
# Designing RESTful Web Applications

Ben Ramsey ▪ php|works ▪ September 13, 2007

# About Me:
## Ben Ramsey

- Proud father of 7-month-old Sean
- Organizer of Atlanta PHP user group
- Founder of PHP Groups
- Founding principal of PHP Security Consortium
- Original member of PHPCommunity.org
- Author, Speaker, & Blogger
- Software Architect at Schematic

# Overview

- What Is REST?
- The REST Interface
  - Verbs
  - Content Types
- RESTful Design
- Things To Consider
- RESTful Web Services

# What Is REST?
## Representational State Transfer

- Term originated in 2000 in Roy Felding's doctoral dissertation about the Web entitled "Architectural Styles and the Design of Network-based Software Architectures"
- Strict: collection of architecture principles for defining and addressing resources
- Loose: any simple interface that transmits data over HTTP without an additional layer such as SOAP or XML-RPC

# What Is REST?
## Theory Of REST

- Focus on diversity of resources (nouns), not actions (verbs)
- Every resource is uniquely addressable
- All resources share the same constrained interface for transfer of state (actions) and content types
- Must be stateless, cacheable, and layered

# What Is REST?
## A Concise Definition

"[REST] is intended to evoke an image of how a well-designed Web application behaves: a network of web pages (a virtual state-machine), where the user progresses through an application by selecting links (state transitions), resulting in the next page (representing the next state of the application) being transferred to the user and rendered for their use."

— *Roy Felding*

# What Is REST?
## Web As Prime Example

- URIs uniquely address resources
- HTTP methods (GET, POST, PUT, DELETE) and content types (text/html, text/plain, etc.) provide a constrained interface
- All transactions are atomic
- HTTP provides cache control

# What Is REST?
## Well-RESTed

- Applications adhering to REST principles are said to be RESTful
- Extreme advocates of REST are often called RESTafarians

# What Is REST?
## Relaxing REST

- Any simple interface using XML over HTTP (in response to GET requests)
- That is also not RPC-based
- May use JSON, YAML, plain text, etc. instead of XML
- In many Web applications, this is what we mean when we say “REST”
- This is a very loose definition; RESTafarians prefer a stricter description

# Benefits Of REST:
## Clean & Well-designed URLs

- RESTafarians often suffer from URL vanity
- Well-designed URLs have a clear hierarchy
- They are hackable and can be reverse-engineered
- They have clear meaning and are not obfuscated
- They can be very long or very short, but must have meaning in either case

# Benefits Of REST:
## Semantic URLs

- The URLs have semantic meaning
- Information is logically architected
- It's easy for any user to find their way around by looking at the URL

# Benefits Of REST:
## Constrained Interface

- Reduces political battles among programmers
- No need to argue how the interface will work or what all the actions will be
- It's already been decided for you, and it's a standard that your team can agree upon
- You can focus on the resources rather than how to access/manipulate each resource

# Benefits Of REST:
## Easier for End-Users

- Constrained interface means no guess-work
- Semantic URLs means it's easy to find/manipulate information
- Use of an established standard content-type means that end-users do not need to learn a new data format
- Simplicity of design

# Verbs
## REST's Constrained Interface

| Methods |
| --- |
| GET |
| PUT |
| POST |
| DELETE |

→

| CRUD |
| --- |
| Read |
| Update |
| Create |
| Delete |

→

| Cut & Paste |
| --- |
| Copy |
| Paste Over |
| Paste After |
| Cut |

# Verbs
## GET

- Transfers (“copies”) a representation from resource to client
- Body must contain enough information for the client to usefully operate on the data
- According to RFC 2616:
  - GET is considered “safe”
  - Should not take any action other than retrieval
  - Has the property of “idempotence”

# Verbs
## GET: Request

```
GET /users/johnd HTTP/1.1
Host: example.org
```

# Verbs
## GET: Response Headers

```
HTTP/1.x 200 OK
Date: Tue, 22 May 2007 16:20:44 GMT
Server: Apache
Content-Length: 239
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/xml
```

