REST is Easy

(... or simple, anyway)

Paul M. Winkler
topp.openplans.org
at PyCon Chicago 2008

http://slinkp.com/pycon08

What is REST?

not an implementation?

not a spec?

not an architecture?

It's an architectural style!

Says who?

Roy T. Fielding
(http://roy.gbiv.com/)

Too abstract!

I just want to make a web app!

REST Defined:
"**Re**presentational **S**tate **T**ransfer"

Eh?

What's a Representation?

Bytes, representing a **resource.**

... so what's a **resource**?

Anything you're interested in.

Resources are the nouns of the web.

1 resource → N representations

What about **State**?

- Resource state (at the server)
- Application state (at the client)

Representational **S**tate **T**ransfer

Client transfers states *to* the server
to update server's resource state

Client gets states *from* the server
to update its own state

REST today → HTTP

(usually)

HTTP example: request & response

request:

```
GET /helloworld HTTP/1.1
Host: example.com
```

response:

```
HTTP/1.0 200 OK
Date: Tue, 11 Mar 2008 14:09:27 GMT
content-type: text/plain

Hello, world!
```

Constraint: Client-server

Constraint: Uniform Interface

- Name resources with URIs
- Standard HTTP methods
  `GET, POST, PUT, DELETE, ...`
- Standard HTTP response codes
  `200 OK, 404 Not Found, ...`
- Self-descriptive messages - headers and MIME types
  `text/plain,image/png, application/atom+xml`

Essential HTTP methods: GET

Fetch some data.

safe (no side effects)

idempotent (one get is as good as ten)

Essential HTTP methods: DELETE

Delete resource at this URI.

unsafe + *idempotent*

Essential HTTP methods: PUT

Update (or create) resource at this URI.

needs a body.

unsafe + *idempotent.*

Essential HTTP methods: POST

Create something subordinate to this resource.

Or ...?

needs a body.

unsafe + ***not*** *idempotent.*

Request & Response example two

```
DELETE /helloworld HTTP/1.1
Host: example.com
```

response:

```
HTTP/1.0 405 Method Not Allowed
Date: Tue, 11 Mar 2008 14:11:31 GMT
allow: POST, GET
content-type: text/plain

Sorry, /helloworld can't be
deleted!
```

Constraint: Stateless

What?? State is part of the acronym!

The ***protocol*** is stateless.

Other REST constraints

- layered system
  (proxies!)
- caching support in protocol (optional)
- code-on-demand (optional)
  (javascript, applets, ...)

Last crucial constraint: Hypermedia

"Hypermedia as the Engine of Application State"
(HATEOAS)

Say what?

Hypermedia as the Engine of Application State

You know two examples:

links

forms

When is HTTP not REST?

Typical ways to break the constraints:

unsafe GET

one “service” URI

overloaded POST

Show me some code already!

```
def hello_world(environ, start_response):
    start_response(
        '200 OK',
        [('Content-Type', 'text/plain')])
    return 'Hello world!\n'


from wsgiref.simple_server import \
    make_server

http = make_server('localhost', 8080,
                   hello_world)
http.serve_forever()
```

What about a client?

```
import urllib
result = urllib.urlopen(
    'http://localhost:8080/helloworld')
print result.read()
```

Let's write a blog app!

Step 1. Make a list of resources

Blogs and Blog Entries.

1a) trivial blog entry

```
class BlogEntry:

    def __init__(self, path, parent,
                 title, body):
        self.path = urllib.quote(path)
        self.parent = parent
        self.update(title, body)

    def update(self, title, body):
        if not (title and body):
            raise ValueError(...)
        self.title = title
        ...
```

1b) trivial blog

```
class Blog:
    def __init__(self, title, path):
        self.path = urllib.quote(path)
        self.title = title
        self.entries = {}

    def create_entry(self, title, body):
        ...
        entry = BlogEntry(
            name, self, title, body)
        self.entries[name] = entry
        return entry
    ...
```

Step 2) Give resources URIs

But maybe not part of your API

http://.../ → *list of blogs*

http://.../{blog} → *a blog (list of entries)*

http://.../{blog}/{entry} → *a single entry*

Step 3) Decide which HTTP methods each resource supports

and what each does.

