What Makes RESTful Services Different
&
The Resource-Oriented Architecture

Kenneth M. Anderson
University of Colorado, Boulder
CSCI 7818 — Lecture 7 — 10/08/2008

Credit Where Credit is Due

- Portions of this lecture are derived from material in “RESTful Web Services” by Leonard Richardson & Sam Ruby. As such, they are Copyright 2007 by O’Reilly

Agenda

- Discuss Semester Project
- Presentation from Susan Philipose
- One other student presentation?
- Chapter 3: What Makes RESTful Services Different (S3)
- Chapter 4: Resource-Oriented Architecture

- Skipping Chapter 2 due to laptop issues, out-of-date code, lack of time
  - Sorry!

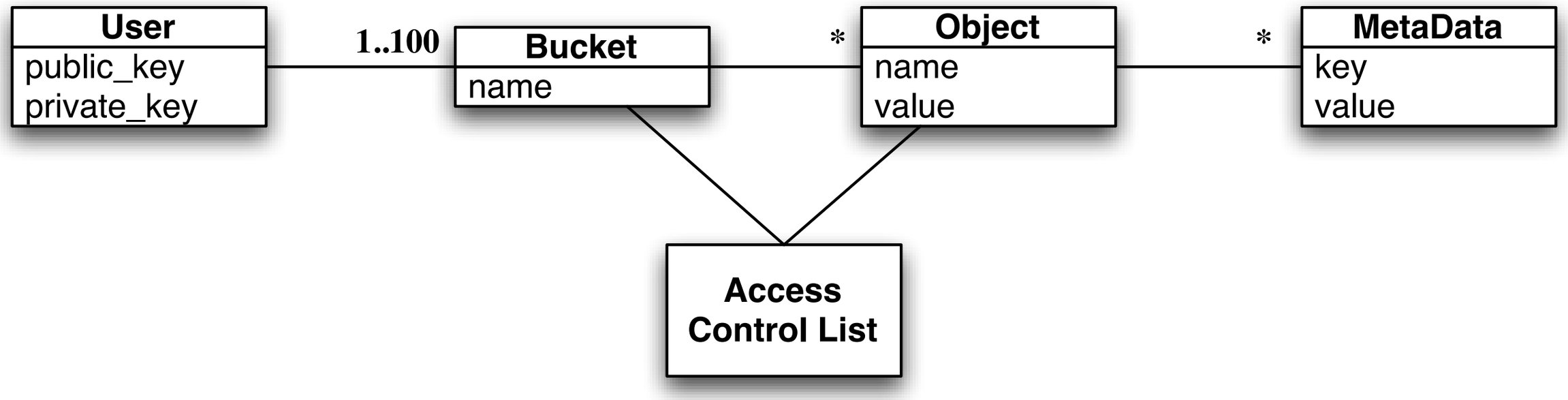
What Makes RESTful Services Different?

- From Last Week: Three Styles of Web Services
  - RPC
  - RESTful
  - REST-RPC Hybrid
- Examples of REST in earlier chapters were REST-RPC Hybrids
  - Flickr's use of GET to "delete" a resource, for instance
- To provide insight into how to DESIGN a RESTful service, we need to look at a fully RESTful example
  - Atom Publishing Protocol or S3
    - The former is more a set of guidelines; so lets look at S3

Amazon's Simple Storage Service (S3)

- S3 is a service for storing data
  - You can use it as a backup service
    - keep all data private, pay Amazon for storage/bandwidth
  - You can use it as a data host
    - Store data on S3 but let other people access it
      - still pay Amazon for storage/bandwidth but at greatly reduced costs when compared to typical ISP fees
- Interesting service since there is no human-readable equivalent website
  - Flickr API is based on Flickr; Delicious API based on Delicious
  - No corresponding "human friendly" site for S3; its pure service

S3 Object Model



User's can create buckets and put objects in them. Objects can have meta-data associated with them

The names of buckets and objects are significant since they help to generate URIs that can be used to access the data (addressability)

MetaData is used to annotate the Object; these key-value pairs are significant since some of them are used to generate HTTP Headers when their corresponding object is retrieved

Each bucket and object has an access-control list that governs access

Correspondence to URIs

- Given bucket with name “media.colorado.edu”
- Given object with name “CSCI_7818_F08_Lecture_7.mov”
  - with an QuicktTime movie as its value
  - with a metadata element “Content-Type: video/quicktime”
- Amazon will make this object available at:
  - http://<bucket_name>.s3.amazonaws.com/<object_name>
  - or rather
  - http://media.colorado.edu.s3.amazonaws.com/CSCI_7818_F08_Lecture_7.mov
- I could then include that URI in my webpage at www.cs.colorado.edu and could then serve my lectures to students at a greatly reduced cost

