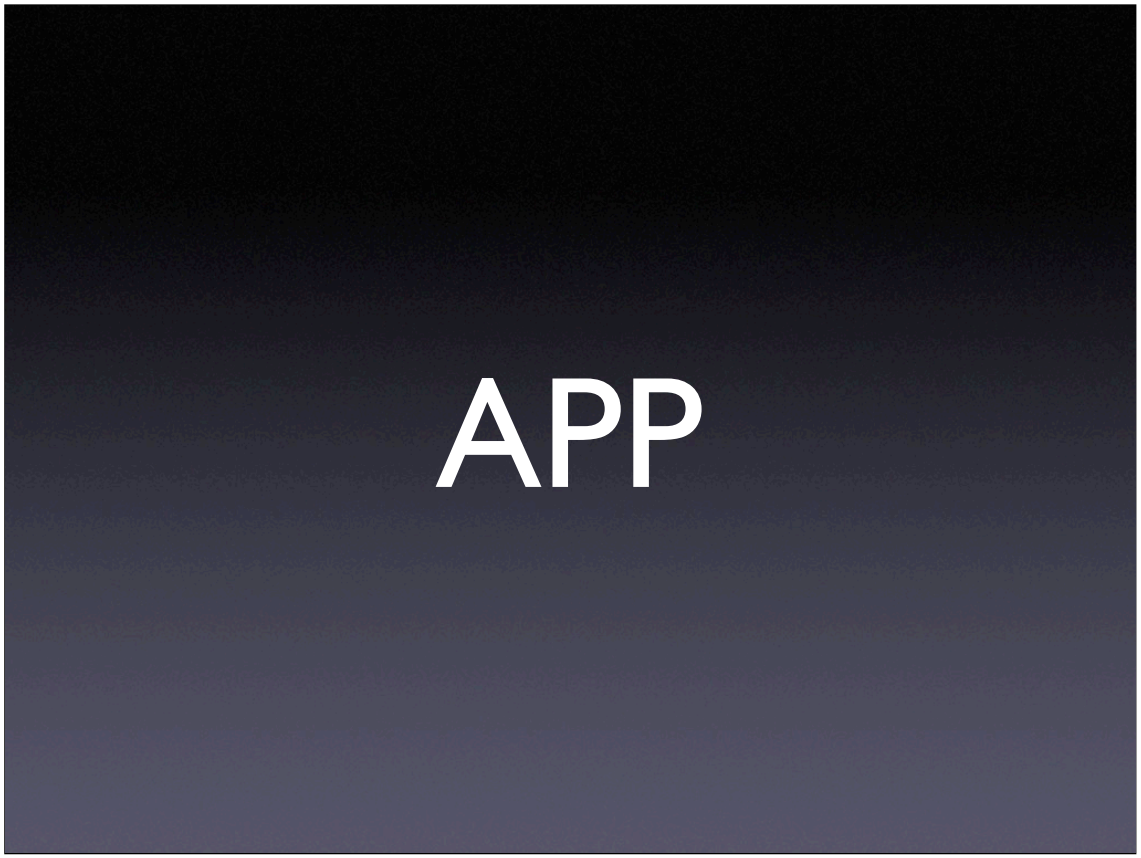
Ruby and the Atom Publishing Protocol

Keith Fahlgren
O'Reilly



APP = Atom Publishing Protocol http://ietfreport.isoc.org/idref/draft-ietf-atompub-protocol/
based on the Atom Syndication Format (you know, “feeds”)
Well-specified, not quite done, at draft 13 (nearly finshed)

Modern

- REST-y Architecture
- Introspective
- Use of Real XML (namepsaces) gives us:
- Easily Extensible

Uses the four HTTP verbs well
Each element/URL responds to HTTP GET with a human-readable, XML response document
Proper, modern XML (namespaced, is an infoset) lead to extensibility (think metadata)

atom:service

```
<service xmlns="http://purl.org/atom/app#"
         xmlns:atom="http://www.w3.org/2005/Atom">
  <workspace>
    <atom:title type="text">O'Reilly</atom:title>
    <collection href="https://atom.oreilly.com/atom/oreilly/images">
      <atom:title type="text">Images</atom:title>
      <accept>image/png,image/jpeg,application/postscript,
              image/bmp,image/gif,image/tiff,application/pdf</accept>
    </collection>
    <collection href="https://atom.oreilly.com/atom/oreilly/articles">
      <atom:title type="text">Articles</atom:title>
      <accept>application/xhtml+xml</accept>
    </collection>
    <collection href="https://atom.oreilly.com/atom/oreilly/books">
      <atom:title type="text">Books</atom:title>
      <accept>application/docbook+xml</accept>
    </collection>
```

A Service is the highest level container.
Inside it are different Workspaces (big, granular buckets)
Inside Workspaces are Collections
They accept new Resources by POST and filter by <accept> (content-type)

