Using Swagger to tame HTTP/JSON interfaces

John Billings
billings@yelp.com
Yelp’s Mission:
Connecting people with great local businesses.
Yelp Stats:
As of Q3 2015

89M
90M
71%
32
HTTP/JSON is amazing!

HAProxy
NGINX
requests
curl
simplejson
Varnish
jq
Apache
Dropwizard
httplib
Pyramid

http://wallakitty.deviantart.com/art/Unicorn-attack-519106761
HTTP/JSON is amazing!

HAProxy  Apache
NGINX  curl
jq  simplejson
httplib  Varnish
Dropwizard  Pyramid
The website is down again!
%&*#! I just pushed the petstore service!
The pet resource takes a string, right?
No, I told you, it’s an int!
Curse you and your strings!
If only there were a better way to specify our API?
Option 1: Write spec docs

**Request:** GET /search

<table>
<thead>
<tr>
<th>attribute</th>
<th>required/no optional</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tag</td>
<td>optional</td>
<td>required tag, can specify more than once</td>
</tr>
<tr>
<td>status</td>
<td>optional</td>
<td>available / pending / sold</td>
</tr>
<tr>
<td>name</td>
<td>optional</td>
<td>the name of the pet</td>
</tr>
</tbody>
</table>

**Response:** An array of Pet objects, defined below
Option 1: Write spec docs

✓ It’s easy to get started
✓ People can comment if you use e.g. gdocs
✓ Approachable by non-technical individuals

✗ Implementation and spec can drift over time
✗ It’s easy to be imprecise
Option 2: Switch to Thrift / Protocol Buffers / Avro / ...

namespace java ns
namespace py ns

typedef i32 int

service MultiplicationService
{
    int multiply(1:int n1, 2:int n2),
}
Option 2: Switch to Thrift / Protocol Buffers / Avro / ...

✓ More efficient on the wire
✓ More efficient to decode than JSON

✗ Cannot use L7 technologies such as HTTP caching
✗ Difficult to debug on the wire
✗ Variable quality of support across languages?
Option 3: Write lots of integration tests

The tests become the de facto spec

“As a client, if I send this request to the service, then I should get back this response.”
Option 3: Write lots of integration tests

✓ You should already have (some) of these tests

✗ Final testing phase; slow to correct bugs at this stage
✗ Integration tests take a (relatively) long time to run
✗ Overall, probably only want to have a few of these?
Option 4: Write client libraries

The client library API becomes the spec for consumers
Option 4: Write client libraries

✓ Consumers don’t need to worry about wire protocol
✓ We’ve used this approach at Yelp, and it can work

✗ Lots of boilerplate
✗ Manual validation
✗ No spec for the wire protocol
✗ Still need integration tests from clientlib / service ifc
Or...

- Stick with our existing HTTP/JSON infrastructure
- Invent a machine-readable specification language to declaratively specify endpoints and return types
- Create tooling to generate client libs from specs
- Create tooling to perform server-side validation against endpoint specifications
- Create a vibrant open source community :)

yelp®
http://swagger.io/specification
A brief history of Swagger

- 2011-08-10 Version 1
- 2012-08-22 Version 1.1
- 2014-03-14 Version 1.2
  - Formal swagger specification document
- 2014-09-08 Version 2
  - Combine Resource Listings and API Declarations
- 2016-01-01 OpenAPI Specification
  - Supported by Google, Microsoft, IBM and others
```json
{
  "id": 42,
  "category": {
    "id": 2,
    "name": "string"
  },
  "name": "jackie",
  "photoUrls": [
    "string"
  ],
  "tags": [
    {
      "id": 10,
      "name": "rotweiler"
    }
  ],
  "status": "available"
}
```
Top-level Swagger spec

```yaml
---
swagger: "2.0"
info:
  description: "This is a sample server Petstore server. [...]"
  version: "1.0.0"
  title: "Swagger Petstore"
  # ...
host: "petstore.swagger.io"
basePath: "/v2"
paths:
  # ...
definitions:
  # ...
```

http://petstore.swagger.io/v2/swagger.yaml
Paths object

```
/pet/{petId}:
  get:
    tags:
    - "pet"
    summary: "Find pet by ID"
    description: "Returns a single pet"
    operationId: "getPetById"
    produces:
    - "application/xml"
    - "application/json"
    parameters:
    - name: "petId"
      in: "path"
      description: "ID of pet to return"
      required: true
      type: "integer"
      format: "int64"
    responses:
      200:
        description: "successful operation"
        schema:
          $ref: "#/definitions/Pet"
```
Another parameter object

- name: "status"
  in: "query"
  description: "Status values that need to be considered for filter"
  required: true
  type: "array"
  items:
    type: "string"
    enum:
      - "available"
      - "pending"
      - "sold"
    default: "available"
  collectionFormat: "multi"

