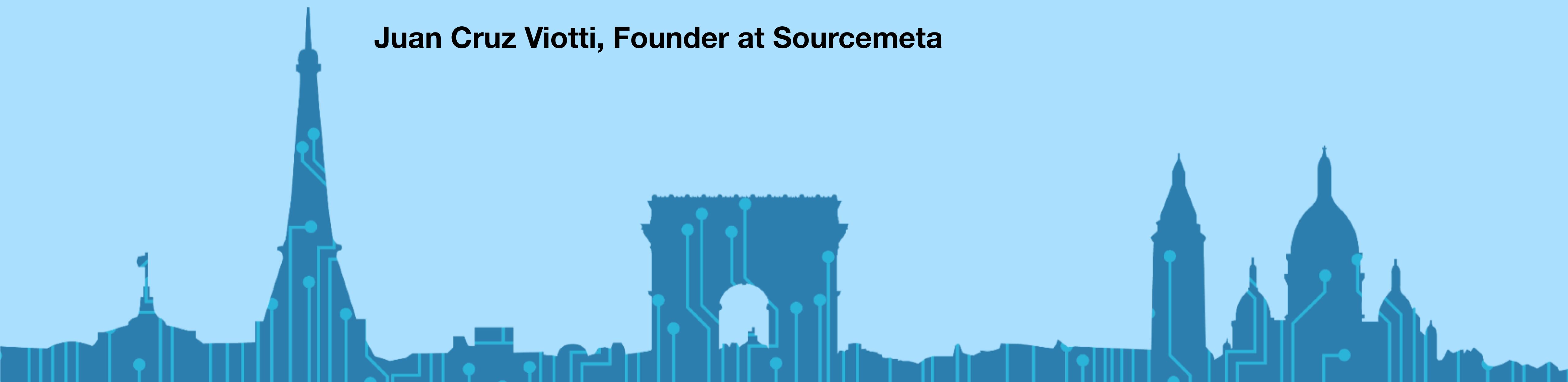


Applying Software Engineering Practices to Schemas

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://www.learnjsonschema.com) and AlterSchema

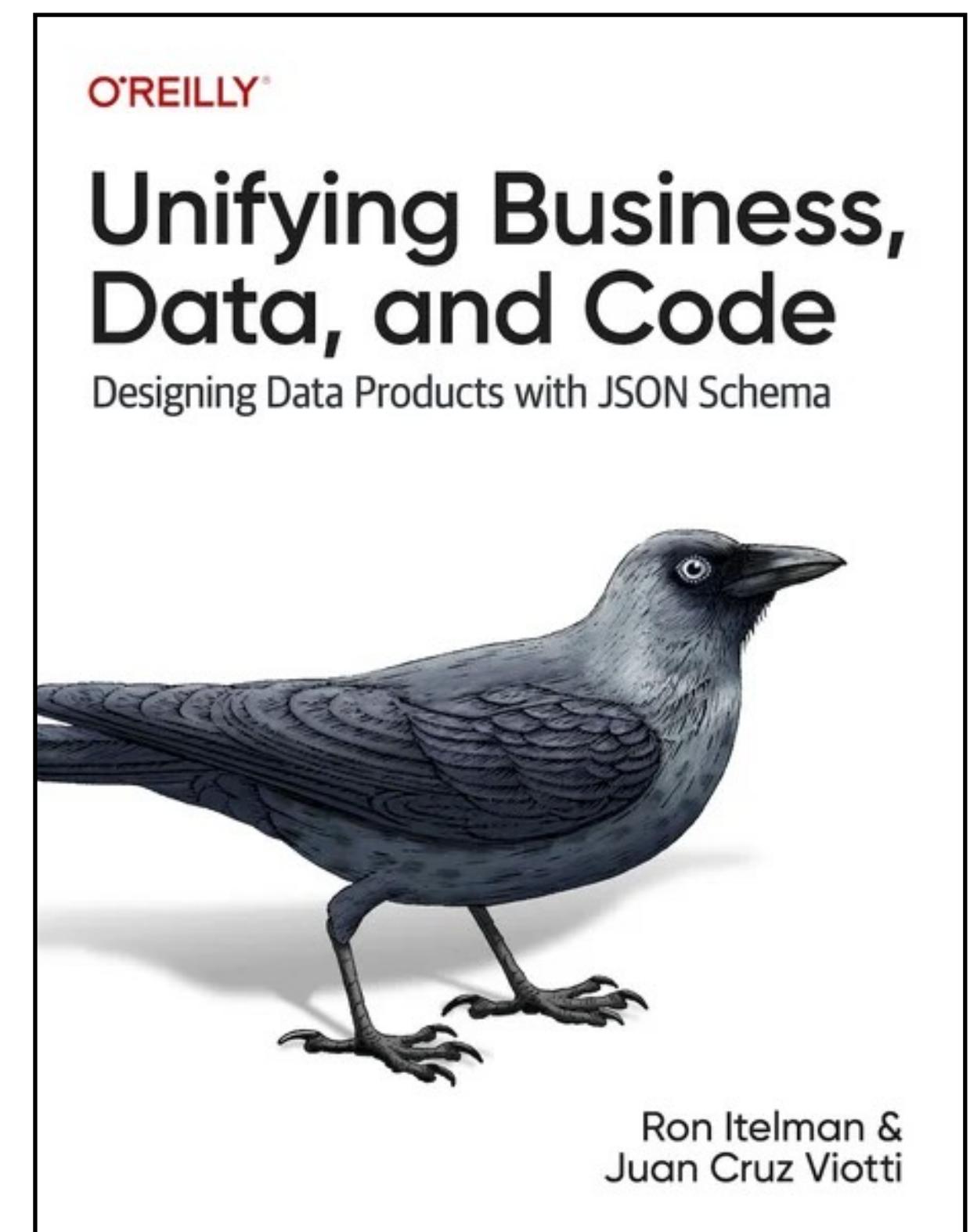


The screenshot shows a web browser window with the URL www.learnjsonschema.com/2020-12/. The page title is "JSON Schema Docs". A red arrow points from the "Author of various JSON Schema tooling" bullet point in the list above to the "AlterSchema" link on this page. Another red arrow points from the "LearnJSONSchema.com" link in the list to the "LearnJSONSchema.com" link on the page.

2020-12 Dialect

JSON Schema 2020-12 is a JSON media type for defining the structure of JSON data. JSON Schema is intended to define validation, documentation, hyperlink navigation, and interaction control of JSON data.

Specification <https://json-schema.org/draft/2020-12/json-schema-core.html>



API Specifications

Human/Machine readable descriptions of APIs

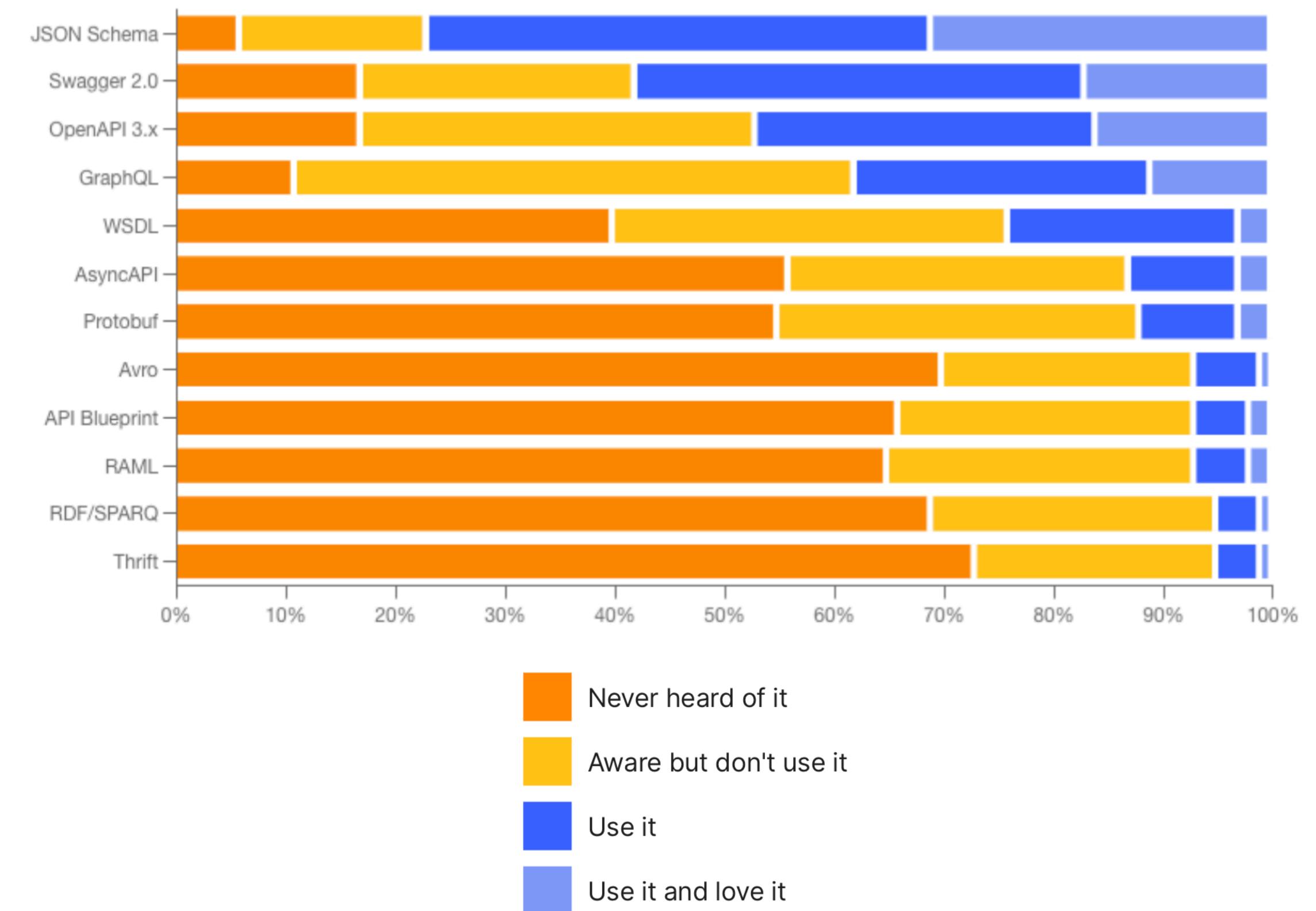
- Single source of truth
- Standardised documentation
- Code generation
- Improved testing / validation
- Increased discoverability



API Specifications

Human/Machine readable descriptions of APIs

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

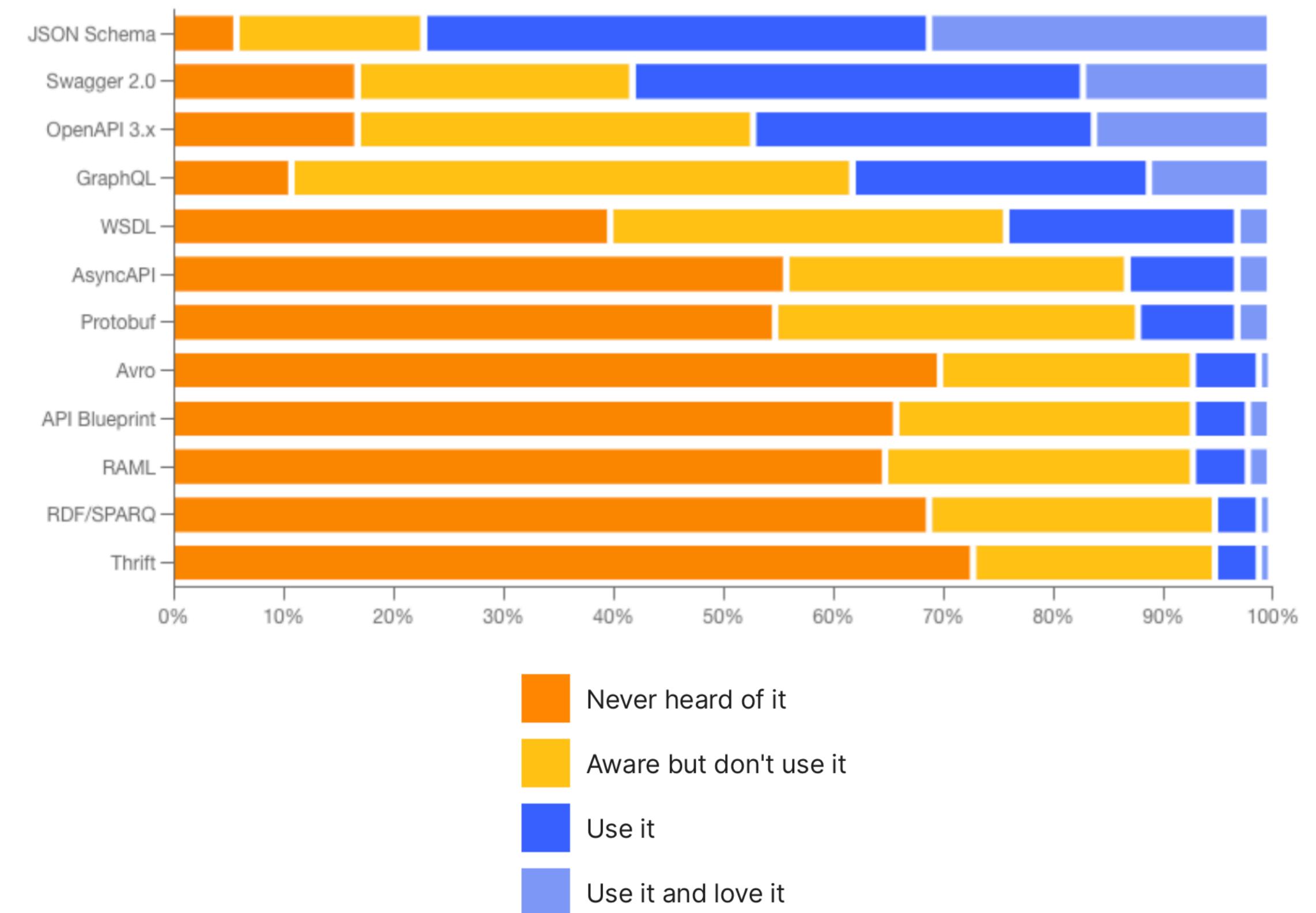
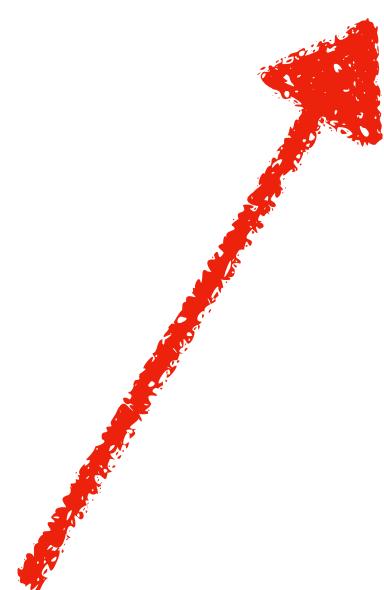


Source: Postman 2023 State of API Report

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

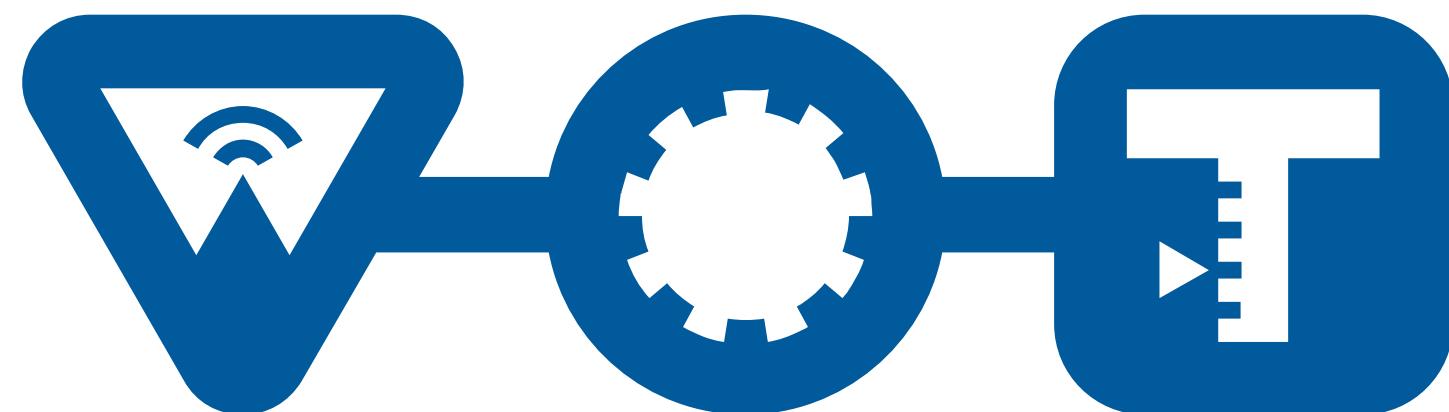
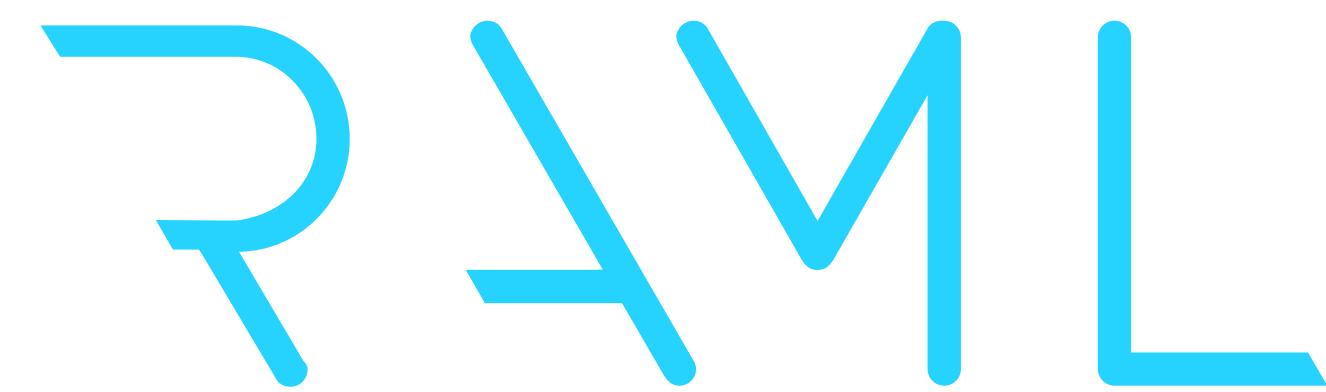
Are you using any of these?