	GET	POST	PUT	DELETE
Blog	list of entries	create new entry	--	--
Blog Entry	entry data	--	update entry	delete entry

4. Choose representations for each case from step 3.

don't invent more than you have to

	GET	POST	PUT	DELETE
Blog	list of entries (HTML)	create entry (form-encoded	--	--

4a) Example html for a blog entry

```
<html>
<body class="entry">
  <h1>%(blogtitle)s</h1>
  <h2>%(title)s</h2>
  <p id="body">%(body)s</p>
  <p><i id="timestamp">
      Last updated:
      %(timestamp)s</i></p>
  <p><a href="%(blogurl)s"> Home
     </a></p>
</body>
</html>
```

Step 5) Status codes

example – POSTing to a blog:

404 Not Found
(the blog itself doesn't exist)

201 Created
(success; send a Location header with the entry's URL)

400 Bad Request
(missing parameters, or wrong content-type)

401 Unauthorized

6. Anything left over?

Get creative with resource design?

Implement HTTP with WSGI

```
def get_entry(environ, start_response):
    # omitted: find blog & entry name
    entry = blog.get(entryname)
    if entry is None:
         start_response('404 Not Found', ...)
         return '%s not found\n' % entryname
    start_response(
        '200 OK',
        [('Content-Type', 'text/html')])
    # omitted: load & populate html file
    return html
```

Glue it all together

```
def get_entry(environ, start_response):
    vars = environ['selector.vars']
    blogname = vars['blogname']
    entryname = vars['entryname']
    ...

app = Selector()
app.add('/{blogname}/{entryname}',
        GET=get_entry,
        DELETE=delete_entry,
        PUT=update_entry)
# ... wire up more URIs ...
```

We're done!

more code linked from the references

Questions?

Permalink

You can find these slides and example code at:
http://slinkp.com/pycon08

Author: Paul Winkler <stuff@slinkp.com>

Bibliography

* Roy Fielding's 2007 Apachecon talk, a must-read (for a good laugh see slides 28-30)
  http://roy.gbiv.com/talks/200711_REST_ApacheCon.pdf

* Roy Fielding's dissertation, this is the origin text:
  http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

* *RESTful Web Services*, Leonard Richardson & Sam Ruby, O'Reilly
  http://www.oreilly.com/catalog/9780596529260/

* REST Wiki: http://rest.blueoxen.net

* HTTP specification:
  http://www.w3.org/Protocols/rfc2616/rfc2616.html

* "How to Create a REST Protocol", Joe Gregorio
  http://www.xml.com/pub/a/2004/12/01/restful-web.html

* "Common Rest Mistakes", Paul Prescod
  http://www.prescod.net/rest/mistakes/

* rest-discuss list http://tech.groups.yahoo.com/group/rest-discuss/

Further Reading

* Atom Publishing Protocol
  a (draft) standard "application-level protocol for publishing and editing Web resources using HTTP and XML". Extensible.
  http://bitworking.org/projects/atom/draft-ietf-atompub-protocol-04.html

* “Managing Content with the Atom Publishing Protocol”, Andrew Savikas of O'Reilly, slides http://2006.xmlconference.org/proceedings/202/slides.pdf

* "Do we need WADL?" by Joe Gregorio
  http://bitworking.org/news/193/Do-we-need-WADL
  (spoiler: he says no.)