S3 Resources and Operations

	GET	HEAD	PUT	DELETE
Bucket List (/)	List your buckets			
Bucket (/{bucket})	List bucket's objects		Create bucket	Delete bucket
Object (/{bucket}/ {object})	Get object's value and metadata	Get object's metadata	Set object's value and metadata	Delete object

Note: There are others; for instance POST object to bucket

Things Change

- Alas, I ran into problems getting the code in the textbook to work
  - Amazon has changed its API to handle
    - virtual hosts (http://www.{bucketname}.com/{object})
    - multiple locations: US, Europe, etc.
      - browser-based uploads using POST
  - and the book's code has not been updated
- I then ran into various problems with Ruby libraries trying to get chapter 1 examples to work (Sigh)
- I did get more recent code to work by downloading examples from Amazon sample code section

Costs of S3

- Storage: 15 cents per GB-Month of storage used
- Data Transfer:
  - 10 cents per GB of data transfer in
  - 17 cents per GB of data transfer out for first 10 TB/Month ⇐ terrabytes!!
    - 13 cents per GB for next 40 TB, 11 cents for next 100 TB; 10 cents rest
- Requests
  - 1 penny per 1000 PUT, POST, or LIST requests
  - 1 penny per 10000 GET and all other requests
  - No charge for delete requests (gives space back to Amazon!)

Official Info

- S3 Home Page: <http://aws.amazon.com/s3/>
- S3 Docs: <http://docs.amazonwebservices.com/AmazonS3/2006-03-01/>
- S3 Getting Started:
  - <http://docs.amazonwebservices.com/AmazonS3/2006-03-01/gsg/>
- S3 Resources:
  - <http://developer.amazonwebservices.com/connect/kbcategory.jspa?categoryID=46>
- S3 Sample Code
  - <http://developer.amazonwebservices.com/connect/kbcategory.jspa?categoryID=47>

Resource-Oriented Architecture

- A systematic way to structure and expose the power of REST in a Web service
  - Key concept
    - Resources (obviously)
      - with names, representations, and links between them
  - ROA Properties
    - addressability (scoping information from last lecture)
    - statelessness
    - connectedness
    - uniform interface (method information from last lecture)

What's a Resource

- Anything (within reason) with at least one URI
- The URI provides a name and address
  - If some data does not have a URI, it can't be a resource
- Example Resources and their URIs
  - Version 1.0.3 of a system
    - <http://www.example.com/software/releases/1.0.3.tar.gz>
  - The latest version of a system
    - <http://www.example.com/software/releases/latest.tar.gz>
  - A directory of resources about jellyfish
    - <http://www.example.com/search/Jellyfish>
  - The next prime number after 1024
    - <http://www.example.com/nextprime/1024>

Relationship Between URIs and Resources

- No two resources can be the same
  - but two resources CAN point at the same data
    - example: latest software release == 1.0.3
  - key point: the ideas behind the resources are different
  - Analogy: `[head = node1; tail = head;]` Two variables, one value
- A resource can have more than one URI
  - <http://www.example.com/Q42004>
  - <http://www.example.com/2004/Q4>
- Makes it easier for clients to find resource but dilutes the value of each
- Finally, a resource once retrieved, may contain URIs to other resources
  - indeed it SHOULD contain such URIs (see connectedness)

Addressability

- A Web service is addressable if it exposes the interesting aspects of its data set as resources
  - resources are exposed via URIs
    - applications can have an infinite number of URIs
      - Google search has one URI per search request
- Addressability enables information sharing and caching
  - Structured URIs allow users/clients to guess where information is located
- Web 2.0 and websites with frames can kill addressability, limiting utility
  - Think AJAX-powered site where URI never changes
  - Gmail's AJAX website is not addressable, turn it off, addressability returns

Statelessness (I)

- As we've discussed previously, one aspect of statelessness is that every request on a RESTful web service should be self contained
    - Contrast with FTP: more complex protocol because client and server have to be synched
        - cd public/software; get doom.exe
    - is different than
        - cd public; get doom.exe
    - One request might succeed, the other fail. Both client and server have to know that cd commands alter the effects of other commands such as put and get

Statelessness (II)

- But there's another aspect of state: consider statelessness in terms of addressability
  - Addressability says that all of an application's interesting data should be exposed as a resource and given a URI
  - Statelessness then says that the possible states of a service are also resources and should be given their own URIs
    - The client should not have to coax the server into a certain state to make it receptive to a certain request
      - GET ftp://ftp.games.com/public/software/doom.exe
- The book discusses an example of stateless search engine and stateful search engine; the latter might make it easier to handle multiple pages of search results, but the complexity of the protocol would go up
  - e.g. issue request, get results, ask for next results, etc. (requires server to remember the first request)

Statelessness (III)

