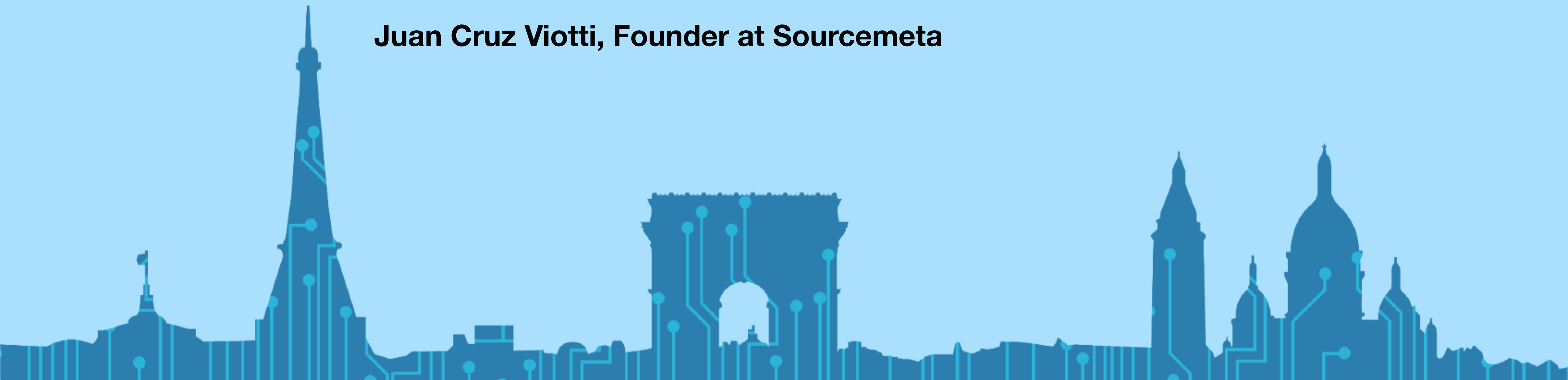




Applying Software Engineering Practices to Schemas

The JSON-Schema-first approach to API specifications

Juan Cruz Viotti, Founder at Sourcemeta



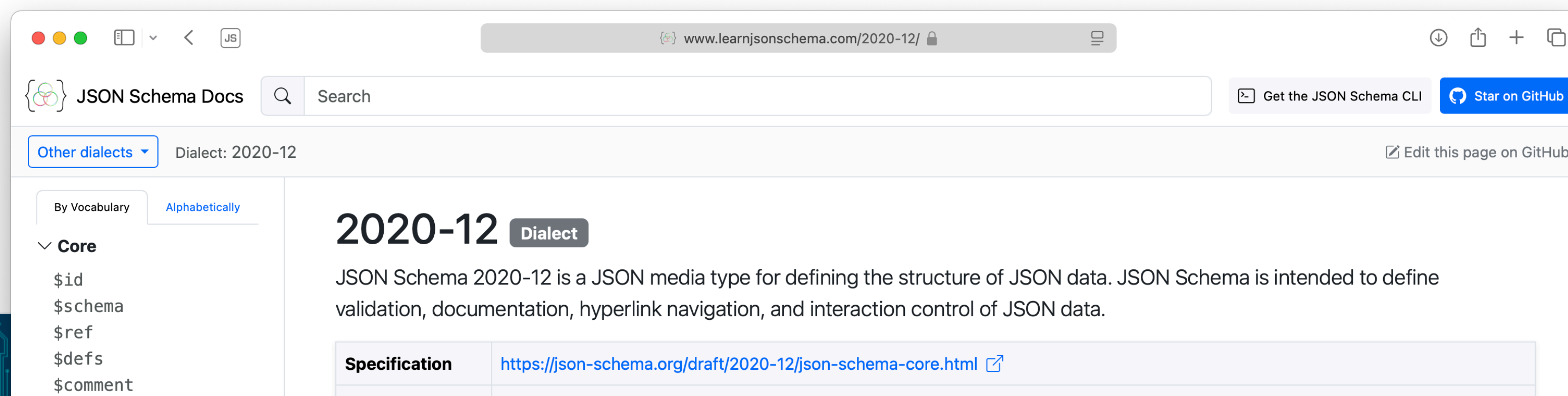
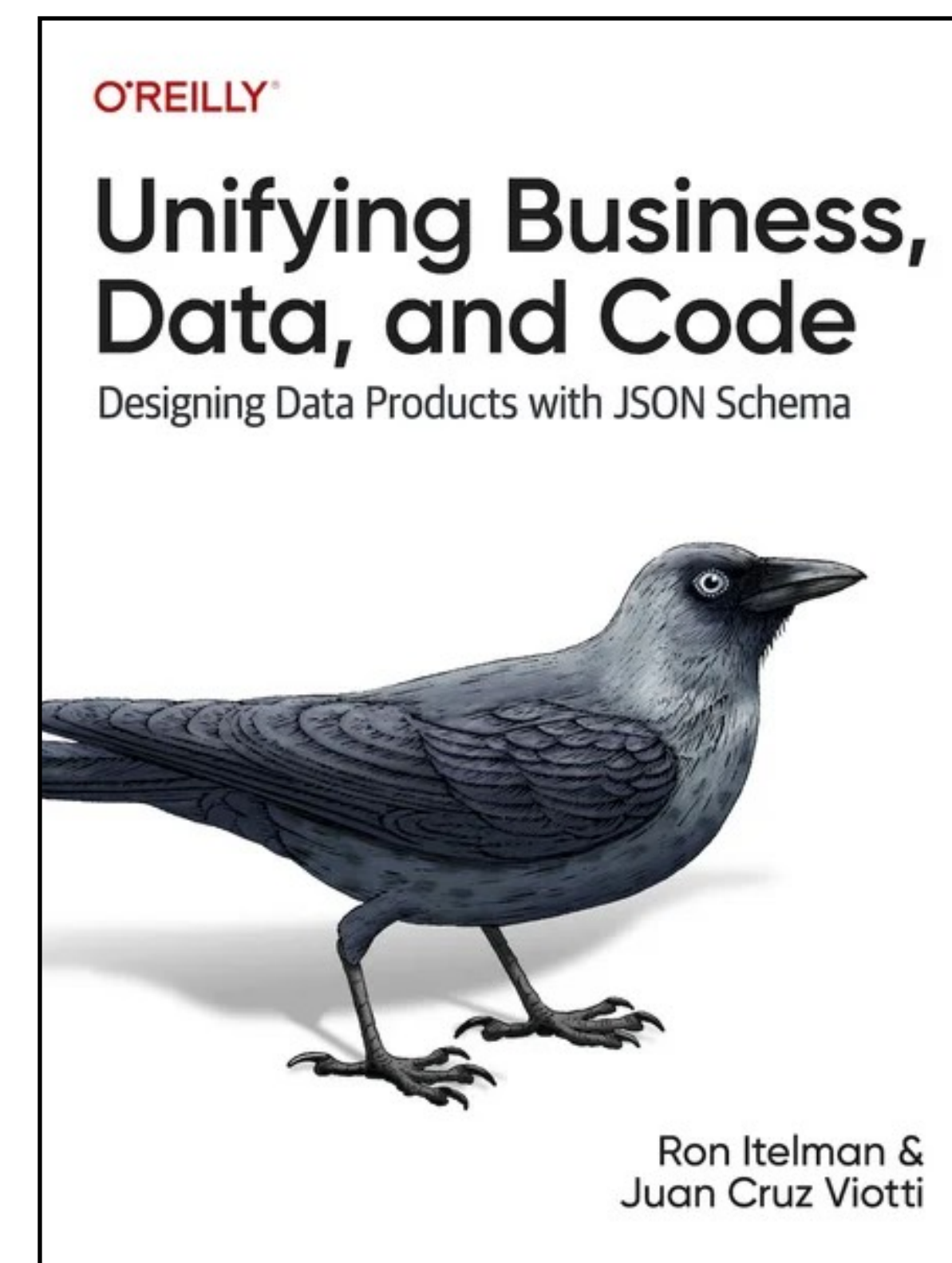


About me: Juan Cruz Viotti

Founder / Consultant at Sourcemeta



- TSC member of JSON Schema
- O'Reilly author on the topic of JSON Schema for data science
- Award-winning research at the University of Oxford on JSON Schema
- Author of various JSON Schema tooling, such as [LearnJSONSchema.com](https://learnjsonschema.com) and [AlterSchema](https://github.com/sourcemeta/alter-schema)



API Specifications

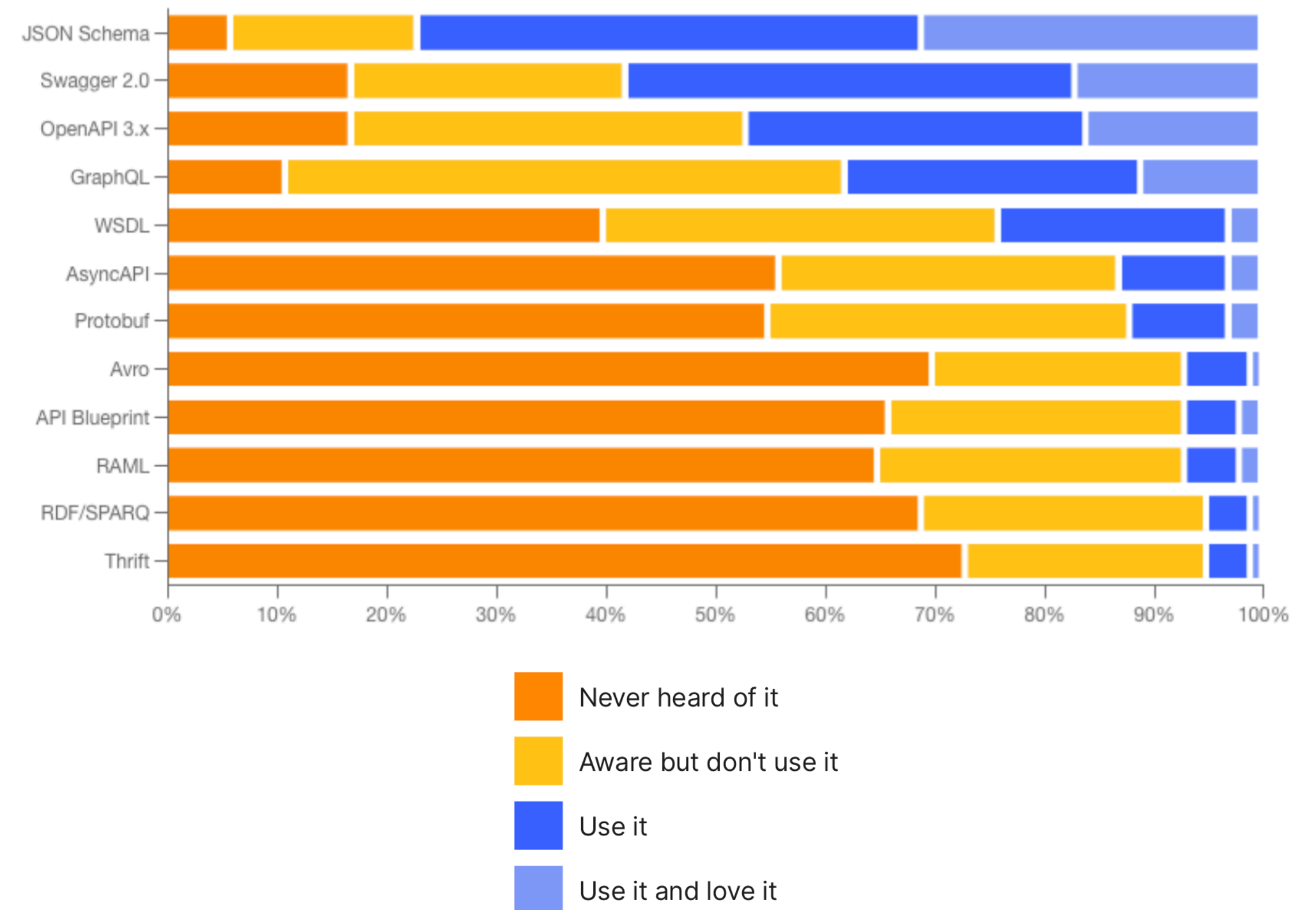
Human/Machine readable descriptions of APIs

- Single source of truth
- Standardised documentation
- Code generation
- Improved testing / validation
- Increased discoverability

API Specifications

Human/Machine readable descriptions of APIs

- Single source of truth
- Standardised documentation
- Code generation
- Improved testing / validation
- Increased discoverability

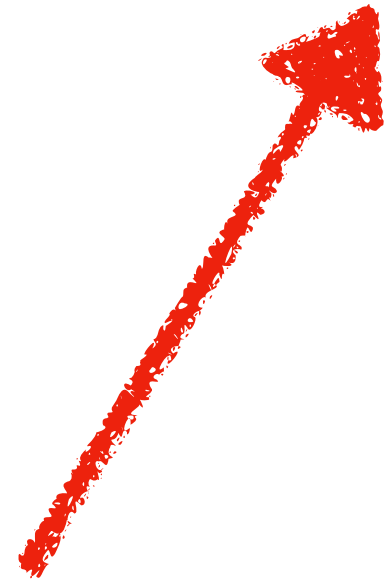


Source: Postman 2023 State of API Report

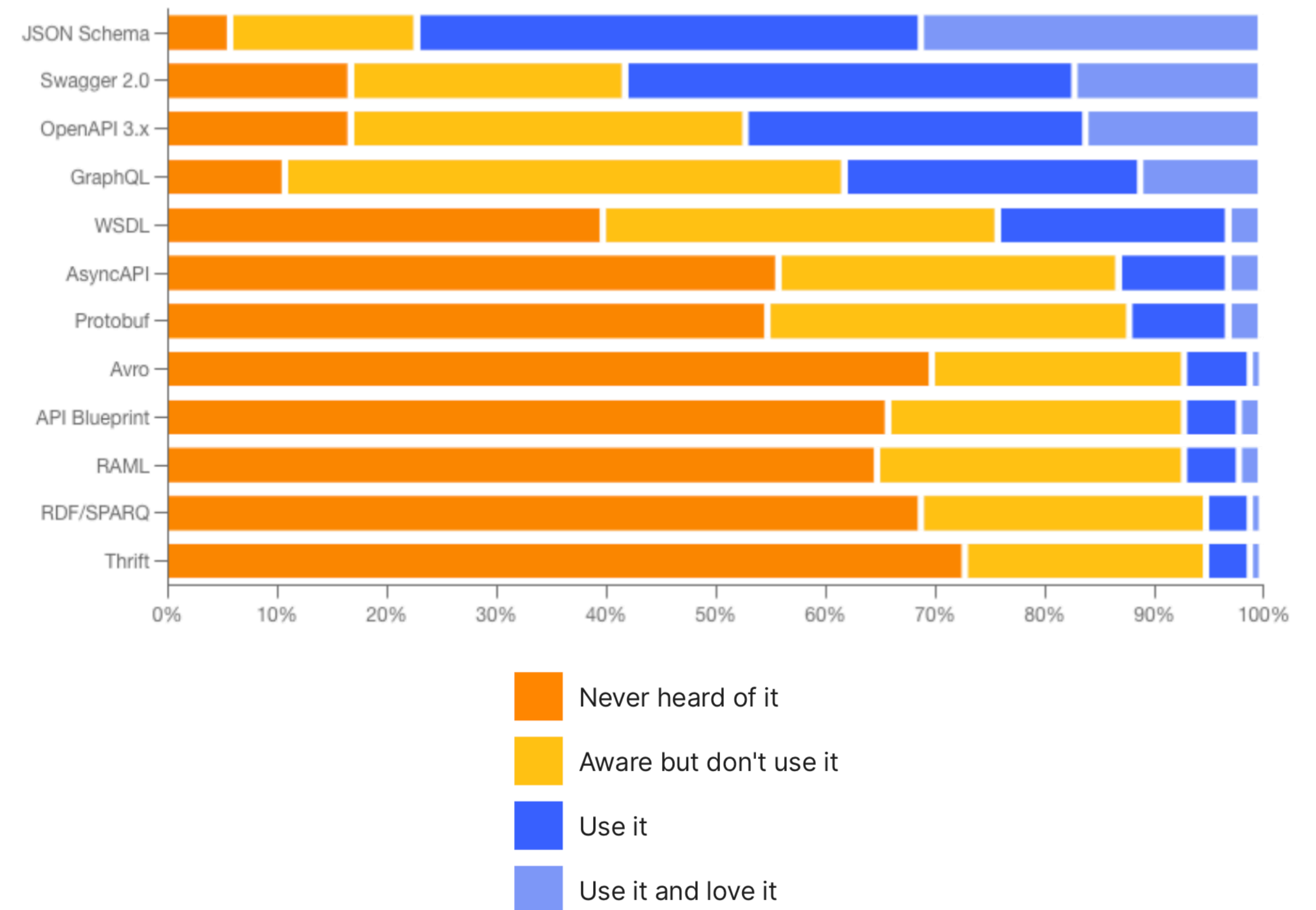
API Specifications

Human/Machine readable descriptions of APIs

- Single source of truth
- Standardised documentation
- Code generation
- Improved testing / validation
- Increased discoverability



Game Changing!

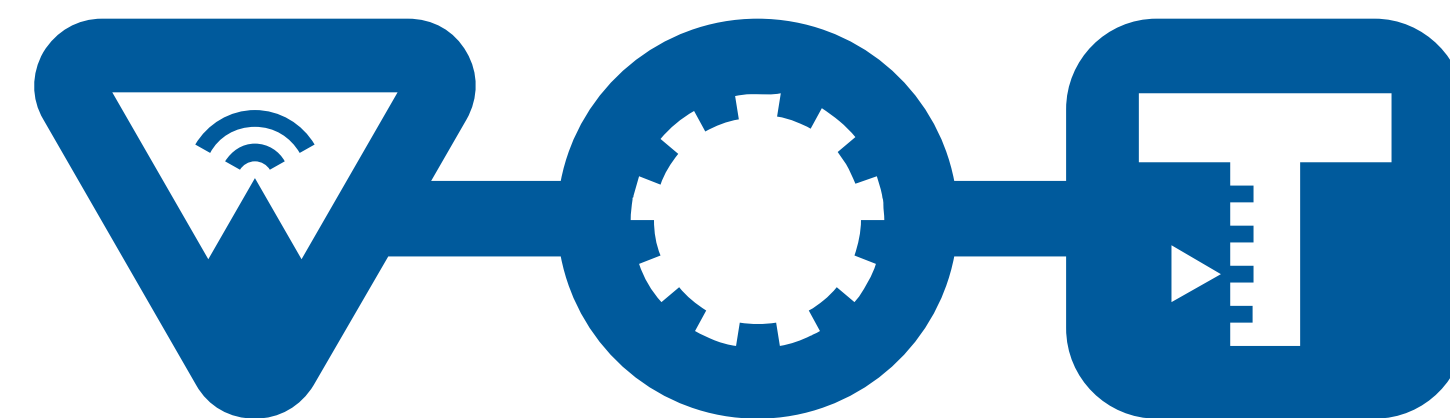


Source: Postman 2023 State of API Report

Are you using any of these?

