# Zen and the art of Hypermedia design
A mostly practical introduction to REST hypermedia design

Ori Pekelman

January 30th 2013 TakeOffConf Lille

# Contents

# Introduction

So.. I am Ori Pekelman, and for the past year I have been running a consulting company called Constellation Matrix; you can find me on twitter/github/linked-in and such as OriPekelman. I have building web stuff and therfore APIs for the past twenty years or so. Over this last year I did mostly Data (biggish) and API oriented projects and I come before you to share some of the insights I gained.

And I am going to try to sell you on doing Hypermedia.

---

**foreword**

REST[1] is in, and by itself a software design pattern.

Design patterns are common solutions to recurring problems in software engineering.

When we talk about REST, we generally talk about the use of accepted Web standards.[2]. As REST, as an architectural style is born from the post-hoc analysis of the efficiencies of the HTTP model.

Its implementation, its specific implementation is very much dependent on the enviroment in which it is needed. Very much like singleton or strategy chain which may or may not make any sens depending on the programming language you are using.

[1]Fielding, Roy Thomas. Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation, University of California, Irvine, 2000.

[2]Design Patterns: Elements of Reusable Object-Oriented Software by ErichGamma, RichardHelm, RalphJohnson, and JohnVlissides (the GangOfFour), AddisonWesley Professional (November 10, 1994)

My favorite citation on design patterns is:

> "Design patterns are bug reports against your programming language"[3] – *Peter Norvig*

What will be proposed here are therfore bug reports againts faulty implementations of REST, or at least recommendations of common solutions to common issues.

A design pattern has a scope, and REST is an architecture, or system level design pattern that specifically treats the issue of orchestrating or interfacing two or more heterogenous software systems.

As many design patterns it has applications and implications beyond its immediate scope. Hypermedia REST promotes leveraging the existing infrastructure of the Web in order to make two differnet pieces of software talk to each other.

[3] http://norvig.com/design-patterns/Design Patterns in Dynamic Languages First put online 17 March 1998; first presented 5 May 1996