- Two types of state
  - application state stored on client
  - resource state stored on server
- To achieve statelessness, client must send all relevant bits of application state in each request
  - http://www.google.com/search?q=jellyfish&start=10
    - I'm searching on jellyfish; give me search results 10 – 20
- How do you break statelessness
  - server-side sessions enabled via cookies and session ids
  - pull that stuff out of the server and put it into the client (if possible)

Statelessness (IV)

- Benefits of Statelessness
  - easy to distribute stateless application across multiple servers
    - since no two requests depend on each other
      - they can be handled by two servers with no need for coordination
  - scaling is easy: add more servers
  - any response can be cached since it does not depend on previous responses
- If you break statelessness, these characteristics become harder to accomplish
  - in particular, a need for session replication and affinity occurs increasing complexity

Representations

- A resource can have multiple representations
  - A book: the contents of the book, metadata about the book
  - A press release: english version/spanish version
  - A set of data: html version, xml version, csv version etc.
- Each representation should have similar URIs perhaps varying by suffix
  - That way the URI contains everything needed by the server to produce the desired representation
- You can also leave the URI “pure” and then request a representation via content-negotiation
  - http://example.com/pr/104 vs. http://example.com/pr/104.es

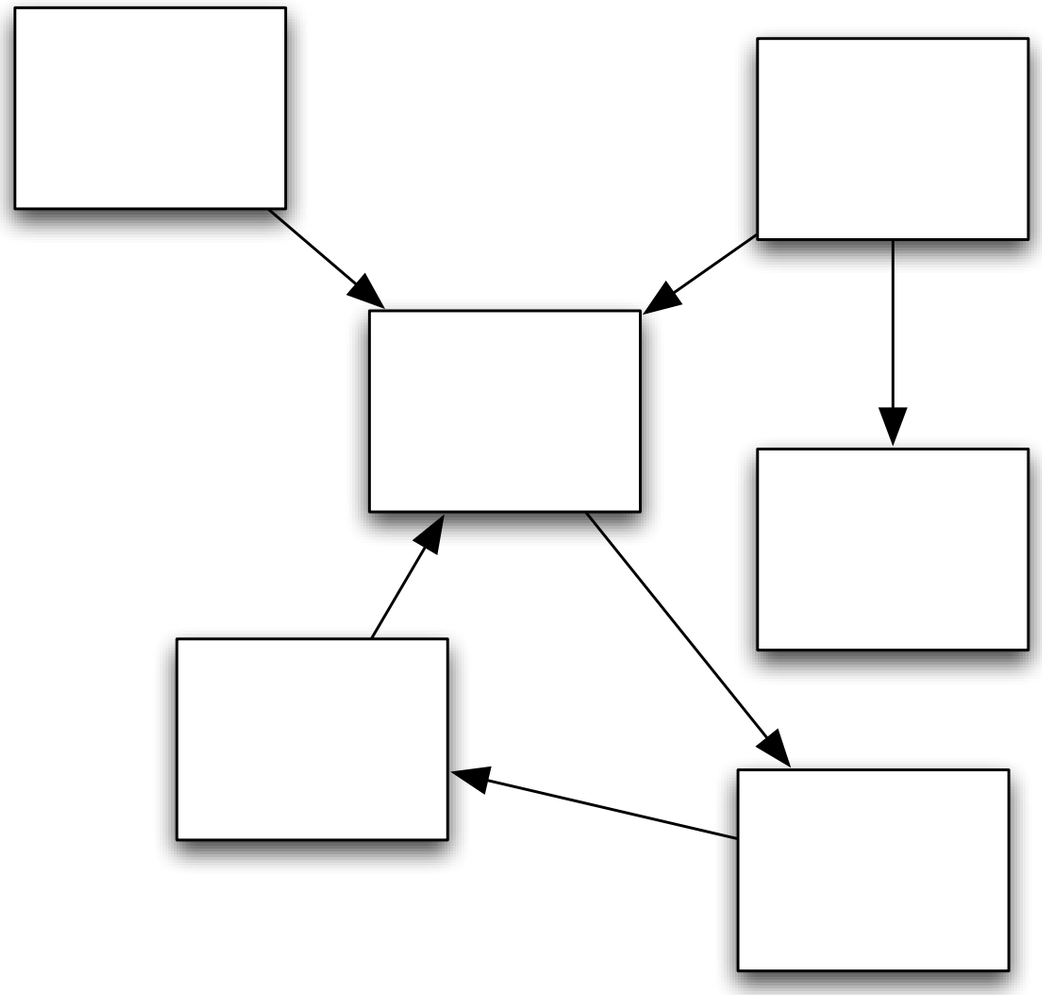
Links and Connectedness

- Resources should contain links to related resources
  - enables connectedness
- From last lecture
  - “The use of Hypermedia both for Application Information and State Transitions”
- Combine with Statelessness
  - Retrieving a resource puts the client in a particular state
    - Links in the resource point to “states that are near the current one”
- The human web is easy to use because its well connected
  - a similar form of usability can be conveyed to services in the same way