* URI Templates draft standard ... a better way to do client-side URI generation, by having the server provide explicit rules:
  http://bitworking.org/projects/URI-Templates/

* Microformats (a nice way to get more semantic mileage out of HTML without having to invent much):
  http://en.wikipedia.org/wiki/Microformats
  http://microformats.org/

Appendix: client tools

* httplib2
  http://code.google.com/p/httplib2/

* restclient library
  http://microapps.sourceforge.net/restclient/

* RESTClient (not the same!), wxpython GUI for hand-testing
  http://restclient.org/

* mechanize
  http://wwwsearch.sourceforge.net/mechanize/

* feedparser
  http://www.feedparser.org/

* curl, featureful command-line HTTP client.
  http://curl.haxx.se/

* pycurl (libcurl integration for python).
  http://curl.haxx.se/libcurl/python/

* More links: http://microapps.org/Useful_Libraries

Appendix: Server tools

* selector
  http://lukearno.com/projects/selector/

* WSGI: http://www.python.org/dev/peps/pep-0333
  and http://www.wsgi.org/

* paste
  http://pythonpaste.org/

* your favorite framework:
  Django: http://code.google.com/p/django-rest-interface/
  Pylons: http://pylonshq.com/docs/module-pylons.decorators.rest.html
  Grok: http://grok.zope.org/documentation/how-to/rest-support-in-grok
  Turbogears: ... I don't know, send me a link!!
  Plone: http://tinyurl.com/3yze9a

Appendix: Troubleshooting HTTP

* TCPWatch, simple gui, in python
  http://hathawaymix.org/Software/TCPWatch

* tcpflow, simple command-line tool watches a network interface:
  http://www.circlemud.org/~jelson/software/tcpflow/

* wireshark, fancy GUI with all the bells and whistles:
  http://www.wireshark.org/

* Firefox live headers extension: http://livehttpheaders.mozdev.org/

Appendix: Real-world Services

* Gdata - Google APIs that build on the Atom protocol.
  http://code.google.com/apis/gdata/

* S3 - Amazon's commercial storage service

  http://developer.amazonwebservices.com/connect/kbcategory.jspa?categoryI

REST is Easy

(... or simple, anyway)

Paul M. Winkler
topp.openplans.org
at PyCon Chicago 2008

http://slinkp.com/pycon08

I'm Paul Winkler, I work at the Open Planning Project in New York City, and I hack web stuff in Python.

Disclaimer: I'm not an authority on REST; this talk is partly an excuse to do a lot of research and exploration.

What is REST?

not an implementation?

not a spec?

not an architecture?

It's an architectural style!

It's an architectural style: a particular set of constraints for an architecture.

The one important real-world architecture that actually follows the REST constraints is the Web.

So REST is the style of the web's architecture.

Says who?

Roy T. Fielding
(http://roy.gbiv.com/)

Roy Fielding invented REST. He's one of the founding fathers of the Web: his name is on various protocols and working groups.

He wrote the dissertation that defines REST. Link in my bibliography.

Too abstract!

I just want to make a web app!

4

Me too.

But we want to understand the constraints of REST, the benefits of following them, and the way HTTP implements those constraints, so we can leverage all that value.

REST Defined:
"**Re**presentational **S**tate **T**ransfer"

Eh?

First stumbling block: The acronym.

What do these words mean?

Let's define them, left to right.

What's a Representation?

Bytes, representing a **resource.**

... so what's a **resource**?

Anything you're interested in.

Resources are the nouns of the web.

1 resource → N representations

Example: "Paul Winkler's talk about REST at Pycon" is a resource, and it has a name -- its URI.

E.g. I might have PDF and HTML representations of my talk.

What about **State**?

- Resource state (at the server)
- Application state (at the client)

Resource state - the state that is represented when you fetch something from the server. The same state is available to all clients (given sufficient authorization).

Application state - the state of whatever the user is trying to do with the client software. This lives strictly on the client.

Representational **S**tate **T**ransfer

Client transfers states *to* the server
to update server's resource state

Client gets states *from* the server
to update its own state

That's really all that happens.

