Reference

Richardson, Leonard and Sam Ruby. 2007. *RESTful Web Services: Web Services for the Real World*. Sebastopol, CA: O'Reilly Media, http://oreilly.com/catalog/9780596529260

Fielding, Roy. 2000. "Representational State Transfer (REST)." Chapter 5 in *Architectural Styles and the Design of Network-based Software Architectures*. Irvine: University of California, http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

JAX-RS: Java™ API for RESTful Web Services. https://jsr311.dev.java.net/nonav/releases/1.1/spec/spec.html

URI Templates. http://code.google.com/p/uri-templates/

JSONP: JSON with padding. http://bob.pythonmac.org/archives/2005/12/05/remote-json-jsonp/

Microformats in REST Web Services. http://microformats.org/wiki/rest

Resources

Jersey: JAX-RS Reference Implementation. https://jersey.dev.java.net/

XStream: simple library to serialize objects to XML and back again. http://xstream.codehaus.org/

HSQLDB: Lightweight 100% Java SQL Database Engine. http://hsqldb.org/

Jetty: HTTP server, HTTP client, and javax.servlet container. http://www.mortbay.org/jetty/

Restlet: RESTful Web framework for Java. http://www.restlet.org/

GWT: Google Web Toolkit. http://code.google.com/webtoolkit/

About the Presenter

The quantum super computer in Tim Taylor's brain has discovered the Ultimate Question, the answer to which is 42. All attempts to coalesce the answer result in an overwhelming urge to swim in the ocean and a ravenous appetite for sushi.

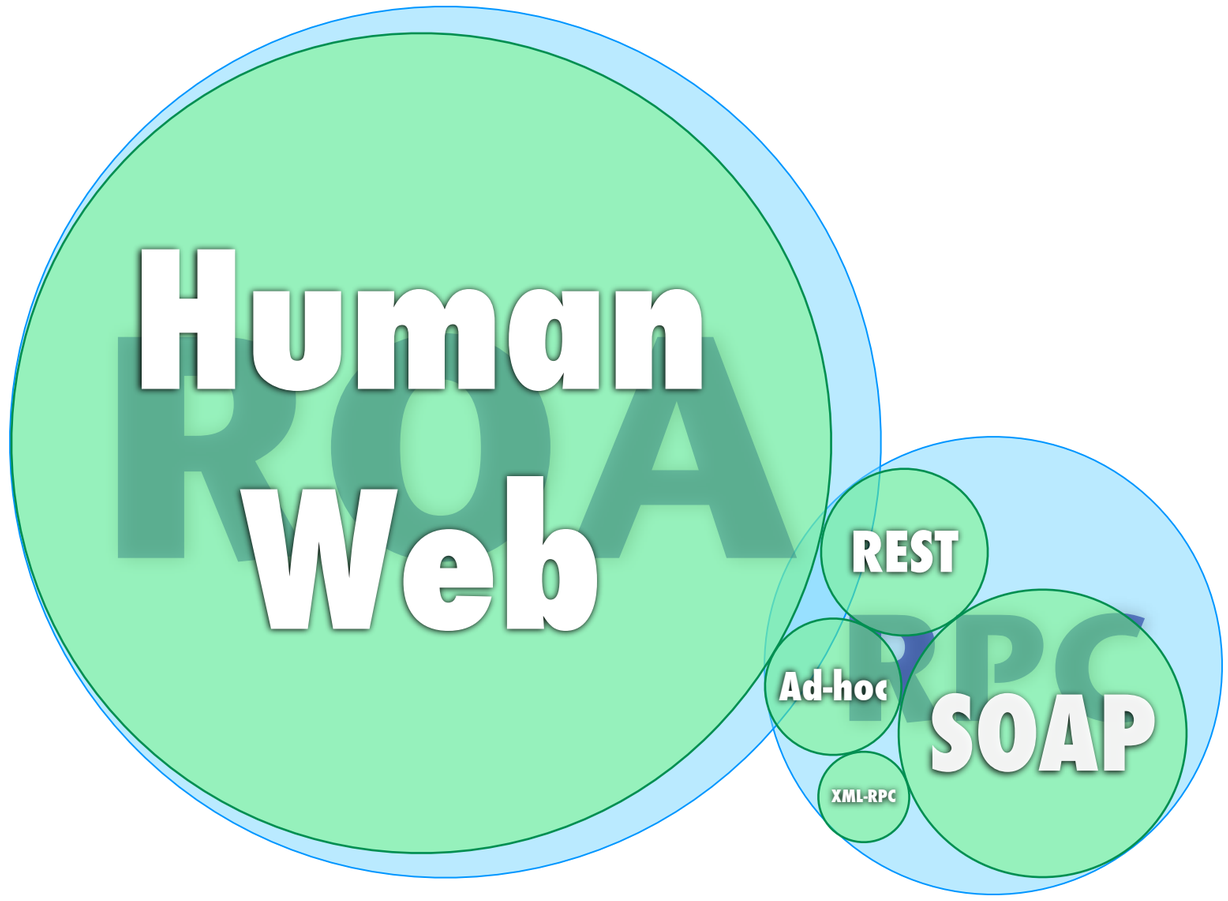
Tim returned to Java after a few years in the Perl wilderness. His travels abroad gave him an appreciation for dynamic languages, a disdain for boilerplate, and a renewed love of good tooling.

