Images haven't loaded yet. Please exit printing, wait for images to load, and try to print again.

Working as a Platform Engineer @WeWork, helping their systems crash less frequently than I crash my bikes.
Aug 27 · 13 min read

# Our API Specification Workflow

A year ago we started trying to figure out the best way to not just document HTTP APIs, but to leverage API specifications to avoid duplicating efforts on loads of similar-but-different tasks; maintaining Postman Collections, creating mocks, contract testing, payload validation, etc. To us, it felt like API developers were wasting a lot of time writing up the same logic over and over again. Listing endpoints, defining what fields should have what data, figuring out validation logic, writing up examples, doing this all over and over again in loads of different formats, then—the few folks that have enough time and interest —would write loads of tests to make sure all these different formats were saying the same thing.

The initial goals were:

- One source of truth where developers need to update stuff
- API Design / Prototyping—Design first, code later when you've agreed on contracts
- Beautiful documentation for humans
- Specifications and Documentation should be kept in sync
- Validate payloads before you send them on the client side
- RSpec/PHPUnit/etc. assertions providing contract testing API responses in existing test suites
- SDK Generators with customisable templates

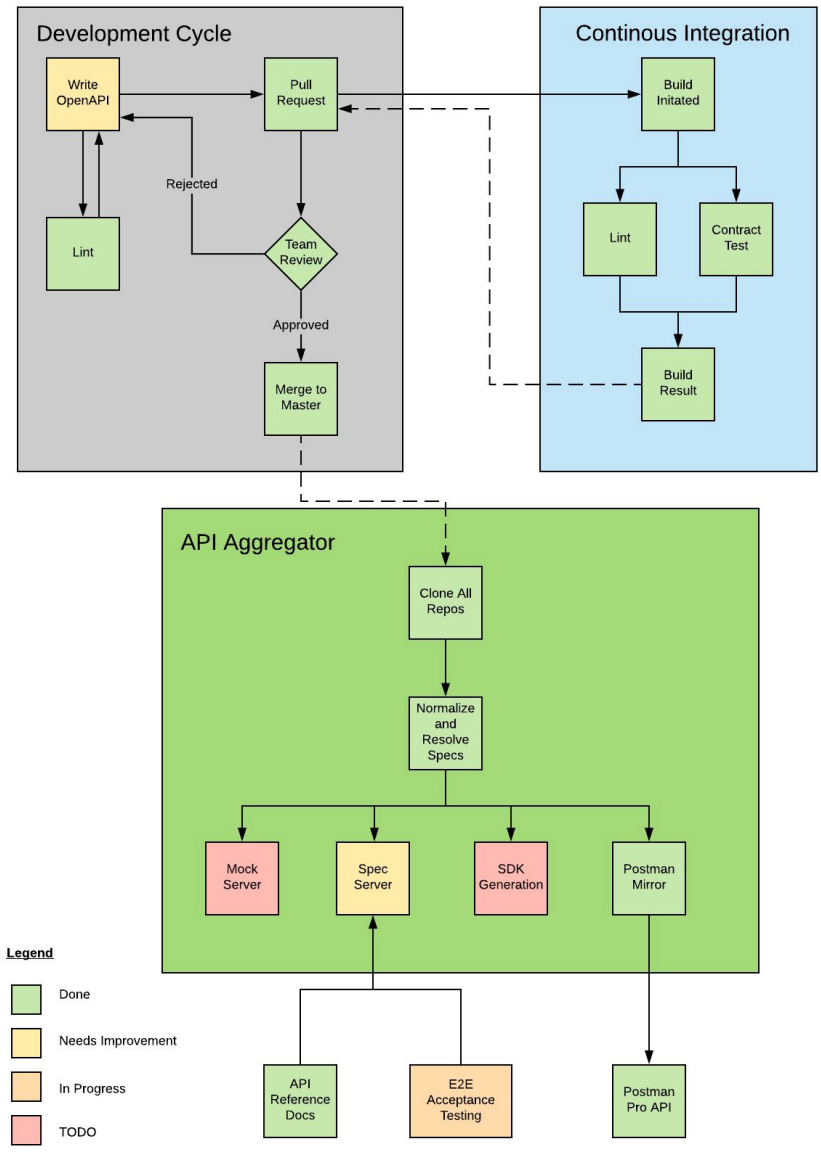
The first low hanging fruit was to get some human readable documentation. We had *nothing* for 99% of our APIs, and engineers were being forced to guess at contracts.

