STScI | SPACE TELESCOPE SCIENCE INSTITUTE

EXPANDING THE FRONTIERS OF SPACE ASTRONOMY

OpenAPI Protocol Transition: Technical Overview

Joshua Fraustro

May 22nd, 2024

What is OpenAPI?

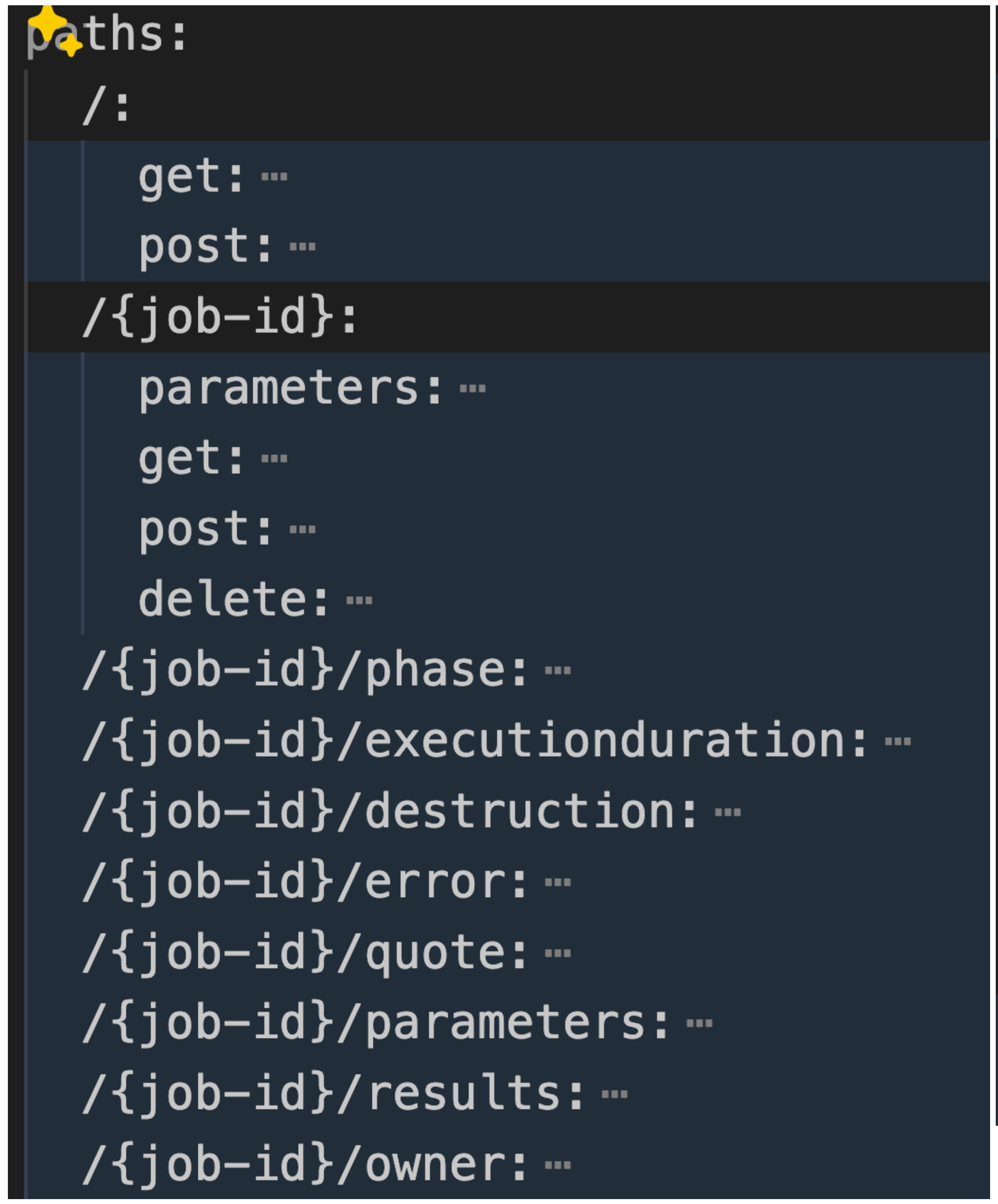
A specification for defining RESTful APIs / web services

- Standardized, language-agnostic interface
- Simple to write, simple to read
  - We’re preferring to use YAML over JSON
- Defines API endpoints, request/response formats, payloads and schemas
- Machine-readable
- Widely used, modern, industry standard
  - Rich ecosystem of tools— editors, validators, generators
  - P3T has been experimenting with OpenAPI 3.0.x, may adopt 3.10

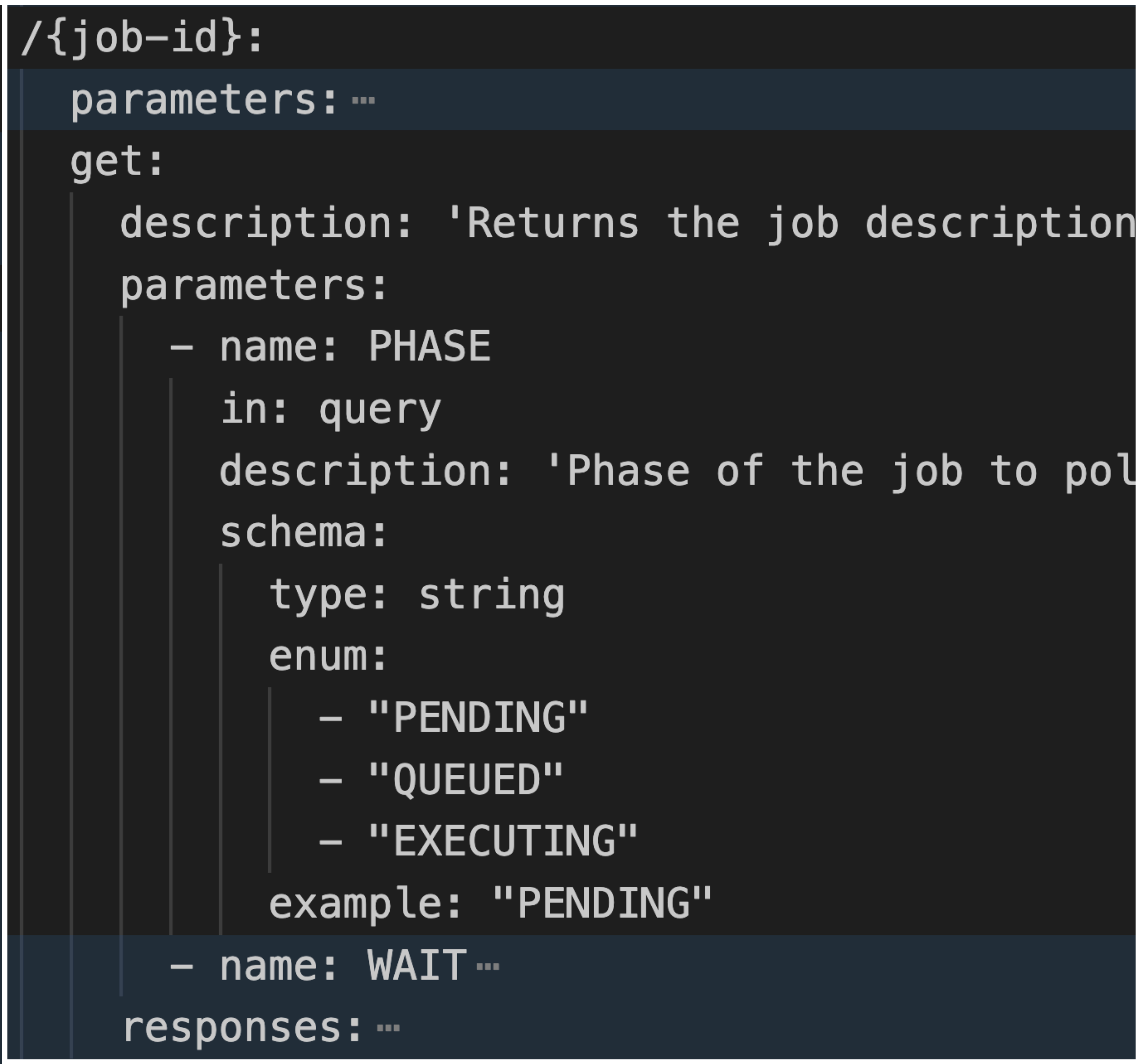
What does an example UWS OpenAPI spec look like?

