I can haz HTTP

Consuming and producing HTTP APIs in the Ruby ecosystem sidu ponnappa

kaiwren

@ponnappa

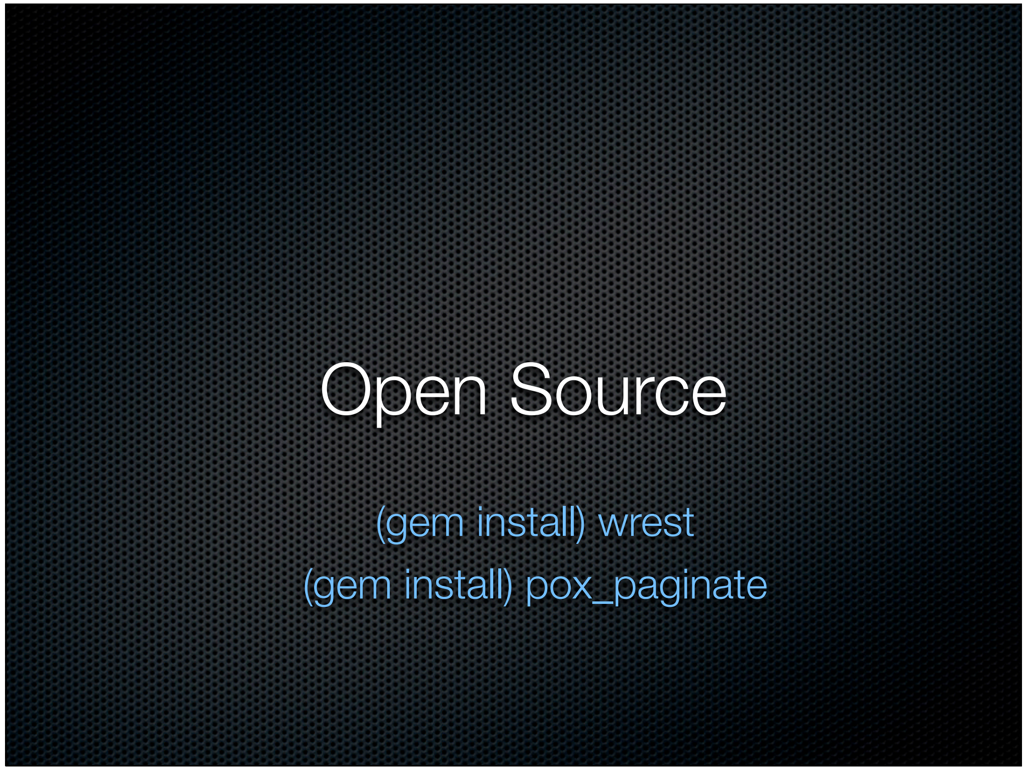
niranjan paranjape

achamian

@niranjan_p

C42 Engineering

http://github.com/c42

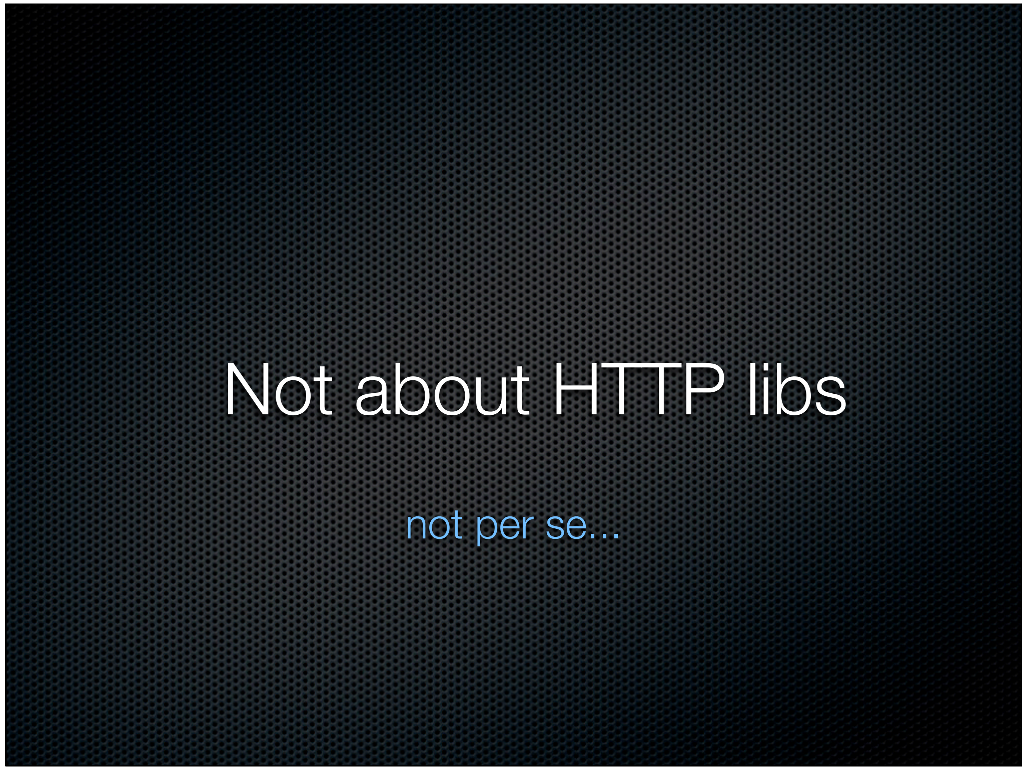


Open Source

(gem install) wrest

(gem install) pox_paginate

Opensource we have done in this area



Not about HTTP libs

not per se...

This talk is not about NetHTTP vs libcurl

More about...

REST

caching

serialisation & deserialisation

authentication

stuff like that...

Producer / Consumer

This talk does have *some* structure; we’ll look at APIs from the perspective of the producer and the consumer. With a little bit left over for stuff that bridges the gap.

Producer

Everyone talks about how easy it is to produce APIs with Sinatra and such like, but lets be honest. There’s an elephant in the room – those of you that freelance know exactly how many requests we get for APIs on Sinatra versus APIs on...

[Handoff]

...Rails. Love it or hate it, Rails has a standard way of producing APIs, and understanding what Rails can or cannot do for you is important. Sinatra does not have these constraints, so you’re free to roll APIs any way you want, so we’ll focus on the framework that does impose significant patterns.

digression

[achamian]

But before we launch into Rails' capabilities, it's important to put things into context first by talking a little about REST

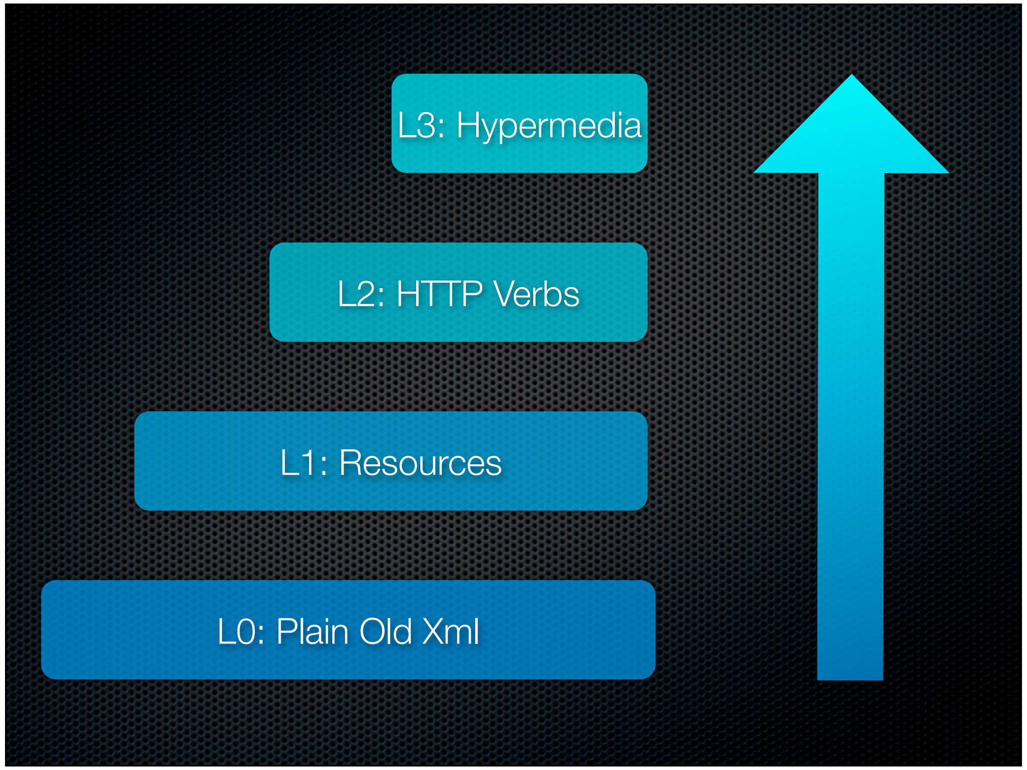
RESTful?

What is a RESTful API?

Web Services for the Real World
RESTful Web Services
O'REILLY
Leonard Richardson & Sam Ruby

Leonard Richardson

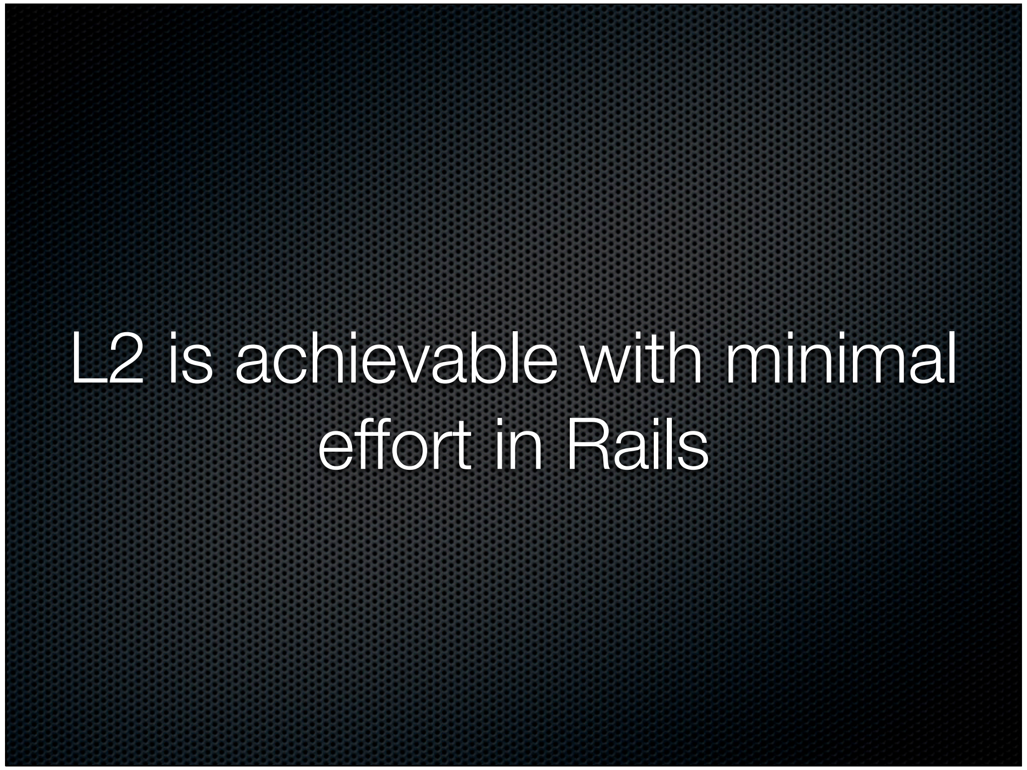
For this we turn to Leonard Richardson who came up with a maturity model that has gotten so famous it has an acronym and looks like...



...this, popularly known as the Richardson Maturity Model, or RMM.
It begins with Level 0, which uses plain old xml (or a similar encoding format) to tunnel requests to a HTTP endpoint. Think RPC. Clearly Rails is better than this.
Next comes Level 1, which introduces the concept of Resources – every URI represents a resource. Easy peasy. Rails makes it easy to do this – but you're free to deviate.
Then comes Level 2, which defines a certain set of constraints around how HTTP verbs are used. Rails violates the semantics by using PUT instead of PATCH – this was fixed on May 07 with Pull request #348, so future releases should be L2 compliant.
Finally, we come to L3 which involves using hypermedia to define the edges of the state graph of an application using hypermedia links. This definitely isn't in Rails. Yet.

reasonable compliance?