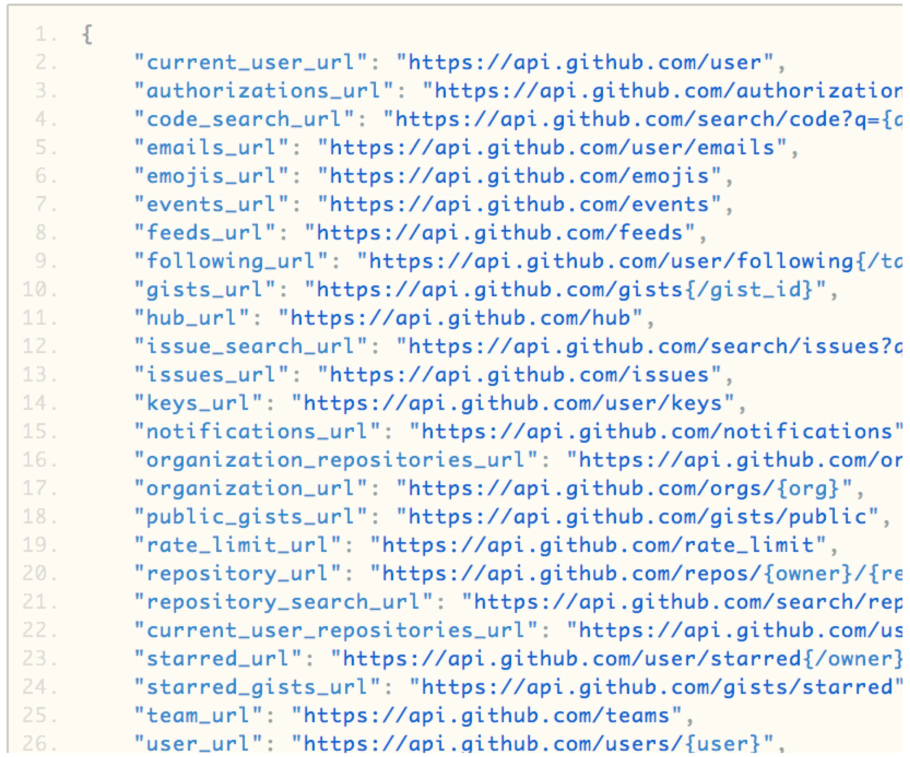
Chapter 1

# State of Hypermedia

**We are in early 2014 where are we at on adoption?**

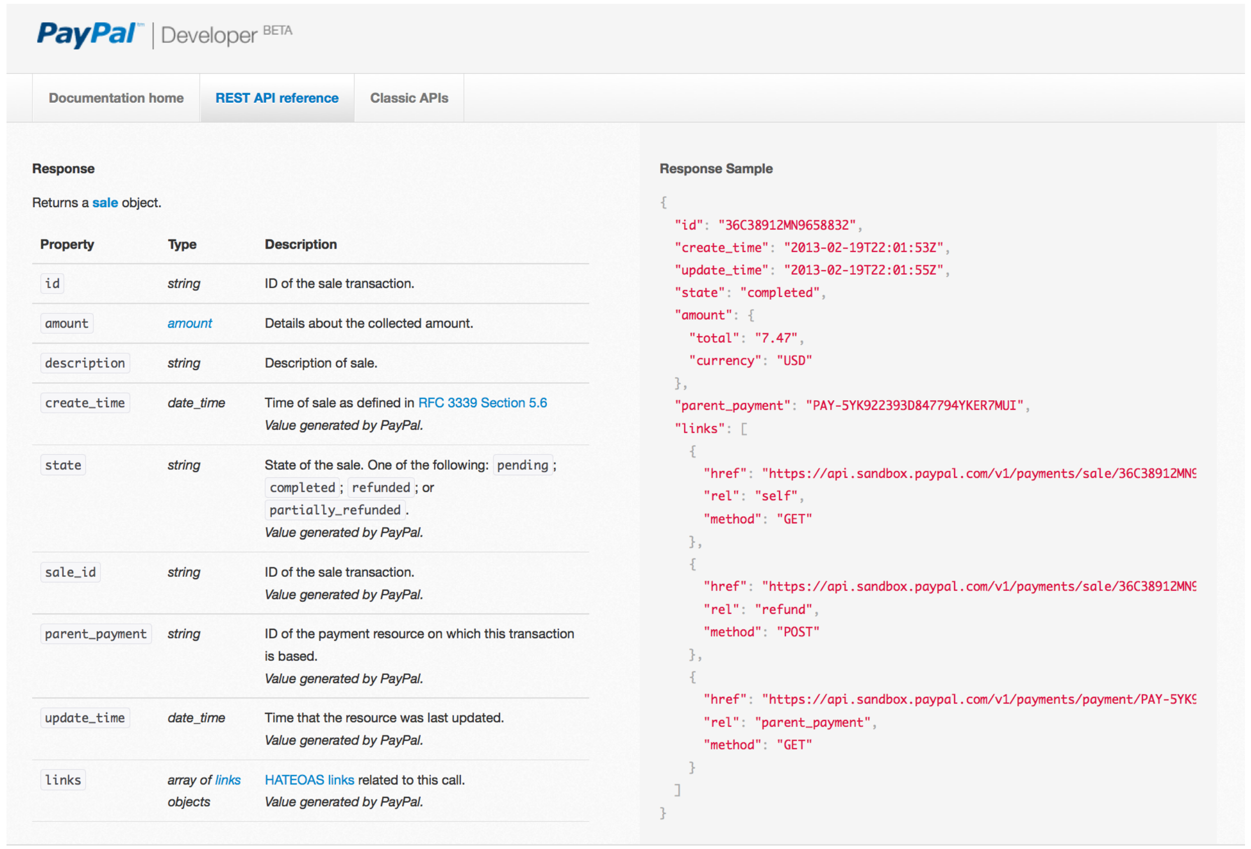
I haven't even told you yet was this was all about, but I promised I will be down to earth on this talk so let me first try to convince you this is not some fringe thing.

