocat
HTTP/1.1 200 OK
X-RateLimit-Limit: 60
X-RateLimit-Remaining: 56
X-RateLimit-Reset: 1372700873
```

You can use a shared key (maybe in Redis or similar) to track that, and have it expire on the reset provided in UTC time in `X-RateLimit-Reset`.

Retry-After

According to the RFCs for HTTP/1.1 (the obsoleted and irrelevant RFC 2616, and the replacement RFC 7230-7235), the header Retry-After is only for 503 server errors, and maybe redirects. Luckily RFC 6584 (the same one which added HTTP status code 429) says it's totally cool for APIs to use `Retry-After` there.

So, instead of potentially infinite proprietary alternatives, you should start to see something like this:

```
HTTP/1.1 429 Too Many Requests
Retry-After: 3600
Content-Type: application/json

{
   "message": "API rate limit exceeded for xxx.xxx.xxx.xxx.",
   "documentation_url": "https://developer.example.com/#rate-limiting"
}
```

An alternative value for Retry-After is an HTTP date:

```
Retry-After: Wed, 21 Oct 2015 07:28:00 GMT
```

Same idea, it just tells the client to wait until then before bothering the API further.

By checking for these errors, you can catch then retry (or re-queue) requests that have failed, or if thats not an option try sleeping for a bit to calm workers down.

Warning: *Make sure your sleep does not block your background processes from processing other jobs. This can happen in languages where sleep sleeps the whole process, and that process is running multiple types job on the same thread. Don't back up your whole system with an overzealous sleep!*

Faraday, a ruby gem I work with often, is now aware of Retry-After. It uses the value to help calculate the interval between retry requests. This can be useful for anyone considering implementing rate limiting detection code, even if you aren't a Ruby fan.

Chapter Seven

Caching

Caching is a huge topic, and there's a lot of different types of caching which makes discussing it tricky. The most basic definition of caching is: avoid wasting time computing the answer to the same questions over and over again.

A backend program will often cache the response to database calls to avoid making them multiple times. It could then also cache the serialized output to avoid needing to convert the model into JSON multiple times, and other stuff can be cached too. In the world of APIs, caching can do a lot more. HTTP calls can be skipped by clients, or memorized by servers to skip the application server having to do get involved to answer a similar request. All of this could happen in the life-cycle of a single request.

For backend applications data stores like Redis and Memcached are the most common caching solutions, but anything can be a cache. The file system (not the most performant) or plain-old SQL can be used to cache data. Frontend applications running in a web browser often use LocalStorage.

Caching and Performance

Often caching is misused by inexperienced developers to try and make slow code look like fast code. This leads other folks to consider caching a crutch that should be avoided whenever possible.

The goal of caching on the client side is not "to make our application faster", but to avoid asking questions if we have an acceptable level of confidence that we already know the answers. Making unnecessary requests, even if they are fast, is of course going to be slower than just not making those requests. This is what caching is about.

Caching in relation to APIs

For the rest of this chapter we're going to skip talking about caching in a general way, and talk about caching as it relates to API requests and responses. As such there are three terms we're going to use.

Application Caching: Software like Memcache, Redis, etc. can be implemented in your application, to cache whatever you want, for however long you want, in whatever data store you want.

Network Caching: Tools like Varnish or Squid (known as "cache proxies") intercept requests that look the same, then return a response early straight out of memory instead of hitting the application server. Doing this allows the application server to spend more time handling other traffic, and can reduce network latency for clients.

Client Caching: The client, browser, app, another service, etc. can treat an API response just like any other CSS, Javascript or Image file. If that resource had an expire time, ETags, or one of a few other cache related headers, the last request could be directly reused, or a conditional request might be triggered to see if the data has changed, instead of grabbing the whole resource every time.

HTTP Caching: A set of conventions written into the HTTP specification, which is a way to collectively refer to both network and client caching when it relates to HTTP. GraphQL for example has it's own set of conventions for supplying network caching, but HTTP caching at its core should theoretically work for any programming language, framework, or paradigm, that respects the specification and doesn't build in its own contradicting conventions on top.

Application Caching

So you're writing a client application, and you like the idea of skipping duplicate requests for a few reasons. Maybe you want to avoid going over the API rate limit, draining the battery of the mobile device, or avoid going over the wire if you don't have to. All good reasons to look into caching responses.

At first you might consider application caching. In Rails, using application caching to wrap an API request usually looks a bit like this:

```
Rails.cache.fetch("users/#{uuid}") do
  UserAPI.find_user(uuid)
end
```

If there is nothing in the cache with the key `user/foo` then it will run the code contained in the "`do`" block, which calls a method on the SDK and goes over the wire to fetch the thing. That is handy and all, but how long does that cache entry last? The answer is forever! Infinity is a long time, so we have to provide a reasonable expire period.

```
Rails.cache.fetch("users/#{uuid}", expires_in: 12.hours.from_now) do
  UserAPI.find_user(uuid)
end
```

That is great and all, but the you - the client - providing "12" is possibly not something you should be doing, as you do not own the data. If this is an RPC API and you are part of the same team as the folks who made it, you might know enough about the data source to be confident in making this call, but if you do not own the data then picking an arbitrary number is a bad idea.

Even if 12 is an appropriate number at the time, cache duration could change for a number of reasons. In the most simple case, this leads to "My email address is showing up differently in multiple places", but beyond that there may be all sorts of business logic potentially involved with how long data should be cached.

I've experienced odd situations where a race condition was causing a resource to fetched immediately after it was created, which would cause it to be cached for 5 hours. An important update would happen a few minutes after creation letting us know what sort of membership the user had. That update was not noticed by the system doing this hard time-based caching for that whole 5 hours. Some folks would set up complex invalidation rules, or say “well that data cannot be cached!” but in reality the data could easily be cached after that initial state. We switched from time-based application caching, to using HTTP caching. This allowed the API to define how long the data should be cached for, and the client just does what it’s told.

```
expiry_time = model.present? ? 15.minutes : 5.hours
expires_in(expiry_time, public: true)
render_json_response :ok, user_memberships if stale?(model)
```

Having the flexibility to let the API tell clients how long to cache things for is really handy, but sounds a lot like magic. Let’s take a look at HTTP caching to see how it all works.

HTTP Caching

HTTP caching is one of the best parts of the HTTP specification. At it most simple it's just telling various actors (like an API client) how long to keep data before they chuck it out, and at its most complex it solves the issue of cache invalidation - one of the toughest problems in programming.

Any "endpoint-based" API can use HTTP caching, so potentially you could run into it quite often, but it's also ignored by a lot of API developers who just do not know it exists.

Basically there are a whole bunch of conventions outlines in RFC 7234: HTTP/1.1 Caching, the most commonly used of which is the Cache-Control HTTP header, which resources can use to define its cacheability. The idea of defining cacheability is one of the main requirements for a REST API, and it refers to a standard way to "control" who can cache the response, what they can do with it, under which conditions, and for how long. If a GET request has a `Cache-Control` header, the client knows it can keep it around for a while.

```
Cache-Control: max-age=120
```

Here the API is telling clients they should keep that information for two minutes, but it can do a whole lot more than that. There is conditional validation, which allow for requests asking "only respond with data if it has changed since my last request", and you can create all sorts of complex instructions.

The main idea is to avoid repeating the same GET requests, because no matter how well optimized the API is at responding, not making a request is quicker than making a request. Storing a local copy of a response allows you to skip network latency entirely, which not only saves the end user from waiting, but can also save money too. Respecting cache control headers provided by an API will make your end users experience feel quicker, cheaper, reduce power usage (important for mobile devices). If the API is leveraging network caching too, then when your client does make requests, there is a chance the responses will be served from the cache server, which is usually quicker than the application server. It is also often sitting "at the edge" if it's a CDN

("Content Delivery Network") style network cache.

There is a lot to learn about HTTP caching, and some of the words used do not mean what you would expect them to mean anything first glance. Google have an amazing guide which is designed at understanding the concepts in general, and the hosting company Fortrabbit have written a brilliant article on HTTP caching for applications in general. To leverage some of their writing, here is an overview of the Cache-Control header from fortrabbit:

> The Cache-Control header controls not just the request and response, but holds instructions for potentially two cache locations: The "local cache" (aka "private cache") and the "shared cache".
>
> The local cache, is a cache on the local disk of the machine running the browser. Your laptop, if you will. Be aware that you don't have "exact control" over that cache. Ultimately, the browser decides whether to follow your "suggestions" or not, which means: don't rely on it. The user might clear all caches whenever the browser is closed and you would not know about it, aside from increased traffic cause those caches invalidate faster then you anticipate.
>
> The shared cache … between the web server and the client. The CDN, in this case. You have full control over the shared cache and should leverage it to the fullest.
>
> OK, let's dive in with some code examples. I'll explain in detail below:

```
Cache-Control: public max-age=3600
Cache-Control: private immutable
Cache-Control: no-cache
Cache-Control: public max-age=3600 s-maxage=7200
Cache-Control: public max-age=3600 proxy-revalidate
```

> That might look a bit confusing, but don't worry, it's not that hard. First you should now that Cache-Control takes three "kinds" of directives: Cachability, expiration and revalidation.
>
> First cachability, which takes care of the cache location, which in includes

whether it should be cached at all. The most important directives are:

- **private:** Means it shall only be cached in the local (private) cache. On your laptop.
- **public:** Means it shall be cached in the shared cache. In the CDN. It can also be cached on the local cache, though.
- **no-cache:** Interestingly this means caching is allowed - just everybody (local cache, shared cache) must revalidate before using the cached value
- **no-store:** Means it shall not be cached. Nowhere. Not ever.

Next up is expiration, which takes care of how long things are cached. The most important directives are:

- **max-age=<seconds>:** Sets the cache validity time. How many seconds shall the cache location keep it? Goes for local and shared cache.
- **s-maxage=<seconds>:** Overrides max-age just for the shared cache. No effect on local cache.

Lastly there is revalidation, which is, more or less, fine control. The most important directives are:

- **immutable:** Means that the document won't change. Ever. Can be cached until the heat death of the universe.
- **must-revalidate:** Means the client (browser) must still check with the proxy (CDN), even while it's cached!
- **proxy-revalidate:** Means that the shared cache (CDN) must check the origin, even while it's cached!

And to put it all together, here is how to read the above code examples in plain English:

1. Cache it both on CDN and laptop for an hour.

2. Don't store in CDN, only on laptop. Once cached (on laptop), no need to ever refresh it.
3. Don't cache it - or do. Just make sure to revalidate always!
4. Cache it for an hour on laptop, but for two hours on the CDN
5. Cache it both on CDN and laptop for an hour. BUT: if a request hits the CDN, although it's cached here for an hour, it still must check with the origin whether the document is still unchanged.

— **Ulrich Kautz,** Fortrabbit.com

Couldn't have put that any better myself! We've spoken a bit here about client caching and network caching, so let's look into both of those concepts in more detail.

Client Caching

Client caching when leveraging the HTTP standard is no different from how caching works for Javascript, images, etc. There are some headers telling the client how long to keep this data around, and after that you can chuck it out entirely, or check to see if it is still valid.

This is how browsers interact with websites: the browser assumes the website is the one in charge of certain things like how long to cache data. Whenever you go to pretty much any website, the server defines various cache-related headers and the browser respects them (unless told to override them via something like a hard refresh).

When we build systems that call other systems, we often skip out this step, and performance can suffer. Hopefully the API you are integrating with has `Cache-Control` headers, if not, you are on your own and have to use the application caching approach we discussed before.

Implementing HTTP Client Caching

At work we built an upstream "Permissions" API, which would talk to a lot of other systems to see if a user should be allowed to complete the action they were attempting to make. One request from the end-user could end up making 5 more HTTP requests to other services which were not always the quickest.

We threw a few `Cache-Control` headers on the different services the Permissions API was calling, like on user profiles, membership information, etc., then enabled HTTP client caching using a middleware for our Ruby HTTP client: faraday-http-cache. This thing took an instance of a Redis client, and no more work was required.

Benchmarking with siege:
```
siege -c 5 --time=5m --content-type "application/json" 'https://
     permissions.example.com/check POST { ...not relevant... }
```

All of a sudden the Permissions API went from this:

```
Transactions:              443 hits
Response time:             3.35 secs
Transaction rate:          1.48 trans/sec
Successful transactions: 443
Failed transactions:       0
Longest transaction:       5.95
Shortest transaction:      0.80
```

… to this:

```
Transactions:              5904 hits
Response time:             0.25 secs
Transaction rate:          19.75 trans/sec
Successful transactions: 5904
Failed transactions:       0
Longest transaction:       1.75
Shortest transaction:      0.12
```

This benchmark is of course somewhat artificial due to requesting the same handfuls of users and their related membership data thousands of times, but repeat requests are down to ~250ms from 3.5s. This is substantial however you spin it.

We also later switched from making these calls synchronously, to asynchronously, which of course saved a buuuuunch of time.

This was done with standard `max-age` based caching, which is often incredibly useful all by itself. These days a lot of people act like their APIs are "big data" and everything must be completely real-time, but in most cases having data be a few minutes out-of-date is fine. Basic profile data for a company could absolutely take a few minutes to update, as they're probably not changing their Opening Hours or name very often. Featured items on an e-commerce store is also not likely to change on the regular. I used to work for a financial company which build stocks and shares monitoring systems, and they'd cache most

An API developer could set an hour long max-age for these things, then clients would only need to make the call to the API once an hour.

For information that is more subject to change, a max-age might still be appropriate, it would just be much shorter. This has a few benefits, like making sure browser-based application users can hit the Back button without replicating every single request again, or improving the speed of a backend-based data import script which has a HTTP request written into a loop. Respecting a 10 second cache is still going to cut down load on the server, and speed things up for the client in many cases.

Conditional Validation

Caching based entirely on time is not always the most helpful, but that does not mean client caching should be thrown out. Basic time-based caching will help a client skip making requests entirely, but conditional requests can be made which are much quicker than standard requests.

A conditional request is one which attaches a HTTP header with some sort of information that basically asks the server: has data changed since this previous request? If the data is the same, you can skip downloading it all Armani, which reduces load on the server, reduces data going over the wire, saves battery use on mobiles, and reduces data transfer.

Using HTTP caching there are two headers that enable this functionality: If-Modified-Since, and If-Match-None. The first accepts a timestamp, and basically the client is letting the server know then time it last got a response, so it only cares about new data. The second is a bit more involved. Maybe you've heard of the concept of Etags, but are not really sure what they are?

Etags are usually some unique hash, which in web frameworks like Rails are a md5 checksum of the type of model, a unique ID, and an updated at timestamp.

```
etag = md5(author/123/2018-12-01)
```

This etag is then returned in a response to a GET request, which the client can save, and reuse on a subsequent request. In the request it goes into the If-Match-None header, and if the API is paying attention if it will rerun the checksum. If the checksum matches it will return a 304 Not Modified with no body, and if there is a mismatch it will shove the normal JSON response into the HTTP body

No Need to Roll Your Own

Writing all the code to handle this on the client side would be a big job. Luckily, there are solutions built in pretty much every single language.

Ruby

```
client = Faraday.new do |builder|
  builder.use :http_cache, store: Rails.cache
  ...
end
```

plataformatec/faraday-http-cache - a Faraday middleware that respects HTTP cache

PHP

```
use GuzzleHttp\Client;
use GuzzleHttp\HandlerStack;
use Kevinrob\GuzzleCache\CacheMiddleware;

// Create default HandlerStack
$stack = HandlerStack::create();

// Add this middleware to the top with `push`
$stack->push(new CacheMiddleware(), 'cache');

// Initialize the client with the handler option
$client = new Client(['handler' => $stack]);
```

Kevinrob/guzzle-cache-middleware - A HTTP Cache middleware for Guzzle 6

Python

```
import requests
import requests_cache

requests_cache.install_cache('demo_cache')
```

requests-cache - Persistent cache for requests library

JavaScript (Browser)

```
// Download a resource with cache busting, to bypass the cache
// completely.
fetch("some.json", {cache: "no-store"})
  .then(function(response) { /* consume the response */ });
```

```
// Download a resource with cache busting, but update the HTTP
// cache with the downloaded resource.
fetch("some.json", {cache: "reload"})
  .then(function(response) { /* consume the response */ });
```

Fetch API - Replacement for XMLHttpRequest build into most modern browsers

JavaScript (Node)

```
const http = require('http');
const CacheableRequest = require('cacheable-request');
const cacheableRequest = new CacheableRequest(http.request);
const cacheReq = cacheableRequest('http://example.com', cb);
cacheReq.on('request', req => req.end());
```

cacheable-request - Wrap native HTTP requests with RFC compliant cache support

Note: *Despite NodeJS having a Fetch API polyfill, it does not support cache mode, and therefore alternatives must be used.*

Go

```
proxy := &httputil.ReverseProxy{
    Director: func(r *http.Request) {
    },
}

handler := httpcache.NewHandler(httpcache.NewMemoryCache(), proxy)
handler.Shared = true

