Designing Hypermedia APIs

Contents

Introduction

I have a habit that I developed a long time ago. It's served me well over the years. The habit is this:

> If I hear someone smart say something I disagree with, rather than argue, I file that away in the back of my head under "investigate later." When I'm bored, I take something off of this list and do the research.

A few years ago, "Rails doesn't actually do REST" was on that list. Then, one day, I decided to look into it. I of course, did some trivial Google-ing, and found something very similar to this. I looked at it and said, simply, "What?"

There's barely any discussion of URLs! What is this 'idempotent requests' and 'RESTful architecture' and… what? So, I decided to take a step back. What did I think REST was? Where did I get that definition from?

Well, I'd gotten them from Rails' conventions. They all made sense to me, and jived with my superficial understanding of HTTP. Okay, next step: If REST is a thing, and Rails doesn't do it, and I understand REST because of Rails, then maybe I need to check out what REST is outside of a Rails context. So, who defined REST?

This is when my journey started. Since then, I've spent the last few years researching, building, and discussing APIs. What I've done with this book is present you with the synthesis of all of the things I've discovered during this quest, and help elucidate them all.

The game plan

This book consists of two big parts, with one of those broken up into smaller ones. The two big parts are "Designing Hypermedia APIs" and "Steve's Hypermedia Notebook." The notebook is a looser, free-association style whirlwind of my notes on the topic, as well as some content that ended up morphed into the first part of this book. Think of it as a bonus.

The first part is more structured content, similar to more traditional books.

REST vs. Hypermedia

The simplest definition of 'Hypermedia APIs' that I've used is 'orthodox REST.' In this book, we'll treat "RESTful" as a code for "REST the way that Rails does it." This can occasionally cause some tension, for example, when referring to Fielding, but I'm sure you can handle it.

Affordances, or "I don't know what I'm doing"

Why are you reading this book?

I'd hope that you're reading this book because you'd like to build an API of some kind, and are looking for guidance on how. Rephrased in another way, you want to *do* something. This is the same reason that people want to use that API of yours: they want to *do something* with it. There's some sort of task at hand. Some sort of motion. Lots of verbs. In fact, I think verbs are a good way to think about hypermedia APIs: if REST is noun-oriented design, then hypermedia is verb-oriented.

So before we dive into how to go about building your API, I first want to show you a few things about what it means 'to do.' This world view is the kind of mindset that you need to be in to build a hypermedia driven design.

So I'd like to take a look at what it means to be doing some kind of thing. Before that happens, we need to introduce some important vocabulary that we can use to talk about APIs.

Some initial terms

RESTful design is often described as 'resource oriented.' This comes straight from Fielding, actually:

> The key abstraction of information in REST is a resource. Any information that can be named can be a resource: a document or

> image, a temporal service (e.g. "today's weather in Los Angeles"), a collection of other resources, a non-virtual object (e.g. a person), and so on. In other words, any concept that might be the target of an author's hypertext reference must fit within the definition of a resource. A resource is a conceptual mapping to a set of entities, not the entity that corresponds to the mapping at any particular point in time.

This brings us to our first bit of terminology, and it's an important one. A **resource** is a name for a conceptual mapping from a name to a set of entities. When I say 'mapping', you can think of a hash map, with arrays in the values:

```
{
  "foo": [1, 2, 3],
  "bar": [3, 4, 5],
}
```

Your API is the hash map itself. A 'resource' is each key-value pair, the mapping from a **name** to a set of **entities**. The names are `foo` and `bar`, and the integers are the **entities**. A **name** is just a label. No funny stuff here! An **entity** is the underlying data that's in your data store. The last bit of terminology here is the **representation**. The representation is the particular format that the set displays itself as.

What it means to be verb oriented

Whew! So what's this have to do with anything? Well, given that the resource enjoys a position as lofty as the 'key abstraction of information,' it only makes sense to feature it, right? This leads to the primary design process being around what resources you have. I've often heard this description of how to design a good API:

> Write down what you want your API to do, and then go through with a highlighter and highlight all the nouns. These nouns will become your resources.

Have you? In a resource-oriented world, you'd want to focus on the resources. It makes perfect sense. But it's the first place in which we'll diverge from what you already know. Resources will still be our primary method of information abstraction, but they won't be our primary method of designing the API.

Why are we making this split? Well, to do this, we need to look to another part of Fielding. One of the ironies of Fielding's thesis is these two sentences:

> REST is defined by four interface constraints: identification of resources; manipulation of resources through representations; self-descriptive messages; and, hypermedia as the engine of application state. These constraints will be discussed in Section 5.2.

If you read section 5.2 in full, you'll discover things about identification of resources. You'll read about manipulating those resources through representations. You'll find out about self-descriptive messages. But you won't find out anything about 'hypermedia as the engine of application state.'

For this, we need to turn to a blog post by Fielding: "REST APIs must be hypertext-driven". In it, Fielding re-emphasises the importance of hypermedia:

> What needs to be done to make the REST architectural style clear on the notion that hypertext is a constraint? In other words, if the engine of application state (and hence the API) is not being driven by hypertext, then it cannot be RESTful and cannot be a REST API. Period. Is there some broken manual somewhere that needs to be fixed?

Yes, there is: 5.2 of the thesis! Luckily, Dr. Fielding expands on what he means:

> A REST API should spend almost all of its descriptive effort in defining the media type(s) used for representing resources and driving application state, or in defining extended relation names and/or hypertext-enabled mark-up for existing standard media types. Any effort spent describing what methods to use on what URIs of interest should be entirely defined within the scope of the processing rules for a media type (and, in most cases, already defined by existing media types). [Failure here implies that out-of-band information is driving interaction instead of hypertext.]

There is a whole lot of greatness packed into this one paragraph, but it gives us the key we need to understand how to design our APIs. That first sentence says it all: almost all of our descriptive effort, ie, our design, is spent in defining the media type used for representing resources and driving application state.

Let's talk about 'driving application state' for a moment. What does 'driving' mean here? This expression calls forth an image of a car. A person who is driving a car is causing the car to be put into motion, and controlling the direction and intensity of that motion. The car is the current state of the application, so to drive that state implies that we want to change it somehow, and we want to be in control of how it changes. We interface with our application through changing the state of these resources, which contain representations of our application's entities.

Repeated a slightly simpler way: our task as an API designer is to figure out a good way to represent the state of our application and then give people tools to modify the state in some way.

The 'resource-first' school of design focuses heavily on the state itself, going so far as to turn the verbs of our API into nouns instead! The issue here is once again well-intentioned, but a mis-reading of the primary information. You see, it *is* true in many ways that the hypermedia style constrains verbs. The misunderstanding here is conflating two different levels of our network stack with each other: the application and the protocol layer.

One of the big seven constraints of REST is the 'uniform interface constraint,' defined in Section 5.1.5 of the dissertation. This section discusses how at the *protocol level*, all APIs must have identical interfaces. This is the reason that HTTP verbs exist, and that there are really only a few of them. Have you ever said something like this: "Ugh, this action doesn't map nicely to GET or POST or PATCH. I wish I could make my own verbs, like COMMIT and LOGIN" ? Fielding knows:

> By applying the software engineering principle of generality to the component interface, the overall system architecture is simplified and the visibility of interactions is improved. Implementations are decoupled from the services they provide, which encourages independent evolvability. The trade-off, though, is that a uniform interface degrades efficiency, since information is transferred in a standardized form rather than one which is specific to an application's needs.

Bummer, right? When mapping a service onto a RESTful protocol like HTTP, you won't always get the verbs you want. This has a benefit, though: protocol verbs are global to all applications. If you're building a website built around version control, you might initially add a `COMMIT` verb. But then someone who's writing a database application would get upset, because 'commit' means something very different in their domain. So now we need a process to discriminate and resolve conflict between names. This bureaucracy would slow the rate of change way, way down. This is what Fielding means by 'independent evolveability': my application can change and not affect your application.

However, this is all for the protocol: within the scope of the protocol, we do need to do a variety of things. We can't be super generic, we need specific names. But we also can't have conflicts. So what do we do?

The answer is that an action in your application is a combination of two things: an action in the protocol, and a name in a namespace that you control. That's it. With APIs on the web, that means "HTTP verb + URL." This is why the documentation around RESTful APIs focuses so much on these two things: "POST /posts : makes a new Post resource." One of the neatest things about URLs that nobody appreciates is that they're basically a global namespace for

resolving name conflicts. I can make whatever names I want in my namespace, and you don't need to worry about it.

Anyway, what this all boils down to: users of our API wish to do something with it to accomplish some kind of task. A resource centric design says "Okay: I want to do a thing. First, what are the nouns, and then from there, we'll try to add on some verbs. Possibly by turning them into nouns." Hypermedia-centric design acknowledges that the state transitions are just as important as the state itself, and possibly even more important. Therefore, we say "Okay: I want to do a thing. What is the verb I want, the nouns will fall out of that."

If I was feeling sarcastic, I might try to turn this into VDD: Verb Driven Design! I can just feel the blog posts and conference talks and books.

Ahem.

This is what you need to do: you need to think about doing, don't think about being. The combination of Object Oriented Programming (which often features its own excessive attraction to nouns over verbs) and resource-oriented design have probably led you to have good expertise with 'being driven design.' So let's talk about what it even means to *do things*.

What it means to 'do'

Let's have a little thought experiment here: imagine that it's your birthday. I've given you a gift card to that swanky new resturant downtown, everyone should eat a good meal on their birthday. You're not sure where it is, so you ask, and I tell you, "It's on the corner of 2nd and 3rd." You show up at that corner, but there are four: one resturant on each corner. How do you know which one is which? Well, there's probably a sign outside, so you check it out, and enter the right resturant. You get a table, and since you haven't been here before, you need to know what to eat. So you read the menu, and place your order.

Easy, eh? This is the kind of stuff we do all the time. The stuff we *do*. Note that almost everything in this story is centered around the action that you're taking. There's some description in there too, of course, but our story is driven forward by verbs.

Futhermore, you were in an unfamilliar environment. How'd you know where to go, what to do? Well, you're (probably) an adult, you've been in the world. You know that things like signs exist, and that you can ask people for directions. In other words, you're aware that your environment will provide certain signals about what you need to do. Often in our lives, these signals are implicit, but sometimes, like when we're driving, they're very explicit. Don't walk. Stop here. Don't go faster than 75. And so on, and on, and on.

Designers and others use the term 'affordance' to describe this phenomena. An affordance is some sort of signal provided by the environment or some object in

it. This signal lets you know that you can perform some kind of action with the object or in the environment.

Affordances will be central to our discussion moving forward. If you want to be good at designing hypermedia APIs, I suggest praticing paying attention to various affordances in your natural environment. Give concious thought to the signs you see, the way that you relate to objects in your environment. These interactions can often give clues to useful affordances you can use yourself.

Here are a few of my favorite affordances that you may recognize:

Signs on doors that say 'push' and 'pull.' What I love about these signs is that they're almost universally unnecessary if you get the opener affordance correct. Have you ever seen a sign on a door that says 'pull' and you push anyway, because the door looks like it needs to be pushed? I do, all the time. The handle or bar or knob on the door is its own affordance, and it communicates a certain kind of interaction. The sign is an explicit attempt to override that affordance with a different one. The sign really should say "I know you think this door needs to be pulled, but that's wrong! Please push instead, trust me."

Street signs are an explict affordances, and one of the most pervasive ones that we have in our environments.

Big, giant, red buttons with glass cases over top. These suggest that you can push one and an incredibly powerful state change will occur.

That's just a few. Keep a look out for them! They're everywhere.

Here's another interesting things about affordances: they only indicate the realm of possibility, they don't require actual understanding. For example, imagine that someone is born with a rare disease whereby they cannot understand intutitively how doors work. That person wouldn't die when they saw a door. They could exist in our world, their options are just limited. For a less ridiculous example, when you travel to a foreign country, you won't be able to understand signs that are written in a language that you don't currently speak.

However, that doesn't mean the signs aren't useful. For example, let's say that you're in Tokyo. You want to go visit Ginza, so you ask the internet how to get there. It tells you that you'll want to board the G train at Ueno Station, and get off at Ginza Station. What do you do? Well, you look up translations, right? So you know that Ueno Station will have a sign that says "uenoeki", and that Ginza Station will have the sign "ginzaeki". Even though you don't understand that "eki" is "station," you can pattern match the two names and find your way.

This is a pretty accurate description of how a computer interacts with a hypermedia API. It has a list of things that it understands, and even though it doesn't have a deep semantic understanding of what the symbols it is processing mean. This also illustrates that you don't need strong artificial intellegence to navigate a hypermedia API.

It also implies one other thing: if you don't know the meaning of something, you can just ignore it. Something bad might happen to you, but then you can handle that bad event. Not knowing how to proceed in a certain way doesn't prohibit you from proceeding in a totally different way. If you've ever said the words "Oh, so *that* is what that meant!", you are well aware that full semantic understanding of affordances isn't strictly neccesary to keep the ball rolling. When we get to building clients, this rule will be incredibly important.

The role of affordances in API design

A noun-orented design doesn't allow for affordances because affordances are verbs, not nouns. A good way to think about hypermedia APIs is 'including affordances to indicate what you can do next.' In other words, it's using hypermedia as the means of driving application state forward. Is that all coming together for you now?

This is why the only actions a RESTful API can take are the protocol-level ones. They're the only verbs that exist. In order to verb, you need to CRUD up some new noun. There's an old REST joke: the answer to almost every question is "make another resource." In order to verb, you first must noun-ify it. Here's an example: let's say that we want to process a deposit on a bank account. In a resource-centric design, we'd make two resources: An "account" resource, and a "new deposit" resource. You might imagine documentation like this:

```
GET  /account           provide account details
POST /account/deposit   desposit money into your account
```

Most people would call this a 'sub-resource,' but from the orthodox Fielding perspective, the URL details don't matter. It could as easily be

```
GET  /birthday          provide account details
POST /d3adb33f/yup/cool desposit money into your account
```

Doesn't matter. Each one is distinct. Now, as an affordance for humans, I wouldn't suggest #2, ever. But for our purposes, they're two distinct resources.

Anyway, this documentation provides the description of how to actually do the task. Anyone who wanats to know how to do this action needs the documentation in order to know how to accomplish the task. In a hypermedia design, we'd include the affordance inside of the response:

```
{
  "balance": 100,
  "_links": {
```

```
    "http://mysite.com/rels/deposit": {"href": "/account/deposit"},
  },
}
```

This is a response using HAL, one hypermedia-enabled media type. HAL includes a `_links` key in with the data to allow for you to include affordances. You don't need the documentation to know that an account allows for deposits, it's right there in the response itself. Now, when I say it's there, I mean that there are two things:

1. Affordances that indicate possible actions.
2. A way to figure out more information about those affordances if you don't know already.

Of course, in order to know that `http://mysite.com/rels/deposit` means "you can deposit money here" takes some degree of human interpretation. I'm not suggesting strong artificial intellegence here. What I am saying is that we can move the possibility of action from the realm of humans reading documentation to the realm of a computer understanding what's up.

What's also interesting is that this affordance comes at little cost. Imagine a client that doesn't understand any affordances; it is strictly pre-programmed to do certain tasks. Like, say, the RESTful API client we mentioned earlier. What kind of information do you think `GET /account` would return?

```
{
  "balance": 100,
}
```

Yup, the same exact thing, just without affordances. This client can still get its job done just fine: it's effectively memorized what tasks it needs to do and in what order. The downside is that if things ever change, a client that understands the concept of affordances can adjust what it's doing to take advantage of these changes. Clients that don't grok affordances could stop working.

Even if a client does understand affordances, that doesn't mean it understands all affordances. For example, the 'full response' of the API above might look more like this:

```
{
  "balance": 100,
  "holder": "Steve Klabnik",
  "_links": {
    "http://mysite.com/rels/deposit": {"href": "/account/deposit"},
     "http://mysite.com/rels/close":   {"href": "/account/close"},
  },
}
```

There's more data and affordances there we didn't even see before! If we were building an application that just let you deposit money into a given account, we might not care about the holder name or the ability to close the account. To a simple client like that, the above response is effectively identical to the one above with only one affordance. It can't properly percieve the extra affordance and data, and that's okay. It knows just enough of the ones to get the job done.