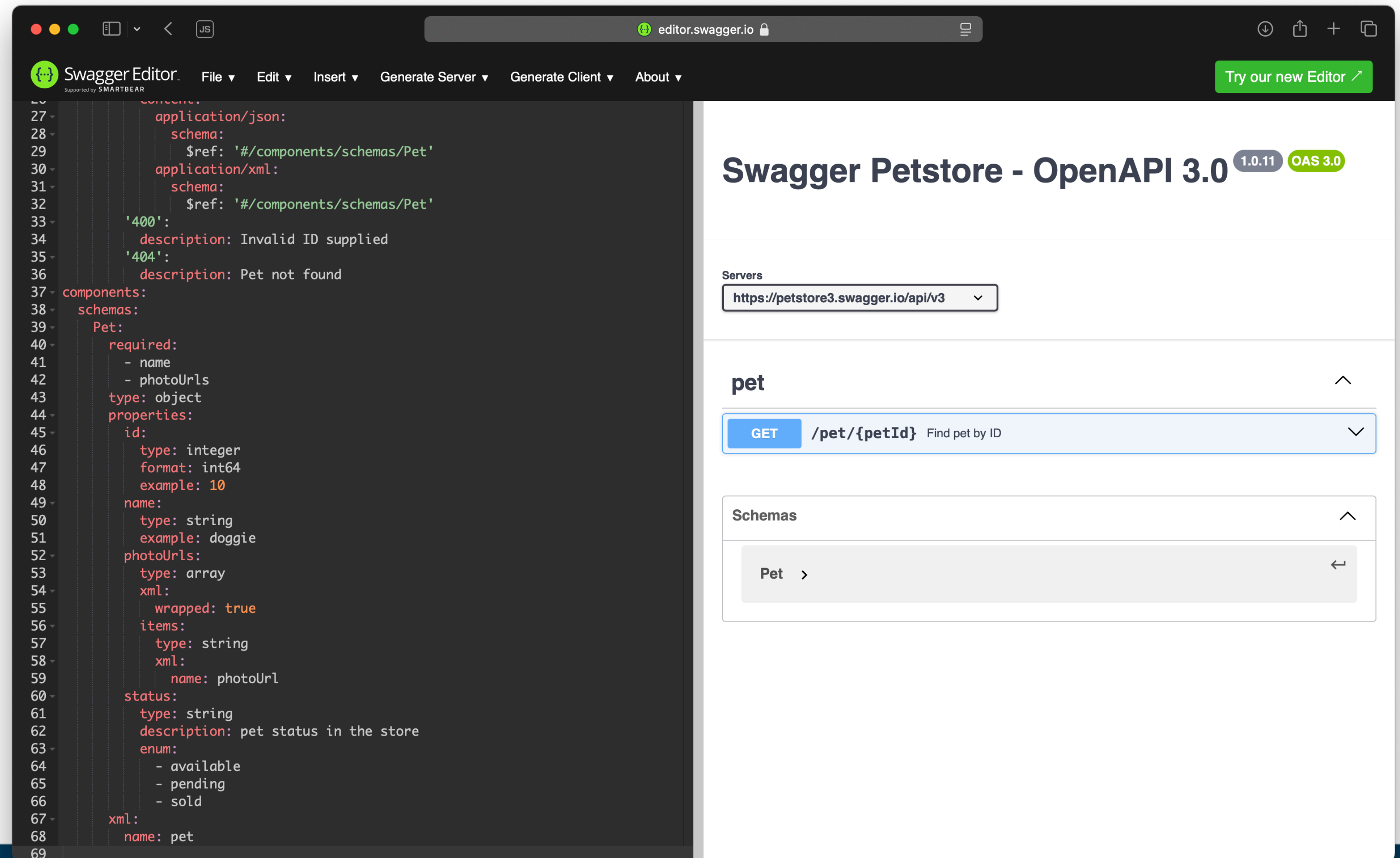


RAML



The OpenAPI Journey: Episode 1

The glamorous life of an API developer



The image shows a screenshot of the Swagger Editor web application. The interface is split into two main sections: a code editor on the left and a rendered API documentation page on the right.

Code Editor (Left): Displays OpenAPI 3.0 JSON code. The visible portion includes:

```
27     application/json:
28       schema:
29         $ref: '#/components/schemas/Pet'
30     application/xml:
31       schema:
32         $ref: '#/components/schemas/Pet'
33     '400':
34       description: Invalid ID supplied
35     '404':
36       description: Pet not found
37 components:
38   schemas:
39     Pet:
40       required:
41         - name
42         - photoUrls
43       type: object
44       properties:
45         id:
46           type: integer
47           format: int64
48           example: 10
49         name:
50           type: string
51           example: doggie
52         photoUrls:
53           type: array
54           xml:
55             wrapped: true
56           items:
57             type: string
58             xml:
59               name: photoUrl
60         status:
61           type: string
62           description: pet status in the store
63           enum:
64             - available
65             - pending
66             - sold
67       xml:
68         name: pet
69
```

Rendered API Documentation (Right): Shows the Swagger Petstore - OpenAPI 3.0 interface. The version is 1.0.11, and it is OAS 3.0. The server URL is set to `https://petstore3.swagger.io/api/v3`. The `pet` endpoint is highlighted, showing a `GET` method for `/pet/{petId}` with the description "Find pet by ID". The Schemas section shows the `Pet` schema.

The OpenAPI Journey: Episode 1

The glamorous life of an API developer

A declarative definition of my main API endpoint in just 68 lines of YAML



The screenshot shows the Swagger Editor interface. On the left, a code editor displays the OpenAPI 3.0 definition in YAML. The definition includes a 'components' section with a 'Pet' schema. The 'Pet' schema is an object with required fields 'name' and 'photoUrls', and optional fields 'id', 'status', and 'xml'. The 'id' field is an integer with format 'int64' and example '10'. The 'status' field is a string with an enum of 'available', 'pending', and 'sold'. The 'xml' field is a boolean with name 'pet'. On the right, the Swagger UI preview shows the 'Swagger Petstore - OpenAPI 3.0' title, a server dropdown set to 'https://petstore3.swagger.io/api/v3', and a 'pet' endpoint with a 'GET' method and path '/pet/{petId}'. The endpoint description is 'Find pet by ID'. Below the endpoint, there is a 'Schemas' section with a 'Pet' schema link.

The OpenAPI Journey: Episode 2

The good life of an API developer

The image shows a screenshot of the Swagger Editor web application. The interface is split into two main sections: a code editor on the left and a rendered API specification on the right.

Code Editor (Left): Displays OpenAPI 3.0 JSON/YAML code for a Pet resource. The code includes properties like `name` (string), `id` (integer), `photoUrls` (array), and `status` (enum). It also includes XML tags and references to other schemas like `Category` and `Tag`.

```
208     name:
209       type: string
210     xml:
211       name: tag
212   Pet:
213     required:
214       - name
215       - photoUrls
216     type: object
217     properties:
218       id:
219         type: integer
220         format: int64
221         example: 10
222       name:
223         type: string
224         example: doggie
225       category:
226         $ref: '#/components/schemas/Category'
227       photoUrls:
228         type: array
229         xml:
230           wrapped: true
231         items:
232           type: string
233           xml:
234             name: photoUrl
235       tags:
236         type: array
237         xml:
238           wrapped: true
239         items:
240           $ref: '#/components/schemas/Tag'
241       status:
242         type: string
243         description: pet status in the store
244         enum:
245           - available
246           - pending
247           - sold
248     xml:
249       name: pet
250
```

Rendered API Specification (Right): Shows the Swagger Petstore - OpenAPI 3.0 specification. The version is 1.0.11, and it supports OAS 3.0. The server URL is `https://petstore3.swagger.io/api/v3`.

API Endpoints:

- PUT /pet**: Update an existing pet
- POST /pet**: Add a new pet to the store
- GET /pet/{petId}**: Find pet by ID
- POST /pet/{petId}**: Updates a pet in the store with form data
- DELETE /pet/{petId}**: Deletes a pet

Schemas:

- Category**: Expandable schema section.

The OpenAPI Journey: Episode 2

The good life of an API developer

CRUD operations of my most important resource is *just* 249 lines of YAML

The screenshot shows the Swagger Editor interface. On the left, a code editor displays the OpenAPI 3.0 specification in YAML format. The code is highlighted in a dark theme. On the right, the Swagger UI preview shows the API's structure. The top of the preview displays 'Swagger Petstore - OpenAPI 3.0' with version '1.0.11' and 'OAS 3.0'. Below this, there is a 'Servers' section with a dropdown menu set to 'https://petstore3.swagger.io/api/v3'. The main part of the preview shows a list of API endpoints under the 'pet' resource. The endpoints are: PUT /pet (Update an existing pet), POST /pet (Add a new pet to the store), GET /pet/{petId} (Find pet by ID), POST /pet/{petId} (Updates a pet in the store with form data), and DELETE /pet/{petId} (Deletes a pet). Below the endpoints, there is a 'Schemas' section with a dropdown menu showing 'Category'. The code editor on the left shows the following YAML snippet for the 'Pet' schema:

```
208     name:
209       type: string
210     xml:
211       name: tag
212   Pet:
213     required:
214       - name
215       - photoUrls
216     type: object
217     properties:
218       id:
219         type: integer
220         format: int64
221         example: 10
222       name:
223         type: string
224         example: doggie
225       category:
226         $ref: '#/components/schemas/Category'
227       photoUrls:
228         type: array
229         xml:
230           wrapped: true
231         items:
232           type: string
233           xml:
234             name: photoUrl
235       tags:
236         type: array
237         xml:
238           wrapped: true
239         items:
240           $ref: '#/components/schemas/Tag'
241       status:
242         type: string
243         description: pet status in the store
244         enum:
245           - available
246           - pending
247           - sold
248     xml:
249       name: pet
250
```

The OpenAPI Journey: Episode 3

The decent life of an API developer

The screenshot displays the Swagger Editor interface. On the left, a code editor shows the OpenAPI 3.0 definition for the Swagger Petstore API. The code is as follows:

```
764     ApiResponse:
765     type: object
766     properties:
767     code:
768     type: integer
769     format: int32
770     type:
771     type: string
772     message:
773     type: string
774     xml:
775     name: '##default'
776 requestBodies:
777 Pet:
778 description: Pet object that needs to be added to the store
779 content:
780 application/json:
781 schema:
782 $ref: '#/components/schemas/Pet'
783 application/xml:
784 schema:
785 $ref: '#/components/schemas/Pet'
786 UserArray:
787 description: List of user object
788 content:
789 application/json:
790 schema:
791 type: array
792 items:
793 $ref: '#/components/schemas/User'
794 securitySchemes:
795 petstore_auth:
796 type: oauth2
797 flows:
798 implicit:
799 authorizationUrl: https://petstore3.swagger.io/oauth/authorize
800 scopes:
801 write:pets: modify pets in your account
802 read:pets: read your pets
803 api_key:
804 type: apiKey
805 name: api_key
806 in: header
```

On the right, the Swagger UI displays the API documentation for "Swagger Petstore - OpenAPI 3.0" (version 1.0.11). The page includes a description of the API, a list of useful links, and a "Servers" section with a dropdown menu showing "https://petstore3.swagger.io/api/v3" and an "Authorize" button. Below this, the "pet" endpoint is listed with a "Find out more" link. The endpoints are:

- PUT /pet** Update an existing pet
- POST /pet** Add a new pet to the store

The OpenAPI Journey: Episode 3

The decent life of an API developer

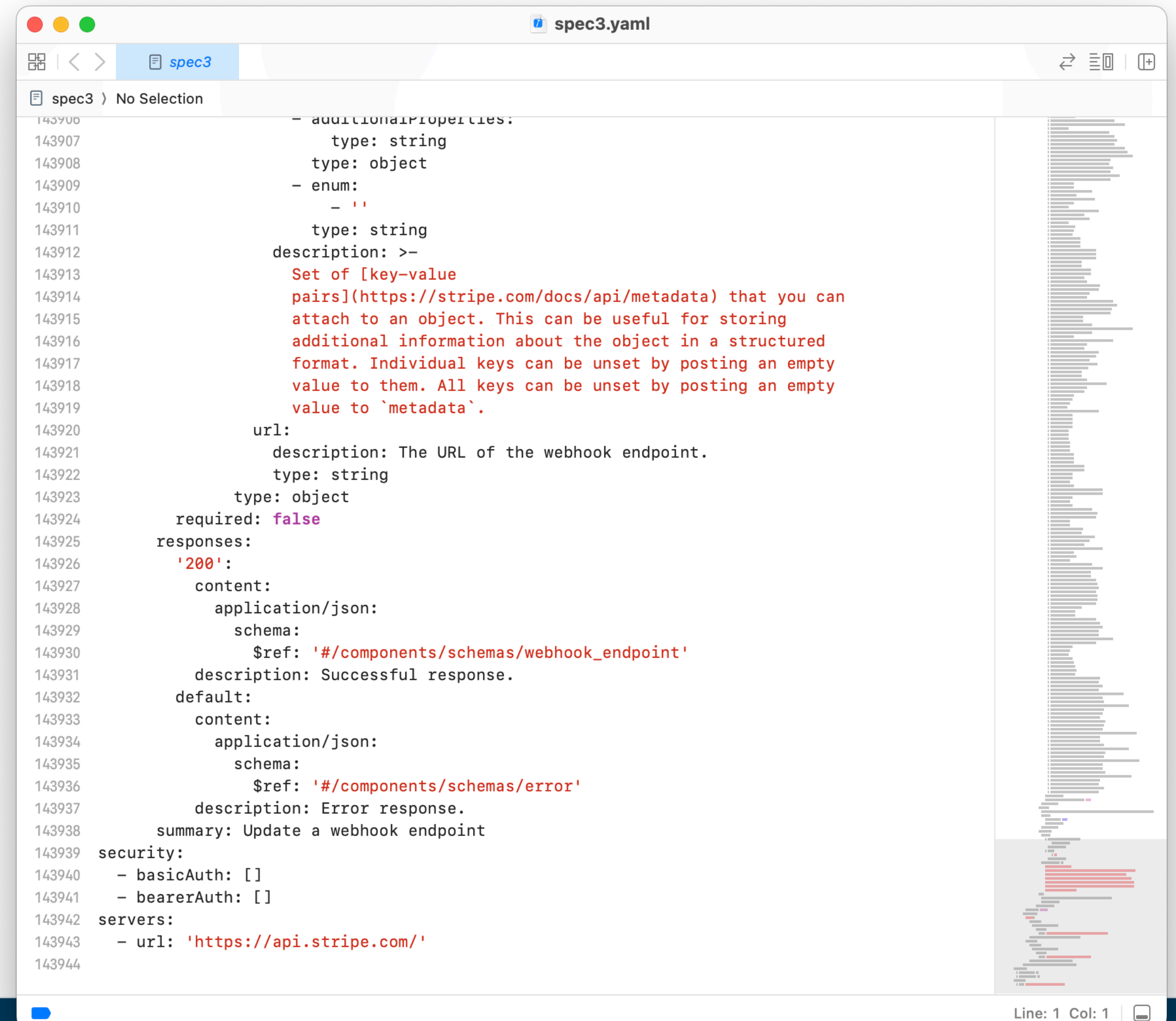
A sample real-world declarative definition of an API in *just* 806 lines of YAML



The screenshot shows the Swagger Editor web application. The left pane displays the OpenAPI 3.0 definition in YAML format, with line numbers 764 through 806 visible. The right pane shows the rendered API documentation for 'Swagger Petstore - OpenAPI 3.0', version 1.0.11. The documentation includes a description of the Pet Store Server, useful links, terms of service, and a list of API endpoints. The 'pet' endpoint is expanded to show two methods: 'PUT /pet' (Update an existing pet) and 'POST /pet' (Add a new pet to the store). The 'api_key' section at the bottom of the code editor is highlighted with a red box, corresponding to the red arrow from the text on the left.

The OpenAPI Journey: Final Episode

The “unless-there-is-tooling-I-will-switch-careers” API developer



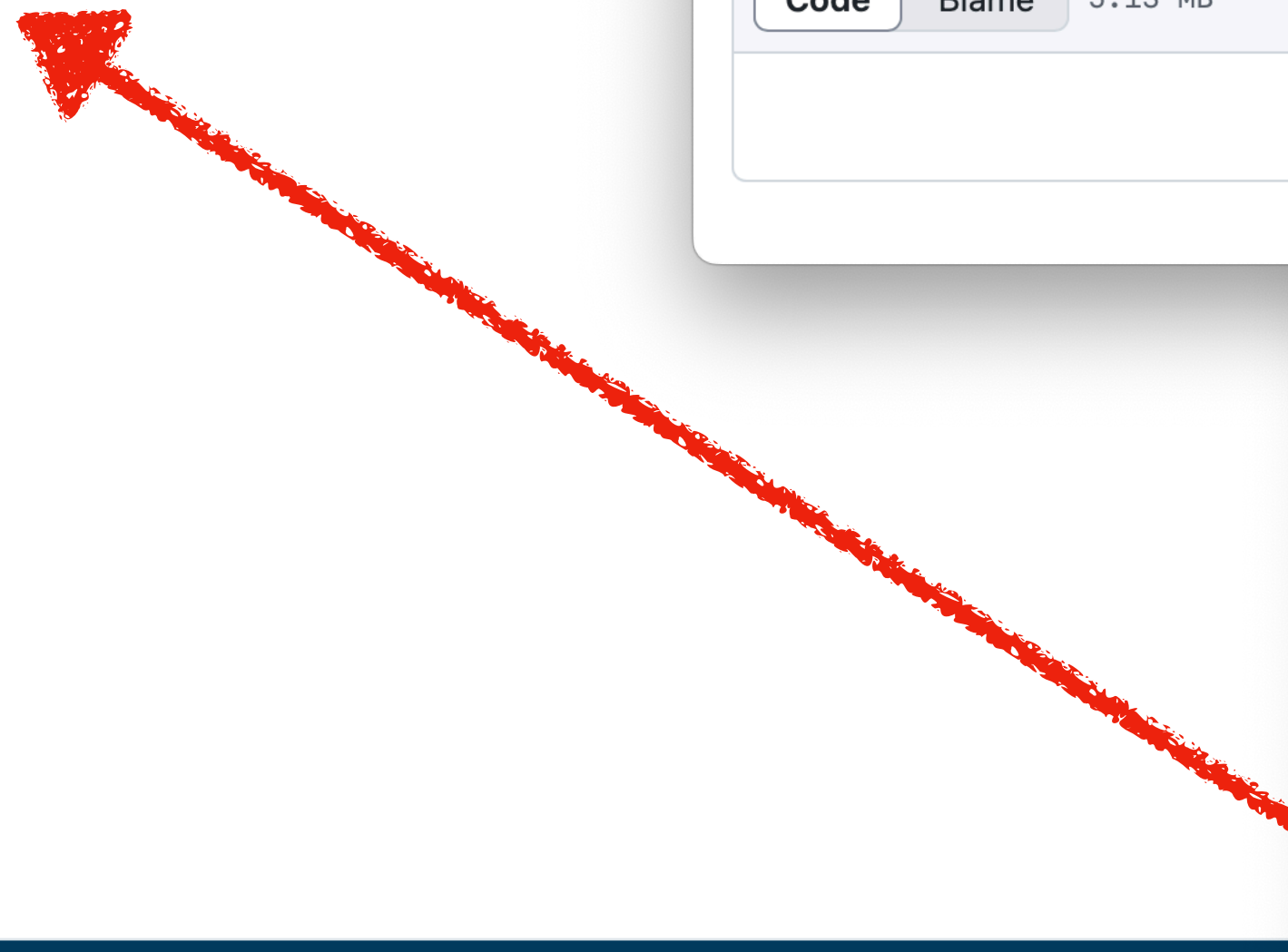
```
spec3.yaml
spec3
spec3 > No Selection
143906         - additionalProperties:
143907             type: string
143908             type: object
143909         - enum:
143910             - ''
143911             type: string
143912         description: >-
143913             Set of [key-value
143914             pairs](https://stripe.com/docs/api/metadata) that you can
143915             attach to an object. This can be useful for storing
143916             additional information about the object in a structured
143917             format. Individual keys can be unset by posting an empty
143918             value to them. All keys can be unset by posting an empty
143919             value to `metadata`.
143920         url:
143921             description: The URL of the webhook endpoint.
143922             type: string
143923         type: object
143924         required: false
143925     responses:
143926         '200':
143927             content:
143928                 application/json:
143929                     schema:
143930                         $ref: '#/components/schemas/webhook_endpoint'
143931             description: Successful response.
143932         default:
143933             content:
143934                 application/json:
143935                     schema:
143936                         $ref: '#/components/schemas/error'
143937             description: Error response.
143938         summary: Update a webhook endpoint
143939     security:
143940         - basicAuth: []
143941         - bearerAuth: []
143942     servers:
143943         - url: 'https://api.stripe.com/'
143944
```