We had two of our APIs documented with API Blueprint, but nobody was particularly interested in writing documentation. We decided we'd need a few more carrots to dangle in order to get folks writing specs, and to do that we'd need a API specification language that could do more. We evaluated RAML and OpenAPI as alternatives, and in the end OpenAPI was chosen.

Switching OpenAPI v2.0 (hereby referred to as OASv2) to power our internal documentation portal got us from 2 documented APIs to 6 documented APIs, but a few people were hitting troubles due to hitting limitations in the language. Luckily OASv3 solved the problems we were facing. Tools like ReDoc supported v2.0 and v3.0 (using swagger2openapi internally), so we could easily produce documentation written in either version.

There was still some confusion about how JSON Schema and OpenAPI fit together. JSON Schema is incredibly powerful, and is the basis for how OpenAPI handles the schema portion of its specifications, but the two have some subtle discrepancies. We knew we wanted to leverage JSON Schema, but we also had to make it fit with OpenAPI, so we built tooling to convert from JSON Schema-proper, to OpenAPI-flavoured JSON Schema.

With documentation nailed, and folks starting to ask questions about more advanced functionality, it was time to think about how to offer the full suite of functionality API specifications can provide. Here is the workflow we created.




The legend there shows which parts we have implemented at WeWork, and which parts are still in progress.

## Spec First or Spec Later?

Most of our APIs already exist, and we're trying to get them caught up with the benefits of API specifications. There are a lot of benefits to writing your specs first, but if you learned the power of API specifications after you'd already built the API, there's nothing you can do short of invent a time machine. That said, we can still use a spec-first approach for new functionality—new endpoints, or whole new global versions in the same codebase, etc.

## Development Cycle

If you're adding a new endpoint, we recommend you add just the specs first. This lets the mocks and SDK steps do their job: allow people to play with it, and get feedback before you bother writing the application code. If you want to jump right in there and write all the code and specs all at once—knowing that you might have to change it later —then that's something you can do too.

This works for us as our APIs are private, but having specs for endpoints that do not exist in `master` might be a problem for some. Perhaps a `develop` branch helps out there, or we need to make mocks and SDKs support feature branches… Let us know how it goes for you.

**Write OpenAPI:** Writing YAML files by hand is daunting and too open-ended. Many beginners (including my past self) just don't want to do it. Thankfully several IDE/editors like Atom, VS Code, Eclipse, and the JetBrains suite, have plugins that help you autocomplete keywords, provide validation, linting, and more.

Many want a more GUI approach, and currently that's not quite there. Some such as Rápido help during the sketching phase, but lacks editing so once you've created your initial specs you're back to editing by hand anyway. There are online editors which work great, but either require you to copy and paste file contents up and down (a recipe for losing changes), or they fail to support multiple files using the `$ref` keyword. This means if you want to continue using an editor you need to maintain one giant megafile.

Some require paying per user, some offer limited import/export functionality, or lock you into using special git repos if you're lucky. Integrating these online editors into a larger workflow can be hard, so we stick to using open-source tools and write the glue ourselves.

One company is working on a GUI editor for Windows, OS X and Linux, which just works with local files. This will be amazing for anyone's workflow because you can use Git, Hg, Bazaar, FTP Sync… whatever you want. This is sadly not yet available, so the "Write OpenAPI" step remains "In Progress".

For now at WeWork we recommend folks use their IDE/editor of choice, which hopefully has a plugin available. If their editor of choice does not have a plugin, we suggest they consider switching to VS Code, because the editor and its OpenAPI plugin are both fantastic. If not, they can rely entirely on our CLI linter to get feedback on the validity of their files.

**Lint:** There are plenty of OASv3 CLI validators around which anyone can use to confirm OpenAPI files are written correctly, but "correct" is not quite enough when you have a landslide of developers coming into OpenAPI for potentially the first time, writing specs for tricky APIs. We wrote up a huge Style Guide on a wiki which nobody was really looking at, and would change, so even when folks read it they'd miss out on updates.