Affordances and Algorithms

Here's another interesting angle on this particular topic: Algorithms are a series of steps. Algorithms are made of verbs, not nouns. Algorithms are a series of state changes in your application's state. A slightly different way of thinking about hypermedia APIs is the usage of affordances to communicate an algorithm to a client.

For example, here's a simple algorithm for ordering a pizza from a website:

1. Get a copy of the menu to see what kinds of toppings you can get.
2. Pick a size of pizza and some toppings, and add them to your order.
3. Once you've added all the pizzas, choose to finalize the transaction.
4. In order to finalize it, your credit card details will be asked for.
5. Wait a while in order to verify that your payment is accepted.
6. A recepit

As an exercise, try to list out the affordances and resources that you'd need to build out this API. Remember that resources are usually nouns or some grouping of nouns, and that affordances are signs of some action that you'd take.

Done? Here's mine:

Affordances: get a menu, add something to an order, finalize the order, supply credit card details, 'please wait,' and get a receipt.

Resources: a menu, a menu item, an order, a receipt.

These are the building blocks that we'd need to build a pizza-ordering algorithm. Any client that can understand these affordances and resources should be able to successfully order a pizza from every environment that has those affordances and resources. Neat, eh?

This is the first step in hypermedia API design, congratulations! The intial act of design involves properly choosing a set of needed affordances and resources. The simplest way to do this is to describe the actual business process you're trying to accomplish, and then paying attention to the nouns and verbs.

Conclusion

In fact, I'm a big fan of the pizza example. It might be because it's a domain that I know well, I worked at a pizza shop for seven years before becoming a professional programmer. And almost everyone has ordered a pizza at some point in their life. For the rest of this book, we'll be using pizza ordering as our motivating example of a hypermedia API. If you hate pizza, it will work well for ordering almost anything. Just substitute some other food or good.

In this chapter, you mastered the concept of affordances, and learned about the differences between resources, their representations, and entities in a system. In the next chapter, we'll get into the nitty-gritty of fully designing and implementing our pizza ordering API. After that, we'll talk about refining it, adding some things, improving our implementation, and discussing common patterns that come up when building APIs in this manner.

Hyperpizza

COMING SOON

This is the example for the rest of the book; a pizza store. Online.

What affordances do we need?

How do we provide them?

Sample code for placing an order, client and server

Media Types

COMING SOON

Case study: JSON API

Caching

Coming SOON

What do people mean by 'efficient apis'?

Caching is the key to 'efficiency'.

Using HTTP caching correctly.

You probably haven't used a client that caches things.

Caching is hard.

Polling is awesome

COMING SOON

What is polling?

Why do people dislike polling?

Why is polling actually awesome?

Creativity when it comes to details.

Reusable tools

COMING SOON

One of the big advantages is re-usable tooling.

This happens with standardized formats.

Users and auth

COMING SOON

How do I do authentication?

How do I do authorization?

Signing up via an API?

Versioning

COMING SOON

You don't need it.

It's about managing change.

It requires a differnet thought process.

This is next-level stuff, you can punt and version if you want to.

Should I build a hypermedia API?

Coming Soon.

When should I build one?

When shouldn't i?

What are the risks?

What are the rewards?

Educating Others

COMING SOON

Basically, we have a long way to go to explain all this to others.

You can't expect them all to read my book before using your API.

Strategies for communicating with users.

Notebook

Everything that follows from this point is part of the 'hypermedia notebook.' Basically, it's a short (~80 pages) collection of essays that I originally wrote when sorting out exactly how I wanted to write this book.

You should consider all this stuff to be a fixed creation of its time and place. For example, while I still think there's a lot of insight in "Transmuting machinery into reality," the W3CLove API no longer exists, and I've refined the design process slightly. That doesn't mean there's not a lot of value! Just that this stuff isn't necessarily 100% cannon. It's 99% so. But it's immutable. I won't be revising it further. It's largely stream of conciousness.

Enjoy.

Hypermedia API

Hypermedia APIs are APIs that consist of two things:

1. Usage of HTTP to its fullest.
2. Responses are served as hypermedia that manages application state.

Super simple.

The API formerly known as REST

Hypermedia APIs happen to coincide with Roy Fielding's definition of REST. However, most RESTful APIs do not follow Fielding's definitions, and so hypermedia devotees have started using new nomenclature.

Hypermedia Affordance

The word 'affordance' was coined by James Gibson:

> The affordances of the environment are what it offers … what it provides or furnishes, either for good or ill. The verb 'to afford' is found in the dictionary, but the noun 'affordance' is not. I have made it up (page 126).
>
> James Gibson, "The Ecological Approach to Visual Perception"

Fielding has also used it:

> When I say Hypertext, I mean the simultaneous presentation of information and controls such that the information becomes the affordance through which the user obtains choices and selects actions
>
> Roy Fielding, "A little REST and Relaxation"

Therefore, a "hypermedia affordance" is one of these controls.

I was introduced to this term by Mike Amundsen. He's done some work on this, first around "H factors," and more recently, a simplified set of hypermedia affordances. Let's talk about this simplified set:

> i claim there is a stable set of aspects that can be found expressed in every affordance. and i claim that the set has only four members.
>
> so here goes; my list of universal hypermedia affordance aspects:
>
> - Safety: the affordance offers either a safe action (HTML.A) or an unsafe action (ATOM.LINK@rel="edit").
> - Idempotence: the affordance represents either an idempotent action (HTML.FORM@method="get") or a non-idempotent action (HTML.FORM@method="post").
> - Mutability: the affordance is meant to support modification (mutable) by the client (HTML.FORM) or the affrodance is immutable (HTML.LINK).

- Presentation: the result of activating the affordance should either be treated as a navigation (HTML.A) or as a transclusion (HTML.IMG).

Mike Amundsen, "hypermedia affordances"

When writing HTML, you evaluate all of these concerns without thinking about it. But when designing a new type, or evaluating a type that may be appropriate for your application, you have to think about such things more explicitly.

For example, if you want to have the user submit information to a page, you'd say, "Duh! HTML <form> will do the trick!" But *why* do you say that? It's because you know you want an non-idempotent, mutable control. So you use a form with POST. Implementing a search? You'd probably use a form with GET. Why? Because you're trying to template a URI.

When building a new media type, you need to consider what affordances your business process requires, and then make sure your type provides those type of affordances. It sounds obvious when you say it out loud! It's an essential part of media type design, however.

Sources Cited

- "hypermedia affordances", Mike Amundsen
- "The Ecological Approach to Visual Perception"
- "A little REST and relaxation", Roy Fielding
- "H Factors", Mike Amundsen

REST

REST is a term that is derived from Roy Fielding's dissertation. It describes a particular archetectural style used to develop "distributed hypermedia systems."

Eventually, it became basically synonymous with a different kind of API: one that uses HTTP more properly than many previous styles. RESTful APIs became a buzzword of sorts, and lost much of the meaning that Fielding intended.

REST vs. Hypermedia

Eventually, people that cared about 'real REST' stopped using the term RESTful to describe their APIs, feeling that it had come to mean something completely different.

Given that the most important aspect that RESTful devotees had ignored was the hypermedia constraint, and given that it's really the focal point of a true REST API, the term "hypermedia API" started to gain steam. This project itself is part of this tradition.

Recognizing a RESTful API

You can tell that you've got a RESTful API by the documentation: if the docs look something like this:

POST /articles

This request creates a new article. Send it JSON with the following keys: - title: A title for the post - content: the body of the post -tags: An array of strings. These should contain tag names.

GET /articles/comments

This request gets a list of comments. You'll receive a response that looks like this:

{"id":1, "title":"My post", "content":"This is the text body.", "tags":["hello", "world"]}

DELETE /articles/:id

This request will delete the given article.

You know you're dealing with a RESTful API. Note that it looks like a bunch of function calls that combine a URI with an HTTP verb, as well as explaining the details of each and every response.

Sources Cited

- "Architectural Styles and the Design of Network-based Software Architectures", Chapter 5
- REST is OVER!

Web Worship

If you're listening to someone give a talk, and they really know their stuff regarding hypermedia, I'm willing to bet you a dollar that you'll hear them say something like this at some point:

> And then, your client will be able to automatically understand new business processes, just like a web browser!
>
> If you think about this change for a while, *gasp*, this works just like a web browser!

What's up with all of this worship of the web and browsers? Why does everyone who's building an API think that every single client needs to be a browser? If I wanted to make something display in a browser, I'd just make a website, don't you think?

It turns out, there's some good reasons.

First of all, Hypermedia API dogma (to keep the religion analogy going) is fundamentally descriptive, not prescriptive. It was formalized around explaining the properties of a system that already existed: the world wide web. Therefore, it only makes sense that the ideal client in such a system is the web browser. It is literally the ideal, the sculpture from which the mold was created. Therefore, it only makes sense that every hypermedia client should attempt to work 'like a web browser'.

That's almost a tautology, in a certain sense. Sure, a web browser is an ideal web client, but why is the web itself special?

To put it simply, the world wide web is the largest, most successful computer platform that has ever been created. When I say the words 'web scale' you may laugh, but think about it for a second: Google, the largest website on the web by traffic, accounts for only 6.4% of web traffic in 2010. Six percent. The web, as a project, needs to scale at a full two orders of magnitude larger than the largest single organization that builds things on its platform.

In order to reach this massive scale, the web *must* be a distributed network. A centralized network couldn't possibly manage the traffic correctly. In order to be distributed, the web *must* be weakly consistent. It *must* be massively decoupled. It *must* be as dynamic as possible. Without all of these things, the web would simply collapse under its own weight.

Google itself is large enough to demonstrate that an incredibly restrictive environment is necessary to handle traffic at its scale. While C++, Java, and Python are the three blessed Google languages, lately, Google has been discouraging Python use because it's simply not able to scale well enough:

> Well, simple common sense is going to limit Python's applicability when operating at Google's scale: it's not as fast as Java or C++, threading sucks, memory usage is higher, etc. One of the design constraints we face when designing any new system is, "what happens when the load goes up by 10x or 100x? What happens if the whole planet thinks your new service is awesome?" Any technology that makes satisfying that constraint harder – and I think Python

> falls into this category – *should* be discouraged if it doesn't have a very strong case made in its favor on other merits. You have to balance Python's strengths with its weaknesses: your engineers may be more productive using Python, but if they have to work around more platform-level performance/scaling limitations as volume increases, do you come out ahead?

This isn't to trash Python, specifically, but just to demonstrate that basically, at web scale, you're left with C++ and Java to develop with. No other options work. That's kinda crazy. It is difficult to truly appreciate the scale that the web operates on.

Anything that requires explicit coordination affects scale, too. Can you imagine the amount of pain and effort it would take if the WWW was versioned? "Hey everyone, update your browser, V1 of the web is going down." We can't even get companies to get rid of IE6. What if there was no uniform interface? Download a new web browser update whenever one of your web sites added some new functionality? It'd be a nightmare.

So if it's good enough for the web, it's good enough for your tiny application that nobody (relatively speaking) cares about. Yet. ;)

The Social

There's another parallel that makes the Web worthy of adulation: the Web embodies a certain kind of democratized mode of production that social theorists have aspired to bring about for centuries.

> The web is more a social creation than a technical one. I designed it for a social effect — to help people work together — and not as a technical toy. The ultimate goal of the Web is to support and improve our weblike existence in the world. We clump into families, associations, and companies. We develop trust across the miles and distrust around the corner.
>
> Tim Berners-Lee, "Weaving the Web"

In a way, human relations form a web. The rise of Facebook has certainly demonstrated this, and they explicitly call it out in the URI and name of their latest API. We are connected in an incredibly complicated web of social relations. This is mirrored in the construction of the web itself with its myriad links between entities.

> There can be no revolutionary actions where the relations between people and groups are the relations of exclusion and segregation.
>
> Michel Foucault, "Anti-Oedipus," Preface

On the web, we are all equals. That's what's truly revolutionary. Anyone can publish, anyone can link, anyone can carve out their own little corner of the web.

Sources Cited

- "Google accounts for 6.4 percent of internet traffic"
- Alexa web rankings
- "Recommendation against Python?"
- "Weaving the Web", Tim Berners-Lee

Hypermedia Benefits in Plain Language

Most of this project explains the 'how' of Hypermedia APIs. This particular node is about the 'why'.

Historically, we have failed

The number one thing that hypermedia advocates have failed to do in the past is properly explain the advantages of hypermedia-based designs in simple, plain language.

Most social organizations take some cues from the early leaders of the organization, and hypermedia enthusiasts are no different. Roy Fielding himself basically washed his hands of explaining the details to lay people:

> As you may have noted, my last post seems to have hit a nerve in various communities, particularly with those who are convinced that REST means HTTP (because, well, that's what they think it means) and that any attempt by me to describe REST with precision is just another elitist philosophical effort that won't apply to those practical web developers who are just trying to get their javascript to work on more than one browser.
>
> Apparently, I use words with too many syllables when comparing design trade-offs for network-based applications. I use too many general concepts, like hypertext, to describe REST instead of sticking to a concrete example, like HTML. I am supposed to tell them what they need to do, not how to think of the problem space. A few people even complained that my dissertation is too hard to read. Imagine that!

> Others will try to decipher what I have written in ways that are more direct or applicable to some practical concern of today. I probably won't, because I am too busy grappling with the next topic, preparing for a conference, writing another standard, traveling to some distant place, or just doing the little things that let me feel I have I earned my paycheck. I am in a permanent state of not enough time. Fortunately, there are more than enough people who are specialist enough to understand what I have written (even when they disagree with it) and care enough about the subject to explain it to others in more concrete terms, provide consulting if you really need it, or just hang out and metablog. That's a good thing, because it helps refine my knowledge of the field as well.

If you haven't noticed, I'm in that category. And more recently, there have been some people doing great work in this area. But it's been twelve years since Fielding's thesis has been published. It's taken us this long to do this kind of work. And I feel that's a shame. Hence, this project itself.

While this whole work falls under this category, this node in particular is about staying away from jargony terms. At least hypermedia API jargony terms. This justification is also longer than the content itself. I want to keep it direct and on target.

The benefits

The simplest explanation

Here's the simplest I can possibly make it:

Hypermedia designs promote scalability, allow resilience towards future changes, and promote decoupling and encapsulation, with all the benefits those things bring. On the downside, it is not necessarily the most latency-tolerant design, and caches can get stale if you're not careful. It may not be as efficient on an individual request level as other designs.

"Sound bite" explanation

Here's a list of slightly more complicated explanations. These start to contain a bit of jargon, but it's hard not to introduce it at some level.

- Scalability through client/server separation and self-contained messages
- Clients and servers can evolve independently of one another
- Visibility: all requests are self-contained, so you can inspect a full request on its own

- Improved efficiency through caching
- Improved user-perceived performance by not hitting the network through caching
- Reduced average latency through caching
- A uniform interface is very general, with all the benefits that brings
- A uniform interface decouples implementations from services given, promoting evolvability
- Layering provides encapsulation
- Intermediaries can help with scalability, via load balancers and caches
- Code on demand improves extensibility

And disadvantages that go along with it:

- Increased network traffic by plain-text messages rather than binary ones and (possibly) chatty interaction patterns
- Server has to trust the client to manage certain application behavior properly
- Stale cache data can decrease reliability
- A uniform interface harms efficiency, since information is in a general format rather than a specific one
- Layering adds latency
- Code on demand reduces visibility and adds coupling to a language implementation

Super-technical explanation

These all deserve their own full nodes. I'd like to write them, but they have yet to be written. Beta! :)

Sources Cited

- "Architectural Styles and the Design of Network-based Software Architectures: Section 5.1", Roy Fielding
- "Specialization", Roy Fielding

Linking

> A link is a connection from one Web resource to another. Although a simple concept, the link has been one of the primary forces driving the success of the Web.

This description comes from the HTML 4 spec. It further elaborates:

> A link has two ends – called anchors – and a direction. The link starts at the "source" anchor and points to the "destination" anchor, which may be any Web resource (e.g., an image, a video clip, a sound bite, a program, an HTML document, an element within an HTML document, etc.).

