

# PyCon 2010, Atlanta

## Creating RESTful Web Services with restish

Grig Gheorghiu  
Evite

[grig.gheorghiu@evite.com](mailto:grig.gheorghiu@evite.com)

# Credits

- Roy Fielding Ph.D. Dissertation
- 'RESTful Web Services' book by L. Richardson and S. Ruby (O'Reilly)
- Joe Gregorio's REST articles for xml.com
- Mark Pilgrim's Web services tutorial in 'Dive into Python'
- Mark Nottingham's Web caching tutorials
- Tim Parkin and Matt Goodall for restish

# What is a web service?

- Specific kind of Web application
- Consumed by applications, not directly by humans/browsers
- Typically emits XML or JSON and not HTML
- Examples: Amazon Web Services (S3, SQS, EC2), Yahoo Web Services (del.icio.us, Flickr)
- 'Programmable Web':  
<http://www.programmableweb.com/apis/directory/>

# What is REST?

- "REpresentational StateTransfer": Roy Fielding, Ph.D. Dissertation, UC Irvine, 2000
- 'Architectural style for distributed hypermedia systems'; start with NULL style and add constraints:
  - Client-Server
  - Stateless
  - Cache
  - Uniform Interface
  - Layered System

# HTTP crash course

- Requests and responses
- HTTP request: a method (GET, POST, PUT, DELETE, HEAD), a path (/index.html), a set of headers, and the entity-body (representation)
- HTTP response: a response code (200, 404, etc.), a set of headers, and the entity-body (representation)
- Lingua franca of the Internet

# Web architectures

- REST(ful)
  - Based on HTTP methods (GET, POST, PUT, DELETE)
  - Arguments to methods go in URIs or in HTTP payloads
  - Amazon S3, last.fm API
- RPC-style (SOAP, XML-RPC)
  - Typically one single URI is exposed
  - Heavy on POST operations
- Hybrid REST-RPC
  - Overloaded GET used to modify data
  - Bad idea (remember Google Web Accelerator?)

# Resource-Oriented Architecture

- Leonard Richardson and Sam Ruby
- 4 concepts: resources, their names, their representations, the links between them
- 4 properties: addressability, statelessness, connectedness, uniform interface

# Resources and URIs

- Resource == anything that is important enough to be referenced (or linked to)
- Comparable to objects in OO design
- Every resource has a 'name': URI == Uniform Resource Identifier
- 1 resource → N URIs
- Examples of URIs:
  - URL(ocator): <http://social-piggies.org/scp>
  - URN(ame): <urn:isbn:0-395-36341-1>

# Representations and links

- Representation == concrete data which describes the current state of a resource
  - Example: description of the most recent bug
- In practice: a specific data format (XML, JSON, etc)
- Client-server state transfer happens by exchange of representations (hence "**RE**presentational **S**tate **T**ransfer")
- Server guides client from one state to another by means of links embedded in representations

# Addressability

- Resources exposed as URIs are addressable
  - Can be linked to
  - Can be bookmarked
  - Can be printed in a book
  - Can be sent in an email
  - Can be cached
- Openness vs. Opacity; REST vs. RPC
- Think 'mashups'!

# Statelessness

- Each HTTP request contains all the state it needs for the server to fulfil it
- State is only on the client side!
- Server can nudge client to different states via links (connectedness)
- Statelessness induces:
  - Visibility: each request can be monitored in isolation
  - Reliability: easier to recover from partial failures
  - **Scalability**: easy to scale horizontally

# The Uniform Interface

- Methods dictated by HTTP verbs (uniformity)
- CRUD operations
  - Create: POST/PUT
  - Retrieve: GET (HEAD for resource metadata)
  - Update: PUT
  - Delete: DELETE
- GET and HEAD should be safe (no changes to server state)
- GET, HEAD, PUT should be idempotent

# So...why REST?

- Client-Server (Web client ↔ Web server)
- Stateless (client keeps state)
- Uniform interface (HTTP methods)
- Caching
  - Reduces latency
  - Reduces network bandwidth
- Layered system (higher application layer calls lower Web services layer)

# HTTP caching in a nutshell

- Special HTTP headers
- Server: freshness indicators ("it's safe to cache")
  - Cache-control → absolute date/time
  - Expires → delta relative to time of request
- Server: validators ("is it still safe to cache?")
  - Last-Modified → absolute date/time
  - Etag → hash of content
- Client: If-Modified-Since (for Last-Modified)
- Client: If-None-Match (for Etag)

# Creating a RESTful service

- Joe Gregorio article "How to create a REST protocol"
- To create a REST service, you need to answer the following questions, and you should answer them in this order:
  - What are the URIs? (resources)
  - What's the format? (representations)
  - What methods are supported at each URI?
  - What status codes could be returned?

# Important rule of thumb

- When thinking about your resources and your URI scheme, think 'NOUNS' and not 'VERBS'!
- The verbs/methods are already specified via the HTTP methods you're using
- No '/do/something' but 'POST to /something' or 'PUT to /something'

# Introducing the restish framework

- Created by Tim Parkin and Matt Goodall
- Part of the 'ish' family of tools at <http://github.com/ish>
  - Formish
  - Adminish
  - Couchish
- Example site developed with restish: [www.profiled.com](http://www.profiled.com)

# Why restish?

- Close-to-the-metal, REST-oriented, WSGI aware
- Well tested and powerful URL handling, parsing and creation
- Powerful but simple HTTP request and response handling (based on WebOb)
- Simple integration with templating frameworks (mako, jinja2, genshi, django)
- Basic HTTP auth via guard module (e.g. using repoze.who as WSGI middleware)

# Demo app: todoish list mgmt

- What are the URIs? (resources)
  - /lists/LISTID/items/ITEMID
- What's the format? (representations)
  - JSON
- What methods are supported at each URI?
  - POST, PUT, GET
- What status codes could be returned?
  - Success: 200 OK, 201 Created
  - Error: 404 Not Found, 501 Not Implemented

# Consuming Web services

- Curl

```
curl -i -X POST 127.0.0.1:9999/lists/ -H "Content-Type: application/json" -d '{"name": "new list"}'
```

- Httpplib2

# Testing a restish Web app

- Unit testing: WebTest
- Functional testing: twill (for HTML representations)

# Deploying a restish Web app

- Run web application process properly as a daemon (e.g. using grizzled.os)
- Use log rotation (rotatelogs, scribe)
- Use automated deployment tools (fabric)
- Use robust reverse proxy/load balancer (e.g. nginx)

# What should you take away?

- blah blah blah RESOURCES blah blah blah  
HTTP UNIFORM INTERFACE blah blah  
STATELESSNESS blah blah JSON/XML blah  
blah RESTISH blah blah TESTING blah blah  
AUTOMATED DEPLOYMENTS blah blah  
EVITE IS HIRING PYTHON PROGRAMMERS