log.Printf("proxy listening on http://%s", listen)
log.Fatal(http.ListenAndServe(listen, handler))
```

lox/httpcache - An RFC7234 compliant golang http.Handler for caching HTTP responses

Real World Considerations

Not every HTTP GET request is one you want to cache. The middleware will generally do the correct thing so long as the server has declared their intentions well, but regardless of how well the server declares its cacheability, you may way to store things for longer, shorter, or not at all.

Maybe Stale is Better Than Nothing

Disrespecting the max age of a response can have similar effects to ignoring the use-by date on a carton of milk, but if you're aware of what you're doing then sometimes ignoring the intentions of the server to persist longer makes sense.

Admin Panels

There will be times when you want to make sure things are as fresh as possible, and don't mind waiting a little longer to get it. If you are calling the same API for both typical frontend functionality for a user-facing web/mobile app, and also using it to populate data for an "admin panel", then you might want to skip cached responses for the admin panel. Sure you can use cached results on many of the admin panel "list" or "overview" pages, but when it gets to the "edit form" you would be better off waiting a little longer to get the latest information.

Hard Refresh in your App

Writing your own application caching logic for requests to other site can lead to unexpected caching in front end applications. End users of web applications expect the refresh button to work, and if you have cached data in a way that won't work with the refresh button 5t can cause trouble. End users of mobile devices generally expect to "pull down to refresh" on feeds or similar interfaces, which can be problematic if its not there. Following the rules of HTTP caching makes it pretty easy to implement this functionality locally in your front end application. Again, you can simply throw a Cache-Control: no-cache on there.

Sometimes HTTP Caching is Inefficient

If you are making multiple calls to APIs with large responses to create one composite resource (one local thing made out of multiple remote things) you might not want to cache the calls.

If the client is only using a few fields from each response, caching all of the responses is going to swamp the cache server. File-based cache stores might be slower than making the HTTP call, and Redis or Memcache caches may well run out of space.

Besides, restitching the data from those multiple requests to make the composite resource locally may be too costly on the CPU. In that case absolutely stick to application-level caching the composite resource instead of using the low level HTTP cache. You can use your own rules and logic on expiry, etc. because the composite item is yours.

One final example: if you have data that changes based on the authenticated user, you'll need to use `Vary: Authentication`, which basically segments the caches by `Authentication` header. Two requests that are identical in all ways other than the `Authentication` header will result in two different cache results.

This can lower cache hit ratios so much it might not be worth worrying about. Depends. Give it a try.

GRPC

Seeing as gRPC is not an "endpoint-based" API implementation, there is no way for HTTP caching to work. That said, if they have implemented the "REST Bridge" then they might have applied Cache-Control headers, so maybe you can hook onto that. The REST Bridge really just means RESTish (they have endpoints instead of firing methods and arguments at it), so same rules apply.

If you want to cache gRPC data and they do not have the REST bridge, then you need to roll your own application caching. Pick an arbitrary number that seems appropriate, and cache away.

GraphQL

The recommendation from the GraphQL documentation suggests the responsibility of caching falls on clients to implement their own application caching:

> In an endpoint-based API, clients can use HTTP caching to easily avoid refetching resources, and for identifying when two resources are the same. The URL in these APIs is a globally unique identifier that the client can leverage to build a cache. In GraphQL, though, there's no URL-like primitive that provides this globally unique identifier for a given object. It's hence a best practice for the API to expose such an identifier for clients to use.**—Source: graphql.org**

This is advertised like a feature, but as we discussed already having a client decide arbitrary cache lifetimes is often rather questionable. Due to the way GraphQL is implemented on a single endpoint (and usually as a POST(, trying to use any existing client caching middleware would not work.

There are some third-party extensions showing up that place extra metadata into the response, and that metadata looks a lot like some of the keywords found in the HTTP caching standard. If you spot these keywords on an API you are working with, check the API documentation to see if there is mention of which of these various extensions it is, so you know how to work with it.

Network Caching

The same conventions that govern HTTP client caching caching also govern HTTP network caching; in that the majority of it is operated through the same HTTP headers like Cache-Control and Etag.

Network caching is a really powerful but often overlooked component in a robust API-centric architecture. Whilst client caching focuses on local caches on each device that is making calls, network caching focuses on sharing responses to requests that pass through the network, which could be potentially made by different clients.

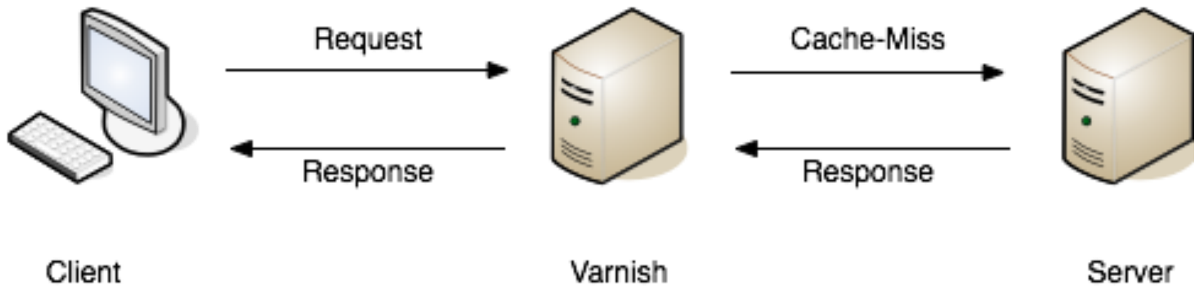
This has the benefit of taking traffic off of application servers, meaning some traffic spikes can be smoothed out whe n clients are request similar data. It also provides similar benefits to caching CSS/JS/images on "the edge".

● Fastly POP
⊖ Multiple POPs
● Planned POPs

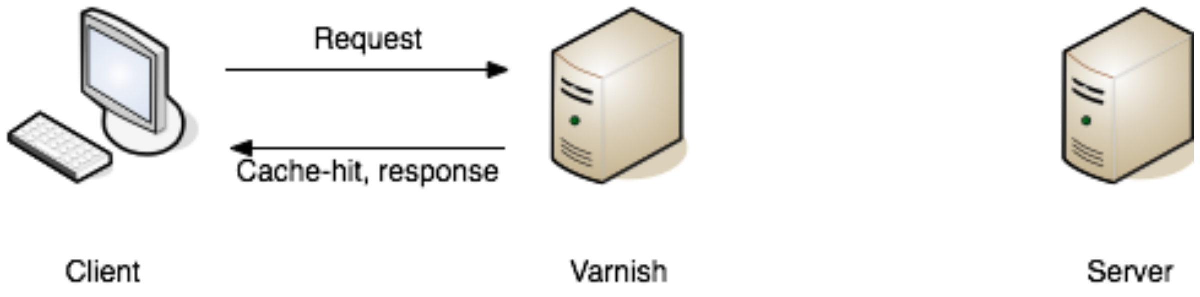
CDNs cache assets on servers physically spread around the world, meaning the assets spend less time traveling over the wire, and that means quicker downloads times for the end users requesting them. API responses can be cached in exactly the same way.

Network caching and client caching can be used together in combination, following the same set of rules, which helps to avoid complex invalidation logic. Thanks to client caching you can skip making request sometimes, then other times you grab data from somewhere physically closer to you than the application server may be. On top of the geo benefit that network cache response is coming straight from memory, instead of waiting for some poorly optimised API written in a dynamic language hastily put together by developers focusing on business goals and maybe not writing the very most performant code possible.

The basic idea looks a little like this:



A request being returned early by a cache server.—book.varnish-software.com



A request failing to find a match (a.k.a cache miss), and being passed on to the API server to fulfill.— book.varnish-software.com

Varnish and Squid are two common tools, and Fastly is a hosted version of Varnish. Hopefully the API developers will mention they use a network cache in their documentation somewhere, but if they do not there are common signs to look out for. Most systems will add some headers, like X-Cache, X-Cache-Hit, and there is also X-Served-By which in the case of Fastly will let you know the names of some cache nodes that served it up.

The HTTP-based cache tools can leverage HTTP headers like Etag, Vary, Cache-Control to handle cache validation, and know all the rules of HTTP, meaning this application caching can essentially be thrown in and function with very little effort from the API developers. Clients will get a speed boost without even having to implement their own client caching, even though they still could, and still should, as requesting data over the wire from the cache server is still slower than not requesting data.

An interesting thing about these HTTP conventions is that they were designed to work in a lot of situations that you probably never considered. I was blown away to hear of a use case, where HTTP cache proxies were installed in towns around Africa, to provide cached responses for websites that either didn't already network cache their responses, or did but didn't have any cache servers anywhere near that town. It meant that everyone saves a bunch of money on their data plans and the internet still works as expected.

The same logic that applies to websites also applies to data. If you are talking to a third-party API which has cache control headers but they didn't bother setting up network caching, you can set that up for them with these tools.

Max-Age Network Caching

The easiest type of network caching to understand is max age based stuff. If the API shoved a max-age=60 in there, the cache server will simply return that value if the request is within 60 seconds of a previous matching request. Theoretically the data cloud have changed, but the API is declaring that using 60 second old data is good enough.

A common misconception abut network caches is that they'll always returned cached data and clients have no say in the matter. In the client caching section we talked about choosing when to skip the local cache — for things like hard refresh, or for whatever other reason the freshest data is required - and its exactly the same with network caching. The Cache-Control header can be used to bypass the cached version, and hit the application server to fetch the freshest data.

Something I find to be very cool, and exceptionally handy, is that API developers can set specific instructions for a network cache on top of the usual rules. If a API developer sets Cache-Control: max-age=30 then sure, clients and the network will both keep that thing around for 30 seconds, but if they set Cache-Control: max-age=30,s-maxage=86400 then its going to keep that thing around on the cache server for a whole day, but the client cache will only last for 30 seconds. This allows the developers to set up their own edge cache purging process, updating the cache server proactively if things change, and it still keeps the client applicatiton performant by skipping repeat requests in a 30 second window. Then when the thing is past thirty seconds its off to the network cache, which is hopefully physically closer!

Conditional Validation In Network Caching

Ok so time based network caching might make enough sense, but what about conditional validation with things like Etag?

When you make a conditional request to a cache server, you are always going to hit the application server. This confused me so much when I first started digging into this. What on earth is the point of that?

Assuming there was a max age on there, the response was considered “fresh” for some time, then the client cache had to validate to see if anything has changed. The conditional check will pass straight through the network cache, and the same conditional check is made to the application server. If the etag does not match, then it is considered “stale”, and the full response will be returned by the API: a 200 OK with a whole bunch of JSON to give the network cache, and the client cache, a full new response to hold onto. If things changed, the 304 Not Modified is returned by the application and passed through the network cache to let the client cache know its got the latest thing.

Network caching is not wildly useful for conditional validation, it is mainly there for max age based stuff, but if the API developers are leveraging both then it still has value.

GraphQL

Due to the way GraphQL generally operates POSTing against a single endpoint, HTTP is demoted to the role of a dumb tunnel, making network caching tools like Varnish, Squid, Fastly, etc. almost entirely useless. If the API developers were kind enough to also allow queries to be made over GET then you can switch to using that, and technically network caching will work at a very basic level, but the chance of a cache hit is minimal, as the entire query has to match perfectly (asking for foo and bar, not bar and foo). It's not even guaranteed that GET is supported for a given API.

With standard HTTP-based network caching essentially removed from the equation, some third-party solutions have started popping up. One of these is FastQL, a name inspired by Fast.ly but built specifically for GraphQL. It might not be easily to tell if the API developers have that enabled.

If they have not enabled a network cache you are out of option. Unlik HTTP-based caching solutions, client developers are not be able to run their own network cache that the API development team is unaware of, as it relies on purge requests being made to it to remove outdated information and replace it with newer information. Basically that means if you set up your own network cache, you would have to find some way to subscribe to data changes on their end, and repopulate the data on the cache server maybe using some sort of scheduled job, all of which sounds like quite the faff.

The Plan!

Ok that was admittedly a lot of information thrown around, so let us take a step back, and figure out how you can get some caching on your API interactions right now: what you can do yourself, and what you might need a little help on.

Which Paradigm is in use?

If gRPC or any other type of RPC API, there will not have any caching metadata to hook onto for automated client caching. Don't even worry about trying to figure it out.

If using GraphQL there might some client side data, but you need to figure out which extension is being used and find a matching client side tool or middleware.

If REST or RESTish, the API developers may well have implemented caching, but it is still not guaranteed. To find out…

Look for Cache-Control and Etag

If the API you are talking to does not have a Cache-Control header, maybe politely ask the API developers to consider it. They might think the data is uncacheable, but they can probably put a 10-30 second cache on it at the very least.

Even if they think their data is so very precious that it could not possibly have any sort of cache time, Etags can be used to speed up requests when data has no changed, by skipping rendering and downloading JSON. Let them know that supporting conditional requests will lighten the load on their servers and make their API quicker with basically no work, and you'll almost certainly get that feature implemented.

Add Client Caching Middleware

Find a middleware for your HTTP client of choice, and if that client does not support middleware you should switch out for one that does. Every programming language has a lot of HTTP clients of varying quality, and the best always support middleware.

If there is no HTTP caching middleware for any HTTP client in the language you are using, it might be time to put on your open-source hat and build one.

You will need a data store for this caching middleware, and that will depend on the language and ecosystem you are building for. If it is a backend application then you'll probably be setting up Redis or Memcache, and if it is front end then check out Local Storage.

Identify no-cache Requests

Figure out which parts - if any - of your application require the freshest possible data, and add Cache-Control: no-cache on there to force revalidation on that request.

Check for Network Caching

Look for hints in the documentation that network caching has been setup, and if there is nothing there scout around the responses for X-Cache headers - or something similar.

If there is no network caching, it might be because the API developers have done such a fantastic job of replicating their servers to data centers all over the world that they didn't see the need, but this is both highly unlikely, and not entirely true even if they have put API servers in Mumbai, Helsinki, Sydney, Peru...

Network caching can often help smooth out traffic spikes, and fulfill a reasonable percentage of traffic when the API server has gone down. Even when API servers are spread all over the world, well load balanced, auto-scaled and finely tuned, having a network cache on there is just going to help speed up requests for max-age based stuff, and there is basically no overhead.

Don't Let APIs Be Slow

If the API is offering cached responses of ~50ms but the uncached responses are taking 500ms, you should have a chat with their development team about how they're using caching to *simulate* good performance, and explain that hiding a performance issue behind caching is an unacceptable poor practice.

Remember, going over the wire is inherently slow and fraught with danger and potential issues. Caching helps clients do that less often by identifying which questions they already have answers to, but that does not mean API developers can stop worrying about performance.

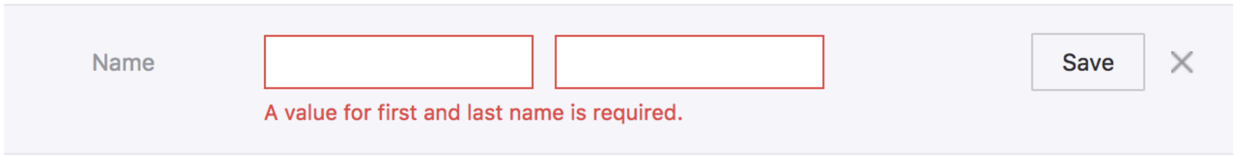
Chapter Eight

Validation

Whenever an API client attempts an operation (creating a REST resource, triggering an RPC procedure, etc.) there are usually some validation rules to consider. For example, the `name` field is required and and cannot be longer than 20 characters long, `email` must be a valid email address, the `date` field should be a valid ISO 8601 date, and either the home phone *or* mobile phone field must be entered to send a text message, etc.

There are two locations in which these rules can live, the server and the client.

Client-side validation is incredibly important, as it provides immediate visual feedback to the user. Depending on the UI/UX of the client application, this might come in the form of making invalid boxes red, scrolling the user to the problem, showing basic alert boxes, or disabling the submit button until the local data looks good.



To create this functionality, a common approach is to reproduce server-side validation in the client-side. This should be avoided at all costs.

It is awfully common for client developers to read the API documentation, take note of prominent validation rules, then write a bunch of code to handle those rules on their own end. If their code considers the data valid, it will pass it all onto the server on some event (probably on submit) and hopefully the server will agree.

This seems to make sense at first, as frontend applications want immediate feedback on input without having to ask the server about validity. Unfortunately this approach does not scale particularly well, and cannot handle even the smallest evolution of functionality. Rules can be added, removed or changed for the API as the business requirements change, and clients struggle to keep up to date. Arguably the API development team should not be randomly changing things, but change always happens in some form. The varying versioning strategies for APIs will be discussed later, but even extremely cautious API developers can introduce unexpectedly breaking change when it comes to validation.

The most simple example would be the above mentioned name field, with a max length of 20 characters. Months later, requirements come in from the business to increase the max length to 40, and the API rolls out that change. The API developers have a chat, and decide that making validation rules more lenient cannot be considered breaking, as all previous requests will still function. So they deploy the change with a max length of 40 characters, and one client-app deploys an update to match, increasing the validation to 40. Despite various client applications still having the hardcoded max length at 20 characters, everyone feels pretty happy about this.

One user jumps on this chance for a long name, going well over the 20 character limit. Later on the iOS application, they try to edit another field on the same form, but notice their name has been truncated to 20 characters as the input field will not take any more than that. Confused, they grab their friends phone and try it out there. The Android application does something a little different: the full 40 characters are populated into the form, but client-side validation is showing an error when the form is submitted. The user profile cannot be updated at all on this application without the user truncating their name…

This is certainly not the only potential issue. The API creating a new allowed value to an "enum" field will cause the same sort of problem.

Depending on the API you are attempting to integrate with, you may be doomed to suffer this sort of fate, but there are ways to mitigate this problem. Depending on the type of API, you can probably find a way to synchronize validation between client and server.

Client-Validation via Contracts

Well built APIs generally offer a copy of their contracts in a programmatically accessible format. The idea is that validation rules should be written down somewhere, and not just inside the backend server. If the validation rules can be seen by API clients, clients are going to be far more robust, and not break on tiny validation changes.

These contracts can be called many things. Some refer to them as "data models" or "schemas", and there are many different implementations with different pros and cons. Here are some of the good ones.

- JSON Schema
- OpenAPI
- GraphQL Types
- Protocol Buffers
- JSON-LD
- XML Schema

These contracts provide a wide array of functionality to API developers. Some API developers generate human-readable documentation from them, some even generate SDKs from them. Some will use them for validating incoming payloads to save writing up the same validation rules themselves. Client-side validation is yet another feature most of these systems are able to provide, saving a trip over the wire, and hopefully avoiding validation hell.

JSON Schema

JSON Schema is a layer of metadata for JSON, that is written in JSON. Unlike some formats which are knocked out by one particularly large company, JSON Schema is a web standard being worked on by smart developers from all sorts of companies like

Cloudflare, Microsoft, and they even let me stick my oars in a bit.

The idea is to point our which fields might exist, which are required or optional, what data format they use, and other validation rules can be added on top of that basic premise. The metadata lives in .json files, which might look a bit like this:

```
{
  "$id": "http://example.com/schemas/user.json",
  "type": "object",
  "definitions": {},
  "$schema": "http://json-schema.org/draft-07/schema#",
  "properties": {
    "name": {
      "title": "Name",
      "type": "string",
      "description": "Users full name supporting unicode but no
     emojis.",
      "maxLength": 20
    },
    "email": {
      "title": "Email",
      "description": "Like a postal address but for computers.",
      "type": "string",
      "format": "email"
    },
    "date_of_birth": {
      "title": "Date Of Birth",
      "type": "string",
      "description": "Date of uses birth in the one and only date
     standard: ISO 8601.",
      "format": "date",
      "example": "1990-12-28"
    }
  },
  "required": ["name"]
}
```

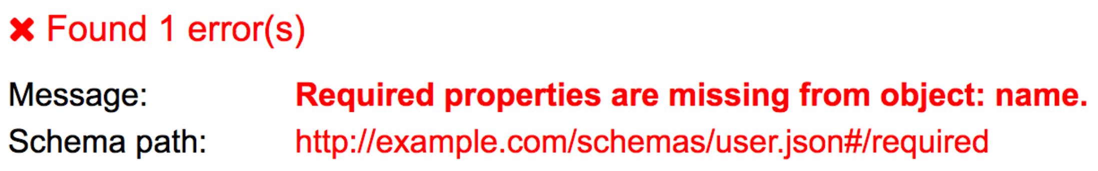
There is quite a lot of stuff here, but most of it should make sense to the human eye without too much guesswork. We are listing "properties" by name, giving them a data type, setting up maxLength for the name according to the example earlier, and putting human-readable descriptions in there too. Also some examples have been thrown in for giggles.

The one bit that probably needs more explaining is the $schema key, which is pointing to the draft version of JSON Schema in use. Knowing which draft you are validating against is important, as a JSON Schema file written for Draft 07 cannot be validated with a Draft 06 validator, or lower. Luckily most JSON Schema tools keep up fairly well, and JSON Schema is not madly changing crap at random.

Anyway, back to it: An example of a valid instance for that .json schema file might look like this.

```
{
 "name": "Lucrezia Nethersole",
 "email":"l.nethersole@hotmail.com",
 "date_of_birth": "2007-01-23"
}
```

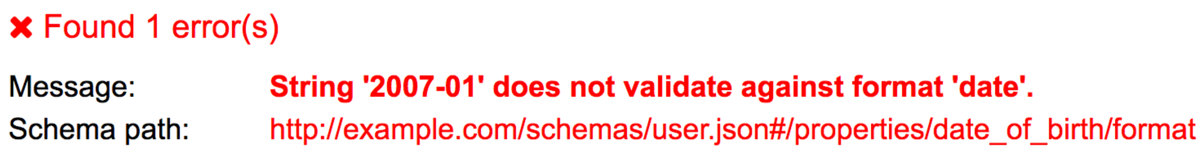
To try playing around with this, head over to jsonschemavalidator.net and paste those in. Removing the name field triggers an error as we have `"required": ["name"]` in there.



✖ Found 1 error(s)

```
Message:        Required properties are missing from object: name.
Schema path:    http://example.com/schemas/user.json#/required
```

Another validation rule could be triggered if you enter date of birth in an incorrect format.



✖ Found 1 error(s)

```
Message:        String '2007-01' does not validate against format 'date'.
Schema path:    http://example.com/schemas/user.json#/properties/date_of_birth/format
```

Conceptually that probably makes enough sense, but how to actually programmatically get this done? The JSON Schema .json files are usually made available somewhere in a HTTP Link header with a rel of describedby.

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

This might look a bit off to those not used to Link headers, but this is how a lot of links are handled these days. The one difficulty here is parsing the value, which can be done with some extremely awful regex, or with a RFC5988 compliant link parser - like parse-link-header for JavaScript.

Another approach that some APIs use (like the Postman Pro API) is to shove a URL in the HTTP body instead. A GET request on `/collections/{id}` will return all the JSON, and a schema field somewhere in the payload.

```
{
    "collection": {
        "info": {
            "name": “Turtles.com”,
            "description": “Postman Collection for Turtles.com“,
            "schema": "https://schema.getpostman.com/json/collection/
     v2.0.0/collection.json"
        },
```

Either way, once a API client has the schema URL they can download the file. This involves simply making a GET request to the URL provided. Fear not about performance, these are usually stored on CDNs, like S3 with CloudFlare in front of it. They are also very likely to have cache headers set, so a HTTP client with caching middleware will keep that locally, or you can manually cache it by inspecting the cache headers. More on that later.

Triggering validation rules on a random website is one thing, but learning how to do that with code is going to be far more useful. For JavaScript a module called ajv is fairly popular, so install that with a simple `yarn add ajv@6`, then shove it in a JavaScript file. This code is available on the GitHub Repository apisyouwonthate/talking-to-other-peoples-apis-code.

```
const Ajv = require('ajv');
const ajv = new Ajv();

// Fetch the JSON content, pretending it was downloaded from a URL
const userSchema = require('./cached-schema.json')

// Make a little helper for validating
function validate(schema, data) {
  var valid = ajv.validate(schema, data);
  if (!valid) {
    return ajv.errors;
  }
  return true;
}

// Pretend we've submitted a form
const input = {
 name: "Lucrezia Nethersole",
 email: "l.nethersole@hotmail.com",
 date_of_birth: "2007-01-23"
}

// Should be valid
console.log('valid', validate(userSchema, input))

// Ok screw up validation...
input['email'] = 123
console.log('fail', validate(userSchema, input))
```

For the sake of keeping the example short, the actual JSON Schema has been "downloaded" from http://example.com/schemas/user.json and put into a local file. This is not quite how you would normally do things, and it will become clear why in a moment.

A `validation()` function is created to wrap the validation logic in a simple helper, then we move on to pretending we have some input. The input would realistically probably be pulled from a form or another dynamic source, so use your imagination there. Finally onto the meat, calling the validation, and triggering errors.

Calling this script should show the first validation to succeed, and the second should fail with an array of errors.

```
node ./1-simple.js
true
[ { keyword: 'type',
    dataPath: '.email',
    schemaPath: '#/properties/email/type',
    params: { type: 'string' },
    message: 'should be string' } ]
```

At first this may seem like a pile of unusable gibberish, but it is actually incredibly useful. How? The dataPath by default uses JavaScript property access notation, so you can easily write a bit of code that figures out the input.email was the problem. That said, JSON Pointers might be a better idea. A much larger example, again available on Github, will show how JSON Pointers can be used to create dynamic errors.

Sadly a lot of this example is going to be specific to AJV, but the concepts should translate to any JSON Schema validator out there.

```
const Ajv = require('ajv');
const ajv = new Ajv({ jsonPointers: true });
const pointer = require('json-pointer');
const userSchema = require('./cached-schema.json')

function validate(schema, data) {
  return ajv.validate(schema, data)
    ? [] : ajv.errors;
}

function buildHumanErrors(errors) {
  return errors.map(function(error) {
    if (error.params.missingProperty) {
      const property = pointer.get(userSchema, '/properties/' +
     error.params.missingProperty);
      return property.title + ' is a required field';
    }
    const property = pointer.get(userSchema, '/properties' +
     error.dataPath);
    if (error.keyword == 'format' && property.example) {
      return property.title + ' is in an invalid format, e.g: ' +
     property.example;
    }
    return property.title + ' ' + error.message;
  });
}
```

The important things to note in this example are the `new Ajv({ jsonPointers: true });` property, which makes dataPath return a JSON Path instead of dot notation stuff. Then we use that pointer to look into the schema objects (using the json-pointer npm package), and find the relevant property object. From there we now have access to the human readable title, and we can build out some human readable errors based off of the various properties returned. This code might be a little odd looking, but we support a few types of error quite nicely. Consider the following inputs.

```
[
  { },
  { name: "Lucrezia Nethersole", email: "not-an-email" },
  { name: "Lucrezia Nethersole", date_of_birth: 'n/a' },
  { name: "Lucrezia Nethersole Has Many Many Names" }
].forEach(function(input) {
  console.log(
    buildHumanErrors(validate(userSchema, input))
  );
});
```

These inputs give us a whole bunch of useful human errors back, that can be placed into our UI to explain to users that stuff is no good.