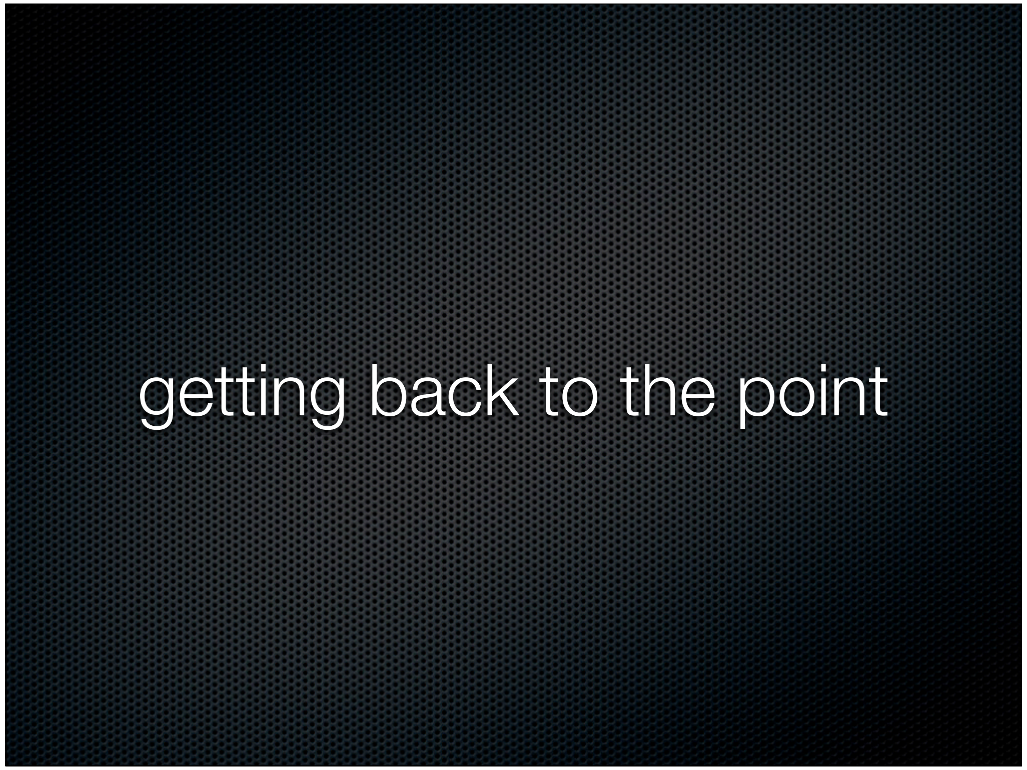
So what level of compliance can we achieve with a minimum of effort?



L2 is achievable with minimal effort in Rails

[Handoff]

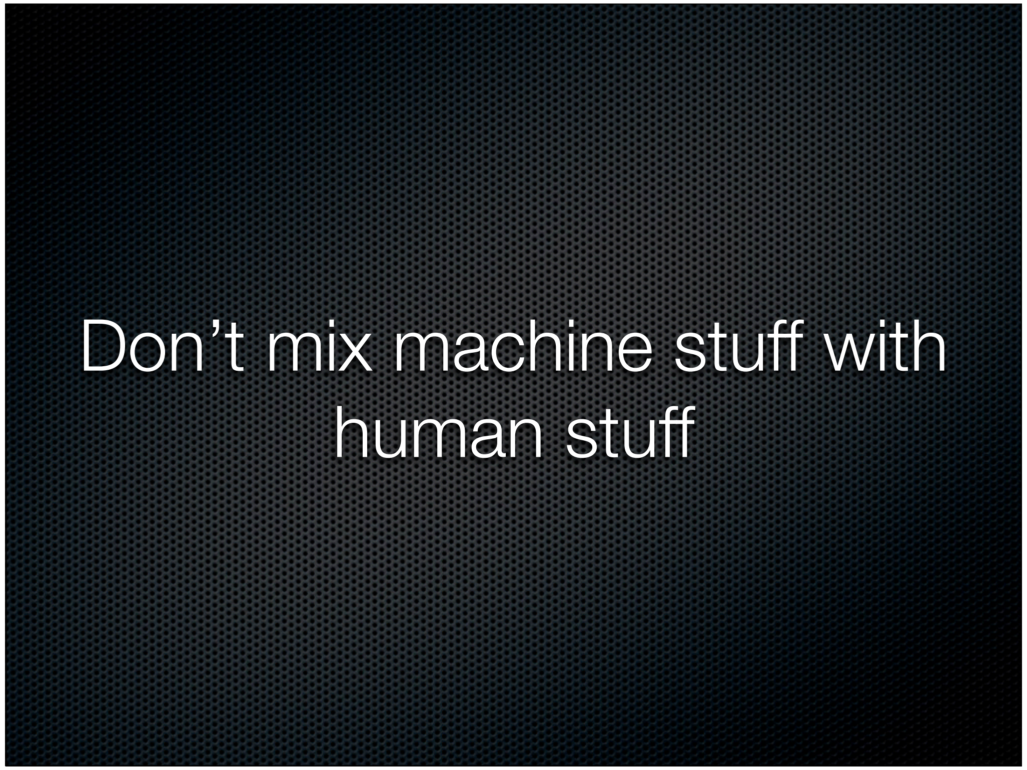
I'd say an L2 is quite easily do-able in rails without too much effort. Now lets talk about how.



getting back to the point

[kaiwren]

So after that little digression, lets go back to what we were originally talking about - how to build APIs on Rails.



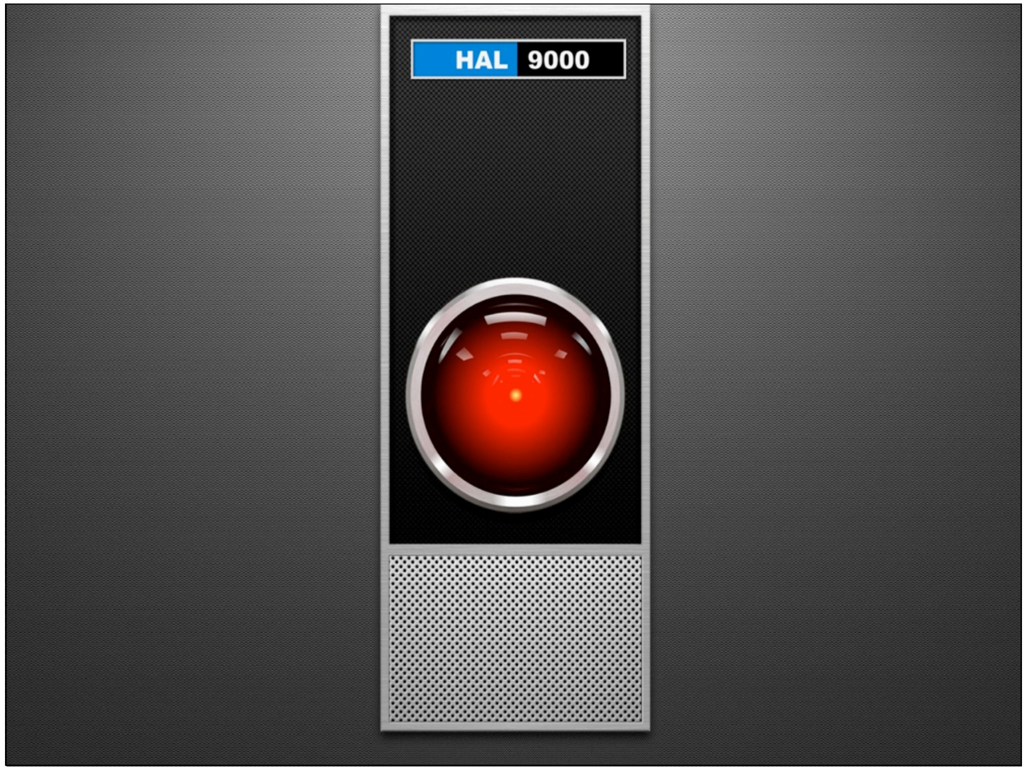
Don’t mix machine stuff with human stuff

[kaiwren]

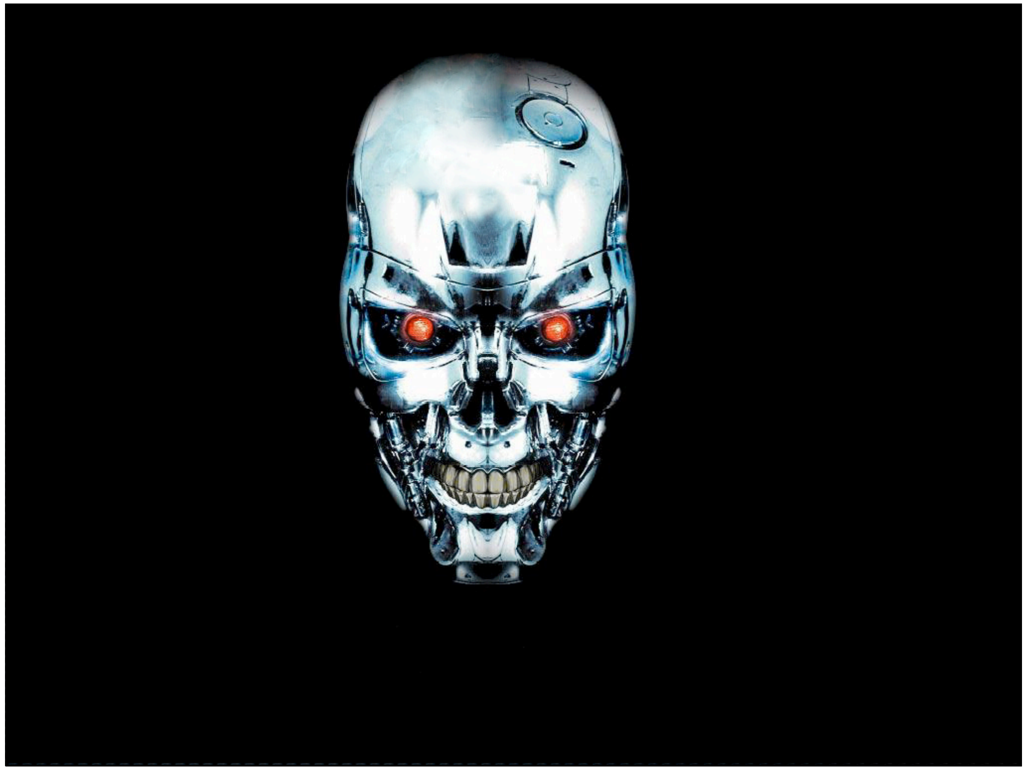
This is one of the easiest thing to forget - APIs are for machines and software, websites are for people. Mixing the two is Not Good.

APIs are for machines

mixing the two causes trouble



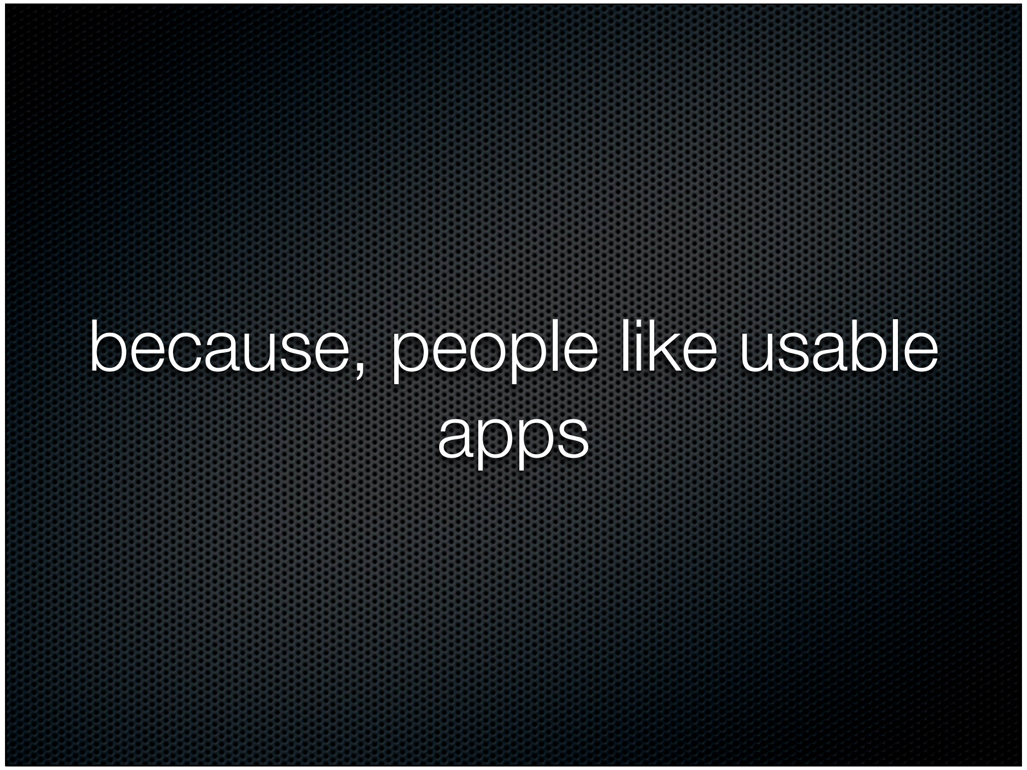
Hollywood has shown us the consequence of mixing people stuff with machine stuff time and again, and it isn’t good.