We're not calling remote procedures for their side effects. Instead we push and pull the state we want.

The client is the driver
hypertext is the engine
the server provides the road maps

(oh god, it's the information highway metaphor back from the dead)

REST today → HTTP

(usually)

From now on, when I talk about REST, I'm really talking about REST as exemplified by the web and HTTP. This is usually what people mean by REST.

HTTP example: request & response

request:

```
GET /helloworld HTTP/1.1
Host: example.com
```

response:

```
HTTP/1.0 200 OK
Date: Tue, 11 Mar 2008 14:09:27 GMT
content-type: text/plain

Hello, world!
```


Let's get concrete.
Basic HTTP anatomy:
The request and the response each consist of text headers in RFC-822 style (just like email), with an optional body after a blank line.

Those are really simple examples, but remember the constraints I keep mentioning? We can see some of them already.

Constraint: Client-server

Any client-server protocol would satisfy the first constraint.

Constraint: Uniform Interface

- Name resources with URIs
- Standard HTTP methods
  `GET, POST, PUT, DELETE, ...`
- Standard HTTP response codes
  `200 OK, 404 Not Found, ...`
- Self-descriptive messages - headers and MIME types
  `text/plain,image/png, application/atom+xml`

The same interface must apply to all resources, everywhere!

Here we see HTTP's uniform interface.

Some people call the HTTP methods “verbs”. I use both terms interchangeably.

Essential HTTP methods: GET

Fetch some data.

safe (no side effects)

idempotent (one get is as good as ten)

Logging is not a side-effect.

Essential HTTP methods: DELETE

Delete resource at this URI.

unsafe + idempotent

Essential HTTP methods: PUT

Update (or create) resource at this URI.

needs a body.

unsafe + idempotent.

Essential HTTP methods: POST

Create something subordinate to this resource.

Or ...?

needs a body.

*unsafe + **not** idempotent.*

“Create something subordinate to this resource”: that might be a child resource with its own URI, or some kind of annotation on this resource.

Often used for arbitrary other purposes. But please read the HTTP spec and try to stick to it!

There are a couple more verbs that aren't used as much.

Request & Response example two

```
DELETE /helloworld HTTP/1.1
Host: example.com
```

response:

```
HTTP/1.0 405 Method Not Allowed
Date: Tue, 11 Mar 2008 14:11:31 GMT
allow: POST, GET
content-type: text/plain

Sorry, /helloworld can't be
deleted!
```


Again we see the uniform interface. The only thing new here is a different request method (DELETE), a new response code (405), and a new response header: “allow” tells you what methods this resource supports.

Constraint: Stateless

What?? State is part of the acronym!

The ***protocol*** is stateless.

This one is important, but hard to show.

What's it mean? We have client state, resource state, and state representations flying back and forth... state everywhere!

But note our two example requests were each totally self-contained. The _protocol_, in this case HTTP, must be stateless. Client state and server state are never implicitly shared, and the server exists in a perpetual history-less “now”: its response is determined only by the interaction of the current request with the server's current state.

The motive for statelessness:
Server implementation is simple since it doesn't have to maintain copies of client state. Same for proxies.
Parallelizing servers for scalability and reliability is easy.
Drawback: Requests and responses can get bulky since they have to be self-contained.)

Cookies are "cheating" - you throw away both the benefits and the drawback of statelessness.

Other REST constraints

- layered system
  (proxies!)
- caching support in protocol (optional)
- code-on-demand (optional)
  (javascript, applets, ...)

There are some more constraints. We'll skip these.

Last crucial constraint: Hypermedia

"Hypermedia as the Engine of Application State" (HATEOAS)

Say what?

Hypermedia as the Engine of Application State

You know two examples:

links

forms

22

Hypermedia is the engine, and the client is the driver.