To improve the quality of our API specs (like reminding people to use Tags, and add descriptions to parameters so humans know what they're for), Speccy was created. Now recommendations can be made programatically, by adding new rules to the existing rules files.

**Pull Request:** Committing to master is generally frowned upon, and by making a pull request we get a chance to ask our team for review. It also allows us to utilize Continuous Integration for our API specification workflow.

**Team Review:** Currently there are a lot of teams at WeWork, and +1 is very much required for most of them. Team Review of specs is helpful, because if somebody is designing a new endpoint you should probably have another pair of eyes on it. Business requirements need to be met, and it's better to catch mistakes early on before people start using those specs.

## Continuous Integration

**Lint:** Speccy again! Just like code linters and unit tests, it's a good idea to run this stuff locally then on CI too. We all forget things when we're in a rush.

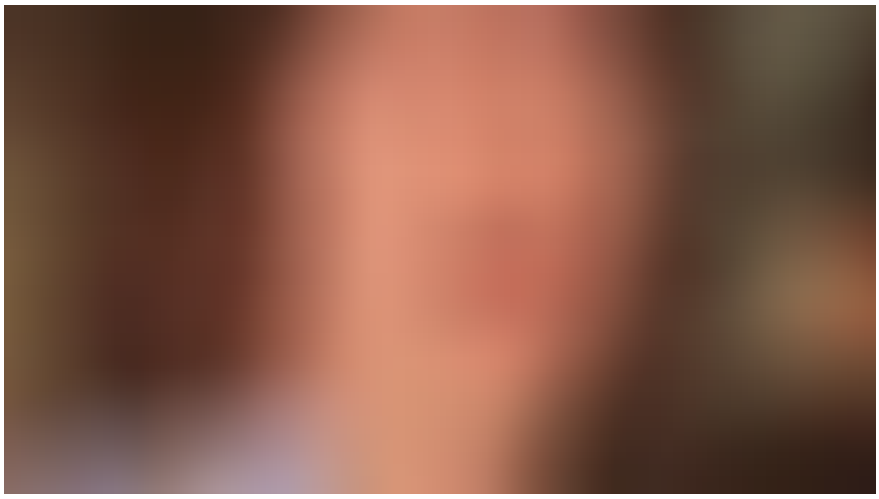
If you're using CircleCI, add the following to your `.circle/config.yml`:

```
version: 2

jobs:
    # other jobs

    speccy:
        docker:
            - image: 'circleci/node:8'
        steps:
            - checkout
            - run: 'npx speccy lint ./api/openapi.yml
--rules=strict'

workflows:
    version: 2
    commit:
        jobs:
            - speccy:
                context: org-global
                requires: [build]
```

This will fail the job not only when somebody creates something like an OpenAPI syntax error, but also when they do something Speccy doesn't like, like adding a parameter without a description. As soon as Speccy is enabled, the quality of specs can only go up.

**Contract Testing:** This step is implemented at the same time as the code, because a JSON Schema file is a great way to check your code is working according to expectations.

I've referred to this as Tricking Colleagues into Writing Documentation via Contract Testing, because developers often hate writing docs. Many developers want contract testing, so getting them to write contract tests can actually create documentation at the same time...



Write contract tests, and get awesome documentation that you know is up-to-date "for free".

JSON Schema is used here because many many more languages have JSON Schema validators, than have OpenAPI validators. JSON Schema can also cover more dynamic cases thanks to `if` / `then` / `else` keywords.

## API Aggregator

Our specification files live in the same repositories as the source code for each API, as we want the usual PR process, want code changes to match specification changes, and generally it's less faffing around for develpers this way.

The teams experimenting with gRPC are making one central repository and everyone sends their schema (protobuf) updates there, but we wanted to avoid this extra step for developers. They can control their own schemas, and we have a central process that aggregates API specifications, doing a lot of handy stuff in the process.