THE
MATRIX
REVOLUTIONS
WWW.THEMATRIX.COM

basically, doing this causes trouble

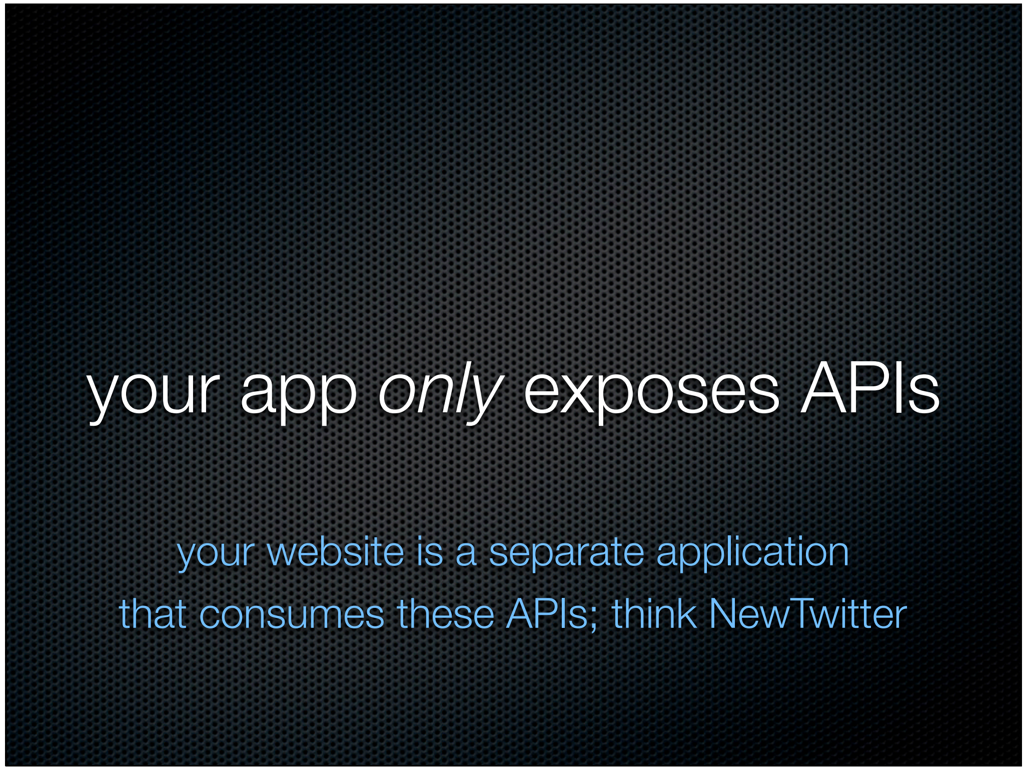
```ruby
def index
  respond_to do |format|
    format.html
    format.json { render :json => Project.all.to_json }
  end
end
```


because, people like usable apps

and REST cares only about state transitions. If you mix these, then remember that this only works in the most trivial of use cases. Beyond that...

separate API controllers from website controllers

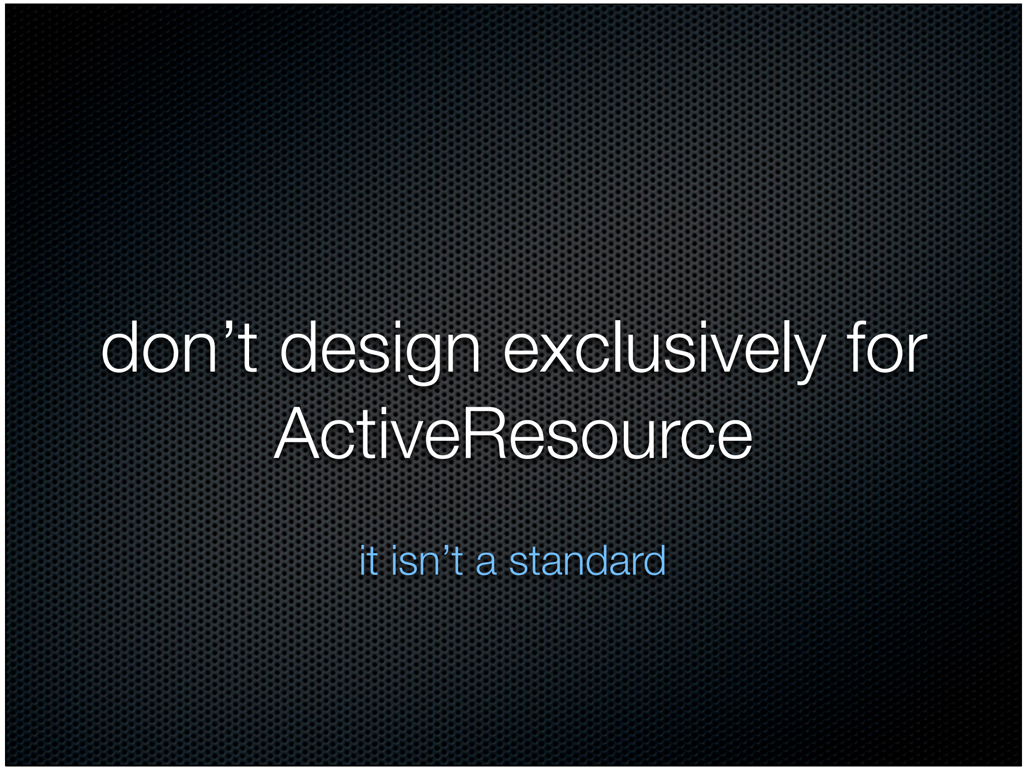
or beyond even that…



your app *only* exposes APIs

your website is a separate application
that consumes these APIs; think NewTwitter

Your app doesn’t have a single HTML page. This is especially useful if you expect 3rd parties to use your APIs – you’re dogfooding your own stuff from day one.

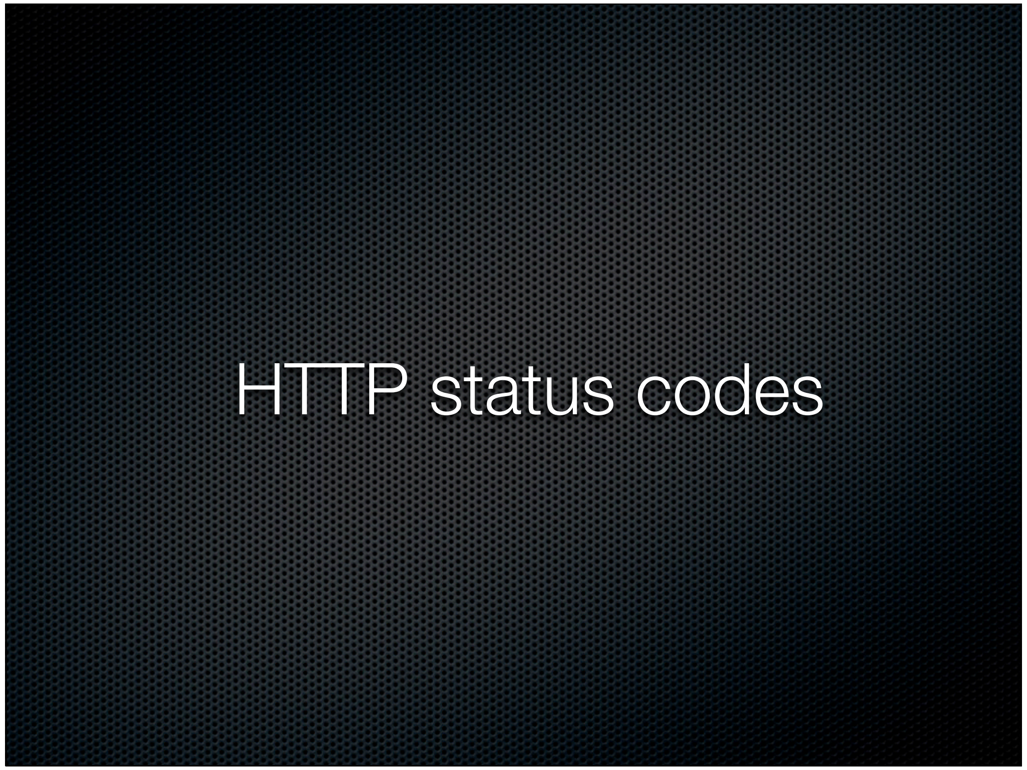


don’t design exclusively for ActiveResource

it isn’t a standard

[Handoff]

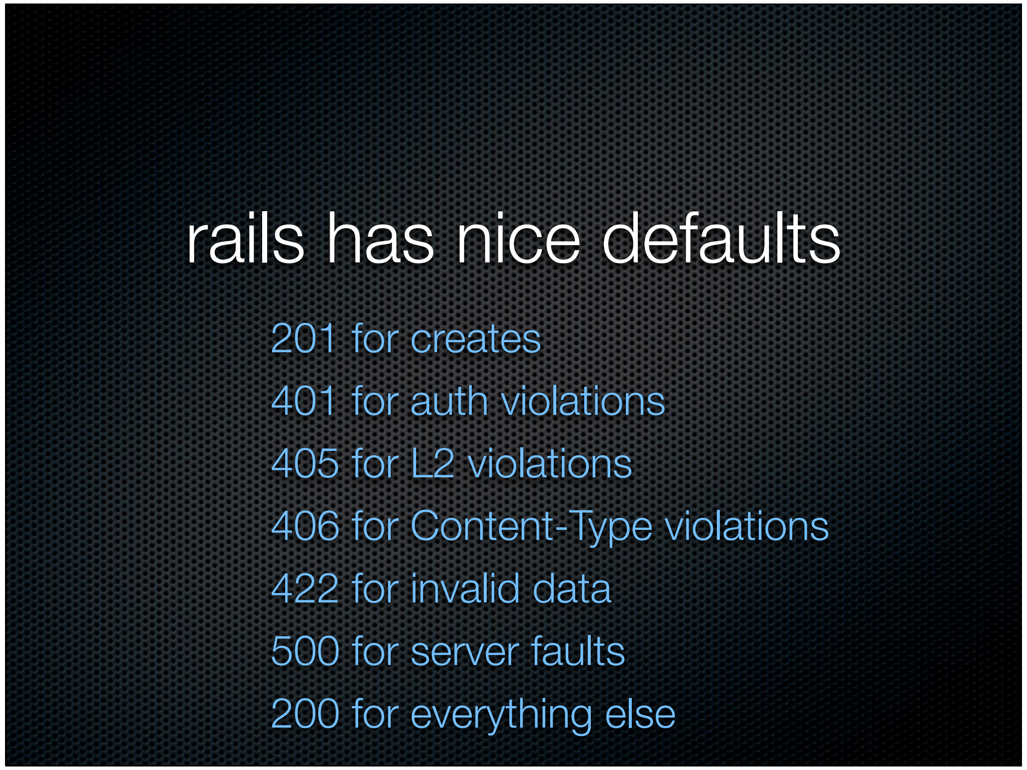
if your api is being consumed by 3rd parties, be aware that they will use a fairly eclectic collection of HTTP libs. What’s convenient/intuitive when using AR may be extremely obscure when using simple HTTP calls.