* Think of following a link in a browser. That's hypermedia. You haven't changed any resource state on the server, but by following the link you've changed the application state on the client: "I am now at *this* page"

* Think of submitting a form in a browser. Again, that's hypermedia. You send the server a representation (application/x-www-form-urlencoded) of new state that you'd like it to have.

One huge benefit of hypermedia is late binding of control flow: this makes the client more robust.

When is HTTP not REST?

Typical ways to break the constraints:

unsafe GET

one “service” URI

overloaded POST

There are lots of ways to violate the constraints. Typically this means breaking the uniform interface in some way.

It's sadly common to expose unsafe operations via GET. Never, never do this. (Sadly, lots of "rest" APIs do, eg. del.icio.us). That's not just un-REST, it violates the HTTP spec.

Or, you might use one "service" URI that responds to only one HTTP method, and identifies the resource and the operation to perform in the message body. This is how both SOAP and XML-RPC work. Which is OK if you really just want a way to make remote calls across a network, but then *you're not part of the web.* Why balkanize your data?

Show me some code already!

```
def hello_world(environ, start_response):
    start_response(
        '200 OK',
        [('Content-Type', 'text/plain')])
    return 'Hello world!\n'


from wsgiref.simple_server import \
    make_server

http = make_server('localhost', 8080,
                   hello_world)
http.serve_forever()
```


Here's a trivial HTTP server using only the python standard library.

Well, actually it's a WSGI (Web Server Gateway Interface) server. WSGI is a simple Python standard (defined in PEP 333) for building HTTP applications.

All you need is a function or method that takes an 'environ' dictionary (this gives you data about the request), and a 'start_response' callback that you will call to set headers. Then your function returns the response body, if any.

Initialize a WSGI server with your function, and off you go!

What about a client?

```
import urllib
result = urllib.urlopen(
    'http://localhost:8080/helloworld')
print result.read()
```


Here's a trivial HTTP client. Again using only the python standard library. There's a more useful example in my sample code, which uses a nice little third-party library called restclient.

Let's write a blog app!

Write yet another blog? Sounds fun. It's a familiar feature set and I have no imagination.

We're going to follow a simple six-step process for creating a RESTful service. This is based on a Joe Gregorio article, and chapter 5 of the Ruby / Richardson book (both listed in my bibliography).

Step 1. Make a list of resources

Blogs and Blog Entries.

27

Identify the resources.

We just have Blogs and Blog Entries.
No comments.

These are so simple I'm going to just go ahead and write some code as we go (with some omissions for brevity; there's a link to the full code samples at the end.)

1a) trivial blog entry

```
class BlogEntry:

    def __init__(self, path, parent,
                 title, body):
        self.path = urllib.quote(path)
        self.parent = parent
        self.update(title, body)

    def update(self, title, body):
        if not (title and body):
            raise ValueError(...)
        self.title = title
        ...
```


Here's my blog entry class.

Notice there's nothing here about HTTP or URIs yet. It's just plain python, simple content with a trivial API.

1b) trivial blog

```
class Blog:
    def __init__(self, title, path):
        self.path = urllib.quote(path)
        self.title = title
        self.entries = {}

    def create_entry(self, title, body):
        ...
        entry = BlogEntry(
            name, self, title, body)
        self.entries[name] = entry
        return entry
    ...
```

Step 2) Give resources URIs

But maybe not part of your API

http://.../ → *list of blogs*

http://.../{blog} → *a blog (list of entries)*

http://.../{blog}/{entry} → *a single entry*

Anyway, we're just going to do a typically simple URI pattern, as shown here. The root URI will be a list of blogs; the first part of the path will be identify a particular blog; the second part of the path will identify a blog entry.

But don't make too much of this. Nice URI design is very valuable for humans, but I think some people think that URI design ***is*** REST. It's not at all. Neither REST nor HTTP cares at all about the structure of a URI: *it's just an identifier.*

Go ahead and make nice URIs for humans, but put more effort into linking resources to each other.

