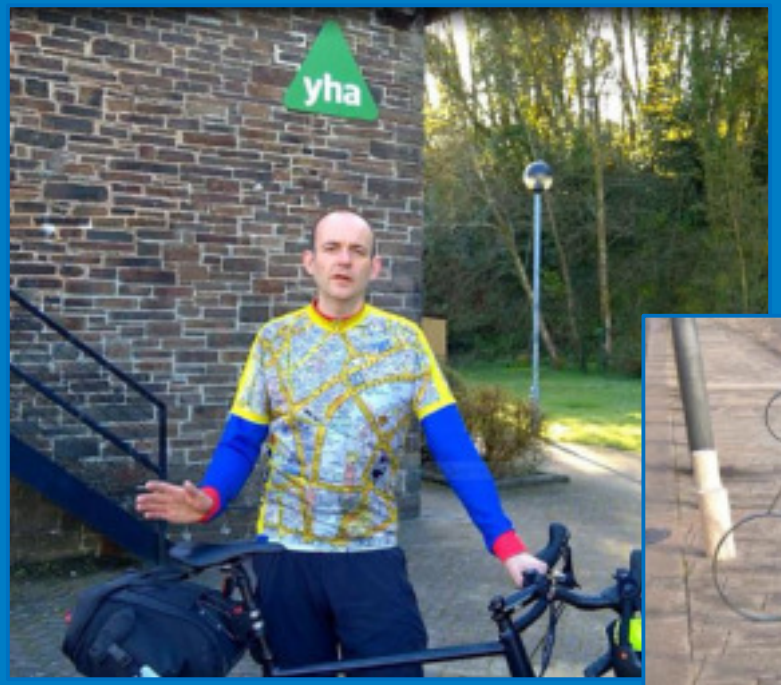
Give it a REST

Tips for designing and consuming public API's



< cycling

Porto >

< work

huddle

Is that really REST?

What is REST?

- Roy Fielding original proposal from 2000
- ~~HATEOAS~~ – Hypermedia API
- Richardson Maturity Model

Roy Fielding

'Hypermedia as the engine of application state" is a REST constraint. Not an option. Not an ideal. Hypermedia is a constraint. As in, you either do it or you aren't doing REST.'

Mike Amundsen tweet highlights that REST ≠ CRUD

'Remember, when designing your WebAPI, your data model is not your object model is not your resource model is not your message model.'

https://sebastianbooksblog.wordpress.com/2015/07/05/3390
Roy Fielding, Hoang Xuan Pham and Daniel A. Anderson / University Communications

Anatomy of a REST request

A REST request is a HTTP* operation on a resource, which may include a message

```
GET https://api.huddle.net/files/users/123/recentitem

Authorization: OAuth2 Imh0dHBzOi8vbG9naW4uaHVkZGxlLm5ldCIsImF1ZCI6IiouaHVkZG
Accept: application/json
User-Agent: HuddleDesktopPC/4.4.0.0
If-Modified-Since: Wed, 25 Apr 2018 08:44:40 GMT
Host: api.huddle.net
```

- HTTP Verb: **`GET`**
- Resource: **`https://api.huddle.net/files/users/123/recentitems`**
- Headers: **`Authorization`**, **`If-Modified-Since`**
- Message (optional): body of request, may be JSON, XML or even just plain text

* for pedants it doesn't have to be HTTP

Anatomy of a REST response

```
Content-Length: 36209
Content-Type: application/json
Last-Modified: Wed, 25 Apr 2018 08:45:53 GMT

{
  "maxItems": 50,
  "items": [
    {
      "type": "Document",
      "links": [
        {
          "rel": "delete",
          "href": "https://api.huddle.net/files/users/123/recentitems/456"
        },
        {
          "rel": "self",
          "href": "https://api.huddle.net/files/documents/456"
        },
        {
          "rel": "alternate",
          "href": "https://my.huddle.net/workspaces/789/files/456"
        }
      ],
      "title": "Notes - Huddle API Design",
      "contentType" : "application/vnd.openxmlformats-
officedocument.wordprocessingml.document",
      "extension" : "docx",
                :              :              :              :
```

- HTTP Status: **`200 OK`**
- Headers: **`Content-Type`**, **`Last-Modified`**
- Message (optional): body of request, may be JSON, XML or even just plain text
- Hypermedia API links: collection of links to resources with **`rel`** attribute to allow identification of the appropriate link

But what about gRPC?

- gRPC
  - Remote Procedure Call system created by Google
  - Always connected, HTTP/2
  - Binary protocol (protobuff),
  - Procedure based rather than resource based

A bit like Java RMI, or CORBA or DCOM or SOAP

We've been here before, however much we tried to forget …

API design – where do you start?