HTTP status codes

[achamian]

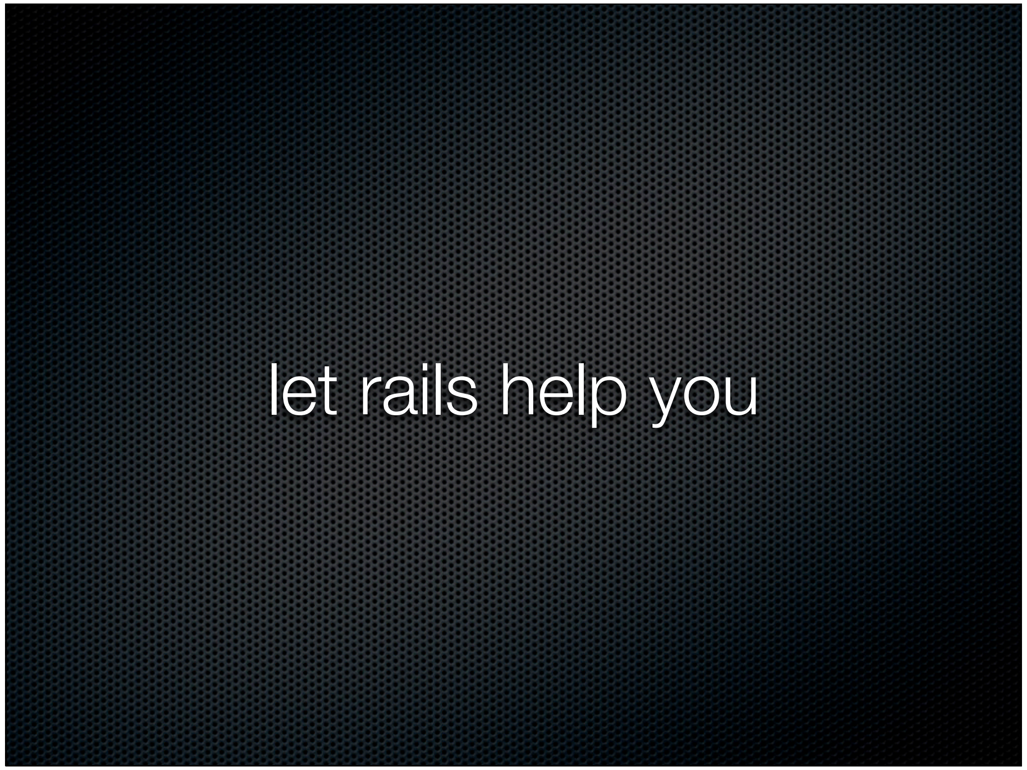
The next thing everybody should keep an eye on when building Rails APIs are the HTTP status codes. There are many of them, and your APIs should respect them.



rails has nice defaults

- 201 for creates
- 401 for auth violations
- 405 for L2 violations
- 406 for Content-Type violations
- 422 for invalid data
- 500 for server faults
- 200 for everything else

The first thing everybody ignores when building Rails APIs are the HTTP status codes. There are many of them, and your APIs should respect them.

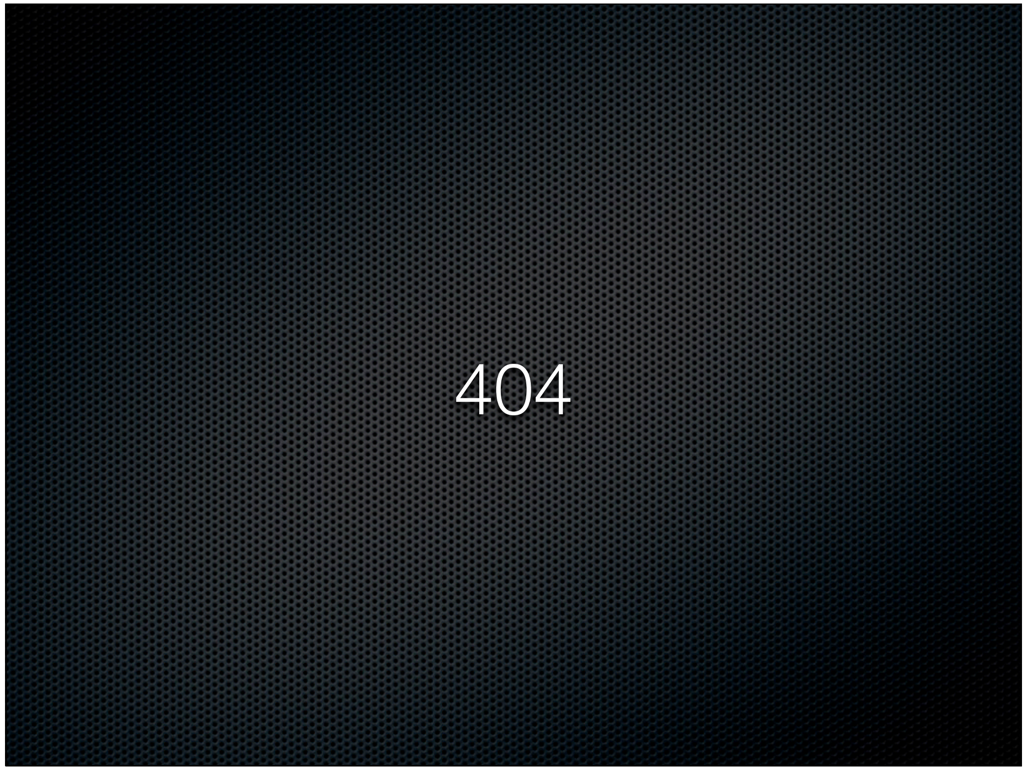


let rails help you

Adhere to these defaults in the code you write. They’re there for a reason. Don’t, for example, return a 200 with an error message in the body – that’s bad form.

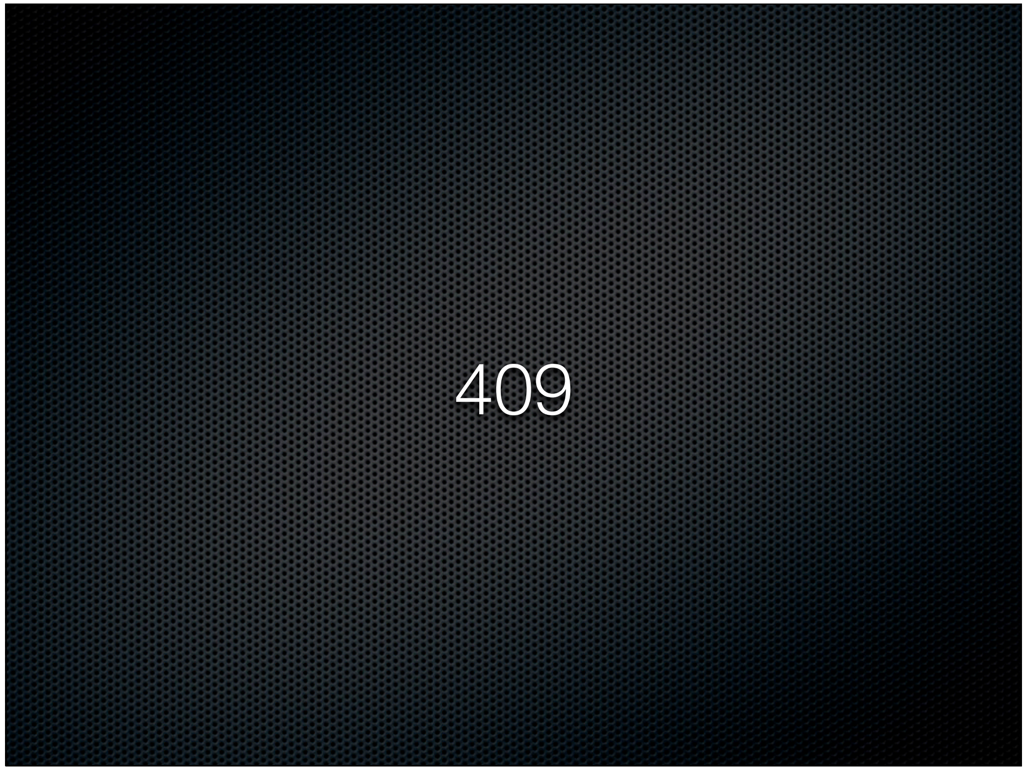
what they missed

There are, however, a couple of codes they deal with weirdly, or not at all

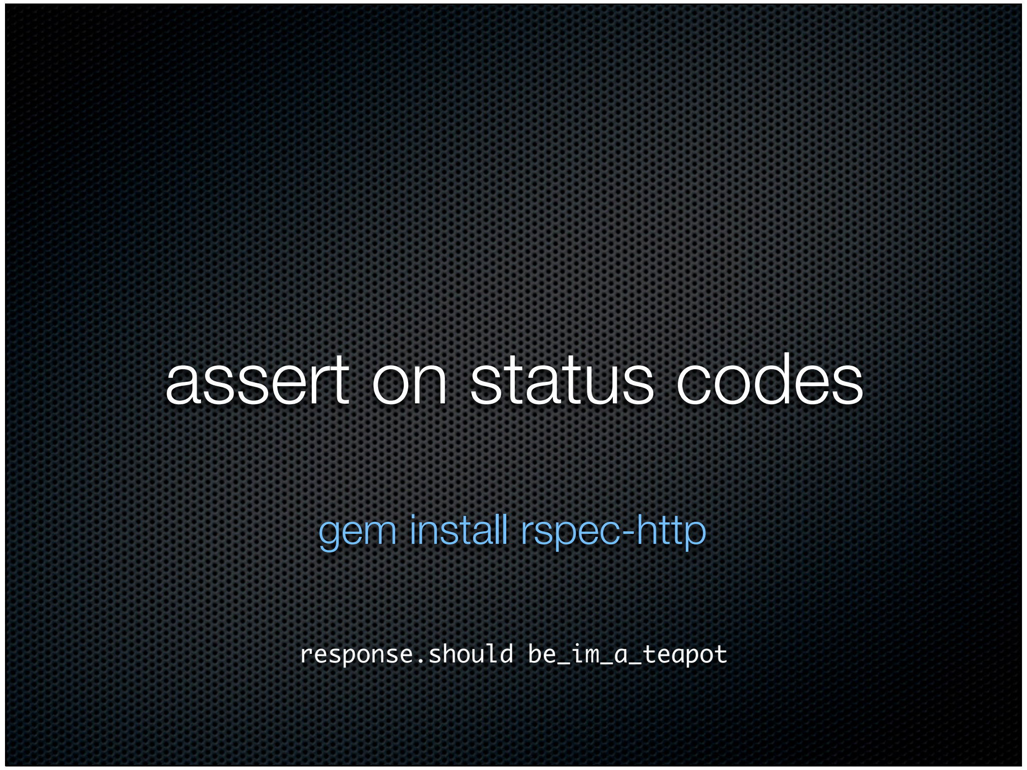


404

The handling of 404s can be strange, driven as it is in certain cases by the ActiveRecord::RecordNotFound exception. Make sure that if a resource is not found, you’re always returning a 404 status code and you write a spec to expect it.


409

Have a unique constraint on a resource identifier which is part of the input and not system assigned? When it's violated, return a 409-Conflict and not a 422-Unprocessable entity.



assert on status codes

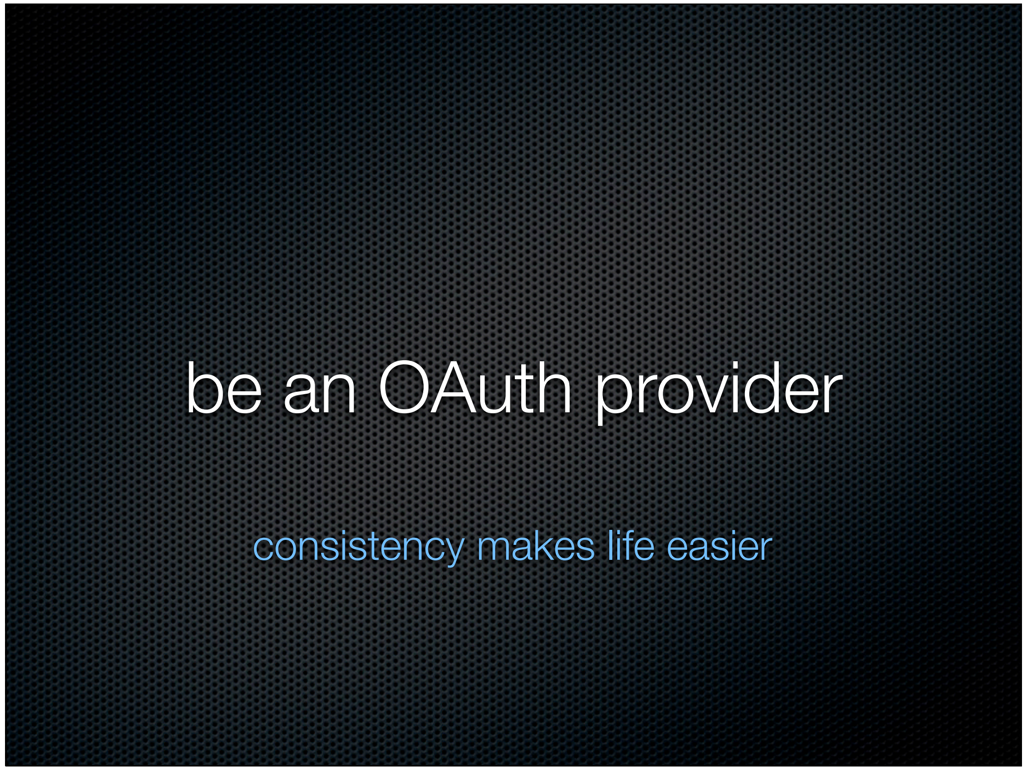
gem install rspec-http

```
response.should be_im_a_teapot
```

It’s important to lock these codes in because they mean a lot to the outside world. When building apis, make sure all your controller specs include assertions on the response code. I’ve got a simple gem that can help if you use rspec. Try it out. That asserts that a response has a code of 418.