Line: 1 Col: 1

The OpenAPI Journey: Final Episode

The “unless-there-is-tooling-I-will-switch-careers” API developer

A (great!) real-world production-ready declarative definition of an API in **JUST 143943 lines of YAML** that GitHub will refuse to preview

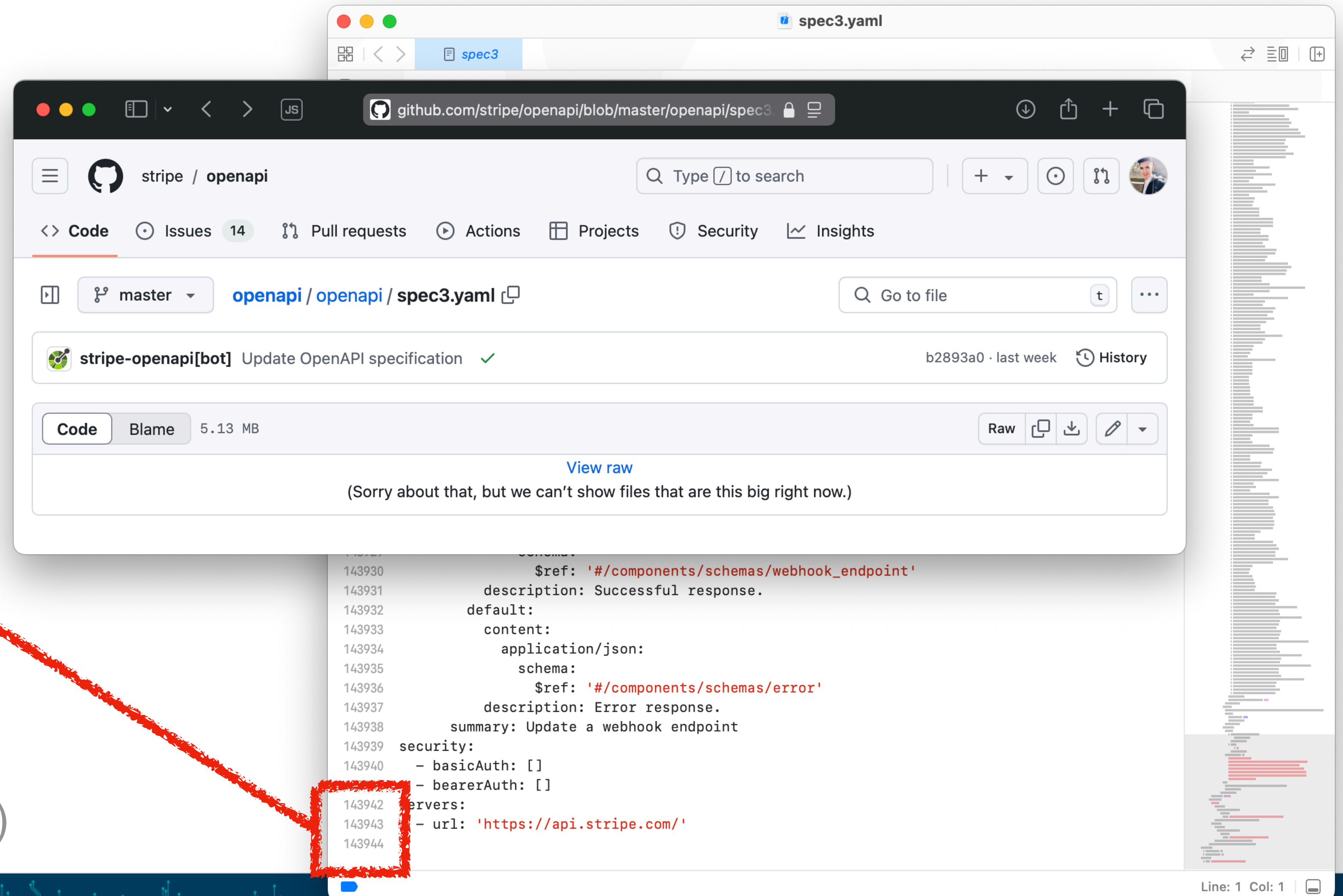
The image shows a screenshot of a GitHub repository page for stripe/openapi. The browser address bar shows 'github.com/stripe/openapi/blob/master/openapi/spec3'. The page header includes the repository name and navigation links like Code, Issues, Pull requests, Actions, Projects, Security, and Insights. A commit message 'stripe-openapi[bot] Update OpenAPI specification' is visible. Below the commit, there are options for 'Code', 'Blame', and '5.13 MB'. A message states '(Sorry about that, but we can't show files that are this big right now.)'. A code editor view shows the end of a YAML file, with line numbers 143930 to 143944. A red-bordered box highlights line 143943, which contains the URL 'https://api.stripe.com/'. The status bar at the bottom right indicates 'Line: 1 Col: 1'.

```
143930           $ref: '#/components/schemas/webhook_endpoint'  
143931           description: Successful response.  
143932           default:  
143933             content:  
143934               application/json:  
143935                 schema:  
143936                   $ref: '#/components/schemas/error'  
143937             description: Error response.  
143938             summary: Update a webhook endpoint  
143939           security:  
143940             - basicAuth: []  
143941             - bearerAuth: []  
143942           servers:  
143943             - url: 'https://api.stripe.com/'  
143944
```

The OpenAPI Journey: Final Episode

The “unless-there-is-tooling-I-will-switch-careers” API developer

A (great!) real-world production-ready declarative definition of an API in **JUST 143943 lines of YAML** that GitHub will refuse to preview



(At companies like Stripe, there indeed seems to be proprietary tooling for generating/managing this)

KEY OBSERVATION

>80% of this is JSON Schema

Either as re-usable components or inlined in path definitions

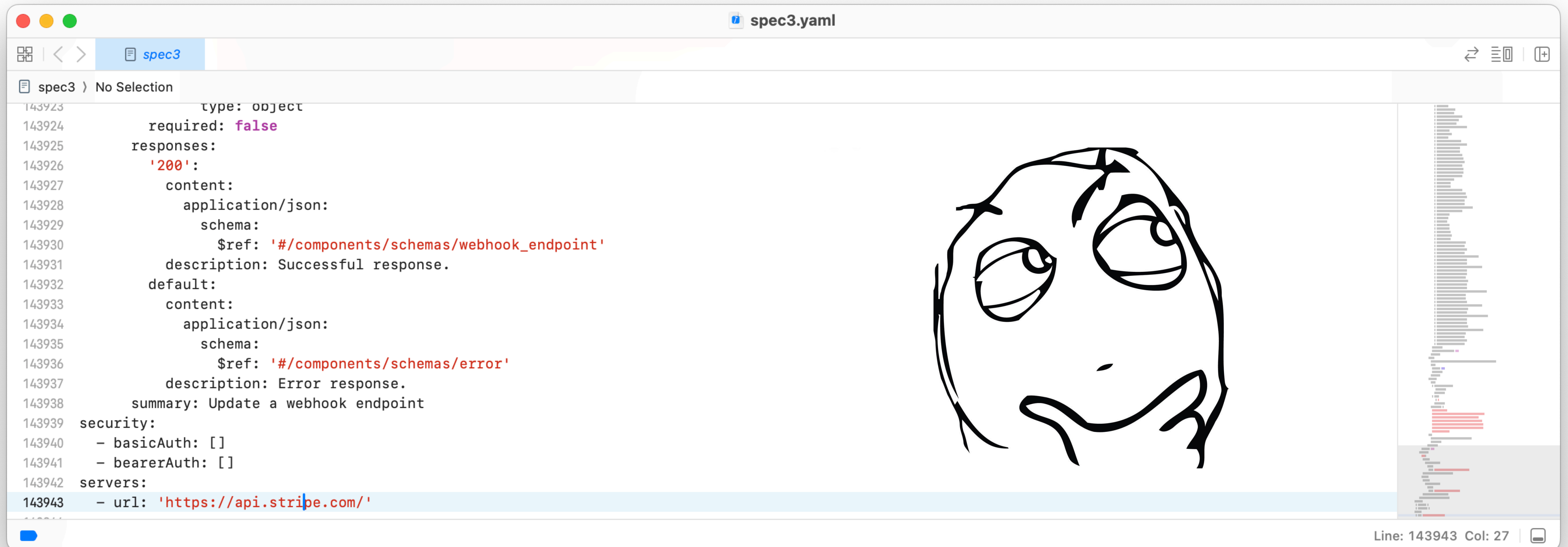
```
143923         type: object
143924         required: false
143925         responses:
143926           '200':
143927             content:
143928               application/json:
143929                 schema:
143930                   $ref: '#/components/schemas/webhook_endpoint'
143931             description: Successful response.
143932           default:
143933             content:
143934               application/json:
143935                 schema:
143936                   $ref: '#/components/schemas/error'
143937             description: Error response.
143938           summary: Update a webhook endpoint
143939         security:
143940           - basicAuth: []
143941           - bearerAuth: []
143942         servers:
143943           - url: 'https://api.stripe.com/'
```

Line: 143943 Col: 27

KEY OBSERVATION

>80% of this is JSON Schema

Either as re-usable components or inlined in path definitions

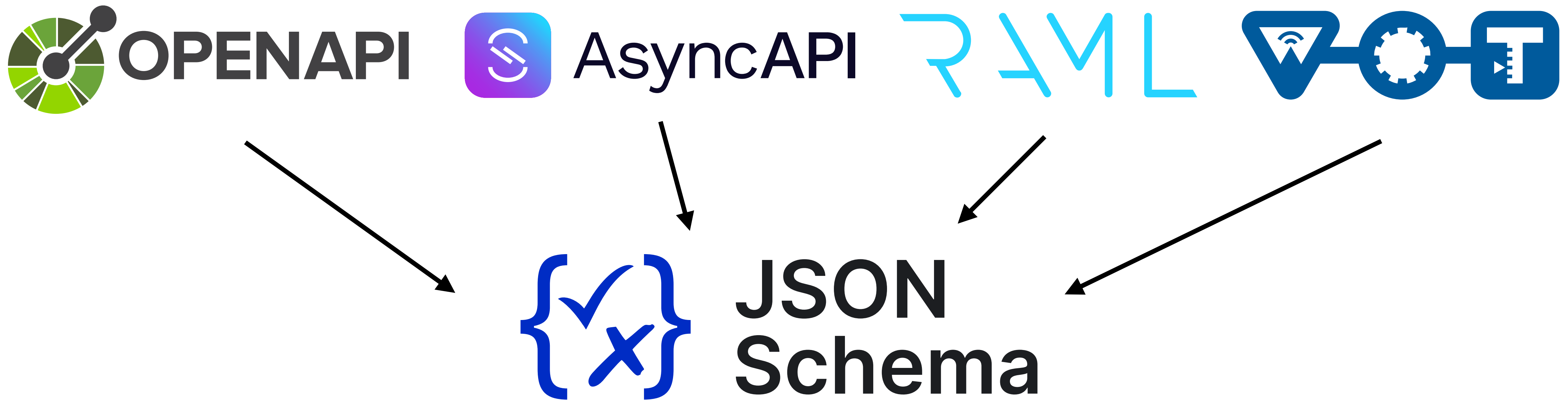


```
143923     type: object
143924     required: false
143925     responses:
143926       '200':
143927         content:
143928           application/json:
143929             schema:
143930               $ref: '#/components/schemas/webhook_endpoint'
143931         description: Successful response.
143932     default:
143933       content:
143934         application/json:
143935           schema:
143936             $ref: '#/components/schemas/error'
143937         description: Error response.
143938     summary: Update a webhook endpoint
143939     security:
143940       - basicAuth: []
143941       - bearerAuth: []
143942     servers:
143943       - url: 'https://api.stripe.com/'
```

Line: 143943 Col: 27

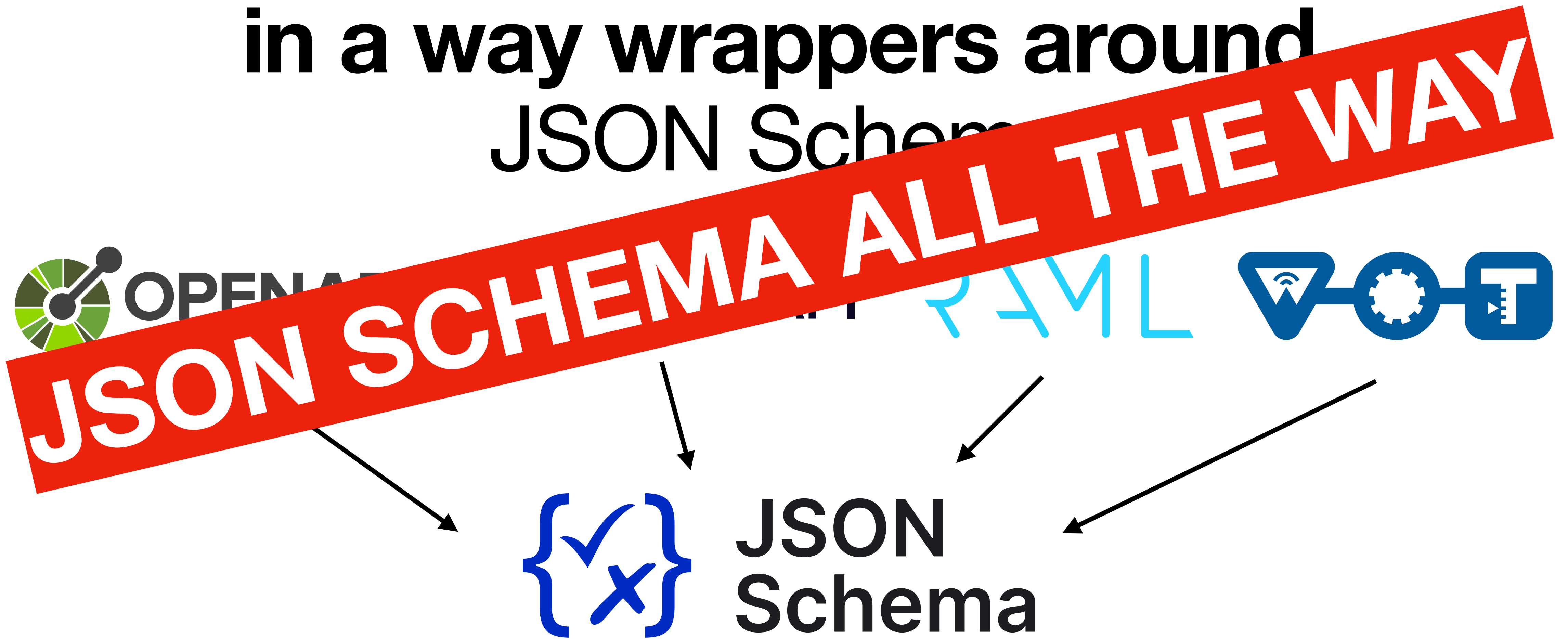
TAKING IT ONE STEP FURTHER...

**All of these specifications are
in a way wrappers around
JSON Schema**

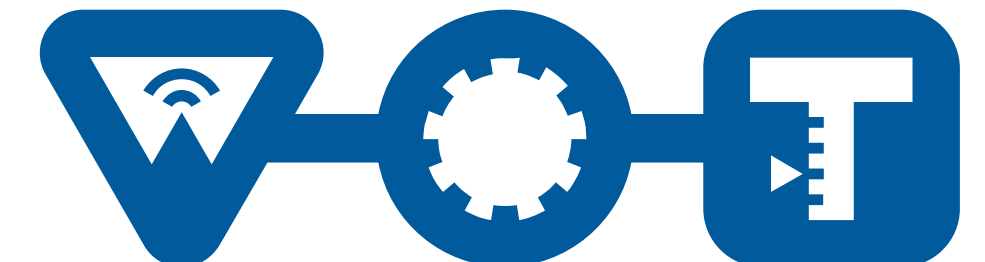


TAKING IT ONE STEP FURTHER...

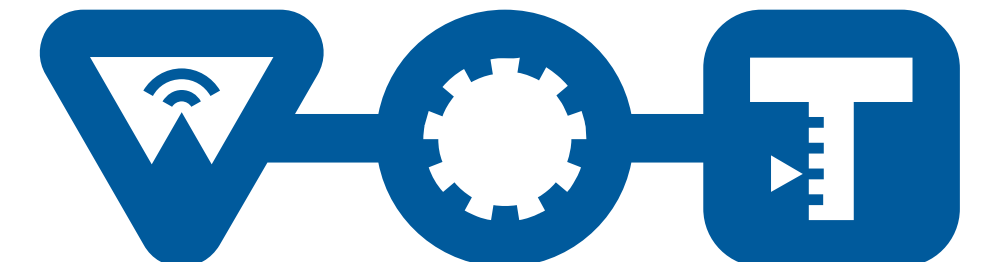
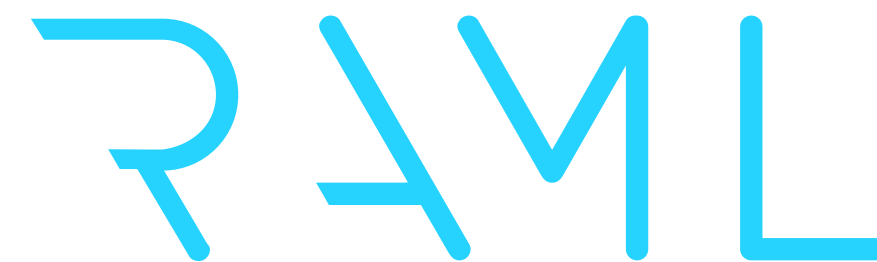
All of these specifications are
in a way wrappers around
JSON Schema



If you want to improve your API specifications...