Remember about the hypermedia constraint, we said that one benefit was late binding? If clients have to generate or parse URIs instead of just following them, you lose that benefit.

The hypermedia constraint wasn't phrased as: "Making up URIs as the engine of application state".

Step 3) Decide which HTTP methods each resource supports

and what each does.

	GET	POST	PUT	DELETE
Blog	list of entries	create new entry	--	--
Blog Entry	entry data	--	update entry	delete entry

4. Choose representations for each case from step 3.

don't invent more than you have to

	GET	POST	PUT	DELETE
Blog	list of entries (HTML)	create entry (form-encoded	--	--

HTML and form-encoding aren't the most semantically rich choices, but one big advantage is that our service can double as a human-readable website. You can play with it in your browser.

A more semantically interesting choice for a blog would be the Atom syndication format, but that deserves a whole other talk, and I don't know enough about it.

4a) Example html for a blog entry

```
<html>
<body class="entry">
  <h1>%(blogtitle)s</h1>
  <h2>%(title)s</h2>
  <p id="body">%(body)s</p>
  <p><i id="timestamp">
      Last updated:
      %(timestamp)s</i></p>
  <p><a href="%(blogurl)s"> Home
     </a></p>
</body>
</html>
```


Here's one of my HTML representations. I used vanilla python string interpolation to avoid a giant tangent about templating languages.

This one is what you see when you GET a single blog entry. It has a link to its parent blog, a title, a body, and a timestamp.

Step 5) Status codes

example – POSTing to a blog:

404 Not Found
(the blog itself doesn't exist)

201 Created
(success; send a Location header with the entry's URL)

400 Bad Request
(missing parameters, or wrong content-type)

401 Unauthorized

For each case in step 3, you need to figure out what all the possible results are, and assign each one a status code from the HTTP spec. A lot of these are self-explanatory.

Some of them require you to send extra response headers; for example, with 201 Created, you should always send a Location: header that gives the URI of the new resource.

6. Anything left over?

Get creative with resource design?

If you have requirements that don't seem to fit into any of the HTTP methods, maybe go back to step 1 and try to imagine decomposing the problem into more resources. REST puts the focus on resources, not operations.

Sometimes this is hard; HTTP lacks some really useful verbs, like MOVE or COPY or PATCH.

CRUD-type applications are typically straightforward to map to the standard HTTP methods.

It's tempting to want to invent new HTTP methods or response codes. You should think long and hard before doing that because you will give up interoperability with all the HTTP servers and clients out there.

Implement HTTP with WSGI

```
def get_entry(environ, start_response):
    # omitted: find blog & entry name
    entry = blog.get(entryname)
    if entry is None:
         start_response('404 Not Found', ...)
         return '%s not found\n' % entryname
    start_response(
        '200 OK',
        [('Content-Type', 'text/html')])
    # omitted: load & populate html file
    return html
```


Our design is done. We just need to hook up our classes and representations to aserver. We'll use WSGI.

This function is a WSGI application that will can respond to a GET request for a blog entry.

We'd write one WSGI application for each box in our chart of resources and HTTP methods.

(Don't worry if that sounds like a lot of “applications”. WSGI applications are meant to be small and composable.)

Next we'll wire it up so it knows which requests to respond to.

Glue it all together

```
def get_entry(environ, start_response):
    vars = environ['selector.vars']
    blogname = vars['blogname']
    entryname = vars['entryname']
    ...

app = Selector()
app.add('/{blogname}/{entryname}',
        GET=get_entry,
        DELETE=delete_entry,
        PUT=update_entry)
# ... wire up more URIs ...
```


We need to map each combination of URI and HTTP method to a WSGI app.

We can do this using a nice little library called selector, it's in the cheese shop.

Selector is going to parse the blog name and entry name out of the URI and put them in the environ. We can now fill in the beginning of get_entry.