As a member of ePrize's Engineering team, Tim works to expose ePrize software as RESTful services for internal and client use. Tim knows enough about REST to pass the Michigan driver exam, but he learned it all from those voices in his head. *Shhh. I told you to be quiet.*

RESTful Web Services

and the

Resource Oriented Architecture



REST and the ROA: Five Minute Primer

Terminology

Remote Procedure Call (RPC)	Architectural style modeled on the function/method invoking techniques of procedural programming languages, e.g. method calls across the web. RPC services have wide variation in their interfaces.
Representational State Transfer (REST)	Architectural style described by Roy Fielding in his doctoral dissertation published in 2000. He showed that desirable traits of the human web were applicable to the programmable web.
The Uniform Interface	The limited, consistent interface of the standard HTTP methods: GET, DELETE, HEAD, OPTIONS, POST, and PUT. All REST services share this interface.
Resource	"The key abstraction of information in REST is a resource" (Fielding 2000). Resources are the nouns in the system that are important enough to be treated as standalone entities. In the example of a blog web service, the list of blog articles, an author, and an individual blog article would all be resources. The 3rd sentence in the 5th paragraph in the blog article "Making Ajax Crawlable" may — depending on the data model — be a noun in the system but wouldn't be a resource.
URI	The address of a resource on the web. The same resource can have multiple URIs. A resource must have at least one URI (Richardson and Ruby 2007, 93).
Representation	The data of a resource in a particular format: XML document, JSON structure, PNG graphic. The same resource can have multiple representations. For example, a time-series data set of visits to a web site can have a representation as a PDF report in English, another as the report in German, and a third representation as the raw data in `text/csv`.
Resource Oriented Architecture (ROA)	Architectural style described in *RESTful Web Services* (Richardson and Ruby 2007). It is REST with the added constraint that scoping information — the indicator of which part of the data set to operate on — is in the URI. Contrast that to SOAP which passes the scoping information in the entity-body.

RPC versus ROA

RPC	ROA
Verb heavy. Many procedures.	Noun heavy. Many resources. Only six verbs.
Natural to programmers who work in procedural languages.	Natural to programmers who work with document or resource-oriented web sites
Reinvents or works against features of the Web.	Builds on and relies on features of the Web.
Wide variation and little uniformity in interfaces across different services.	All REST services use the Uniform Interface.