Authentication

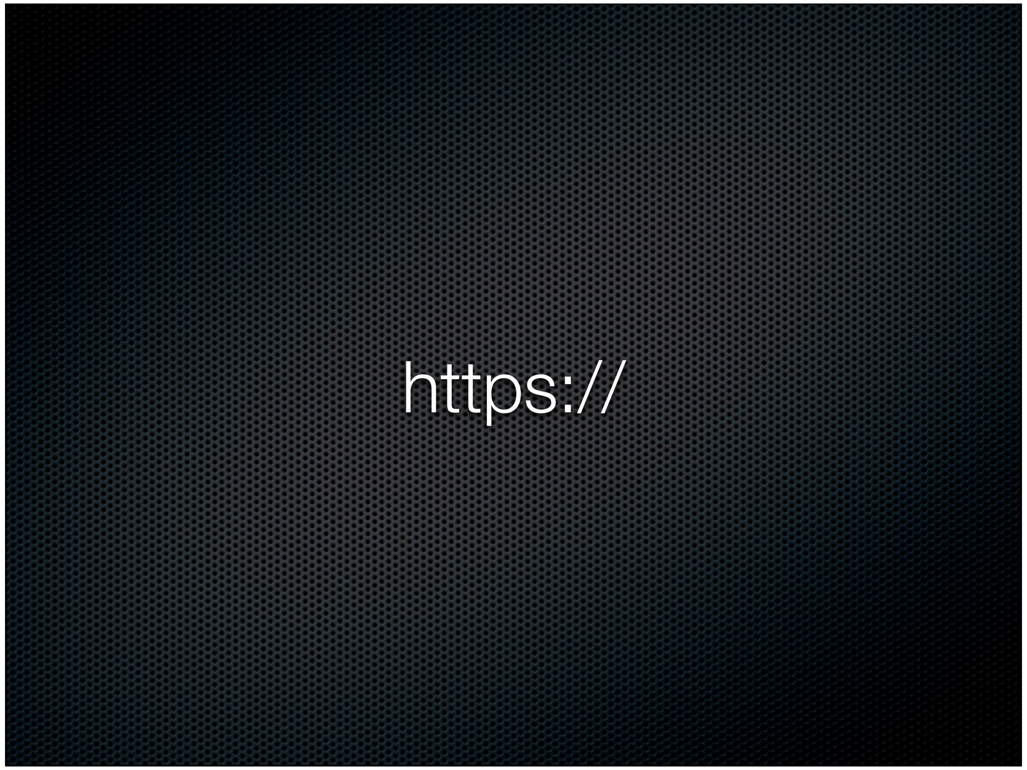
OAuth 2 is nice



be an OAuth provider

consistency makes life easier

This way your application doesn’t need to deal with your own users through one auth mechanism and those that come through fb/twitter/github/sf through another

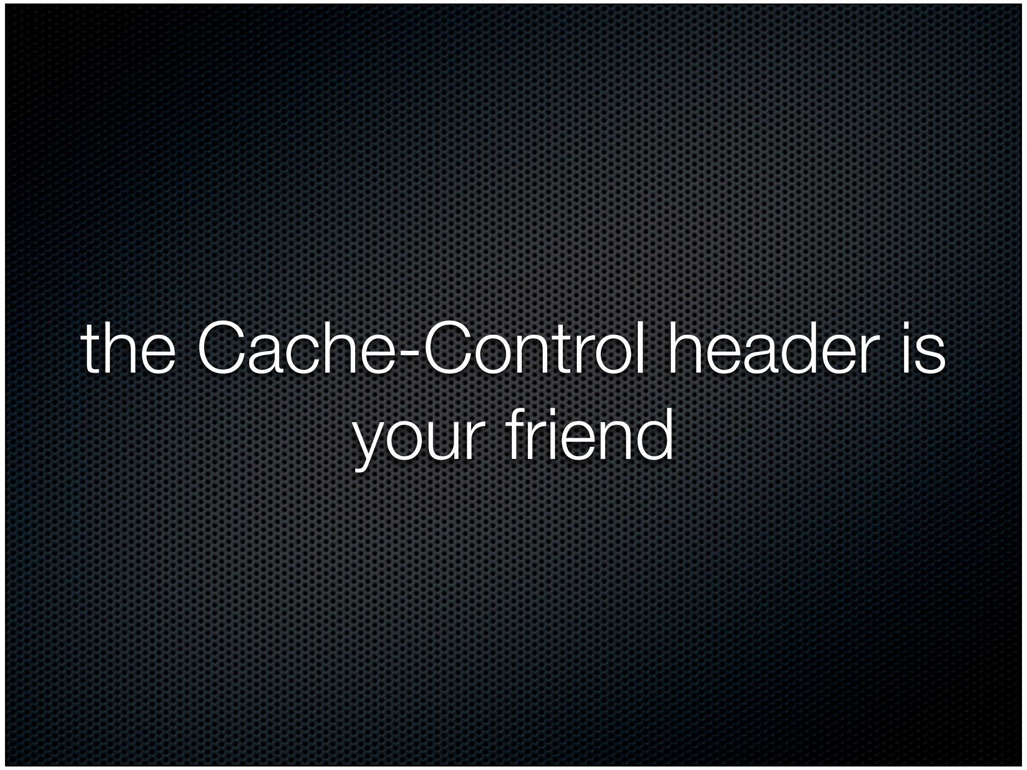


https://

Oauth 2 requires SSL, so do keep that in mind.

Caching

go beyond Rails’ local caches

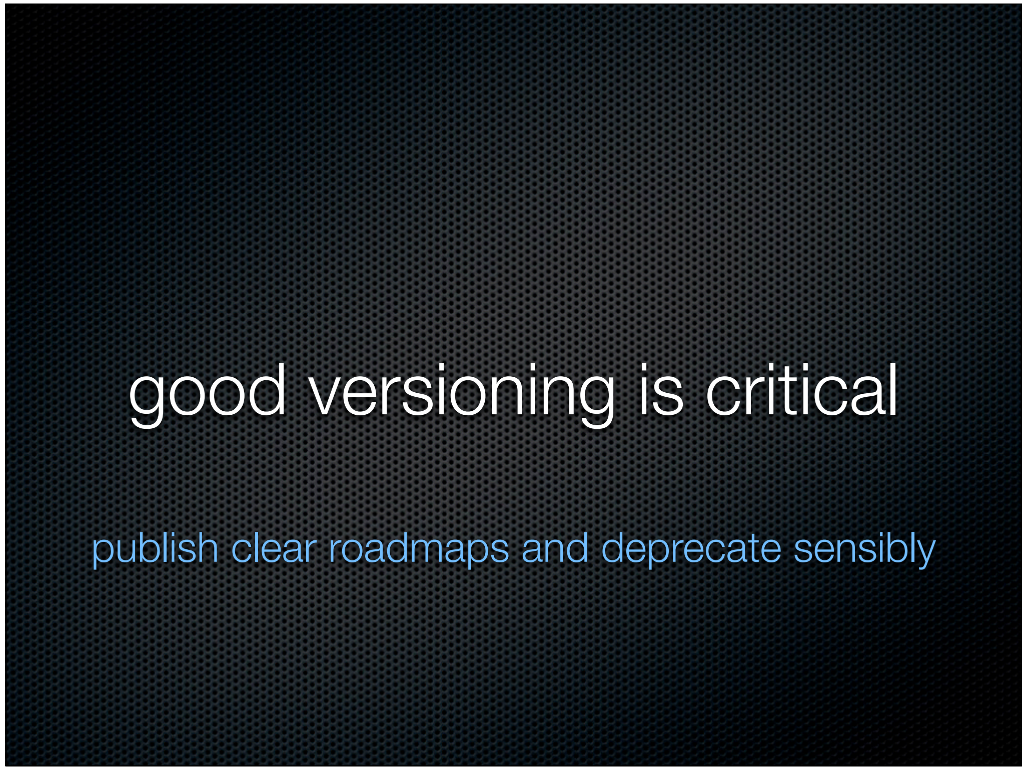


the Cache-Control header is your friend

If you have resources that don’t need to be refreshed as soon as they change, you should use an appropriate expires header to allow intermediate nodes on the network to cache them for a specified period of time. This can massively reduce load on your servers.

Versioning

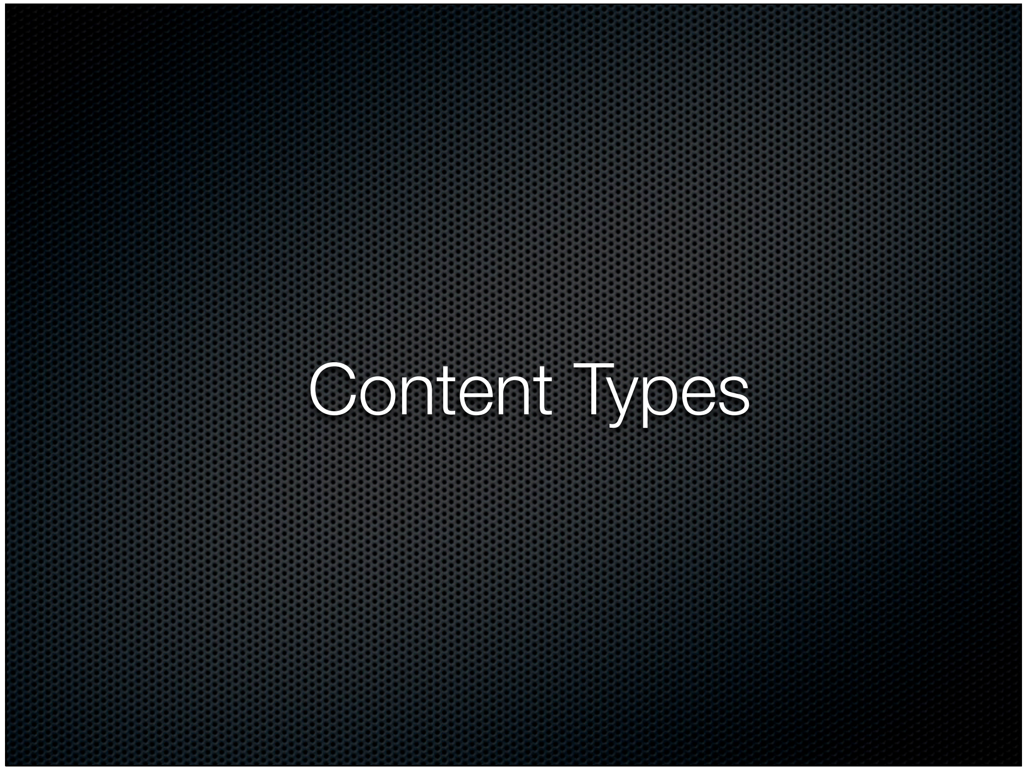
successful APIs change



good versioning is critical

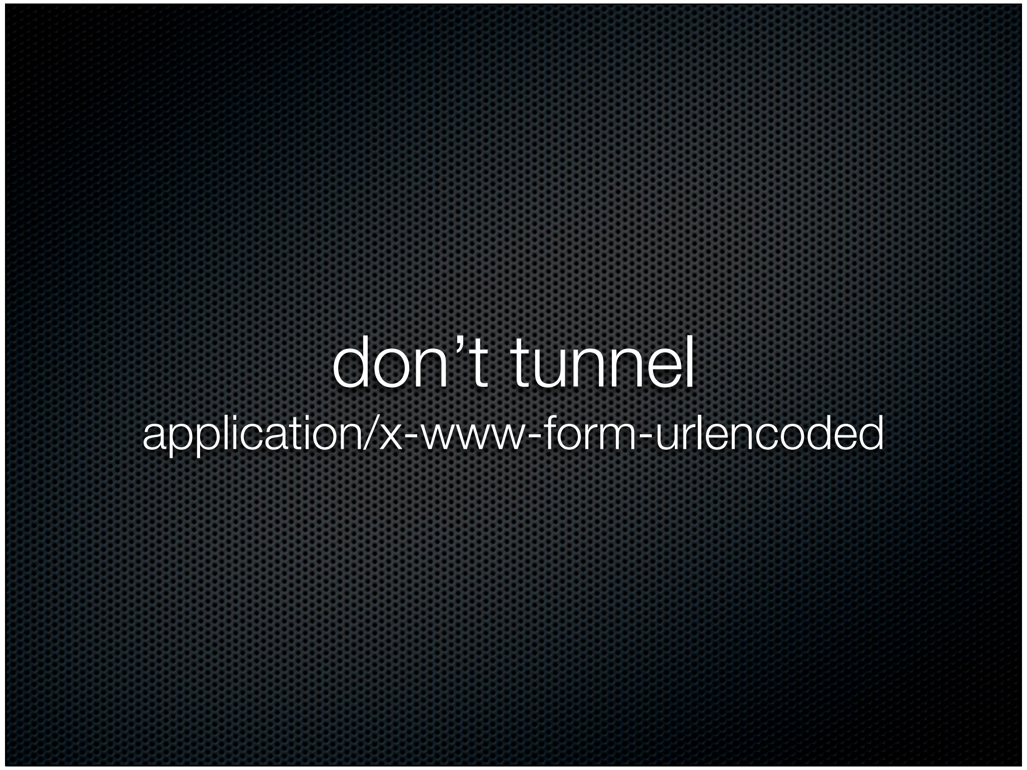
publish clear roadmaps and deprecate sensibly

Just like any library or gem, your users require visibility and stability. Whether you use URL based versioning (by including v1/v2/v3) in the API url, or by headers that specify the version, make sure your versioning roadmap is clear, consistent and sensible. Try maintaining backward compatibility as far as is reasonable.