OPENAPI



AsyncAPI



The OpenAPI Journey: Episode 1

The glamorous life of an API developer

The screenshot shows the Swagger Editor interface. On the left, a code editor displays the OpenAPI 3.0 specification for the Petstore API. The code is written in YAML and defines components like Pet, schemas for Pet, and responses for errors (400, 404). On the right, the Swagger Petstore interface is shown, featuring a navigation bar with 'Try our new Editor' and version information (1.0.11 OAS 3.0). Below the navigation, the 'Servers' dropdown is set to 'https://petstore3.swagger.io/api/v3'. The main content area shows the 'pet' resource with a 'GET /pet/{petId} Find pet by ID' operation. A 'Schemas' section below it contains a 'Pet' schema definition.

```
components:
  application/json:
    schema:
      $ref: '#/components/schemas/Pet'
  application/xml:
    schema:
      $ref: '#/components/schemas/Pet'
  '400':
    description: Invalid ID supplied
  '404':
    description: Pet not found
components:
schemas:
  Pet:
    required:
      - name
      - photoUrls
    type: object
    properties:
      id:
        type: integer
        format: int64
        example: 10
      name:
        type: string
        example: doggie
      photoUrls:
        type: array
        xml:
          wrapped: true
      items:
        type: string
        xml:
          name: photoUrl
      status:
        type: string
        description: pet status in the store
        enum:
          - available
          - pending
          - sold
      xml:
        name: pet
```

Swagger Petstore - OpenAPI 3.0 1.0.11 OAS 3.0

Servers
https://petstore3.swagger.io/api/v3

pet

GET /pet/{petId} Find pet by ID

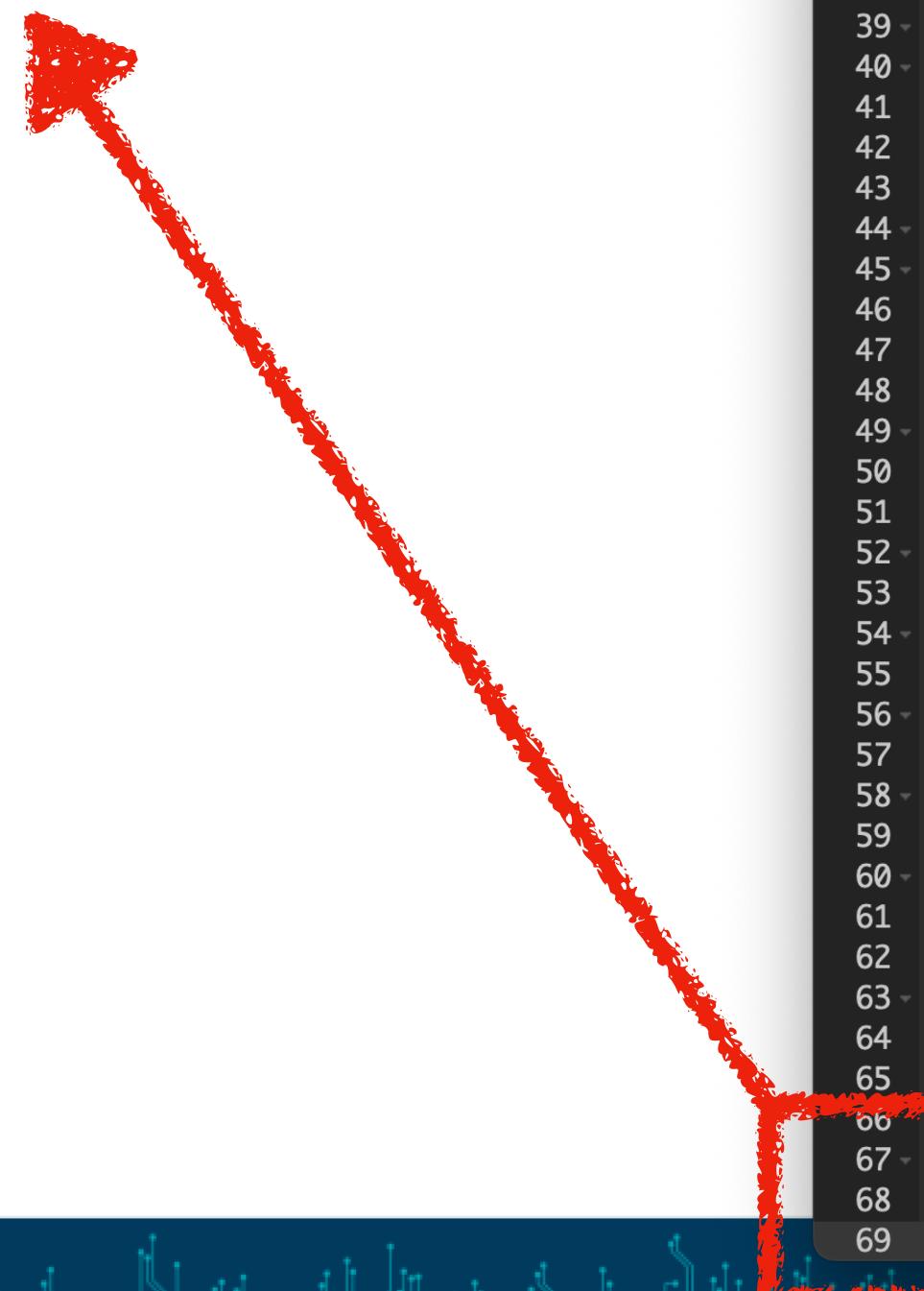
Schemas

Pet >

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 running in a web browser. The title bar reads "editor.swagger.io". The main area displays an OpenAPI 3.0 specification for a Petstore API. The YAML code is as follows:

```
components:
  schemas:
    Pet:
      required:
        - name
        - photoUrls
      type: object
      properties:
        id:
          type: integer
          format: int64
          example: 10
        name:
          type: string
          example: doggie
        photoUrls:
          type: array
          xml:
            wrapped: true
          items:
            type: string
            xml:
              name: photoUrl
        status:
          type: string
          description: pet status in the store
          enum:
            - available
            - pending
            - sold
          xml:
            name: pet
```

To the right of the code editor, the Swagger Petstore interface is visible, showing the "pet" resource with a GET method for "/pet/{petId}". Below it, the "Schemas" section shows a "Pet" schema. The top right of the interface has a green button that says "Try our new Editor".

The OpenAPI Journey: Episode 2

The good life of an API developer

The screenshot shows the Swagger Editor interface running in a web browser. The title bar indicates it's on editor.swagger.io. The main area displays the **Swagger Petstore - OpenAPI 3.0** specification, version 1.0.11, OAS 3.0. On the left, there's a code editor window showing the JSON schema for a Pet object, with line numbers from 208 to 250. The schema includes properties like name, type, id, and photoUrls. On the right, the API documentation is displayed under the **pet** resource. It lists five operations: 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 API documentation is a **Schemas** section, which currently shows a Category schema. The entire interface has a modern, clean design with a dark header and light body.

```
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
```

Swagger Petstore - OpenAPI 3.0 1.0.11 OAS 3.0

Servers
https://petstore3.swagger.io/api/v3

pet

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 >

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 for the Petstore API. On the left, the `Pet` schema is displayed in YAML format, containing definitions for name, type, XML, Pet, required fields (name, photoUrls), type (object), properties (id, type integer, format int64, example 10), name, type string, example doggie, category (\$ref to Category schema), photoUrls, type array, wrapped true, items, type string, XML, name photoUrl, tags, type array, wrapped true, items, \$ref to Tag schema, status, type string, description pet status in the store, enum (available, pending, sold), XML, name pet. On the right, the `pet` resource is listed with five operations: 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). The interface also includes a Servers dropdown set to `https://petstore3.swagger.io/api/v3` and a Schemas section showing the Category schema.

The OpenAPI Journey: Episode 3

The decent life of an API developer

The screenshot shows the Swagger Editor interface running in a web browser. The left side features a code editor with the OpenAPI 3.0 specification for the Petstore API, displaying various definitions like `ApiResponse`, `Pet`, and `User`. The right side shows the generated API documentation for the `pet` endpoint, including a `PUT /pet` operation to update an existing pet and a `POST /pet` operation to add a new pet to the store.

Swagger Petstore - OpenAPI 3.0 1.0.11 OAS 3.0

This is a sample Pet Store Server based on the OpenAPI 3.0 specification. You can find out more about Swagger at <https://swagger.io>. In the third iteration of the pet store, we've switched to the design first approach! You can now help us improve the API whether it's by making changes to the definition itself or to the code. That way, with time, we can improve the API in general, and expose some of the new features in OAS3.

If you're looking for the Swagger 2.0/OAS 2.0 version of Petstore, then click [here](#). Alternatively, you can load via the [Edit > Load Petstore OAS 2.0](#) menu option!

Some useful links:

- [The Pet Store repository](#)
- [The source API definition for the Pet Store](#)

Terms of service
Contact the developer
Apache 2.0
Find out more about Swagger

Servers [Authorize](#)

pet Everything about your Pets [Find out more](#)

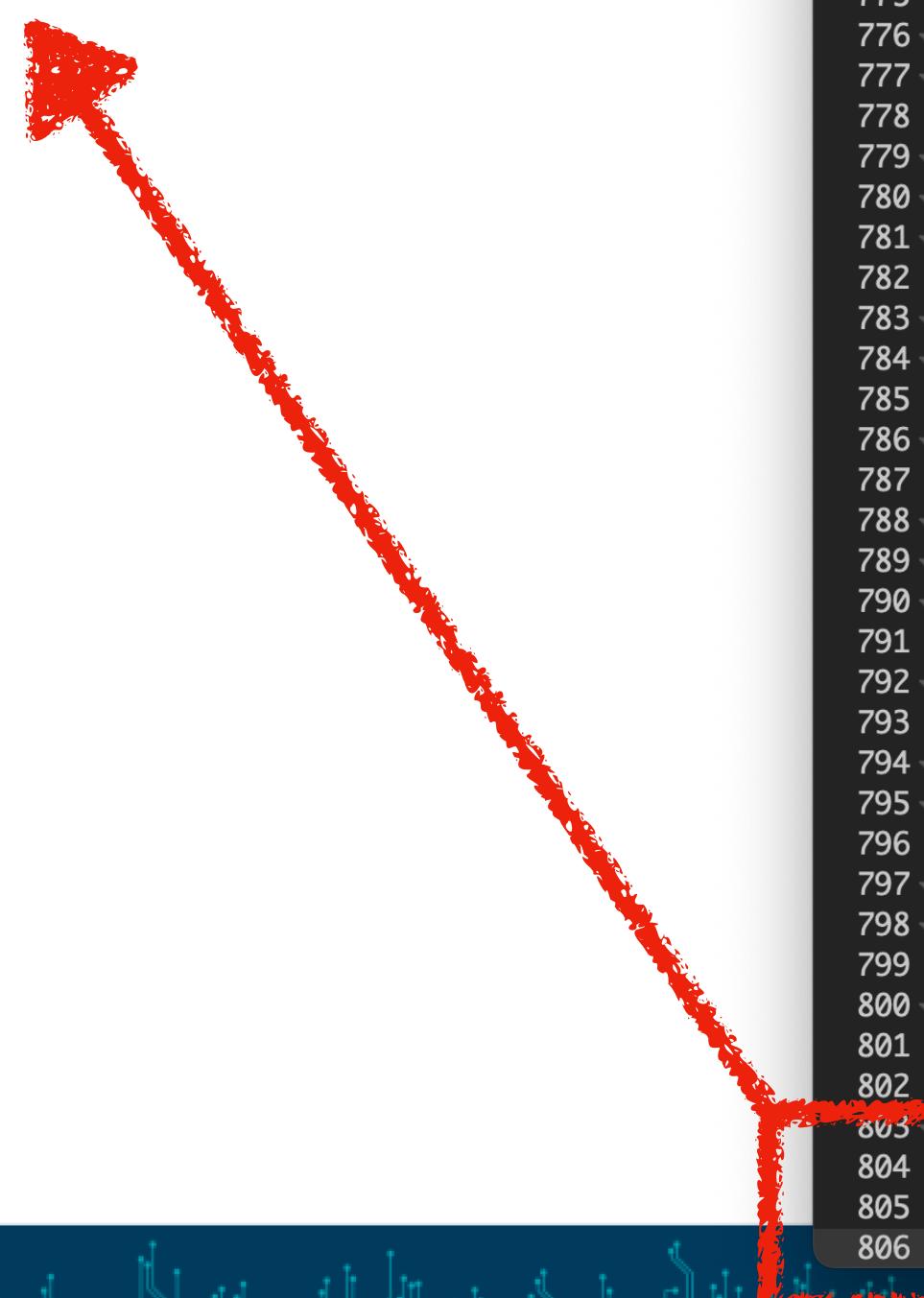
PUT `/pet` Update an existing pet

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



editor.swagger.io

Try our new Editor ↗

Swagger Petstore - OpenAPI 3.0 1.0.11 OAS 3.0

This is a sample Pet Store Server based on the OpenAPI 3.0 specification. You can find out more about Swagger at <https://swagger.io>. In the third iteration of the pet store, we've switched to the design first approach! You can now help us improve the API whether it's by making changes to the definition itself or to the code. That way, with time, we can improve the API in general, and expose some of the new features in OAS3.

If you're looking for the Swagger 2.0/OAS 2.0 version of Petstore, then click [here](#). Alternatively, you can load via the [Edit > Load Petstore OAS 2.0](#) menu option!