Comparison of RPC and ROA Architecture Styles

Replace stubs in UserResource with DB backed implementation

Start from	roa-workshop-03
Overview	An HSQLDB database and `UserDAO` have been added behind the scenes. Replace stubbed methods with real behavior by searching for and creating real users via GET and PUT methods. Handle error scenarios such as user doesn't exist or attempting to create a user that already exists.

Require HTTP Basic Auth to change a user password

Start from	roa-workshop-04
Overview	Require an authenticated PUT request to create a new user. A MockLoginService has been behind the scenes. The `@Context` annotation and `SecurityContext` interface are the JAX-RS facade to authentication.

Tagging someone in a photo

Start from	roa-workshop-06
Overview	`TagDAO` and related DAOs have been added behind the scenes. Tag creation is done via authenticated POST. Use "Factory Resource" pattern; URI of the newly created tag returned via the `Location` header.

Viewing a tag

Start from	roa-workshop-07
Overview	Add an XML representation to a tag resource, constructed from the data collected when the tag was created. The final representation includes links from the tag resource to the photo resource it's tagging.

The Photo Resource

Start from	roa-workshop-08
Overview	Add an XML representation for a photo resource. The representation includes the URL to the actual image on the web, plus all the tags created against that image. Close the loop by including links to the tag resources that are listed.

The Beginning

A presentation and workshop can only scratch the surface of any topic. I've succeeded if you've come away knowing more about the subject of REST and the Resource Oriented Architecture than when you walked into the room a few hours before. I've succeeded stupendously if I've armed you with enough information and instilled you with enough curiosity to take a deeper dive. See References and Resources for the starting points that I found helpful when I embarked on my RESTful journey.

```
Server: Jetty(7.0.1.v20091125)

Hello ROA Workshop Attendees
```

You're all set!

Workshop Eclipse Projects

roa-workshop-dependencies	contains and exports required libraries: Jersey, HSQLDB, Jetty, etc.. The roa-workshop-## projects depend on this.

The projects below capture the evolution of our service over time. They implement the same service, with progressively more features complete from our design (think "cooking show" where the host pulls a finished dish from the oven minutes after putting it in).

roa-workshop-01	Starting point. A RESTful "Hello World"
roa-workshop-02	UserResource with stub behavior for GET and PUT HTTP methods
roa-workshop-03	HSQL database and rudimentary User DAO
roa-workshop-04	UserResource with real behavior and persistence
roa-workshop-05	Edit an existing User resource. Require "mostly fake" authentication.
roa-workshop-06	Tag and Photo DAOs. Stub "factory" resource to create tags via `POST`.
roa-workshop-07	Real behavior and persistence for creating tags.
roa-workshop-08	Represent a Tag resource as designed.
roa-workshop-09	Represent a Photo resource as designed: list people tagged in the image.

Workshop Outline

The following is an overview of each segment of the workshop. Tim has a "presenter friendly" script with more detail on each code change which he can provide on request. The "Start from:" indicates which Eclipse project you should begin the segment in.

Hello Workshop

Start from	roa-workshop-01
Overview	Introduce project structure, embedded Jetty as a REST server, and the Sandbox app for interactively exercising the REST service.

Our first resource: UserResource

Start from	roa-workshop-01
Overview	Implement GET and PUT methods stubs to begin implementing the User resource as designed. Use JAX-RS annotations to extract URI template values. Add a createUser method which accepts PUT requests.

The ROA Procedure By Example

Procedure for splitting a problem into RESTful resources (Richardson and Ruby 2007, 223-224) .