If you want to improve your API specifications...



The best thing you can do is

Improve your JSON Schemas

The Most Common Pitfalls with JSON Schema

At least based on my consultancy experience!

The Most Common Pitfalls with JSON Schema

At least based on my consultancy experience!

- ✘ Defining entire ontologies of schemas within the API specification

The Most Common Pitfalls with JSON Schema

At least based on my consultancy experience!

- ✘ Defining entire ontologies of schemas within the API specification
- ✘ Copy pasting schemas in various API specifications because there is not a single place to reference them from

The Most Common Pitfalls with JSON Schema

At least based on my consultancy experience!

- ✘ Defining entire ontologies of schemas within the API specification
- ✘ Copy pasting schemas in various API specifications because there is not a single place to reference them from
- ✘ Schemas are not being unit tested at all

The Most Common Pitfalls with JSON Schema

At least based on my consultancy experience!

- ✘ Defining entire ontologies of schemas within the API specification
- ✘ Copy pasting schemas in various API specifications because there is not a single place to reference them from
- ✘ Schemas are not being unit tested at all
- ✘ Schemas are plain invalid, using wrong keywords, etc

The Most Common Pitfalls with JSON Schema

At least based on my consultancy experience!

- ✘ Defining entire ontologies of schemas within the API specification
- ✘ Copy pasting schemas in various API specifications because there is not a single place to reference them from
- ✘ Schemas are not being unit tested at all
- ✘ Schemas are plain invalid, using wrong keywords, etc
- ✘ Schemas are overly complicated from what they are supposed to match (bad practices, etc)

The Most Common Pitfalls with JSON Schema

At least based on my consultancy experience!

- ✘ Defining entire ontologies of schemas within the API specification
- ✘ Copy pasting schemas in various API specifications because there is not a single place to reference them from
- ✘ Schemas are not being unit tested at all
- ✘ Schemas are plain invalid, using wrong keywords, etc
- ✘ Schemas are overly complicated from what they are supposed to match (bad practices, etc)
- ✘ Relying on non-fully-compliant JSON Schema implementations

The JSON Schema first approach to API specifications



The JSON Schema first approach to API specifications

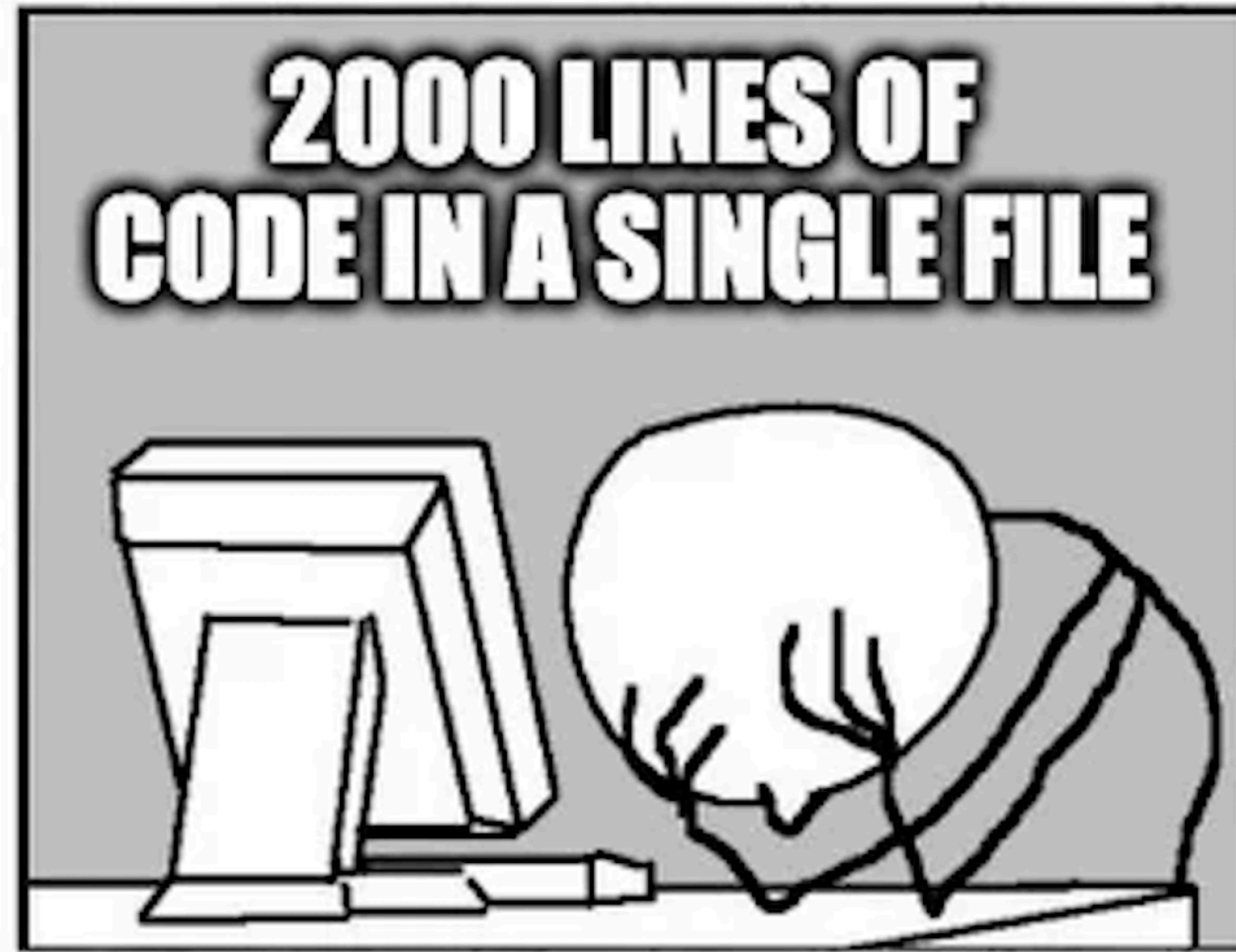
TLDR; Just treat you schemas as code. You already know all of this!



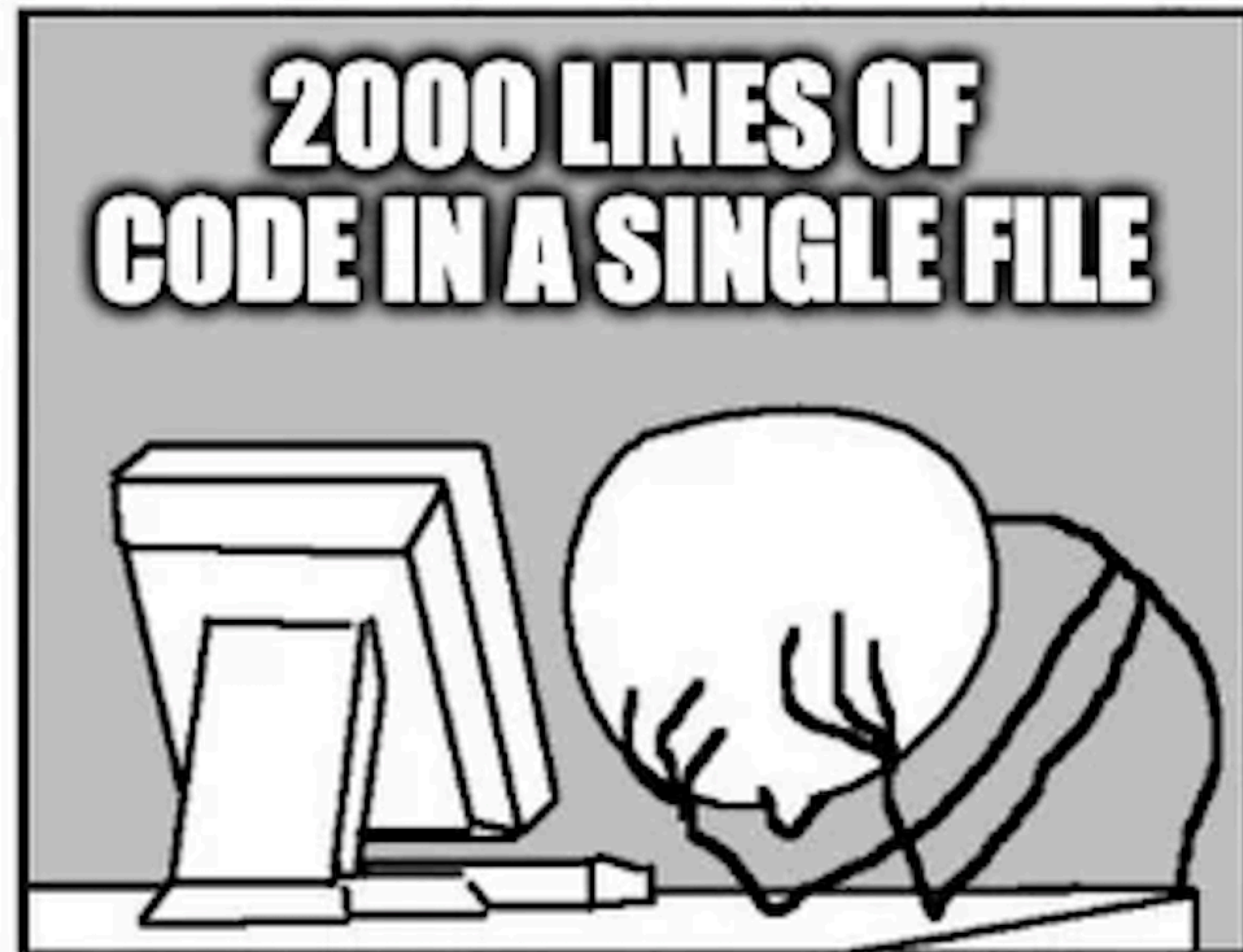
The JSON Schema first approach to API specifications

TLDR; Just treat you schemas as code. You already know all of this!





Surely you don't write all your projects as single huge code files

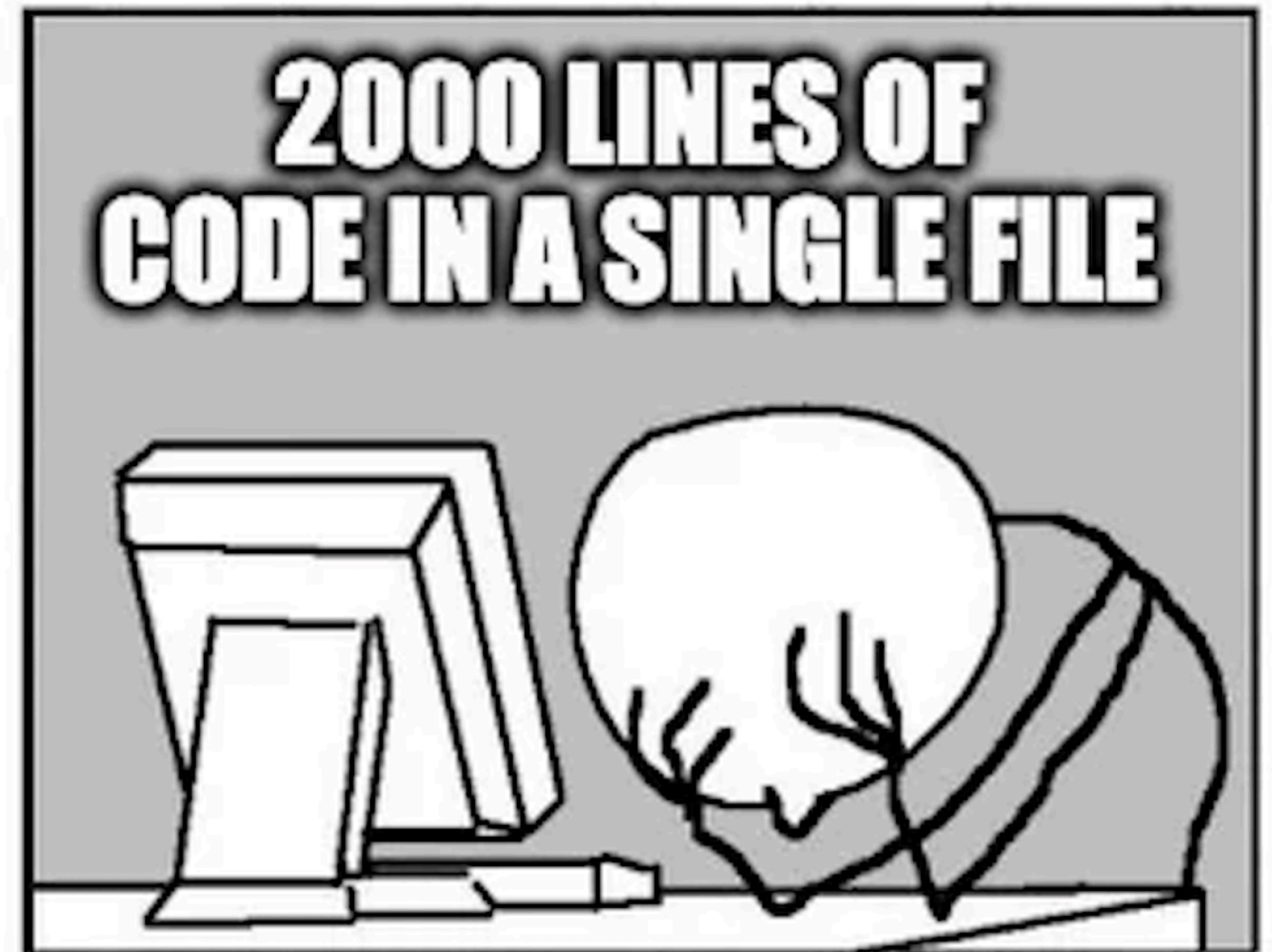


Surely you don't write all your projects as single huge code files

What about your OpenAPIs?

The JSON Schema first approach: Step #1

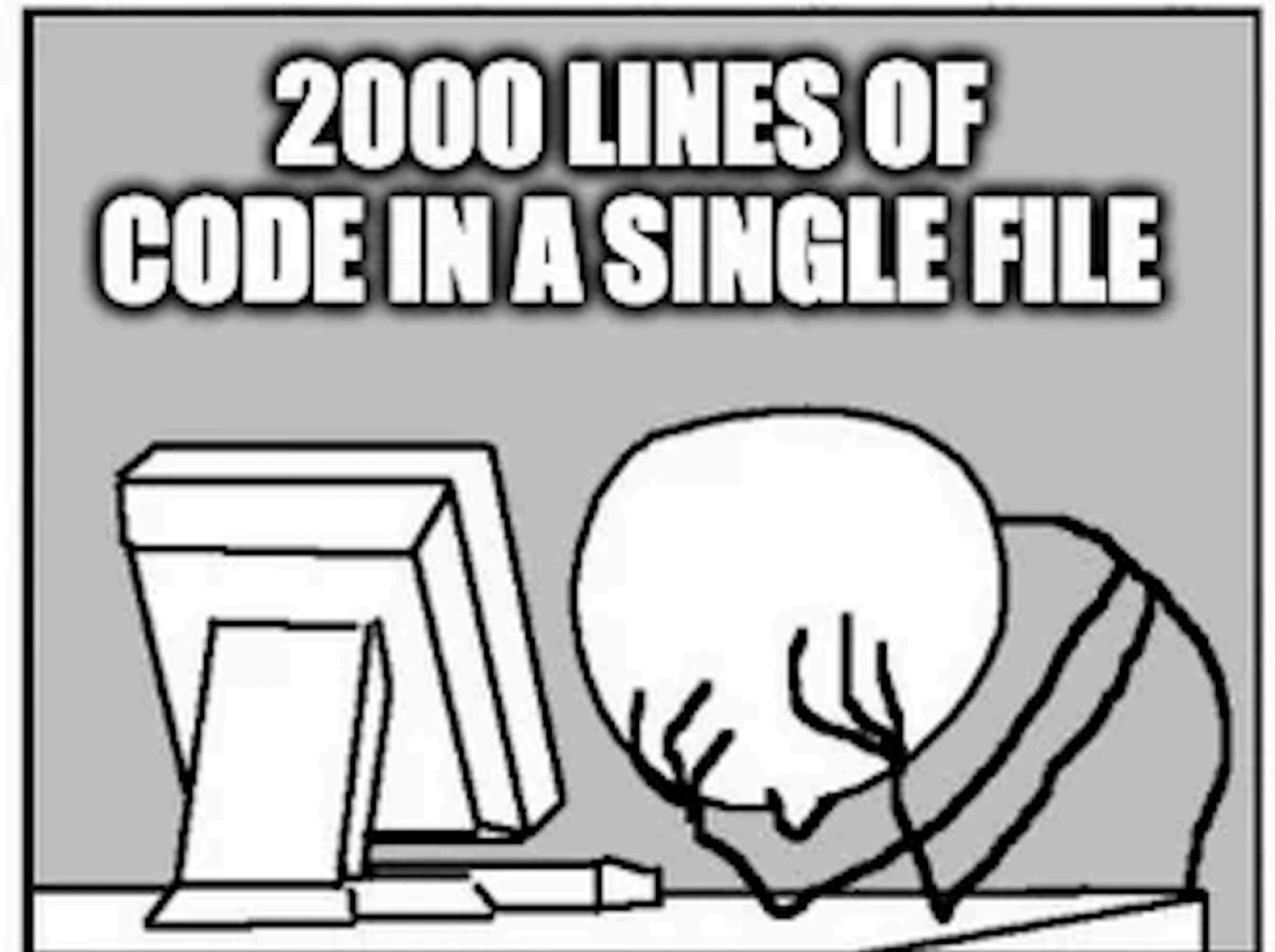
Extract your JSON Schemas as individual files on a GitHub repo