Range of Usability

RPC Style

single URI; not addressable; not connected



RESTful

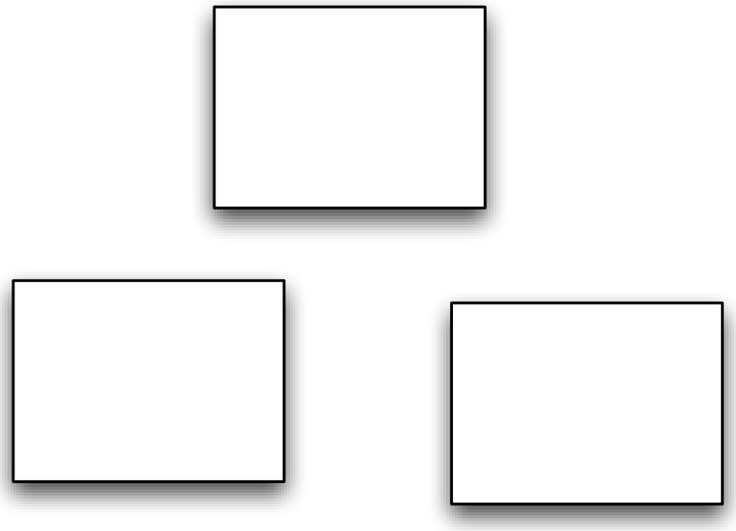
multiple URIs; addressable; connected

multiple URIs; addressable; not connected

REST-RPC Hybrid

or

RESTful service like S3



Uniform Interface

- RESTful services have a uniform interface defined by HTTP's primary methods
  - POST causes problems as we will see
- Retrieve a Representation of a Resource
  - GET
- Create a new Resource
  - PUT to new URI; POST to an existing URI (create child)
- Modify an existing Resource
  - PUT to existing URI
- Delete a Resource
  - DELETE

GET, DELETE, and PUT

- GET and DELETE have obvious semantics
- To create or modify resource, use PUT and
  - send a proposed new representation for the resource
  - This is when application state (client) turns into resource state (server)
- Again, need “smart” clients
  - When modifying a resource, the new representation overwrites the old representation on the server

HEAD and OPTIONS

- HEAD
  - retrieve metadata about a specified resource
- OPTIONS
  - request information about the methods a resource supports
  - is delivered in HTTP’s Allow header
  - not well supported

POST

- POST has two purposes, one that fits within REST, the other with RPC
  - POST (a): “append”
    - append information to this resource
    - create a new child of this parent resource
  - POST (p): “process”
    - most common form of post: fill out form, click submit, post data to a “data handling process”, wait for result
    - aka, overloaded POST: I’m using POST but the service isn’t uniform
      - i.e. each HTML form invokes a different “data processing service”
      - the method information is contained in the URI, headers, or body

RESTful POST

- Annotate existing resources or create new subordinate (child) resources

	PUT new resource	PUT existing resource	POST existing resource
/weblogs	N/A resource exists	No effect	Create new weblog
/weblogs/bees	Create this weblog	Modify weblog	Create new entry
/weblogs/bees/entries/12478	N/A no way to guess URI	Edit this entry	Post a comment on this entry
/log	Create log	Overwrite log	Add new log entry

Safety and Idempotence (I)

- If you expose HTTP's uniform interface in the way it was designed you gain
  - Safety
    - GET or HEAD is a request to READ data
      - not a request to change server state
    - Side effects should be non-existent or minimal (hit counters)
  - Idempotence
    - Math example: 4 x 0 x 0 x 0 == 4 x 0
      - idempotent operation has same effect no matter how many times it is applied
    - HTTP example:
      - an operation on a resource is idempotent if making one request is the same as making a series of requests
        - PUT and DELETE are idempotent

Safety and Idempotence (II)

- Why does these properties matter?
  - Safety and Idempotence let a client make reliable HTTP requests over an unreliable network
    - If you make a GET request and get no response, make another one
      - Its safe
    - If you make a PUT request and get no response, make another one
      - The second request will have no additional effect even if the first one DID get processed
- POST is neither safe nor idempotent
- Misuse of GET (like using GET to delete on Flickr) is dangerous:
  - Google Web Accelerator: first deployment deleted data

Benefits of Uniform Interface

- Service-specific code of a client goes into handling resource representations and URIs
    - GET, POST, PUT, DELETE, HEAD, and OPTIONS just behave as expected
- Think of benefits gained from uniform UI standards
    - Cut, Copy, Paste work the same across multiple platforms/applications
        - No relearning of skills; UI stays in background allowing user to focus on task

Coming Up Next

- Chapter 5: Designing Read-Only ROA Services
- Chapter 6: Designing Read/Write ROA Services