- Human and automated users
- Readable by a human
  - SOAP and XML were not inherently readable
- While you have a model on the server it may not fit all your clients
- Should you design for the server or for the client?
  - Design for the business
  - Balance of Security, Reliability and Speed

At Huddle we have an API design Slack channel

Remember – there is **NO** right answer for everyone!

Prohibited icon made by https://www.flaticon.com/authors/roundicons
Green letters by http://gimpchat.com/viewtopic.php?f=11&t=5232

An API is a contract

An API is a contract between the publisher and the consumer, which creates a set of mutual obligations

- For the person publishing the API
  - Document the API – responses, headers, acceptable media types, sample code
  - Make it clear what is optional, and what might be extended
  - Consider how different clients might use the API and what local caching might exist
- For the person consuming the API
  - Read the documentation and expect to handle all the possible responses
  - Use only what you need from the response to minimise your coupling

Documentation is key whatever form it is in; wiki, Swagger, Apiary.io. Include as much information as possible, including expected HTTP responses, media types, headers as well as URI and resource formats

Versioning an API

On the Nordic API web site, there is a blog which references Zdenek Nemec talking about API Change Management,

What versions exist?

- Client version
- Message Format version
- API implementation (server) version,
- API documentation version
- Resources, relationships between resources and the API itself does not have a version

In an ideal world versioning is **not** simply adding v2, v3 to the URI. At best use extension strategies, and if you have radically new formats they better modelled as new resources with new URIs

Extending an API

What are the extension rules?

- You **must not** take anything away
- You **must not** change processing rules
- You **must not** make optional things required
- Anything you add **must** be optional (you may use default values, either within the form in the case of hypermedia, or on the API implementation on the server)

Monitor feature usage

- see who keeps using deprecated fields/methods

Formalise how you communicate API changes and timescales

How do we introduce breaking changes?

Zdenek Nemec has an answer for this question as well, again for brevity; if a change to either of the following violate the extension rules listed earlier, you simply need to create a new resource:

- Resource Identifier (the URI) including any query parameters and their semantics
- Resource metadata (such as the HTTP headers)
- Resource data (such as the HTTP body) fields and their semantics
- Actions the resource affords (e.g., available HTTP Methods)
- Relations with other resources (e.g., Links)

If an existing client does not produce the same results with the same resources as previously you have broken the API. A client can not be expected to intelligently handle new data items, or use different or additional HTTP methods to perform the same operations

Patterns and strategies

Deprecation

Elegant deprecation

Too many failed attempts. Please fill out the captcha to protect your account.

Email

liam.westley@tigernews.co.uk

Password Forgotten your password?