Some useful links:

- [The Pet Store repository](#)
- [The source API definition for the Pet Store](#)

Terms of service
Contact the developer
Apache 2.0
[Find out more about Swagger](#)

Servers

https://petstore3.swagger.io/api/v3 ▾

Authorize 🔒

pet Everything about your Pets ▾

Find out more

PUT /pet Update an existing pet

POST /pet Add a new pet to the store

🔒 ▾

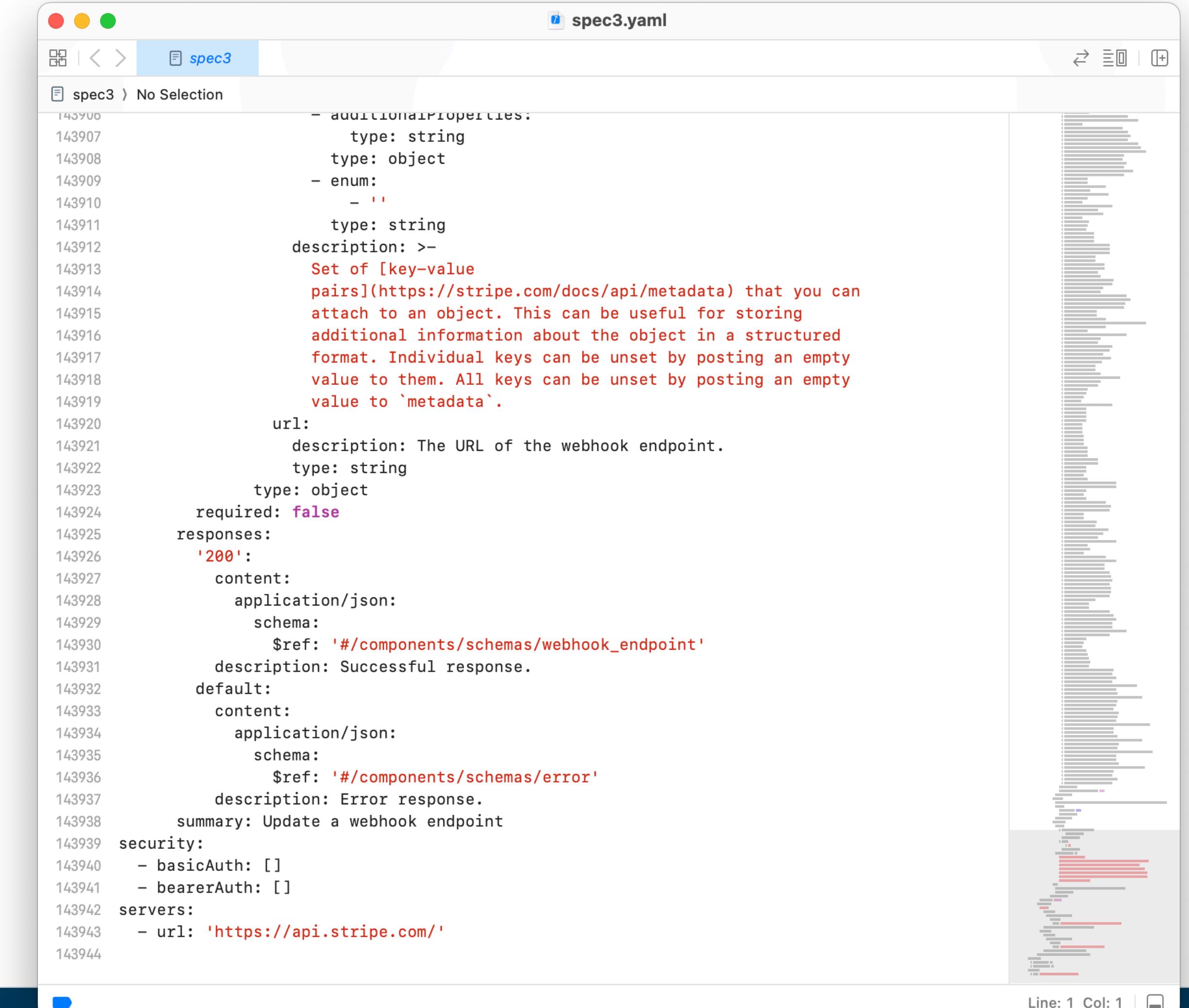
🔒 ▾

in: header

```
name: pet
  ApiResponse:
    type: object
    properties:
      code:
        type: integer
        format: int32
      type:
        type: string
      message:
        type: string
      xml:
        name: '#default'
  requestBodies:
    Pet:
      description: Pet object that needs to be added to the store
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Pet'
        application/xml:
          schema:
            $ref: '#/components/schemas/Pet'
  UserArray:
    description: List of user object
    content:
      application/json:
        schema:
          type: array
          items:
            $ref: '#/components/schemas/User'
  securitySchemes:
    petstore_auth:
      type: oauth2
      flows:
        implicit:
          authorizationUrl: https://petstore3.swagger.io/oauth/authorize
          scopes:
            write:pets: modify pets in your account
            read:pets: read your pets
      api_key:
        type: apiKey
        name: api_key
        in: header
```

The OpenAPI Journey: Final Episode

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



```
spec3.yaml
spec3 > No Selection
spec3
- additionalProperties:
  - type: string
  - type: object
  - enum:
    - ''
  - type: string
  description: >-
    Set of [key-value
    pairs](https://stripe.com/docs/api/metadata) that you can
    attach to an object. This can be useful for storing
    additional information about the object in a structured
    format. Individual keys can be unset by posting an empty
    value to them. All keys can be unset by posting an empty
    value to `metadata`.
url:
  description: The URL of the webhook endpoint.
  type: string
  type: object
  required: false
responses:
  '200':
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/webhook_endpoint'
        description: Successful response.
    default:
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/error'
        description: Error response.
    summary: Update a webhook endpoint
  security:
    - basicAuth: []
    - bearerAuth: []
servers:
  - url: 'https://api.stripe.com/'
```

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



spec3.yaml

github.com/stripe/openapi/blob/master/openapi/spec3

stripe / openapi

Code Issues 14 Pull requests Actions Projects Security Insights

master openapi / openapi / spec3.yaml

stripe-openapi[bot] Update OpenAPI specification ✓ b2893a0 · last week History

Code Blame 5.13 MB View raw

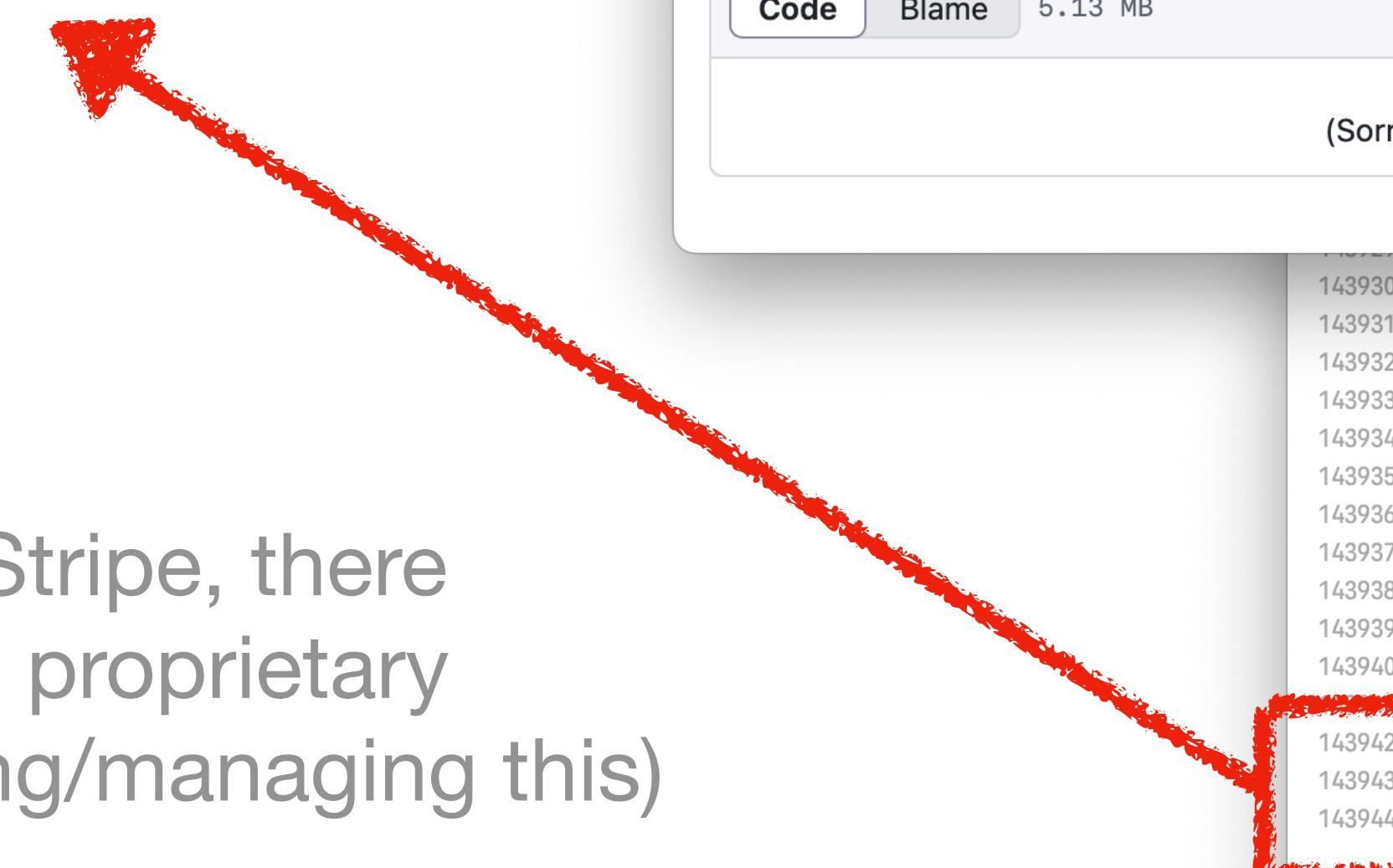
(Sorry about that, but we can't show files that are this big right now.)

```
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
```

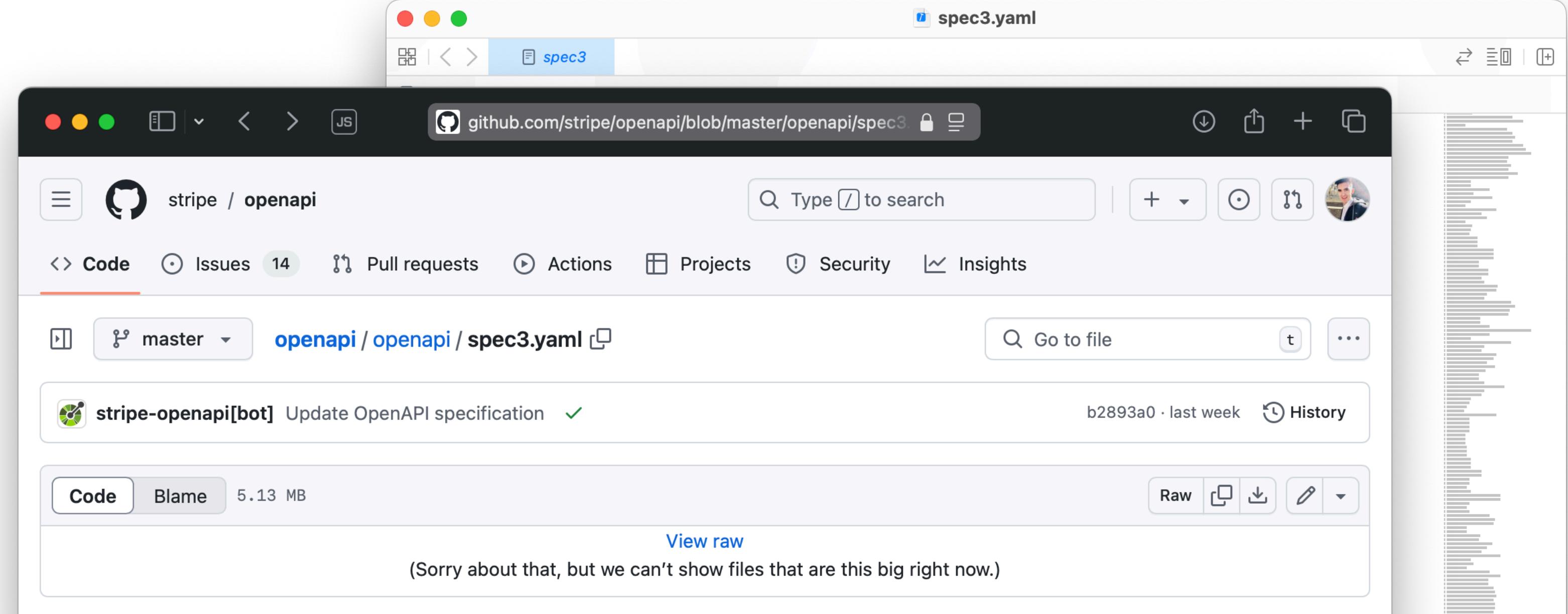
The OpenAPI Journey: Final Episode

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)



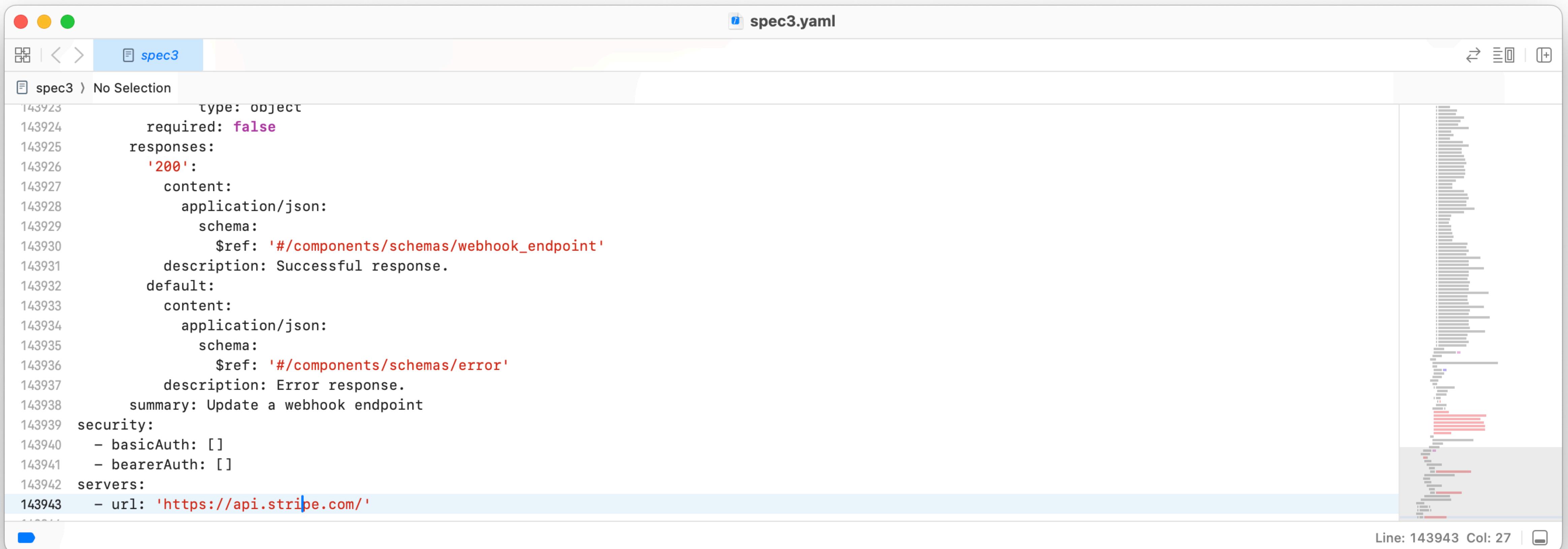
The screenshot shows a GitHub repository for Stripe's OpenAPI specification. The main page displays a large file named "spec3.yaml" with a size of 5.13 MB. A message indicates that the file is too large to be previewed. Below the message, a portion of the YAML code is visible, showing definitions for webhook endpoints, error responses, and security configurations. A specific line of code is highlighted with a red box.

```
        $ref: '#/components/schemas/webhook_endpoint'
        description: Successful response.
      default:
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/error'
              description: Error response.
            summary: Update a webhook endpoint
      security:
        - basicAuth: []
        - bearerAuth: []
    servers:
      - url: 'https://api.stripe.com/'
```