# Verbs
## GET: Response Body (Entity)

```
<?xml version="1.0" encoding="UTF-8"?>
<users>
  <user id="652">
    <username>johnd</username>
    <firstname>John</firstname>
    <lastname>Doe</lastname>
    <birthday>1975-04-23</birthday>
    <email>johnd@example.org</email>
  </user>
</users>
```

# Verbs
## PUT

- The exact opposite of GET; transfers the state from client to a resource (equivalent to “paste over”)
- The body must contain enough information for the resource to operate on the data
- May also create a new resource at the requested URI
  - If created at time of request, send a 201 response (or 202 if creation deferred)
  - If updated, send a 200 response with the entity (or 204)
- Considered idempotent

# Verbs
## PUT: Request Headers

```
PUT /users/johnd HTTP/1.1
Host: example.org
Content-Type: text/xml
Content-Length: 273
```

# Verbs
## PUT: Request Body (Entity)

```
<?xml version="1.0" encoding="UTF-8"?>
<users>
  <user id="652">
    <username>johnd</username>
    <firstname>John</firstname>
    <middlename>Henry</middlename>
    <lastname>Doe</lastname>
    <birthday>1975-04-24</birthday>
    <email>johnd@example.com</email>
  </user>
</users>
```

# Verbs
## PUT: Response Headers

```
HTTP/1.x 204 No Content
Date: Tue, 22 May 2007 16:35:23 GMT
Server: Apache
```

# Verbs
## POST

- Has the same meaning as “paste after;” that is: “add to what you have; don’t overwrite it”
- If resource is created, return 201 with Location
- If resource exists, return 200 or 204
- POST a representation of the additional state to append to the resource or POST a representation of the entire resource to create a new resource

# Verbs
## POST: Request

```
POST /users HTTP/1.1
Host: example.org
Content-Length: #
...
POST /users/johnd HTTP/1.1
Host: example.org
Content-Length: #
...
```

# Verbs
## POST: Response

```
HTTP/1.x 201 Created
Date: Tue, 22 May 2007 16:43:56 GMT
Server: Apache
Location: /users/johnd
```

# Verbs
## DELETE

- Acts like “cut;” requests that the resource identified be destroyed or removed from public web
- Returns 200 if response includes a status entity
- Returns 202 if accepted but deferred
- Returns 204 if enacted but contains no entity
- DELETE is considered idempotent

# Verbs
## DELETE: Request & Response

```
DELETE /users/johnd HTTP/1.1
Host: example.org
```

```
HTTP/1.x 204 No Content
Date: Tue, 22 May 2007 16:53:15 GMT
Server: Apache
```

# Verbs
## Idempotence

- The side-effects of N > 0 identical requests is the same as for a single request
- That is: every time you make the request, as long as it is an identical request, exactly the same action occurs
- GET, HEAD, PUT and DELETE share this property
- POST is not considered idempotent

# Content Types

- Your application needs to deliver content in some sort of format that is readable by end-users
- Finding a standard content type “out in the wild” that works for your application will attract end-users
  - Ease of use
  - Low barrier to entry; low learning curve
  - Faster development for you and end-users
- Do not rule out creating your own schema if needed

# Content Types

- text/html
- text/plain
- application/calendar+xml
- application/atom+xml
- application/rdf+xml
- microformats

# RESTful Design

1. Determine your resources
2. Decide what methods each resource will support
3. Link the resources together
4. Develop your data schemas
5. Rationalize your schemas
6. Choose the best content type/format to represent your schemas