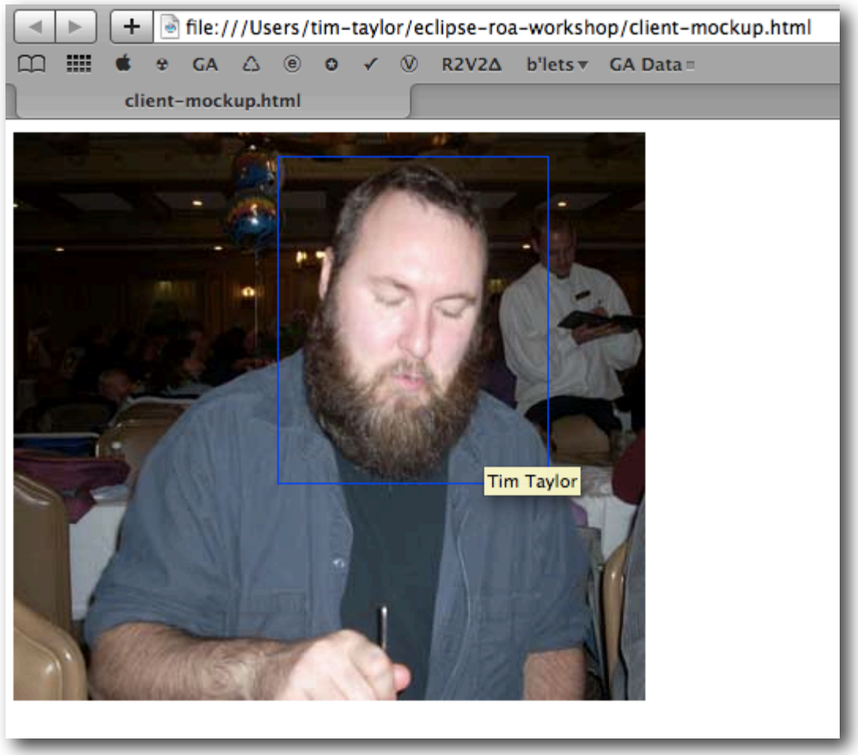
1. Figure out the data set
2. Split the data set into resources

For each kind of resource:

3. Name the resources with URIs
4. Expose a subset of the uniform interface
5. Design the representation(s) accepted from the client
6. Design the representation(s) served to the client
7. Integrate this resource into existing resources, using hypermedia links and forms
8. Consider the typical course of events: what's supposed to happen?
9. Consider error conditions: what might go wrong?

The Problem Space

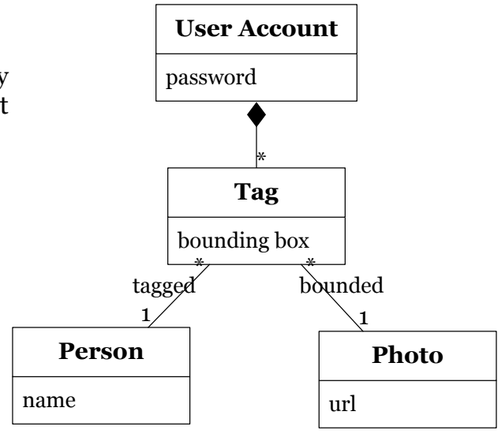
A RESTful, resource-oriented service for tagging photos on the web with the name of a person in the photo.



Mockup of possible UI built using this service.

Figure Out the Data Set

Users tag existing photos on the web. They tag a photo by outlining a face or person and associating a name to that person. Users must have an account to create or modify tags.



Split the Data Set Into Resources

User Account - user who creates and edits tags

Photo - URL to a photo to be tagged

Person - someone tagged in a photo

Tag - coordinates of a shape outlining a person in a photo

Name the Resources With URIs

User account - `http://api.nameta.gs/v1/user/{user-name}`

Photo - `http://api.nameta.gs/v1/photo/{uri-hash}`

Person tagged in a photo - `http://api.nameta.gs/v1/photo/{uri-hash}/{person-name}`

Tags you created - `http://api.nameta.gs/v1/user/{user-name}/tags`

Photos X is tagged in - `http://api.nameta.gs/v1/person/{person-name}`

Reconsidering `/user/` and `/person/` URIs

A user is someone who creates and edits tags of people. What if a you tag yourself or are tagged by someone else? Now the system has two URIs addressing you. Is this a problem? Not necessarily.