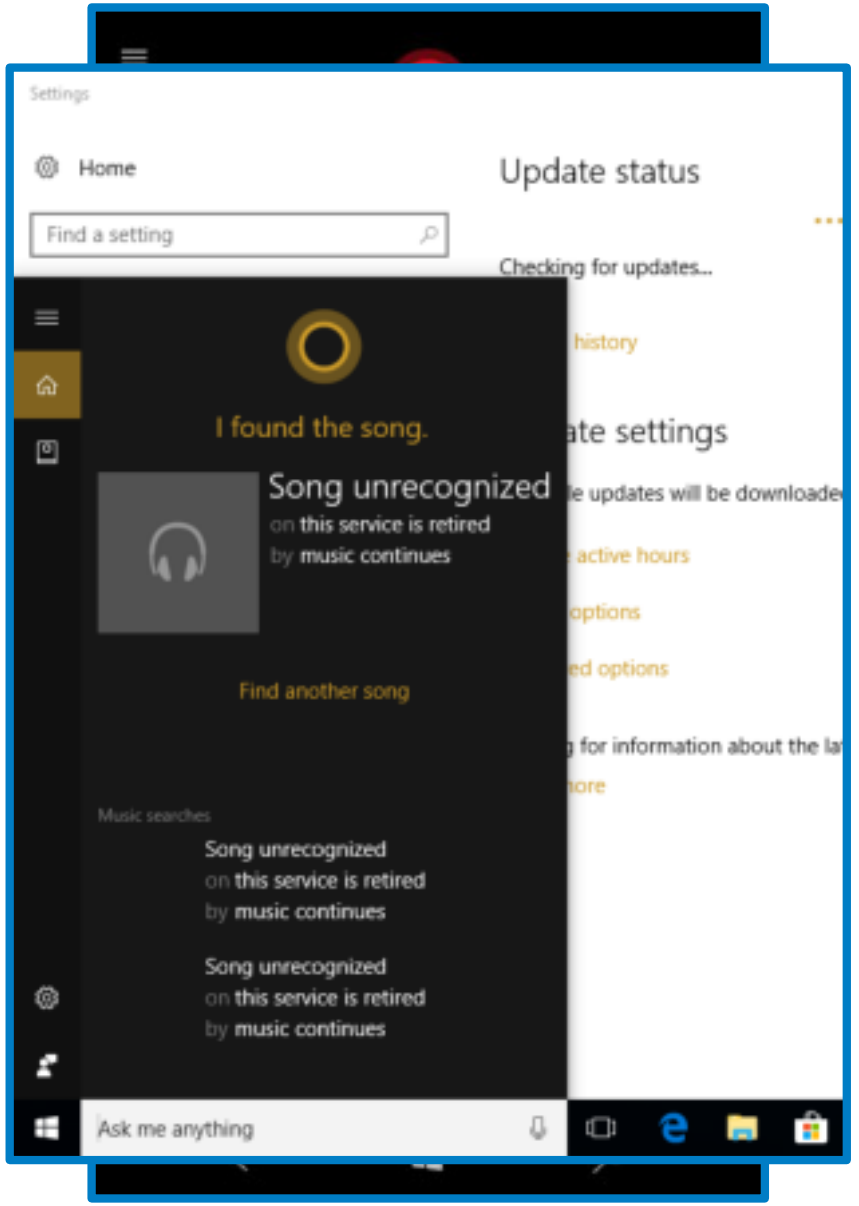
V1 UNSUPPORTED

Please direct siteowner to g.co/recaptcha/upgrade

reCAPTCHA™

Type the text

Privacy & Terms



huddle

BFF – for compatibility and tailoring

BFF – Backend For Frontend (Thoughtworks Tech Radar 2016)

Compatibility

- If you are re-engineering your data within the backend, you may generate a new API with different semantics
- You can create a BFF to replicate the old API semantics, making use of the new API

Tailoring results to specific clients

- Some clients may want data in a different form to the original API
- Mobile clients have network performance issues so it can be useful aggregate calls
- A BFF may be only used by internal clients, making changes to the BFF much easier

Using links to construct a user interface

With a Hypermedia API when you deliver a resource, you include various links within the response to provide information to the consumer of what operations are available

The presence and absence of links in the response can provide the consumer with a means of constructing a valid user interface for a resource, and can work well with user permissions

You can supply edit, update, delete, download **rel** links

Based on presence, or absence of links the UI can include the appropriate buttons and links

Links reflect the permission you might have on a document; i.e. the difference between a viewer and an editor of content

Hypermedia UI (consumer/publisher)

We use this model in Huddle, but there are limitations if those links are affected by not just permissions, but the state of a resource

When a document is locked for editing we do not provide an edit link, so there is no Edit button (acting as if a user had a read only permission)

When we added web push notifications (via WebSockets) our Single Page Application receives a notification saying the document is not unlocked …

… but there is no Edit button to enable, as it was never created in the HTML DOM, as it did not receive a link to support editing the document, as when it first loaded the document resource it was locked

Caching – `304 Not Modified`

`Last-Modified, If-Modified-Since` headers

- **`Last-Modified`** header included when a resource is provided
- The consumer than uses that value in an **`If-Modified-Since`** header when requesting a resource
- The API returns **`304 Not Modified`** if nothing has changed and the consumer knows that the resource it already has is up to date

ETag (Entity Tag)

- Strong and weak Etags
  - "123456789“ – a strong ETag validator, based on a hash of the resource
  - W/"123456789" – a weak ETag validator, often hashed on a version of the resource
- Response header – **`ETag: "686897696a7c876b7e"`**
- Request header – **`If-None-Match: "686897696a7c876b7e"`**

Last-Modified header (producer)

Real world example – BFF to 'replicate' existing API

- Original API included a `Last-Modified` header
- The BFF did not include this header (an oversight, not be design)
- Clients (iOS in this case) were now making multiple requests rather than a single request when data is unchanged

Mitigation

- The header was part of the original API contract
- Replicate that behaviour in the new BFF, achieved by passing the header from the new API straight through to the BFF

We could have also modified the client app to use the new API, or handle the new BFF response better … but updating every iOS app might take **SIX** months

Rate limiting – `429 Too Many Requests`

API's can utilise rate limiting, especially for particularly chatty clients

- Provide a **`Retry-After`** header to indicate to consumers how long to wait before retrying an operation
- Can rate limited based on specific clients
- Can be used with leaky bucket strategy to monitor API usage

Consumers

- There is a responsibility on consumers to honour **`429`** responses
- Consumers should honour the back off time
- If consumers treat rate limiting as just a standard error and immediately retry an operation, the benefit can be lost

Even if consumers ignore the **`429`** response it can still protect servers, as the response is much more efficient to produce.