# RESTful Design
## 1. Determine your resources

/users

/users/username

/users/username/favorites

/users/username/tags

/content

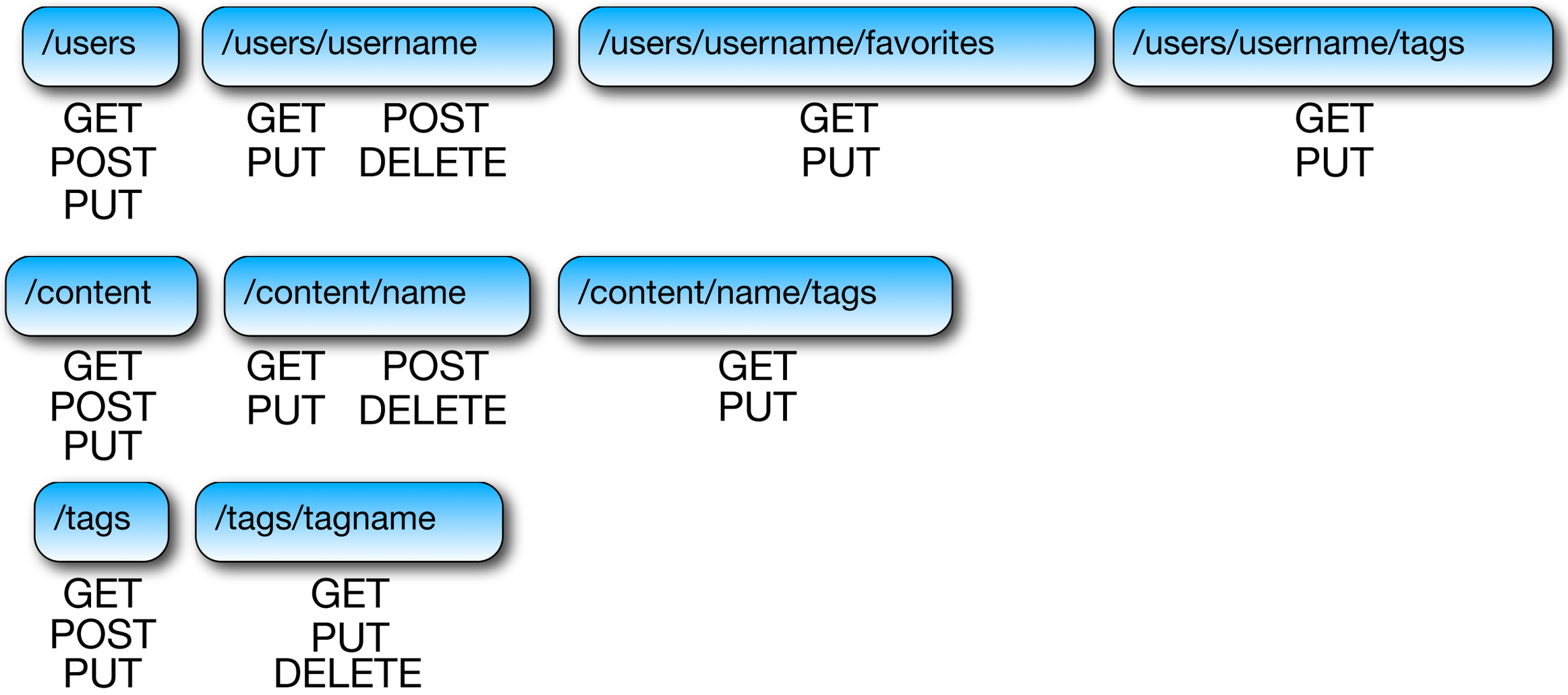
/content/name

/content/name/tags

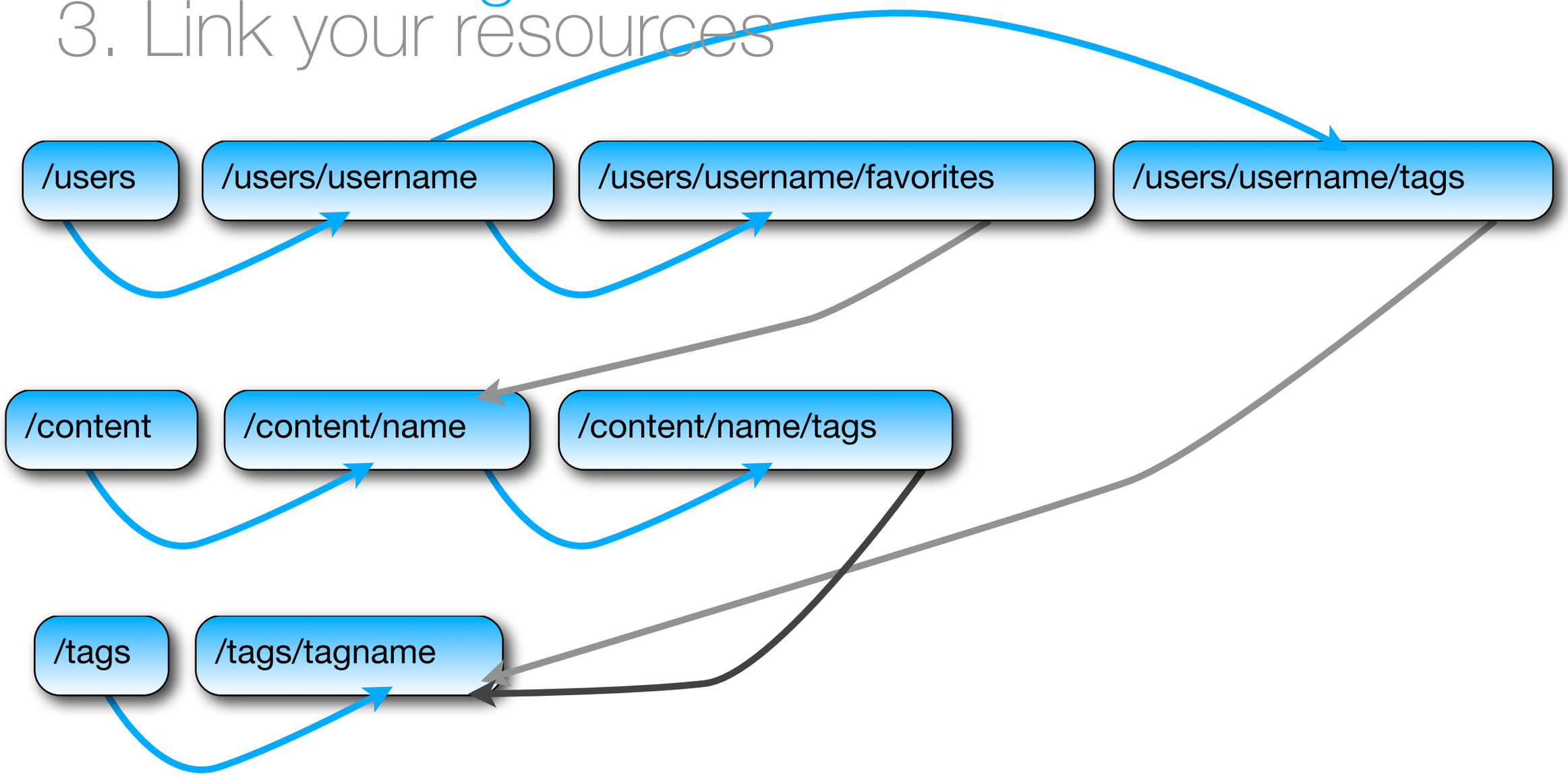
/tags

/tags/tagname

# RESTful Design
## 2. Decide the methods for each resource



# RESTful Design
## 3. Link your resources



# RESTful Design
## 4. Develop your data schemas

| /users |
| --- |
| id |
| username |
| firstname |
| lastname |

| /users/username |
| --- |
| id |
| username |
| firstname |
| lastname |

# RESTful Design
## 5. Rationalize your schemas

| /users |
| --- |
| user |

| /users/username |
| --- |
| id |
| username |
| firstname |
| lastname |

# RESTful Design
## 5. Rationalize your schemas

```
<?xml version="1.0"?>

<users>
  <user id="237">
    <username>johnd</username>
    <firstname>John</firstname>
    <lastname>Doe</lastname>
  </user>
</users>
```