atom:collection

That looks like a Feed?!

```
<?xml version="1.0" encoding="UTF-8"?>
<feed xmlns="http://www.w3.org/2005/Atom">
  <title type="text">Books</title>
  <link href="https://atom.oreilly.com/atom/oreilly/books" rel="first"/>
  <link href="https://atom.oreilly.com/atom/oreilly/books?page=3" rel="last"/>
  <link href="https://atom.oreilly.com/atom/oreilly/books?page=2" rel="next"/>
  <subtitle type="text">Stores O'Reilly Book Content</subtitle>
  <entry xmlns:app="http://purl.org/atom/app#">
    <summary type="text">Running Mac OS X Tiger is the ideal resource for power users
and system administrators like you who want to tweak Tiger,
the new release of Mac OS X, to run faster, better, or just
differently....</summary>
    <id>tag:oreilly.com,2007-02-16:/oreilly/books/60656</id>
    <app:edited>2007-02-16T03:04:40.000Z</app:edited>
    <updated>2007-02-16T03:04:34.000Z</updated>
    <link href="https://atom.oreilly.com/atom/oreilly/books/60656" rel="edit"/>
    <link href="https://atom.oreilly.com/source/oreilly/books/60656" rel="edit-media"/>
    <title type="text">Running Mac OS X Tiger</title>
    <author><name>James Duncan Davidson</name><email>duncan@x180.net</email></author>
    <author><name>Jason Deraleau</name><email>jldera@mac.com</email>
    </author>
    <subtitle type="text">A No-Compromise Power User's Guide to the Mac</subtitle>
    <!-- your XML here -->
    <content src="https://atom.oreilly.com/source/oreilly/books/60656"
             type="application/docbook+xml"/>
  </entry>
```

GETing a Collection gives you a feed (hooray!)
A feed, just like an Atom Syndication Format feed, is made up of Entries
The use of feeds provides the easy bridge between early adopters and regular folks
- just tell them to subscribe in their feedreader or Firefox

What Real XML Gets You

Arbitrary Embedded XML, hooray!

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:orm="http://oreilly.com/rdf#">
  <rdf:Description
    rdf:about="https://atom.oreilly.com/atom/oreilly/books/60656">
    <orm:isbn>9780596009137</orm:isbn>
    <orm:uploader>keith</orm:uploader>
  </rdf:Description>
</rdf:RDF>
```

Namespaces suck for XML users, but aren't that bad once you know how to use it
And they give you tremendous gains, by allowing embedding
...like this RDF triple describing my Entry

Things I liked about making my library and thought I did well

HTTP

- Net::HTTP Just Works*
- Easy to DRY

*this isn't entirely true

You don't have to go beyond the Standard Library's Net::HTTP to get all the HTTP goodness you need (even with the need for 4 verbs)
And it's easy to hide all of the complexity from library users (as usual for Ruby)

The Four HTTP Verbs

- POST

```
h.post2(uri.path + query, data.to_s, headers) {|resp|
  unless resp.is_a?(Net::HTTPCreated)
    # Kaboom
```

- PUT

```
h.put2(uri.path + query, data.to_s, headers) {|resp|
```

- GET

```
h.get2(uri.path + query, {"Accept" => 'text/xml'}) {|resp|
```

- DELETE

```
resp = h.delete(uri.path + query, headers)
# may want headers = {"Depth" => "Infinity"}
```

Yeah, these *2 methods all exist, and when writing library code, I like the flexibility that the extra arguments provide

XML

- REXML Works*
- ..or Libxml-Ruby works
- Relatively Easy to DRY

*this isn't entirely true

As with HTTP, you can get decent XML support right out of the Standard Lib
Why choose one XML library over the other?
 - Speed, mainly, though I prefer the libxml bindings, but that's only because I'm an XPath freak (XPath seems more central to libxml than REXML, but you can do it in both)
When writing real XML-consuming libraries, you really need to use XPath rather that some ad-hoc, array-based lookup (especially because Atom Entries are unordered...)

HTTP->XML

Pick your poison

```
def self.rexml_from_url(url)
  REXML::Document.new(data_from_url(url))
end
def self.libxml_from_url(url)
  XML::Parser.string(data_from_url(url)).parse
end
```

Right from the start, I chose not to choose which library I'd be bound to
This small step allowed me to experiment a lot more with implementation, which was a good choice

Gotchas

(well, got-me-s, actually)

or

Things I did Wrong

REXML

- REXML Doesn't Work*
- Namespaces Are a Pain
- Use REXML::XPath, not anything else

*this isn't entirely true

As an XPath guy, I tripped a lot on REXML's methods and use of Namespaces, XPath

```
ATOM_NS = {"atom" => "http://www.w3.org/2005/Atom"}
class Entry
  attr_reader :atom_id, :title,
              :subtitle, :content_type,
              :content_src, :href,
              :media_href, :xml, :summary
  def initialize(xml, auth=nil)
    @xml = xml
    @atom_id =
      REXML::XPath.first(@xml.root,
           "/atom:entry/atom:id/text()",
           ATOM_NS).to_s
    @title =
      REXML::XPath.first(@xml.root,
           "/atom:entry/atom:title/text()",
           ATOM_NS).to_s
....
```

I really really like REXML::XPath rather than other methods
If you need More than one Namespace, join your hashes into a big hash

Bad HTTP

- When to use_ssl?
- HTTP Auth is Funny

Some things in Net::HTTP are pretty annoying

Gross Auth

```
http.start{|h|
  h.read_timeout = DEFAULT_TIMEOUT
  query = uri.query ? "?" + uri.query : ""
  headers = {"Depth" => "Infinity"}
  # that's right, base64!
  headers["Authorization"] = "Basic " +
  Base64.encode64("#{auth[:username]}:#{auth[:password]}").chomp if auth
  resp = h.delete(uri.path + query, headers)
  return false unless resp.is_a?(Net::HTTPOK)
}
```

Also, .is_a? isn't cool

SSL Who?

```
uri = URI.parse(url)
http = Net::HTTP.new(uri.host, uri.port)
if url =~ /^https|:443\// # hmm this looks brittle
  http.use_ssl = true
  # I'm not cool enough to have a cert
  http.verify_mode = OpenSSL::SSL::VERIFY_NONE
end
```

Why doesn't URI.parse() know whether or not I should use SSL?