**Clone All Repos:** This started off as a really simple Rakefile (Ruby-flavoured make) which looks through a `data/apis.yml` file which lists repositories, and adds metadata about that API.

```
service-name:
```

```
  name: Service Name
  description: Short blurb about the thing
  repo: "git@github.com:wework/service-name.git"
  spec_dir: api/
  entry_file: openapi.yml
```

Simple bit of ruby glues YAML and the shell together:

```
apis = YAML.load(File.open('./data/apis.yml'))

task :clone do
  apis.each do |nickname, api|
    sh "rm -rf ./tmp/#{nickname} 2>&1"
    # Shallow clone and only get the branch we want
    sh "git clone #{api['repo']} ./tmp/#{nickname} -b
#{api['branch'] || 'master'} — single-branch — depth=1 2>&1"
  end
end
```

Done! So long as the user running this has SSH access to all the Git repos we're talking to, we'll have a `./tmp/` directory full of cloned repos.

The next step is where things get extra interesting!

**Normalizing and Resolving Specs:** Whilst we strongly recommend OASv3 be used as an input, reality had other ideas. Some teams are still on OpenAPI v2.0, and one or two folks are on an older "Postman First" workflow. Whilst that's been deprecated (more on that later), we wanted their APIs to at least have some level of inclusion in our workflow.

We pass OpenAPI v2.0 through swagger2openapi, and if the `data/apis.yml` indicates the input file is `format: postman`, we chuck it through the Apimatic Transformer API and have them convert it OASv3.

Even the OASv3 specifications need a little work done. Most contain `$ref`, and some weaker OpenAPI tooling does not support that feature. On top of that, some of the referenced files are written in

JSON Schema-proper instead of OpenAPI-flavoured JSON Schema.

Speccy's `resolve` command solves these problems:

```
$ speccy resolve api/openapi.yml --json-schema
```

This turns multiple files into one megafile, and downgrades JSON Schema-proper to OpenAPI-flavoured JSON Schema in the process. Now the aggregator service has one format to work with, and it's all in one file, so we don't have to worry about how good the OpenAPI tooling is that's being used for the task at hand.

**API Server:** Our API Aggregator has a web API (meta!) and exposes a lot of this information for other aspects of the workflow.

```
[
  {
    "name": "Service Name",
    "slug": "service-name",
    "contact": {
      "team": "Some Team",
      "email": "some-team@example.com"
    },
    "openapi_url": "http://localhost:3000/specs/service-
name/openapi",
    "postman_url": "http://localhost:3000/specs/service-
name/postman-collection",
    "mock_server_url": "http://localhost:3000/mocks/service-
name/"
  },
  {
    ...
  }
]
```

The root of our API server contains links to all the other useful stuff it has to offer, so it can be navigated around without documentation (HATEOAS!). Needing documentation to find the documentation would just be... oww.

**Mock Server:** There are two different types of "mock server". What you usually see when talking about "mocks" is a HTTP server running static endpoints, which will respond with the same sort of example you see in your documentation when you send it a request that matches the path and method. There will usually be no validation, no way to trigger errors, no way to get dynamic values back, etc.

Then there's a sandbox, which is often a fairly realstic API instance which may well even have a database in the background, that saves values you send it, can trigger errors if your payload is invalid, and various other configurable things. These are really powerful, but we're ignoring them for now as they're quite advanced, and a lot of them require user interfaces, user sign in, which is another walled garden to think about paying for and integrating with.

Sticking to traditional "mocking", we're still deciding between Prism and APISprout. Prism has been around for a long time and is very powerful, but it's stuck waiting for one of its Go dependencies to get OAIv3 support. APISprout is very new and looks very promising, but we've had a few issues getting it to work.

We're focusing on other bits for now and will loop back in another month to finish up the mock server.

**SDK Generator:** Currently it's very common for every application contacting another API/service to build a "Connection" class. This class will usually be a HTTP wrapper, and it's essentially an SDK which the caller builds.

These are hand written, in every application, by different developers. The connection classes vary across applications, and they vary within the same application (one connection class will work differently to another in the same application). They often miss common error cases, ignore concepts like "Retry-After", do not help with validation errors, force clients to figure out appropriate timeouts, etc. It means we're all using the very bare minimums of HTTP and ignoring the most useful features.