https://github.com/spacetelescope/vo-openapi

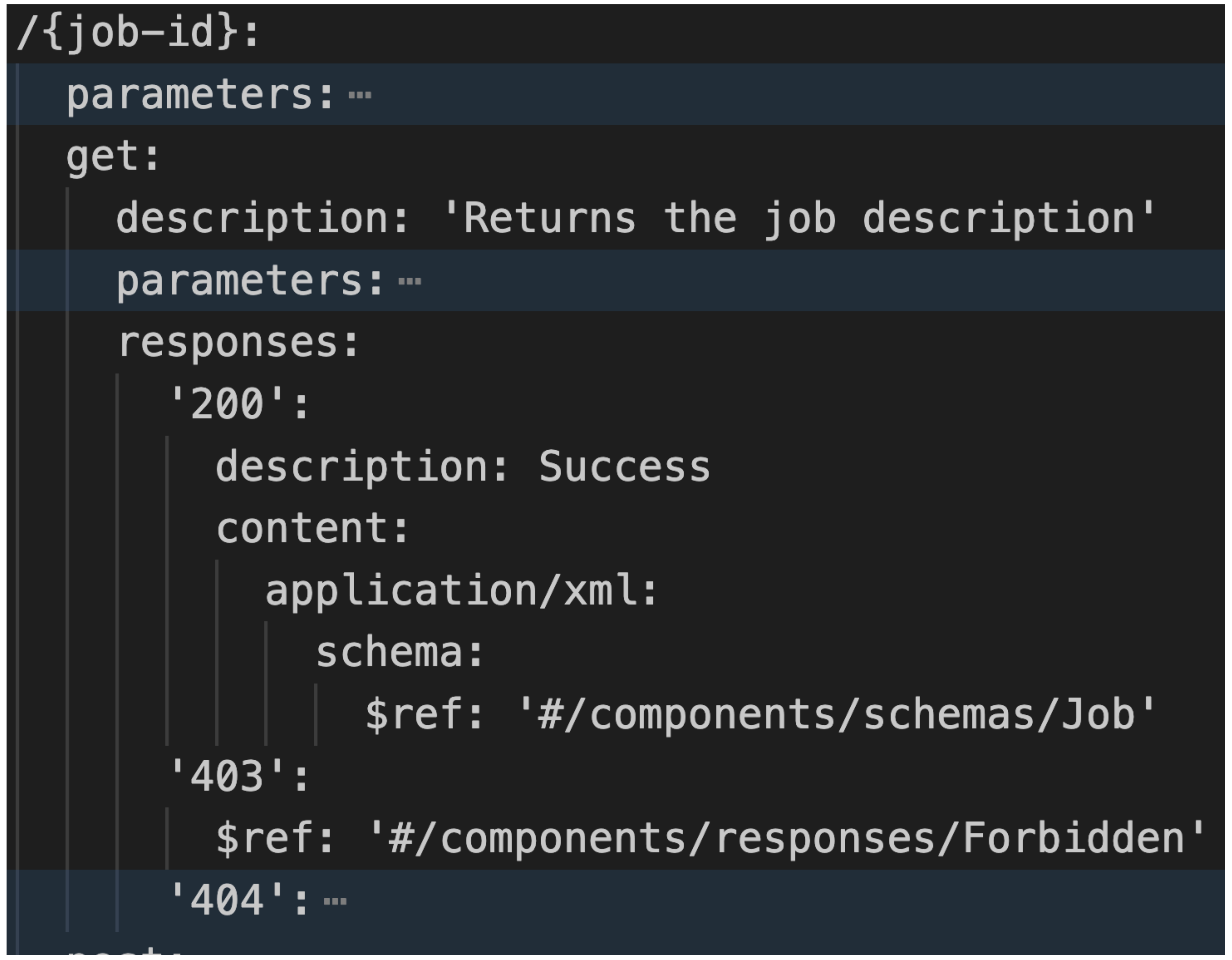
- Paths & Operations
- Parameters
- Responses



```
paths:
  /:
    get: ⋯
    post: ⋯
  /{job-id}:
    parameters: ⋯
    get: ⋯
    post: ⋯
    delete: ⋯
  /{job-id}/phase: ⋯
  /{job-id}/executionduration: ⋯
  /{job-id}/destruction: ⋯
  /{job-id}/error: ⋯
  /{job-id}/quote: ⋯
  /{job-id}/parameters: ⋯
  /{job-id}/results: ⋯
  /{job-id}/owner: ⋯
```



```
/{job-id}:
  parameters: ⋯
  get:
    description: 'Returns the job description
    parameters:
      - name: PHASE
        in: query
        description: 'Phase of the job to pol
        schema:
          type: string
          enum:
            - "PENDING"
            - "QUEUED"
            - "EXECUTING"
          example: "PENDING"
      - name: WAIT ⋯
    responses: ⋯
```

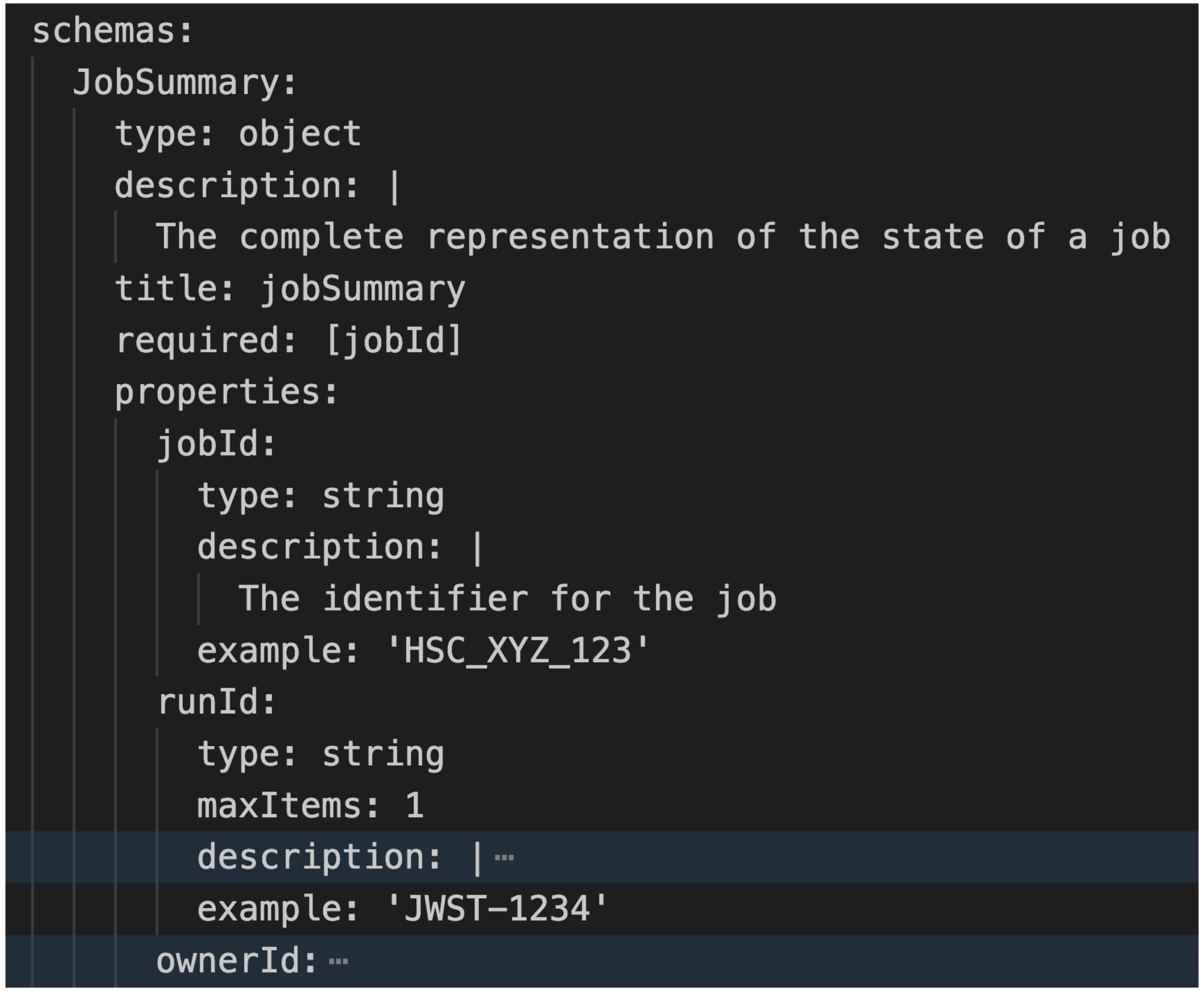


```
/{job-id}:
  parameters: ⋯
  get:
    description: 'Returns the job description'
    parameters: ⋯
    responses:
      '200':
        description: Success
        content:
          application/xml:
            schema:
              $ref: '#/components/schemas/Job'
      '403':
        $ref: '#/components/responses/Forbidden'
      '404': ⋯
```