Content Types

Content Types are rarely done wrong, so I won’t talk about them for too long. There’s only one thing that I’ve observed out here...



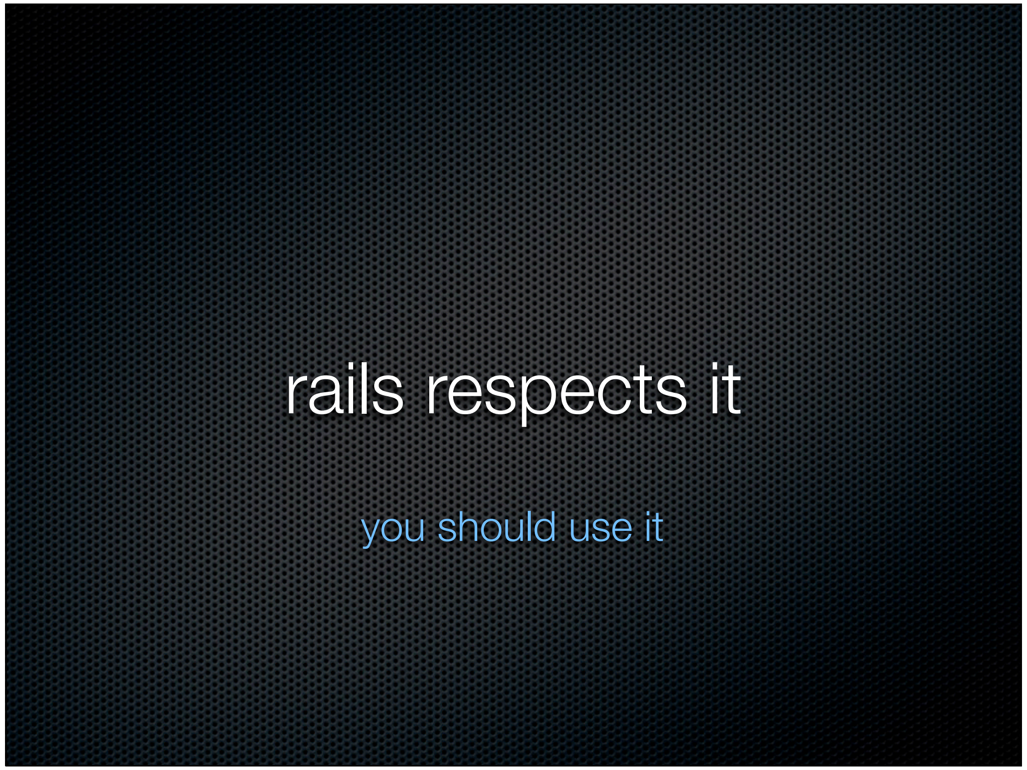
don’t tunnel

application/x-www-form-urlencoded

... is folks tunneling application/x-www-form-urlencoded over application/xml or application/json.

Accept headers

“http://localhost:3000/user.~~xml~~”



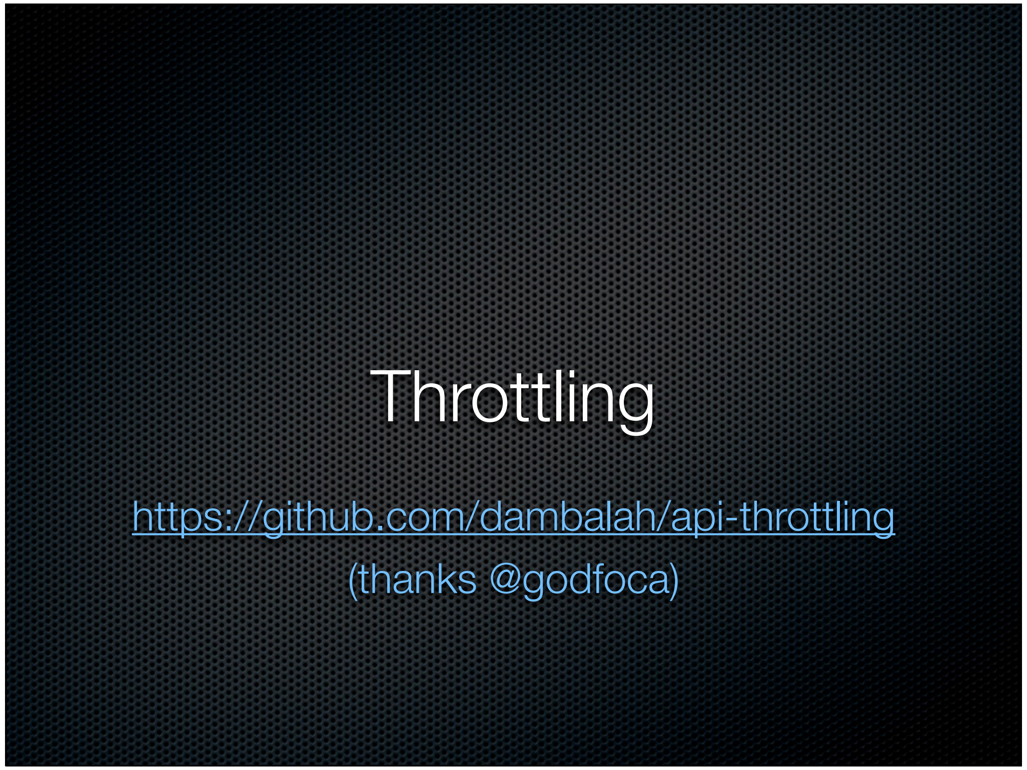
rails respects it

you should use it

a suffix of .xml or .json in the URI is to make it easier to look at api responses in the browser. While consuming an API you should stick to the Accept header.

Important stuff that we’ve never done

There are a couple of important things when dealing with APIs that I’ve never done and have no first-hand experience with



Throttling

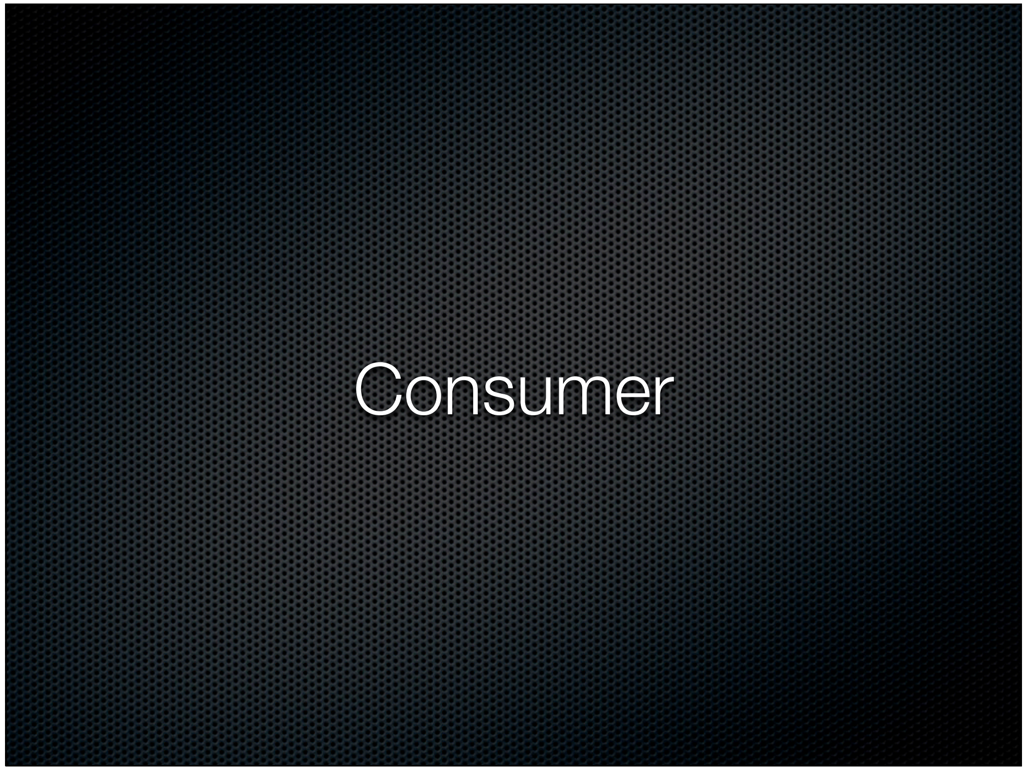
https://github.com/dambalah/api-throttling

(thanks @godfoca)

Solved using rack middleware

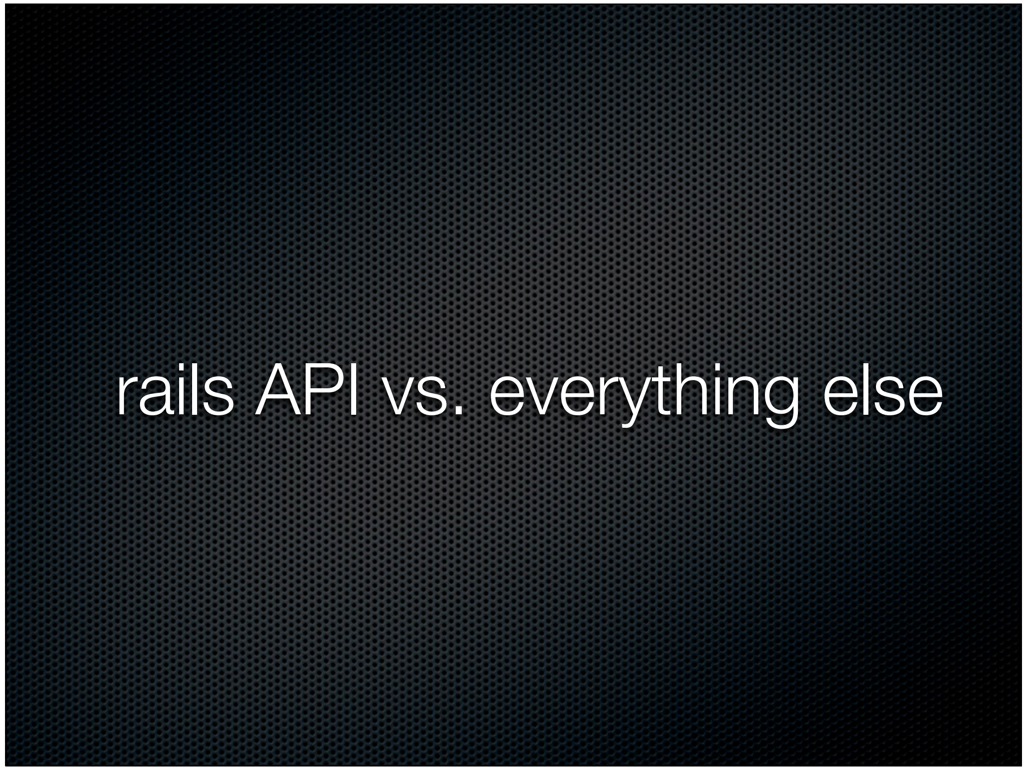
Metering

[Handoff]