The JSON Schema first approach: Step #1

Extract your JSON Schemas as individual files on a GitHub repo

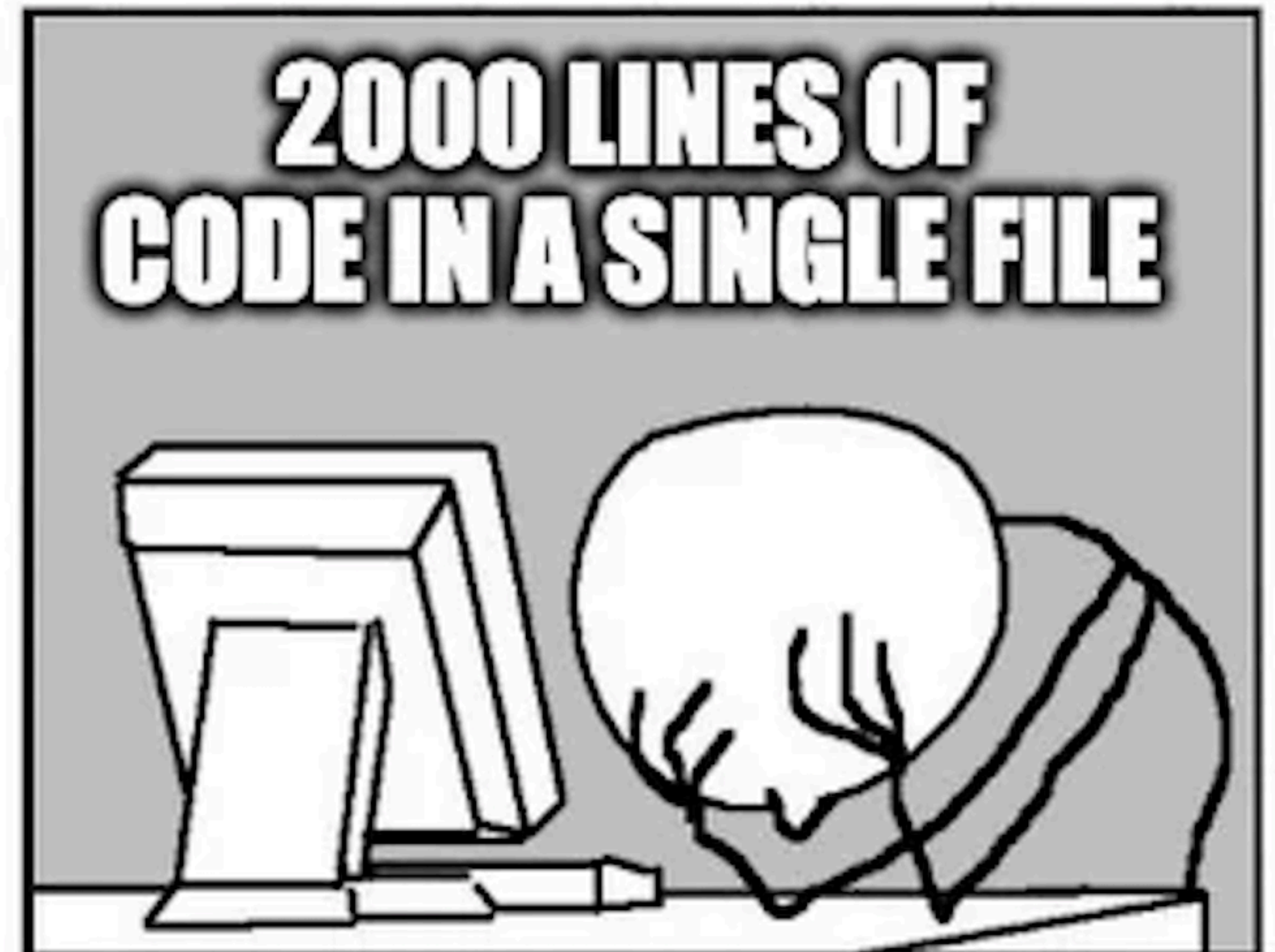
- Otherwise, its a lot harder to do everything within the constraints of i.e. an OpenAPI wrapper!



The JSON Schema first approach: Step #1

Extract your JSON Schemas as individual files on a GitHub repo

- Otherwise, its a lot harder to do everything within the constraints of i.e. an OpenAPI wrapper!
- And you can share the same schemas with more than one API specification without copy-pasting (yay!)



The JSON Schema first approach: Step #1

Extract your JSON Schemas as individual files on a GitHub repo

- Otherwise, its a lot harder to do everything within the constraints of i.e. an OpenAPI wrapper!
- And you can share the same schemas with more than one API specification without copy-pasting (yay!)



A “fundamental of getting your schema house in order”

Kin Lane, the API Evangelist

Help You Manage Your JSON Schema via a GitHub Repository

Written by Kin Lane
Nov 15, 2024

I am assembling a toolbox of API governance services for my customers based upon what I've been doing for the last year, but also based upon the needs of folks I am talking to right now. One of these services is the management of JSON Schema via GitHub. There are a number of schema registries out there, but they are mostly in service of a specific type of API implementation or protocol like Kafka or GraphQL, and I am looking to start with the fundamentals of getting your schema house in order without the distraction of these other

Here is where I am helping my customers get started—focused on the fundamentals of managing schema in GitHub.

The JSON Schema first approach: Step #1

Extract your JSON Schemas as individual files on a GitHub repo



The image displays three overlapping browser screenshots of GitHub repositories, illustrating the process of extracting JSON schemas as individual files. The top-left screenshot shows the AsyncAPI repository (spec-json-schemas) with a list of schema files. The middle screenshot shows the krakenD repository (krakend-schema) with a list of schema files. The bottom-right screenshot shows the NASA repository (gcn-schema) with a table of schema files.

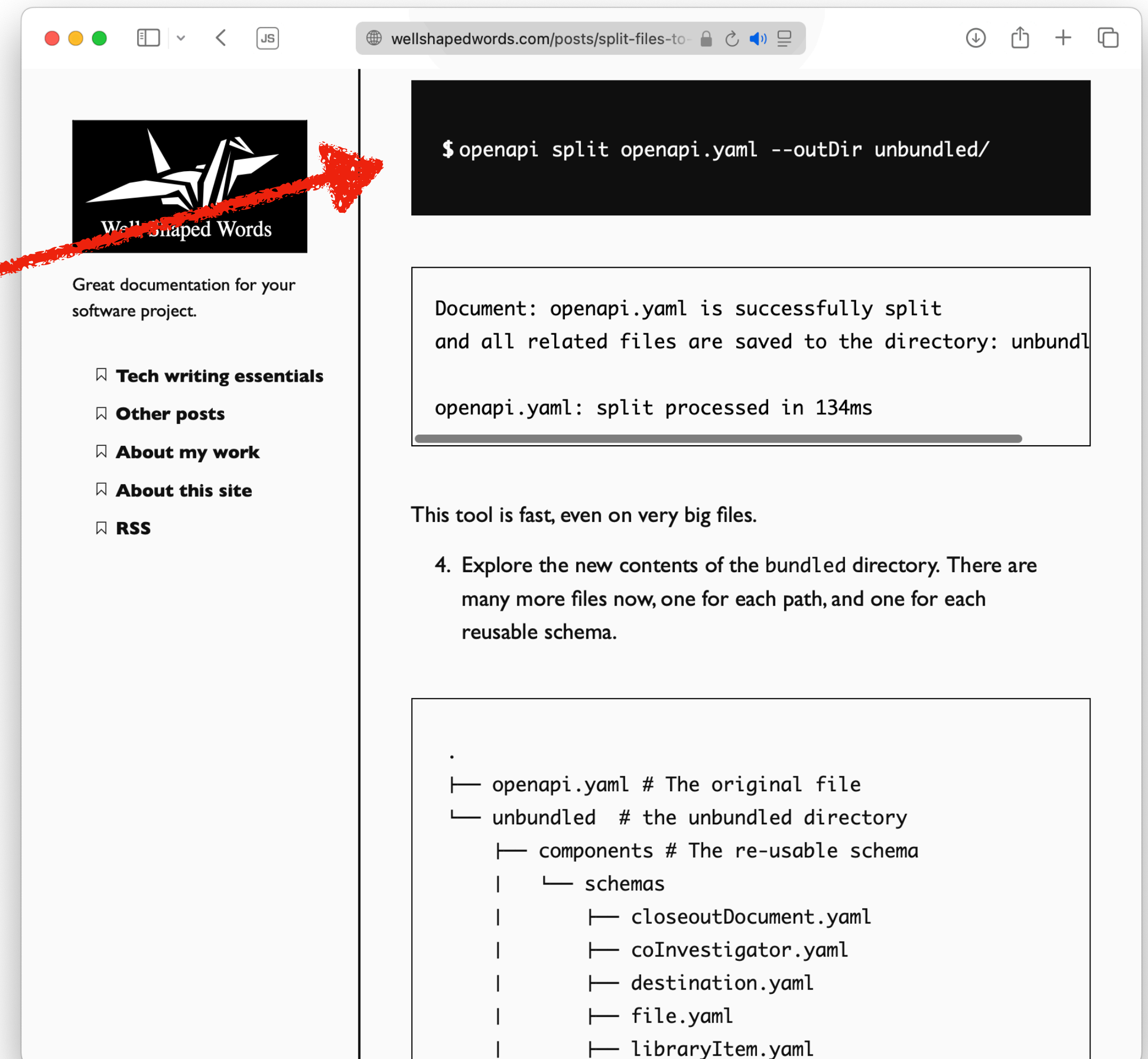
Name	Last commit message	Last commit date
..		
spectral	Back to development	last month
AdditionalInfo.schema.json	Back to development	last month
Alert.schema.json	Back to development	last month
DateTime.schema.json	Back to development	last month
DetectorStatus.schema.json	Back to development	last month
Distance.schema.json	Back to development	last month
Duration.schema.json	Back to development	last month

The JSON Schema first approach: Step #1

Extract your JSON Schemas as individual files on a GitHub repo

There are various **EXISTING** tools to “unbundle” an OpenAPI and extract its schemas as separate files

<https://wellshapedwords.com/posts/split-files-to-save-time/>



The screenshot shows a browser window at `wellshapedwords.com/posts/split-files-to-...`. The page content includes:

- A terminal command: `$ openapi split openapi.yaml --outDir unbundled/`
- A success message: `Document: openapi.yaml is successfully split and all related files are saved to the directory: unbundled/`
- A performance note: `openapi.yaml: split processed in 134ms`
- A statement: "This tool is fast, even on very big files."
- A numbered list item: "4. Explore the new contents of the bundled directory. There are many more files now, one for each path, and one for each reusable schema."
- A directory tree structure:

```
.
├── openapi.yaml # The original file
├── unbundled # the unbundled directory
│   ├── components # The re-usable schema
│   │   └── schemas
│   │       ├── closeoutDocument.yaml
│   │       ├── coInvestigator.yaml
│   │       ├── destination.yaml
│   │       ├── file.yaml
│   │       └── libraryItem.yaml
```

The JSON Schema first approach: Step #2

Find a fully-compliant tool to work with JSON Schema



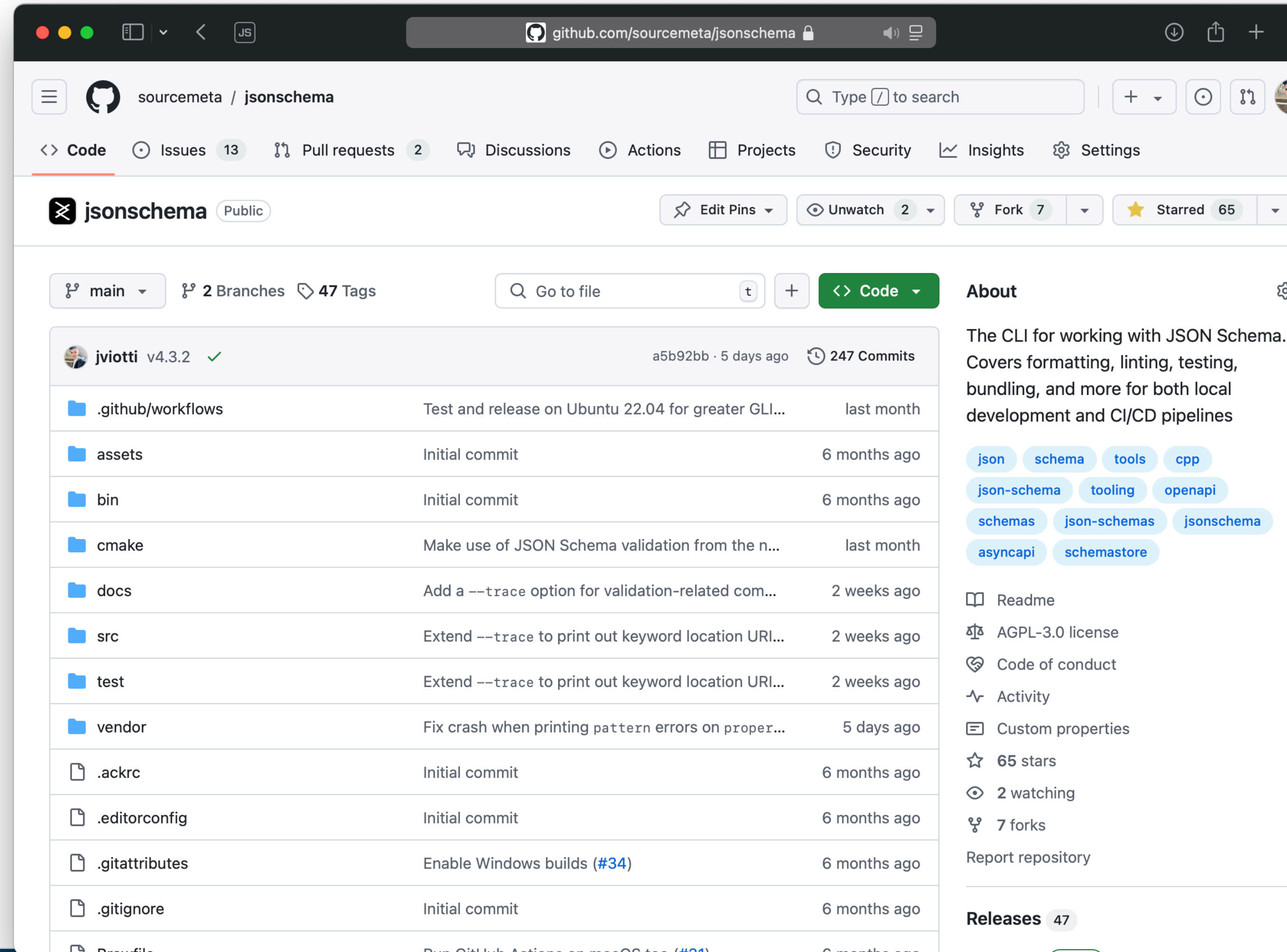
The JSON Schema first approach: Step #2

Find a fully-compliant tool to work with JSON Schema

Shameless plug:

You may enjoy my own CLI, as it was specifically created with these use cases in mind

<https://github.com/sourcemeta/jsonschema>



The screenshot shows the GitHub repository page for `sourcemeta/jsonschema`. The repository is public and has 65 stars, 2 watchers, and 7 forks. The current commit is `v4.3.2` by `jviotti`, committed 5 days ago. The repository contains several folders and files, including `.github/workflows`, `assets`, `bin`, `cmake`, `docs`, `src`, `test`, and `vendor`. The `src` folder is currently selected, showing a list of files and folders with their commit history.

File/Folder	Commit Message	Commit Date
<code>.github/workflows</code>	Test and release on Ubuntu 22.04 for greater GLL...	last month
<code>assets</code>	Initial commit	6 months ago
<code>bin</code>	Initial commit	6 months ago
<code>cmake</code>	Make use of JSON Schema validation from the n...	last month
<code>docs</code>	Add a <code>--trace</code> option for validation-related com...	2 weeks ago
<code>src</code>	Extend <code>--trace</code> to print out keyword location URI...	2 weeks ago
<code>test</code>	Extend <code>--trace</code> to print out keyword location URI...	2 weeks ago
<code>vendor</code>	Fix crash when printing pattern errors on proper...	5 days ago
<code>.ackrc</code>	Initial commit	6 months ago
<code>.editorconfig</code>	Initial commit	6 months ago
<code>.gitattributes</code>	Enable Windows builds (#34)	6 months ago
<code>.gitignore</code>	Initial commit	6 months ago
<code>Brewfile</code>	Run GitHub Actions on macOS too (#31)	6 months ago

The JSON Schema first approach: Step #2

Find a fully-compliant tool to work with JSON Schema

Shameless plug:

You may enjoy my own CLI, as it was specifically created with these use cases in mind



```
$ brew install sourcemeta/apps/jsonschema
```

<https://github.com/sourcemeta/jsonschema>

The screenshot shows the GitHub repository page for `sourcemeta/jsonschema`. The repository is public and has 65 stars, 2 watchers, and 7 forks. It is currently on the `main` branch, with 2 other branches and 47 tags. The repository was last updated 5 days ago by user `jviotti` with commit `a5b92bb`. The commit history shows several recent updates, including adding a `--trace` option for validation-related commands and extending it to print out keyword location URIs. The repository also includes a `Readme`, `AGPL-3.0 license`, and `Code of conduct`. The `Releases` section shows 47 releases.

File	Commit	Time
<code>.github/workflows</code>	Test and release on Ubuntu 22.04 for greater GLL...	last month
<code>assets</code>	Initial commit	6 months ago
<code>bin</code>	Initial commit	6 months ago
<code>cmake</code>	Make use of JSON Schema validation from the n...	last month
<code>docs</code>	Add a <code>--trace</code> option for validation-related com...	2 weeks ago
<code>src</code>	Extend <code>--trace</code> to print out keyword location URI...	2 weeks ago
<code>test</code>	Extend <code>--trace</code> to print out keyword location URI...	2 weeks ago
<code>vendor</code>	Fix crash when printing pattern errors on proper...	5 days ago
<code>.ackrc</code>	Initial commit	6 months ago
<code>.editorconfig</code>	Initial commit	6 months ago
<code>.gitattributes</code>	Enable Windows builds (#34)	6 months ago
<code>.gitignore</code>	Initial commit	6 months ago
<code>Brewfile</code>	Run GitHub Actions on macOS too (#31)	6 months ago

The JSON Schema first approach: Step #2

Find a fully-compliant tool to work with JSON Schema



Avoid AJV-based tools! AJV is non-compliant!

The JSON Schema first approach: Step #2

Find a fully-compliant tool to work with JSON Schema



The image shows a screenshot of the Bowtie report for the implementation 'js-ajv'. The report is titled "Compliance" and displays a table of results for various drafts. A red box highlights the "tests" section of the table, and a red arrow points to it from below.

Supported Dialects	tests			Badge
	Failed	Skipped	Errored	
Draft 4	8	0	49	Draft 4 90% Passing
Draft 6	8	0	69	Draft 6 90% Passing
Draft 7	8	0	131	Draft 7 84% Passing
Draft 2019-09	11	0	197	Draft 2019-09 82% Passing
Draft 2020-12	26	0	215	Draft 2020-12 80% Passing

Avoid AJV-based tools! AJV is non-compliant!

The JSON Schema first approach: Step #2

Find a fully-compliant tool to work with JSON Schema



The image shows a screenshot of the Bowtie report for Ajv implementations. The report is titled "Compliance" and displays a table of test results for various drafts. The table has columns for "Supported Dialects", "Failed", "Skipped", "Errored", and "Badge". The "Failed", "Skipped", and "Errored" columns are highlighted with a red box. A red arrow points to the table from below.

Supported Dialects	Tests			Badge
	Failed	Skipped	Errored	
Draft 4	8	0	49	Draft 4 90% Passing
Draft 6	8	0	69	Draft 6 90% Passing
Draft 7	8	0	131	Draft 7 84% Passing
Draft 2019-09	11	0	197	Draft 2019-09 82% Passing
Draft 2020-12	26	0	215	Draft 2020-12 80% Passing

Avoid AJV-based tools! AJV is non-compliant!