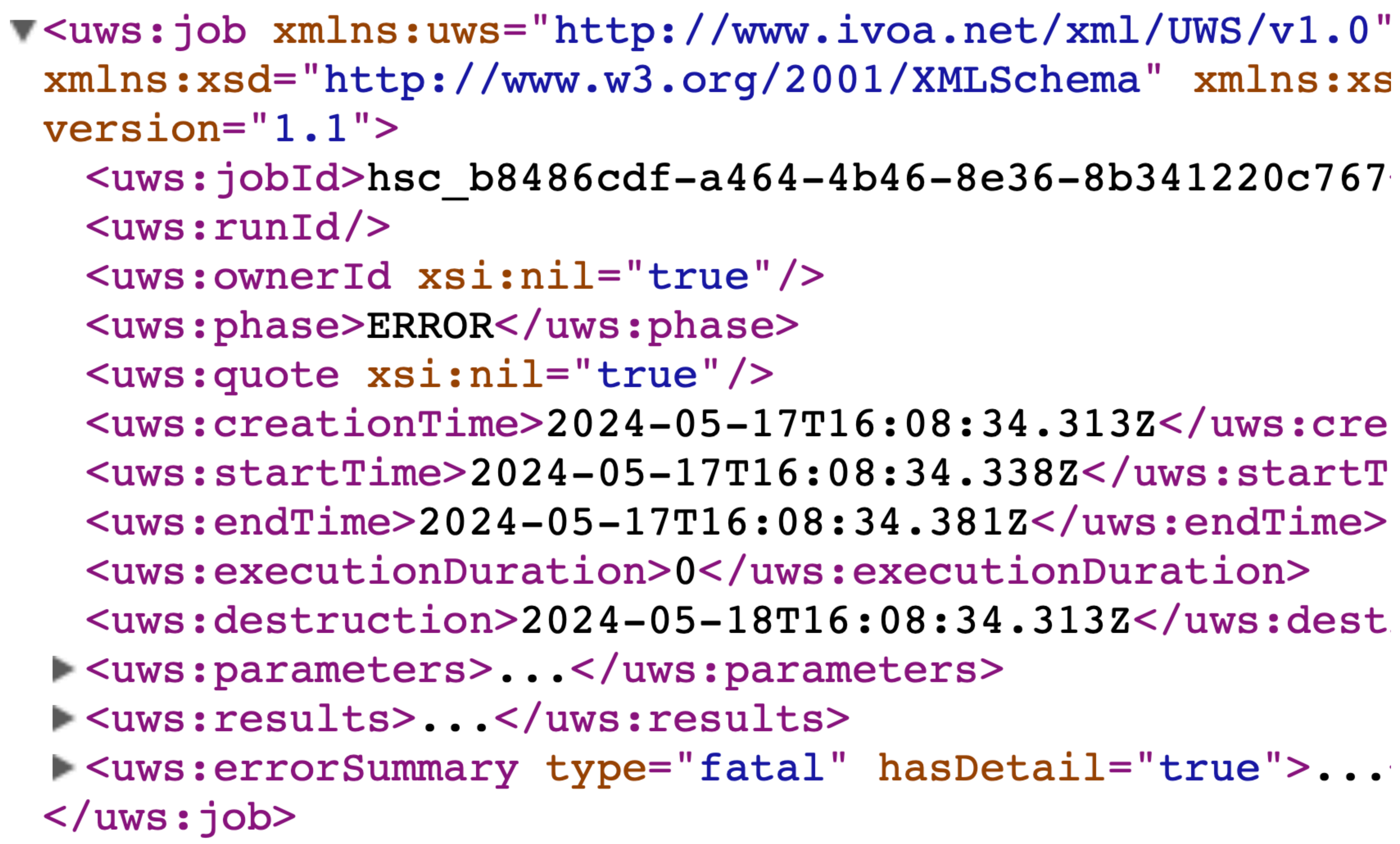
What does an example UWS OpenAPI spec look like?

https://github.com/spacetelescope/vo-openapi

- Schema Objects (Payloads & Responses)



```
schemas:
  JobSummary:
    type: object
    description: |
      The complete representation of the state of a job
    title: jobSummary
    required: [jobId]
    properties:
      jobId:
        type: string
        description: |
          The identifier for the job
        example: 'HSC_XYZ_123'
      runId:
        type: string
        maxItems: 1
        description: |⋯
        example: 'JWST-1234'
      ownerId:⋯
```



```
▼<uws:job xmlns:uws="http://www.ivoa.net/xml/UWS/v1.0"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xs
 version="1.1">
   <uws:jobId>hsc_b8486cdf-a464-4b46-8e36-8b341220c767
   <uws:runId/>
   <uws:ownerId xsi:nil="true"/>
   <uws:phase>ERROR</uws:phase>
   <uws:quote xsi:nil="true"/>
   <uws:creationTime>2024-05-17T16:08:34.313Z</uws:cre
   <uws:startTime>2024-05-17T16:08:34.338Z</uws:startT
   <uws:endTime>2024-05-17T16:08:34.381Z</uws:endTime>
   <uws:executionDuration>0</uws:executionDuration>
   <uws:destruction>2024-05-18T16:08:34.313Z</uws:dest
  ▶<uws:parameters>...</uws:parameters>
  ▶<uws:results>...</uws:results>
  ▶<uws:errorSummary type="fatal" hasDetail="true">...
 </uws:job>
```

What are we proposing?

New (and updated) standards be described by an OpenAPI specification.

Standards be broken into two documents:

- Narrative (non-normative): Motivations, background, use cases
- Technical (normative):
    - Behavior of the service (blocking vs. non-blocking endpoints)
    - Explanations of parameters / payloads (beyond type descriptions)
    - + the OpenAPI specification

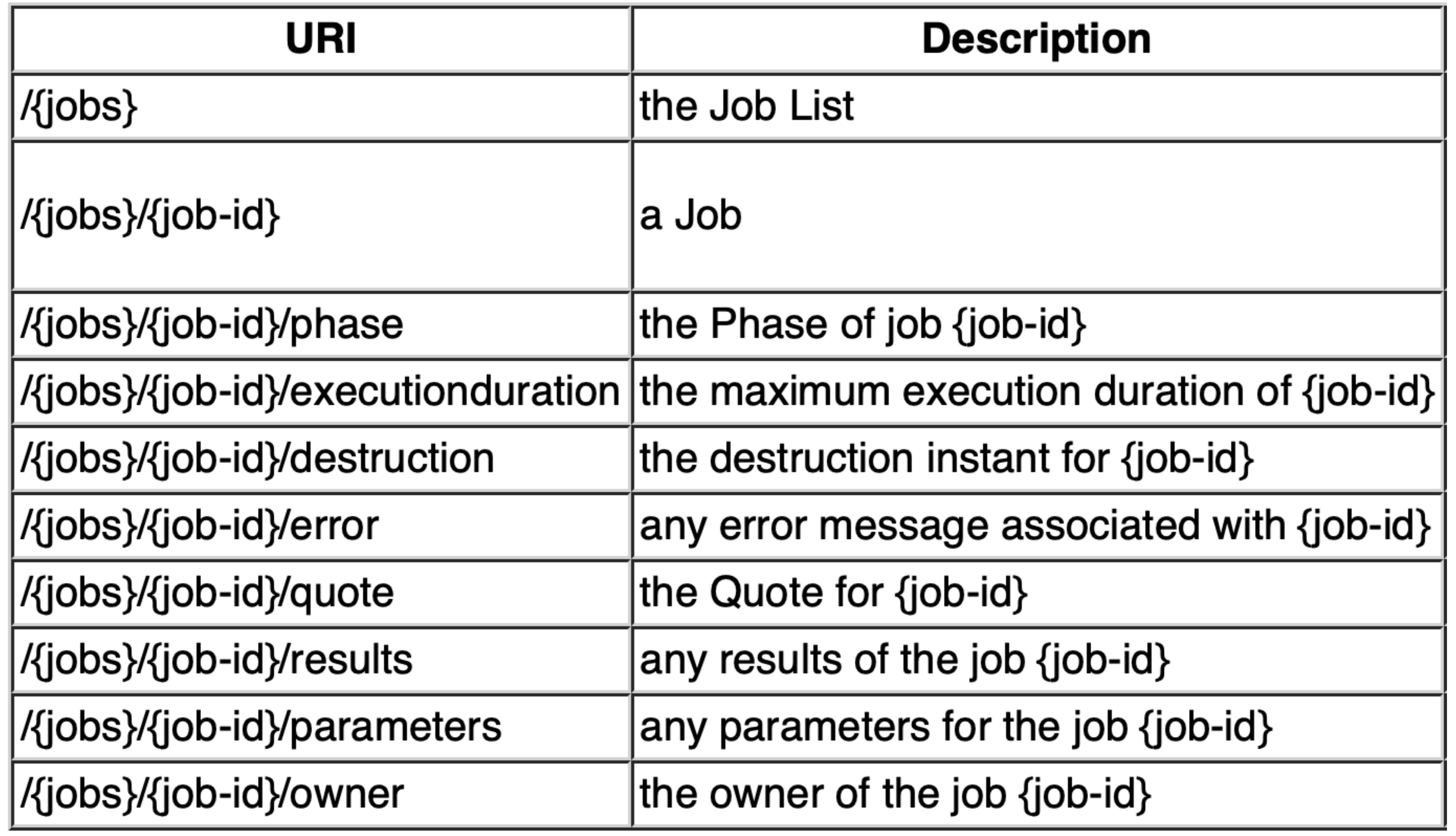
Standards process will remain much the same:

- Reference implementations, validators, etc.

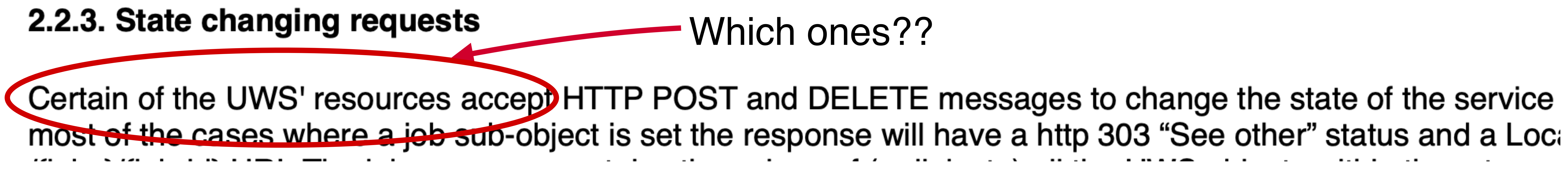
What are the implications?

Shorter, simpler standards documents:

- The OpenAPI spec makes a lot self-evident
- Endpoints and parameters can be self-documenting
- Explanations in the tech doc will be reserved for underlying service behavior
- Removal of ambiguity in request & response behavior
  - What operations on which endpoints, with which parameters, and what responses?
  - Now, clear and explicit.



URI	Description
/{jobs}	the Job List
/{jobs}/{job-id}	a Job
/{jobs}/{job-id}/phase	the Phase of job {job-id}
/{jobs}/{job-id}/executionduration	the maximum execution duration of {job-id}
/{jobs}/{job-id}/destruction	the destruction instant for {job-id}
/{jobs}/{job-id}/error	any error message associated with {job-id}
/{jobs}/{job-id}/quote	the Quote for {job-id}
/{jobs}/{job-id}/results	any results of the job {job-id}
/{jobs}/{job-id}/parameters	any parameters for the job {job-id}
/{jobs}/{job-id}/owner	the owner of the job {job-id}



2.2.3. State changing requests

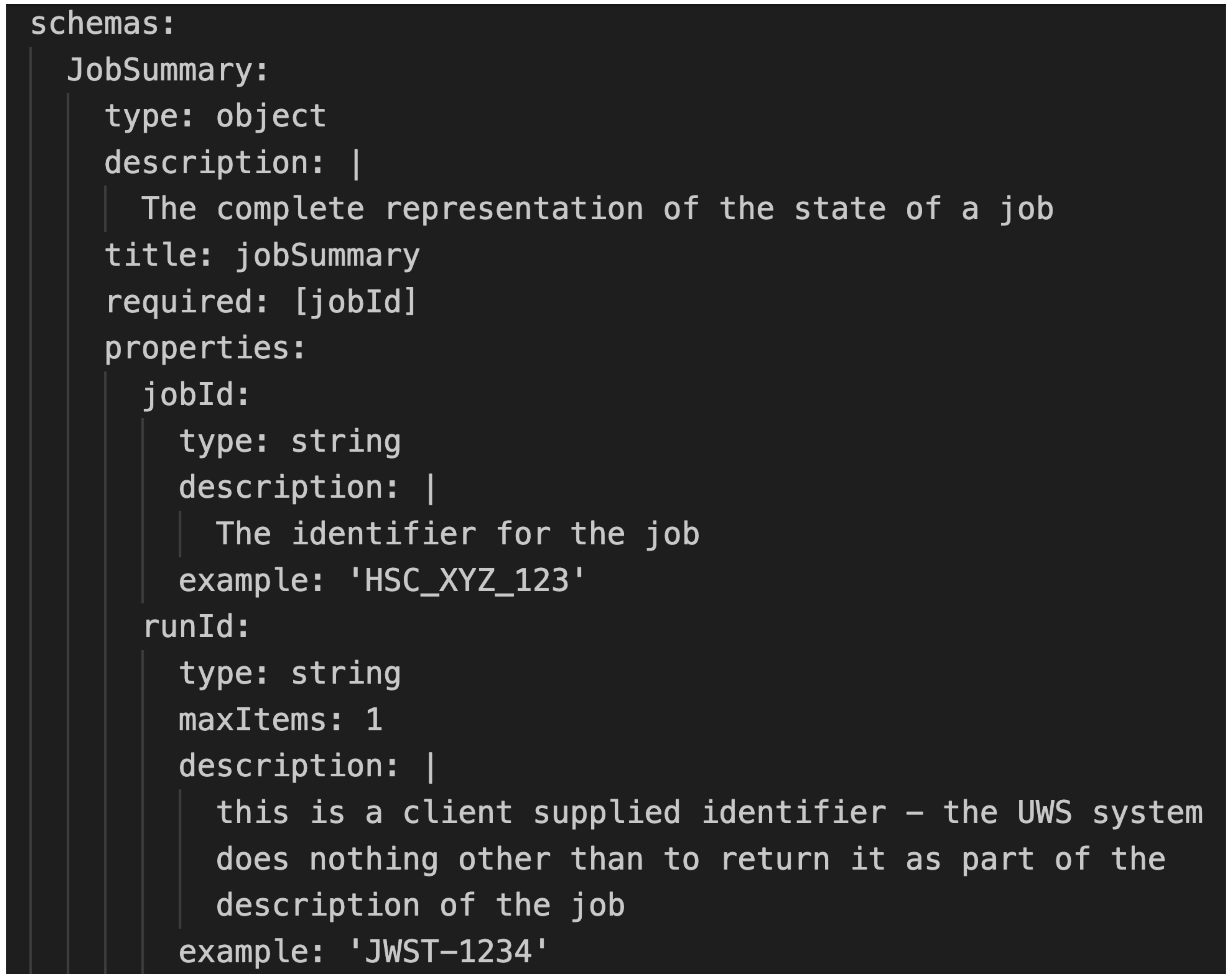
Certain of the UWS' resources accept HTTP POST and DELETE messages to change the state of the service most of the cases where a job sub-object is set the response will have a http 303 “See other” status and a Loc

Which ones??