KEY OBSERVATION

>80% of this is JSON Schema

Either as re-usable components or inlined in path definitions



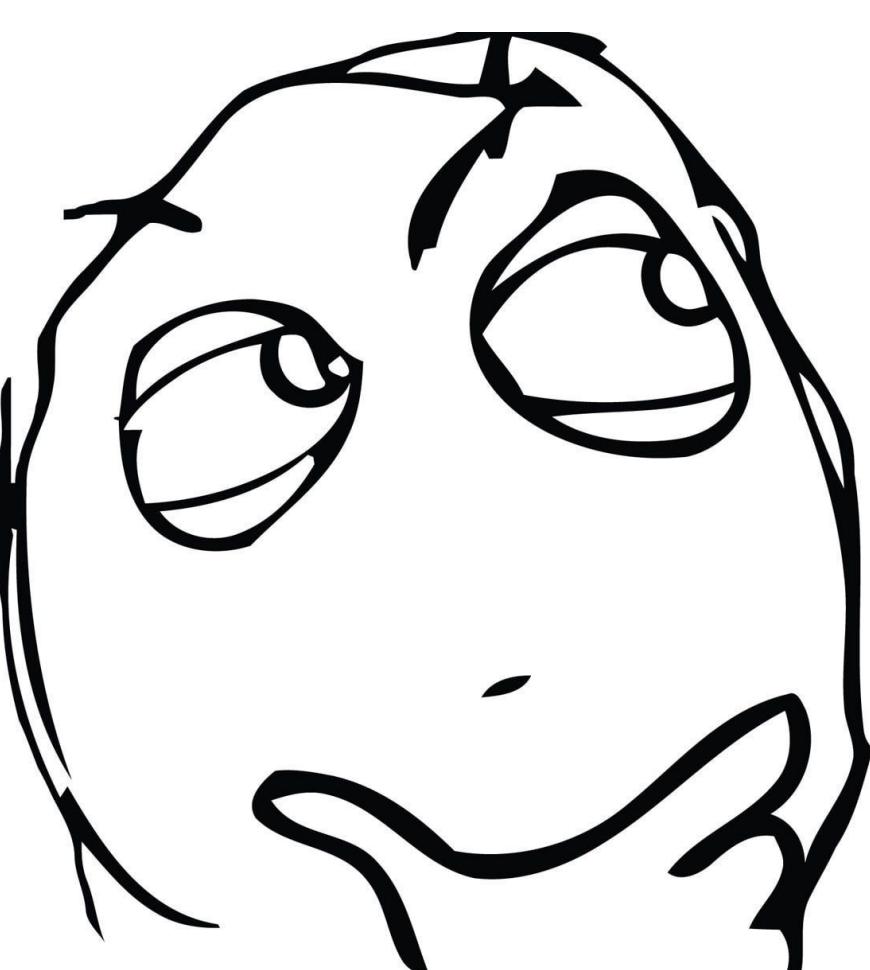
```
spec3.yaml
spec3
spec3 > No Selection
143923     type: object
143924     required: false
143925     responses:
143926       '200':
143927         content:
143928           application/json:
143929             schema:
143930               $ref: '#/components/schemas/webhook_endpoint'
143931             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



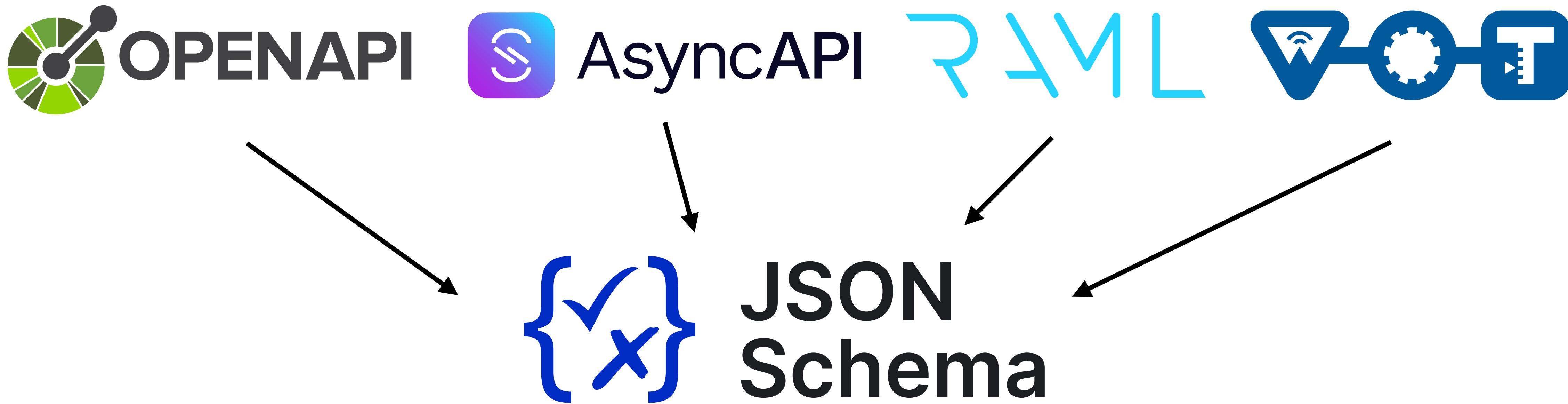
A screenshot of a code editor window titled "spec3.yaml". The code is a YAML file defining a schema for a webhook endpoint. The editor has a dark theme with syntax highlighting. A vertical scroll bar is visible on the right side of the window.

