5 Features of a Good API Architecture

Rob Allen

19ft.com ~ @akrabat ~ October 2016

Fit for Purpose

But first...

Your target audience matters!

Features of a good API

Malleability
Correctness
Client focus
Documented
Secure

A good API is malleable

Malleability

- The representation is independent of the DB schema

It's about focus

Different objectives

- API is client focussed, DB schema is application focussed
- Representation of data is different
- Your API becomes brittle

Malleability

- The representation is independent of the DB schema
- Design is based on state changes

Workflows are more than CRUD

- Operations are a series of steps
- Think about goals and sequences of actions
- Make it easy for a user to accomplish their tasks!

Malleability

- The representation is independent of the DB schema
- Design is based on state changes
- Hypermedia aware

Hypermedia

Hypermedia as the source of client flexibility!

- Rename endpoints at will
- Re-home endpoints on different servers (e.g. CDN resources)
- API is explorable

A good API is correct

Correct

- Media type format suits the design
- Correct use of HTTP verbs
- Understanding of Idempotency
- Richardson Maturity Model 2+

Media types matter

With *application/json* you abdicate responsibility.

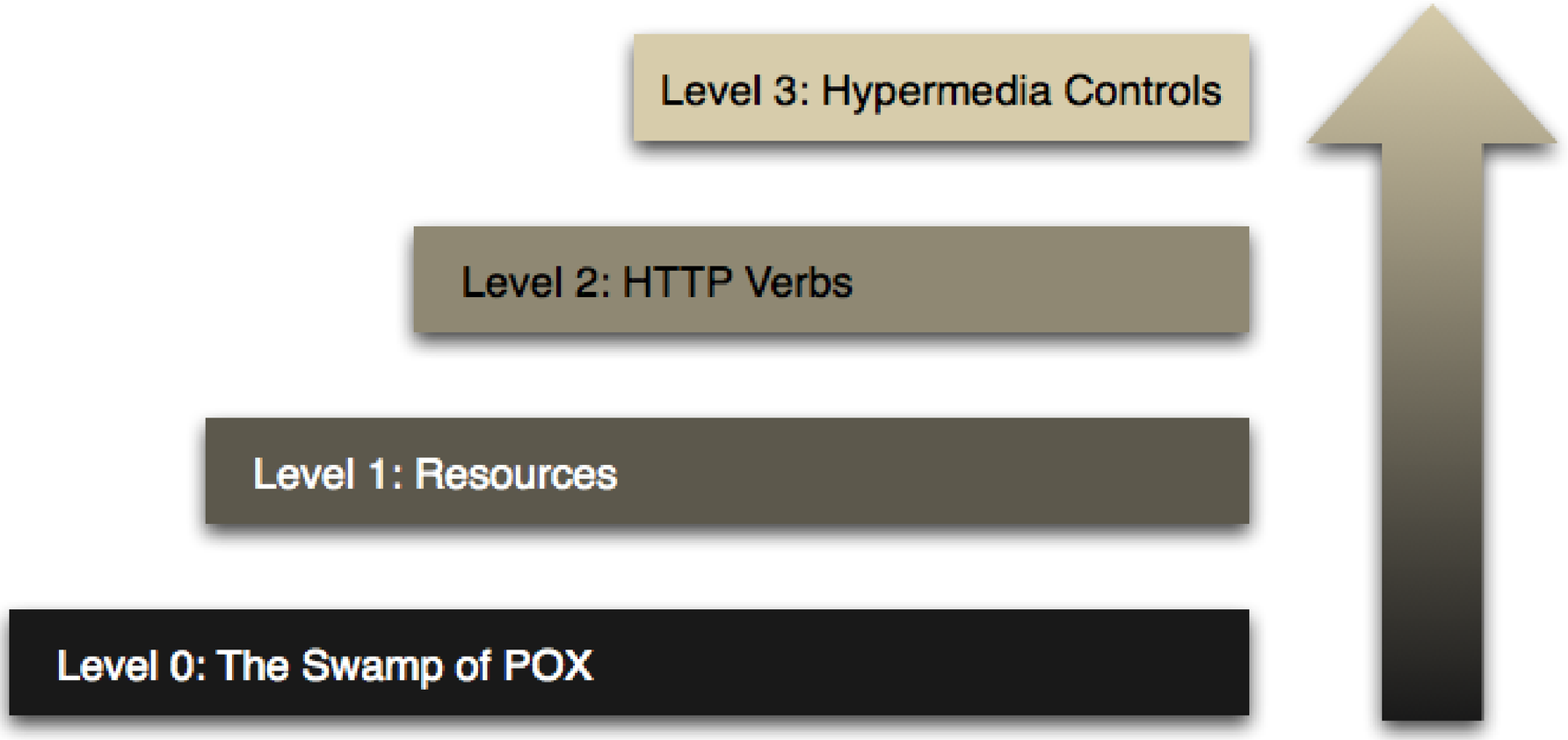
The correct media type:

- Tells the client how to interpret the data
- Enforces structure of the payload
- Informs on what the payload data means

HTTP verbs

Method	Used for	Idempotent?
GET	Retrieve data	Yes
PUT	Change data	Yes
DELETE	Delete data	Yes
POST	Change data	No
PATCH	Update data	No

Richardson Maturity Model



source: http://martinfowler.com/articles/richardsonMaturityModel.html

A good API respects the client dev

Great error handling

- Error representations are first class citizens
  - Honours accept header
  - Correct content-type
  - Uses correct HTTP status code
- Provides application error code & human readable message
  - ideally in a known media type such as api-problem
- Pretty print JSON for the humans!

Error messages

- End user needs a short, descriptive message
- Client developer needs detailed information
- Client application needs an error code
- The API application needs logs!

Error codes

Send correct HTTP status code

- `2xx` for success
- `4xx` for failures the client can fix
- `5xx` for failures the client cannot fix

Include an application error code

- For the client application to parse

HTTP Problem (RFC 7807)

```
HTTP/1.1 503 Service Unavailable
Content-Type: application/problem+json
Content-Language: en
```

```
{
    "status": 503,
    "type": "https://example.com/service-unavailable",
    "title": "Could not authorise user due to an internal problem - try later.",
    "detail": "The authentication service is down for maintenance.",
    "instance": "https://example.com/maintenance-schedule/2016-08-30",
    "error_code": "AUTHSERVICE_UNAVAILABLE"
}
```

Handling changes

- Avoid major new versions
- Make changes backwards-compatible
- Think about forwards-compatibility

BC changes

- Add resources
- New verbs to existing resources
- New endpoints
- New format via content negotiation
- Avoid change: New query arguments / new fields

A new version is a new API

- Creates URL proliferation
- Doesn't matter how you define it:
    - api.example.com/v2/user
    - api2.example.com/user
- Use Server header for minor and patch info

Deprecation policy

- Provide plenty of warning
- Remove from test servers first
- Communicate widely

A good API is documented

Documentation within the API

- Content-type header
- Link relations
- Profile links
- Structured data

Profile links

Profile links provide the application semantics (RFC 6906)

In the header:

```
Link: Link: <https://www.example.com/docs>;rel="profile"
Content-Type: application/hal+json;profile="https://www.example.com/docs"
```

In the body:

```
"_links": {
  "profile": {
    "href": "https://www.example.com/docs/"
  }
}
```

Structured data

There's no such thing as self-explanatory!

```
{"christian": "Rob", "patronymic" : "Allen"}
{"forename": "Rob", "surname" : "Allen"}
{"firstname": "Rob", "lastname" : "Allen"}
```

Structured data

There's no such thing as self-explanatory!

```
{"christian": "Rob", "patronymic" : "Allen"}
{"forename": "Rob", "surname" : "Allen"}
{"firstname": "Rob", "lastname" : "Allen"}
```

https://schema.org/Person:

```
{"givenName" : "Rob", "familyName" : "Allen"}
```

Human documentation

- Sensible URLs
- Tutorials
- Semantic information

Sensible URLs

- The actual URL does not matter to the client API
- It does matter to the client developer!

```
GET /events
GET /events?filter=upcoming
GET /events/455dbab6
GET /events/455dbab6/talks
GET /talks/6cf948e5
GET /talks/6cf948e5/comments
```

The for-human documentation

- Tutorials to show how to use the API
- Describe all the semantic information
- Examples in multiple languages

A good API is secure

Security

- SSL
- Authentication
- Rate limited

Authentication

- HTTP Basic
- OAuth2

OAuth 2

- Requires an access token
- More than one way to get a token:
    - Via username/password
    - Via API server
- Tokens are short life with refresh token to get a new one

Rate limits

Limit by application or user token
Provide information

```
HTTP 429 Too Many Requests

X-RateLimit-Limit: 5000
X-RateLimit-Remaining: 4999
X-RateLimit-Reset: 1471549573
```

To sum up

Thank you!

Rob Allen ~ 19ft.com ~ @akrabat