The Double whammy, Leaky and the Fishy Cod

Ori Pekelman

December 5th 2013 ApiDays

Contents

Introduction 5

1 The double whammy 7

2 Leaky 9

3 Intentional 11

4 WYAIWYA 15

5 Curious curies 17

6 Lazy versioning 19

7 Fishy Cod & Fishy Cod on steroids 21

Introduction

Hello, 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 therfor 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.

This is in no way going to be a structured dissertation but more of an anecdotal review of several patterns I find useful. There are others.

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

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

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

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.

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

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

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

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.

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

Chapter 1

The double whammy

proxying your business classes

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.

One of the most important aspects of REST architecture is that it proposes a standard way of managing caching. Because caching is extremly important, useful and hard, this is not a detail. And this is why implementing REST correctly with vigilance towards verb usage as well as return codes and headers is extremly useful.

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

If your system is well behaved you can often simply put a caching reverse HTTP proxy in front of it and handle a gazillion calls. You can even fail gracefully in a much easier fashion.

Not everything will be magicky. You will need to spend time and thought designing the “hard parts to cache”, but maybe only those.

Because, usually your interal system will have shitty semantics for caching. One of the reasons for this is that you were a good developer. You optimized last. You sprinkled caching into your system only on the parts that were being called often, that were slow, you handle the safety and the management of stale data closest to where it will have effects.

This is one of the reasons your “main business code” should not talk directly HTTP. It should not be called directly from your front API controllers. You want to wrap it with a thin layer (a façade, a proxy) that handles hiding your internals and translating your semantics. This means you want to have seralizer classes. They are a bridge between your models and their representations.

But remember : there is no requirement there will be a 1 on 1 mapping between your models, or business objects and the resources you expose on the API. This is a job for this bridge. And this is where you can put for example the cache control mecanismes of your API.

A resource is an API level semantic construct. On the API controller level you should only have API level semantics. This second level of decoupling allows you to have two seperate rythmes of development one for your system the other for the API. And the API will want to live in the rythme of the Web.

This will also make it easier to decompose your API into smaller services; You might seriously want to consider having multiple endpoints. Unix style doing one small thing well. The smaller your APIs are the easier you can make them rapidly evolve. Hypermedia makes it easy to link those back again. Your root api should serve the directory of services.

Chapter 2

Leaky

Using a decorated third party API

Designing an efficient R/W API is hard.

Writing is hard, well because it is.

- It requires the client to understand the model of the server. What values are acceptable, which are obligatory, what is the format of this string, is there an order in which I should call stuff? What can I do with a resource in this specific state.
- Writing is hard because it may require you to bust caches. And caching is hard.
- Writing is hard because designing a stateless transactional system makes your head hurt.
- Writing is hard because when a read fails you get just read again, when a write fails, you need to know it did, and you need to ensure idempotence, which is, well hard.

Reading is hard for mainly three reasons:

- You want to represent the most useful common usage patterns in the simplest way possible;
- You don't really know how people will want to use your data so you want some abstract, generique, fast and incredibly feature rich interface.
- Being hit by a gazillion read requests with a lot of cache

misses can be hard. Implementing a hierarchical, partial caches is, well, hard.

A useful pattern on this is to couple your API with someone elses. I use ElasticSearch. Basically this means that you wrap ElasticSearch with a bit of your own sugar, letting it handle anything that is generic, leaving you with ample time and resources to treat only the specifics.

Basically this means my API behaves as a superset of ElasticSearch. It knows how to do whatever ES knows, and it has tricks of it's own.

We call this leaky because this is a leaky abstraction. Instead of putting a façade I am leaking techincal stuff about my search engine. When ElasticSearch changes so will my API, and it may very well do so implicitly. But the gain can be enormous, anyway on any read-only use case. I don't even need to document most of my API.

This does not contredict the earlier idea that APIs should be designed. Sometimes when you design a new car you think about how many wheels it should have. Often you go with the useful default of four.

Chapter 3

Intentional

But how the frigging hell do I do transactions with the REST 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.

[1] http://jsonapi.org/

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

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 multiple 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 4

WYAIWYA

Who you are is where you are, or content adressable

This one is tricky and easy to get wrong.

But think about how cool Git is. How cool bittorrent is. How cool Bitcoin is. These are all content adressable systems. The address of something is a hash of the content. We are used to making nice “URL Plans” this is in part a legacy of thinking about APIs as we thought about web sites. With a hierarchy. But if you remember part of the hypermedia thing is treating URLs as opaque identifiers.

The problem here, is that URLS should represent stable entities, and here, any change to the document will yield a new URL. So . . . this one should be used with parcimony. But mixing this one with the intentional pattern creates a very robust distributed thing. Because we can now even forgo the whole “IF-NOT-MODIFIED” thing, every write can reference a clear reference state.

http://api.example.org/anything/sha1-50ee373d208b4ad06 1bb415e0f31f226ec2e13c4

It can also be a very good way to link two services. It lowers the amount of information one system needs to have about the other. You can use this just as a resolution service; These URLS can redirect to the canonical URL. But. . . also, if you did implement the “self” relationship . . . well that would just work. . .

Chapter 5

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.

Both guys speaking before and after me propose to you a reasonable default. A framework 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/docs/rels/{rel}", "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 6

Lazy versioning

Version later, handle it later

Version numbers belong **in** the **document** not the **URL**. You want to change from `_links` to `links` Do you need now to increment the version in the URL? that would break all clients. Do you need to create a new content-type? Well no;

Here is a nice pattern : Always version your resources; The versioning should contain the version of the API (ver) as well as the version of the resource (rev). Use semantic versioning for your API. Clients should be versioned to the minor version of the server with minor of minor versions for themselves.

You should probably ask your clients to verify the version number. A client should probably not try to interact too much with a server that sends documents with a different major version (hey but it could try!). The client should probably log somewhere a warning on minor version mismatch. This has a lot of very useful side effects. It does not make all of your client code magically forward compatible but you can reduce a lot of breakage and we have very clear semantics around this.

This mostly means clients of all versions can interact with the same endpoint. If these are true hypermedia clients they will mostly just follow links.. and even if we upgraded our server **during a transaction** we might very well be able to treat it.

You might very well also want to combine this one with WYAIWYA! and reference the content of the previous revision we have good IANA relations for this one (either “prev” or “supercedes” depending on your use case)

Chapter 7

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]

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

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;

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 organizational 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 serialization for Humans, throw in a bit of Javascript COD and you got yourself a console.

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!

---

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