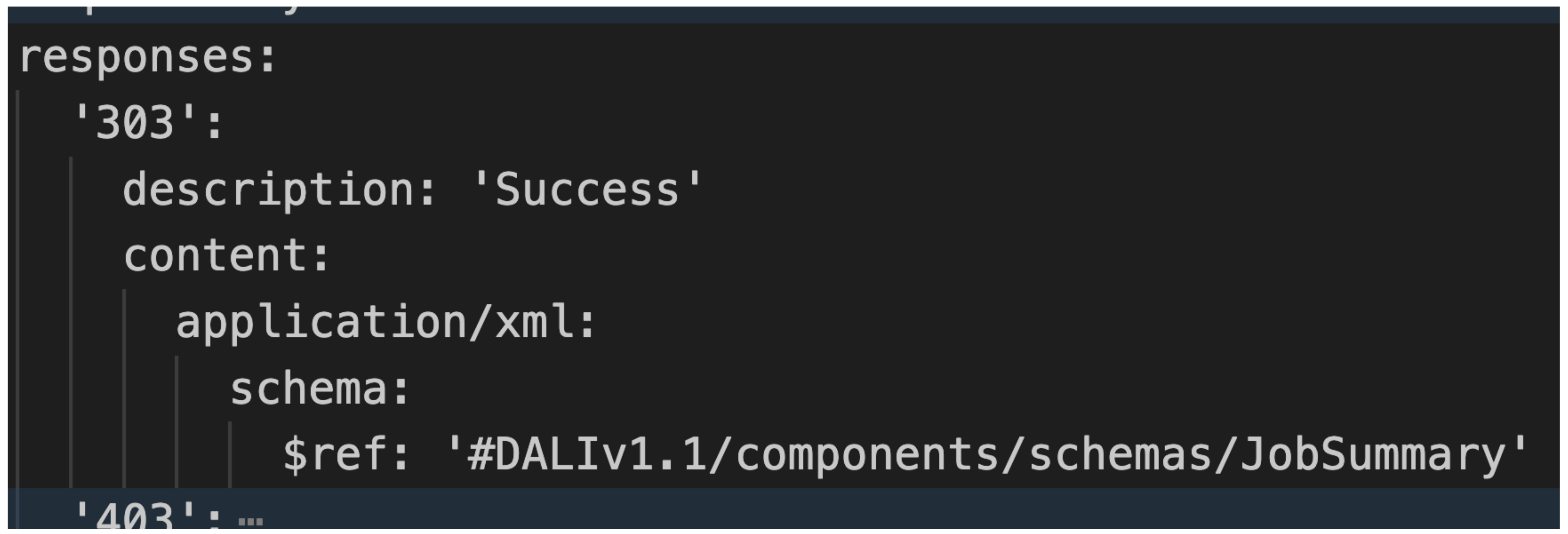
What are the implications?

Simpler and re-useable component definitions

- Error responses can be defined once, and used across the standard
- “Schemas” (payloads, responses) are language-agnostic
  - Defined as objects, not tied to a specific encoding method
  - Can be defined and versioned in one standard (DALI) and imported to others



```yaml
schemas:
  JobSummary:
    type: object
    description: |
      The complete representation of the state of a job
    title: jobSummary
    required: [jobId]
    properties:
      jobId:
        type: string
        description: |
          The identifier for the job
        example: 'HSC_XYZ_123'
      runId:
        type: string
        maxItems: 1
        description: |
          this is a client supplied identifier - the UWS system
          does nothing other than to return it as part of the
          description of the job
        example: 'JWST-1234'
```



```yaml
responses:
  '303':
    description: 'Success'
    content:
      application/xml:
        schema:
          $ref: '#DALIv1.1/components/schemas/JobSummary'
  '403': …
```

What are the implications?

Parts of the standards will need to change, complying with modern web development principles.

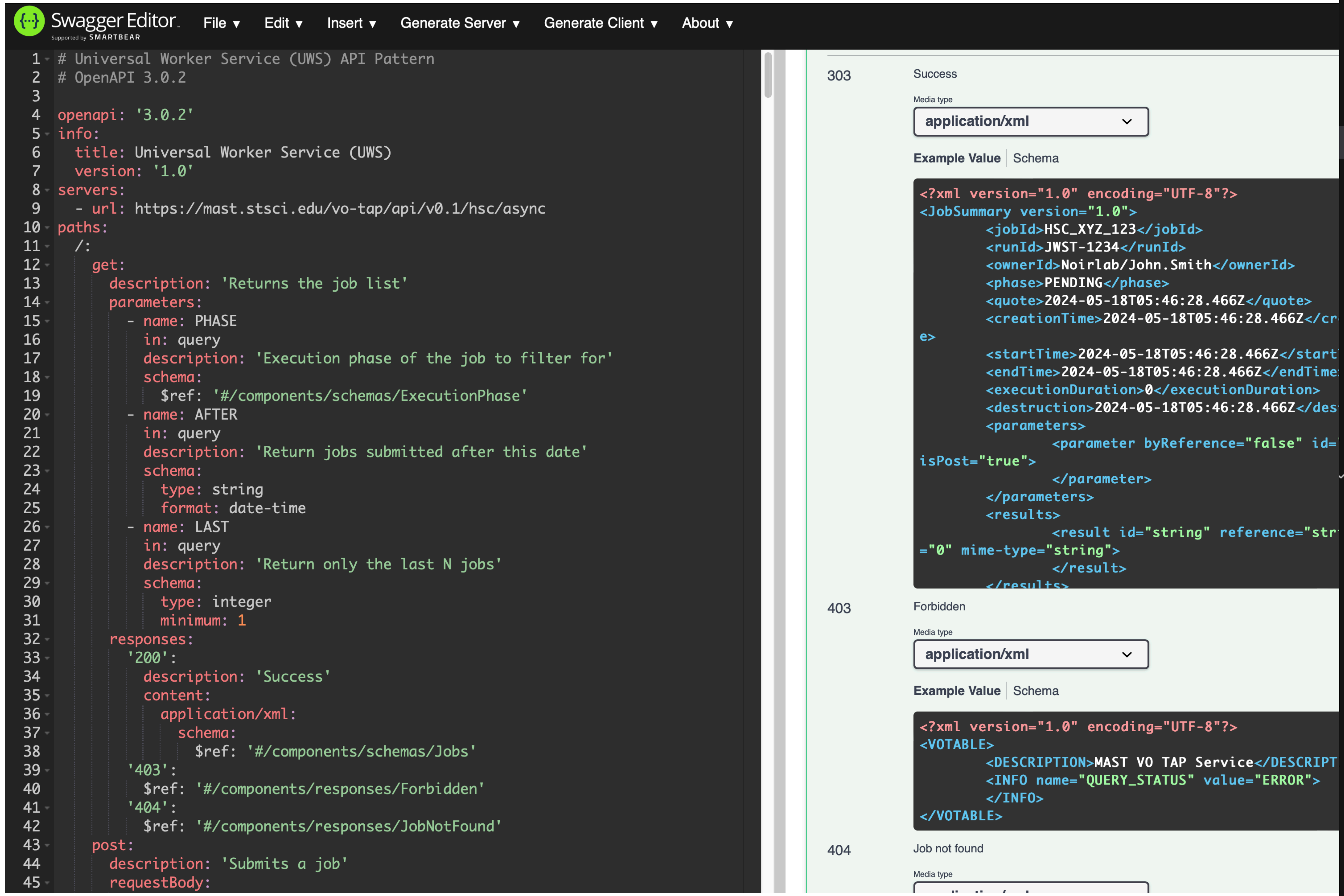
- Avoiding anti-patterns:
  - If something isn't easy to do with OpenAPI, we probably shouldn't do it
    - Case-insensitive DALI parameters— not typical in HTTP behavior, almost impossible in OpenAPI specifications
- Great opportunity for removing bad behavior:
  - Simple POSTs with x-www-form-urlencoded— no preflight checks— vulnerable to CSRF
  - State-changing GET requests — also vulnerable to CSRF attacks
- Move towards supporting modern protocol serialization formats
  - XML support can be spotty depending on language / framework.
  - By defining payloads as objects, we're not strictly bound to it any more.
  - We're not touching the VOTable!!

What are the benefits? Let me count the ways!

Immediate benefits:

- Clearer, simpler standards
- Modularity of service definitions
  - "See DALI" is now an actual import from a versioned DALI spec
- Much lower developer spin-up time
  - Far faster for a new-hire web developer to program against an API definition, than a 22-page academic text
  - Don't need a deep understanding of DALI / UWS / TAP / VOSI to understand how the basics work
- Clear, obvious definitions for every parameter, payload, etc.
- Flexibility for future protocol serialization methods.