Because of it, many developers inadvertently create bad schemas

The JSON Schema first approach: Step #3

Check all schemas against their meta-schemas



The JSON Schema first approach: Step #3

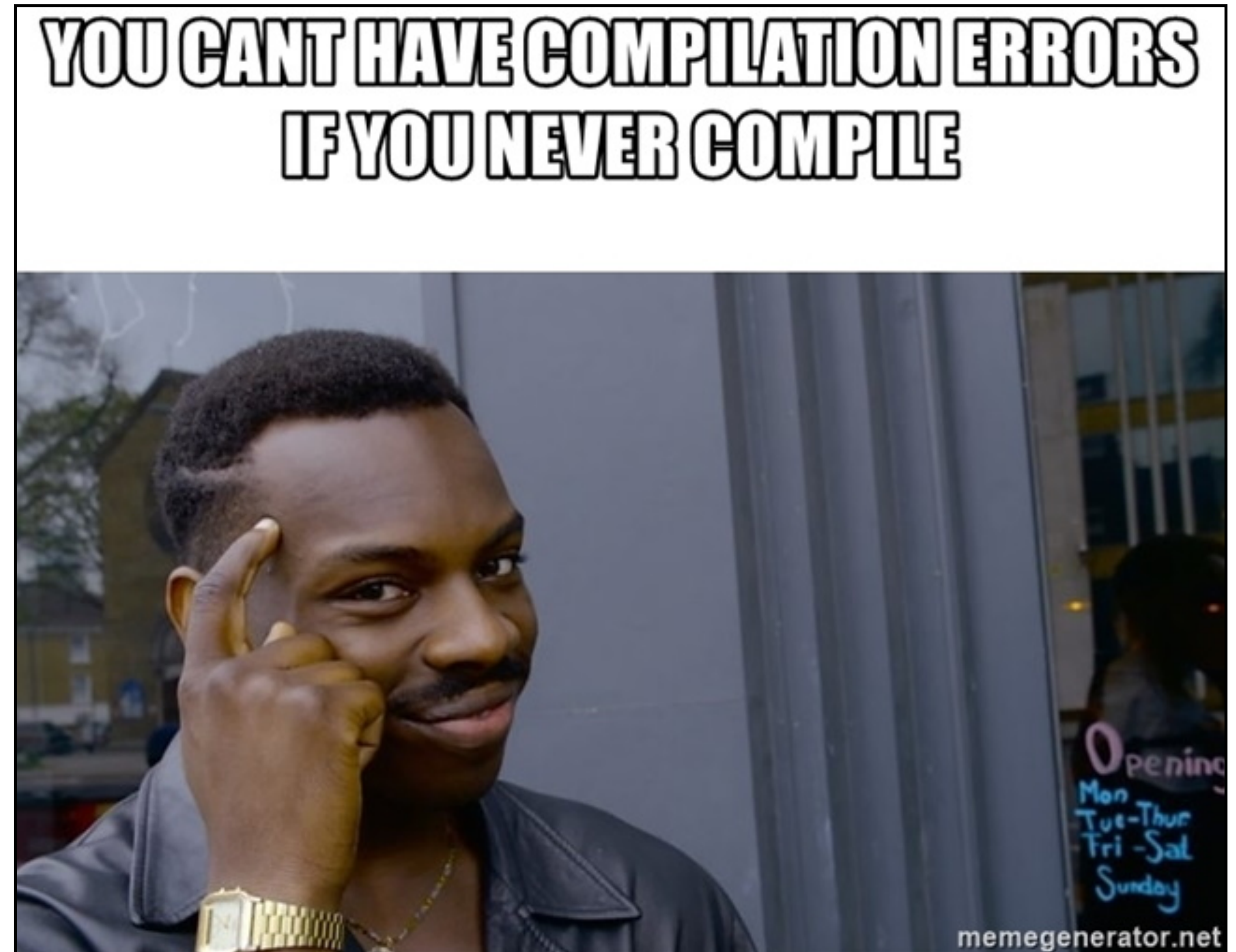
Check all schemas against their meta-schemas

This is analogous to
checking if your code
actually compiles

The JSON Schema first approach: Step #3

Check all schemas against their meta-schemas

This is analogous to checking if your code actually compiles



The JSON Schema first approach: Step #3

Check all schemas against their meta-schemas



```
{  
  "$schema": "https://json-schema.org/draft/2020-12/schema",  
  "items": [  
    { "type": "string" }  
  ]  
}
```

The JSON Schema first approach: Step #3

Check all schemas against their meta-schemas

```
● ● ●  
{  
  "$schema": "https://json-schema.org/draft/2020-12/schema",  
  "items": [  
    { "type": "string" }  
  ]  
}
```

The array variant of `items` in 2019-09 and before was replaced by `prefixItems`

The JSON Schema first approach: Step #3

Check all schemas against their meta-schemas

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "items": [
    { "type": "string" }
  ]
}
```

The array variant of `items` in 2019-09 and before was replaced by `prefixItems`

```
$ jsonschema metaschema --verbose schema.json
fail: schema.json
error: Schema validation failure
The value was expected to be of type object, or boolean but it was of type array
at instance location "/items"
  at evaluate path "/allOf/1/$ref/properties/items/$dynamicRef/allOf/0/$ref/type"
The array value was expected to validate against the 10 given subschemas
at instance location "/items"
  at evaluate path "/allOf/1/$ref/properties/items/$dynamicRef/allOf"
The array value was expected to validate against the first subschema in scope that declared the dynamicRef
at instance location "/items"
  at evaluate path "/allOf/1/$ref/properties/items/$dynamicRef"
The object value was expected to validate against the 15 defined properties subschemas
at instance location ""
  at evaluate path "/allOf/1/$ref/properties"
The object value was expected to validate against the 10 given subschemas
at instance location ""
  at evaluate path "/allOf"
```

The JSON Schema first approach: Step #3

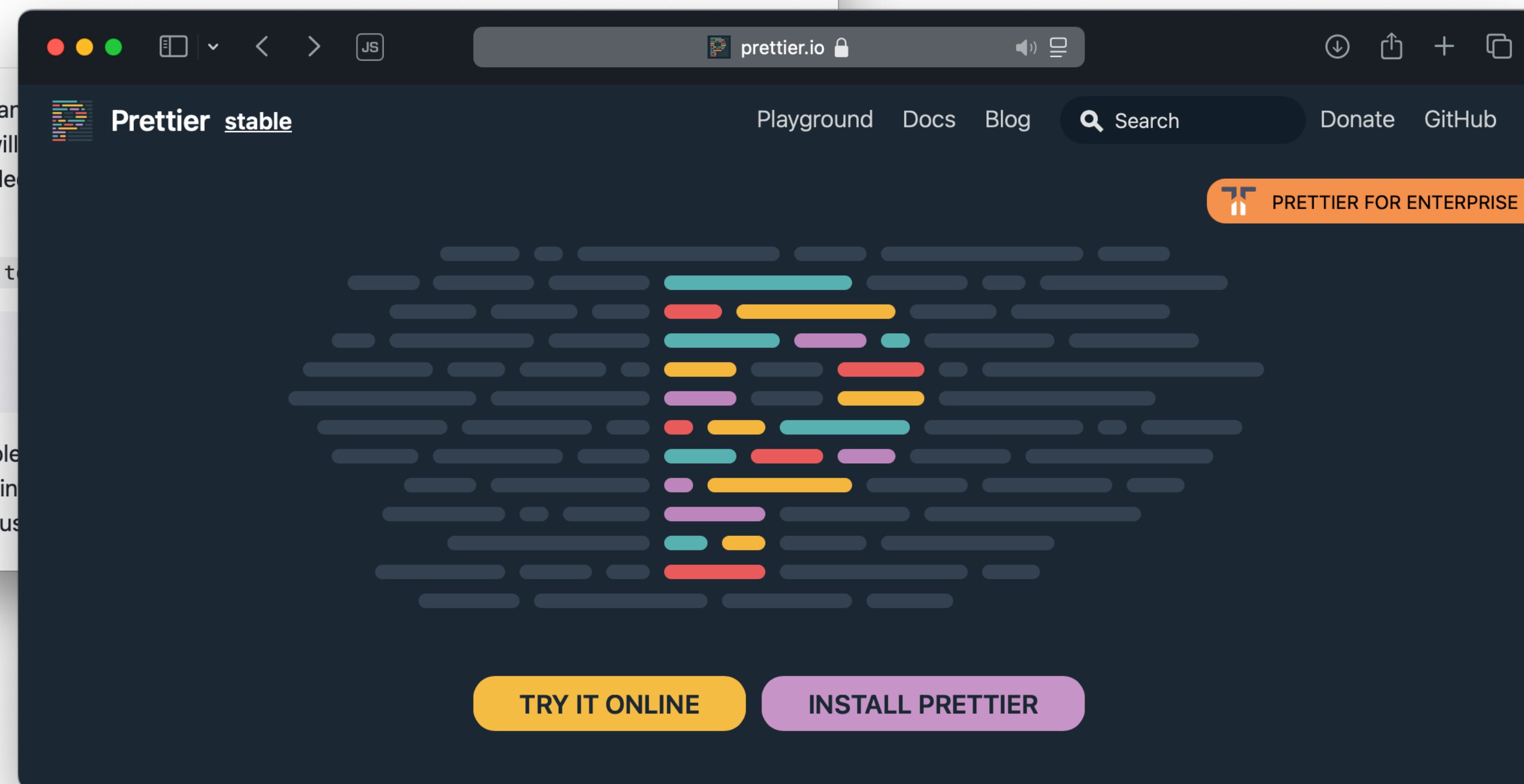
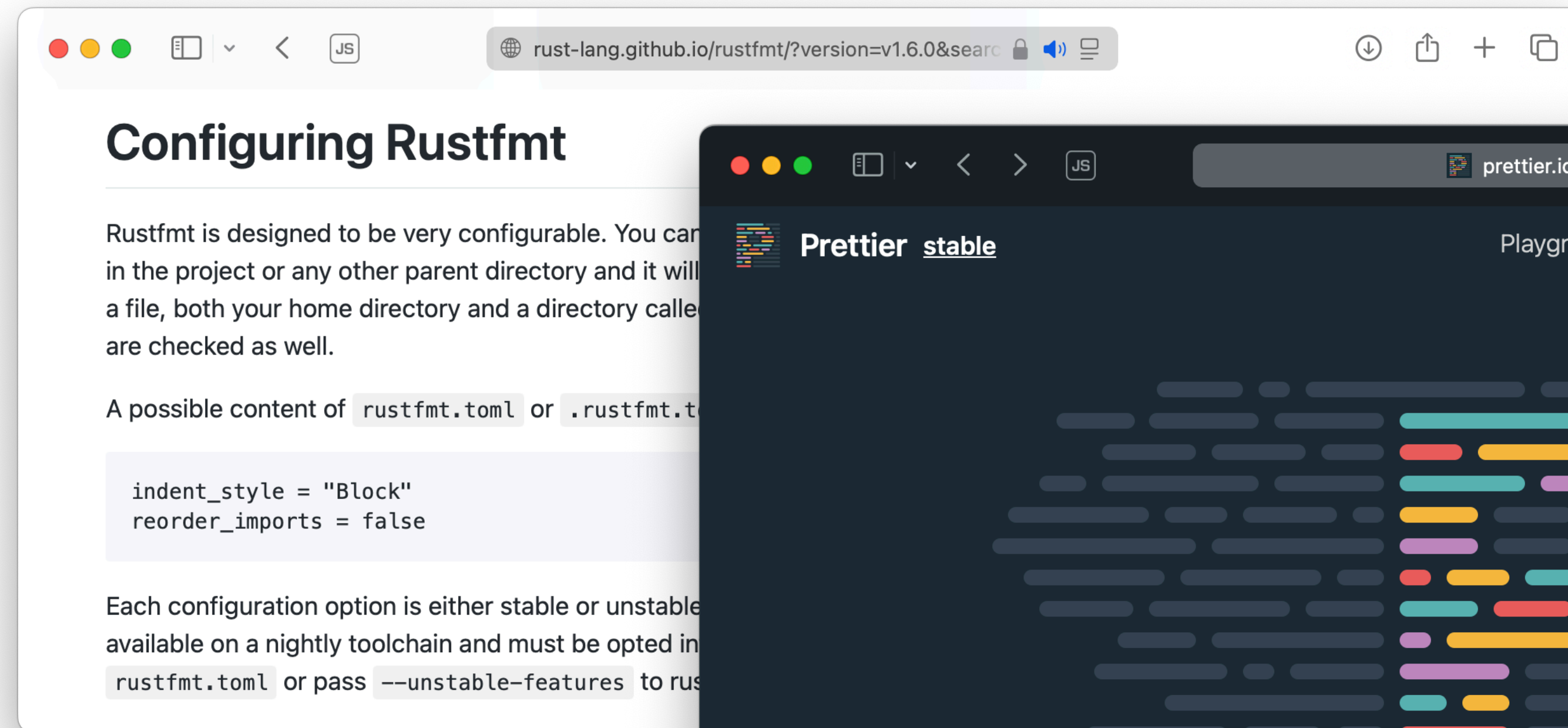
Check all schemas against their meta-schemas

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "items": [
    { "type": "string" }
  ]
}
```

The array variant of `items` in 2019-09 and before was replaced by `prefixItems`

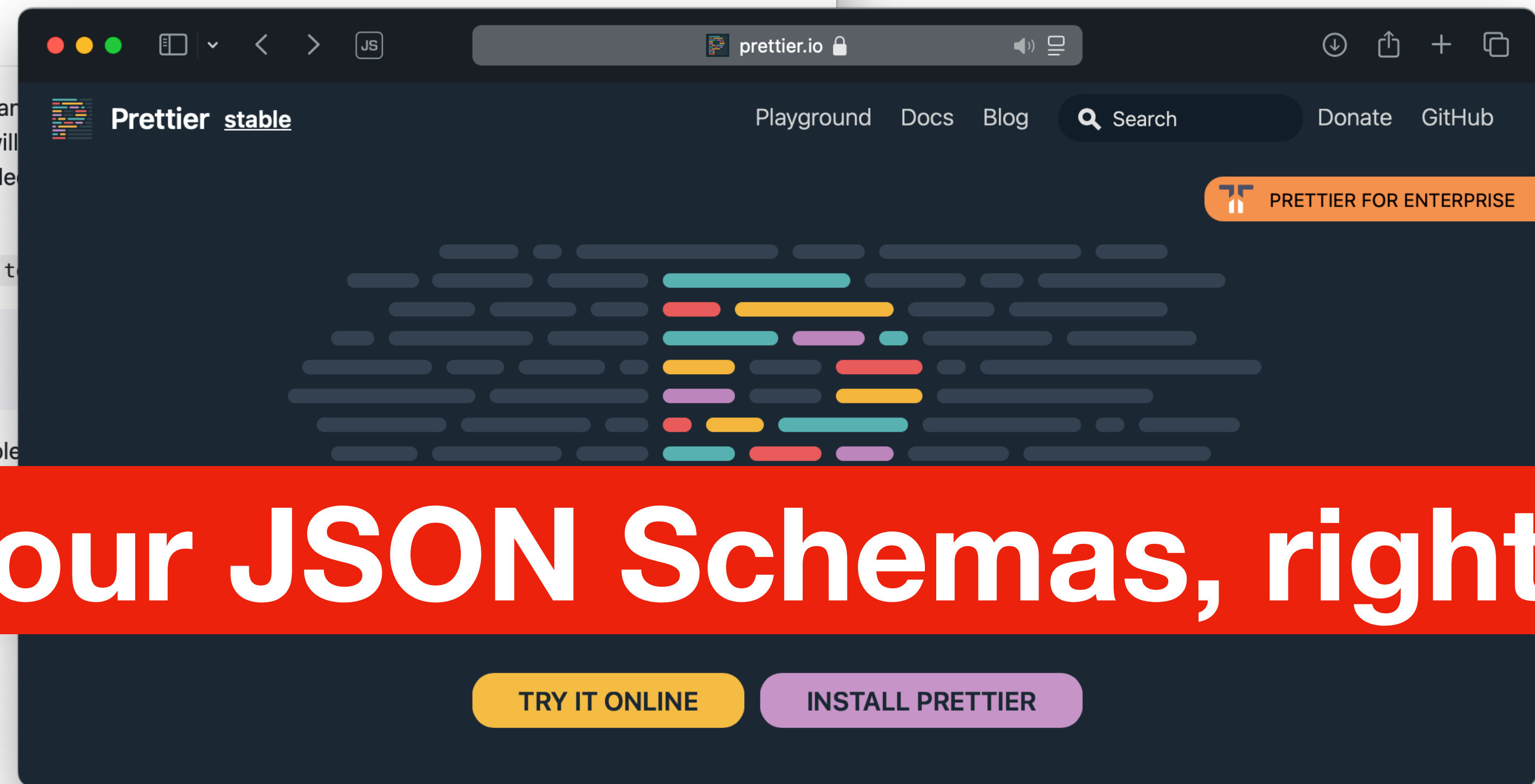
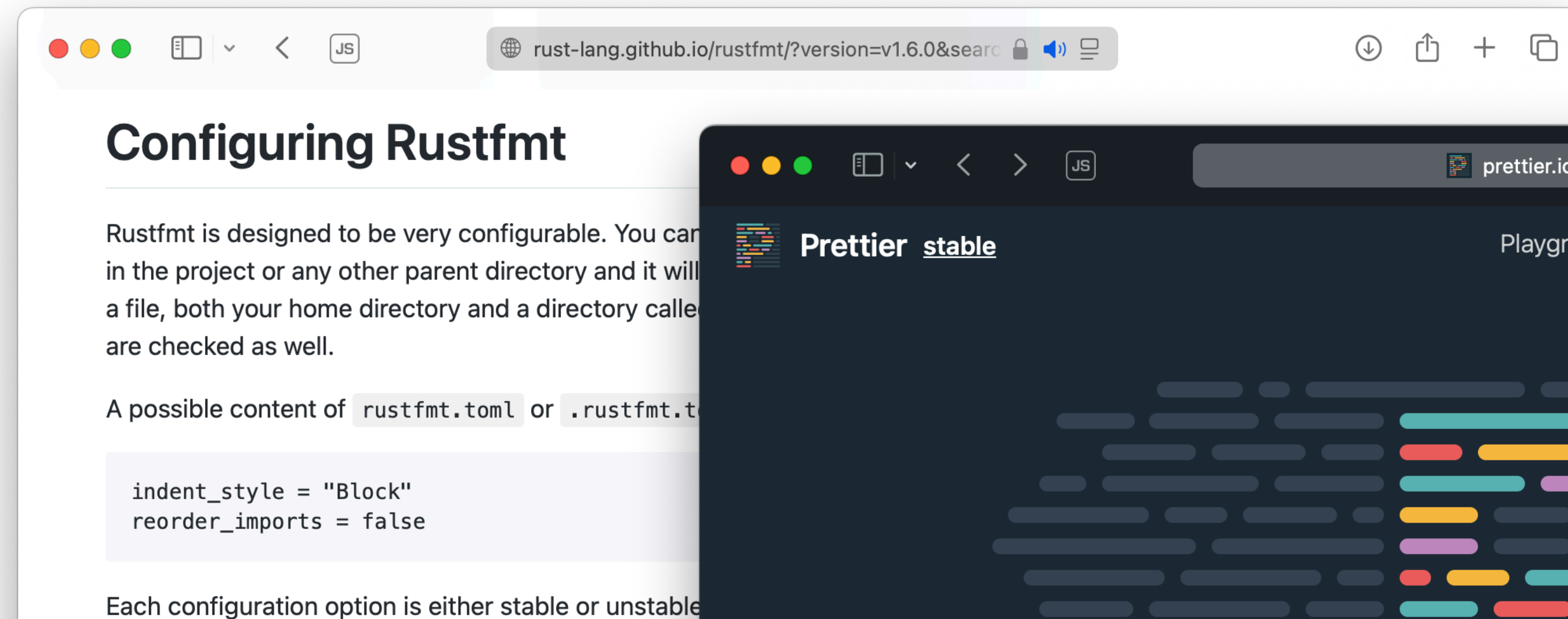
```
$ jsonschema metaschema --verbose schema.json
fail: schema.json
error: Schema validation failure
The value was expected to be of type object, or boolean but it was of type array
at instance location "/items"
  at evaluate path "/allOf/1/$ref/properties/items/$dynamicRef/allOf/0/$ref/type"
The array value was expected to validate against the 10 given subschemas
at instance location "/items"
  at evaluate path "/allOf/1/$ref/properties/items/$dynamicRef/allOf"
The array value was expected to validate against the first subschema in scope that declared the dynamicRef
at instance location "/items"
  at evaluate path "/allOf/1/$ref/properties/items/$dynamicRef"
The object value was expected to validate against the 15 defined properties subschemas
at instance location ""
  at evaluate path "/allOf/1/$ref/properties"
The object value was expected to validate against the 10 given subschemas
at instance location ""
  at evaluate path "/allOf"
```

Sounds obvious, but you would be surprised at how many people upgrade their schemas by just bumping `$schema` without taking a look at anything else



Are you using any code formatters?