A handful of teams have written a module/library to act as an SDK, but these take a lot of effort, are manually built and tested, and usually only available in one language (usually Ruby).

Due to the very low-level nature of working with the HTTP APIs, integrating another API into a codebase takes several days, when it should be much much quicker than that.

Using OpenAPI Generator (and our own custom templates) we've just finished a proof of concept gem, which is going into production as I write.

```
sdk = We::Sdk.new
user = sdk.apis.locations.get_building('abc-uuid')
puts user.data # { pegasus: 123, name: Adam Neumann, ... }

user = sdk.apis.memberships.get_user_memberships(user_uuid:
123, status_scope: 'active')
puts user.data # [ ... ]
```

The `.data` property is an abstraction on `.body` which could contain a string of JSON:API data or our own propietary data format.

```
def data
```

```
  if headers['content-type'] =~ /^application\/vnd.api
\+json/
    check_json_api_response
  else
    check_we_response
  end
end
```

Seeing as we've only recently made the JSON:API recommendation, it's really handy that we can roll out code which dynamically supports both. It's also baked in faraday-sunset, so more teams will start to notice deprecation warnings showing up in their logs (and since faraday-sunset v0.2.0 things can also get reported to Rollbar).

Over time we can programatically add other things like HTTP caching and circuit breakers and enable them one API at a time if we so want, and thanks to Dependabot we can have reasonable expectations of updates happening within a decent timeframe. We also have an async interface being added later which will allow for HTTP/1.1 (and later HTTP/2) calls to be executed in a batch. We'll see what parts of this we can open-source too of course.

**Postman Mirroring:** We used to have teams that would use Postman for documentation, but Postman documentation is very limited. It's basically a bookmark collection in HTML hosted on their website, and it doesn't mention anything about the actual body of the request/response unless you specifically copy and paste some JSON into their GUI. These examples may be entirely inaccurate and stagnate quickly.

Postman is an amazing HTTP client, and having shared collections is very useful, so we still use it! We convert our normalized OASv3 files to shared Postman Collections via Apimatic Transformer. Once it's given us a Postman Collection v2, we upload that file to the Postman API, and within seconds everyones Postman GUI will have updated to contain any new endpoints, improved examples, etc.

## API Reference Docs

Our documentation hub initially had to do all the normalization and resolution itself, but thanks to the API Aggregator it now just has a

single Rakefile which looks at the API Server, and passes each `openapi_url` through ReDoc. The static HTML is put in the `public/` folder along with all our other technical documentation, and job done.

## E2E Acceptance Testing

We have a fantastic cross-API testing suite, which is similar in functionality to Stoplight: Scenarios. API's don't exist in isolation, clients talk to them and APIs talk to other APIs, so you need a test suite that exists outside of the one APIs repo.

This end-to-end test suite runs periodicially and on build, leveraging docker compose to spin up and dependencies it requires. Built using RSpec and Faraday, it's easy for us to add middlewares into these interactions. One middleware we cooked up in a hackathon is faraday-openapi (we'll open-source this as soon as we've got it more fleshed out!), and it basically just looks at the provided OpenAPI spec, checks the request and response that comes through it, then raises an exception on failure.

You'd probably never want to enable this in production, but in a test suite this is fantastic! Now instead of relying on each and every team to enable contract testing (to confirm that both their code is doing what the specs say, and their specs are correct to the code), we can enable this automatically for any API in the E2E suite.

## The Future

OpenAPI and JSON Schema are fantastic tools and the tooling is only getting stronger. Every week it feels like a new tool pops up that makes me improve another section of the workflow, and we've not even got into server generation yet. Meetup have done some really amazing work in this space and we're looking to take what they've done and further it over time.

The API Aggregator is being wrapped up in Docker to make it something we can open-source, so you too can have mocks, docs, SDKs and Postman mirroring, all spawned from your own Git repos with very little work. We figure we've spent a lot of time working all this out, no reason why you should too.

## We're Hiring

WeWork is hiring all over the world and we have a lot of our tech people in SF, NYC and Tel-Aviv. If you would like to specifically work on the team which handles this sort of thing ☝ (we call it Engineering Architecture!) reach out to me, Phil Sturgeon, and we'll talk about it!