```
spec3
  type: object
  required: false
  responses:
    '200':
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/webhook_endpoint'
          description: Successful response.
    default:
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/error'
          description: Error response.
  summary: Update a webhook endpoint
  security:
    - basicAuth: []
    - bearerAuth: []
  servers:
    - url: 'https://api.stripe.com/'
```

Line: 143943 Col: 27

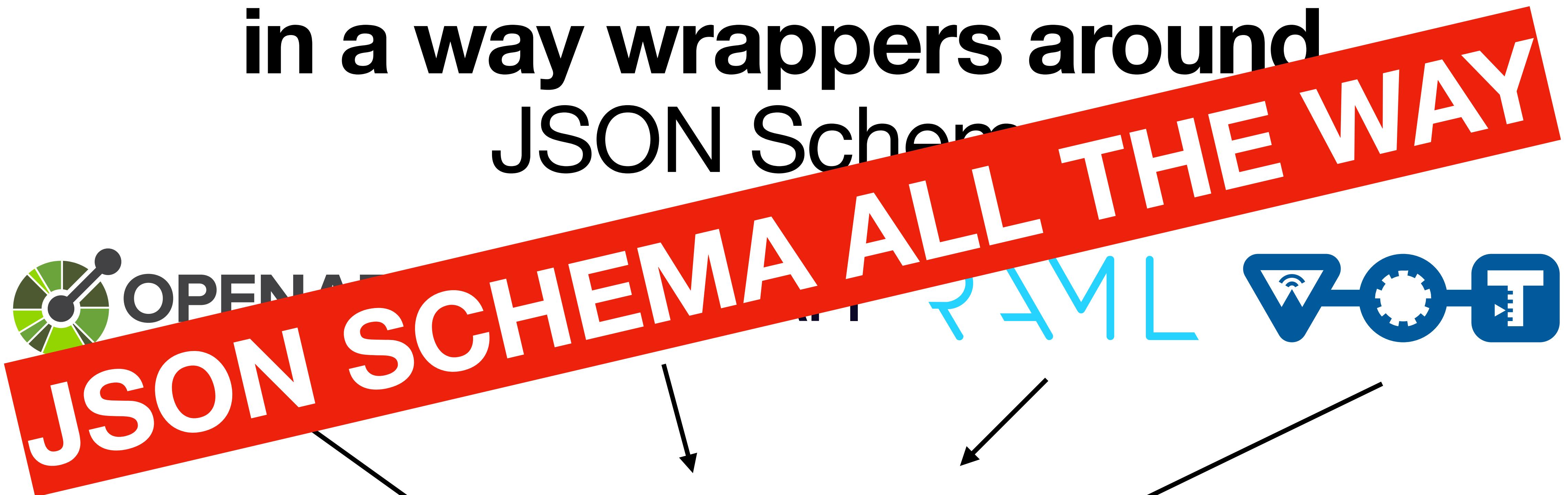
TAKING IT ONE STEP FURTHER...

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

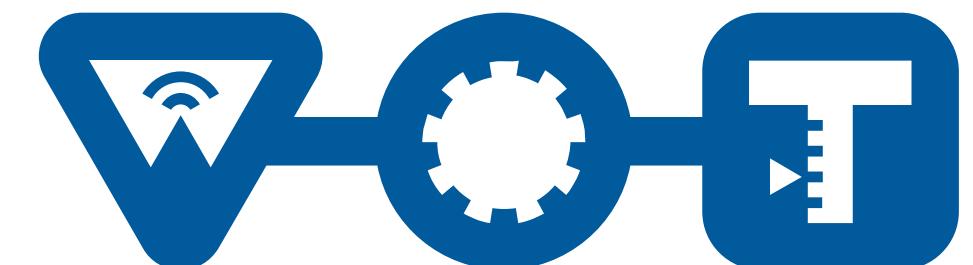
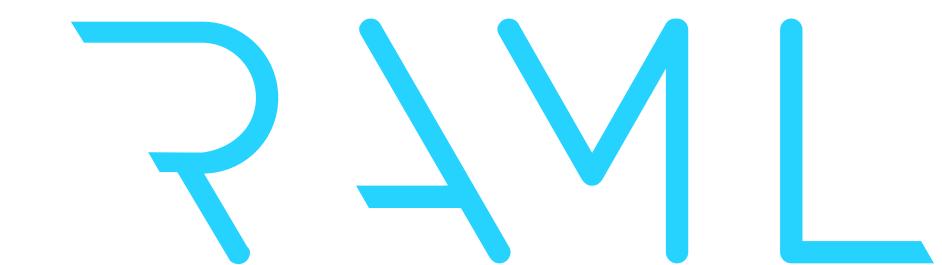


JSON
Schema

If you want to improve your
API specifications...



AsyncAPI



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



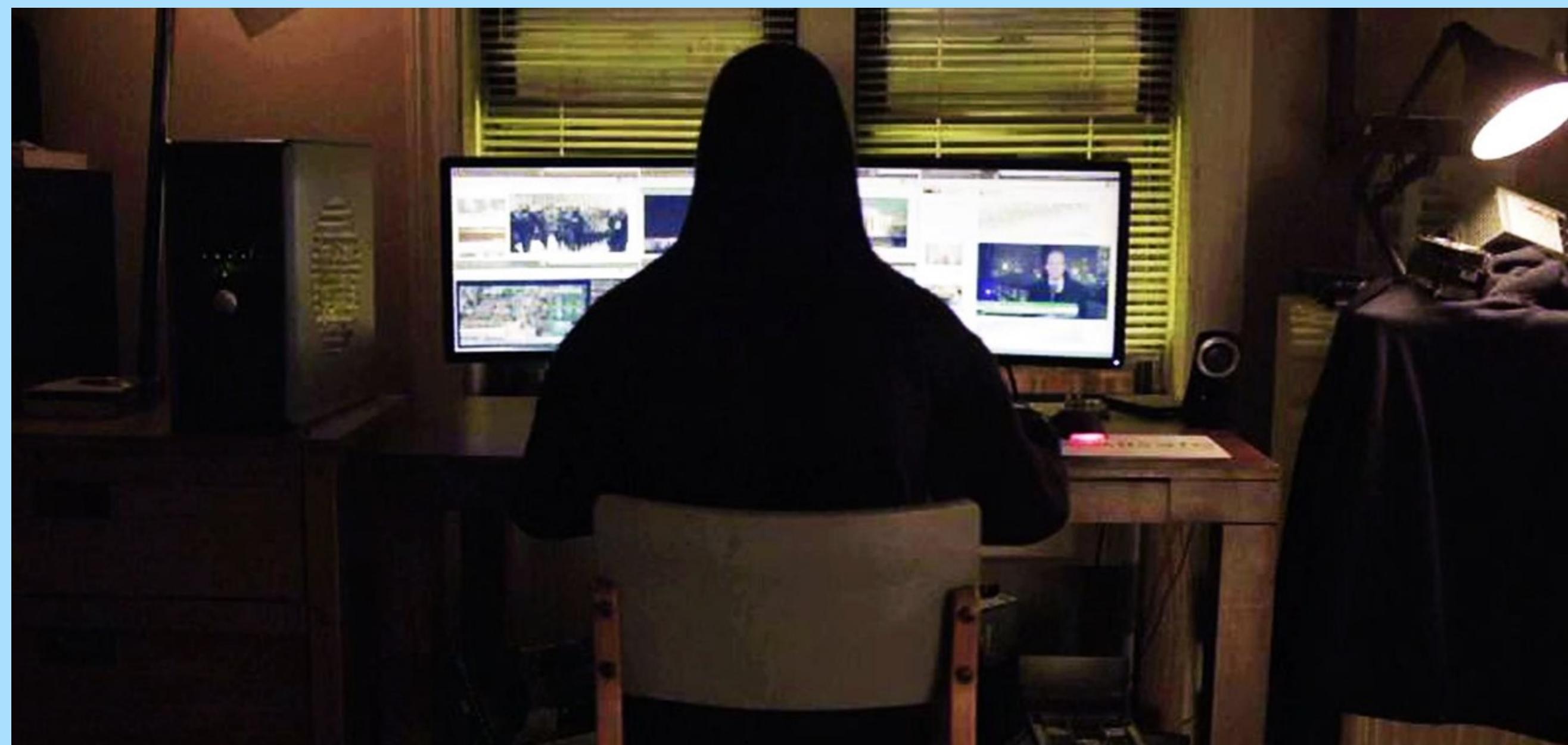
The JSON Schema first approach to API specifications

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

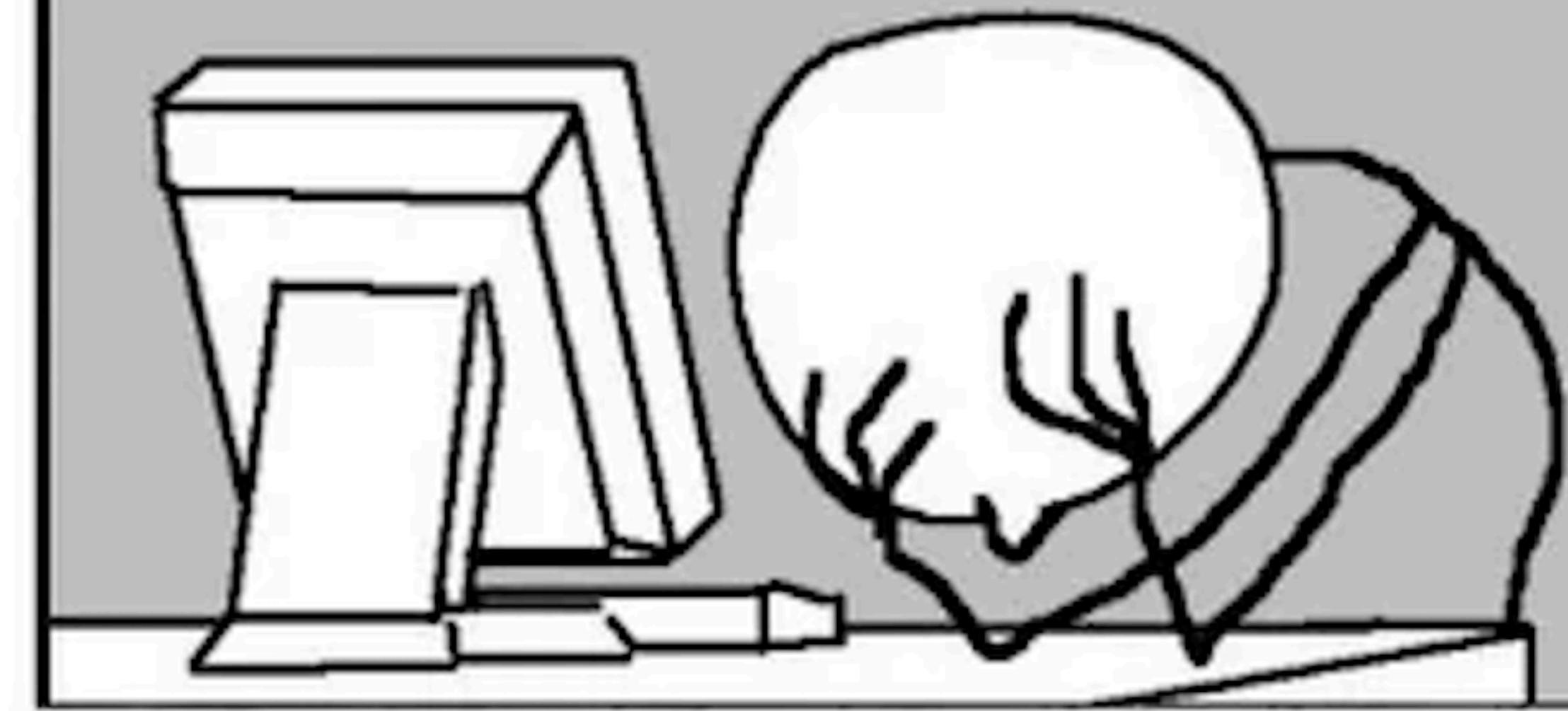


The JSON Schema first approach to API specifications

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

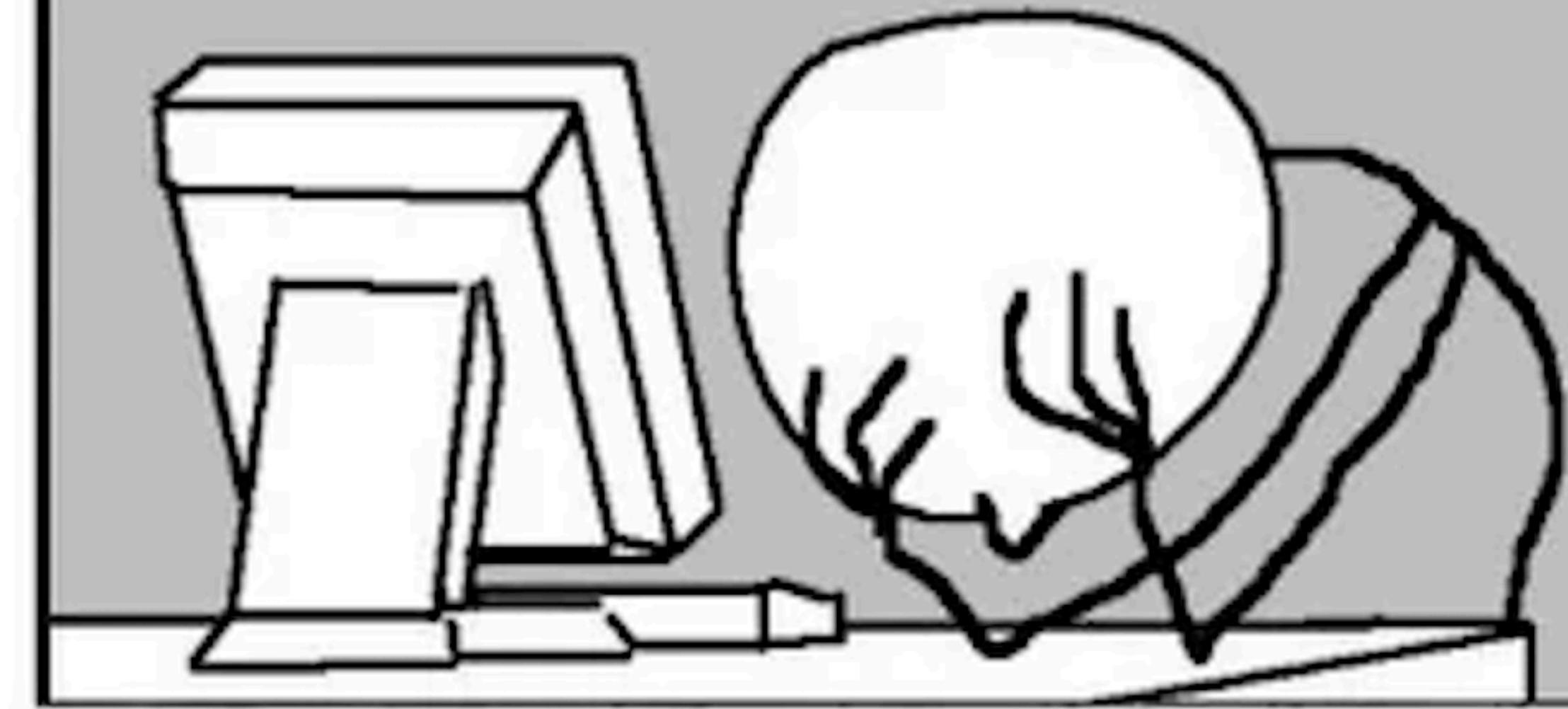


**2000 LINES OF
CODE IN A SINGLE FILE**



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

**2000 LINES OF
CODE IN A SINGLE FILE**



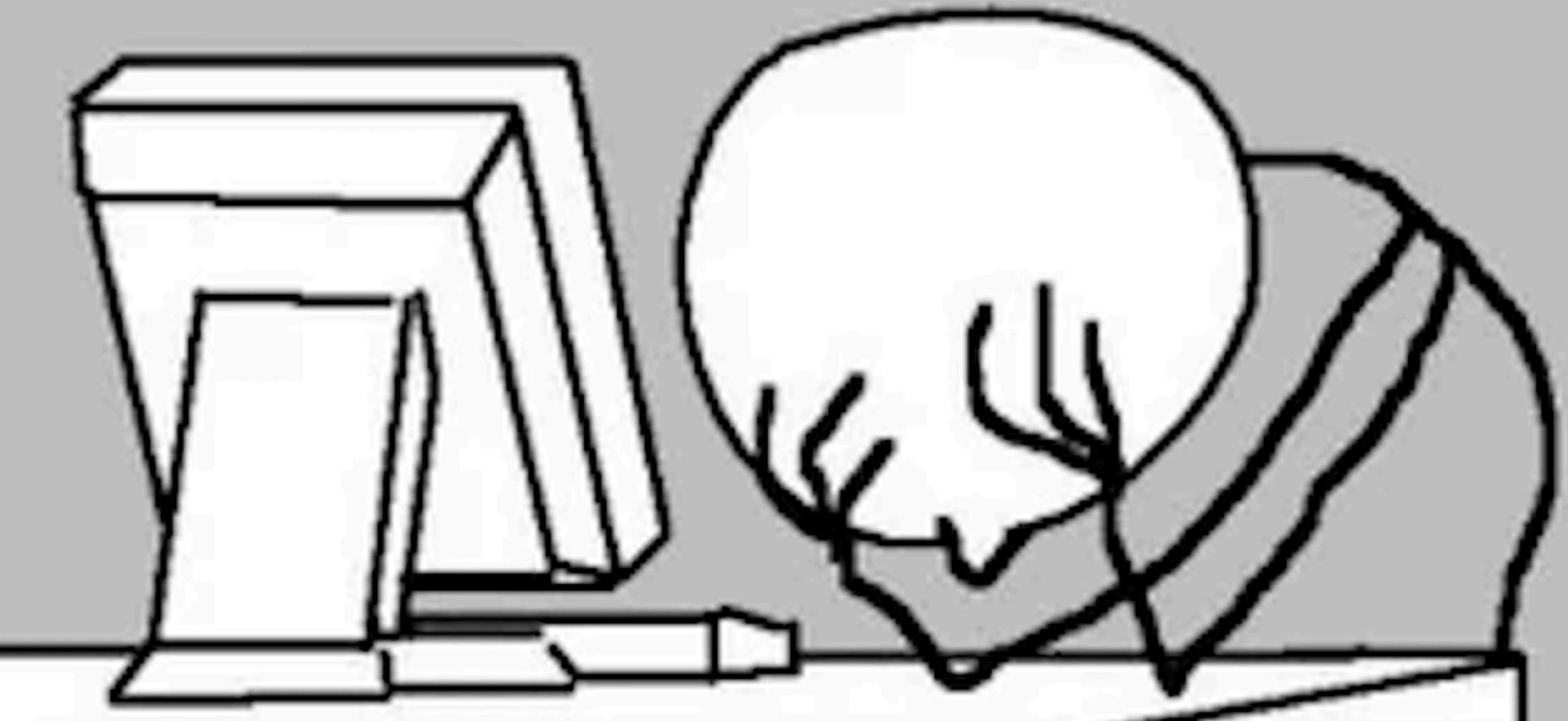
Surely you don't write all your projects as single huge code files

What about your OpenAPIs?

The JSON Schema first approach: Step #1

Extract your JSON Schemas as individual files on a GitHub repo

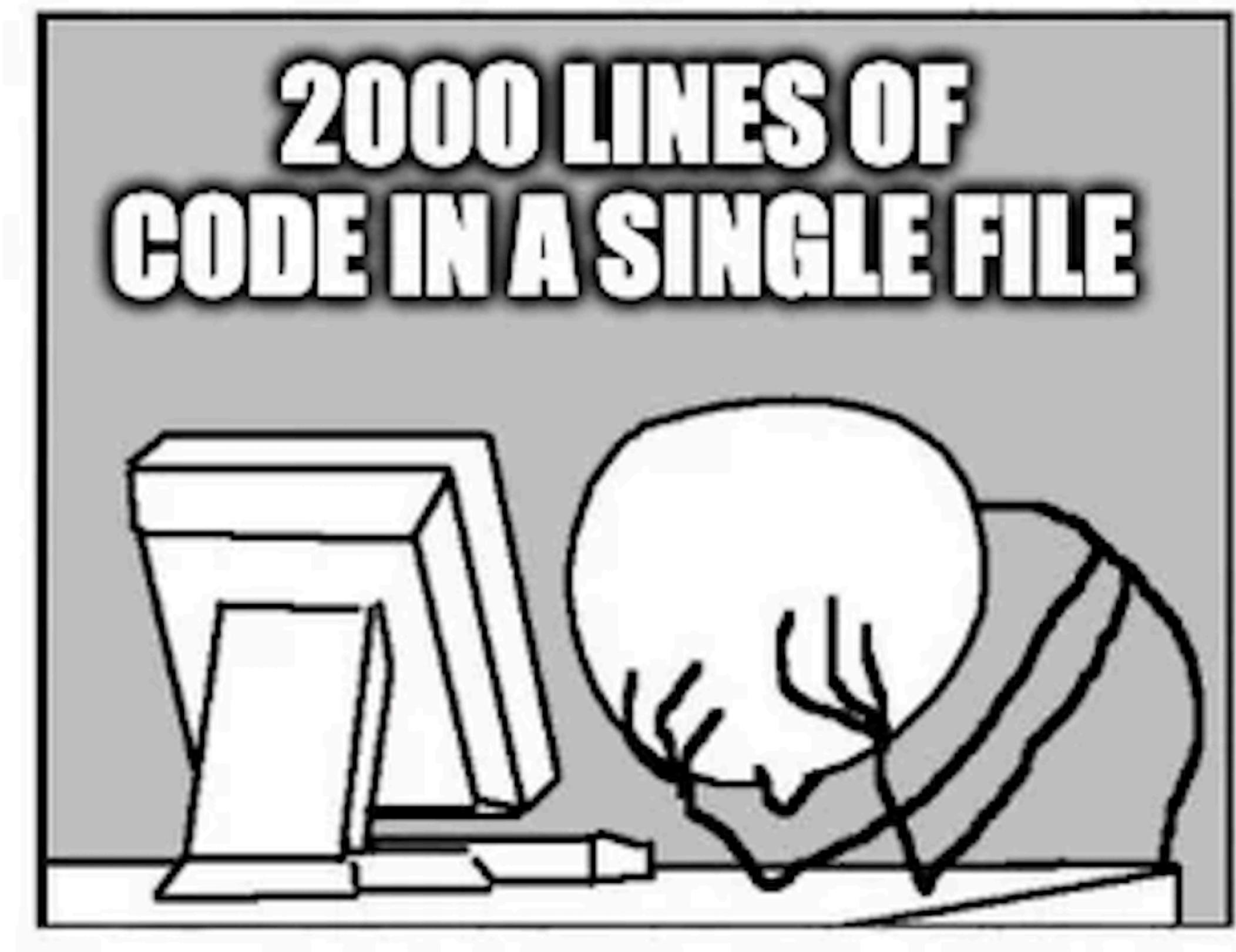
**2000 LINES OF
CODE IN A SINGLE FILE**



The JSON Schema first approach: Step #1

Extract your JSON Schemas as individual files on a GitHub repo

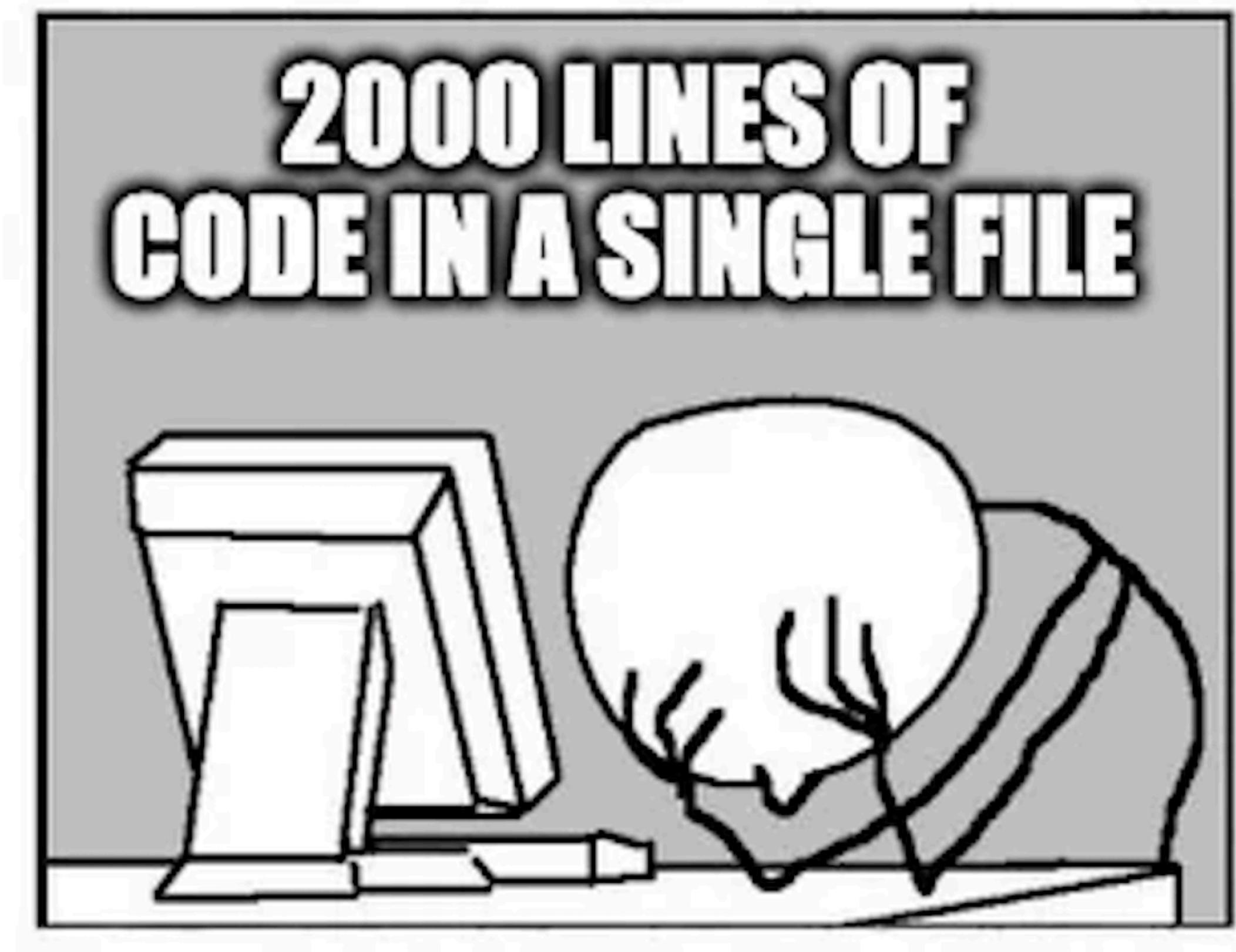
- Otherwise, its a lot harder to do everything within the constraints of i.e. an OpenAPI wrapper!



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



The screenshot shows a web browser window displaying a blog post. The title is "Help You Manage Your JSON Schema via a GitHub Repository" by Kin Lane, written on November 15, 2024. The post features a green-toned background image of a complex circuit board or bridge structure. A large red rectangular box highlights the beginning of the post's content, which discusses API governance services and JSON Schema management. Below this, another red box highlights a section about starting with fundamentals. At the bottom, there is a note about helping customers get started with schema management in GitHub.

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 GitHub repository interfaces side-by-side, each showing a different organization's approach to managing JSON schema files.

- Left Repository (asyncapi/spec-json-schemas):** This repository contains several JSON schema files under the `spec-json-schemas` directory. The files include `1.0.0-without-$id.json`, `1.0.0.json`, `1.1.0-without-$id.json`, `1.1.0.json`, `1.2.0-without-$id.json`, `1.2.0.json`, `2.0.0-rc1-without-$id.json`, and `2.0.0-rc1.json`. The repository has 6 issues and 6 pull requests.
- Middle Repository (krakend/krakend-schema):** This repository contains a series of schema files named `v2.1`, `v2.2`, `v2.3`, `v2.4`, `v2.5`, `v2.6`, and `v2.7`. Each file is associated with a commit message such as "Fix sever", "Merge p", or "Commit". The repository has 11 issues and 13 pull requests.
- Right Repository (nasa-gcn/gcn-schema):** This repository contains schema files for various NASA GCN notices. The files are organized into a tree structure under `gcn-schema/gcn/notices/core`. Examples include `AdditionalInfo.schema.json`, `Alert.schema.json`, `DateTime.schema.json`, `DetectorStatus.schema.json`, `Distance.schema.json`, and `Duration.schema.json`. The repository has 11 issues and 13 pull requests.

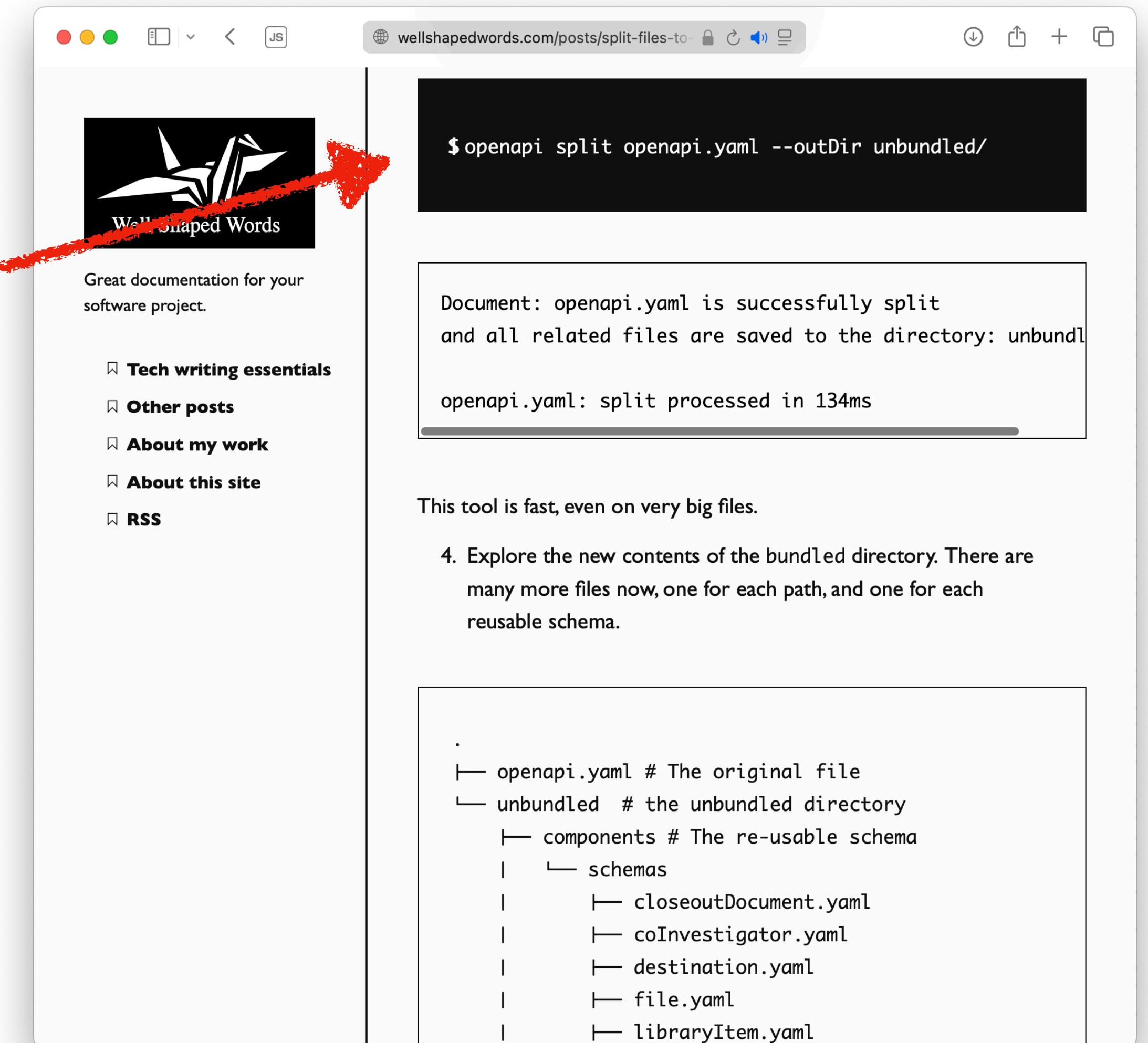
Logos for AsyncAPI, krakenD, and NASA are positioned above their respective repository screenshots.

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/>



A screenshot of a Mac OS X desktop showing a browser window and a terminal window. The browser window displays the homepage of 'wellshapedwords.com' with a logo and navigation links. The terminal window shows the command '\$ openapi split openapi.yaml --outDir unbundled/' being run, followed by the output: 'Document: openapi.yaml is successfully split and all related files are saved to the directory: unbundled'. Below this, it says 'openapi.yaml: split processed in 134ms'. At the bottom, it states 'This tool is fast, even on very big files.' and lists step 4: '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.' The terminal output also shows a file tree for the 'unbundled' directory.