A link is what transforms media into hypermedia. Links are what make the web a web; without links, you have a loose collection of text files. Links provide relationships between resources, and provide the transitions for the state machines you use to model your business processes.

RFC5988 defines links in a slightly different way than the HTML4 spec:

> In this specification, a link is a typed connection between two resources that are identified by Internationalised Resource Identifiers (IRIs) [RFC3987], and is comprised of:
>
> - A context IRI,
> - a link relation type (Section 4),
> - a target IRI, and
> - optionally, target attributes.
>
> A link can be viewed as a statement of the form "{context IRI} has a {relation type} resource at {target IRI}, which has {target attributes}".
>
> Note that in the common case, the context IRI will also be a URI [RFC3986], because many protocols (such as HTTP) do not support dereferencing IRIs. Likewise, the target IRI will be converted to a URI (see [RFC3987], Section 3.1) in serialisations that do not support IRIs (e.g., the Link header).

This introduction of the 'relation type' is important, as it adds semantic information as to what the link is all about, and is used in hypermedia APIs to determine which link to follow next.

Tim Berners-Lee thinks links are incredibly important:

> In an extreme view, the world can be seen as only connections, nothing else. We think of a dictionary as the repository of meaning, but it defines words only in terms of other words. I liked the idea that a piece of information is really defined only by what it's related to, and how it's related. There really is little else to meaning. The structure is everything. There are billions of neurons in our brains, but what are neurons? Just cells. The brain has no knowledge until connections are made between neurons. All that we know, all that we are, comes from the way our neurons are connected.
>
> Tim Berners-Lee, "Weaving the Web", p14

Sources Cited

- RFC5988: Web Linking
- HTML 4.0.1: Links
- "Weaving the Web", Tim Berners-Lee

Hypertext

Hypertext is a word that consists of two parts: 'hyper' and 'text'. 'Text' refers to a work that's human-readable: plain text. The 'hyper' part is where it gets interesting. The prefix hyper- comes from the Greek " -", which means "over" or "beyond." "Hypertext," then, is text that's 'beyond text.'

Hypertext was coined by Theodor H. Nelson in the 1960s. Here's a quote from his book "Literary Machines":

> I mean non-sequential writing - text that branches and allows choices to the reader, best read at an interactive screen. As popularly conceived, this is a series of text chunks connected by links which offer the reader different pathways.

It's this linking that makes the hypertext 'beyond text.'

Hypertext has a sister term: "hypermedia." This term generalizes 'beyond text' to 'beyond media,' allowing non-textual digital objects to be linked together. Therefore, all hypertext is hypermedia.

In general, I'll be discussing 'hypermedia' in this project rather than hypertext, since we're concerned about media in general and not just text. In George P. Landow's seminal work "Hypertext 3.0," he makes a similar generalization:

> *Hypermedia* simply extends the notion of the text in hypertext by including visual information, sound, animation, and other forms of data. Since hypertext, which links one passage of verbal discourse to images, maps, diagrams, and sound as easily as to another verbal passage, expands the notion of text beyond the solely verbal, I do not distinguish between hypertext and hypermedia. *Hypertext* denotes an information medium that links verbal and non-verbal information. In this network, I shall use the terms *hypermedia* and *hypertext* interchangeably.

Hypermedia seems really simple, but it's ultimately something that's incredibly special. Obviously, when writing a book with the word in the title, you give it a lot of consideration, but the simple act of adding links to text has massive power. I write more about this in 'Web Worship'.

Sources Cited

- "Hypertext," Wikipedia
- "Literary Machines", Nelson
- "Hypertext 3.0", Landow

Programming the media type

Mike Amundsen, author of "Building Hypermedia APIs with HTML5 and Node", first introduced me to this concept. In a discussion with him, he once mentioned to me that while building hypermedia clients, he often felt like he wasn't programming for the server, he was 'programming the media type'. This idea stuck in my head, mostly because I wasn't quite sure what he meant.

Then I read this quote from Roy Fielding:

> A REST API should spend almost all of its descriptive effort in defining the media type(s) used for representing resources and driving application state, or in defining extended relation names and/or hypertext-enabled mark-up for existing standard media types. Any effort spent describing what methods to use on what URIs of interest should be entirely defined within the scope of the processing rules for a media type (and, in most cases, already defined by existing media types). [Failure here implies that out-of-band information is driving interaction instead of hypertext.]
>
> Roy Fielding, "REST APIs Must be Hypertext Driven"

And one from Jan Algermissen:

> REST limits the possible contract changes to just two kinds: adding data or controls and removing data or controls from a message
>
> Jan Algermissen

Clients have guidelines too

After reading these quotes, it started to fall into place. Similar to the five rules of design that I've created for servers, clients *also* have a set of design guidelines. While I haven't quite yet formalized my personal rules for clients yet, I do know the first one: program your client for the media type, not for the specific responses involved.

Let's break down what Fielding has to say, step by step.

Most of the documentation is of media types

> A REST API should spend almost all of its descriptive effort in defining the media type(s) used

Essentially, this is the only information that's published about a particular API: what media types it expects. That's a radical departure from what one is used to. However, it's essentially accurate. And, in fact, if you recall the canonical example of the web browser, you'll realize that the only thing it knows about is HTML, CSS, and a few other media types.

To build an effective client, you must *only* rely in the media types served and their definition.

Let's read further:

> for representing resources and driving application state, or in defining extended relation names and/or hypertext-enabled mark-up for existing standard media types.

This is more about what media types should define than programming clients, but it's still useful. The phrase 'for existing standard media types' signifies something interesting, though. For more on that,

Required methods belong in the media type

> Any effort spent describing what methods to use on what URIs of interest should be entirely defined within the scope of the processing rules for a media type (and, in most cases, already defined by existing media types).

As always, naming specific URIs is a form of coupling, but this is talking about which methods to be using on which URIs. If you require the use of certain methods, it needs to be defined in the media type itself, and not outside of it. For example, HTML defines an attribute of the <form> element, "method", which allows for either GET or POST requests to be made. ATOM, on the other hand, defines that a link with the attribute "rel" set to "edit" should have PUT requests sent to it. These are two different options: define an attribute with the method name or define that a certain control only respects one verb. The important part is that both are defined within the media type itself and not outside.

Consequences of failure

> [Failure here implies that out-of-band information is driving interaction instead of hypertext.]

Here's the *why* to all of this. Basically, the media type is the only sort of contract between the client and the server. Some people have a hard time with this idea of 'out-of-band'. This property is part of the "self-descriptive messages" constraint, which is part of the "uniform interface" constraint. I'll explain what 'out-of-band' means in the context of an example:

Let's imagine that we have our data and we'd like to render it for a user. There are two different components in this interaction: a renderer and the data. There's one kind of output: the rendering. Now, if we're operating in a client-server manner, we have three options:

1. Have the server render the data and then send the rendering to the client. In this case, the client needs to do very little work, but the server has to do a lot. As the server handles more and more clients, this may become a scalability problem.
2. Send the data and a renderer to the client, and let the client mix them together to form the rendering. This option is unattractive because it vastly increases network traffic. It also limits you to certain kinds of clients since you'll be passing around executable code and the client has to support that particular runtime environment.

3. Send the data to the client, allow them to choose a rendering engine that they already possess, and require the client to handle the rendering. This lets the server remain simple and scalable, keeps network traffic down, and gives the client choices, which translates to giving the user choices, which is good.

Okay, so we'll choose option three. Now we need to allow the client to choose an appropriate rendering engine based on the data. So we tell the client what kind of data it is: "Hey, this is some text/html. This is some image/png." This is what we mean by a self-descriptive message: it contains both the data that we need to render as well as information about what the data is so that the client can choose a rendering engine.

We wouldn't want to go outside of the boundaries of the type that we tell the client, because then, it wouldn't be able to understand what we were saying. There's no mechanism to say "This is text/html, but hey, we use this <blah> tag that I made up too." That information would therefore be out-of-band because we haven't explicitly told the client about it.

Now, it's true that we do have to have some kind of communication outside of the protocol itself. We say "This is text/html", but we don't say what text/html *is*. In order to do that, we'd have to operate in the context of choice 2, and we've already discussed the downsides to that. If we made that architectural choice, we wouldn't need to provide any documentation at all, as our services would be 100% discoverable. This is what a WSDL provides, and so now hopefully you can see how significantly different of an architectural design this is.

For more on this, see Section 5.2.1 of Fielding's dissertation.

Summary

The exact description of a media type is the only information that we don't send along to the client, and therefore, it's the only thing we should describe in documentation. This also means that clients need to be prepared to handle any response that's a valid form of the media type and don't need to know specifics outside of that. Therefore, when you build a client, you're really 'programming a media type'.

Sources Cited

- "REST APIs Must be Hypertext Driven", Roy Fielding
- "Understanding the role of media types in RESTful applications", Jan Algermissen
- "Architectural Styles and the Design of Network-based Software Architectures, Section 5.2.1" Roy Fielding

The Design Process: An Overview

It turns out that designing Hypermedia APIs isn't actually that complicated. It is *different* than your usual kind of design, though. It consists of six steps:

1. Evaluate processes
2. Create state machine
3. Evaluate media types
4. Create or choose media types
5. Implementation!
6. Refinements

Note that there's nothing about URLs, nothing about status codes. It's all about state machines and media types. Since we have HTTP, we don't need to redo the work it does adhering to most of Fielding's architectural constraints. This leaves most of the challenge in designing the actual hypermedia types that your application uses. In order to do that, you have to understand what hypermedia affordances your system will need. Your business process state machine will inform you of these needs.

Fielding also supports this:

> A REST API should spend almost all of its descriptive effort in defining the media type(s) used for representing resources and driving application state, or in defining extended relation names and/or hypertext-enabled mark-up for existing standard media types. Any effort spent describing what methods to use on what URIs of interest should be entirely defined within the scope of the processing rules for a media type (and, in most cases, already defined by existing media types). [Failure here implies that out-of-band information is driving interaction instead of hypertext.]
>
> Roy Fielding, "REST APIs Must be Hypertext Driven"

Anyway, let's dig into the process!

Step 1: Evaluate processes

The first thing you need to do is figure out what your API needs to do.

This sounds really obvious, but it's really the first step. The key here is to think in terms of business processes and workflows. How many steps does each process take? What are the paths that one would navigate to accomplish whatever tasks you're trying to enable?

Step 2: Create state machine

Step two is where the magic really starts. Take all of the information that you wrote down in step 1 and build an actual state machine out of it.

Wait, a state machine? Yep. A finite state machine.

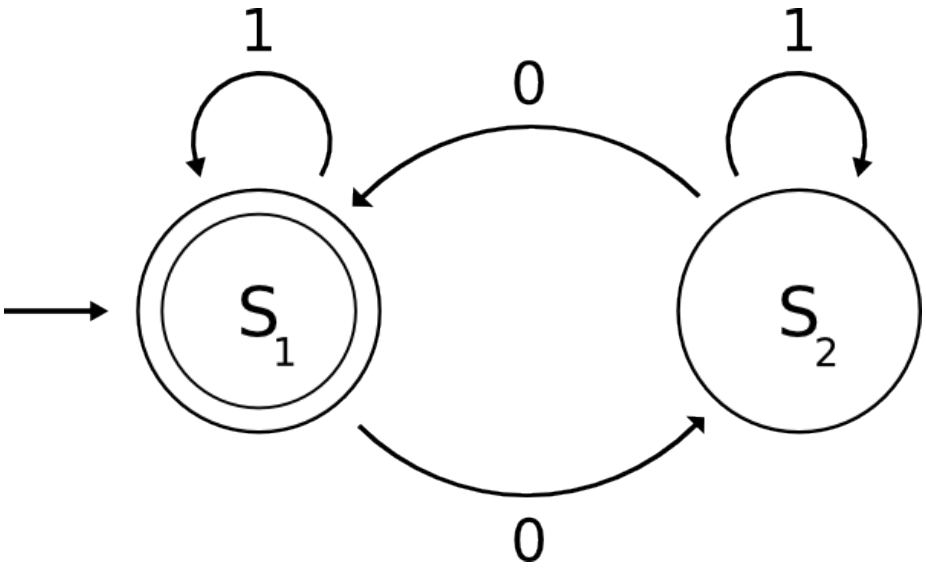


Figure 1: fsm

Remember how you wrote out business processes in step one? This is where you formalize them a bit. Choose the states your app can be in.

Begin with a starting state. This will end up being your API root. Make a node for each process you want to provide, and make them states. Pick transitions off of each one that make sense. Don't forget error states!

As a tiny example, let's say you're building a service that provides search. You might build a state machine that looks like this:

If you're wondering how to create these graphs, it's with DOT:

```
digraph graphname {
  start[label="API Root"];
  search_form[label="Search Form"];
  search_results[label="Search Results"];


  start -> search_form;
  search_form -> search_results;
}
```

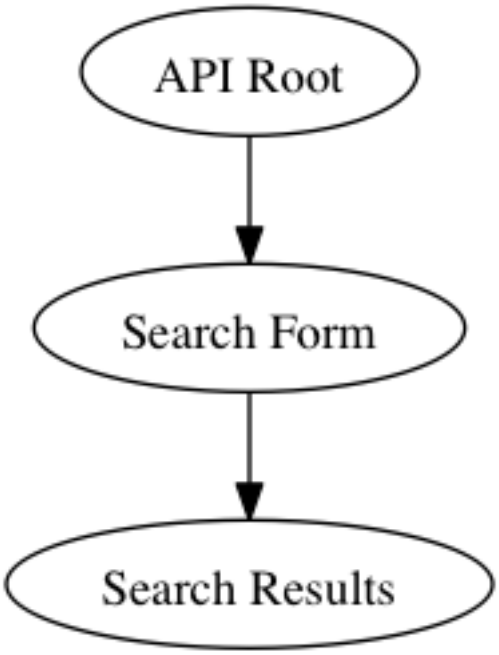


Figure 2: search example dfa

After installing `graphviz` on your system, you can run the `dot` command to make a .png from the above text with something like `dot state.dot -o state.png -Tpng`

Step 3: Evaluate media types

Now it's time to examine the graph we made and start thinking about hypermedia.

Each transition will represent some kind of hypermedia affordance. It may be a link, it may be a form, it may be something else. But each state and each transition are significant moving forward. Consider what kind of affordance each transition needs.

In our case, we need a regular link from the root to the search form and a templated link from the form to the results.

Also, consider what attributes you'll be trying to share. In our case, we have search results. They'll have a title, a URL, and a summary. Pretty simple.

Step 4: Create or choose media types

Next up, take all of your media type needs and compare them to what existing media types give you. It's always better to work with as generic of a type as possible. If you use an existing type, libraries will already exist. Your users will be more familiar with them. Clients that use your API will be less specialized.

There are four main media types that you should consider for an existing type:

1. HTML (I prefer the XHTML variant)
2. Collection+JSON
3. HAL
4. ATOM

All of these types have advantages and disadvantages that are out of the scope of this article. I consider these to be the standard, go-to hypermedia formats that you should reach for first when building an API.

If a generic type does not suit your needs, you have the option of building your own media type. You essentially have three options when building your own type:

1. JSON

2. XML
3. Something Custom

Nobody ever picks #3. That's an exaggeration, but it's also true. So yeah: build a type based on JSON or XML. Include the hypermedia affordances that you determined you needed earlier. Include the data types you needed earlier.

Building a media type is a fairly in-depth endeavor. I plan on writing a few more articles on this topic specifically. In the interest of keeping this short and to the point, I'll leave the rest of the details up to those future posts. "Building Hypermedia APIs with HTML5 and Node" is a fantastic resource on hypermedia media type design.

Step 5: Implementation!

Here's where the magic happens. Build it! Easier said than done, of course. This step is mega-specific and based on your tooling. But implement your client and server. Obviously, this step has a zillion details.

Step 6: Refinements

Nobody ever gets it right on the first try. Now that you've got something basic in place, look at how it can be improved. The easiest places to improve your application are caching and collapsing collections.

HTTP caching is a complicated beast. It's not too bad, but in general, caching is a Hard Problem. It's the best way to improve your application's scalability, response time, and other performance metrics. Appropriate caching is a huge boon.