Consumer

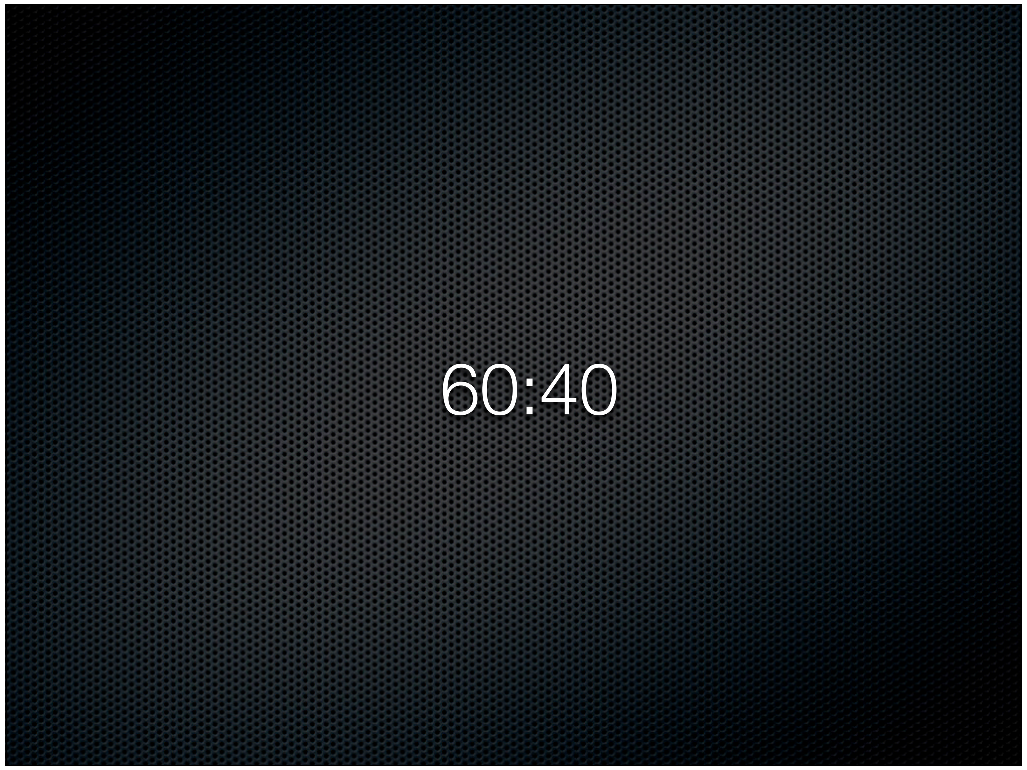
[kaiwren]


rails API vs. everything else

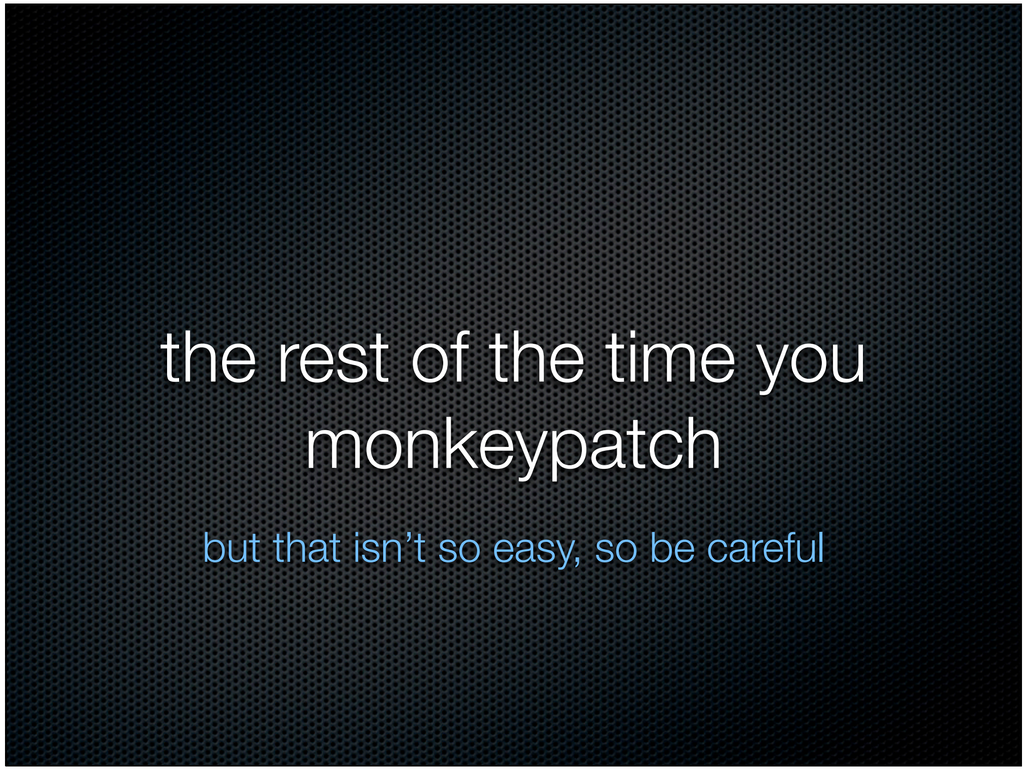
Are you talking to a rails app (typically within your own walled garden) or to an API out on the internets?

ActiveResource

Let’s talk about the consumer that ships with Rails - ActiveResource


60:40

ActiveResource follows the 60/40 rule not 80/20 – it works for the straightforward use cases, and does so smoothly and well



the rest of the time you monkeypatch

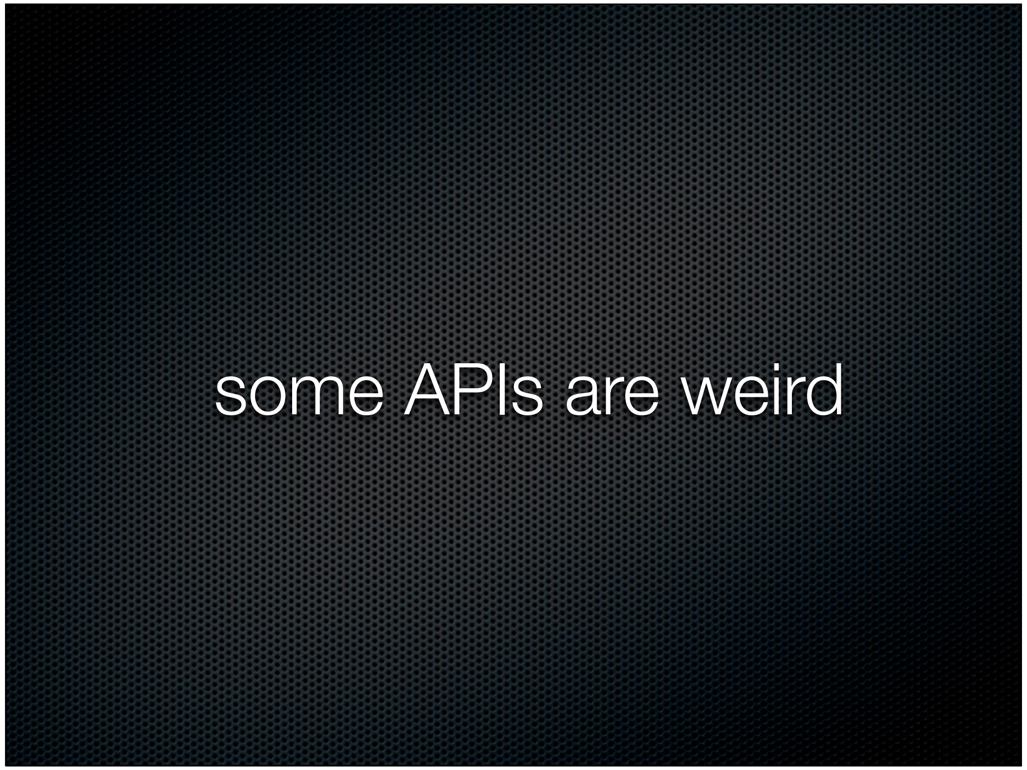
but that isn’t so easy, so be careful

Trying accessing the header information for a response

ActiveResource works best inside a walled garden

(IMHO)

Producer maturity



some APIs are weird

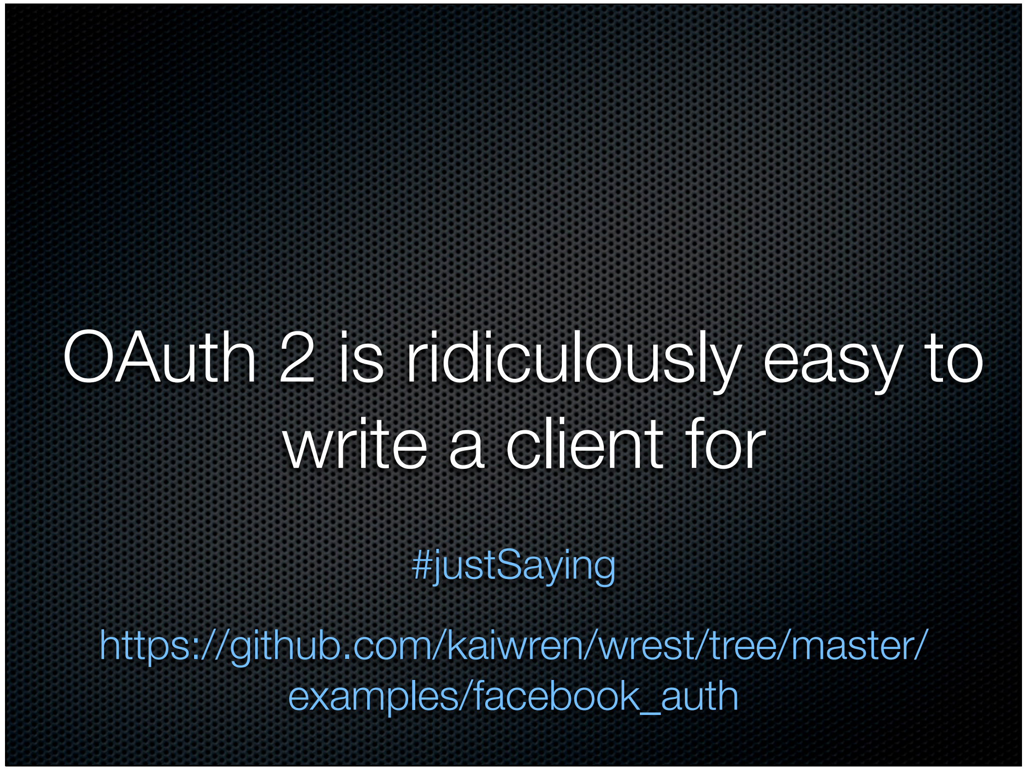
Passing parameters for a post request. Accepting access token as a parameter to a Get over https

everyone’s standardizing on REST

but it’s what *they* call REST

suffice to say, one size doesn’t fit all

except for OAuth 2, which everyone’s adopting building a clean domain model for the API is key



OAuth 2 is ridiculously easy to write a client for

#justSaying

https://github.com/kaiwren/wrest/tree/master/examples/facebook_auth

You don’t need gems for OAuth2 clients. Just roll your own - it takes 20 minutes tops.

authorisation on the other hand still has no standard implementation

What should a good HTTP client do for you?

this one needs bullets

- Verbs - should support GET, POST, PUT & DELETE.
- If it supports PATCH & OPTIONS, epic.
- Transparent deserialization of XML and JSON, with the option to easily add more
- HTTP caching (RFC 2616)
- Keep-alive connections (because negotiating SSL can get expensive)
- Thread safety
- Transparent compression/decompression using Accept-Encoding
- A fluent, easy to use API

most importantly...

verbs

serialization

I rather like ActiveSupport

caching

RFC 2616

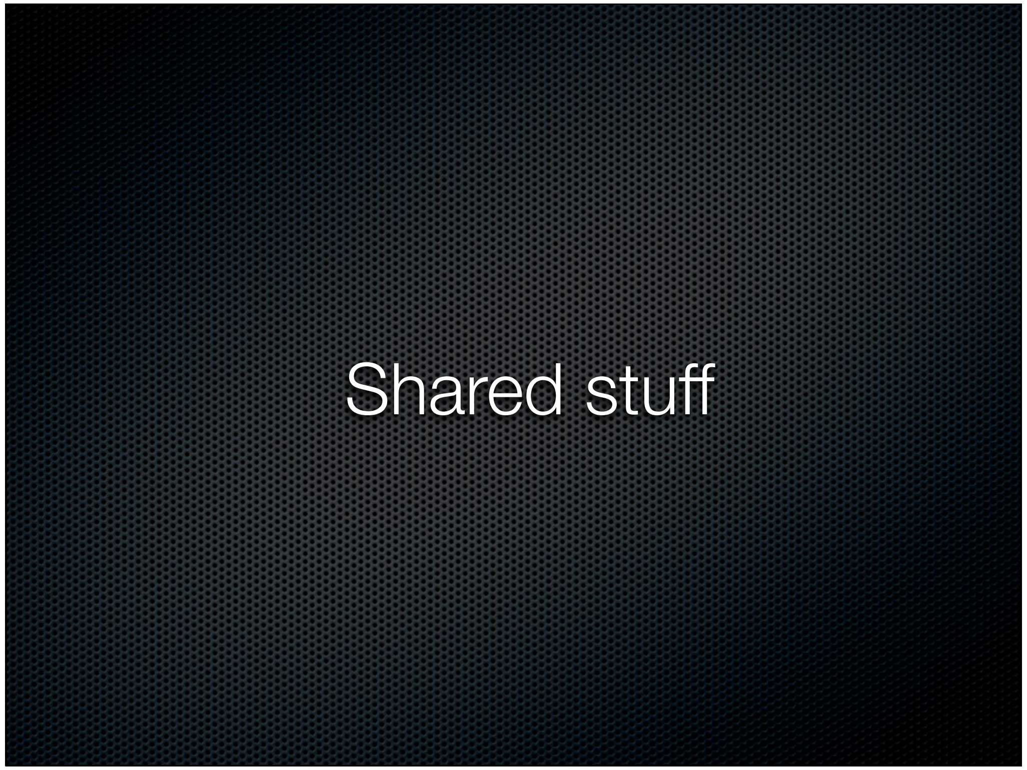
thread safety

fluent api

is it easy to work with uris, because you’ll do a fair bit of that when building domains is it easy to build a domain model using it?