# RESTful Design
## 6. Choose your content type/format

- Up to you
- Consider existing formats; do they fit?
- Consider your audience
- Consider using multiple formats (XML, JSON, HTML, etc.)
- Most popular are XML, JSON, and plain text

# Things To Consider:
## POST vs. PUT & DELETE

- They all serve distinctly different purposes
- POST is widely supported by default in Web servers
- To support PUT or DELETE, you must configure your Web server to handle them
- It's all about semantics: the meaning you wish to imply with the action you're taking/allowing
- Security concerns with PUT/DELETE are the same as with POST; ensure the user has permission to do it

# RESTful Web Services
## What Is A Web Service?

- Public interface (API)
- Provides access to data and/or procedures
- On a remote/external system (usually)
- Often uses XML for data exchange

# RESTful Web Services
## Why Use Web Services?

- Access to content/data stores you could not otherwise provide (zip codes, news, pictures, reviews, etc.)
- Enhance site with a service that is not feasible for you to provide (maps, search, products, etc.)
- Combine these services into a seamless service you provide (mash-ups)

# RESTful Web Services
## Why Provide A Web Service?

- You have a service that benefits your users best if they can get to their data from outside the application
- You want others to use your data store in their applications
- All the cool kids are doing it

# del.icio.us
## RESTful Web Services

- Public and authenticated REST access
- All requests over SSL using HTTP-Auth
- Requests a 1-second delay between queries
- Very simple API
- http://del.icio.us/help/api/

# delicious.php

```php
<?php

$un = 'your_username';
$pw = 'your_password';
$cache_file = '/tmp/blogroll.xml';

// check for updates to del.icio.us account
$update_req = "https://{$un}:{$pw}@api.del.icio.us/v1/posts/update";
$update = new SimpleXMLElement($update_req, NULL, TRUE);

if (strtotime($update['time']) > filemtime($cache_file))
{
    // del.icio.us has been updated since last cache; recache
    $uri = "https://{$un}:{$pw}@api.del.icio.us/v1/posts/all?tag=blogroll";
    $data = file_get_contents($uri);
    file_put_contents($cache_file, $data);
}

// read links from cached del.icio.us data
$blogroll = new SimpleXMLElement($cache_file, NULL, TRUE);

foreach ($blogroll->post as $blog)
{
    echo '<a href="' . htmlentities($blog['href']) . '">';
    echo htmlentities($blog['description'], ENT_COMPAT, 'UTF-8');
    echo "</a><br />\n";
}

?>
```

# Yahoo!
# RESTful Web Services

- Web Search Service is RESTful
- Requires an application ID, but no special authentication or handshake
- Limit 5,000 queries per IP address per day
- http://developer.yahoo.com/search/web/V1/webSearch.html

# yahoo.php

```php
<?php

$query = 'PHP';

$request  = 'http://search.yahooapis.com/';
$request .= 'WebSearchService/V1/webSearch?';
$request .= 'appid=ramsey&query=' . urlencode($query);

$ResultSet = new SimpleXMLElement($request, NULL, TRUE);

foreach ($ResultSet as $Result)
{
    echo '<p><a href="';
    echo htmlentities($Result->Url, ENT_COMPAT, 'UTF-8');
    echo '">';
    echo htmlentities($Result->Title, ENT_COMPAT, 'UTF-8');
    echo '</a><br/>';
    echo htmlentities($Result->Summary, ENT_COMPAT, 'UTF-8');
    echo '</p>';
}

?>
```