```
$ openapi split openapi.yaml --outDir unbundled/
Document: openapi.yaml is successfully split
and all related files are saved to the directory: unbundled
openapi.yaml: split processed in 134ms

This tool is fast, even on very big files.

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.

.
├── 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



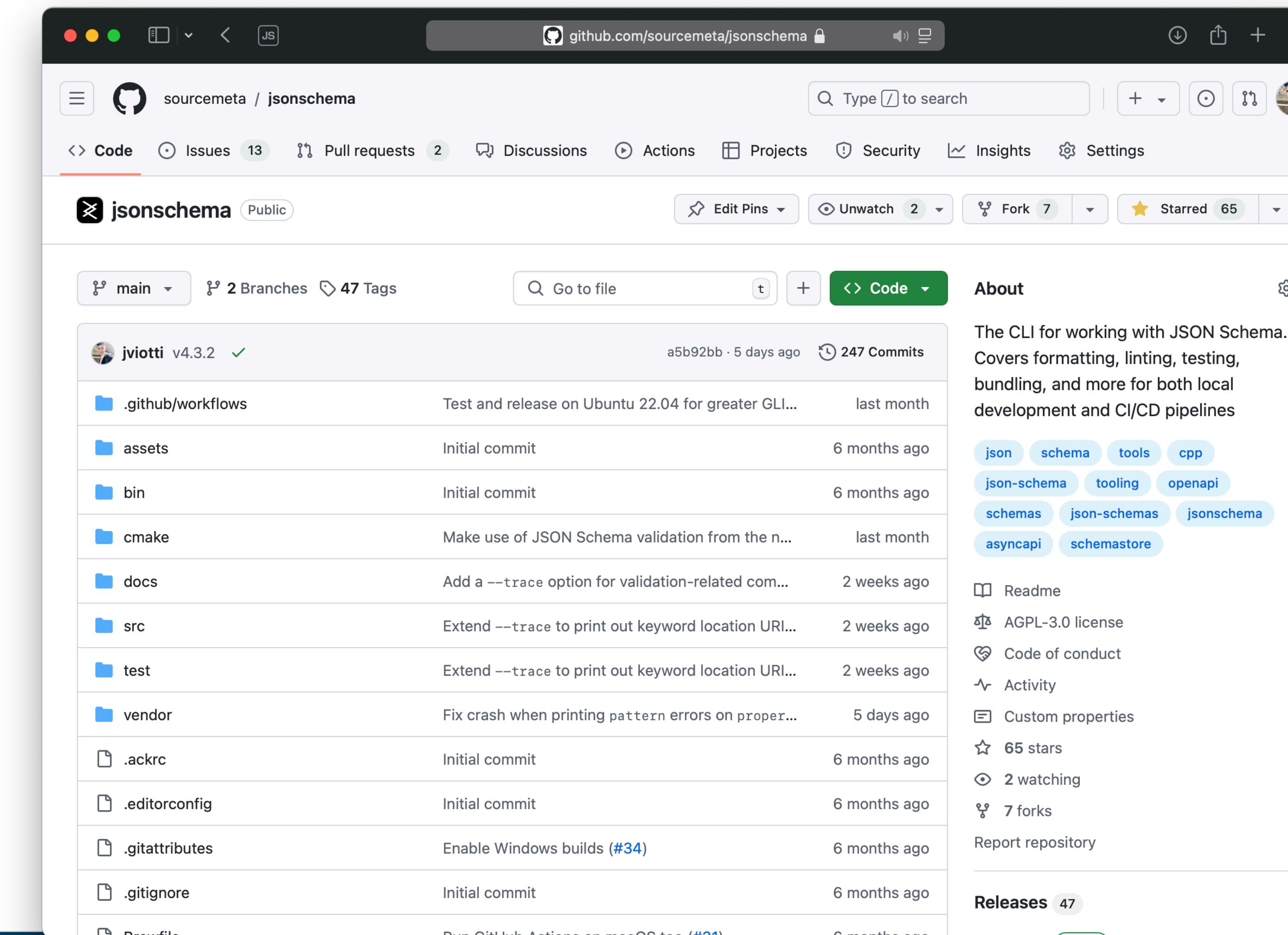
The JSON Schema first approach: Step #2

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/sourcmeta/jsonschema>



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 name 'jsonschema' is displayed prominently at the top. Below the repository name, there are tabs for 'Code', 'Issues' (13), 'Pull requests' (2), 'Discussions', 'Actions', 'Projects', 'Security', 'Insights', and 'Settings'. A search bar is located at the top right. The main area shows a list of commits by user 'jviotti' (v4.3.2) with a green checkmark, dated 'a5b92bb · 5 days ago'. The commit message is '247 Commits'. Below the commits, there are links to '.github/workflows', 'assets', 'bin', 'cmake', 'docs', 'src', 'test', and 'vendor'. On the right side of the repository page, there is a sidebar with sections for 'About', 'The CLI for working with JSON Schema. Covers formatting, linting, testing, bundling, and more for both local development and CI/CD pipelines', 'Tags' (47), 'Releases' (47), 'Readme', 'AGPL-3.0 license', 'Code of conduct', 'Activity', 'Custom properties', '65 stars', '2 watching', '7 forks', and 'Report repository'.

The JSON Schema first approach: Step #2

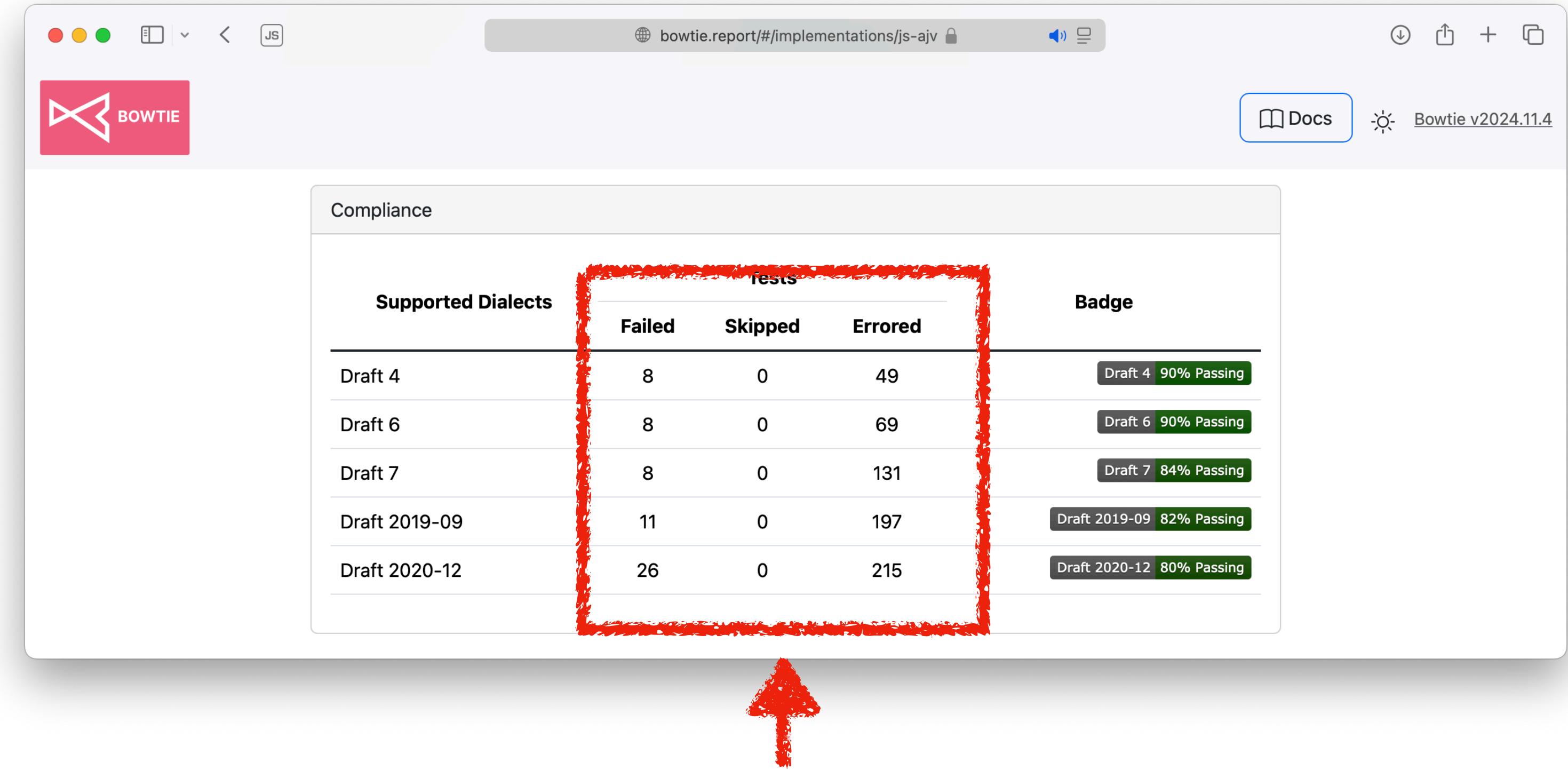
Find a fully-compliant tool to work with JSON Schema



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

The JSON Schema first approach: Step #2

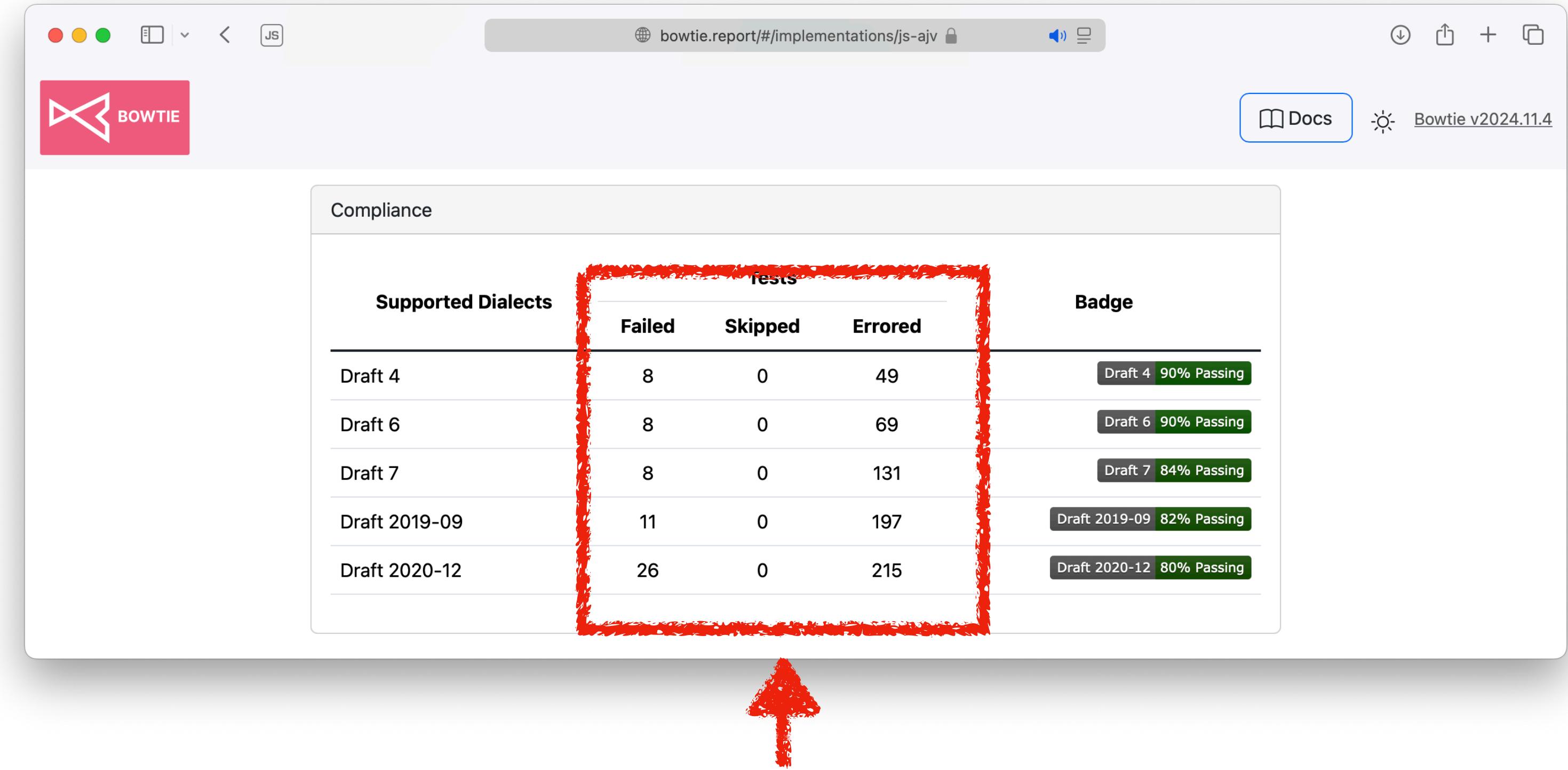
Find a fully-compliant tool to work with JSON Schema