Collapsing collections and other resources is another great technique. Our search type can take advantage of this. For example, rather than have a link to our search form, we can embed the search form into our API root. This collapses our state machine from three nodes to two and eliminates an HTTP request in the process. There are pros and cons to collapsing things into your API root, but it illustrates the process. If you have a collection of resources, often it can be a good idea to strike a balance between having the collection expose every attribute of each child resource and no attributes of the child resource. Creativity will be rewarded.

Conclusion

Obviously, each of these steps has a lot of depth. This post is titled "An Overview" for a reason. I expect it to raise more questions than it answers. But that's okay.

Sources Cited

- "REST APIs Must be Hypertext Driven", Roy Fielding
- DOT language
- HTML specification
- Collection+JSON Specification
- HAL specification

Media Type

> "And by the way, when people would ask me, 'Why do you care so much about putting media into e-mail?' I always said because someday I'm going to have grandchildren and I want to get pictures of them by e-mail. And people's reaction was to laugh and laugh."
>
> - Ned Freed

A 'media type' is a more generic name for what is called a "MIME type" in HTTP parlance.

MIME types are interesting, as they're sort of a historical accident that has happened to become incredibly important. MIME stands for "Multipurpose Internet Mail Extensions." Wait, mail? Yep, mail.

The original standards for email were created in 1982, with RFC 821. Because it was 1982, some limits that seemed reasonable were placed on the contents of messages, mostly around length. Email was also ASCII-only. That last one is the real kicker: what happens when I want to tell my Japanese colleague ? So extensions were created in many following RFCs. The authors of the MIME standard, Ned Freed and Nathaniel Borenstein, were really passionate about getting multimedia into email. The IETF didn't care about that a whole lot, but the internationalization angle appealed to them, and so Freed and Borenstein were successful.

Okay, so you want to send an image with your email. Sounds reasonable. How do you do that? Well, first of all, you've really got two things: the text of the email and the image itself. So you'd probably want to split it up into two chunks, right? So that's the whole 'multi-part' bit. But which part is an image and which part is text? And what kind of image? Letting mail clients figure this out would be burdensome on said clients. Why not just tell the client "this is a .gif" and "this is the text body"? That's what MIME types are. Eventually, the Content-Type header was added to HTTP as well.

Media types are a pretty simple idea: a tiny piece of metadata that describes what the chunk of data you're looking at is. Yet it's incredibly important. For instance, what kinds of data do we care about? If the actual metadata itself isn't standardized and machine-readable, it's basically useless.

> This is required functionality for X Mosaic; we have this working, and we'll at least be using it internally. I'm certainly open to suggestions as to how this should be handled within HTML; if you have a better idea than what I'm presenting now, please let me know. I know this is hazy wrt image format, but I don't see an alternative than to just say "let the browser do what it can" and wait for the perfect solution to come along (MIME, someday, maybe).
>
> Marc Andreessen, "proposed new tag: IMG"

It's important to ship, but it's also important to get it right. RFC 2046 made affordances for both: it includes a list of default types, as well as an initial hierarchy for types to go into, and then says that the IANA will have a registry for new types.

Media types are also incredibly important in Fielding's work:

> The data format of a representation is known as a media type. A representation can be included in a message and processed by the recipient according to the control data of the message and the nature of the media type. Some media types are intended for automated processing, some are intended to be rendered for viewing by a user, and a few are capable of both. Composite media types can be used to enclose multiple representations in a single message.

He also talks about design concerns:

> The design of a media type can directly impact the user-perceived performance of a distributed hypermedia system. Any data that must be received before the recipient can begin rendering the representation adds to the latency of an interaction. A data format that places the most important rendering information up front, such that the initial information can be incrementally rendered while the rest of the information is being received, results in much better user-perceived performance than a data format that must be entirely received before rendering can begin.

Let's go over one of those sentences again: "A representation can be included in a message and processed by the recipient according to the control data of the message and the nature of the media type." Let's shorten it a bit: "A representation can be processed according to the media type." The media type definition becomes our guide to process the response. That's why hypermedia clients are often said to be 'programming the media type.'

Sources Cited

- The MIME guys: How two Internet gurus changed e-mail forever
- A MIME Overview
- RFC 821
- RFC 2045
- RFC 2046
- RFC 2047
- RFC 2049
- RFC 4288
- RFC 4289
- Proposed New Tag: IMG
- "Architectural Styles and the Design of Network-based Software Architectures", Roy Fielding, Section 5.2.1.2: Representations

Building the W3CLove API

W3CLove is a project started by a student at Mendicant University. This student asked for some feedback about the API, and while it's perfectly fine, it's not exactly RESTful. I'm going to cover the process of transforming it from its current RPC style into an actual RESTful one here.

Step 1: Evaluate processes

Let's look at what the API actually does. What workflows do we need to support with this API?

Let's check out the W3CLove API page. There are two basic functions: evaluate an entire site or evaluate a page.

With an API this simple, you might wonder how it can possibly improve. We have two API calls; they accept one parameter. What's the matter with this design?

The problem, as is with most software, is hidden coupling.

First, we have coupling of our URLs to our implementations. Note that if we change our URLs, the API breaks. That sucks. We've coupled them quite tightly to our implementation.

Secondly, we use out of band information when returning our results. This couples our documentation of the responses to our implementation code as well. If we change some fields in our responses, the URL doesn't change and the media type doesn't change. Clients have no idea things are wrong! Then stuff breaks. Sad… we need to include information about how our responses are formed in the response.

We'll deal with both of these kinds of coupling in turn.

Step 2: Create state machine

We need to create a state machine for our two processes. Let's talk about decoupling these URLs. How do we get to our two processes without knowing what the URLs are? By linking, of course!

The basic idea is this: we only want our entry point URL to be published. We'll make sure it's always available, but after that, clients discover the URLs they need to do their processing. Let's ignore *how* they decide which URL is which and focus on what URLs we need.

Well, we'll need two resources, one for each kind of computation: our sitemap API and our webpage API. In order to process the computation, we need to pass in a parameter, though. Forms are a method that we can use to parameterize GET URLs, so we'll need two things from our API: we need to request a form to tell us how to evaluate our computation, and then we expect to process that form and get some sort of useful information back. A workflow version of this might look like this:

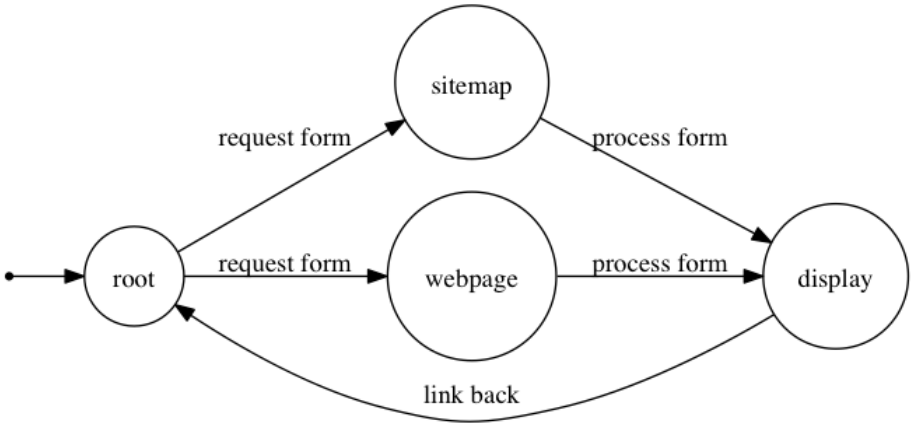


Figure 3: State machine

But really, we're forgetting a resource here. Displaying a sitemap response is different than viewing a webpage response, so we really need something like this:

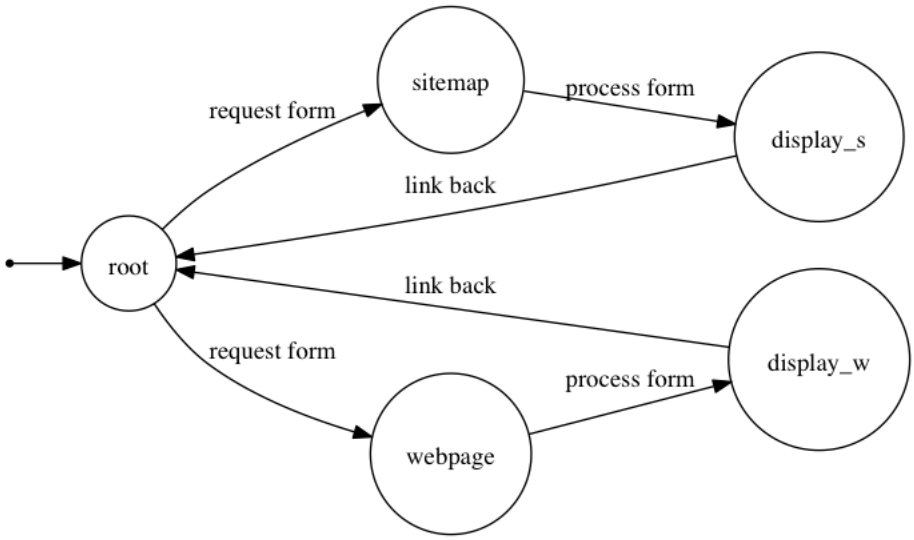


Figure 4: Second state machine

You can see that we're doing nearly the same thing in each step. Even easier for our design!

You'll also notice that I have a link back to the root in each of these displays. After performing one computation, you'll probably want to do another, so I've added a link back for convenience. It's not strictly necessary, but I prefer to wrap back around to the start. The flow is nicer; you can see what the users of your API will be doing.

Now that our workflows are settled, let's evaluate what media types we need.

Step 3: Evaluate Media Types

Media type design is almost as much art as it is science, and so I'm going to be a little bit terse here. It's really important to get media types right, as once they're out there, they need to be forward-compatible. So think about it hard!

Since JSON is all the rage with the kids these days, let's create a media type based on JSON. Why can't we use stock JSON? Well, we've already established that we need a form to template our URL, *and* we need to have a link back to our origin. JSON does not include semantics for links and forms! That doesn't mean the structure of JSON is bad, but we need to add those semantics on top of it and that means minting a new type.

Making a new media type has a few different steps, but it's kinda outside of the scope of this post. Normally, we'd put up some documentation at a stable

URL, but let's do this quick and dirty for now. We just want to talk at a high level here.

Step 4: Create Media Types

These media types are going to be 99% compatible with the current types that are being returned but with one change. I'll talk about the change afterwards. Let's discuss this new type:

The site validation+json media type

We're going to call this type 'validation+json'. For now, since we haven't registered it with anyone, we should give it the name `vnd.w3clove.validation+json`. This is a vendor specific media type since we're the only one that uses it.

The `vnd.w3clove.validation+json` type will conform to JSON structurally but have these additional semantics:

Elements A response MAY contain `created_at`, `scraped_at`, `scraping_success`, `updated_at`, `url`, `web_pages_count`, `validation_errors_count`, `validation_warnings_count`, and `pending_count` elements. All of these elements contain exactly what you'd think. In a real type declaration, I'd explain them further and individually.

It MAY contain a `web_pages` key that holds an array of responses. These responses will have these keys: `created_at`, `updated_at`, `url`, `validated_at`, `validation_errors_count`, `validation_warnings_count`, `w3c_validation_success`. Same deal here: in real documentation, I'd explain these fully.

A response MAY include a `links` element which is an array of objects. These objects MUST have these elements: `href` and `rel`.

A response MAY include a `forms` element which is an array of objects. These objects MUST have these elements: `href`, `rel`, and `data`. `data` will be an array of objects that MUST have two keys, `name` and `value`.

Rels Rels are included both in our `form` elements as well as in our `links` element. These names provide semantic meaning in any of these given places. Here are the ones for `validation+json`:

- `sitemap-form`: Following a link with this rel will lead you to a resource with a form for generating a Sitemap API request.
- `sitemap`: Processing a form with this rel will lead you to a resource that gives you validation information about a sitemap.

- `website-form`: Following a link with this rel will lead you to a response with a form for generating a Website API request.
- `website`: Processing a form with this rel will lead you to a resource that gives you validation information about a website.
- `root`: A link with this rel always leads back to the site root.

And with that, we're done with the media type! The big changes from the existing type are:

1. Adding `forms` and `links` portions to the responses. This is the hypermedia we were missing!
2. Adding link relations. We need these to know which links to follow.
3. Even single-site responses are returned in a `web-sites` array of one element. This simplifies our need from two different responses to one. Why define two types when you can make it all work in one type?

Step 5: Implementation!

Let's pretend the site exposes this API at `http://w3clove.com/api/`. Here's a sample cURL session:

```
$ curl -H "Accept: application/vnd.w3clove.validation+json" \
    http://w3clove.com/api/
{
  "links":[
    {"rel":"website-form", "href":"http://w3clove.com/api/..."},
    {"rel":"sitemap-form", "href":"http://w3clove.com/api/..."}
  ]
}
```

I haven't even filled in the URLs. They shouldn't matter. So I'm not gonna tell you what they are. :p

We parse this with JSON and follow the link (in Ruby notation):

```
$ curl -H "Accept: application/vnd.w3clove.validation+json" \
    response["links"].find{|l| l["rel"] == "sitemap-form"}["href"]
{
  "forms":[
    {"href":"http://w3clove.com/api/...",
     "rel":"sitemap",
```

```
     "data":[
        "name":"check",
        "value":""]}
  ]
}
```

That's not valid; the point is that I'm using the Ruby notation to emphasize that we follow the link, not calculate the URL.

Anyway, Now we want to make this request…

```
$ curl -H "Accept: application/vnd.w3clove.validation+json" \
    response["forms"].find{|f| f["rel"] == "sitemap"}["href"]
    + "?" + response["forms"].find{|f| f["rel"] == "sitemap"}["data"]["name"]
      + "=" + "http://www.zeldman.com"
```

Okay, so that calculation was awkward. You'd do it in code. Anyway, we get a response back:

```
{
    "created_at": "2012-01-30T01:17:04Z",
    "scraped_at": "2012-01-30T01:17:10Z",
    "scraping_success": true,
    "updated_at": "2012-01-30T01:17:10Z",
    "url": "http://www.zeldman.com",
    "web_pages_count": 57,
    "validation_errors_count": 2951,
    "validation_warnings_count": 8,
    "pending_count": 0,
    "web_pages": [{
      "created_at": "2012-01-30T01:17:09Z",
      "updated_at": "2012-01-30T01:17:23Z",
      "url": "http://www.zeldman.com/",
      "validated_at": "2012-01-30T01:17:23Z",
      "validation_errors_count": 0,
      "validation_warnings_count": 0,
      "w3c_validation_success": false
    }, {
      "created_at": "2012-01-30T01:17:10Z",
      "updated_at": "2012-01-30T01:21:14Z",
    "url": "http://www.zeldman.com/2011/12/21/the-big-web-show-no-61-khoi-vinh-of-mixel-a
      "validated_at": "2012-01-30T01:21:14Z",
      "validation_errors_count": 7,
      "validation_warnings_count": 0,
      "w3c_validation_success": true
```

```
    }, {
      "created_at": "2012-01-30T01:17:10Z",
      "updated_at": "2012-01-30T01:21:09Z",
    "url": "http://www.zeldman.com/2011/12/22/migrate-if-you-like-but-touristeye-is-not-a
      "validated_at": "2012-01-30T01:21:09Z",
      "validation_errors_count": 8,
      "validation_warnings_count": 1,
      "w3c_validation_success": true
    }, {
      "created_at": "2012-01-30T01:17:10Z",
      "updated_at": "2012-01-30T01:21:08Z",
    "url": "http://www.zeldman.com/2011/12/23/hitler-reacts-to-sopa/",
      "validated_at": "2012-01-30T01:21:08Z",
      "validation_errors_count": 6,
      "validation_warnings_count": 0,
      "w3c_validation_success": true
    }],
    "links":[
      {"rel":"root", "href":"http://w3clove.com/api"}
    ]
}
```

Bam! We've got all of our data. You can imagine how this would work for the other process, too.