# Flickr
## RESTful Web Services

- Provides a variety of Web Service interfaces, including REST
- Accomplished in an RPC fashion
- Uses a complex token authentication handshake to access user data
- http://flickr.com/services/api/

# login.php

```php
<?php

require_once 'conf.php';

$flickr = $_SESSION['flickr'];

// Create API Signature for authentication
$api_sig  = $flickr['secret'];
$api_sig .= 'api_key' . $flickr['api_key'];
$api_sig .= 'perms' . $flickr['perms'];
$api_sig = md5($api_sig);

// Create link for Flickr authentication
$link  = 'http://flickr.com/services/auth';
$link .= '?api_key=' . urlencode($flickr['api_key']);
$link .= '&perms=' . urlencode($flickr['perms']);
$link .= '&api_sig=' . urlencode($api_sig);

?>

<a href="<?php echo $link; ?>">Authenticate With Flickr</a>
```

# flickr.php

```php
<?php

session_start();

// Signature elements to reuse
$sig  = $_SESSION['flickr']['secret'];
$sig .= 'api_key' . $_SESSION['flickr']['api_key'];

// Get frob and convert it to a token for this user
$tok_sig  = $sig;
$tok_sig .= 'frob' . $_GET['frob'];
$tok_sig .= 'methodflickr.auth.getToken';
$tok_sig = md5($tok_sig);

// Create a token request URI
$tok_req  = 'http://api.flickr.com/services/rest/';
$tok_req .= '?api_key=';
$tok_req .= urlencode($_SESSION['flickr']['api_key']);
$tok_req .= '&frob=' . urlencode($_GET['frob']);
$tok_req .= '&api_sig=' . urlencode($tok_sig);
$tok_req .= '&method=flickr.auth.getToken';
```

# flickr.php

```
// Get a token
$tok_rsp = new SimpleXMLElement($tok_req, NULL, TRUE);
if ($tok_rsp->auth)
{
    $token = $tok_rsp->auth->token;
}
else
{
    echo htmlentities($tok_rsp->err['msg'],
                      ENT_COMPAT, 'UTF-8');
    exit;
}
```

# flickr.php

```php
// Create authorization signature
$auth_sig  = $sig;
$auth_sig .= 'auth_token' . $token;
$auth_sig .= 'methodflickr.contacts.getList';
$auth_sig = md5($auth_sig);

// Create a contacts request URI
$contact_req  = 'http://api.flickr.com/services/rest/';
$contact_req .= '?api_key=';
$contact_req .= urlencode($_SESSION['flickr']['api_key']);
$contact_req .= '&auth_token=' . urlencode($token);
$contact_req .= '&api_sig=' . urlencode($auth_sig);
$contact_req .= '&method=flickr.contacts.getList';
```

# flickr.php

```php
// Get list of contacts
$contact_rsp = new SimpleXMLElement($contact_req, NULL, TRUE);

foreach ($contact_rsp->contacts->contact as $contact)
{
    if (strlen(trim($contact['realname'])) > 0)
    {
        $name = $contact['realname'];
    }
    else
    {
        $name = $contact['username'];
    }

    echo htmlentities($name, ENT_COMPAT, 'UTF-8') . "<br />\n";
}

?>
```

# Tools for Creating RESTful Web Services

- Zend Framework includes:
  - Zend_Rest_Client
  - Zend_Rest_Server
  - http://framework.zend.com/manual/en/zend.rest.html
- Tonic: “A RESTful Web App Development Framework”
  - http://tonic.sourceforge.net/

# Resources
## For More Information

- http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm
- http://www.w3.org/Protocols/rfc2616/rfc2616.html
- http://rest.blueoxen.net/cgi-bin/wiki.pl
- http://www.welldesignedurls.org/
- 
- Slides: http://benramsey.com/archives/phpworks07-slides/
- My company: http://www.schematic.com/