Github, you know the nice, lovely, competent, Github right? They have some very nifty hypermedia API action going around. http://developer.github.com/v3/



```
{
    "current_user_url": "https://api.github.com/user",
    "authorizations_url": "https://api.github.com/authorizatior
    "code_search_url": "https://api.github.com/search/code?q={c
    "emails_url": "https://api.github.com/user/emails",
    "emojis_url": "https://api.github.com/emojis",
    "events_url": "https://api.github.com/events",
    "feeds_url": "https://api.github.com/feeds",
    "following_url": "https://api.github.com/user/following{/tc
    "gists_url": "https://api.github.com/gists{/gist_id}",
    "hub_url": "https://api.github.com/hub",
    "issue_search_url": "https://api.github.com/search/issues?c
    "issues_url": "https://api.github.com/issues",
    "keys_url": "https://api.github.com/user/keys",
    "notifications_url": "https://api.github.com/notifications"
    "organization_repositories_url": "https://api.github.com/or
    "organization_url": "https://api.github.com/orgs/{org}",
    "public_gists_url": "https://api.github.com/gists/public",
    "rate_limit_url": "https://api.github.com/rate_limit",
    "repository_url": "https://api.github.com/repos/{owner}/{re
    "repository_search_url": "https://api.github.com/search/rep
    "current_user_repositories_url": "https://api.github.com/us
    "starred_url": "https://api.github.com/user/starred{/owner}
    "starred_gists_url": "https://api.github.com/gists/starred"
    "team_url": "https://api.github.com/teams",
    "user_url": "https://api.github.com/users/{user}",
```

And... evil PayPal is in the game too...


PayPal | Developer BETA

Documentation home | REST API reference | Classic APIs

**Response**

Returns a sale object.

| Property | Type | Description |
|---|---|---|
| `id` | *string* | ID of the sale transaction. |
| `amount` | *amount* | Details about the collected amount. |
| `description` | *string* | Description of sale. |
| `create_time` | *date_time* | Time of sale as defined in RFC 3339 Section 5.6 *Value generated by PayPal.* |
| `state` | *string* | State of the sale. One of the following: `pending`; `completed`; `refunded`; or `partially_refunded`. *Value generated by PayPal.* |
| `sale_id` | *string* | ID of the sale transaction. *Value generated by PayPal.* |
| `parent_payment` | *string* | ID of the payment resource on which this transaction is based. *Value generated by PayPal.* |
| `update_time` | *date_time* | Time that the resource was last updated. *Value generated by PayPal.* |
| `links` | *array of links objects* | HATEOAS links related to this call. *Value generated by PayPal.* |

**Response Sample**

```
{
  "id": "36C38912MN9658832",
  "create_time": "2013-02-19T22:01:53Z",
  "update_time": "2013-02-19T22:01:55Z",
  "state": "completed",
  "amount": {
    "total": "7.47",
    "currency": "USD"
  },
  "parent_payment": "PAY-5YK922393D847794YKER7MUI",
  "links": [
    {
      "href": "https://api.sandbox.paypal.com/v1/payments/sale/36C38912MN
      "rel": "self",
      "method": "GET"
    },
    {
      "href": "https://api.sandbox.paypal.com/v1/payments/sale/36C38912MN
      "rel": "refund",
      "method": "POST"
    },
    {
      "href": "https://api.sandbox.paypal.com/v1/payments/payment/PAY-5YK
      "rel": "parent_payment",
      "method": "GET"
    }
  ]
}
```

As are Amazon, for appstream for example (and probably all new APIs they are going to push).

Closer to here (and we will look closer at these ones)

- The Credit Agricole has a hypermedia API to let App developers access personal banking data!
- Canal TP offers a Multi-modal transportation hypermedia API
- VideoMuseum lets you discover modern and contemporary art through their own...
- If you can't see a pattern emerging you are right. There isn't a specific sweetspot for this.
- The only common point is : these are APIs born last year.

And A year ago I would have been hard pressed to find a single example in production, now I can think of dozens off the top of my head.

Not everyone was converted but slowly, hypermedia has become mainstream; I posit it is soon to become the standard.