Improvements

We can do a few things that might help performance. First, some client-side caching would help a lot, especially on our root page: it probably doesn't change very often.

Secondly, we can just embed the forms into our root as well, if we'd like: Since they probably won't change often either, that might make sense, and then we wouldn't need to make as many requests. It all depends!

Sources Cited

- W3CLove
- W3CLove API

APIs Should Expose Workflows

One of the hardest conceptual jumps for me in understanding hypermedia is that Ruby on Rails teaches bad habits. The combination of ActiveRecord plus RESTful Routes means that by default, it tells you that the correct way to expose your service is by putting your database on the web almost directly. Those of you who aren't Rubyists are probably laughing, but consider most Web 2.0 style APIs: this is exactly what they do. "Here's the endpoint for Users, here's the endpoint for Posts, here's the endpoint for Comments." This is fundamentally flawed.

Duplication of business logic

Long ago, I heard someone talking about the advantages of server side Javascript. One of their points: "You can share your models on the server and the client!" I was pretty horrified. This is the absolute opposite of DRY. Then again, I also felt their pain: often, implementing a client library means taking large chunks of your business logic and copying them over to your new library. And what else is ActiveResource than a way to have 'remote models'?

This idea is what Fielding calls the 'mobile object' style: your object travels over the network to do some processing. This style can be worth it if your data is large and your code is small. But we don't need to do that with the web.

Tight coupling to the data model

If you expose all of your tables over the web, you are forever tied to those tables. Well, unless you want to break backwards compatibility. This is a very strong form of coupling, and one that Rubyists in particular are very bad at recognizing. Yes, Java people, start laughing now: the hot topic at conferences in the Ruby world is how you should use POROs and use a separate class to persist them, since otherwise, you tie yourself to the database. We're rediscovering 2001 all over again.

Why does this happen?

Developers build APIs in this fashion because the tooling encourages it, they don't think it through, and they default to making APIs at a lower level than they should.

Tooling

I've already mentioned Rails. ActiveRecord encourages you to map your domain models 1:1 with your database tables, and the generated routes let you CRUD via HTTP. It's so simple that you have to actively work to not let this happen.

On top of that, the scaffold generator defaults to also giving you XML or JSON responses as well as HTML, which encourages you to have your API mirror your site exactly. This isn't a problem, except that we've already discussed how the site ends up being a reflection of the database.

Thinking it through

The tooling aspect feeds into this. Because it's so easy, we assume it's right. And because we've built two or three APIs this way, it's now the default way.

Also, many developers don't think. Especially those who are just in it for the 9-5 work, who don't practice their craft, and don't contribute to Open Source. There's nothing wrong with that: I don't love my bicycle the way enthusiasts do! But socially, this becomes reinforcing. Programmers are as susceptible to trends as anyone else, maybe even more. So as soon as those first Rails apps came out, everyone else tried to copy them. Including other Rails apps. Pretty soon, doing this was considered obvious and maybe even a best practice.

Low level access

I think that developers also tend to make their APIs too low level. That's sort of what I'm saying already with "don't expose your tables", but this is really at the root of it: that's too low of a level to be appropriate. Exposing your data model *forces* your clients to recreate your business logic to copy your business processes, which is probably why they use your service in the first place!

I don't use GitHub because they expose Git over HTTP, I use them because that green merge button is so handy. So why make your clients redo all of the work to copy your awesome process?

The answer: workflows

Since your process is what your users want, just give that to them! This is the essence of hypermedia: use the links, forms, and other affordances to guide your users through your business processes and workflows.

This is also why we design state machines as part of the design process: they model these workflows, and that's what we want to expose.

By exposing your workflow rather than your data model, you're free to change data models at a later time. Your clients don't have to duplicate your business logic, they can just follow the hypermedia and let it guide them through. This also means that your clients will be more resilient to change, due to this decreased coupling. If you offer a new workflow, a well-made client will be able to automatically present it to your users, with no code updates. Machine to machine interactions won't be able to get this benefit, but they'll still enjoy the lack of duplication.

Transmuting Philosophy into Machinery

> Kessler's idea was, that besides *the law of mutual struggle* there is in nature *the law of mutual aid*, which, for the success of the struggle for life, and especially for the progressive evolution of the species, is far more important than the law of mutual contest. This suggestion - which was, in reality, nothing but a further development of the ideas expressed by Darwin himself in *The Descent of Man*, seemed to me so correct and of so great an importance, that since I became acquainted with it I began to collect materials for further developing the idea, which Kessler had only cursorily sketched in his lecture, but had not lived to develop. He died in 1881.
>
> - Peter Kropotkin, "Mutual Aid: A Factor of Evolution", p21

Rob Conery is a pretty cool guy. While I always enjoy reading his blog, he's been at battle with the Hypermedia crowd recently. It's always good natured, though, and he means well.

He recently posed an interesting question to his blog:

> I would like to invite the good people who have engaged with me over the last few days to jump in and write me up an API – and by way of explanation – show how their ideas can be translated into reality.

Great! After all, social exchange is one of the building blocks of society:

> In so far as the process of exchange transfers commodities from hands in which they are non-use-values to hands in which they are use-values, it is a process of social metabolism.
>
> Karl Marx, "Capital, Volume 1", p198

Let's apply what we've learned about the basics of designing hypermedia APIs. Here are his requirements:

Use Cases

This is step one: simple authentication and then consumption of basic data. The client will be HTML, JS, and Mobile.

Logging In

Customer comes to the app and logs in with email and password. A token is returned by the server upon successful authentication and a message is also received (like "thanks for logging in").

Productions

Joe User is logged in and wants to see what he can watch. He chooses to browse all productions and can see on the app which ones he is aloud[sic] to watch and which ones he isn't. He then chooses to narrow his selection by category: Microsoft, Ruby, Javascript, Mobile. Once a production is selected, a list of Episodes is displayed with summary information. Joe wants to view Episode 2 of Real World ASP.NET MVC3 – so he selects it. The video starts.

Episodes.

Kelly User watches our stuff on her way to work every day, and when she gets on the train will check and see if we've pushed any new episodes recently. A list of 5 episodes comes up – she chooses one, and watches it on her commute.

The design process

Step 1: Evaluate Process

Fortunately, this has been done for us, in the Use Cases above. Sweet!

Step 2: Create state machine

Taking all of this into account, I drew out this state machine:

Basically, you start at a root. Two options: see the newest list of productions, or see them all. You can filter all of them by a category. Eventually, you end up picking one. This workflow should be enough to support all of our use cases.

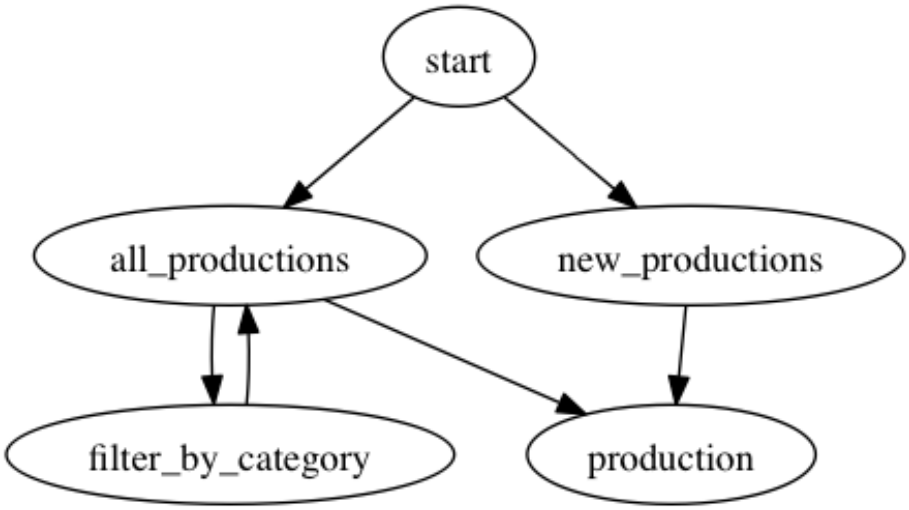


Figure 5: tekpub state machine

Step 3: Evaluate Media Type

Okay, so, he mentions this in use cases:

> The client will be HTML, JS, and Mobile.

I'm not 100% sure what he means here: I think it's that we'll be building a site on this API (html), it'll have heavy JS usage (js), and probably a mobile version or possibly a native client (mobile).

Given this information, I'm going to choose JSON as a base format. Besides, our developers tend to like it. ;)

After that choice is made, we also need these things:

- Filtering things means a templated query of some kind, so we'll need some kind of templating syntax.
- We need lists of things as well as singular things. I like to simply this by representing singular things as a list of one item. So, lists and individual items.
- We also have a few attributes we need to infer from these loose requirements. No biggie. :)

Step 4: Create Media Types

Based on this, I've made up the application/vnd.tekpub.productions+json media type. Key features, based on our evaluation:

- Each transition in our state machine has a relation attribute
- Each transition that needs to be parameterized has some sort of template syntax
- Each attribute that we need someone to know about has a definition
- Everything is always a list. It may contain just one item. Our client's interface can detect this special case and display something different if it wants.

Step 5: Implementation!

That's for Rob! ahaha!

However, you might want a sample implementation. I might be building one in the future, however, for now, I'm working on the get_a_job project, which you can read in the series on it.

What about auth?

Oh, I didn't handle the auth case. That's because auth happens at the HTTP level, not at the application level. HTTP BASIC + SSL or Digest should be just fine.

But, but, but... I didn't get any verbs! Or URLS!

I know. Fielding:

> A REST API should spend almost all of its descriptive effort in defining the media type(s) used for representing resources and driving application state, or in defining extended relation names and/or hypertext-enabled mark-up for existing standard media types. Any effort spent describing what methods to use on what URIs of interest should be entirely defined within the scope of the processing rules for a media type (and, in most cases, already defined by existing media types). [Failure here implies that out-of-band information is driving interaction instead of hypertext.]

As well as

> 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].

And

> A REST API should be entered with no prior knowledge beyond the initial URI (bookmark) and set of standardized 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.]

Soooooooooo yeah.

Sources Cited

- "Mutual Aid: A Factor of Evolution", Kropotkin
- "Capital, Volume I", Karl Marx
- "Someone save us from REST"
- "Moving the Philosophy into Machinery"
- "REST APIs Must be Hypertext Driven"

Distributed Application Architecture

> The Internet is based not on directionality nor on toughness, but on flexibility and adaptability. Normal military protocol serves to hierarchize, to prioritize, while the newer network protocols of the Internet serve to *distribute*.
>
> Protocol, p30, Galloway

> It is at work everywhere, functioning smoothly at times, at other times in fits and starts. It breathes, it heats, it eats. It shits and fucks. What a mistake to have ever said *the* id. Everywhere *it* is machines - real ones, not figurative ones: machines driving other machines, machines being driven by other machines, with all the necessary couplings and connections.
>
> Deleuze and Guattari, "Anti-Oedipus"

This is one of those compound terms that sounds more impressive than it really is. Let's unpack it.

Application Architecture

All software applications have some sort of underlying architecture that governs the design of the system. It may not be consciously built, but it's there. There are many different software architectural styles.

Some applications even use multiple styles in different portions of an application. You can talk about the overall style, or the style of a particular locality. This is similar to a city; different neighborhoods have different styles. But they all have *some* kind of underlying logic.

Distributed Application

One way of categorizing software architectures is to consider which portion of a system maintains control. Of course, no mention of systems of control is complete without mentioning Gilles Deleuze, and in fact, his concept of a 'control society' is identical to the concept of control in a software system:

> We're definitely moving toward "control" societies that are no longer exactly disciplinary. Foucault's often taken as the theorist of disciplinary societies and of their principal technology, confinement (not

> just in hospitals and prisons, but in schools, factories, and barracks). But he was actually one of the first to say that we're moving away from disciplinary societies, we've already left them behind. We're moving toward control societies that no longer operate by confining people but through continuous control and instant communication.

While Deleuze mentions computers explicitly in several places, it is Alexander Galloway who has really applied his work to software. In "Protocol," Galloway classifies three kinds of network systems:

Centralized Networks

Centralized networks are a method of organization that looks like an asterisk. A central node holds all control, and is connected to many other nodes, none of which are connected to each other.

This network has advantages in that it's simple, unambiguous, and clear. It also has many, many weaknesses, the chief of which is simple: remove the controlling node, and the whole thing collapses.

Decentralized Networks

I'll defer to Galloway on this one:

> A _de_centralized network is a multiplication of the centralized network. In a decentralized network, instead of one hub there are many hubs, each with its own array of dependent nodes. While several hubs exist, each with its own domain, no single zenith point exercises control over all others.

A great example he cites of a decentralized network is the airline system. For example, this weekend, I'll be going to Chicago. I live in Pittsburgh. I have to fly to Philadelphia first. Crazy. But that's how it works. Fly to a hub, fly to the destination.

While a decentralized network is more democratic, it's still hierarchical. There are more nodes in charge, but some nodes are still subordinate to others.

Distributed Networks

A distributed network is one where no node is actually in control. Equality, democracy, and horizontalism are the names of the game. Galloway again has a spectacular example, which is the Dwight D. Eisenhower System of Interstate and Defense Highways. Yes, that's it's actual name. :) No city is in charge, yet

they're all interconnected in a network. This design was explicit, as troops and supplies would need to be moved between locations if there were ever war on American soil.

We need to recognize the web is one giant application

Thinking about the Internet as a network, it's really one big giant application that functions in a distributed fashion. On a technical and/or protocol level, Google does not have power over Facebook. Every application is on equal footing with every other.

Think about the process of building a web application, from this perspective. There exists a swirling mass of nodes, interconnected in an unfathomable, tangled web. You build your own little node, toss it into the sea, and connections develop. It's really fantastic in many ways.

We subtly acknowledge this in the ways we talk about it: "I used the Internet to solve the problem." "I love the Internet." "APIs have really created a programmable web." "I found this comic on the web by searching." The Internet, in many ways, really is a single, monolithic thing that we interact with. It may have many pieces, but over time, some go away, and new ones appear. In 1997, I might have used a different search engine that I'd use today, but I'd say the same thing: I found the answer on the web.

Sources Cited

- "Control and Becoming", Negri and Deleuze
- "Protocol: how control exists after decentralization", Alexander Galloway
- "Anti-Oedipus", Deleuze and Guattari

Software Architectural Style

REST is often referred to as a 'software architectural style.' From Fielding's dissertation:

> This chapter introduces and elaborates the Representational State Transfer (REST) architectural style for distributed hypermedia systems,

He explains what 'architectural style' is in Section 1:

> An architectural style is a coordinated set of architectural constraints that restricts the roles/features of architectural elements and the allowed relationships among those elements within any architecture that conforms to that style.

He elaborates further in Section 5:

> The design rationale behind the Web architecture can be described by an architectural style consisting of the set of constraints applied to elements within the architecture. By examining the impact of each constraint as it is added to the evolving style, we can identify the properties induced by the Web's constraints. Additional constraints can then be applied to form a new architectural style that better reflects the desired properties of a modern Web architecture. This section provides a general overview of REST by walking through the process of deriving it as an architectural style. Later sections will describe in more detail the specific constraints that compose the REST style.

As it turns out, there's actually a Wikipedia page named "Architectural Style":

> Architectural styles classify architecture in terms of the use of form, techniques, materials, time period, region and other stylistic influences. It overlaps with, and emerges from the study of the evolution and history of architecture. In architectural history, the study of Gothic architecture, for instance, would include all aspects of the cultural context that went into the design and construction of these structures. Hence, architectural style is a way of classifying architecture that gives emphasis to characteristic features of design, leading to a terminology such as Gothic "style".

Programmers love their architecture analogies. Putting the two definitions together, if buildings can have an architectural style based on its 'form, techniques, materials, and other stylistic influences,' then software can have a style based on its 'set of constraints applied to elements within the architecture.'

Why does this matter?

Well, one of the difficult things about REST is that there's no standard to follow. Styles don't exactly have a checklist of things that you must comply with to be considered 'in style,' they're gray, not black and white. While we do have a list of things that make something RESTful, we don't have something as strong as, say, an ISO standard. And since REST is fundamentally an *architectural* style, in order to determine if something is RESTful, you need to examine its architecture.