Looking at the bottom half of the slide, you just configure selector with a URI pattern, and pass it WSGI applications for each HTTP verb that matching URIs should respond to. For any verbs you don't specify, it'll respond with 405 Method Not Allowed.

We're done!

more code linked from the references

Questions?

Permalink

You can find these slides and example code at:
http://slinkp.com/pycon08

Author: Paul Winkler <stuff@slinkp.com>

Bibliography

* Roy Fielding's 2007 Apachecon talk, a must-read (for a good laugh see slides 28-30)
  http://roy.gbiv.com/talks/200711_REST_ApacheCon.pdf

* Roy Fielding's dissertation, this is the origin text:
  http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

* *RESTful Web Services*, Leonard Richardson & Sam Ruby, O'Reilly
  http://www.oreilly.com/catalog/9780596529260/

* REST Wiki: http://rest.blueoxen.net

* HTTP specification:
  http://www.w3.org/Protocols/rfc2616/rfc2616.html

* "How to Create a REST Protocol", Joe Gregorio
  http://www.xml.com/pub/a/2004/12/01/restful-web.html

* "Common Rest Mistakes", Paul Prescod
  http://www.prescod.net/rest/mistakes/

* rest-discuss list http://tech.groups.yahoo.com/group/rest-discuss/

Further Reading

* Atom Publishing Protocol
  a (draft) standard "application-level protocol for publishing and editing Web resources using HTTP and XML". Extensible.
  http://bitworking.org/projects/atom/draft-ietf-atompub-protocol-04.html

* “Managing Content with the Atom Publishing Protocol”, Andrew Savikas of O'Reilly, slides http://2006.xmlconference.org/proceedings/202/slides.pdf

* "Do we need WADL?" by Joe Gregorio
  http://bitworking.org/news/193/Do-we-need-WADL
  (spoiler: he says no.)

* URI Templates draft standard ... a better way to do client-side URI generation, by having the server provide explicit rules:
  http://bitworking.org/projects/URI-Templates/

* Microformats (a nice way to get more semantic mileage out of HTML without having to invent much):
  http://en.wikipedia.org/wiki/Microformats
  http://microformats.org/

Appendix: client tools

* httplib2
  http://code.google.com/p/httplib2/

* restclient library
  http://microapps.sourceforge.net/restclient/

* RESTClient (not the same!), wxpython GUI for hand-testing
  http://restclient.org/

* mechanize
  http://wwwsearch.sourceforge.net/mechanize/

* feedparser
  http://www.feedparser.org/

* curl, featureful command-line HTTP client.
  http://curl.haxx.se/

* pycurl (libcurl integration for python).
  http://curl.haxx.se/libcurl/python/

* More links: http://microapps.org/Useful_Libraries

Appendix: Server tools

* selector
  http://lukearno.com/projects/selector/

* WSGI: http://www.python.org/dev/peps/pep-0333
  and http://www.wsgi.org/

* paste
  http://pythonpaste.org/

* your favorite framework:
  Django: http://code.google.com/p/django-rest-interface/
  Pylons: http://pylonshq.com/docs/module-pylons.decorators.rest.html
  Grok: http://grok.zope.org/documentation/how-to/rest-support-in-grok
  Turbogears: ... I don't know, send me a link!!
  Plone: http://tinyurl.com/3yze9a

Appendix: Troubleshooting HTTP

* TCPWatch, simple gui, in python
  http://hathawaymix.org/Software/TCPWatch

* tcpflow, simple command-line tool watches a network interface:
  http://www.circlemud.org/~jelson/software/tcpflow/

* wireshark, fancy GUI with all the bells and whistles:
  http://www.wireshark.org/

* Firefox live headers extension: http://livehttpheaders.mozdev.org/

Appendix: Real-world Services

* Gdata - Google APIs that build on the Atom protocol.
  http://code.google.com/apis/gdata/

* S3 - Amazon's commercial storage service

  http://developer.amazonwebservices.com/connect/kbcategory.jspa?categoryI