# Chapter 2

# Design for clarity

**Clarity comes from within**

---

> Rinzai Zen is marked by the emphasis it places on kensho ("seeing one's true nature") as the gateway to authentic Buddhist practice, and for its insistence on many years of exhaustive post-kensho training to embody the free functioning of wisdom within the activities of daily life. [1]

Lets not forget the master word here: "design". And design is about intent.

```
The API is our users' path to us

We must be our APIs first users

Therfore the API is a path to self knowledge
```

[1] [^http://en.wikipedia.org/wiki/Rinzai_school](^http://en.wikipedia.org/wiki/Rinzai_school) Rinzai School, Wikipedia

From master Fielding:

> A REST API must not define fixed resource names or hierarchies (an obvious coupling of client and server). Servers must have the freedom to control their own namespace. Instead, allow servers to instruct clients on how to construct appropriate URIs, such as is done in HTML forms and URI templates, by defining those instructions within media types and link relations. [Failure here implies that clients are assuming a resource structure due to out-of band information, such as a domain-specific standard, which is the data-oriented equivalent to RPC's functional coupling]. [2]

---

A REST API should be entered with no prior knowledge beyond the initial URI (bookmark) and set of standard-

[2] http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven REST APIs must be hypertext-driven, Roy T. Fielding, 20 Oct 2008

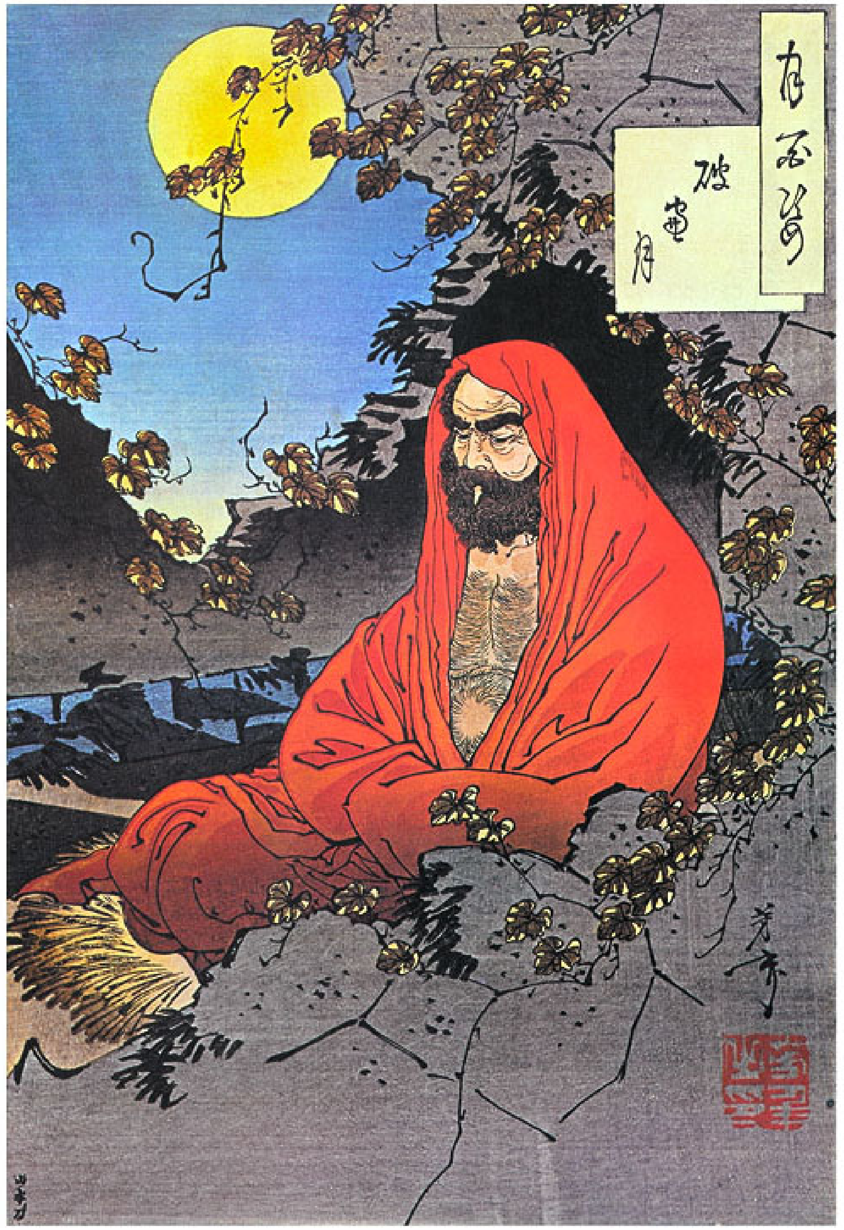


Figure 2.1: Bodhidharma, by Yoshitoshi, 1887

ized media types that are appropriate for the intended audience (i.e., expected to be understood by any client that might use the API). From that point on, all application state transitions must be driven by client selection of server-provided choices that are present in the received representations or implied by the user's manipulation of those representations. The transitions may be determined (or limited by) the client's knowledge of media types and resource communication mechanisms, both of which may be improved on-the-fly (e.g., code-on-demand). [Failure here implies that out-of-band information is driving interaction instead of hypertext.]