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



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

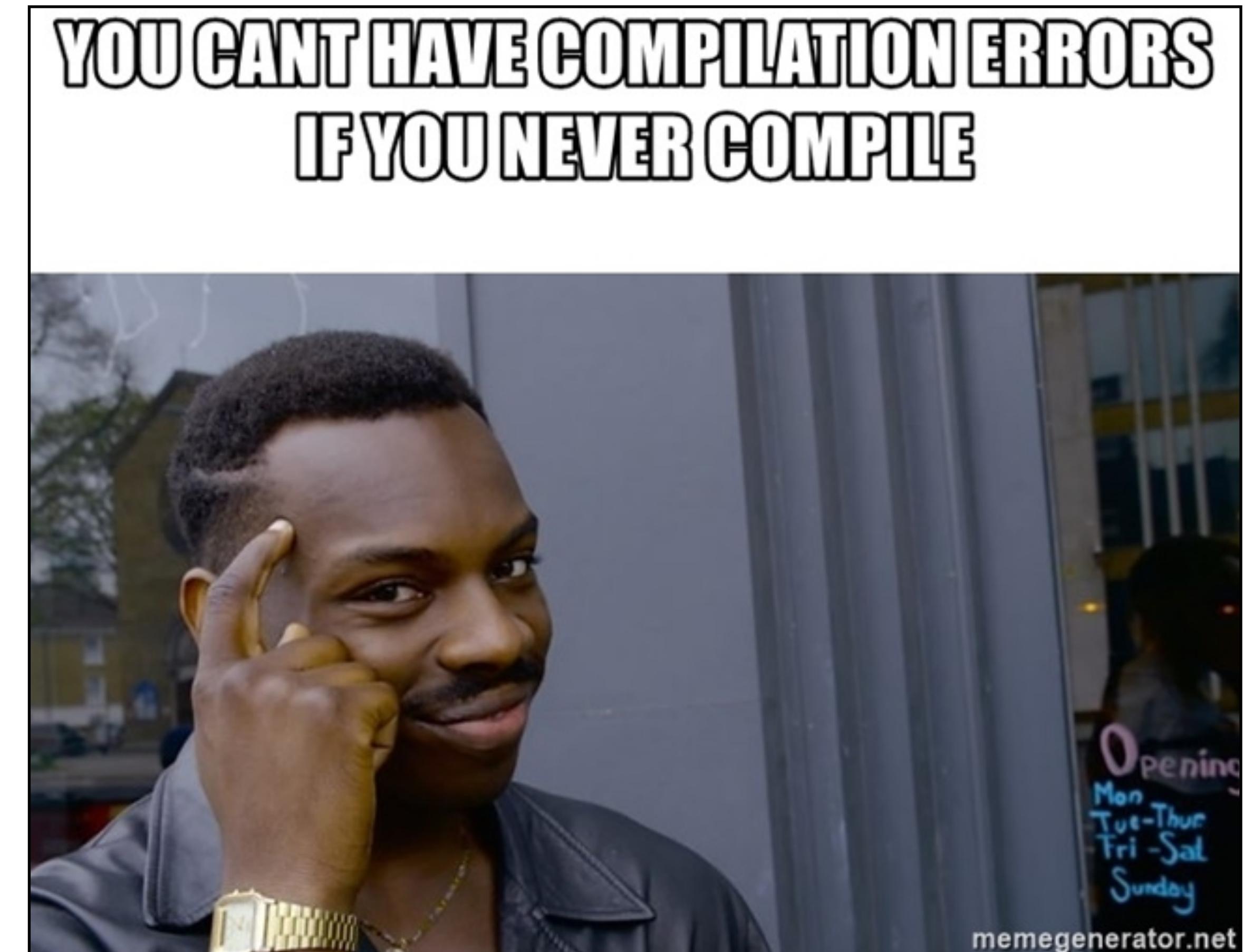
This is analogous to
checking if your code
actually compiles



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/$dynamicRef/allOf"  
  The array value was expected to validate against the first subschema in scope that declared the dy  
    at instance location "/items"  
      at evaluate path "/allOf/1/$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



{

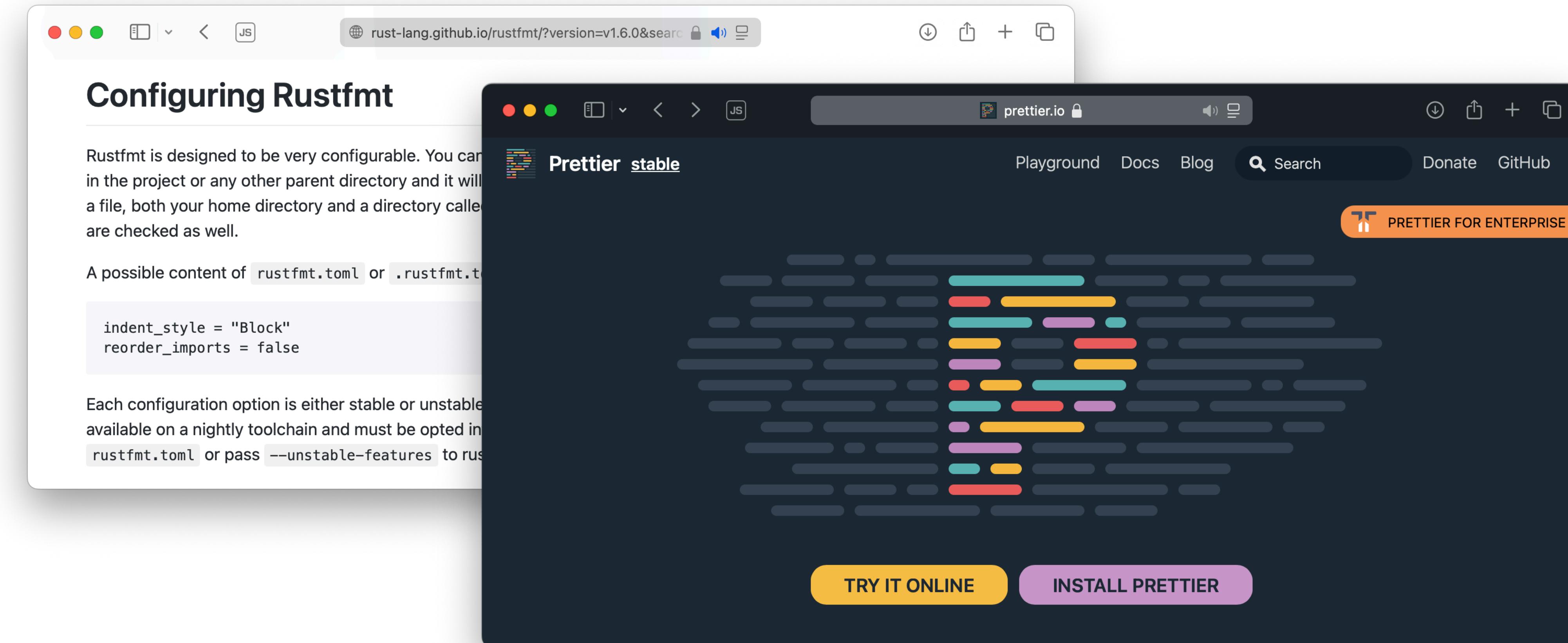
```
$schema: "https://json-schema.org/draft/2020-12/schema",
"items": [
  { "type": "string" }
]
```

}

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

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/$dynamicRef/allOf"
The array value was expected to validate against the first subschema in scope that declared the dy
  at instance location "/items"
    at evaluate path "/allOf/1/$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"
```



Configuring Rustfmt

Rustfmt is designed to be very configurable. You can put configuration files in the project or any other parent directory and it will check all files in that directory, both your home directory and a directory called `.rustfmt.toml` if present. Configuration files are checked as well.

A possible content of `rustfmt.toml` or `.rustfmt.toml`:

```
indent_style = "Block"
reorder_imports = false
```

Each configuration option is either stable or unstable. Stable options are available on all toolchains. Unstable options are available on a nightly toolchain and must be opted in by setting the `unstable-options` field in `rustfmt.toml` or pass `--unstable-features` to rustfmt.

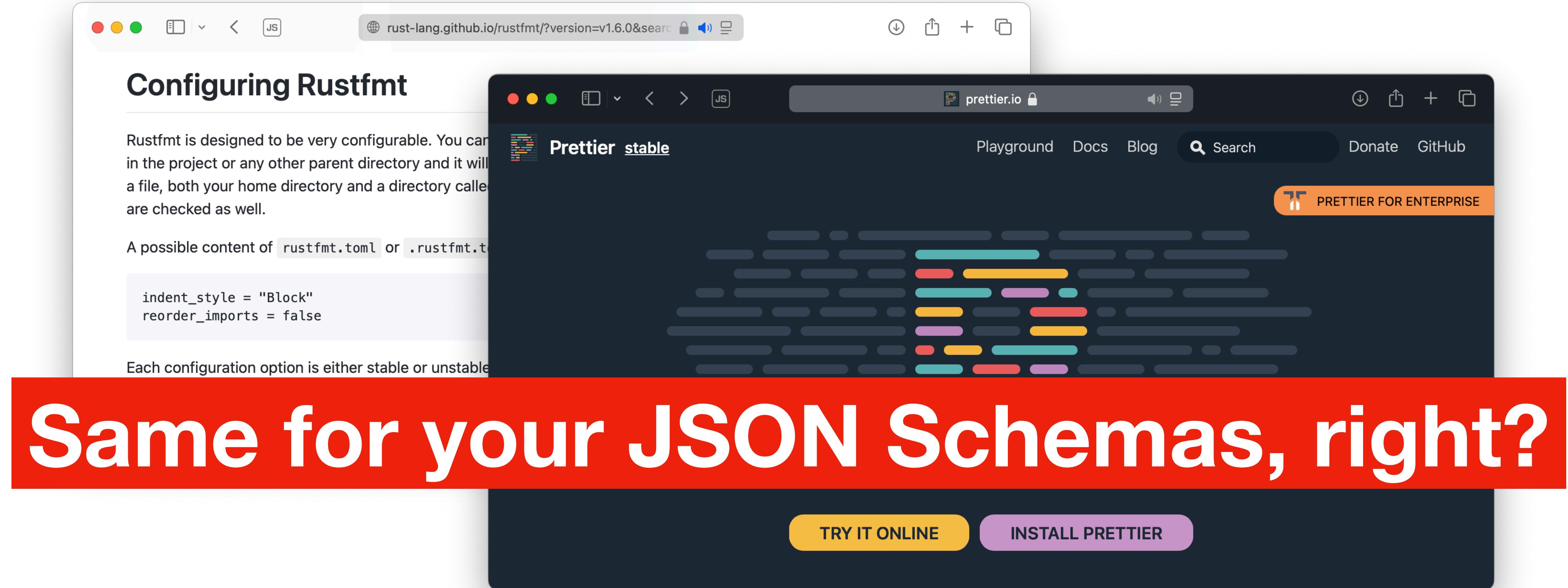
Prettier stable

Playground Docs Blog Search Donate GitHub PRETTIER FOR ENTERPRISE

TRY IT ONLINE INSTALL PRETTIER

Are you using any code formatters?

Like prettier, rustfmt, gofmt, etc



Configuring Rustfmt

Rustfmt is designed to be very configurable. You can run it from the command line or add configuration to your project or any other parent directory and it will read it. If you run it from your home directory, both your home directory and a directory called `.rustfmt.toml` in any parent directory are checked as well.

A possible content of `rustfmt.toml` or `.rustfmt.toml`:

```
indent_style = "Block"
reorder_imports = false
```

Each configuration option is either stable or unstable.

Prettier stable

Playground Docs Blog Search Donate GitHub PRETTIER FOR ENTERPRISE

TRY IT ONLINE INSTALL PRETTIER

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

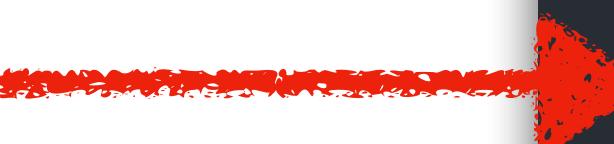
Format your schemas for readability and unified styling



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



Dialect first, so we
know how to read
the rest



```
$ 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)" }
```

Schema-wide
metadata at the top



```
$ 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



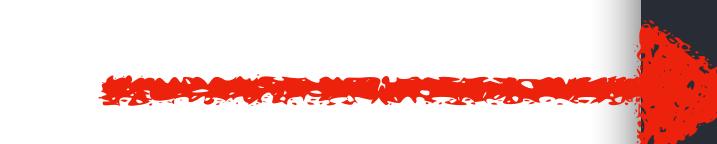
```
{ "$id": "https://example.com/iso8601/v1.json",
  "pattern": "^(?!0000)\\d{4}$",
  "type": "string",
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "title": "ISO 8601 four-digit year (YYYY)" }
```



```
$ 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}$"
}
```

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)" }
```



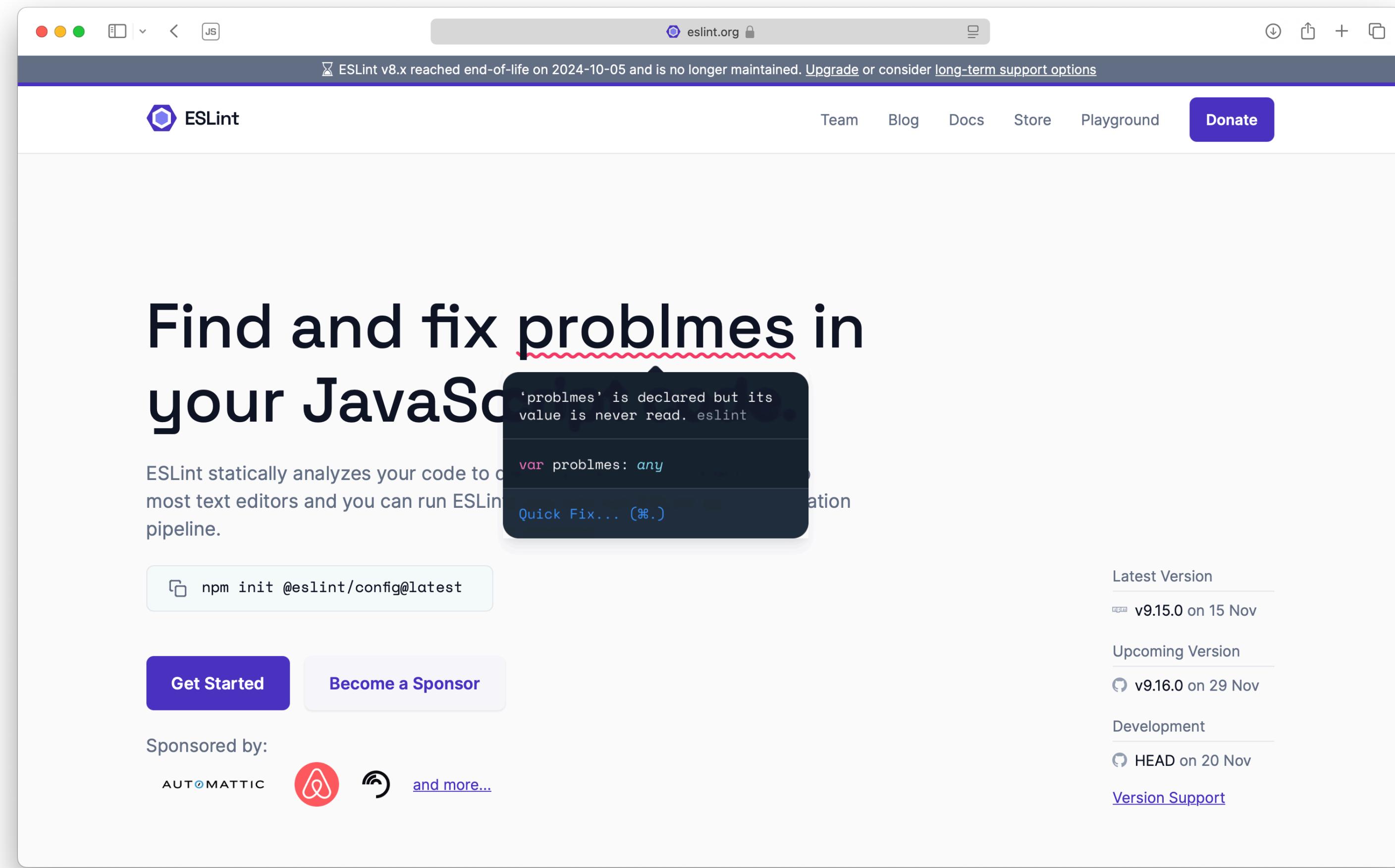
```
$ jsonschema fmt schema.json
$ cat schema.json
```