This is why routing can't make your API RESTful. Returning JSON or XML can't make your API RESTful. The underlying architecture has to comply with the constraints that define the style. Everything else is secondary.

Sources Cited

- "Representational State Transfer (REST)", Roy Fielding
- "Architectural Style," Wikipedia

Human vs Machine Interaction

There are two broad categories of interaction models: Human (H2M) and machine (M2M). Humans have the ability to reason and make decisions based on new information, while machines currently do not. In order for M2M interactions to become more like H2M interactions, we'd need strong AI to get to the point where machines could pass the Turing Test.

Hypermedia APIs do two things: convert previous M2M interactions into H2M interactions by exposing workflows rather than data models, and allow for dynamic updating of H2M interactions via generalized media types.

Introduction to Hypermedia Clients

A client/server architecture means that we have two distinct parts to worry about: clients and servers. This seems obvious, but I've been largely talking about servers so far. There are good reasons for this, but now it's time to dive into some client work. I have made an example client to show you. Let's dig into it and see how it works.

ALPS and microblogging

There's a great technique for building hypermedia media types made by Mike Amundsen called "ALPS". It describes how to use the (X)HTML 'profile' attribute of the meta tag to define your application's semantics. He's created one of these profiles for microblogging. You can find a link to the full spec at the end of the article, but for now, let's check out the client we're going to build. Here are some screenshots:

Basically, we put in our credentials, and we can see a list of posts and post our own updates. Super easy.

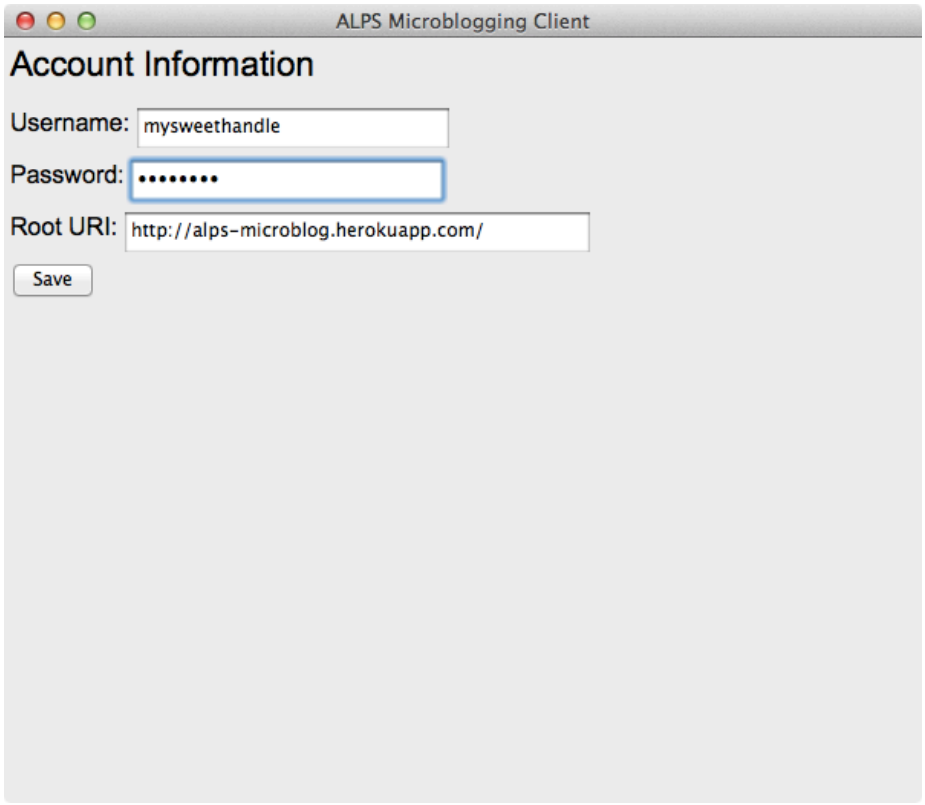


Figure 6: Preferences screen

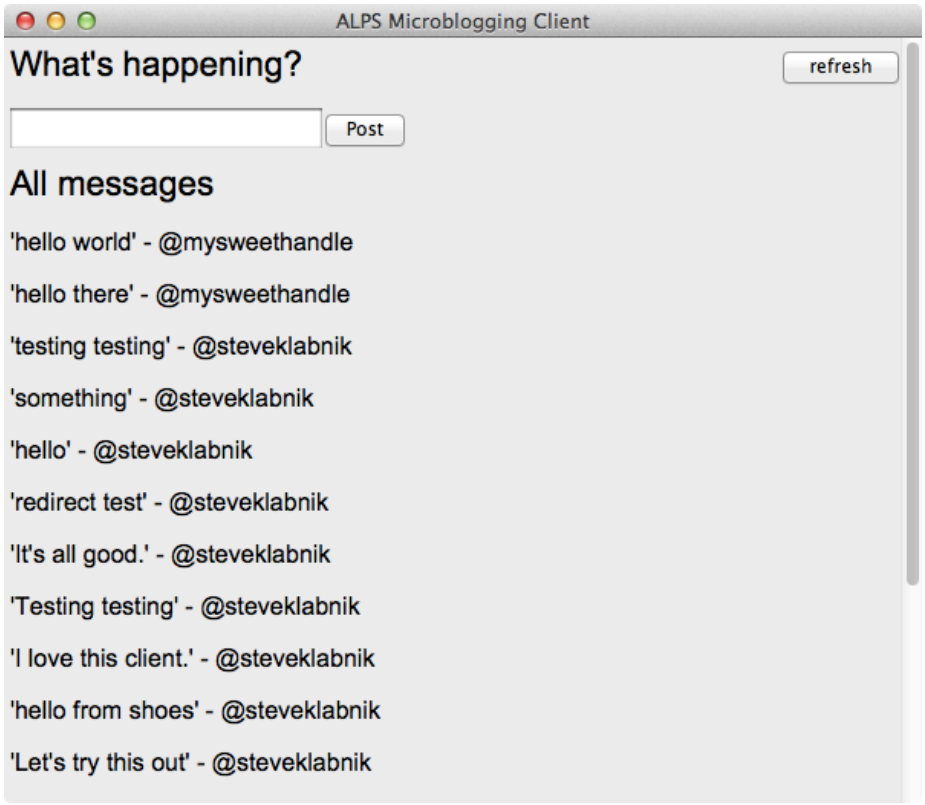


Figure 7: Main Screen

Here's the code. If you don't read Ruby, don't worry: it's not too bad, and while I'm going to dump it all up front here, we'll talk about bits and parts that are important later. I just want to provide it all so that you can grab it and run it.

Speaking of running it, we're using Shoes to build the GUI. Shoes makes heavy use of Ruby's blocks to build callbacks. To run the app, put both files in the same directory, start Shoes, and click "Open an app" and select microblog_client.rb.

First, a little wrapper around Ruby's default HTTP libraries for simplicity:

[]

Then, our main client code:

[]

You can check out the server code in Node.js here. Mike built the first version and then I did some stuff to get it going on Heroku.

Rule 1: No URIs!

The first and most important rule of building a proper hypermedia client is to not use any URIs but the first and to even make the first configurable, if possible.

Searching through the source, you can see that there's only one URI coded into the app, and I even give you the option of changing it on the preferences pane. This is really cool because if you want to use it with a different ALPS service than the one I built it with, you can just change the root and it'll work with that service too. For example, I hear rstat.us may do this with their API soon.

This server-agnostic-ness is given to us because we've truly decoupled the client and server through hypermedia. As a user, this is useful because we can use our favorite client with a variety of services.

I thought that I remembered you could do this with the official Twitter client, and @mislav pulled through:

> @steveklabnik Twitter for iPhone when adding a new account. See mislav/twin on GitHub

Rule 2: Hypermedia drives the interface

Let's examine the `index` method that is our main interface. Collapsed, the blocks look like this:

[]

We have one filter to make sure that we have our username/password set properly and then three user interface elements. There are two kinds: the ones with the if and the one without. Basically, the one without is an option to always return to the root again. This element is *not* hypermedia driven because we know the root URI will always exist and we always want to give users this option. We'll talk more about the other two first.

Hypermedia driven

The other two, with the ifs, are driven by the hypermedia: just show this interface element if the particular transition appears in the response from the server. This is the core of any hypermedia client. Note that each element is driven by a particular chunk of the response and that they also directly correspond to different actions that we could take from here. When I built this client, I did it in sections: first the element that displayed the list of results and then the one that let me post updates. This lets you partition off chunks of the specification to implement and keeps things simple.

Since we've done this, the server could change which transitions are valid and our interface would work just fine with no changes. Maybe we're overloaded, so we temporarily disable posting updates. We could stop returning the 'post an update' portion of our response, and our clients would no longer let people overwhelm our servers. Once we're good, we can turn it back on, and our clients work again!

Related to that, in an effort to keep this client simple, I left out a lot of error handling. In particular, if you mis-type your username and password, the error is basically silent. Bring up the Shoes Console with control-/ or command-/ to see the errors.

Not hypermedia driven

Now, I've spent time talking about how your interface should be driven by hypermedia, and then I made an element that isn't! What gives? Well, clients have the ability to add things if they want. The key is that **application state transitions** should be driven by hypermedia, but client state transitions don't have to be.

What if we wanted to change our client to look more like the official Twitter Mac client? Where you click a button to bring up the 'make a new post' box:

We could totally do this in code. This kind of transition is about hiding or showing a UI element for design's sake, not limiting the kinds of transitions we can make from a particular application state. Therefore, showing or hiding both the button and pop-up box should be hypermedia driven, but the decision to have it be a pop-up or part of the main interface is our choice as a client.

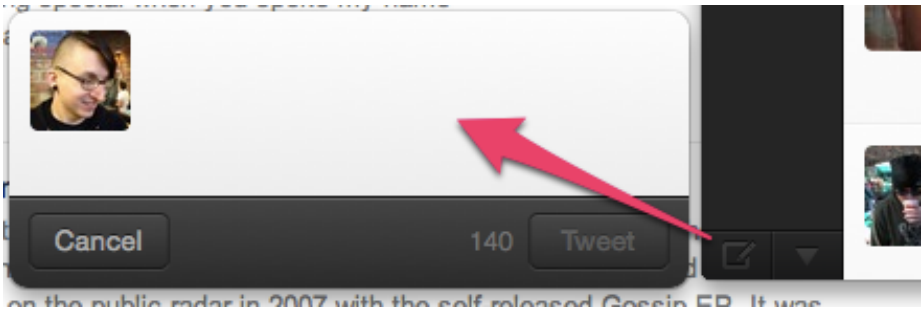


Figure 8: tweetie post tweet

The button we have is sort of a weird exception: we want to be able to update the list, so we make a button that always takes us back to the root. We know that action is stable, and it may or may not actually be in the response, so we just went ahead and did it. This functionality is **incredibly** unlikely to change, as it is the API root.

Rule 3: Ignore what you don't understand

Some people think this should be rule #1. It is incredibly important: basically, if you see something in the response that you don't know what to do with, don't crash, just ignore it!

You can see this with the level of validation we do on the response: we just search for two specific kinds of elements in the response and absolutely ignore everything else. This isn't just useful for flexibility's sake but also important to be forwards-compatible, for instance. We want to be able to add things later and not have old clients break, and if they broke when seeing something they didn't understand, it'd be much harder to add functionality.

Rule 4: Use the spec!

Try to implement your client by looking at what the spec for your media type says, not what exactly the server returns. Remember, client programming is very much "programming the media type", not programming for the specific service. The media type definition should give you all the information you need to implement a client for that service.

It can be fun to spin up a tiny sinatra app and just copy/paste any example response that the media type definition gives you to give your prototype client a try.

Rule 5: Don't tie too close to structure

Check out the two XPath expressions I used to see if the elements we were looking for existed:

```
//form[@class='message-post']
//div[@id='messages']/ul[@class='all']
```

Note specifically the `//`. The following XPath expressions would find the same elements that the previous XPaths would in the current iteration of the server's responses:

```
/html/body/form[@class='message-post']
/html/body/div[@id='messages']/ul[@class='all']
```

In fact, the single slash before the `ul` in the first set of XPath expressions may even be a mistake; I probably should have made that a `//` as well. What's important to note is that while these sets are 'equivalent' in the sense that they'll both find the proper elements, one of them is super brittle in terms of structure of the response. Basically, the second ones imply that they're both always top-level elements, while the first ones just see if they exist anywhere in the document. This flexibility allows our client to cope with server-level changes much more easily. Unless the spec specifies *exactly* where the element exists in relation to other elements, use fuzzy matches like `//` rather than exact ones like `/html/body`.

Rule 6: It's okay to implement just part of a spec

Our client does not implement the full application semantics. That's okay! We have a slimmed-down client. Maybe we don't really use microblogging all that much and so we just want a simple client. Maybe we want to follow agile methodology and implement an MVP rather than a fully-featured client. That's totally okay. Combined with the 'ignore stuff you don't understand' rule, this can let one server handle many different kinds of apps. Our full client (a web browser, in this case) just exposes extra transitions (like message permalinks) and our simple client (the Shoes app) just exposes a subset. There's nothing wrong with this at all.

Rule 7: Caching

This is something that wasn't directly demonstrated here but is also very important. It's so important, in fact, that it's getting its own article or two. Right now, we make more requests than we need to because we're ignoring client-side

caching. Client-side caching is the #1 thing you can do to make your clients more performant. Caching is *hard* though, from a non-technical angle: making sure your caches don't go stale for too long can be a complex challenge. Just remember to consider this for now, and we'll come back to it as a topic in a future article.

Sources Cited

- ALPS specification
- Shoes site

Versioning is an anti-pattern

We often say "use the right tool for the job", but when managing change in software systems, we always use versioning. Hypermedia APIs are actually hindered by introducing versioning and manage change in a different way. With that in mind, there are also a lot of options for managing change in a Hypermedia API. We'd like to change our service and break as few clients as possible. Versioning is only one way to manage change, though… and my contention is that it's not appropriate for hypermedia services.

> REST limits the possible contract changes to just two kinds: adding data or controls and removing data or controls from a message
>
> Jan Algermissen

How and Why Software Changes

There's a really great book about changing software titled "Working Effectively with Legacy Code". In it, Michael Feathers talks about how working with legacy code really means managing change. From the book:

> For simplicity's sake, let's look at four primary reasons to change software.
>
> 1. Adding a feature
> 2. Fixing a bug
> 3. Improving the design
> 4. Optimizing resource usage

and

> In general, three different things can change when we do work in a system: structure, functionality, and resource usage.

I consider 'removing a feature' to be a subset of 'fixing a bug.' You're taking some unwanted behavior out of a system.

There's another good model for thinking about changes: the backwards/forwards compatibility model. Something is considered 'backwards compatible' if it is able to work with older versions of the same software. Something is considered 'forwards compatible' if it can work with newer versions of the same software. Ultimately, we want to shoot for designs that are both forwards and backwards compatible.

We can combine both of these axes of change. We may add a feature and make it backwards compatible when doing so. We may fix a bug and make it forwards compatible. One model is about the *kind* of change, the other is about the *compatibility* that the change has with our system.

Versioning is Coupling

The main problem with versioning is that it encourages tight coupling between clients and servers. Before I justify that opinion, let's consider why coupling between clients and servers is bad.

From Fielding:

> Perhaps most significant to the Web, however, is that the separation [of clients from servers] allows the components to evolve independently, thus supporting the Internet-scale requirement of multiple organizational domains.

And again:

> [Via the uniform interface] implementations are decoupled from the services they provide, which encourages independent evolvability.

Basically, coupling between clients and servers means that it's hard to scale. When changes need to be made, you have to notify clients to upgrade. Imagine if your web browser needed to be updated every time Google deployed. Granted, Chrome feels like that sometimes, but you get my drift. ;) In fact, Chrome auto-updates so often and silently *because* getting clients to upgrade is such a problem.

Versioning as coupling should be obvious: you want a particular client version with a particular server version. This is normally considered a good thing,

though: by introducing this coupling, you protect yourself against future breakage. Basically, versioning allows us to be really reckless with our changes. We can totally re-write things between version 1 and version 2. The cost we pay is that coordination cost of getting all clients and servers updated to the newest version as quickly as possible. However, if we took a more conservative and careful approach to change, we could do a little bit of extra work and gain a lot of benefits.