[3]

So these are qualities we ask of any REST API. Hypermedia is a design pattern on top of this one that makes achieving these qualities easier… and more predictable.

## This is what in Restspeak we call Affordance

[3] http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven REST APIs must be hypertext-driven, Roy T. Fielding, 20 Oct 2008

Chapter 3

# What makes a good API?

**Tracking back a little**

1. It is abstract: requires minimal knowledge about the implementation of the program exposed
2. It is standard: imposes the absolute minimum client side dependencies
3. It is general: it makes minimal assumptions on the way it is going to be used
4. It is simple: it allows to implement very quickly the most common use-cases (sometimes orthogonal to generality)
5. It is self-describing: it requires only minimal documentation
6. It is predictable: it does not surprise the user and it adheres to common patterns and conventions

**It is about Ubiquity. Simplicity. Reuse. Affordance. Scale, and economy of scale.**

In previous talks about hypermedia APIs I insisted on the fact that the API is a product[1]; not a by-product of your system. It should never leak internals. It must be designed.

[1] http://www.pekelman.com/presentations/apidays/ Lipstick on a pig: How (not) to design a modern API over legacy systems

## Abstract, Standard, General, Simple, Self-describing, Predictable

These are our design goals.

### 3:30:3 My personal Litmus test

**On the homepage of your API** A developer should:

1. Understand in 3 seconds what your API is for.
2. Be able to indentify the entry point in 30 seconds.
3. Be able to create an account, call the system, and use the result in under 3 minutes.

Chapter 4

# Hypermedia is affordable REST

**The simplest low down**

- Basically this means Hypermedia is self describing RESTful APIs.
- On the "read side" it allows you to discover one resource by the other
- On the "write side" it allows the client to manage state transitions and the server to describe the next steps to the client
- **The API itself being described is a resource. Even an important one**

---

- Software deals with entities that have data
- In RESTspeak we call these entities "resources"
- An API will exposes methods to read and write data
- Writing data can be expressed as a state transition

**So. . . Whats is a resource?**

- Something that can be uniquely addressed.
- Something that can be addressed, can be addressed by a url.

Figure 4.1: Basically this is about URLs and relations between URLSs

Chapter 5

# How do REST / Hypermedia urls look like?

**REST urls look like just normal urls**

http://www.w3.org is a good one. Also http://pekelman.com/time or http://pekelman.com/time.json or https://github.com/OriPekelman/paris-rb-grape-talk/commits/master

I will get back to this ### But they usually look like:

- http://api.example.com/{resource-type}/{resource-id}.{output-format}?{filters .. and .. api_key .. as .. arguments} **or**
- http://api.example.com/{collection} / {item} / {collection} / {item} / {sub_resource} **or something like**
- http://api.example.com/ {ver} / {lang} / {resource-type} / {resource-id}.{output-format}

This last one is probably a **bad example** it is not DRY at all… we repeat twice the content-type (headers + extension).

**So, again, what is hypermedia?**

We have seen that "REST urls are just urls" well a hypermedia API is just a REST API with links.

And even more down to earth.. a REST api with elements that have a "href" can be considered hypermedia. Not a perfect one.

There is more to it. But just linking your resources together gives already tremendous value to those using it; It works because the Web works.

If you don't know what to chose chose HAL http://stateless.co/hal_specification.html

Otherwise look at http://jsonapi.org/

Links make stuff discoverable, crawlable, automatable, **composable**, interoperable!

It makes your services possibly a part of a larger whole!