```
{
```

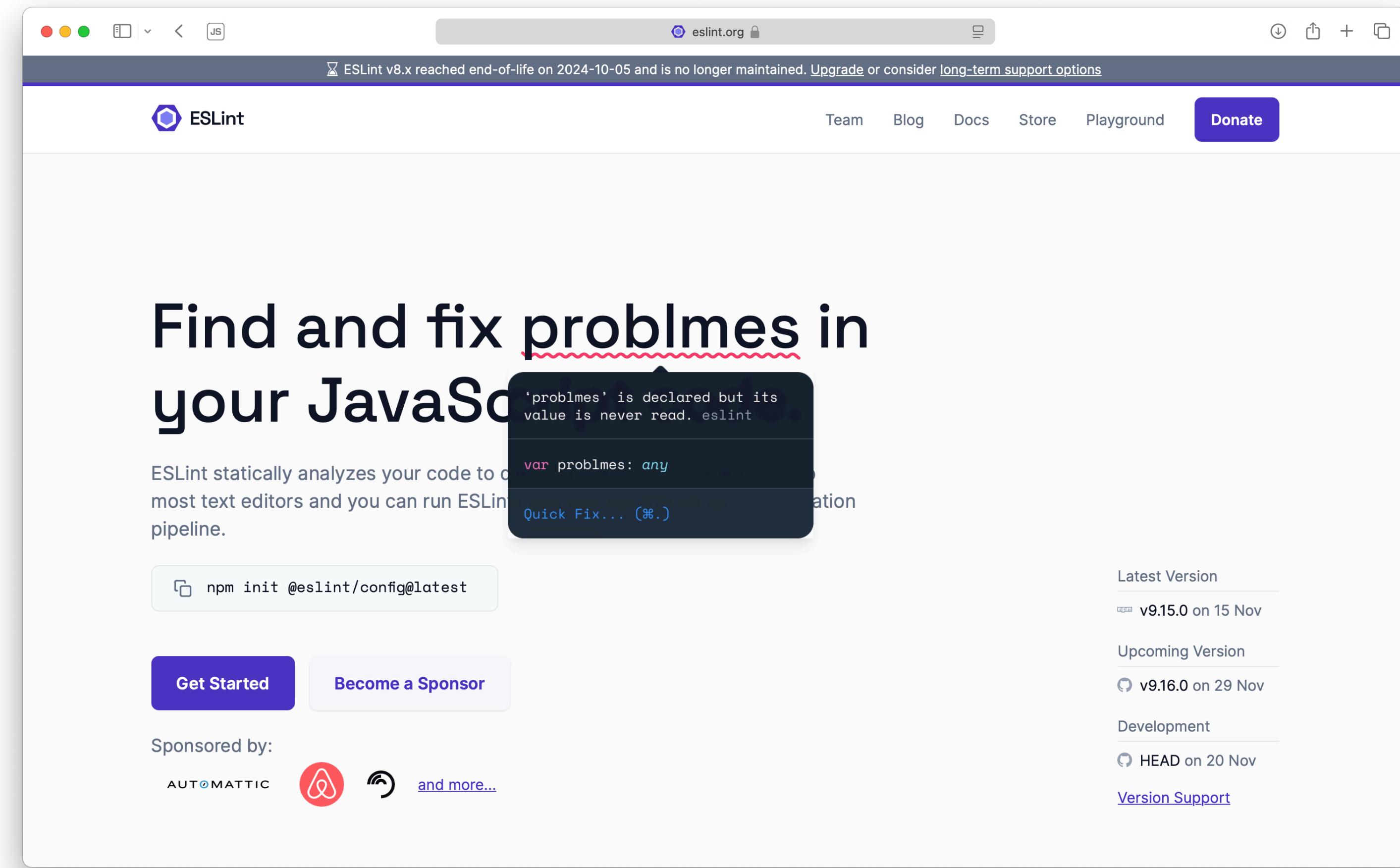
```
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://example.com/iso8601/v1.json",
  "title": "ISO 8601 four-digit year (YYYY)",
  "type": "string",
  "pattern": "^(?!0000)\\d{4}$"
```

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"  
    },  
    {  
      "description": "Invalid year (too early)",  
      "valid": false,  
      "data": "2000"  
    }  
  ]  
}
```



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}$"  
}
```



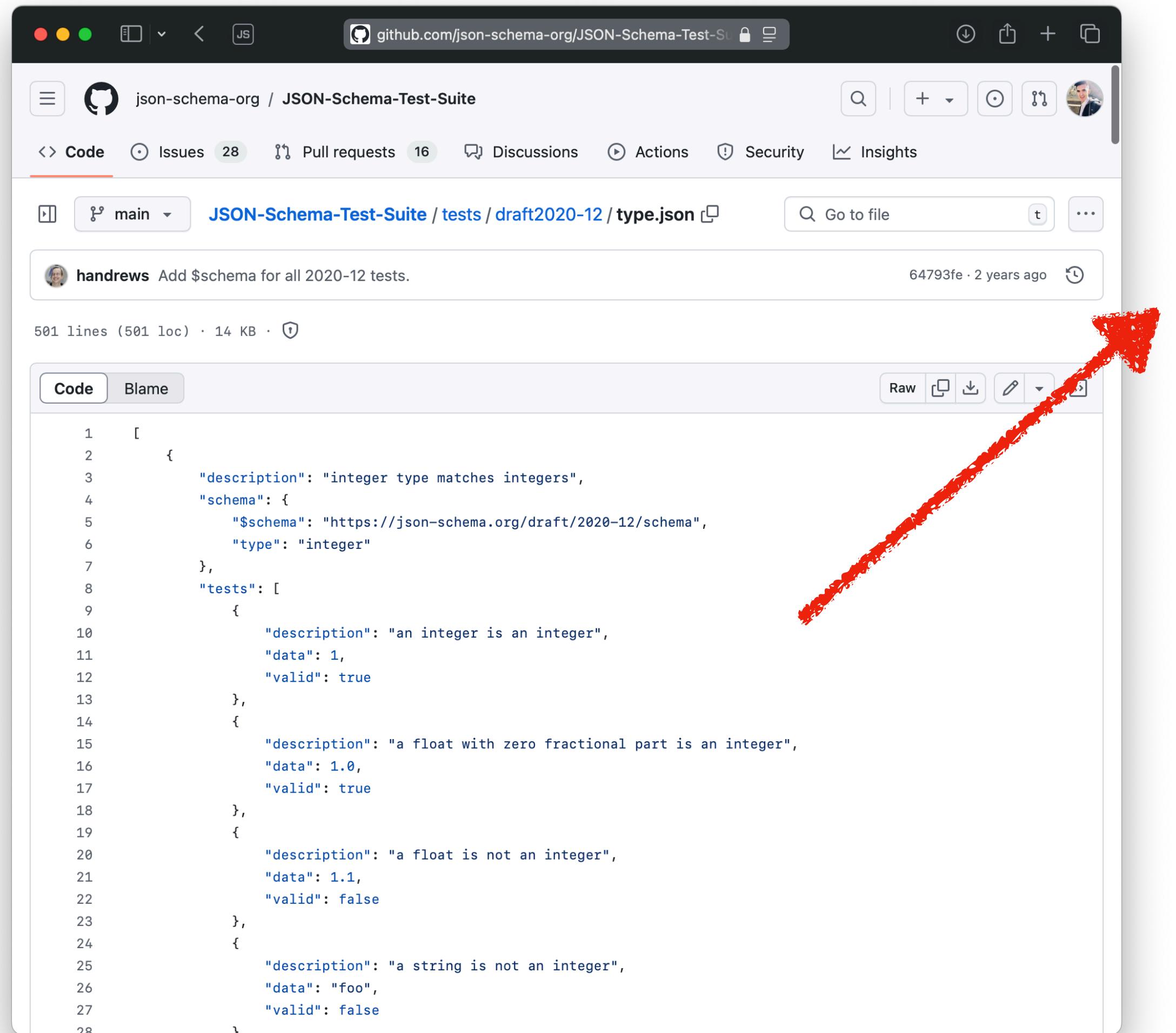
```
$ jsonschema test --verbose test.json --resolve schema.json  
Importing schema into the resolution context: schema.json  
test.json:  
  1/3 PASS Valid year  
  2/3 PASS Zero is not a valid year  
  3/3 PASS Non-string is invalid
```



```
{  
  "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

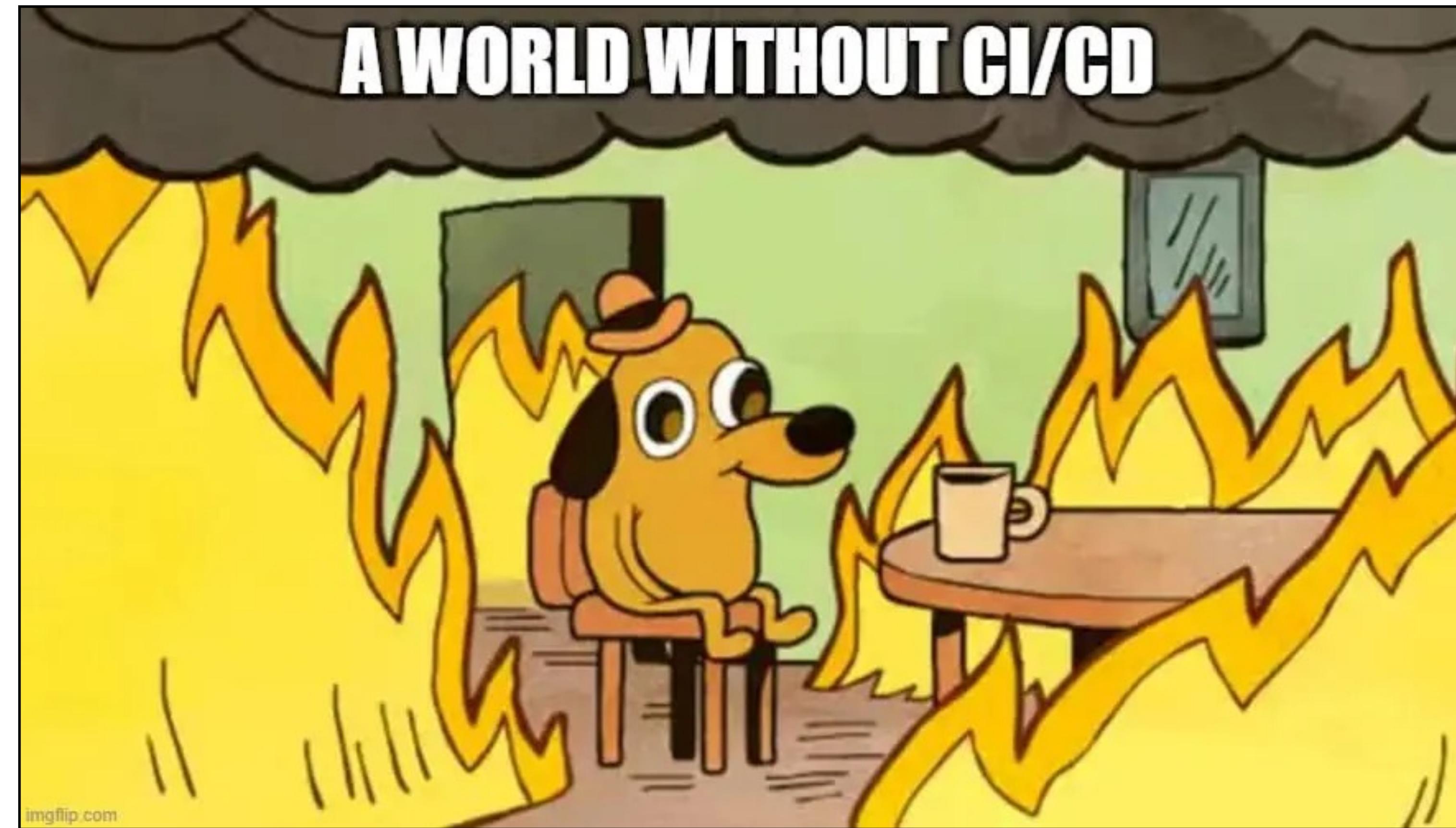
Unit test your schemas to ensure they match what you intend



A screenshot of a GitHub repository page for 'json-schema-org / JSON-Schema-Test-Suite'. The page shows a file named 'type.json' under the 'tests / draft2020-12' directory. The code in the file is a JSON schema test suite. A large red arrow originates from the text in the adjacent paragraph and points directly at the code in the screenshot.

```
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     }  
]
```

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/
```



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



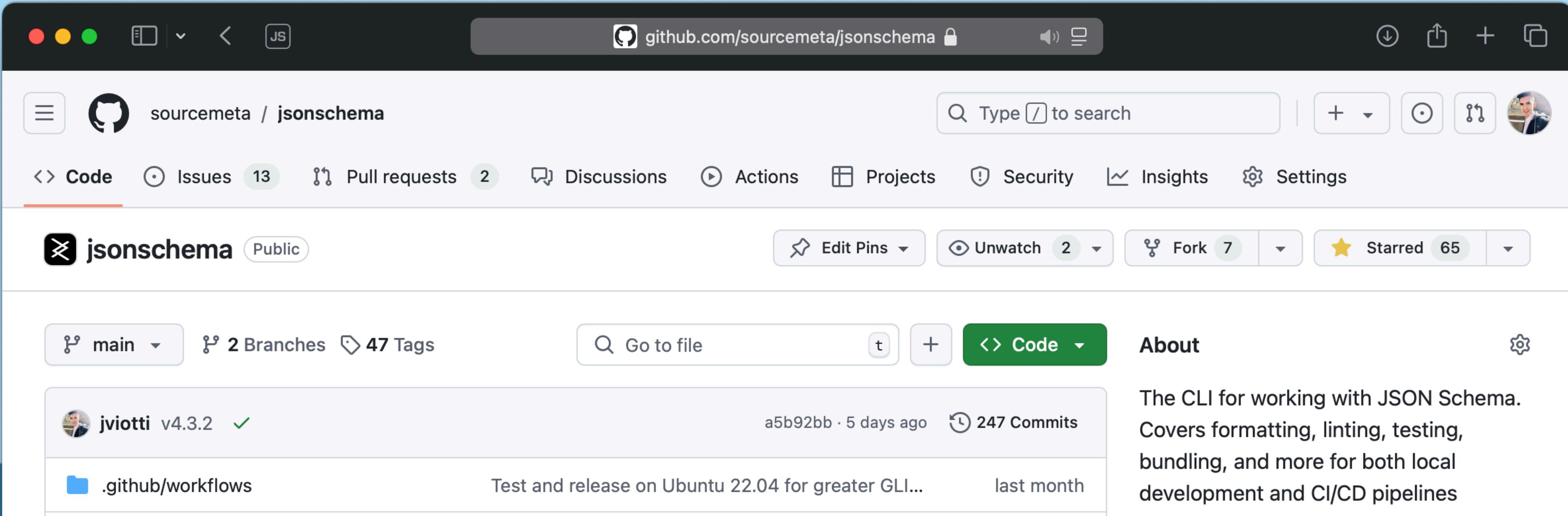
Make sure the
schemas actually
do what you intend

Thanks a lot!



JSON Schema CLI

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



A screenshot of a GitHub repository page for 'jsonschema' by 'sourcmeta'. The page features a dark mode interface. At the top, there's a navigation bar with links for Code, Issues (13), Pull requests (2), Discussions, Actions, Projects, Security, Insights, and Settings. Below the navigation is a header with the repository name 'jsonschema' and a 'Public' badge. To the right are buttons for Edit Pins, Unwatch (2), Fork (7), and Starred (65). Further down, there are buttons for main, 2 Branches, 47 Tags, Go to file, and Code. A commit history section shows a recent commit by 'jviotti' (v4.3.2) with 247 commits made 5 days ago. The 'About' section describes the repository as 'The CLI for working with JSON Schema. Covers formatting, linting, testing, bundling, and more for both local development and CI/CD pipelines.'