Like prettier, rustfmt, gofmt, etc



Same for your JSON Schemas, right?

Are you using any code formatters?

Like prettier, rustfmt, gofmt, etc

The JSON Schema first approach: Step #4

Format your schemas for readability and unified styling



```
{ "$id": "https://example.com/iso8601/v1.json",  
  "pattern": "^(?!0000)\\d{4}$",  
  "type": "string",  
  "$schema": "https://json-schema.org/draft/2020-12/schema",  
  "title": "ISO 8601 four-digit year (YYYY)" }
```

The JSON Schema first approach: Step #4

Format your schemas for readability and unified styling



```
{ "$id": "https://example.com/iso8601/v1.json",  
  "pattern": "^(?!0000)\\d{4}$",  
  "type": "string",  
  "$schema": "https://json-schema.org/draft/2020-12/schema",  
  "title": "ISO 8601 four-digit year (YYYY)" }
```



```
$ jsonschema fmt schema.json  
$ cat schema.json
```


The JSON Schema first approach: Step #4

Format your schemas for readability and unified styling



```
{ "$id": "https://example.com/iso8601/v1.json",  
  "pattern": "^(?!0000)\\d{4}$",  
  "type": "string",  
  "$schema": "https://json-schema.org/draft/2020-12/schema",  
  "title": "ISO 8601 four-digit year (YYYY)" }
```



```
$ jsonschema fmt schema.json  
$ cat schema.json  
  
{  
  "$schema": "https://json-schema.org/draft/2020-12/schema",  
  "$id": "https://example.com/iso8601/v1.json",  
  "title": "ISO 8601 four-digit year (YYYY)",  
  "type": "string",  
  "pattern": "^(?!0000)\\d{4}$"  
}
```

The JSON Schema first approach: Step #4

Format your schemas for readability and unified styling



```
{ "$id": "https://example.com/iso8601/v1.json",  
  "pattern": "^(?!0000)\\d{4}$",  
  "type": "string",  
  "$schema": "https://json-schema.org/draft/2020-12/schema",  
  "title": "ISO 8601 four-digit year (YYYY)" }
```



```
$ jsonschema fmt schema.json  
$ cat schema.json  
  
{  
  "$schema": "https://json-schema.org/draft/2020-12/schema",  
  "$id": "https://example.com/iso8601/v1.json",  
  "title": "ISO 8601 four-digit year (YYYY)",  
  "type": "string",  
  "pattern": "^(?!0000)\\d{4}$"  
}
```

Dialect first, so we
know how to read
the rest



The JSON Schema first approach: Step #4

Format your schemas for readability and unified styling



```
{ "$id": "https://example.com/iso8601/v1.json",  
  "pattern": "^(?!0000)\\d{4}$",  
  "type": "string",  
  "$schema": "https://json-schema.org/draft/2020-12/schema",  
  "title": "ISO 8601 four-digit year (YYYY)" }
```



```
$ jsonschema fmt schema.json  
$ cat schema.json  
  
{  
  "$schema": "https://json-schema.org/draft/2020-12/schema",  
  "$id": "https://example.com/iso8601/v1.json",  
  "title": "ISO 8601 four-digit year (YYYY)",  
  "type": "string",  
  "pattern": "^(?!0000)\\d{4}$"  
}
```

Schema-wide
metadata at the top



The JSON Schema first approach: Step #4

Format your schemas for readability and unified styling

```
{ "$id": "https://example.com/iso8601/v1.json",  
  "pattern": "^(?!0000)\\d{4}$",  
  "type": "string",  
  "$schema": "https://json-schema.org/draft/2020-12/schema",  
  "title": "ISO 8601 four-digit year (YYYY)" }
```



```
$ jsonschema fmt schema.json  
$ cat schema.json  
  
{  
  "$schema": "https://json-schema.org/draft/2020-12/schema",  
  "$id": "https://example.com/iso8601/v1.json",  
  "title": "ISO 8601 four-digit year (YYYY)",  
  "type": "string",  
  "pattern": "^(?!0000)\\d{4}$  
}
```

Type information
first, if any



The JSON Schema first approach: Step #4

Format your schemas for readability and unified styling



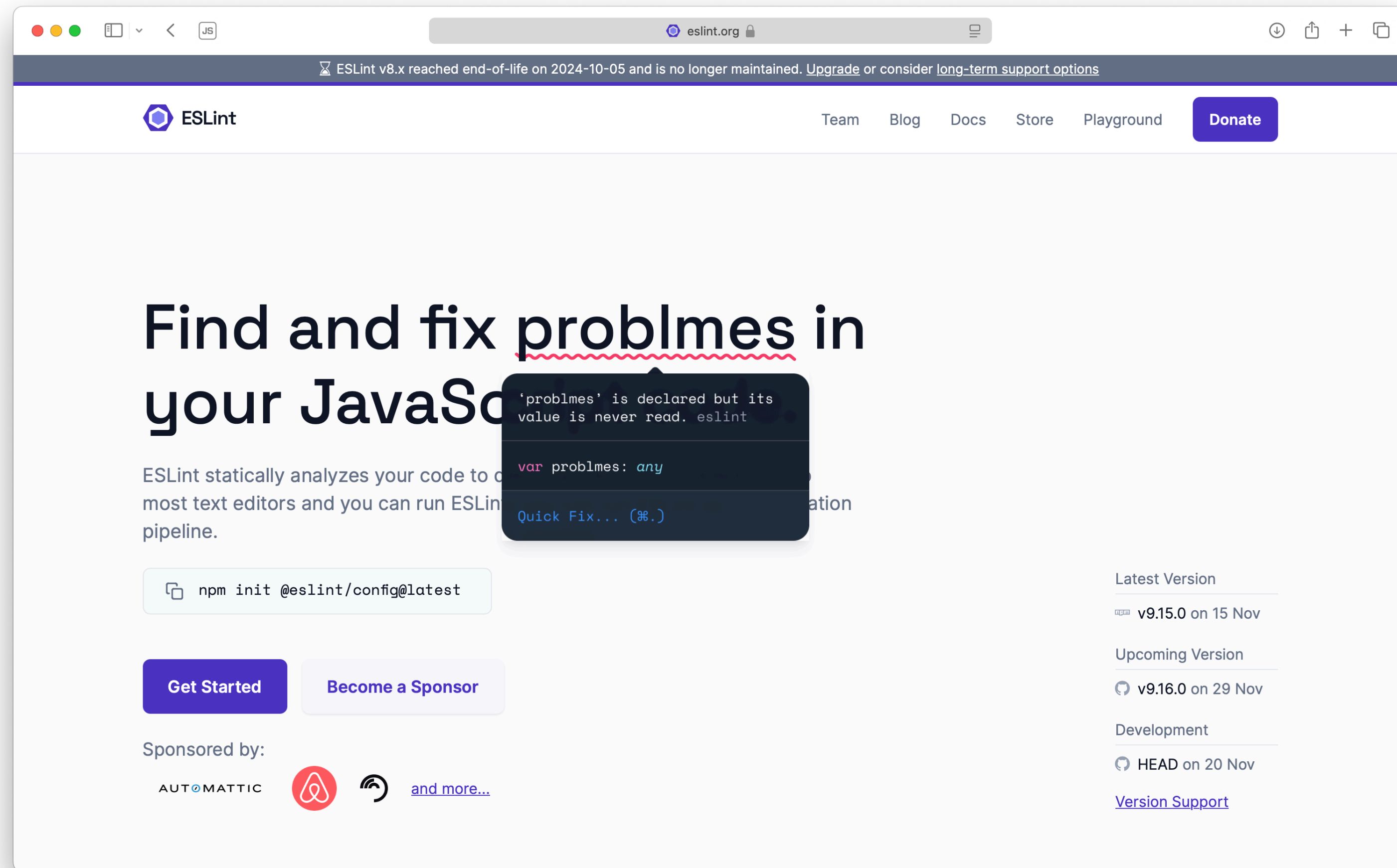
```
{ "$id": "https://example.com/iso8601/v1.json",  
  "pattern": "^(?!0000)\\d{4}$",  
  "type": "string",  
  "$schema": "https://json-schema.org/draft/2020-12/schema",  
  "title": "ISO 8601 four-digit year (YYYY)" }
```



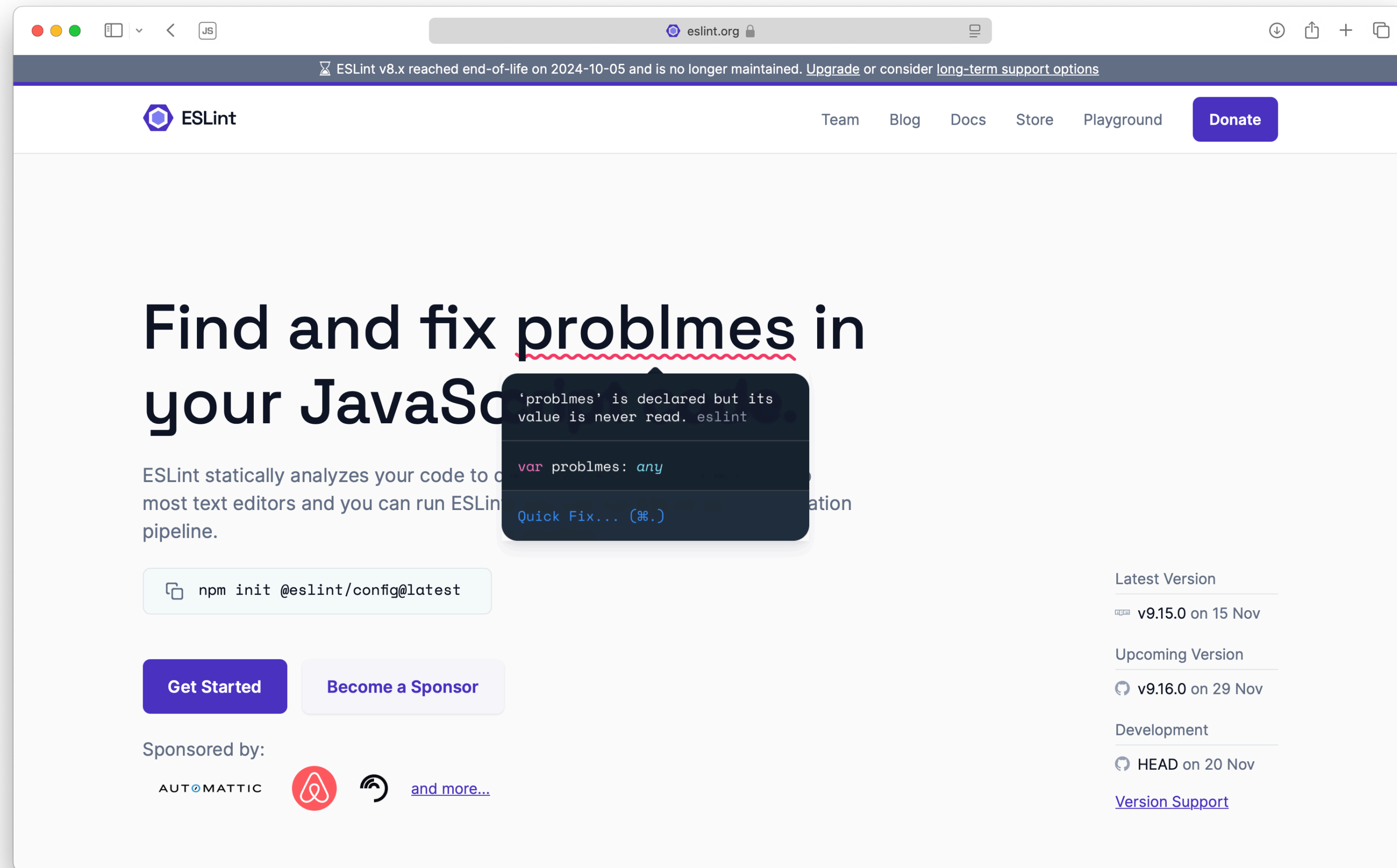
```
$ jsonschema fmt schema.json  
$ cat schema.json  
  
{  
  "$schema": "https://json-schema.org/draft/2020-12/schema",  
  "$id": "https://example.com/iso8601/v1.json",  
  "title": "ISO 8601 four-digit year (YYYY)",  
  "type": "string",  
  "pattern": "^(?!0000)\\d{4}$"  
}
```

Type-specific
constraints last





Are you using any code linters at work?



Are you using any code linters at work?

You are linting your JSON Schemas too, right?

The JSON Schema first approach: Step #5

Lint your schemas to find issues early and avoid bad practices



```
{  
  "$schema": "https://json-schema.org/draft/2020-12/schema",  
  "type": "string",  
  "enum": [ "foo", "bar", "baz" ]  
}
```


The JSON Schema first approach: Step #5

Lint your schemas to find issues early and avoid bad practices

```

{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "type": "string",
  enum: [ "foo", "bar", "baz" ]
}
```

This constrain is doing nothing

The JSON Schema first approach: Step #5

Lint your schemas to find issues early and avoid bad practices

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "type": "string",
  "enum": [ "foo", "bar", "baz" ]
}
```

This constrain is doing nothing

```
$ jsonschema lint schema.json
Setting `type` alongside `enum` is considered an anti-pattern, as the enumeration choices already imply their respective types (enum_with_type)
at schema location
```

The JSON Schema first approach: Step #5

Lint your schemas to find issues early and avoid bad practices

```
● ● ●  
  
{  
  "$schema": "https://json-schema.org/draft/2020-12/schema",  
  "type": "string",  
  "enum": [ "foo", "bar", "baz" ]  
}
```

This constrain is doing nothing

```
● ● ●  
  
$ jsonschema lint schema.json  
Setting `type` alongside `enum` is considered an anti-pattern, as the enumeration choices already imply their respective types (enum_with_type)  
at schema location
```

For most rules, you can do: `jsonschema lint -fix schema.json`



Do you write automated tests for your code?



Do you write automated tests for your code?

You are testing your JSON Schemas too, right?

The JSON Schema first approach: Step #6

Unit test your schemas to ensure they match what you intend



```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://example.com/iso8601/v1.json",
  "title": "ISO 8601 four-digit year (YYYY)",
  "type": "string",
  "pattern": "^(?!0000)\\d{4}$"
}
```

The JSON Schema first approach: Step #6

Unit test your schemas to ensure they match what you intend



```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://example.com/iso8601/v1.json",
  "title": "ISO 8601 four-digit year (YYYY)",
  "type": "string",
  "pattern": "^(?!0000)\\d{4}$"
}
```



```
{
  "target": "https://example.com/iso8601/v1.json",
  "tests": [
  ]
}
```

The JSON Schema first approach: Step #6

Unit test your schemas to ensure they match what you intend

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://example.com/iso8601/v1.json",
  "title": "ISO 8601 four-digit year (YYYY)",
  "type": "string",
  "pattern": "^(?!0000)\\d{4}$"
}
```

```
{
  "target": "https://example.com/iso8601/v1.json",
  "tests": [
    {
      "description": "Valid year",
      "valid": true,
      "data": "2024"
    },
  ]
}
```


The JSON Schema first approach: Step #6

Unit test your schemas to ensure they match what you intend

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://example.com/iso8601/v1.json",
  "title": "ISO 8601 four-digit year (YYYY)",
  "type": "string",
  "pattern": "^(?!0000)\\d{4}$"
}
```

```
{
  "target": "https://example.com/iso8601/v1.json",
  "tests": [
    {
      "description": "Valid year",
      "valid": true,
      "data": "2024"
    },
    {
      "description": "Zero is not a valid year",
      "valid": false,
      "data": "0000"
    },
  ]
}
```

The JSON Schema first approach: Step #6

Unit test your schemas to ensure they match what you intend

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://example.com/iso8601/v1.json",
  "title": "ISO 8601 four-digit year (YYYY)",
  "type": "string",
  "pattern": "^(?!0000)\\d{4}$"
}
```

```
{
  "target": "https://example.com/iso8601/v1.json",
  "tests": [
    {
      "description": "Valid year",
      "valid": true,
      "data": "2024"
    },
    {
      "description": "Zero is not a valid year",
      "valid": false,
      "data": "0000"
    },
    {
      "description": "Non-string is invalid",
      "valid": false,
      "data": 2024
    }
  ]
}
```

The JSON Schema first approach: Step #6

Unit test your schemas to ensure they match what you intend

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://example.com/iso8601/v1.json",
  "title": "ISO 8601 four-digit year (YYYY)",
  "type": "string",
  "pattern": "^(?!0000)\\d{4}$"
}
```

```
$ jsonschema test --verbose test.json --resolve schema.json
Importing schema into the resolution context: schema.json
test.json:
  1/3 PASS Valid year
  2/3 PASS Zero is not a valid year
  3/3 PASS Non-string is invalid
```