We could hardly make a Google without links right? and "rel" oh beautiful **"rel"** it could really bring us closer to the semantic singularity…

Chapter 6

# Reading I can understand, but how do I write ?

**. . . and how the frigging hell do I do transactions with the REST hypermedia thing ?!**

---

I'd be lying if I said this is the easiest part of designing a good RESTful API. Because while applying CRUDy/Collection/Itemy patterns is a breeze (just do whatever Y. Katz and S. Klabnick tell you to see http://jsonapi.org[1]), doing slow transactions that depend on remote systems, that may have many, many states can be difficult.

Because this really requires you to think about the domain, the "big state machine" and what are the "real resources" you are manipulating.

Let's look at an example from the posterboys out at Stripe: **POST https://. . . /v1/transfers/{id}/cancel**

[1] http://jsonapi.org/

This is bad. Really bad. You have a transfer. You wish it to be cancelled. But here we are breaking the whole model. What happens if both the customer and the mechant can cancel the transfer? and both submit the POST around the same time? So much possibly breakage. We will need to write 10 pages of documentation to explain what this verb does, "cancel".

Better:

1. Just have a "status" element in the document. Yup the same transfer resource. And POST it back. Now we are very clearly asking the server to change the state of the resource from "active" to "cancelled". We can very simply use the "IF-NOT-MODIFIED" semantics. If we don't get a response from the server (remember stuff breaks) getting the **same** resource is all we need to do. This is what REST is about.
2. You can also create a "cancellation" resource. So **PUT https://. . . /v1/transfers/{id}/cancellation** this is also beautiful, because we understand we can have mul-

tiple cancellation requests from multiple parties at the same time. Hey we can even decide here we need a quorum. And its easy! something in the lines of **PUT https://. . . /v1/transfers/{id}/ cancellation_requests**

Please remember. We are acting on a single resource, but there are no requirements the resource we are modifiying will not in its turn modify other resources. So we can create a cancellation request. It will have its own state machine, which will depend on the state of the parent resource (the transfer).

By the way, Stripe already have the `status` element. So this just comes from the "let's not be ideaological about REST department". Which I totally adhere to. Except here its totally wrong. This is not easier to implement, or understand.

Please remember, a resource is something that can be adressed. Clearly a "cancellation" or a "cancellation request" is a thing. We want to have a specific log of these.

This works much much better in a distributed, asynchronous world. We still get n-phase transactions. We can poll for the result, we can ue PRG (+Long Polling), we can use a webhook.

Basically if you are implementing REST on everything **BUT** transactions I believe you are missing out on the real fun (or efficiency) of the whole idea.

A note on "DELETE". I have a big, big problem with the concept of DELETE. Well because DELETE has no semantics. There is a thing. I don't want the thing to be. Why is it not a thing?

1. the thing is not a real thing (Wrong address: Invalidate)
2. it was another thing (Duplicate address: Merge)
3. the thing is no longer a thing (Old address: Archive)

In 99% of cases you do not want to use DELETE because it will simply mean you are going to reuse the semantics of DELETE for something else. Basically you should probably "PUT" or "POST" depending on the semantics the intention you have. Say why this resource should respond with 404.

# Chapter 7

# Curious curies

**Hypermedia needs not be verbose.**

---

**FACT: Hyperlinks work very well with relative paths. I would say they even work better.**

You know, we asked you to put in “self” relation right? Now go ahead and use it. This can be your “Closure”, your scope (it has some other nifty uses), your “base_url”.

Often having verbose URLs in the responses is really not a problem. But if it is in your very specific case… Here you go Hypermedia can be terse. Call it “ID” style. Call it curies.

There are frameworks for using those (HAL from Mike Kelly, JSON-API from Steve Klabnik and Yehouda Katz). Choose either, or if you really feel a need for NIH roll out your own (please don’t).

If you are doing more then collection/item you can use something like:

`"curies": [{ "name": "ea", "href": "http://example.com/` `"templated": true }],` with real true to god templates. Note that some of the propositions are extremly feature rich (SOAP style rich) they can tell you how to compose the forms, how to write, how to manage the state machine.

Now there is a debate going on, maybe a bit of bikeshedding (“_links" againts “link”), and you might be afraid of choosing the wrong one. Well if you listened to me on the earlier points, and you proxied your internals and are using a serializer class this should not be a huge deal (you’ll also see we can get clients to break less, later).