Used for /pet/findByStatus endpoint
Definition object

```
Pet:
  type: "object"
  required:
    - "name"
    - "photoUrls"
  properties:
    id:
      type: "integer"
      format: "int64"
    category:
      $ref: "#/definitions/Category"
    name:
      type: "string"
      example: "doggie"
    photoUrls:
      type: "array"
      items:
        type: "string"
    tags:
      type: "array"
      items:
        $ref: "#/definitions/Tag"
    status:
      type: "string"
      description: "pet status in the store"
      enum:
        - "available"
        - "pending"
        - "sold"

{
  "id": 42,
  "category": {
    "id": 2,
    "name": "string"
  },
  "name": "jackie",
  "photoUrls": [
    "string"
  ],
  "tags": [
    {
      "id": 10,
      "name": "rotweiler"
    }
  ],
  "status": "available"
}
```
More definition objects: Maps

```
StringToStringMap:
  type: object
  additionalProperties:
    type: string
```

```
StringToFooMap:
  type: object
  additionalProperties:
    type: '#/definitions/Foo'
```
# Datatypes and formats

<table>
<thead>
<tr>
<th>Common Name</th>
<th>type</th>
<th>format</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>integer</td>
<td>int32</td>
<td>signed 32 bits</td>
</tr>
<tr>
<td>long</td>
<td>integer</td>
<td>int64</td>
<td>signed 64 bits</td>
</tr>
<tr>
<td>float</td>
<td>number</td>
<td>float</td>
<td></td>
</tr>
<tr>
<td>double</td>
<td>number</td>
<td>double</td>
<td></td>
</tr>
<tr>
<td>string</td>
<td>string</td>
<td></td>
<td></td>
</tr>
<tr>
<td>byte</td>
<td>string</td>
<td>byte</td>
<td>base64 encoded characters</td>
</tr>
<tr>
<td>binary</td>
<td>string</td>
<td>binary</td>
<td>any sequence of octets</td>
</tr>
<tr>
<td>boolean</td>
<td>boolean</td>
<td></td>
<td></td>
</tr>
<tr>
<td>date</td>
<td>string</td>
<td>date</td>
<td>As defined by full-date - RFC3339</td>
</tr>
<tr>
<td>dateTime</td>
<td>string</td>
<td>date-time</td>
<td>As defined by date-time - RFC3339</td>
</tr>
<tr>
<td>password</td>
<td>string</td>
<td>password</td>
<td>Used to hint UIs the input needs to be obscured.</td>
</tr>
</tbody>
</table>
Custom formats

- **EmailAddress**: type: string, format: email
- **IPV6Address**: type: string, format: ipv6

Ignored by Swagger, but some tooling may allow you to register your own validator.
Where do Swagger specs live?

- At Yelp we check them into the service codebase
- Serve from a well-known endpoint of the service
- This minimizes distance between spec and code
- Could also store all specs in a central repo
Modifying specs

- There’s no magic here
- Swagger will not prevent you doing something bad
- You-the-programmer need to make sure that all spec changes are backwards compatible
- If you like living safely, only add new endpoints
- If you like living dangerously, change some existing endpoints or remove some endpoints :}
A brief interlude

What's the best thing about UDP jokes?
A brief interlude

What's the best thing about UDP jokes?
I don't care if you get them
A brief interlude

What's the best thing about TCP jokes?
A brief interlude

What's the best thing about TCP jokes?
I get to keep telling them until you get them
A brief interlude

What's the best thing about TCP jokes?
What can I do with a spec?

• Review an API
• Browse other specs
• Generate a client library
• Perform server-side validation
• Testing
API reviews
Browsing specs

API for selected service

Different services

http://swagger.io/swagger-ui/
Perform a real query
Curl


Request URL

http://swagger_ui.paasta-norca-devc.yelp/italiapi/photos/v2/list?photo_ids=1

Response Body

[
  {
    "url_prefix": "https://s3-ap-southeast-1.amazonaws.com/yelp-business-images/026c1958-2197-4b4b-828b-9b95b130acb0/",
    "user_id": 3,
    "uploading_user_type": "user",
    "business_id": "v9z1q29I7iw",
    "time_created": 1115416311,
    "enc_user_id": "1234567890",
    "caption": "yelp street team @ marketbar",
    "encrypted_id": "1234567890",
    "slideshow_order": 71,
    "id": 1,
    "url_suffix": ".jpg"
  }
]
Brief aside: Same-origin policy
Solution using Cross-Origin Resource Sharing

swagger_ui  service_1  service_2

Access-Control-Allow-Origin: http://swagger_ui
Solution using a proxy

swagger_ui → NGINX → service_1, service_2
Generating client libs

```
   -o swagger-codegen-cli.jar
$ java -jar swagger-codegen-cli.jar generate
   -l python -i http://petstore.swagger.io/v2/swagger.json -o clientlib
reading from http://petstore.swagger.io/v2/swagger.json
writing file clientlib/swagger_client/models/order.py
writing file clientlib/swagger_client/models/category.py
writing file clientlib/swagger_client/models/user.py
[...]```

Try this out!
Using generated clientlibs