What are the benefits? - Interactive Swagger Editor / IDEs



```yaml
# Universal Worker Service (UWS) API Pattern
# OpenAPI 3.0.2

openapi: '3.0.2'
info:
  title: Universal Worker Service (UWS)
  version: '1.0'
servers:
  - url: https://mast.stsci.edu/vo-tap/api/v0.1/hsc/async
paths:
  /:
    get:
      description: 'Returns the job list'
      parameters:
        - name: PHASE
          in: query
          description: 'Execution phase of the job to filter for'
          schema:
            $ref: '#/components/schemas/ExecutionPhase'
        - name: AFTER
          in: query
          description: 'Return jobs submitted after this date'
          schema:
            type: string
            format: date-time
        - name: LAST
          in: query
          description: 'Return only the last N jobs'
          schema:
            type: integer
            minimum: 1
      responses:
        '200':
          description: 'Success'
          content:
            application/xml:
              schema:
                $ref: '#/components/schemas/Jobs'
        '403':
          $ref: '#/components/responses/Forbidden'
        '404':
          $ref: '#/components/responses/JobNotFound'
    post:
      description: 'Submits a job'
      requestBody:
```

303 Success

Media type: application/xml

Example Value | Schema

```xml
<?xml version="1.0" encoding="UTF-8"?>
<JobSummary version="1.0">
        <jobId>HSC_XYZ_123</jobId>
        <runId>JWST-1234</runId>
        <ownerId>Noirlab/John.Smith</ownerId>
        <phase>PENDING</phase>
        <quote>2024-05-18T05:46:28.466Z</quote>
        <creationTime>2024-05-18T05:46:28.466Z</cr
e>
        <startTime>2024-05-18T05:46:28.466Z</start
        <endTime>2024-05-18T05:46:28.466Z</endTime
        <executionDuration>0</executionDuration>
        <destruction>2024-05-18T05:46:28.466Z</des
        <parameters>
                <parameter byReference="false" id=
isPost="true">
                </parameter>
        </parameters>
        <results>
                <result id="string" reference="str
="0" mime-type="string">
                </result>
        </results>
```

403 Forbidden

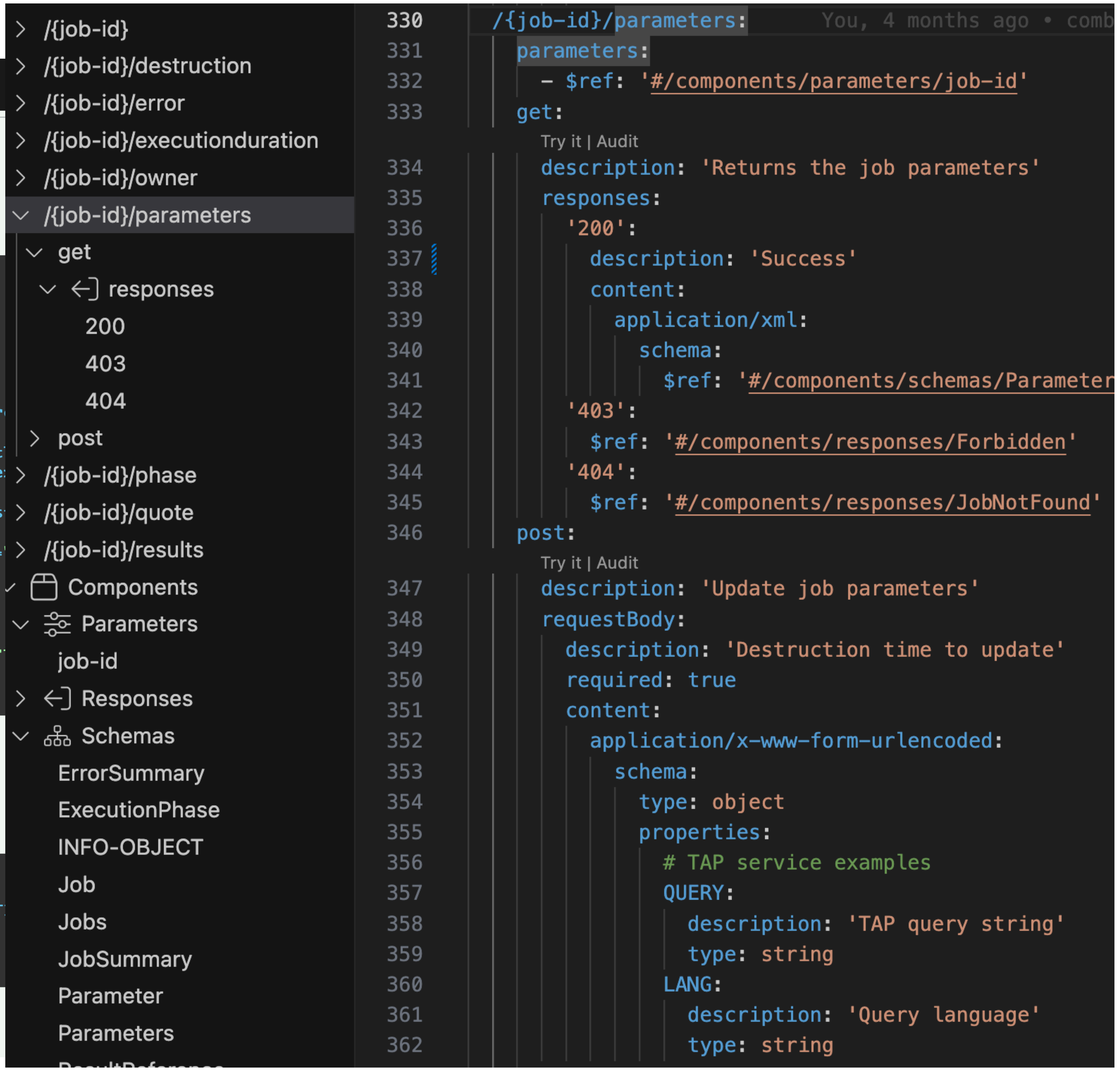
Media type: application/xml

Example Value | Schema

```xml
<?xml version="1.0" encoding="UTF-8"?>
<VOTABLE>
        <DESCRIPTION>MAST VO TAP Service</DESCRIPT
        <INFO name="QUERY_STATUS" value="ERROR">
        </INFO>
</VOTABLE>
```