Managing change without versioning

So what do we actually do, then? Let's examine the two types of change: forwards compatibility and backwards compatibility.

Forwards compatibility

There's one primary consideration with designing clients for forwards compatible changes: ignore what you don't understand. That way, new functionality won't break you.

To build on the RESTbucks example (used in the book Rest in Practice as well as the article How to GET a Cup of Coffee by the same authors), let's examine this response for paying for a cup of coffee:

```
<order>
 <drink>latte</drink>
 <cost>3.00</cost>
 <link rel="payment" uri="...">
</order>
```

Let's add some functionality to use a gift card to get a discount. The response would look like this:

```
<order>
 <drink>latte</drink>
 <cost>3.00</cost>
 <link rel="payment" uri="...">
 <link rel="giftcard" uri="...">
</order>
```

If an old client is correctly written, it shouldn't break when it sees this response. It should do the exact same thing as the first response. It just ignores the new functionality.

You can also make your media type amenable to making old clients pick up new changes. For example, Collection+JSON has a queries section that looks like this:

```
"queries" : [
   {"rel" : "search", "href" : "http://example.org/friends/search",
"prompt" : "Search"
       "data" : [
         {"name" : "search", "value" : ""}
       ]
     }
   ]
```

Let's say that we used to have this search response. We want to remove it, and add a new query, filter. The response looks like this:

```
"queries" : [
   {"rel" : "filter", "href" : "http://example.org/friends/filter",
"prompt" : "Filter"
       "data" : [
         {"name" : "filter", "value" : ""}
       ]
     }
   ]
```

You can imagine that now, since our client was coded against the ideas of 'queries' rather than one of search directly, it'd be pretty trivial for the client to not display a search box and display a filter box instead when it gets this new response.

Backwards compatibility

From time to time, we want to remove functionality that used to exist. We may want to make our old process 'deprecated', and so we tell new clients to ignore it if it's there, or to prefer some sort of affordance before using an older one.

This can be handled in the opposite way we dealt with new functionality: if you don't see something, don't display it. So when we drop the Search functionality, even if we can't find the new Filter stuff or display it, we shouldn't be displaying a search box. The client isn't broken directly, as our business process is no longer valid. It can't make invalid requests.

Considerations for Media Type Design

If you're making a new media type, consider *very* carefully if something needs to be REQUIRED for your design. Removing these elements later will force you to create a new explicit version or support it as deprecated for the rest of time. Tread carefully!

While many people discuss versioning within the context of 'version the URI or version the media type', they often forget another option: version within the message. HTML does this, for example, with its DOCTYPEs. This is how `text/html` as a media type has been unversioned for almost three decades. However, note that going forward, `<!DOCTYPE html>` is the only doctype, and HTML will no longer be versioned. Everything will be fully backwards-compatible as well as forwards-compatible forever.

Sources Cited

- "Understanding the role of media types in RESTful applications", Jan Algermissen
- "Working Effectively With Legacy Code", Michael C. Feathers
- Forward Compatability
- Backward Compatability
- "Architectural Styles and the Design of Network-based Software Architectures, Section 5.1.2" Roy Fielding
- "Architectural Styles and the Design of Network-based Software Architectures, Section 5.1.5" Roy Fielding
- "How to GET a Cup of Coffee"
- "HTML is the new HTML5"

Out of Band

"Out of band" is a term that's used a lot in discussions about HTTP, and it's often totally misunderstood. I'll explain by way of example:

Five of us are out to dinner, and we're having a discussion. I reach under the table and send you a text message with a joke about the person who's currently talking. That text message would be outside of the primary mechanism of communication that we're interested in, and is therefore 'out of band.'

The term originally comes from radio, where different transmissions are sent over different portions (bands) of the spectrum.

Now, what's interesting about this is that we can remove participants, and it's still true: If we're having a one-on-one conversation, and I send you a text, that text is still out-of-band with concern to our discussion.

HTTP is a 'stateless' protocol. Everything that the server needs to process a request must be contained in the message itself. Another way of saying this is that there should be no out-of-band communications occurring.

This is one of those 'simpler, but more complicated' things. It makes processing the message much more simple, because you have all of the information that you need in one place. It makes creating the message a bit more complicated, as you have to actually say what you mean, and *everything* that you mean.

Are media types out of band?

Out of band doesn't mean 'human processable.' Consider this: You're aware that we're speaking in English, because you've identified the words as such. And you know that English (roughly speaking ;)) follows the words that we use in a dictionary. There also may be extensions and such, but the point is, with a few books, you can reasonably understand everything that's said in such a message.

By the same token, when the start of my message says "HTTP 1.1," you know that the message will be governed by the HTTP 1.1 spec. And that when it says "Content-type: application/json" that you can use the JSON spec to interpret it. But if you had to *guess* that a particular message was JSON, or *guess* how to process JSON, or remember our previous conversation where I told you how to process JSON, that'd be much harder, no?

How can I tell if I'm using out-of-band information?

Look at the message headers. Is there anything that you're relying upon a user to know that's not contained there somewhere? For example:

```
Content-type: application/json
{"post":{"title":"hello, world"}}
```

If your client says 'post' somewhere, stop! The JSON RFC says nothing about a 'post' or what that is. You're using out-of-band information about details inside the message body.

You can fix this by either providing a `Content-type` that includes information about posts, or you can use the `profile` link relation to add semantics on top of the `application/json` format.

Sources Cited

- Wikipedia: Out of Band
- Roy Fielding: Architectural Styles and the Design of Network-based Software Architectures, Section 5.1.3: Stateless.

Partial application of Hypermedia

While most of this work is about understanding Hypermedia architecture and principles, unless you're building a new system from scratch, and you can convince your team, it's not likely that you'll be working with a pure hypermedia system. That doesn't mean that what you've learned is useless! As a matter of fact, careful application of portions of hypermedia principles can yield some benefits.

The real message here is "understand your system, and what certain design elements introduce into it."

Hyperglyph

I have a friend who goes by the name tef. He's been studying hypermedia for a while now, and needed to build a system for work. It primarily is a job queue, where lots of workers need to make requests over the network to do things.

The most straightforward way of building network systems is often RPC. Make a method call over the internet, what could be more straightforward? But RPC has a few flaws: you can't take advantage of existing HTTP tooling, such as caches. You pass service documents around, and if you want to update the server, clients need to be updated too. Essentially, what you get in ease of use is gained only by giving up flexibility. So tef decided to try to build an RPC system but with hypermedia principles underlying it. Those choices led to gaining a few benefits over traditional RPC, even if it's not a full hypermedia system.

What's it do?

Okay, so, here's an example of Hyperglyph client code:

```
require "hyperglyph"
s = Hyperlyph.get(ARGV[0])
q= s.Queue('a queue')
q.push('some text')
p q.pop()
```

And the server:

```
require "glyph"
require "rack"

$queues = {}

class Service < Glyph::Router
  class Queue < Glyph::Resource
    def initialize(name)
      @name = name
    end

    def push(a)
      if not $queues[@name]
        $queues[@name] = []
      end
      $queues[@name].push(a)
    end

    def pop
      $queues[@name].pop
    end
  end
end

Rack::Handler::WEBrick.run(Service.new, :Port => 12344)
```

As you can see, we have a `Queue` class defined on our server, and it inherits from `Hyperglyph::Resource`. This makes all the magic happen. Our `Queue` defines all the Ruby code needed to make our stuff work: a simple push and pop. On the client, we create a client object with Hyperglyph, and ask it to make us a `Queue`. That `Queue` then transparently uses the network to be able to understand `push` and `pop`.

The intermediate representation

Hyperglyph uses its own custom media type, which it calls an 'encoding.' It is a hypermedia type, supporting links, forms, and embeds. It's a binary format, which means that it's sorta hard to just look at and say 'ah, that's how it works!' For example:

```
[]
```

This is from the Python REPL. There are client and server implementations of Glyph in both Ruby and Python.

Let's check out that first form:

```
Xu4:form;Du6:method;u4:POST;u3:url;u4:/url;u6:values;LXu5:input;
Du4:name;u3:one;;N;;Xu5:input;Du4:name;u3:two;;N;;;;N;;
```

Giving it some formatting:

```
X
  u4:form;
  D
    u6:method;  u4:POST;
    u3:url; u4:/url;
    u6:values;  L
          X
            u5:input;
            D
              u4:name;   u3:one;
            ;
            N;
          ;
          X
            u5:input;
            D
              u4:name;   u3:two;
            ;
            N;
          ;
        ;
  ;
  N;
;
```

I added whitespace so that you can see the encoding stuff. Each letter is a type, so for example, 'X' means "You're about to see a hypermedia type", then 'u' means "a unicode (utf-8) string". Strings are prefixed with their length, '4:' means 'a four byte utf-8 string is coming up', then the four bytes 'form', followed by a semi-colon.

Communicating via the type

Here's the magic you need to understand Hyperglyph: what's the actual difference between this Ruby and this HTML?

```
def Foo.bar(num, str)
end

Foo.bar(1, "hello")
```

and

```
<form action="/Foo/bar" method="POST">
<input type="text" name="num"/>
<input type="text" name="str"/>
</form>

POST /Foo/bar
num=1&str="hello"
```

Not much, right? So you can see, a Hyperglyph client hits a server, and grabs a document that defines what objects the server knows how to handle. We can then dynamically make Ruby objects that correspond to those definitions that the server knows about, and directly translate method calls into HTTP requests. Neat!

What's the benefit?

While the current method calls use 'POST,' if you knew that a method was cachable, you could use GET instead and take advantage of intermediaries. It's pretty simple to do load balancing, etc.

The biggest benefit that I see is flexibility with updating. This goes hand-in-hand with my versioning article; you don't need to version your service to enable evolveability. If your server adds new capabilities, newer clients can simply use them, yet older clients will continue to work. If your client code is written flexibly enough, your old clients may even be able to tolerate changes to things they already understand, but that's harder. The big win is that 'legacy' systems can still keep on working just fine. If it ain't broke, don't fix it.

Does this actually work?

tef has been using Hyperglyph in production since february 2012, and is developing the media type and client/server code based on his actual in-production usage.

So, 'yes.'

Sources Cited

- Hyperglyph on GitHub
- A presentation by tef about Glyph
- The Hyperglyph media type

Get A Job! - An Introduction

Almost every service has the internal need for a job queue, and some even expose a queue externally, too. In the interest of more code, less theory, I figured I'd build a hypermedia-enabled job queue to demonstrate some of what we've learned.

Super duper edge

To make this extra fun, I decided to use super extra edge versions of *everything*. Screw legacy, we're going for broke! I'm building this service exactly as I would if I were building it today. Because I did. ;) This doesn't mean that it's not applicable to older versions of these things. But you have to know what you want to build *to* in order to migrate old things to new, so I think discussing the cutting edge is absolutely valuable.

There's also a bigger point here: there's currently a *lot* of work going into Rails 4 to make it awesome for APIs. I'd like to show that off. Two of these components were included in Rails, but then yanked out because some of Rails core doesn't think they're important enough. But once you see their power, I think you'll want to use them too, and maybe we can change those members' minds.

With that said, here's our components:

Edge Rails

Mostly this is necessary to use the new Queue API, but also just because we can. ;)

One trick with using edge Rails: We don't just want to use edge Rails, we want to generate our app with it. This means building and installing the gems ourselves. It's pretty easy:

```
$ cd ~/src # or wherever
$ git clone https://github.com/rails/rails.git
$ cd rails
```

```
$ rake install
$ gem build rails.gemspec # I had to build the meta-gem too...
$ gem install rails-4.0.0.beta.gem # ... and install it
```

You might get some odd warning or two, but `gem list rails` should show `4.0.0.beta` when you're done. If so, you're good to go!

Rails::API

Here's the first component that was yanked from Rails. The Rails::API gem is a custom controller stack that makes serving up JSON super awesome. It does this by essentially removing Rails' defaults that assume we're serving up HTML, as well as removing middlewares that aren't useful for backend servers.

Why was it pulled? Rails Core would feel more comfortable if it had some more tests, was used more, and demonstrated that people want to actually use it before it moves into core. I think it's really important that Rails gets this use-case right, so we're totally using it.

It's really easy to use. You basically just

```
gem install rails-api
rails-api new my_api
```

Bam! You have a Rails project named `my_api`. Of course, I've done this part for you already, just clone the project repository down:

```
$ git clone https://github.com/steveklabnik/get_a_job.git
```

So what's it actually *do*? Well, for one thing, your controllers change a bit:

```
class ApplicationController < ActionController::API
```

This controller has a custom set of middlewares. No frills. Just the JSON, please!

ActiveModel::Serializers

ActiveModel::Serializers provides an object-oriented way to generate JSON from your ActiveModels. It's not perfect yet, but it's pretty damn good. This was also put in Rails Core, and then reverted. JBuilder was added instead.

JBuilder is stupid.

I don't say this lightly. The biggest issue is this: JBuilder is a DSL for *generating* JSON from a model. It doesn't give you anything for the conversion back to a model. It also basically makes you re-build the structure of each kind of JSON by hand, every time. This feels very un-Rails-y. By default, it should just give you something awesome.

It's also pretty ugly. Maybe I'm just becoming a curmudgeon, but `instance_eval` based DSLs just don't impress me any more. ;)

Whatever, anyway, we have to bundle my version, because they define a hard dependency on Rails 3. Luckily, Bundler makes this easy, so you do nothing special.

Anyway, ActiveModel::Serializers works like this: When you generate a model, it makes a serializer in `app/serializers`. They look like this:

```
class JobSerializer < ActiveModel::Serializer
  attributes :status, :links
end
```

You wrap your model in a serializer:

```
render :json => JobSerializer.new(Job.first)
```

And it spits out everything you need. You can declare associations, check out what the `current_user` has permissions for, all kinds of stuff. Super cool. Nice and OO.

Rails Queue

An awesome feature in Rails 4, and kinda central to what we're trying to do. Basically, Rails comes with a default, super simple, in memory job queue. Not designed for production, this provides a way for Real Job Queue projects to hook into Rails in a standardized way. And while in dev or test mode, we can just use the in-memory one. Brilliant!

Basically, it's super simple:

```
Rails.queue.push job
```

Where `job` is anything that responds to `run`. The queue pops stuff off and runs them. Nifty!

Profile Link Relations

There is a draft RFC for a link relation named 'profile.' It makes generic a concept that was included in HTML already, META.profile. Here's the abstract from the draft:

```
This specification defines the 'profile' link relation type that
allows resource representations to indicate that they are following
one or more profiles.  A profile is defined to not alter the
semantics of the resource representation itself, but to allow clients
to learn about additional semantics (constraints, conventions,
extensions) that are associated with the resource representation, in
addition to those defined by the media type and possibly other
mechanisms.
```

You've seen this before in the ALPS Microblogging example with XHTML based profiles. We're going to follow the draft (which is super simple) and use it with our JSON via a Link header, since `application/json` doesn't have linking semantics.

Basic Usage

You can check out the README for instructions on getting it up and running, as well as trying it out.

Wrap-up

That's the overview of all of the components. In another part of this series, we'll discuss details about how the queue actually works, and possibly enhance it a bit. For now, play around with these cutting-edge tools and check out how they interact.

Sources Cited

- Get A Job! on GitHub
- Rails::API on GitHub
- ActiveModel::Serializers on GitHub
- JBuilder on GitHub
- Rails Queue implementation

- Profile Link Relation Draft
- HTML profile
- Profiles - mnot's Blog
- Link header

Get a Job: The Basics

While I tried to make "Get a Job" live on the edge with all of those components, I also let it stick to Rails conventions in many ways. Let's talk about the basic way that things work.

If you looked at the code before, you should look at it at revision 8f7f3800, as I've added some things.

The business needs

Building a job queue is pretty simple: We want to be able to submit a job, give that job some parameters, and have a way to find out when the job is finished and what the result is. The trick is the gap in time between the submission and the result. There are two basic options for handing this kind of case: push and poll.