```python
from swagger_client import ApiClient
from swagger_client import PetApi
client = ApiClient()
pet_api = PetApi(client)
pet_api.get_pet_by_id(42)
{'category': [{'id': 2, 'name': 'string'},
              'id': 42,
              'name': 'jackie',
              'photo_urls': ['string'],
              'status': 'available',
              'tags': [{'id': 10, 'name': 'rotweiler'}]}
```
Bravado: dynamic clientlibs for Python

```python
from bravado.client import SwaggerClient
client = SwaggerClient.from_url("http://petstore.swagger.io/v2/swagger.json")
client.pet.getPetById(petId=42).result()
Pet(category=Category(id=2L, name=u'string'),
    id=42L,
    name=u'jackie',
    photoUrls=[u'string'],
    status=u'available',
    tags=[Tag(id=10L, name=u'rotweiler')])
```

https://github.com/Yelp/bravado
pyramid_swagger

This project offers convenient tools for using Swagger to define and validate your interfaces in a Pyramid webapp.

Features include:

- Support for Swagger 1.2 and Swagger 2.0
- Request and response validation
- Swagger spec validation
- Automatically serving the swagger schema to interested clients (e.g. Swagger UI)
pyramid_swagger: usage

Matched in swagger spec

def get_things(request):
    # Returns thing_id as an int (assuming the swagger type is integer)
    thing_id = request.swagger_data['thing_id']
    ...
    return {...}
pyramid_swagger: custom formats

```python
import base64
from pyramid_swagger.tween import SwaggerFormat

user_format = SwaggerFormat(format='base64',
                             to_wire=base64.b64encode,
                             to_python=base64.b64decode,
                             validate=base64.b64decode,
                             description='base64 conversions')
```

```json
{
    "name": "petId",
    "in": "path",
    "description": "ID of pet to return",
    "required": true,
    "type": "string",
    "format": "base64"
}
```
Oops!

Failed validating 'type' in schema:

```json
{
  'description': 'ID of pet to return',
  'format': 'int64',
  'in': 'path',
  'name': 'petId',
  'required': True,
  'type': 'integer'
}
```

On instance:

```plaintext
u'fourty-two'
```
def get_pet(pet_id):
    return {
        'id': 'foo',
        'category': {'id': 2, 'name': 'string'},
        'name': 'jackie',
        'photoUrls': ['string'],
        'tags': [
            {'id': 10, 'name': 'rotweiler'},
        ],
        'status': 'available'
    }
(venv) john@grunt:..tore/my_petstore$ curl -w\
localhost:8080/v2/pet/42
<html>
  <head>
    <title>500 Internal Server Error</title>
  </head>
  <body>
    <h1>500 Internal Server Error</h1>
    The server has either erred or is incapable of performing the requested operation.<br/>
    u'foo' is not of type 'integer'
  </body>
</html>

Failed validating 'type' in schema['properties'][id]:
  {'format': 'int64', 'type': 'integer'}

On instance[id]:
  u'foo'
Testing without Swagger

There could be inconsistencies across both of these interfaces
Testing with Swagger

There could still be inconsistencies across these interfaces

This interface is consistent by construction (*)
Testing with Swagger

- This is a fairly standard testing problem
- Your type-checker can help here (if you have one :)
- Future work: add support for returning mock data
Testing with Swagger

• Validate your responses as part of your testing
• Fairly easy if your service already contains a validator?
• Could also use an external validator
Other spec langs: API Blueprint by Apiary

# Message of the Day API
A simple [MOTD](http://en.wikipedia.org/wiki/Motd_(Unix)) API.

# Message [/messages/{id}]
This resource represents one particular message identified by its *id*.

## Retrieve Message [GET]
Retrieve a message by its *id*.

+ Response 200 (text/plain)

   Hello World!

## Delete Message [DELETE]
Delete a message. **Warning:** This action **permanently** removes the message from the database.

+ Response 204
Other spec langs: I/O Docs by Mashery

```json
{
    "name": "Lower Case API",
    "description": "An example api.",
    "protocol": "rest",
    "basePath": "http://api.lowercase.sample.com",
    "publicPath": "/v1",
    "auth": { ... },
    "headers": { ... },
    "resources": {
        "Resource Group A": {
            "methods": {
                "MethodA1": {
                    "name": "Method A1",
                    "path": "/a1/grab",
                    "httpMethod": "GET",
                    "description": "Grabs information from the A1 data set."
                }
            }
        }
    }
}
```
Conclusions

• Swagger provides an easy way to define JSON/HTTP interfaces for new and existing services
• Once you have an interface, you get lots of tooling ‘for free’
  • Automatic generation of clientlibs for many different languages
  • Automatic validation of requests and responses
Any questions?