Async and REST

Progress endpoint pattern

- POST operation
- Returns **`202 Accepted`** and a link for progress / exception reporting
- Client polls the link for completion of operation
- On completion the response either returns the final response or provides a link to the resource for the completed operation

Once had a developer ask for how to make an API call asynchronous (using **`async`** in C#), in that it could return control immediately after issuing a request, but if anything went wrong they would like it to somehow provide a **`HttpStatusCode`** straight away

The progress endpoint pattern provides a better model – no need to make the request async in the client, if initial validation and/or authentication are successful the server returns **`202 Accepted`** with a progress endpoint for obtaining the final result

Mini PUT/POST …../status example

Often you want to update a subset of information in a resource, often only a single property of a resource, such as status.

- PATCH operation could be used but requires a specially format patch data format such as JSON-patch, and PATCH is not guaranteed to be idempotent
- Can use PUT (or POST) on a 'child' resource
  - PUT **`https://api.huddle.net/files/document/123`** updates whole resource
  - PUT **`https://api.huddle.net/files/document/123/readonly`** updates just the read only status of the document
- Can return a full representation of the resource after the PUT is successful
  - If you do provide a **`Content-Location`** header to indicate that the response is at a different location to the resource on which the PUT was made

Real world examples

Updating FileRequest status

Huddle has a concept of a task that is a request for a file, which includes a **`status`** property. The front end team realised that the standard REST model was really heavy and asked the back end for a solution to updating the status … a mini PUT.

- If you update the status in the web UI these calls have to be made
  - GET **`https://api.huddle.net/tasks/123`** to get latest task resource
  - PUT **`https://api.huddle.net/tasks/123`** to save with updated status
  - GET **`https://api.huddle.net/tasks/123`** to get latest task resource
- Currently under development this will change to
  - PUT **`https://api.huddle.net/tasks/123/status`** to save the new status and return the latest state of the resource

This reduces server calls, and clearly indicates the business process being invoked.

Over consuming (consumer)

Just because a response includes data does not mean it all has to be deserialized into a local version of a resource

Responses are messages, there is no requirement to process all the data returned, especially on GET calls

Real world example – Huddle Desktop

When a QA tested putting an invalid URI into the personal website of their user profile they not only proved it is not validated on our server, but Huddle Desktop also had problems; despite being a string in the JSON response, Huddle Desktop deserialized the personal website to a `System.URI` which failed

At no point was the personal web site used by Huddle Desktop, the inclusion in the deserialization was for 'completeness' and possible 'future use'

Enumerated values (consumer)

Real world example – Android ToDo list processing, when a new type was added, FileRequest, the code threw an exception and the list failed to load

```
switch (type) {
  case "task":
    taskType = ActionType.TASK;
    break;
  case "approval":
    taskType = ActionType.APPROVAL;
    break;
  default:
    throw new JSONException("Unknown item type in todo list.");
}
```

Mitigation

- List could only show tasks that it knows how to display
- The app could provide information to user that there are items it cannot display

Relying on link order (consumer)

Real world example – BFF to ‘replicate’ existing API

Android app was grabbing first link to perform a **`PUT`** on updating last-seen item

```
{
    "unseenCount": 0,
    "links": [ {
        "rel": "self",
        "href": "https://api.huddle.net/notifications/user/20906501/received/last-seen"
    }, {
        "rel": "alternate",
        "href": "https://api.huddle.net/notifications/user/20906501/received",
        "title": "Received Notifications"
    } ],
    "lastNotificationSeen": {
        "id": "urn:uuid:1eb8b39a-72d3-4666-9451-3cfcd337cf79",
        "createdDate": "Tue, 24 Apr 2018 14:38:36 GMT"
    }
}
```

Mitigation

- Had to order the links in the response in **exact** order of original API event though that is not relevant in a Hypermedia API, clients should use the **`rel`** attribute not ordering

Resources

- https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
  - Roy Fielding's original doctorate dissertation from 2000
- https://nordicapis.com/blog/
  - Great active discussion of everything related to APIs
- http://slack.httpapis.com
  - Slack channel, created by Sebastien Lambla (serialseb) where you can
- http://www.bizcoder.com/http-pattern-index
  - HTTP REST patterns, such as progress endpoint pattern
- http://restcookbook.com/
  - Full of guidance on how to create RESTful API and basic fundamentals of REST

https://blog.liamwestley.co.uk

@westleyl

Huddle is hiring - https://www.huddle.com/about/careers-huddle/

London

2 Leman Street

2nd Floor, Aldgate Tower

London, E1 8FA

San Francisco

156 2nd Street

San Francisco

CA, 94105

Washington DC

7910 Woodmont Avenue #1250

Bethesda

MD, 20814