```
node 2-useful-errors.js
[ 'Name is a required field' ]
[ 'Email should match format "email"' ]
[ 'Date Of Birth is in an invalid format, e.g: 1990-12-28' ]
[ 'Name should NOT be longer than 20 characters' ]
```

The errors we built from the JSON Schema using the properties that exist can get really intelligent depending on how good the schema files are, and how many edge cases you cover. Putting the examples in is a really nice little touch, and makes a lot more sense to folks reading the messages than just saying the rather vague statement "it should be a date".

If you were to instead find a way to tie these back to the DOM, you could update your forms with visual updates as discussed earlier: making invalid boxes red, scroll the user to the problem, show basic alert boxes, or disable the submit button until the local data looks good!

What about Validation Hell?

Earlier validation hell was mentioned, and JSON Schema is supposed to avoid it. But how? The API client now has this JSON Schema file locally, and if the server changes… how does it know? This sample code storing the schema in the repo along with the source code, which - generally speaking - is a pretty bad idea, only done for simplicity of the example.

Put very simply, if the API developers change the schema file to have a maxLength of

40, any client should then get that change the next time they request the schema file. That is a fluffy simplicity which has a few details to explain…

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

This URL is not versioned, which suggests that it might change. This is… possibly ok, as long as they have not set a long cache. If a client application is respecting cache headers, and the schema file has cache headers, then your application could suffer from validation hell for the duration of the cache. If the cache is only set to something short like 5 minutes, and the change is only a minor one, honestly might not be too bad. The whole "multiple devices being used to try to make profile changes and getting clobbered by a maxLength change" scenario we have been discussing actually would not really be an issue if the cache was reasonably short, but if it was set to days or longer you could be in trouble. The Caching chapter will help make more sense of this.

Some APIs version their schema files, and as such new versions *should* be published as a new URL.

```
Link: <http://example.com/schemas/v1.0.0/user.json#>;
      rel="describedby"
```

When a minor change is released like the maxLength one, API developers may well release another version.

```
Link: <http://example.com/schemas/v1.0.1/user.json#>;
      rel="describedby"
```

So long as URLs are not hardcoded in your application, and the URL is being read from the response (taken from wherever the API provides the link: body or link header), then the change of URL will automatically cause your application to fetch the new schema, allowing your application to notice the new validation essentially immediately.

Protobuff

If you are interacting with an API using Protobuf, setting up a payload is as easy as writing code. Protobuf has types defined in `.proto` files, which you will need access to. The example Google use is an `addressbook.proto` , which looks a bit like this.

```
syntax = "proto2";
package tutorial;

message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phones = 4;
}

message AddressBook {
  repeated Person people = 1;
}
```

A fairly complex example to get started with (classic Google), but the idea here is quite simple. By providing a bunch of information about what fields are expected, which are optional/required, what possible values are permitted for `PhoneType`, etc., it is possible to validate data locally and ensure the server is not going to freak out when information is sent up to it.

```
import addressbook_pb2
person = addressbook_pb2.Person()
person.id = 1234
person.name = "John Doe"
person.email = "jdoe@example.com"
phone = person.phones.add()
phone.number = "555-4321"
phone.type = addressbook_pb2.Person.HOME
```

Note that these assignments are not just adding arbitrary new fields to a generic Python object. If you were to try to assign a field that isn't defined in the .proto file, an

AttributeError would be raised. If you assign a field to a value of the wrong type, a TypeError will be raised. Also, reading the value of a field before it has been set returns the default value.

```
person.no_such_field = 1  # raises AttributeError
person.id = "1234"        # raises TypeError
```

Pretty much just the same as JSON Schema, you can turn those errors into UI feedback if the client has a UI.

Chapter Nine

Timeouts, Retries, and Circuit Breakers

Whenever possible it is best to make calls to other APIs asynchronously. Doing something in the background and letting the user know it is done usually beats making the user wait for the slow thing to happen in real time. This might mean they see a blank page, or have no idea of progress for seconds, minutes, etc.

Sometimes things cannot be done asynchronously. This could be making a request to an authentication system, to find out if the username and password, token, etc. are valid. It could also be an API following a design pattern known as the Broker Pattern. This pattern is useful when you're trying to build a new interface to disguise complexity in the background, maybe temporarily while you help two systems converge.

I have seen the proper pattern used at work after acquiring another company, we made a new "membership system" and that API would talk to the two different user/ membership systems behind the scenes, converting those systems payloads into one unified interface for consumers. It was not the most performant, but it let clients move over to the new system while we worked on unifying the systems in the background. Eventually when we were able to completely merge the systems (getting both sources of data into the new membership system), we kept the response contracts exactly the same, making it no longer a broker (and therefore a bunch quicker!)

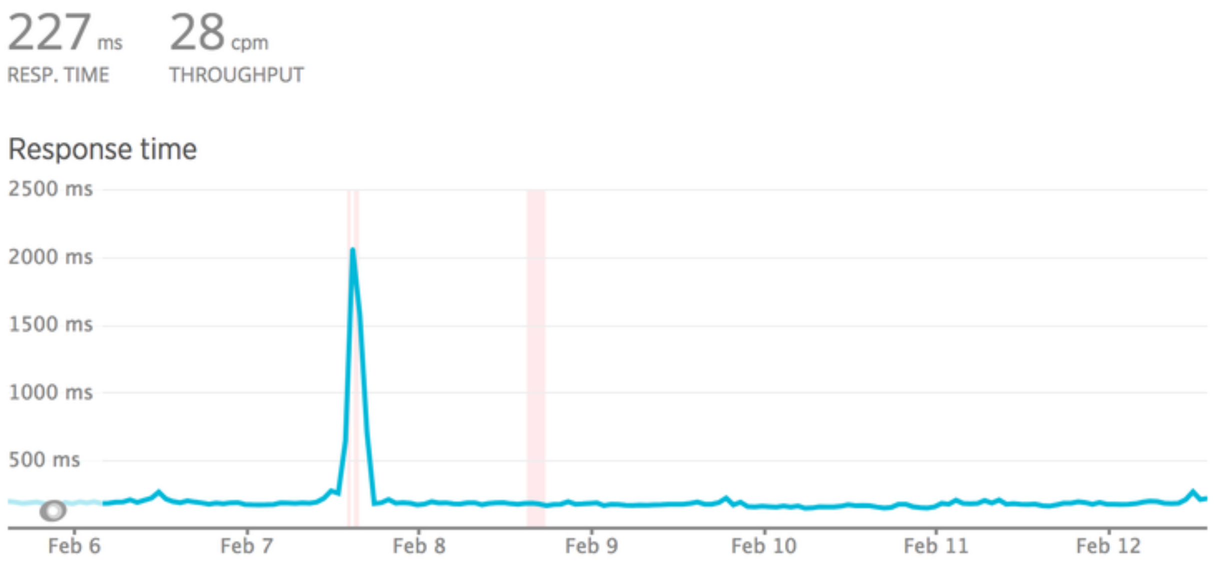
In an ideal world each service knows enough information to satisfy its clients requests, but often there are unfortunate requirements (like the one above) for data to be fetched, or actions to be confirmed, on the fly. All of these things take time, and many developers make a mistake in assuming that it will always be quick.

Frontend applications (desktop, web, iOS, Android, etc.) talk to services, and services talk to other services. This chain of calls can stack up, as service A calls service B, unaware that system is calling service C and D… So long as A, B, C and D are functioning normally, the frontend application can hope to get a response from service A within a "reasonable time", but if B, C or D are having a bad time, it can cause a domino effect that takes out a large chunk of your architecture, and the ripple effects result in a slow experience for the end users.

Slow applications can cost you a lot of money. A Kissmetrics survey suggests that for every additional second a page takes to load, 7% fewer conversions will occur, so its important to keep as much of the user experience functioning as well as possible, for as long as possible, even when upstream dependencies are having a rough time.

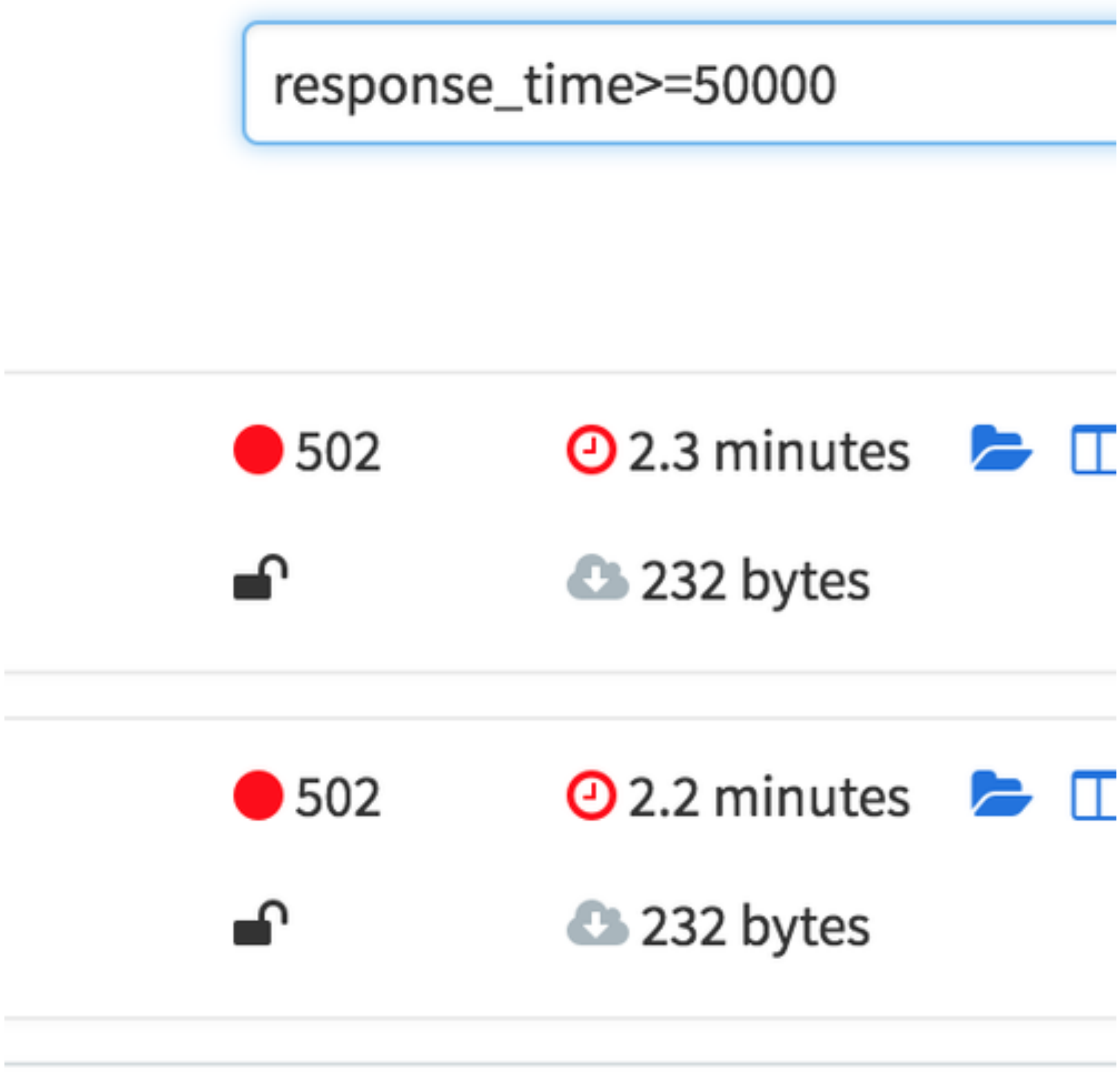
Other People's Problems

You own service A, and are making calls to service B. What happens when service B has a bad day, and instead of responding within the usual ~350ms, it starts to take 10 seconds? Do you want to wait 10 seconds for that response?



What about if service B is fine, but C is taking 20s and D is taking 25s? Are you happy to wait 45 seconds for the response from B?

What about two minutes?! 😱


response_time>=50000

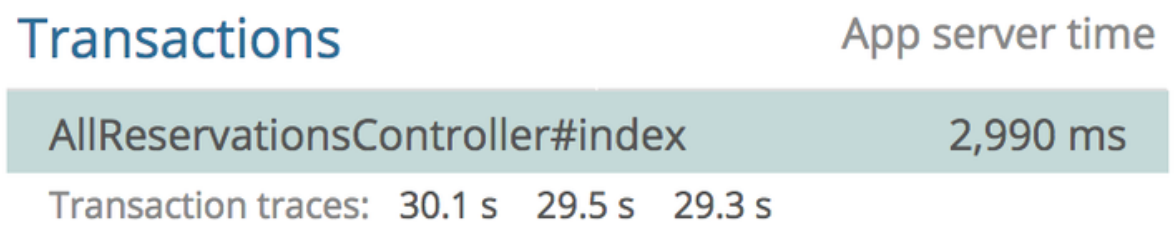
502 2.3 minutes
232 bytes

502 2.2 minutes
232 bytes

When a server is under load, it can do some pretty wild stuff, and not all servers know to give up. Even fewer client applications know when to give up waiting for the server, and those that do will take a while to do it.

For example, if service B is on Heroku, we can be confident the request is not going to last for more than 30 seconds. Heroku's router has a policy: applications get 30 seconds to send the first byte, and if that doesn't happen then the request gets dropped.

Quite often in monitoring systems like NewRelic or CA APM, you will see things like this:



Transactions	App server time
AllReservationsController#index	2,990 ms

Transaction traces: 30.1 s 29.5 s 29.3 s

This controller has an average response time of 2.9s, but the slow ones are floating right around that 30s mark. The Heroku Chop saves the caller from being stuck there indefinitely, but this behavior is not widespread. Other web servers with different policies could hang for longer, or forever.

For this reason, never presume any service is going to respond as fast as it usually does. Even if that team’s developers are super confident. Even if it autoscales. Even if it's built in Scala. If you don't set timeouts, other people's problems become your problems.

So, how can we prevent slow requests from making our clients hang indefinitely?

A client can simply say “If it ain't done in 2 seconds, I’m done. I got other stuff to do.” This concept is known as a “timeout”.

Set Timeouts in the HTTP Client

HTTP clients usually do not set a timeout by default, but accept configuration to control various aspects of timeout settings.

For Ruby users, the HTTP client Faraday might look like this:

```
conn = Faraday.new('http://example.com');
```

```
conn.get do |req|
  req.url '/search'
  req.options.timeout = 5           # open/read timeout in seconds
  req.options.open_timeout = 2      # connection open timeout in
     seconds
end
```

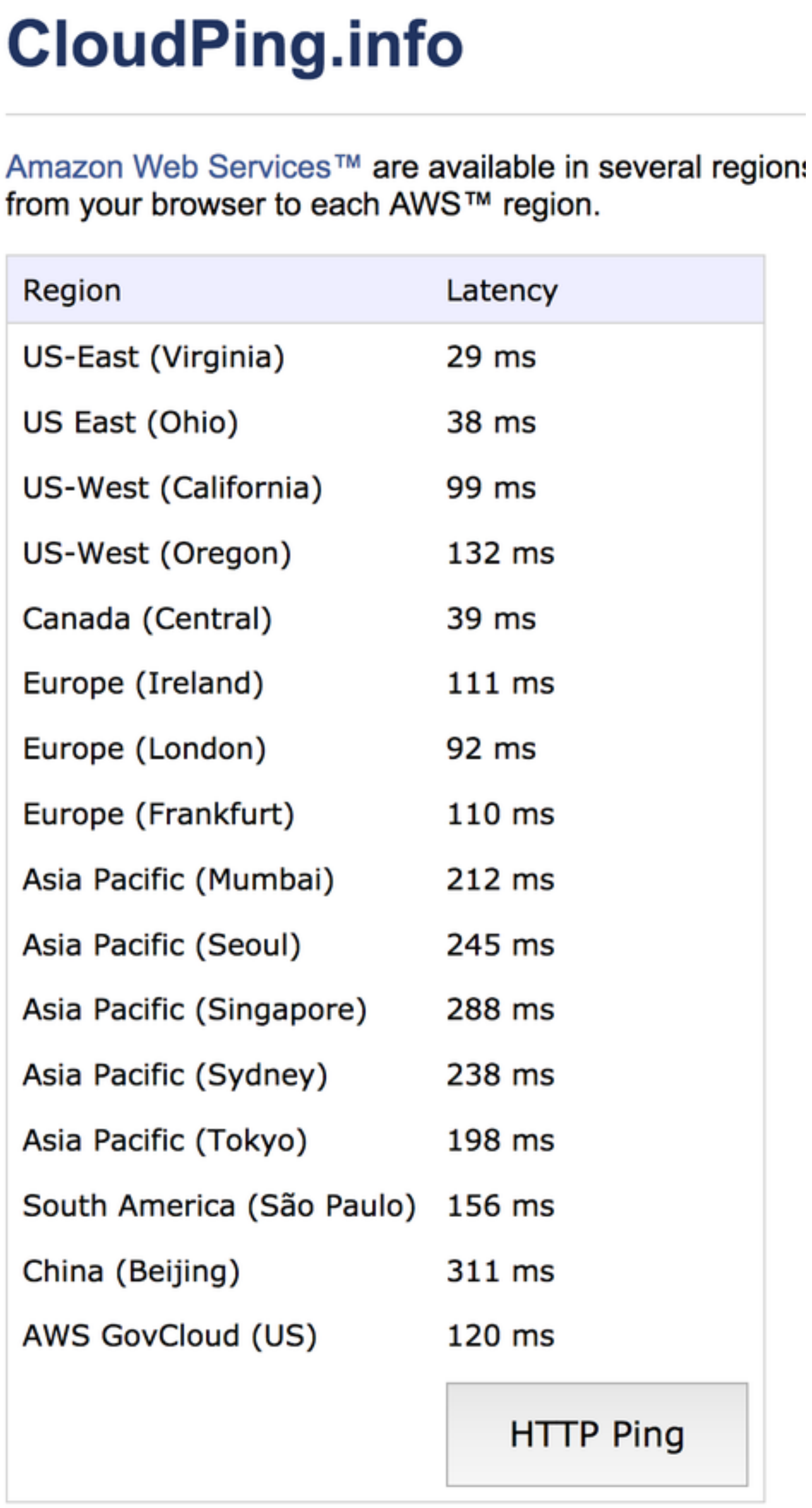
For PHP, Guzzle does this:

```
$client->request('GET', '/delay/5', ['timeout' => 5]);
```

There are two types of timeout that a lot of HTTP clients use:

1. Open (Connection) Timeout
2. Read Timeout

An **open timeout** asks: how long do you want to wait around to see if this server is actually accepting requests. That can mean many things but often means a server is too busy to take a request (there are no available workers processing the traffic). It also depends in part on expected network latency. If you are making an HTTP call to another service in the same data center, the latency is going to be a few milliseconds, but going to another continent takes time.



CloudPing.info

Amazon Web Services™ are available in several region
from your browser to each AWS™ region.

Region	Latency
US-East (Virginia)	29 ms
US East (Ohio)	38 ms
US-West (California)	99 ms
US-West (Oregon)	132 ms
Canada (Central)	39 ms
Europe (Ireland)	111 ms
Europe (London)	92 ms
Europe (Frankfurt)	110 ms
Asia Pacific (Mumbai)	212 ms
Asia Pacific (Seoul)	245 ms
Asia Pacific (Singapore)	288 ms
Asia Pacific (Sydney)	238 ms
Asia Pacific (Tokyo)	198 ms
South America (São Paulo)	156 ms
China (Beijing)	311 ms
AWS GovCloud (US)	120 ms

The **read timeout** is how long you want to spend reading data from the server once the connection is open. It's common for this to be set higher, as waiting for a server to generate an answer (run queries, fetch data, serialize it, etc.) should take longer than opening the connection.

When you see the term "timeout" on its own (not an open timeout or a read timeout) that usually means the total timeout. Faraday takes `timeout = 5` and `open_timeout = 2` to mean "I demand the server marks the connection as open within 2 seconds, then regardless of how long that took, it only has 5 seconds to finish responding."

For JavaScript users using `fetch()` you're stuck wrapping fetch and making your own deadline option…