404 Job not found

Media type

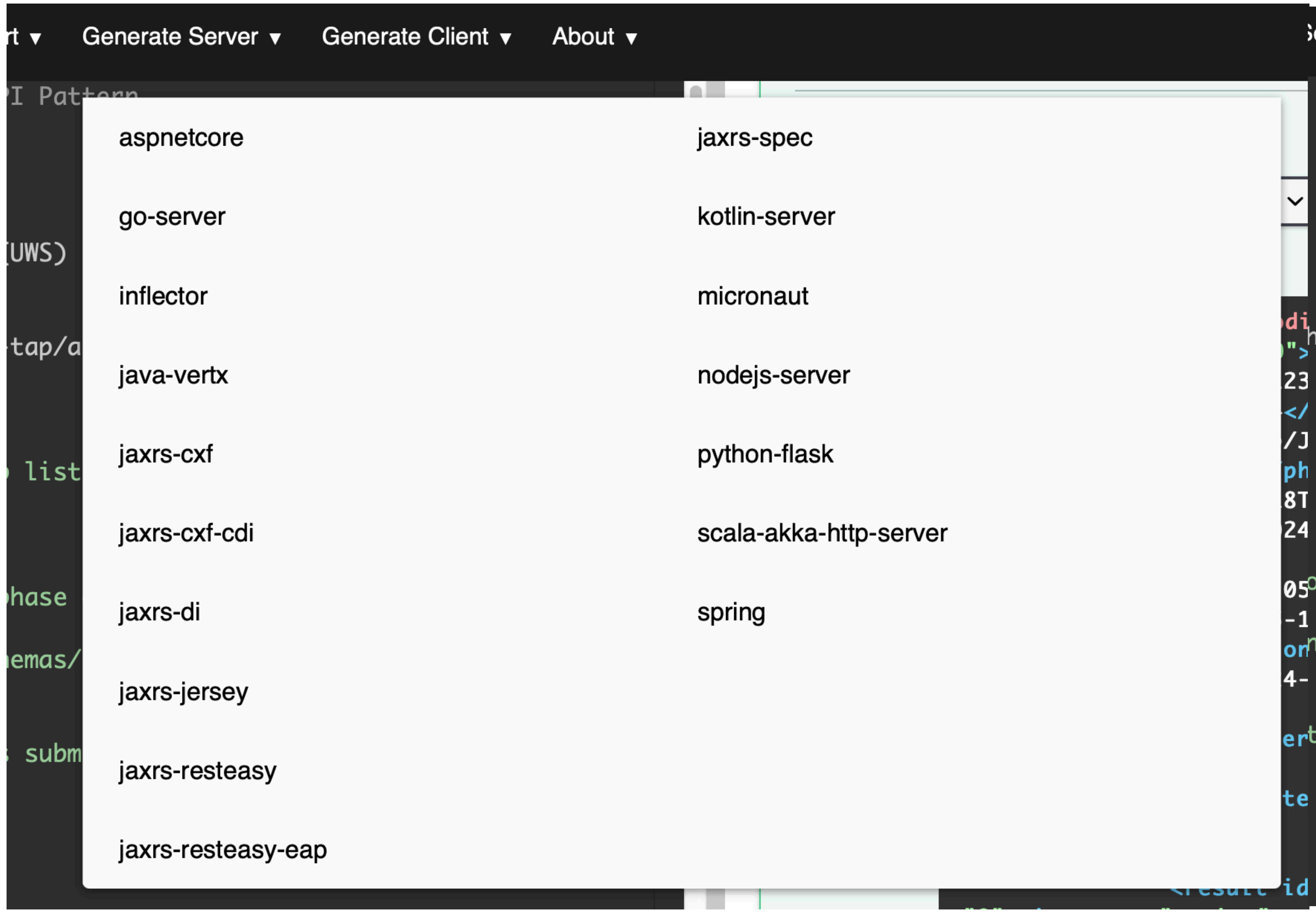


- /{job-id}
- /{job-id}/destruction
- /{job-id}/error
- /{job-id}/executionduration
- /{job-id}/owner
- /{job-id}/parameters
  - get
    - responses
      - 200
      - 403
      - 404
  - post
- /{job-id}/phase
- /{job-id}/quote
- /{job-id}/results
- Components
  - Parameters
    - job-id
  - Responses
  - Schemas
    - ErrorSummary
    - ExecutionPhase
    - INFO-OBJECT
    - Job
    - Jobs
    - JobSummary
    - Parameter
    - Parameters

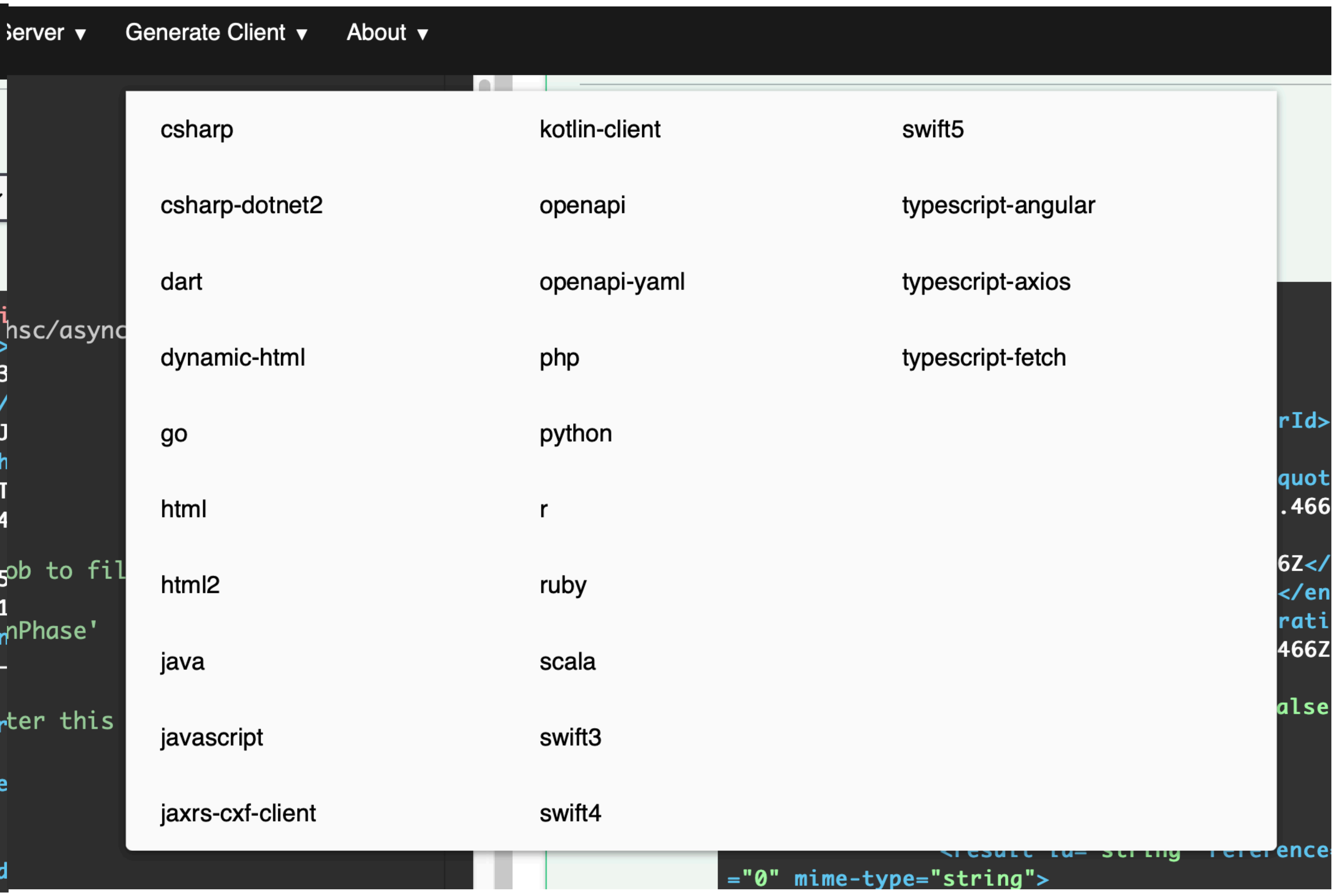
```yaml
  /{job-id}/parameters:          You, 4 months ago • comb
    parameters:
      - $ref: '#/components/parameters/job-id'
    get:
      Try it | Audit
      description: 'Returns the job parameters'
      responses:
        '200':
          description: 'Success'
          content:
            application/xml:
              schema:
                $ref: '#/components/schemas/Parameter
        '403':
          $ref: '#/components/responses/Forbidden'
        '404':
          $ref: '#/components/responses/JobNotFound'
    post:
      Try it | Audit
      description: 'Update job parameters'
      requestBody:
        description: 'Destruction time to update'
        required: true
        content:
          application/x-www-form-urlencoded:
            schema:
              type: object
              properties:
                # TAP service examples
                QUERY:
                  description: 'TAP query string'
                  type: string
                LANG:
                  description: 'Query language'
                  type: string
```

What are the benefits?