There is only one you, but two different representations of you will exist: your account with the tags you own and the photos you've been tagged in. The ROA prefers distinct URIs for different representations. The drawback is dilution: the same resource with different URIs can look like different resources. We can mitigate this by including links between one or both representations.

Expose a Subset of the Uniform Interface

Services consisting entirely of read-only resources need only expose those resources via GET. Our service will allow clients to create new resources. For that we expose PUT or POST methods (and likely GET as well). The ROA provides this guideline for determining which method to use.

URI is composed of constant values determined by the client → client PUTs to that URI

Server determines the URI of a new resource → client POSTs to a factory or parent URI

Our service will have both examples of resources created via PUT and created via POST.

User Account

New user accounts will be created with PUT. The only variable part of the URI is `user-name`, which the client knows. We will never return the password, but we'll still provide a GET method so the client can check if a user name is taken.

Implementing Our Service

Tools

- Eclipse IDE (3.4+)
- Jetty 7
- Jersey/JAX RS
- HSQLDB

Getting Started

Download

roa-workshop.zip, http://bit.ly/bvCkNg (5 MB)

Import Projects into Eclipse

1. Download roa-workshop.zip
2. Launch Eclipse. It is prefered, but not required, to switch to a brand new workspace
3. Select "File" → "Import"
4. Under "General" choose "Existing Projects into Workspace". Click "Next"
5. Choose "Select archive file: " radio button, click "Browse..."
6. Locate roa-workshop.zip and choose it
7. Make sure projects roa-workshop-01 through roa-workshop-09 and roa-workshop-dependencies are checked
8. Click Finish

Verify

The workshop projects rely on running an HTTP server on port 8080. The ROA Dependencies JUnit tests require port 8081. If you have anything running on either of these ports, you wil need to shut it down to follow the examples and run the tests.

1. Run "Test ROA Dependencies", either from Eclipse run history or by opening roa-workshop dependencies and running "Test ROA Dependencies.launch"
2. All JUnit tests should pass
3. Run RestServer-01. Console should show succesful startup without exceptions similar to this (the `NO JSP Support for /` error is safe to ignore):

```
2010-03-11 02:44:12.775:INFO::Logging to StdErrLog::DEBUG=false via
org.eclipse.jetty.util.log.StdErrLog
2010-03-11 02:44:12.818:INFO::jetty-7.0.1.v20091125
2010-03-11 02:44:12.967:INFO::NO JSP Support for /, did not find
org.apache.jasper.servlet.JspServlet
2010-03-11 02:44:13.422:INFO::Started SelectChannelConnector@0.0.0.0:8080
```

4. Run Sandbox-01. In the Eclipse console it will prompt for a username. Any username will do.
5. After answering the username prompt, the console for Sandbox-01 will show output like this:

```
username: tim-taylor
GET http://localhost:8080/v1/user/tim-taylor
200 OK
Transfer-Encoding: chunked
Content-Type: text/plain
```

Similarly the photo resource can link back to the tag resource.

```
<?xml version="1.0" encoding="utf-8"?>
<photo>
    <img src="http://billandval.com/en/travel/pictures/RoyFieldingAndBillPeer.JPG"/>
    <area shape="rect" coords="133,47,190,119"
        href="http://api.nameta.gs/v1/photo/8swz9nhp…">Roy T. Fielding</area>
    <area shape="rect" coords="230,70,284,133"
        href="http://api.nameta.gs/v1/photo/3bde9x44…">Bill Peer</area>
</photo>
```

We could go one step further and link the Person to the list of other photos they are tagged in.

```
    <area shape="rect" coords="133,47,190,119"
        href="http://api.nameta.gs/v1/photo/8swz9nhp…">
            Roy T. Fielding
            <link href="http://api.nameta.gs/v1/person/Roy%20T.%20Fielding"/>
    </area>
```

Consider the Typical Course of Events

There's no entity body for a user. A GET on an existing user will return "204 No Content". A GET on an available `user-name` will return "404 Not Found". New accounts are created by PUTing to an unclaimed `user-name` with a password which will return "201 Created" and include the URI of the user in the `Location` header (mostly out of correctness, since it's the same URI the client did the PUT to). Changing the password of an existing user is also accomplished by a PUT to the user resource, but requires that the client request be authenticated with basic auth. The response on edit will be "204 No Content" (alternatively we could have chosen "205 Reset Content".