```
{
  "target": "https://example.com/iso8601/v1.json",
  "tests": [
    {
      "description": "Valid year",
      "valid": true,
      "data": "2024"
    },
    {
      "description": "Zero is not a valid year",
      "valid": false,
      "data": "0000"
    },
    {
      "description": "Non-string is invalid",
      "valid": false,
      "data": 2024
    }
  ]
}
```

The JSON Schema first approach: Step #6

Unit test your schemas to ensure they match what you intend

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://example.com/iso8601/v1.json",
  "title": "ISO 8601 four-digit year (YYYY)",
  "type": "string",
  "pattern": "^(?!0000)\\d{4}$"
}
```

```
$ jsonschema test --verbose test.json --resolve schema.json
Importing schema into the resolution context: schema.json
test.json:
  1/3 PASS Valid year
  2/3 PASS Zero is not a valid year
  3/3 PASS Non-string is invalid
```

```
{
  "target": "https://example.com/iso8601/v1.json",
  "tests": [
    {
      "description": "Valid year",
      "valid": true,
      "data": "2024"
    },
    {
      "description": "Zero is not a valid year",
      "valid": false,
      "data": "0000"
    },
    {
      "description": "Non-string is invalid",
      "valid": false,
      "data": 2024
    }
  ]
}
```

The JSON Schema first approach: Step #6

Unit test your schemas to ensure they match what you intend

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://example.com/iso8601/v1.json",
  "title": "ISO 8601 four-digit year (YYYY)",
  "type": "string",
  "pattern": "^(?!0000)\\d{4}$"
}
```

```
$ jsonschema test --verbose test.json --resolve schema.json
Importing schema into the resolution context: schema.json
test.json:
  1/3 PASS Valid year
  2/3 PASS Zero is not a valid year
  3/3 PASS Non-string is invalid
```

```
{
  "target": "https://example.com/iso8601/v1.json",
  "tests": [
    {
      "description": "Valid year",
      "valid": true,
      "data": "2024"
    },
    {
      "description": "Zero is not a valid year",
      "valid": false,
      "data": "0000"
    },
    {
      "description": "Non-string is invalid",
      "valid": false,
      "data": 2024
    }
  ]
}
```

The JSON Schema first approach: Step #6

Unit test your schemas to ensure they match what you intend

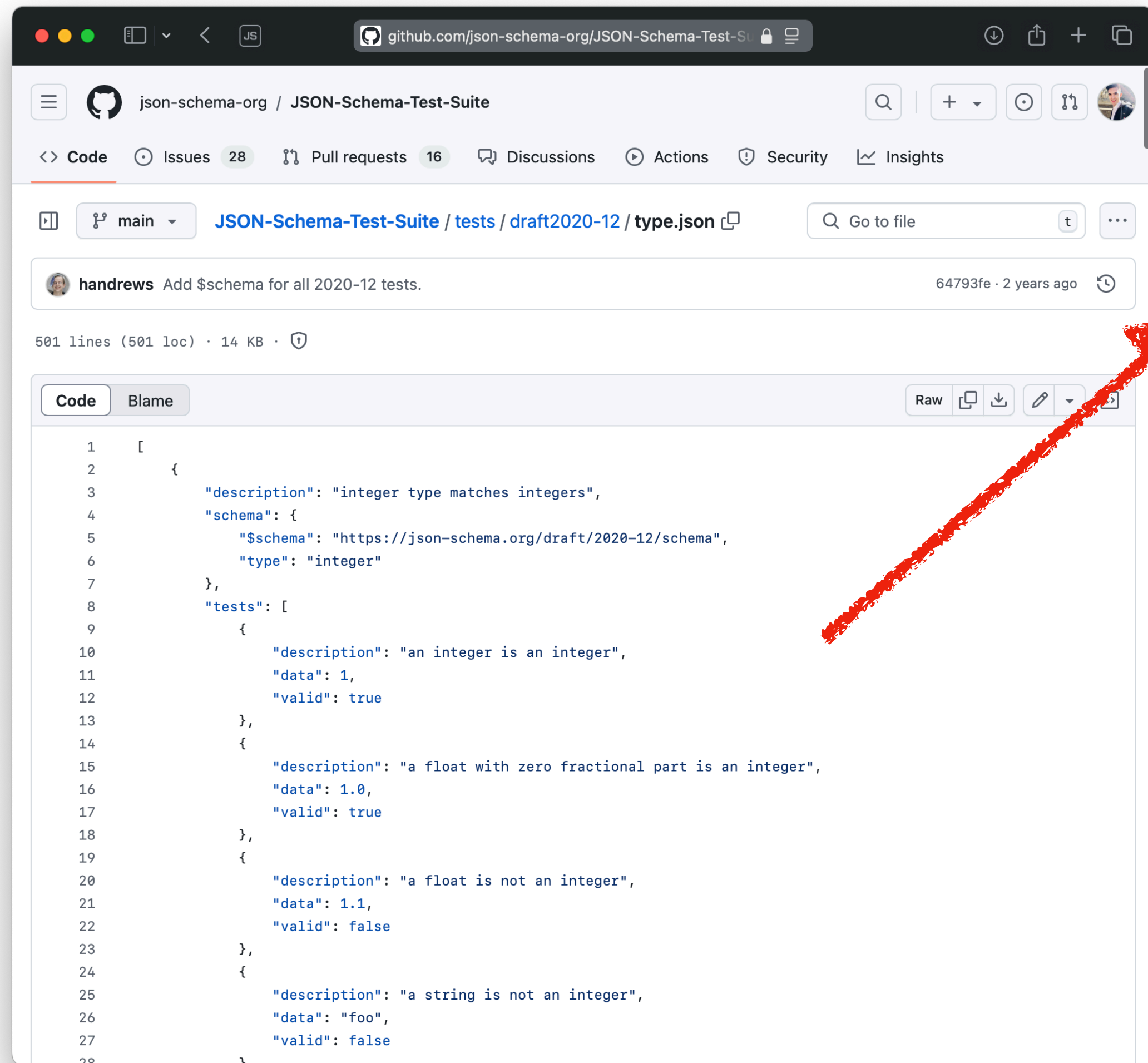
```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://example.com/iso8601/v1.json",
  "title": "ISO 8601 four-digit year (YYYY)",
  "type": "string",
  "pattern": "^(?!0000)\\d{4}$"
}
```

```
$ jsonschema test --verbose test.json --resolve schema.json
Importing schema into the resolution context: schema.json
test.json:
  1/3 PASS Valid year
  2/3 PASS Zero is not a valid year
  3/3 PASS Non-string is invalid
```

```
{
  "target": "https://example.com/iso8601/v1.json",
  "tests": [
    {
      "description": "Valid year",
      "valid": true,
      "data": "2024"
    },
    {
      "description": "Zero is not a valid year",
      "valid": false,
      "data": "0000"
    },
    {
      "description": "Non-string is invalid",
      "valid": false,
      "data": 2024
    }
  ]
}
```

The JSON Schema first approach: Step #6

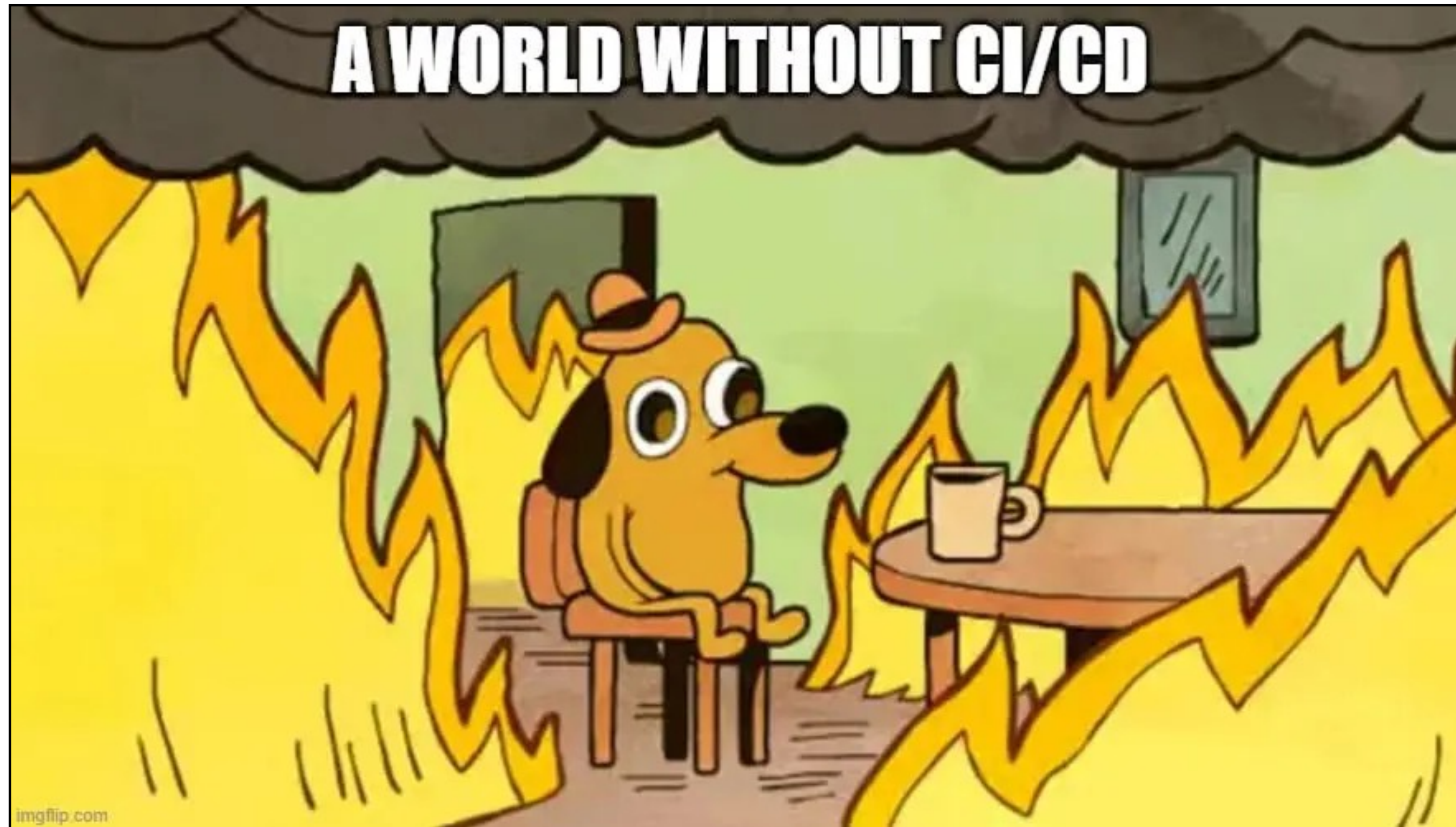
Unit test your schemas to ensure they match what you intend



The screenshot shows a GitHub pull request for the file `JSON-Schema-Test-Suite / tests / draft2020-12 / type.json`. The pull request is titled "Add \$schema for all 2020-12 tests." and was created by user `handrews` 2 years ago. The code is displayed in a diff view, showing a JSON object with a schema and a list of tests. A red arrow points to the `"tests"` array in the code.

```
1  [
2    {
3      "description": "integer type matches integers",
4      "schema": {
5        "$schema": "https://json-schema.org/draft/2020-12/schema",
6        "type": "integer"
7      },
8      "tests": [
9        {
10         "description": "an integer is an integer",
11         "data": 1,
12         "valid": true
13       },
14       {
15         "description": "a float with zero fractional part is an integer",
16         "data": 1.0,
17         "valid": true
18       },
19       {
20         "description": "a float is not an integer",
21         "data": 1.1,
22         "valid": false
23       },
24       {
25         "description": "a string is not an integer",
26         "data": "foo",
27         "valid": false
28     }
29   ]
30 }
```

The syntax of my test runner is *intentionally* inspired by the official JSON Schema Test Suite



**Just like you would do with your code,
run all of this on CI/CD!**



```
name: JSON Schema CI
```

```
jobs:
```

```
  test:
```

```
    runs-on: ubuntu-latest
```

```
    steps:
```

```
      - uses: actions/checkout@v4
```

```
      - name: Install the Sourcemeta JSON Schema CLI  
        uses: sourcemeta/jsonschema@v4.3.2
```

```
      - name: Check schemas against their metaschemas  
        run: jsonschema metaschema --verbose schemas/
```

```
      - name: Check schemas are formatted  
        run: jsonschema fmt --check --verbose schemas/
```

```
      - name: Lint schemas  
        run: jsonschema lint --verbose schemas/
```

```
      - name: Run test suite  
        run: jsonschema test --verbose tests/ --resolve schemas
```



GitHub Actions

We provide an easy
GitHub Actions
integration



```
name: JSON Schema CI
```

```
jobs:
```

```
  test:
```

```
    runs-on: ubuntu-latest
```

```
    steps:
```

```
      - uses: actions/checkout@v4
```

```
      - name: Install the Sourcemeta JSON Schema CLI  
        uses: sourcemeta/jsonschema@v4.3.2
```

```
      - name: Check schemas against their metaschemas  
        run: jsonschema metaschema --verbose schemas/
```

```
      - name: Check schemas are formatted  
        run: jsonschema fmt --check --verbose schemas/
```

```
      - name: Lint schemas  
        run: jsonschema lint --verbose schemas/
```

```
      - name: Run test suite  
        run: jsonschema test --verbose tests/ --resolve schemas
```



GitHub Actions



Make sure your schemas “compile”



```
name: JSON Schema CI
```

```
jobs:
```

```
  test:
```

```
    runs-on: ubuntu-latest
```

```
    steps:
```

```
      - uses: actions/checkout@v4
```

```
      - name: Install the Sourcemeta JSON Schema CLI  
        uses: sourcemeta/jsonschema@v4.3.2
```

```
      - name: Check schemas against their metaschemas  
        run: jsonschema metaschema --verbose schemas/
```

```
      - name: Check schemas are formatted  
        run: jsonschema fmt --check --verbose schemas/
```

```
      - name: Lint schemas  
        run: jsonschema lint --verbose schemas/
```

```
      - name: Run test suite  
        run: jsonschema test --verbose tests/ --resolve schemas
```



GitHub Actions

Enforce a common
readable style



```
name: JSON Schema CI
```

```
jobs:
```

```
  test:
```

```
    runs-on: ubuntu-latest
```

```
    steps:
```

```
      - uses: actions/checkout@v4
```

```
      - name: Install the Sourcemeta JSON Schema CLI  
        uses: sourcemeta/jsonschema@v4.3.2
```

```
      - name: Check schemas against their metaschemas  
        run: jsonschema metaschema --verbose schemas/
```

```
      - name: Check schemas are formatted  
        run: jsonschema fmt --check --verbose schemas/
```

```
      - name: Lint schemas  
        run: jsonschema lint --verbose schemas/
```

```
      - name: Run test suite  
        run: jsonschema test --verbose tests/ --resolve schemas
```



GitHub Actions

Catch obvious
issues and avoid
bad practices



```
name: JSON Schema CI
```

```
jobs:
```

```
  test:
```

```
    runs-on: ubuntu-latest
```

```
    steps:
```

- uses: actions/checkout@v4
- name: Install the Sourcemeta JSON Schema CLI
 uses: sourcemeta/jsonschema@v4.3.2
- name: Check schemas against their metaschemas
 run: jsonschema metaschema --verbose schemas/
- name: Check schemas are formatted
 run: jsonschema fmt --check --verbose schemas/
- name: Lint schemas
 run: jsonschema lint --verbose schemas/

```
- name: Run test suite  
  run: jsonschema test --verbose tests/ --resolve schemas
```



GitHub Actions



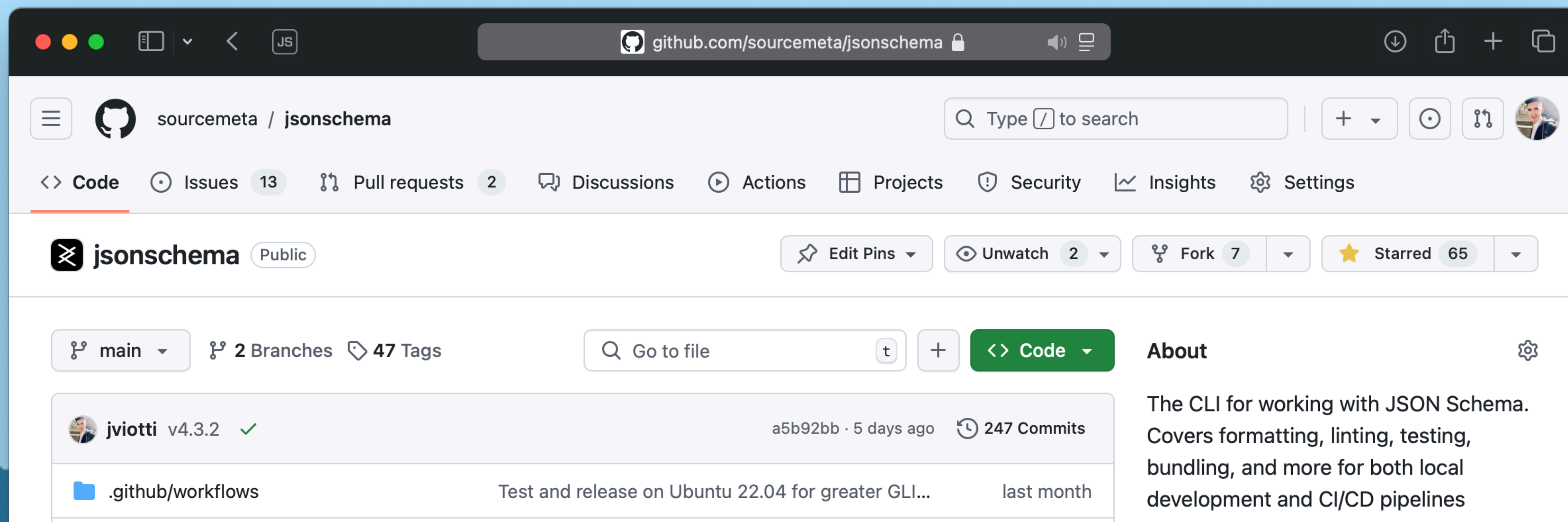
Make sure the
schemas actually
do what you intend

Thanks a lot!



JSON Schema CLI

<https://github.com/sourcemeta/jsonschema>

A screenshot of the GitHub repository page for 'sourcemeta/jsonschema'. The browser address bar shows 'github.com/sourcemeta/jsonschema'. The repository name 'sourcemeta / jsonschema' is displayed at the top left. A search bar with the placeholder 'Type / to search' is on the right. Below the repository name, there are navigation links for 'Code', 'Issues 13', 'Pull requests 2', 'Discussions', 'Actions', 'Projects', 'Security', 'Insights', and 'Settings'. The repository is marked as 'Public'. Action buttons include 'Edit Pins', 'Unwatch 2', 'Fork 7', and 'Starred 65'. The main content area shows the 'main' branch selected, with '2 Branches' and '47 Tags'. A search bar 'Go to file' is present. A commit by 'jviotti' for version 'v4.3.2' is shown, dated '5 days ago' with '247 Commits'. A folder named '.github/workflows' is listed with the description 'Test and release on Ubuntu 22.04 for greater GLI...' and a date of 'last month'. An 'About' section on the right describes the CLI as a tool for working with JSON Schema, covering formatting, linting, testing, bundling, and CI/CD pipelines.