Server Generators

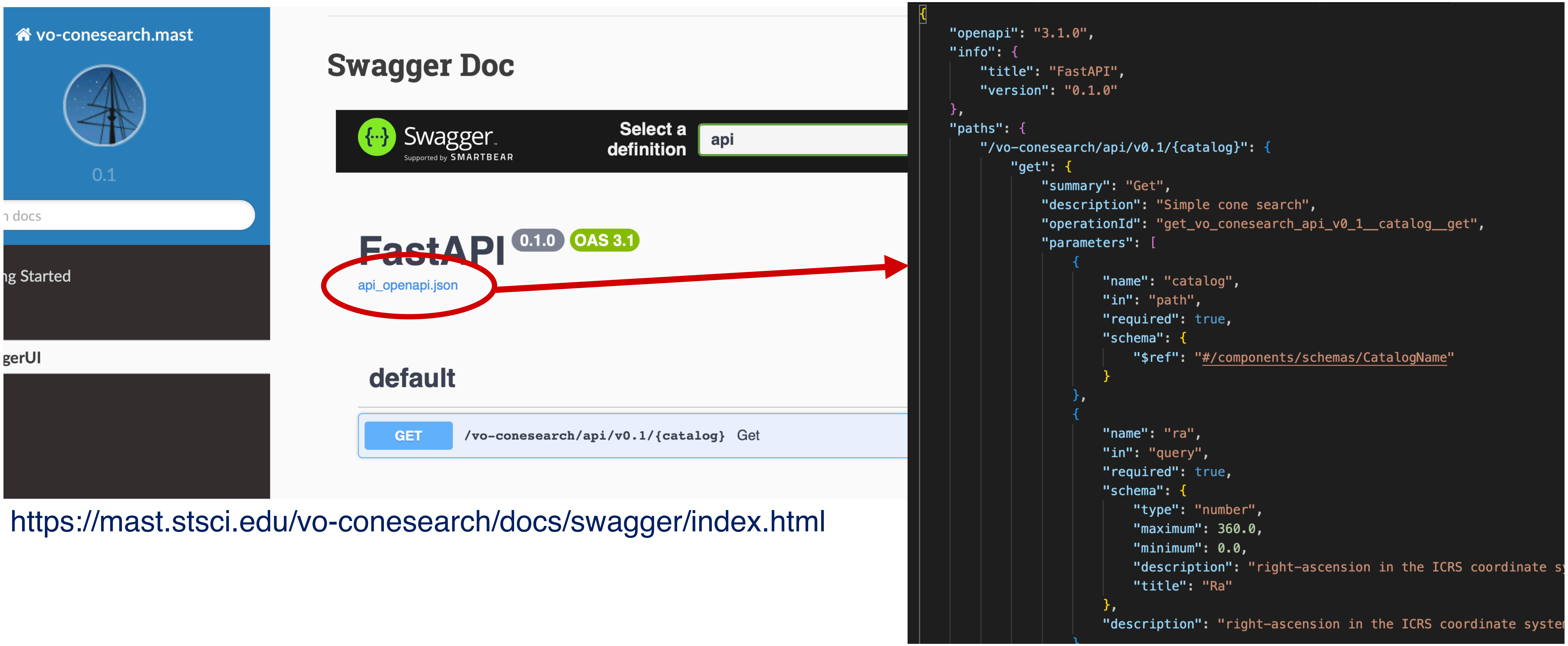


Client Generators



What are the benefits?

Your service might already have an OpenAPI spec…



/vo-conesearch/api/v0.1/{catalog} Get

```
{
    "openapi": "3.1.0",
    "info": {
        "title": "FastAPI",
        "version": "0.1.0"
    },
    "paths": {
        "/vo-conesearch/api/v0.1/{catalog}": {
            "get": {
                "summary": "Get",
                "description": "Simple cone search",
                "operationId": "get_vo_conesearch_api_v0_1__catalog__get",
                "parameters": [
                    {
                        "name": "catalog",
                        "in": "path",
                        "required": true,
                        "schema": {
                            "$ref": "#/components/schemas/CatalogName"
                        }
                    },
                    {
                        "name": "ra",
                        "in": "query",
                        "required": true,
                        "schema": {
                            "type": "number",
                            "maximum": 360.0,
                            "minimum": 0.0,
                            "description": "right-ascension in the ICRS coordinate sy
                            "title": "Ra"
                        },
                        "description": "right-ascension in the ICRS coordinate syste
```

https://mast.stsci.edu/vo-conesearch/docs/swagger/index.html

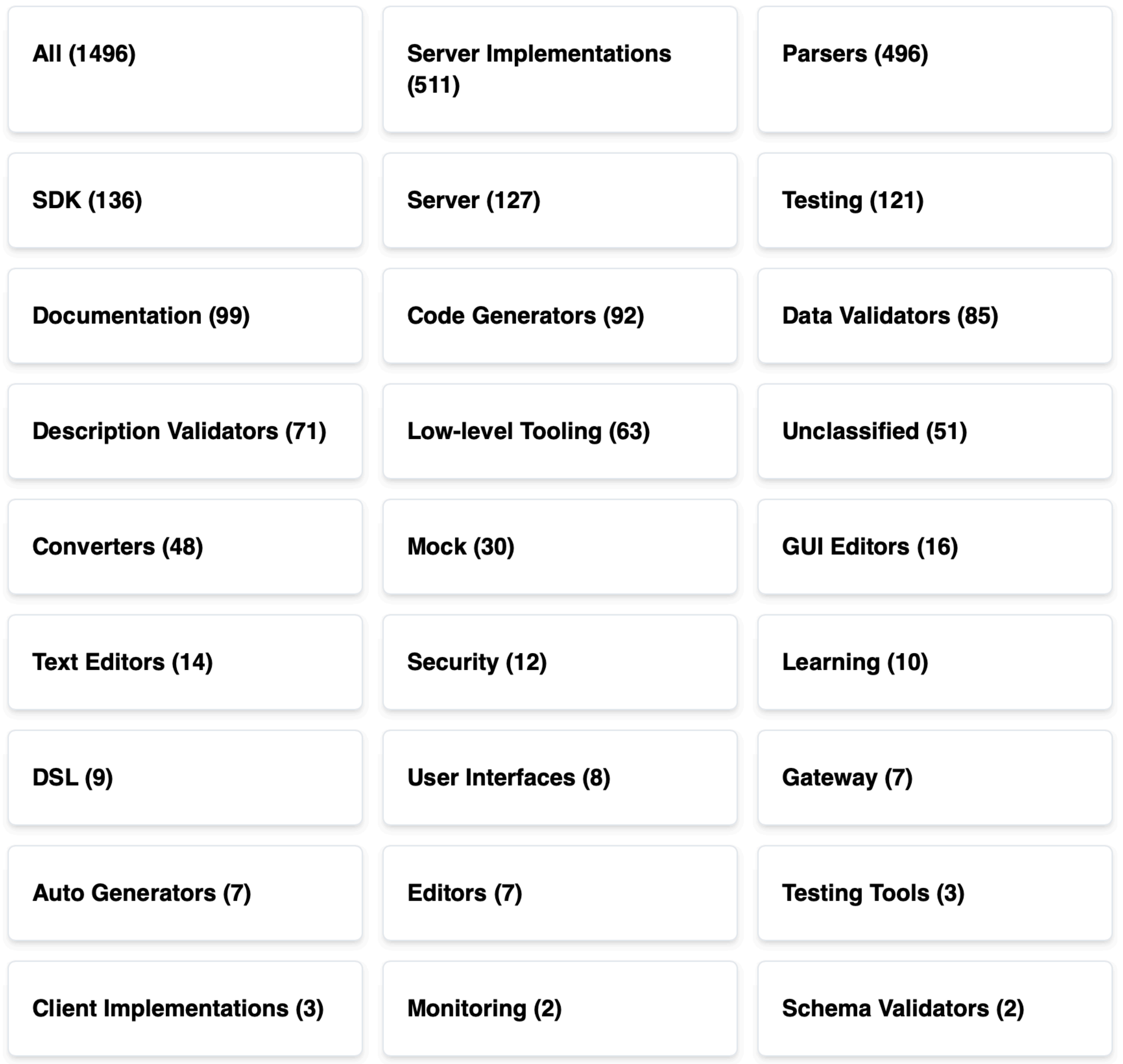
What are the benefits?

Long-term benefits:

- Automated testing / validation of future standards changes (CI/CD)
  - Changes to a standard could automatically be checked against any others that use them.
  - Immediately know potentially incompatible / breaking changes
- Standards *(code)* coverage:
  - What percentage of the standards do validators cover?
  - Reference implementations / validators can be tested against their own standards.
- Consistency between service providers
- Lower barrier to entry =
  more implementers with more clients, in more languages
- Easier to update with the next spec

It’s going to be okay!

- Nothing is going away tomorrow.
  - We’ll have prototypes, phases, parallel services…
- There will be breaking changes… *and there always has been…*
  - …and they probably won’t be as bad as we think…
  - …and it will be easier than before to update.
- <u>We’re not going into this alone</u>
  - We’ve got institutional buy-in.
  - OpenAPI is proven tech.
  - There is a rich ecosystem of tools, development and support.



All (1496)	Server Implementations (511)	Parsers (496)
SDK (136)	Server (127)	Testing (121)
Documentation (99)	Code Generators (92)	Data Validators (85)
Description Validators (71)	Low-level Tooling (63)	Unclassified (51)
Converters (48)	Mock (30)	GUI Editors (16)
Text Editors (14)	Security (12)	Learning (10)
DSL (9)	User Interfaces (8)	Gateway (7)
Auto Generators (7)	Editors (7)	Testing Tools (3)
Client Implementations (3)	Monitoring (2)	Schema Validators (2)