In Hypermedia REST we decided to forgo the mountain of metadata SOAP gave us. Because the SOAP thing anyway didn’t work. We get less magic with less breakage. But we do add some back through hypermedia links. We annotate and guide.

But really for the moment this is not about clients that automagically bind to your API. This is soapy. We accept people write code. Its ok. Let's just make their lives as easy as possible. Let's make the coupling light. I imagine you are going to be delivering client libraries. So this is really ok.

**I allow you to use URL fragments. Really, I do. Go ahead. If they yell at you, tell them I said it was ok.**

Please use "HTML" style semantics for URL fragments. A local reference to an ID. You are allowed to use other semantics (look at what RDF people are doing with SKOS). But keep it simple please.

This is a yet to be resolved issue; How do I reference other stuff in the same response. We are using JSON and JSON is really light on this. There are some emergeant stuff but the dust has not settled.

# Chapter 8

# Fishy Cod & Fishy Cod on steroids

**Code is data, it is a document you can send.**

---

> 3.4.3 Client-Stateless-Server (CSS)
> The client-stateless-server style derives from client-server with the additional constraint that no session state is allowed on the server component. Each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server. Session state is kept entirely on the client.[1]

Well one of the most often misunderstood notions of HATEOAS is that a stateless server means maintaining a lot of state on the client. Which the client would usually not really know how to do without resorting to a lot of code. Because the "domain model" of the server is not exposed to the client. Which kind of breaks the whole promise. It means we are creaing very strong coupling. There is a magicky solution to this;

[1]Fielding, Roy Thomas. Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation, University of California, Irvine, 2000.

> 5.1.7 Code-On-Demand
>
> The final addition to our constraint set for REST comes from the code-on-demand style of Section 3.5.3 (Figure 5-8). REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts. This simplifies clients by reducing the number of features required to be pre-implemented. Allowing features to be downloaded after deployment improves system extensibility. However, it also reduces visibility, and thus is only an optional constraint within REST.
>
> The notion of an optional constraint may seem like an oxymoron. However, it does have a purpose in the architectural design of a system that encompasses multiple organi-

zational boundaries. It means that the architecture only gains the benefit (and suffers the disadvantages) of the optional constraints when they are known to be in effect for some realm of the overall system. For example, if all of the client software within an organization is known to support Java applets [45], then services within that organization can be constructed such that they gain the benefit of enhanced functionality via downloadable Java classes. At the same time, however, the organization's firewall may prevent the transfer of Java applets from external sources, and thus to the rest of the Web it will appear as if those clients do not support code-on-demand. An optional constraint allows us to design an architecture that supports the desired behavior in the general case, but with the understanding that it may be disabled within some contexts.[2]

## HTML is a good hypermedia format (duh!)

### and a good serialization for Humans

#### Throw in a bit of Javascript COD and you got yourself a console maybe even your whole admin backend.

Even better your web site / web app can simply be the HTML serialization of your api + some nice ember/angular bindings that calls itself getting the data with a second serialization.

so http://api.example.com is basically http:/www.example.com (why the hell do we need to have something else?) One responds to Content-Type text/html well.. Just a bit of content negotiation and the website/application/api are the same!

If you go Hypermedia all the way your website is your api is your mobile app!

---

[2]Fielding, Roy Thomas. Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation, University of California, Irvine, 2000.

# Chapter 9

# Some tools:

**Hot of the press In App Cloud by Nicolas Mérouze (he is around find him talk to him.)**

https://github.com/inappcloud/inappcloud

**You might need this one to be able to play with APIs that have custom auth schemes .. in the browser**

- https://addons.mozilla.org/en-us/firefox/addon/modify-headers/developers

**A great Chrome API console**

- http://restconsole.com/

**And a bunch of automagic clients.**

- https://github.com/codegram/hyperclient
- https://github.com/xcambar/halbert
- http://weluse.github.io/hyperagent
- https://github.com/jmarquis/angular-hateoas

Chapter 10

# Contact

Hello my name is Ori Pekelman. I am OriPekelman everywhere (twitter/github/linked-in).

My blog is on http://blog.constellationmatrix.com

Figure 10.1: constellationmatrix