The remaining resources have representations. Performing a GET on them will return "200 OK".

Tagging a user will be done by POST the URI of the photo, the shape coordinates, and the name of the person to the /photos factory. The service will calculate a unique hash of the photo URI to use in the photo resource's URI. On success the response will be "204 No Content" with the Location header set to the URI of the resource constructed using the hash.

Consider Error Conditions

Username already exists	Client attempts to create a user by PUTing to an account resource, and that username already exists. Respond with "409 Conflict".
Missing or invalid password when creating user	The client attempts to add a user account via PUT and omits the password in the entity body or the password fails to meet our password requirements. Respond with "400 Bad Request".
No authentication when creating a tag	Client request omits the `Authorization` header when creating or editing a resource that requires authentication (`/photos` or `/photo/{uri-hash}/{person-name}`). Respond with "401 Unauthorized" and include the `WWW-Authenticate` header indicating we accept Basic authentication. In the special case of doing a PUT /user/{user-name}, if `Authorization` is omitted from the request, we assume the client is attempting to create a user account and respond with "409 Conflict" instead.
Invalid username or password when attempting to authenticate	Client includes the `Authorization` header, but the username or password is invalid. Respond with "401 Unauthorized".

Verb and Resource	Action
`PUT /v1/user/{user-name}`	Create a new user account or change password of existing account
`GET /v1/user/{user-name}`	Check if account exists

Person Tagged in a Photo

The URI for a tag consists of two variable parts:

- Name of the person
- A unique hash of the URI to a photo elsewhere on the Web

The client knows the value for `person-name`. To use PUT the client would also need to know `uri-hash`. We could specifying a public algorithm like MD5 and require that the client compute it for the URI of the photo.

Instead tags will be created by POSTing to a factory URI, the server will compute the hash, and the response will include the URI of the newly created tag.

Verb and Resource	Action
`POST /v1/photos`	Tag a person in a photo
`GET /v1/photo/{uri-hash}/{person-name}`	Get bounding box (and any other information) stored in a tag
`PUT /v1/photo/{uri-hash}/{person-name}`	Change a tag
`DELETE /v1/photo/{uri-hash}/{person-name}`	Remove this tag

We've introduced the new factory URI, `/photos`. An equally valid design would be to POST to the parent URI of all photo resources, `/photo`.

You must have a user account to create or edit tags. We accomplish that by requiring that the client send a value user-name and password in the `Authorization` header on POST or PUT.

Remaining Resources

The remaining resources are derived from creating accounts and tagging people in photos. We can expose them as read-only resources via GET.

Verb and Resource	Action
`GET /v1/user/{user-name}/tags`	List all your tags
`GET /v1/photo/{uri-hash}` `GET /v1/photo(uri)?uri={photo-uri}`	List people tagged in photo
`GET /v1/user/{user-name}/tags`	List all your tags
`GET /v1/person/{person-name}`	List all photos person is tagged in

Design the Representations Accepted from the Client

Unless the data clients must send to the server is complex, the simplest representation format to accept is form-encoding: the same format web browsers use when POSTing forms to a web application.

Create a User Account or Change Password on Your Account

The two bits of state are `user-name` and `password`, with `user-name` conveyed in the URI and `password` sent form-encoded in the request body. The HTTP request will look something like this:

```
PUT /v1/user/{user-name} HTTP/1.1
Host: api.nameta.gs
Content-Type: application/x-www-form-urlencoded

password={password}
```

Tag Someone in a Photo

A tag of a person in a photo has this state:

- URI of the photo
- Name of the person within the outlined area
- Coordinates of a shape outlining an area in the photo

The representation for the URI and the name is straightforward: simple scalars. But how to represent the shape outlining an area in the photo? Consider:

- What shapes will we accept? Simple rectangles, basic shapes, or any polygon?
- How will the shapes be represented? A comma separated list of coordinates? A homegrown XML structure? An existing standard markup for shapes?