logging/debugging

Can I figure out the verb, the connection (if keep alive), corelate the req and response a situation where parallel threads are logging to the same log

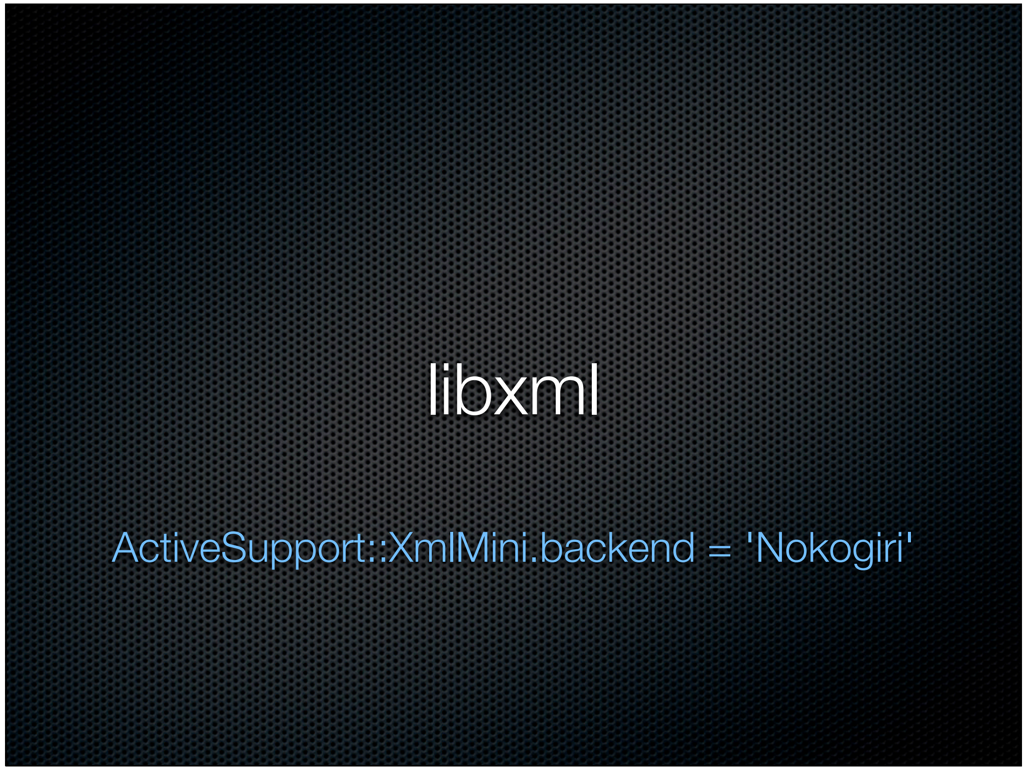


Shared stuff

These are the bits that both producers and consumers deal with

serialization/deserialization

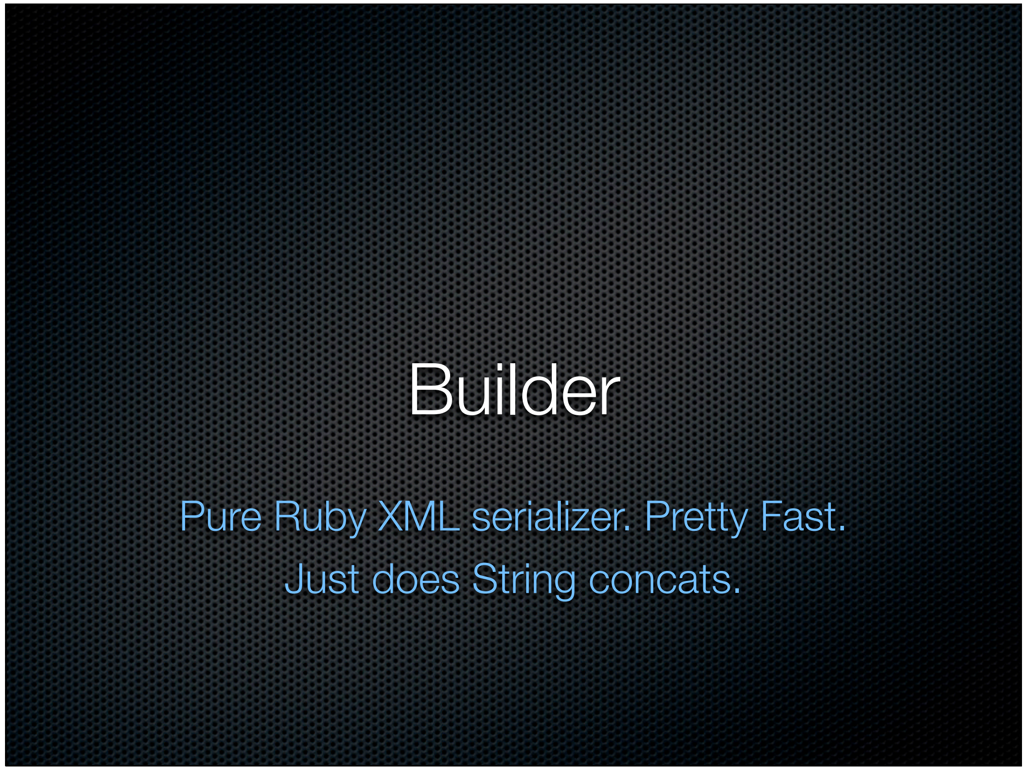
A significant proportion of the time an API takes to process something is spent doing serialisation and deserialization



libxml

```
ActiveSupport::XmlMini.backend = 'Nokogiri'
```

Remember to switch your deserialization backend from REXML to Nokogiri which uses the native libxml library. It’s way way faster.



Builder

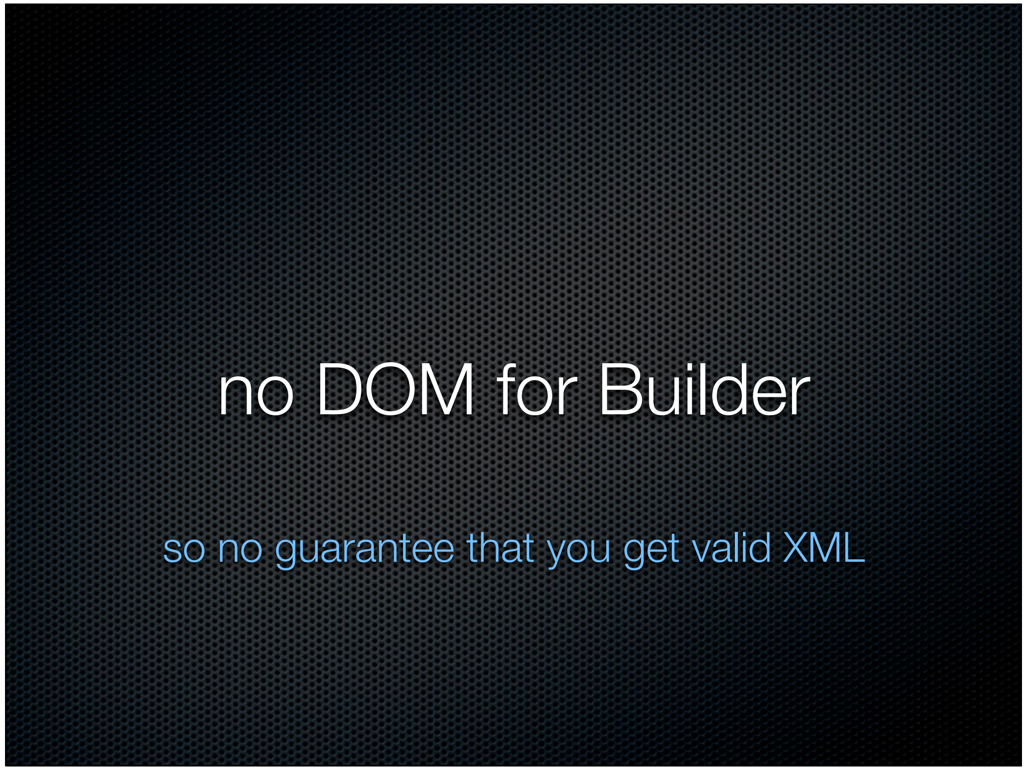
Pure Ruby XML serializer. Pretty Fast.

Just does String concats.

The default XML serializer in Rails is written in pure Ruby. It’s pretty fast, and there are drop in native alternatives based off libxml that are upto 50% faster, but that aren’t particularly popular.

gotcha

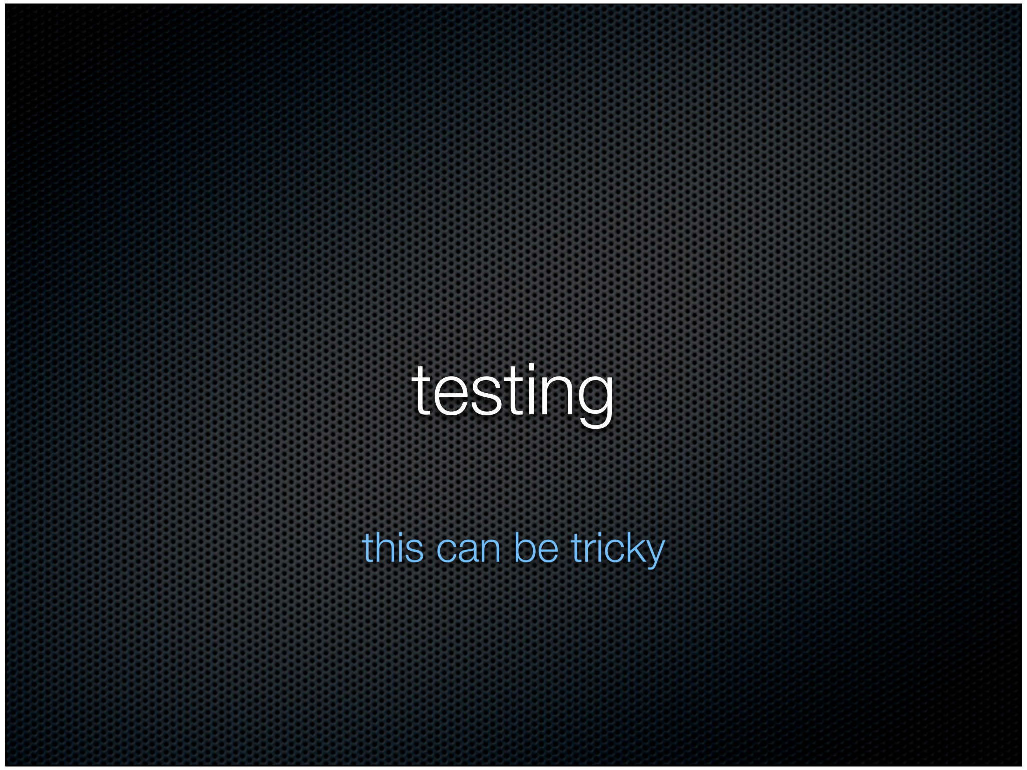
The big gotcha with builder occurs when you’re generating custom XML



no DOM for Builder

so no guarantee that you get valid XML

you want to run all the custom xml you create using builder's DSLs through Nokogiri's deserialise to make sure its valid XML in your specs


testing

this can be tricky

In addition to controller specs, it makes sense to have full, live tests hitting a real server, especially if you're integrating across multiple APIs that you've authored

Conclusion

respect HTTP

respect REST constraints

spend less time messing with plumbing

focus on building a clean domain model

Photo credits

David Blackwell, Elephant in Room: http://www.flickr.com/photos/mobilestreetlife

Funny picture?

cat with mausambi hat

42

Sidu Ponnappa	Niranjan Paranjape
http://twitter.com/ponnappa https://github.com/kaiwren	http://twitter.com/niranjan_p https://github.com/achamian