Push

Push notifications are interesting, but difficult. If we wanted to make our job push updates to us, we'd be submitting some kind of callback URI to the job, and when it was finished, the server would send a request (probably a POST) to that URI.

In order to have the server do this, we need some sort of URI associated with ourselves. This means that push is basically dead in the water for most applications, traditionally. There are ways of getting around this, of course, but without some trickery, it doesn't really work well.

Pushing is also really, really hard to scale. On the notification, our server becomes a client and we become a server. This inversion of the relationship is the other major reason that push is awkward. Doesn't mean it's wrong, but it is most certainly awkward.

Poll

Polling is the default way that web applications handle updates that happen outside of a single request/response cycle. Basically, every n seconds, we make a request to the server, and check if there's been an update.

Polling is nice for a few reasons. Primarily, it preserves the relationship dynamics of client/server, which means we're working with the grain of HTTP. All of our previous analysis about scaling still applies here. Caching plays a key role in this relationship: without the proper caching, polling is crazy inefficient.

In addition, polling warms up caches. GET requests are amazingly cache-able, and our first request will set up that cache properly, so subsequent requests by others will also be able to take advantage of the cached response. This won't really apply with our queue, exactly, but is generally useful.

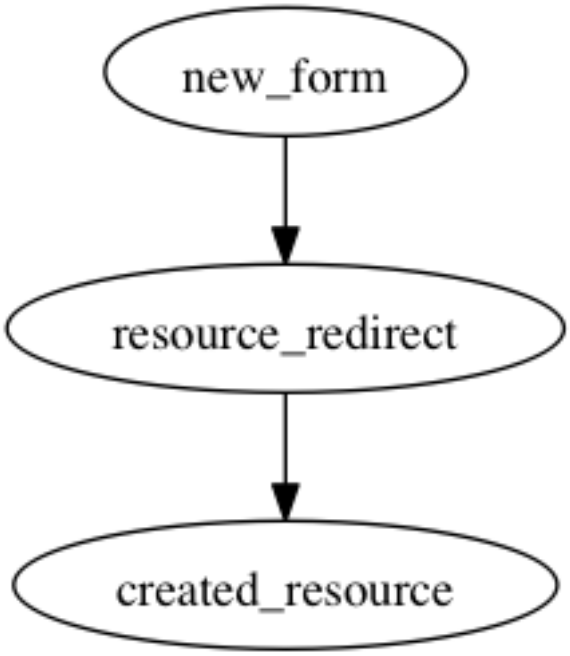
PuSH

Google came up with a really interesting architecture that combines push and poll. It's called "PubSubHubbub," and it works by introducing a third party, the 'hub.' You pull down an ATOM feed, and inside, there's a link to the hub's address. You register yourself with the hub, and tell it you'd like updates. When the publisher adds new content, they notify the hub, and the hub then pushes the notification out to everyone who's subscribed.

This allows you to have one beefy hub that handles the workload of sending out tons of notifications, while allowing the publisher and each subscriber to have much less load.

State machine

In this particular case, the most basic state machine doesn't really help us too much:



We get a form of some kind, we POST to a URI to create the job, it redirects us to the place where we can poll, and then we poll until it's finished.

I think it gets better with another arrow:

Anyway, this is what we need to build.

In action

Our job is to build this out. It's pretty easy, overall. Rails supports the routing pretty easily:

```
resources :jobs, except: [:edit]
```

We're not really using the edit action, so we can exclude it if we feel like it. It's a generally good idea to exclude URIs you don't use.

The controller is also very simple. There are only two interesting bits: in our `create`, we push the job onto Rails' queue:

```
# quite possibly could be in the model in after_create
Rails.queue.push job
```

And we also set up our responses to send the right media type:

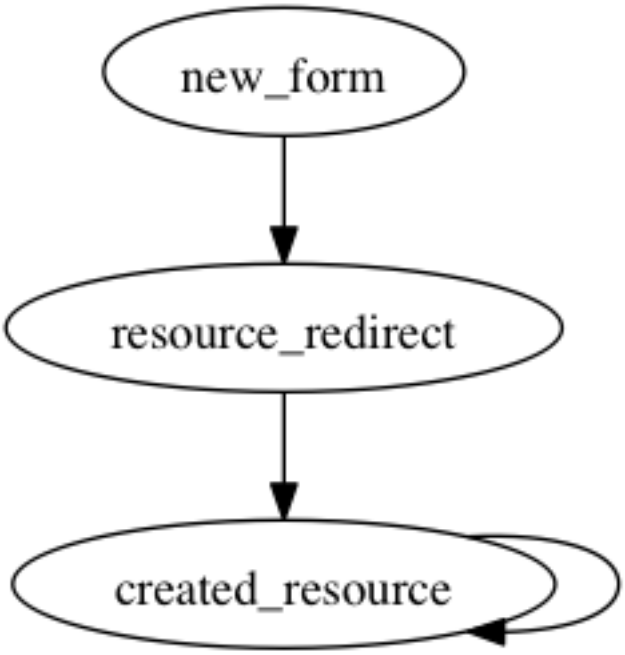


Figure 9: Second state machine

```
before_filter :set_profile

def set_profile
  response.headers["Link"] = %Q{<#{profile_url}>; rel="profile"}
end
```

We're providing a particular flavor of JSON via a profile. So we have to notify people of where our profile documentation is. Note that this uses the `_url` helper, so it's a full URI, complete with hostname and everything.

We're providing 'views' via Serializers:

```
class JobSerializer < ActiveModel::Serializer
  attributes :status, :links, :number_one, :number_two

  def initialize(object, options={})
    @object, @options = object, options
  end

  def serializable_hash
    hash = attributes
    if attributes[:status] == "finished"
      hash = hash.merge(:answer => @object.answer)
    end
    hash
  end
end
```

When this gets rendered, it uses the `serializable_hash` method to determine what attributes get included in the representation. We only want to include our answer if we've got a `finished` status. This gives us representations that look like this:

```
{"job":{"number_one":1,"number_two":2,"status":"in_progress",
"links":[{"href":"/jobs/1","rel":"self"},{"href":"/jobs","rel":"index"}]}}

{"job":{"number_one":1,"number_two":2,"answer":3,"status":"finished",
"links":[{"href":"/jobs/1","rel":"self"},{"href":"/jobs","rel":"index"}]}}
```

When our job is new, they also give us a convenient template:

```
{"job":{"number_one":null,"number_two":null,"status":null,
"links":[{"href":"/jobs","rel":"index"}]}}
```

I've found that using Serializers is much nicer than traditional templates, simply because I can interact with them with code. Really, outputting something to JSON is more of an implementation detail; I care what's contained in the response, but minimally about how it's formatted. Some things are important, like the 'links' array, but Serializers can handle it and just generally feel much nicer. Having a Ruby class for presentational concerns is also an all-around good thing, too.

We can interact with this API via any client by simply sending requests. Here's some examples with cURL:

```
$ curl -i -H "Accept: application/json" http://localhost:3000
HTTP/1.1 200 OK
Link: <http://localhost:3000/profile>; rel="profile"
Content-Type: application/json; charset=utf-8
Etag: "c177410112a0887129017c8272c987e6"
Cache-Control: max-age=0, private, must-revalidate
X-Request-Id: fa0b08e8-2e77-452c-851b-91c22c641ccb
X-Runtime: 0.003510
Content-Length: 64
Server: WEBrick/1.3.1 (Ruby/1.9.3/2012-02-16)
Date: Tue, 08 May 2012 20:51:24 GMT
Connection: Keep-Alive

{"job":{"number_one":null,"number_two":null,"status":null,
"links":[{"href":"/jobs","rel":"index"}]}}
```

This lets us know we need a status attribute, two numbers, and the href to push it to:

```
$ curl -i -H "Accept: application/json" -X POST \
    -d "job[status]=in_progress" \
    -d "job[number_one]=1" \
    -d "job[number_two]=2" \
    http://localhost:3000/jobs
HTTP/1.1 302 Found
Location: http://localhost:3000/jobs/1
Link: <http://localhost:3000/profile>; rel="profile"
Content-Type: text/html; charset=utf-8
Cache-Control: no-cache
X-Request-Id: f3709bd8-961a-4091-b3cd-4841a08b5a85
X-Runtime: 0.012685
Content-Length: 94
Server: WEBrick/1.3.1 (Ruby/1.9.3/2012-02-16)
Date: Tue, 08 May 2012 20:00:53 GMT
Connection: Keep-Alive
```

```
<html><body>You are being
<a href="http://localhost:3000/jobs/1">redirected</a>.
</body></html>
```

Then you GET it:

```
$ curl -i -H "Accept: application/json" http://localhost:3000/jobs/1
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Link: <http://localhost:3000/profile>; rel="profile"
Etag: "1ab24154f124af6d36b0c65e126cabeb"
Cache-Control: max-age=0, private, must-revalidate
X-Request-Id: 10c304ed-1f9f-4362-895e-a896cf6f4e0d
X-Runtime: 0.004891
Content-Length: 47
Server: WEBrick/1.3.1 (Ruby/1.9.3/2012-02-16)
Date: Tue, 08 May 2012 20:08:28 GMT
Connection: Keep-Alive

{"job":{"number_one":1,"number_two":2,"status":"in_progress",
"links":[{"href":"/jobs/1","rel":"self"},{"href":"/jobs","rel":"index"}]}}
```

After ten seconds, you should see the status switch, and an answer appear:

```
$ curl -i -H "Accept: application/json" http://localhost:3000/jobs/1
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Link: <http://localhost:3000/profile>; rel="profile"
Etag: "031eb7931004812303c6003c763f0272"
Cache-Control: max-age=0, private, must-revalidate
X-Request-Id: 1ae1f3d1-9644-4ad3-9782-4118e45fc741
X-Runtime: 0.002961
Content-Length: 44
Server: WEBrick/1.3.1 (Ruby/1.9.3/2012-02-16)
Date: Tue, 08 May 2012 20:09:10 GMT
Connection: Keep-Alive

{"job":{"number_one":1,"number_two":2,"answer":3","status":"finished",
"links":[{"href":"/jobs/1","rel":"self"},{"href":"/jobs","rel":"index"}]}}
```

That's it! Nice and simple.

Next time, we're going to talk about several extra things we can do with HTTP to tweak this behavior and improve upon it. The time after that, we're going to build a client in the browser with Ember.js. The time after that, we're going to change our back-end and watch how the client adapts.

Sources Cited

- PubSubHubbub Protocol

The Hypermedia Proxy pattern

Jon Moore's talk at Øredev was one of the biggest moments in my own personal understanding of the hypermedia style. In his talk, he refactored his server a bunch of times, and the clients automatically knew how to deal with the new representations. The first time, the client made 11 requests, and at the end, it made two. I'm here to share with you how this pattern works.

The premise

If we have some sort of collection resource, we may only use some attributes of the items in the collection often, and use some other attributes less often. Wouldn't it be nice if we didn't have to worry about that stuff? We could load only the most often used attributes in the collection view, and use a `self` link to tell the client where to find full information about the item. If we need a lesser used attribute, the client can automatically fetch that data.

Obviously, reducing HTTP requests can make your application more performant. This also demonstrates some of the dynamic-ness of a hypermedia client.

The representation

We're going to use the example of a 'post.' Obviously this is incredibly generic. Maybe I should have used `Foo`. Here's an individual representation:

```
{
  "links":[
   {"rel":"self", "href":"http://<%= request.host %>:<%= request.port %>/status/3"}
  ],
  "id":<%= @id %>,
  "name":"name<%= @id %>"
}
```

That has some erb embedded in it. Now, maybe we use the `id` all the time, but don't often use the `name`. A collection will just be a JSON array of these individual objects. So we may or may not include the `name` attribute in our collection representation.

The sample code

```
class Post
  attr_accessor :self_rel, :id

  def initialize(opts={})
    @id = opts['id']
    @name = opts['name']
   @self_rel = opts['links'].find{|link| link['rel'] == "self"}['href']
  end

  def name
    @name ||= begin
      fetch_data(self_rel)['name']
    end
  end
end

data = fetch_data("http://localhost:9292/")
posts = data.collect {|datum| Post.new(datum) }

posts.each do |post|
  puts post.name
end
```

`fetch_data` is a method that gets the information at that URI and parses it from JSON.

As you can see, the class takes in the list of attributes, and it knows how to grab out the link with a `self rel`. If we don't give it a `name`, it knows how to get it from the `self` link.

If you run this against a representation that contains the `name`, it will make one query. If you run it against one that doesn't, it will make N+1 queries, where N is the number of elements in the collection.

Sources Cited

- Jon Moore at Øredev 2010
- Jon's XHTML representation of the pattern
- Full client and server code

Much ado about PATCH

I often say that Rails was the best and worst thing to happen to REST. It did so much to promote its ideas, but also didn't present them correctly. A while ago, Rails did something to help rectify this, but it's not perfect.

Let's talk about updates.

Ways to update resources in HTTP

If I say the word 'update,' then it seems pretty simple: we all know what an update means. But updates are really hard! There are a few different things that an update can consist of:

1. Creation. A update from '' to 'something' is still an update.
2. Addition. An update from 'nothing' to 'nothing something' is still an update. Note also that creation is a subset of addition.
3. Removal. An update from 'something' to 'some' is still an update.
4. Change. An update from 'something' to 'somefoo' is still an update. This combines an addition and a removal.

All this stuff! So what do we do?

POST

Well, originally, there is POST. POST is, of course, the most generic and least semantic action. We can always use POST to do whatever we want, like an escape hatch. Essentially, there are no rules, and we can make them all up.

This is the most free-form option, and in some ways, the 'easiest.' It certainly places very little burden on the implementor, but it's hard to make clients know exactly what to do with it due to its total generic-ness.

PUT

The old standby. PUT does the dumbest thing that can possibly work: it's a full replacement for the existing resource. This is kinda awesome, in a way: it's very very easy, and handles all of our cases. Just one rule: "What I give you goes here." It also happens to map nicely to file uploads, which was sort of it's original intent. Upload a file, it just replaces what's on disc.

One other nice thing about full replacements is that they're easy to make idempotent, which is a huge benefit of PUT. As anyone who's designed a protocol knows, idempotency is a nice property.

So what's the catch? Well, full replacements are hard. Often, we have things in our representations that make this awkward, like `updated_at` timestamps, for example. While there's nothing saying that we can't modify this afterwards, it eliminates lots of the caching benefits that PUT would give us.

Also, if our representation is huge, but we only need to make one small change, it can be annoying to have to PUT the whole representation. Why not just send the bytes that change?

PATCH

Enter the new kid on the block. PATCH lets you make changes by just sending a diff across the wire! Neat!

… but you have to actually send a diff.

That's the thing with media types: they convey the information about how you work with the data. `application/json` has no concept of a diff. So you end up having these conversations about 'well, how are we gonna tell if this is new or removed or what?' Guess what? That's making a diff type!

This idea of sending a diff isn't optional. As the spec says:

> The PATCH method requests that a set of changes described in the request entity be applied to the resource identified by the Request-URI. The set of changes is represented in a format called a "patch document" identified by a media type.

This 'patch document' is a diff.

So what diff types are there?

Here's a funny thing: the diff that your `git diff` or `svn diff` or `diff` produces doesn't have an official type. We're working on that with a `text/diff` type, but until that comes, what else is there?

There is currently a JSON-patch format being worked on as well.

Aaaaand that's basically it. To my knowledge, there isn't really a totally standardized diff type. This, of course, makes things tricky.

GitHub does serve diffs, but they serve it with `text/plain`.

What to do?

Well, in this case, you follow things as best you can.

Personally, if I was using JSON, I would implement the I-D as it currently exists. For everything else, I would use the output of `diff` served as `text/diff`. This really illustrates how standards are intended to be used: if they don't have an answer, you try to follow the spirit as much as possible. The `diff` utility has been used for *decades*, and while it's not yet fully specced and registered as far as the IANA is concerned, using it means that you will take advantage of the 'standardization' that's come about by using a common tool.

If standards don't make sense, do the best you can!

Sources Cited

- RFC 5789: PATCH method for HTTP
- Edge Rails: PATCH is the new update method
- text/diff on GitHub
- appsawg-json-patch