The only shape we will support will be rectangle, e.g. a bounding box. This can be represented as two coordinates (xmin, ymin) and (xmax, ymax); simple enough that we will use form-encoded name-value pairs for this resource, too.

```
POST /v1/photo HTTP/1.1
Host: api.nameta.gs
Content-Type: application/x-www-form-urlencoded

photo-uri=http://billandval.com/en/travel/pictures/RoyFieldingAndBillPeer.JPG&person-
name=Roy T. Fielding&xmin=133&ymin=47&xmax=190&ymax=119
```

Borrowing from Existing Standards

Prefer using existing standards in your representations over crafting your own vocabularies; use your judgement when the fit doesn't feel right. The goal is a service that's a pleasure to write software against, not creating a patchwork of XML standards.

For the client request, the representations accepted are in application/x-www-form-urlencoded format, the same format used by web browsers to submit forms via POST.

For the representations served in the responses, we've chosen an adhoc XML format. But we borrow from HTML in cases where there's no need for reinvention (`href` attributes, `img` elements, and others). Other standards, such as Atom, can be mined.

An alternative to borrowing HTML's area element as we did, we could have chosen SVG or gone with CSS-style properties of top, left, width and height to define the bounding box.

There's a wealth of options that may fit your purpose. Search for "micro formats" for more on this topic.

The same name-value pairs without form-encoding:

```
photo-uri=http://billandval.com/en/travel/pictures/RoyFieldingAndBillPeer.JPG
person-name=Roy T. Fielding
xmin=133
ymin=47
xmax=190
ymax=119
```

Design the Representations Served to the Client

User

Our user resource has no state to display; the username is already known by the client and we don't return the password.

Tag

```
<?xml version="1.0" encoding="utf-8"?>
<tag>
    <img src="http://billandval.com/en/travel/pictures/RoyFieldingAndBillPeer.JPG"/>
    <area shape="rect" coords="133,47,190,119">Roy T. Fielding</area>
</tag>
```

Photo

```
<?xml version="1.0" encoding="utf-8"?>
<photo>
    <img src="http://billandval.com/en/travel/pictures/RoyFieldingAndBillPeer.JPG"/>
    <area shape="rect" coords="133,47,190,119">Roy T. Fielding</area>
    <area shape="rect" coords="230,70,284,133">Bill Peer</area>
</photo>
```

Photos X Is Tagged In

```
<?xml version="1.0" encoding="utf-8"?>
<person name="Roy T. Fielding">
    <photo href="http://api.nameta.gs/v1/photo/8swz9nhp…"/>
    <photo href="http://api.nameta.gs/v1/photo/1x5mdr7l…"/>
</person>
```

List of Tags

```
<?xml version="1.0" encoding="utf-8"?>
<tags>
    <tag href="http://api.nameta.gs/v1/photo/8swz9nhp…/Roy%20T.%20Fielding"/>
    <tag href="http://api.nameta.gs/v1/photo/emvy9zer…/Sam%20Ruby"/>
</tags>
```

Integrate This Resource Into Existing Resources

The Tag resource includes the URI to the image on the web that is being tagged. It could also include a link to the photo resource that it's tagging.

```
<?xml version="1.0" encoding="utf-8"?>
<tag>
    <img src="http://billandval.com/en/travel/pictures/RoyFieldingAndBillPeer.JPG"/>
    <area shape="rect" coords="133,47,190,119">Roy T. Fielding</area>
    <link rel=”photo” href=” http://api.nameta.gs/v1/photo/8swz9nhp…”/>
</tag>
```