```
const oldfetch = fetch;
fetch = function(input, opts) {
    return new Promise((resolve, reject) => {
        setTimeout(reject, opts.deadline);
        oldfetch(input, opts).then(resolve, reject);
    });
}
```

Code snippet taken from a comment on whatwg/fetch. Same idea in that it runs the error case, but the connection is not actually being aborted. That could lead to some interesting issues… hopefully fetch() gets proper timeouts before too long.

Some Must Die, So Others May Live

Any time spent waiting for a request that may never come is time that could be spent doing something useful. When the HTTP call is coming from a background worker in a backend application, that's a worker blocked from processing other jobs. Depending on how you have your background workers configured, the same threads might be shared for multiple jobs. If Job X is stuck for 30s waiting for this server that's failing, Job Y and Job Z will not be processed, or will be processed incredibly slowly.

That same principle applies when the HTTP call is made within the web thread of a backend application. That's a web thread that could have been handling other requests! For example, you are building a backend application with an endpoint `GET /external` which is making an HTTP calls some API. This call is usually blazing fast and has been forever, until suddenly squirrels chew threw some impotent cables, and that API is down a data centre.

Your application is still working for now, and as usual other endpoint are still responding in 100ms. They will continue to respond so long as there are threads available in the various workers... but if the performance issues for the squirrel chewed API continue, every time a user hits `GET /something`, another thread becomes unavailable for that 30s.

Let's do a bit of math. For each thread that gets stuck, given that thread is stuck for 30s, and most requests go through in 100ms, **that's 3000 potential requests not being handled**. 3000 requests not being handled because of a single endpoint. There will continue to be fewer and fewer available workers, and given enough traffic to that payment endpoint, there might be zero available workers left to work on any the traffic to any other endpoints. Setting that timeout to 10s would result in the processing of 2000 more successful requests.

As a general advice for backend developers it's always better to avoid making requests from the web thread. Use background jobs whenever possible.

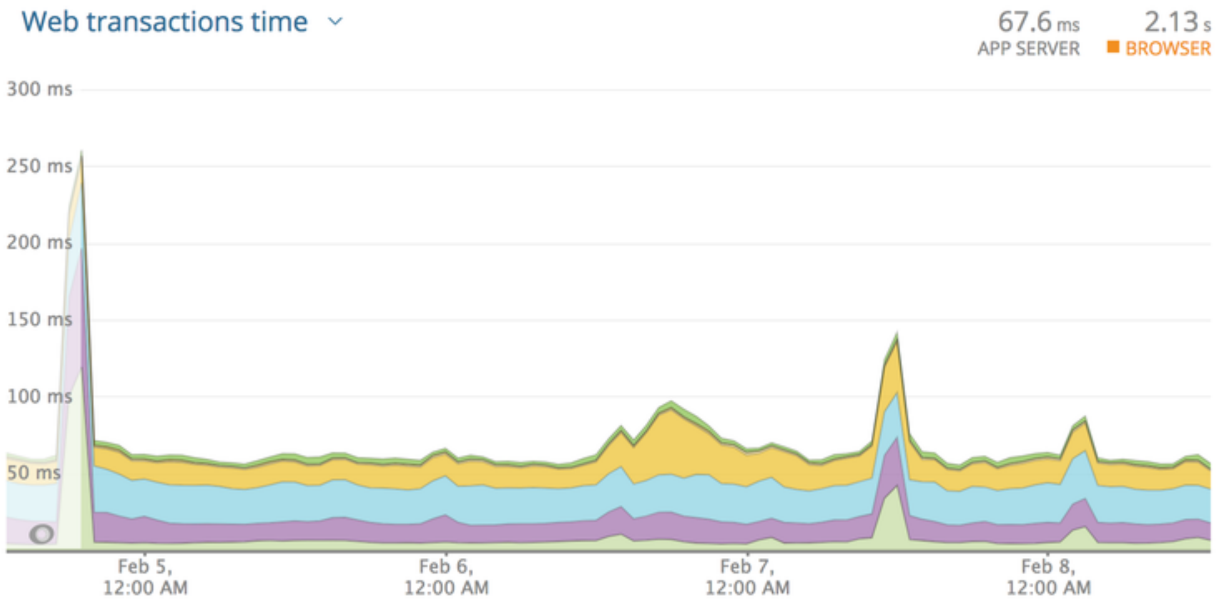
Making timeouts happen early is much more important than getting a fast failure. The most important benefit of failing fast is to give other resources the chance to work, and it gives users update into what's going on.

Frontend developer might not have to worry about freeing up server resources, but they do have to worry about the UI freezing, or other requests being blocked due to browsers HTTP/1.1 connection limits! The concept is very similar for both frontend and backend, don't waste resources waiting for responses which probably aren't going to come.

Picking Timeouts

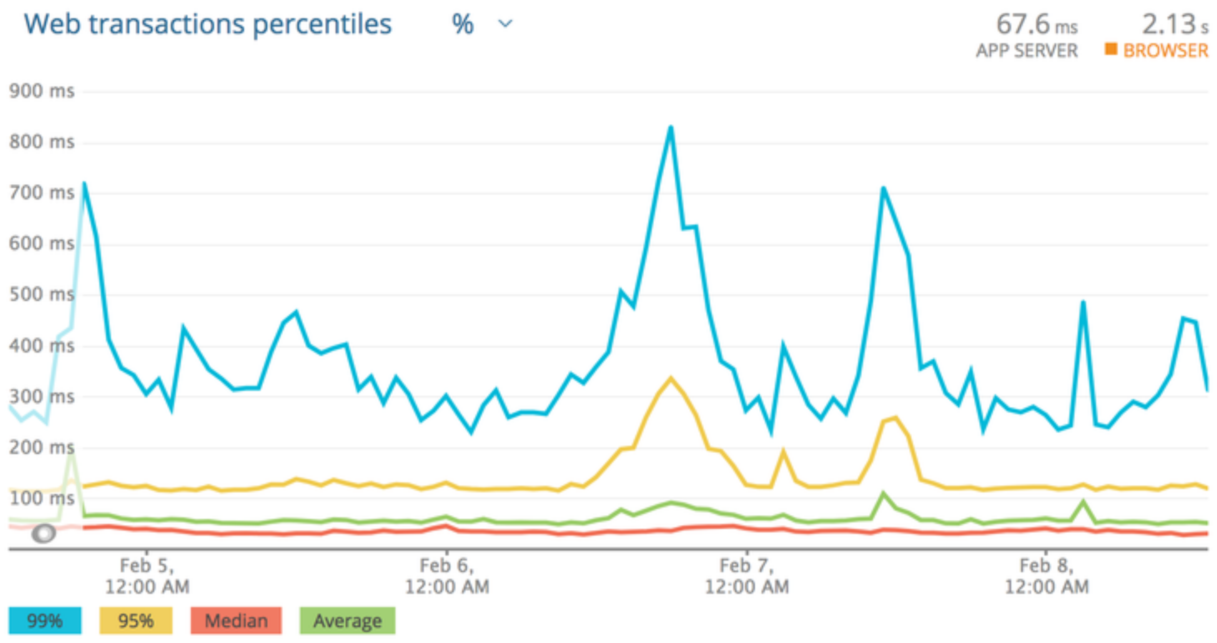
If the server is a third party company, you might have a service-level agreement stating: "Our API will always respond in 150ms". Great, set it to 150ms (and retry on failure if the thing is important.)

If the service is in-house, then try to get access to NewRelic, CA APM or whatever monitoring tool is being used. Looking at the response times, you can get an idea of what should be acceptable. Be careful though, **do not look only at the average**.



Looking at this graph may lead you to think 300ms is an appropriate timeout. Seems fair right? The biggest spike there is 250ms and so round it up a bit and let's go for 300ms?

Nope! These are averages, and averages are going to be far far lower than the slowest transactions. Click the drop-down and find "Web transaction percentiles."



That is a more honest representation. Most of the responses are 30-50ms, and the average is usually sub 100ms. That said, under **high load** this service starts to stutter, and these peaks can lead to responses coming in around 850ms! Clicking around to show the slowest traces will show a handful of requests over the last few weeks coming in at 2s, 3.4s, and another at 5s!

Those are ridiculous, and looking at the error rate we can see that those requests didn't even succeed. Whatever happens, setting the timeout low enough to cut those off is something we want to do, so far I'm thinking about 1s. If the transactions are failing anyway, there is no point waiting.

Next: if the call is being made from a background worker, that 99 percentile of 850ms may well be acceptable. Background workers are usually in less of a rush, so go with 1s and off you go. Keep an eye on things and trim that down if your jobs continue to back up, but that's probably good enough.

Retrying Slow Requests

If it's a web process... well, 2s+ is certainly no good, especially seeing as it might fail anyway. Waiting around for this unstable transaction to complete is as much of a good plan as skydiving with just the one chute. Let's create a backup plan using retries.

So we have this special web application that absolutely has to have this web request to Service B in the thread. We know this endpoint generally responds in 35-100ms and on a bad day it can take anywhere from 300-850. We do not want to wait around for anything over 1s as its unlikely to even respond, but we don't want this endpoint to take more than 1s...

Here's a plan: set the timeout to 400ms, add a retry after 50ms, then if the first attempt is taking a while *boom*, it'll give up and try again!

```
conn = Faraday.new('http://example.com');
conn.post('/payment_attempts', {  }) do |req|
  conn.options.timeout = 0.4
  conn.request :retry, max: 1, interval: 0.05
end
```

There is potential for trouble here, as the second and first attempts might end up in a race condition. The interval there will hopefully give the database long enough to notice the first response was successful, meaning the 2nd request will fail and say "already paid" or something intelligent, which can be inspected and potentially treated as a success by the client.

Anyway, (400 * 2) + 50 = 950, with another 50ms for whatever other random gumf is happening in the application, should mean that we come in at under 1 second!

This is a good place to be in. You have 2x the chance of success, and you're setting tight controls to avoid service B messing your own application up.

An important note for Ruby users: you are already using retries on idempotent requests, and you probably had no idea. It's wild that NetHTTP does this by default, even in Ruby v2.5.0 where it is configurable.

Circuit Breakers

Timeouts are a great way to avoid unexpected hangs from slowing a service down too much, and retries are a great solution to having another when that unexpected problem happens. These two concepts are both reactive, and as such can be improved with the addition of a third proactive concept: circuit breakers.

Circuit breakers are just a few lines of code, maybe using something like Redis to maintain counters of failures and their timestamps. With each failure to a service (or a particular endpoint on that service), the client increments a failure counter and compares it to a certain threshold. Maybe that threshold is 10 failures in 1 minute, or for higher volume systems maybe 5 failures in a second.

So in our example, Service A might notice that service B is down after the 10th error in 1 second, and at that point it opens the circuit breaker, meaning it completely stops making calls to that system. This will decrease the load on downstream services (B, C, and D), giving them a chance to recover. This also avoids the "running out of threads" issue we discussed previously. Even with service A giving up after 1s, that's still 1s that thread could have spent handling other requests.

What to do when a circuit breaker is open? It depends on the feature the circuit breaker is wrapping.

- Immediately respond with an error, letting the user know the required system is down, and to try again later
- Have a secondary system kick in that handles things in a different way
- Divert traffic to a cluster of servers elsewhere
- Record information about the attempt and have customer services reach out

That's only a quick intro to circuit breakers, so head over to see Martin Fowler explain circuit breakers in depth if you want more information on the topic.

"Service Meshes" like Envoy or Conduit.io are also great solutions for this problem.

Instead of asking you to code it up yourself and the application level, it can be handled with network-level tools which are able to be controlled more centrally.

Chapter Ten

Change Management

Change Management in relation to APIs, is a collective term for making improvements over time. Hopefully the change is handled in a non-breaking fashion, giving you ample time to make necessary changes.

There are a two main approaches to change.

Versioning - Shoving a version number (v1, v2, v3) next to things, or in some odd cases the version is a date. The version might apply to the entire API (global versioning), or a single resource (resource versioning).

Evolution - Instead of versioning the whole API, or specific resources/endpoints, some APIs will instead chose to carefully evolve their interface over time.

Let's go more in depth into these concepts, and how they come into effect in various API paradigms.

REST / "Endpoint-based" APIs

<$Scr_H::2>Global Versioning

<!$Scr_H::2>API endpoints are grouped (essentially "namespaced") under a version number, e.g. `https://hostname/v1/companies`, to make it clear that a `/v1/companies` has no relation whatsoever to a `/v2/companies`.

In SemVer terms, the version number in the URL is a major version. You should expect no breaking changes to occur in this major version, as that would be the job for a new major version. All resources should be copied from the old version to the new version, unless they're being discontinued.

Most APIs will not bother with minor or patch versions, because minor and patch changes are backwards compatible. Adding new fields to responses, new optional fields to requests or new resources will not break implementations, so that can be done without needing a major version. Changing validation rules, removing fields, or drastic changes to relationships, are things that may require a new major version be created.

Usually old versions will be maintained for a reasonable timeframe, but what's considered reasonable is completely up to the API maintainers. It could be one month, a year, or anything in between.

Evolution

API evolution is the concept of striving to maintain the "I" in API, the request/response body, query parameters, general functionality, etc., only breaking them when you absolutely, absolutely, have to; then when you do, you manage that change with sensible warnings to clients. It's the idea that API developers bending over backwards to maintain a contract, no matter how annoying that might be, is often more financially and logistically viable than dumping the workload onto a wide array of clients. No version numbers exist in the URL, headers, or query string, and no two versions of the same resource will exist at any given time. There is only the "current version".

Just like in global versioning, backwards compatible changes like adding new methods to existing resources, new fields to responses, new optional fields to requests, new resources, etc. These can happen any time, without clients needing to change anything. For example, splitting "name" into "first_name" and "last_name". The "name" field would still appear in responses along with the new first_name and last_name fields, and could still be sent. This is a perfect evolution, because nothing broke for the client, and the server supports the new functionality - allowing folks to be specific about their first and last names instead of the server guessing by splitting on space or something silly.

Where evolution differs from global versioning, is that instead of creating a whole new namespace (forcing clients that don't even use the specific changing resources to update/test code at great time and expense), a single new resource is made to replace the existing resource when adding new fields is not going to cover the use-case.

For example, a carpooling company that has “matches” as a relationship between ”drivers” and “passengers”, suggesting folks who could ride together, containing properties like “passenger_id” and ”driver_id”. They realized they need to support carpools with multiple drivers (i.e. Frank and Sally both take it in turns to drive), so this whole matches concept is garbage.

At a lot of startups, this sort of conceptual change is common. No number of new properties is going to help out here, as the whole “one record = one match = one driver + one passenger” concept is foiled.

Deprecating the whole concept of “matches”, a new concept of ”riders” is created. This resource tracks folks far beyond just being “matched”, through the whole lifecycle of the carpool, thanks to a `status` property containing pending, active, inactive, blocked, etc.

By creating the `/riders` endpoint, this resource can have a brand new representation. The API maintainers may well be converting matches to riders - or riders to matches - in the background, but clients don’t need to care about that. Clients can then use either one, allowing them the change to upgrade at their own pace.

GraphQL

GraphQL also uses the evolution approach, and this works the same as the REST / "Endpoint-based" API evolution approach. GraphQL APIs following evolution carefully will try to avoid breaking changes, sticking to backwards compatible changes like adding optional fields to mutations and queries, new arguments on fields, new relationships, etc.

When these backwards compatible changes are not going to be enough to provide the new functionality required, new mutations and new types might be created: just like the matches and riders example above. In another system, maybe a “Company” becomes an “Account” if that is more appropriate to the constantly changing business language, but maybe Company2 is created if not. This should only be done when the existing type is utterly unable to bend to the new rules of the context, and simply suggesting new fields cannot satisfy requirements.

Global versioning is not possible in GraphQL. When raised with Facebook the topic was closed with a WONTFIX, so you will not have to worry about that.

gRPC

TODO

Deprecations

Regardless of the type of API you are interacting with, and which approach it takes to change management, deprecation is going to be an important part of keeping up with the change. Any time a change is not just additive, deprecation will be used to communicate a breaking change.

Deprecation is the art of telling people to stop using a thing, because that thing is probably going away at some point. Even if there are no solid plans to delete the thing entirely, the thing might be deprecated just because it sucks to work with, but usually it’s because it has been replaced with a better idea.

Sadly there are no promises the team in charge will deprecate things cautiously. Some API providers act like jerks and just arbitrarily change things, remove things, and break stuff. Closing APIs down with no warning (or poor warning) is utterly disgraceful behavior, but it does happen. Facebook especially like to pull the rug out on their API users. Sometimes breaking changes are introduced accidentally.

Usually a more reasonable approach is taken, and API providers will communicate change in a variety of ways. There are a few common methods to keeping up with the changing happening to an API integrated into your codebase.

Low-Tech General Solutions

As ludicrous as it might sound, the entire extent of some API providers attempts to communicate upcoming change is “write a blog about it and hope you notice.”

These blog articles explain things like “Hey we’re closing off the free access to this API”, or “We are replacing this Foo API with a Bar API, and you have 3 months to stop using the Foo API before we shut it down”. Fingers crossed you see it!

This low-tech approach has no real solutions I can think of, other than… subscribe to the api providers engineering blog on RSS, and - so long as the RSS reader you’re using doesn't get shut down - you might notice the article. If they have a newsletter then subscribe to that, maybe with a "devs@" email address or team-specific email address, to improve the bus factor on this. You don't want the announcement going to a member of the team who's quit, then the API breaks just because the rest of the team didn't know about the change.

APIs requiring authentication will usually ask for an email address in return for the authentication credentials (API keys, access tokens, etc.) This gives API providers one more way to get in touch about change, and again, make sure it's not the email of a manager or team lead, but a distribution email address for the team.

REST / Endpoint-based APIs

If an API is using global versioning, then they might deprecate all the endpoints under `/v1/` at the same time, and suggest you use all the `/v2/` endpoints. If they're using evolution they might suggest that `/companies` is going away, and you should start to work with `/accounts` instead. The suggestions here may come in the form of low-tech announcements, but they can also be communicated in-band.

Deprecated Endpoints

There is a a proposed standard: Sunset Header, which at the time of writing is at draft 05. APIs can add a simple header to the HTTP response, to let clients know the endpoint is nearing the end of its life.

Supporting Sunset is as simple as sniffing for the `sunset` header, and it contains a HTTP date which looks a little like this:

```
Sunset: Sat, 31 Dec 2018 23:59:59 GMT
```

The proposed standard also allows responses to contain a HTTP `link` header, with `rel=sunset`. The link can be a link to anything, the endpoint replacing it, a link to some human readable documentation, a blog post, whatever.

Keep an eye out for sunset headers and their accompanying links coming back in responses your code is getting, and log, or alert, or pass to a dashboard accordingly. If you are using Ruby or PHP there are already easy to use middleware for the most popular HTTP clients:

- **PHP:** hskrasek/guzzle-sunset
- **Ruby:** wework/faraday-sunset

Sunset is on GitHub, and the issue tracker is used for discussion around it's development. Some API gateways like Tyk are building in support, and it is going to become far more wide-spread over time. If you make an implementation for your favorite HTTP client please get in touch with @apisyouwonthate on Twitter.

Another approach commonly used by API providers is to provide SDKs, and if they are kept up to date they will fling out deprecation warnings about resources that are going away.

When you get these notifications, check the errors for what to do next, or go to their documentation to figure it out, or shout at their customer support for not making it clear what you're meant to be doing next and they'll do a better job next time.

Deprecating Properties

Other than the entire endpoint going away, specific properties *may* over time be deprecated. This is less common in global versioning as they would just remove those in the next global version. It is not a regular occurrence in evolution as they too would just wait until the creation of a new resource for the concept, but it can happen if a property absolutely has to go away.

JSON is the main content type in use these days, which does not have a type system built in. There is no way to mark a JSON field as deprecated in just JSON, and no standards exist to help, even JSON Schema still has this on the todo list.

Again SDKs can mark things as deprecated over time, especially those built with OpenAPI v3.0.0 or later. OpenAPI added the `deprecated: true` keyword, so SDKs can now look for this and fire off deprecation warnings for clients using that property in their code. Keeping up to date is important, so make sure something like Snyk or Greenkeeper is implemented to keep tabs on dependencies.

GraphQL

GraphQL pushes hard for evolution in its marketing and most advice in the ecosystem, which is awesome. They do not have a huge amount of documentation covering deprecations, and much of it comes from third-parties like Apollo, but it’s certainly possible.

Deprecating Types

Instead of endpoints, GraphQL has types. There is no first class support for deprecating types in GraphQL at the time of writing, so an API developer is unable to evolve the concept through adding new properties, a new type will pop up somewhere. A low-tech solution like a blog or email announcement may be used to communicate this new type, and the deprecation of the old type, as there is no way to do it in band. Another solution is to deprecate all the properties in the type, and mark in the reason that the whole type is going away.

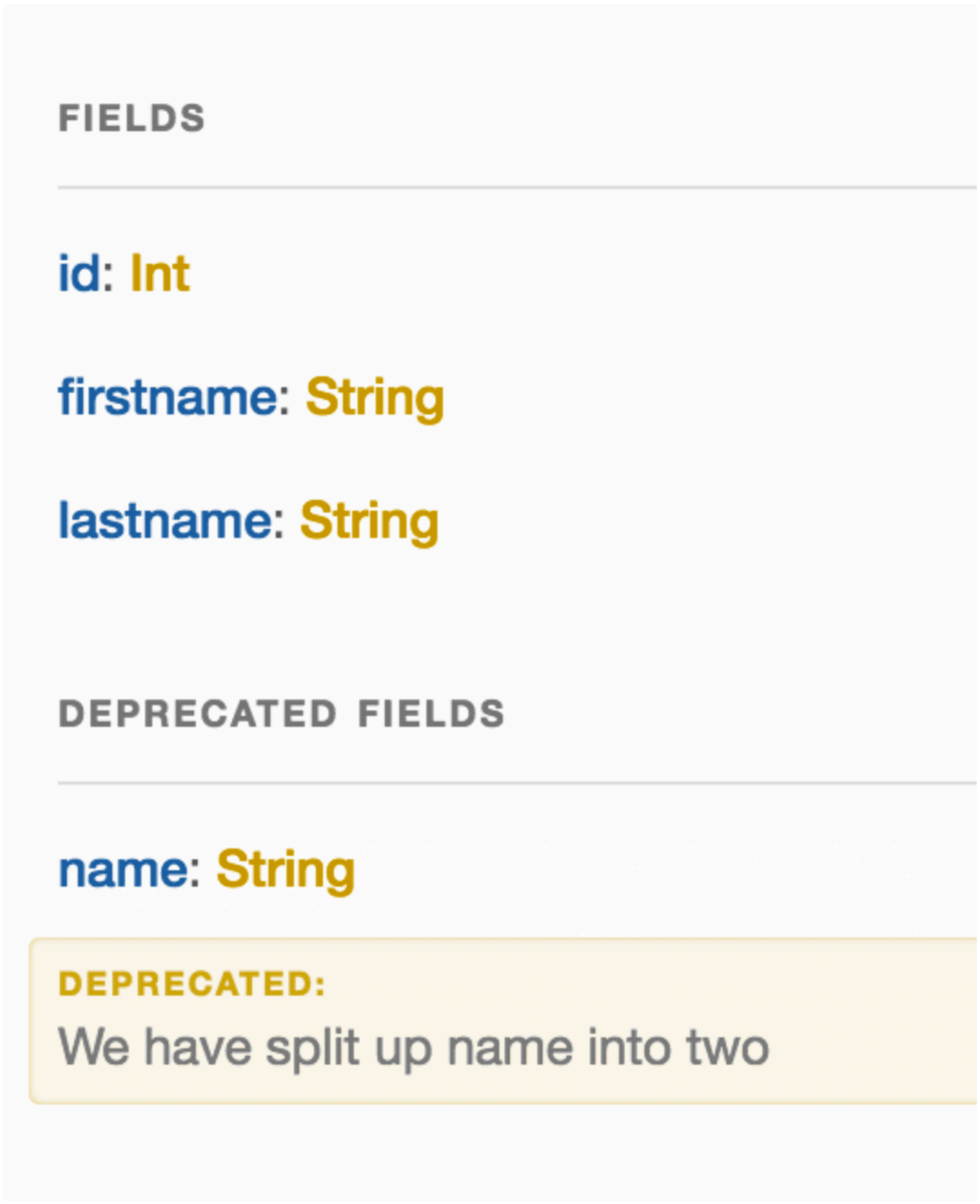
Deprecating Properties

The API provider will add the `@deprecated` keyword to the type:

```
type User {
  name: String @deprecated(reason: “Property ’name’ was split into
      ‘firstname’ and ‘lastname’”)
  firstname: String
  lastname: String
}
```

When looking at a GraphQL API through GraphiQL, the documentation that it autogenerates will show deprecated fields, visually separated from the other fields

(smart!)



FIELDS

id: Int

firstname: String

lastname: String

DEPRECATED FIELDS

name: String

DEPRECATED:
We have split up name into two

This image is from Kevin Simper's awesome article: *How to deprecate fields in GraphQL*.

Coming Soon

This book is a still an ongoing process, and more chapters will be coming. Much like *Build APIs You Won't Hate*, I'l be aiming for one chapter a month. Here's a few that I expect to add in before considering the book complete:

- Likely Errors
- API Proxies
- State Management
- Form Generation
- Offline Syncing
- Asynchronous Requests

I'm also trying to figure out how to get AMQP and WebSocket in here, possible with their own sections. It will get there.

If you have any ideas, requests, questions, or feedback, please get in touch:

phil@apisyouwonthate.com

I hope you enjoyed the content so far!