

Effective RESTful services

Do's and Don'ts for building great RESTful services

#1: Do NOT put the verb in URL

A.k.a. the Flickr rule

Bad, bad, bad!

`http://flickr.com/
?method=flickr.photos.delete&photo_id=123`



Use the Uniform Interface

Uniform

Get(URI)

Put(URI, Resource)

Delete(URI)

Non Uniform

getCustomer()

updateCustomer(Customer)

delete(customerId);



HTTP's Uniform Interface

GET

- Cacheable
- SAFE – no side effects

POST

- Unsafe operations, which can't be repeated

PUT

- Idempotent

DELETE

- Idempotent

HEAD

- SAFE – no side effects
 - No message body
-



Resources, resources, resources

- ▶ **Everything** is a resource
- ▶ Resources are addressable via **URIs**
- ▶ Resources are **self descriptive**
 - ▶ Typically through content types (“application/xml”) and sometimes the resource body (i.e. an XML QName)
- ▶ Resources are ***stateless***
- ▶ Resources are manipulated via **verbs** and the **uniform interface**



Why you should follow uniform interface semantics

- ▶ Cacheability
- ▶ Reliability semantics



Idempotent Operations

Same
Request

yields

Same
Result



Some Basic Scenarios:

1. Getting resources
2. Deleting resources
3. Updating a resource
4. Creating a resource

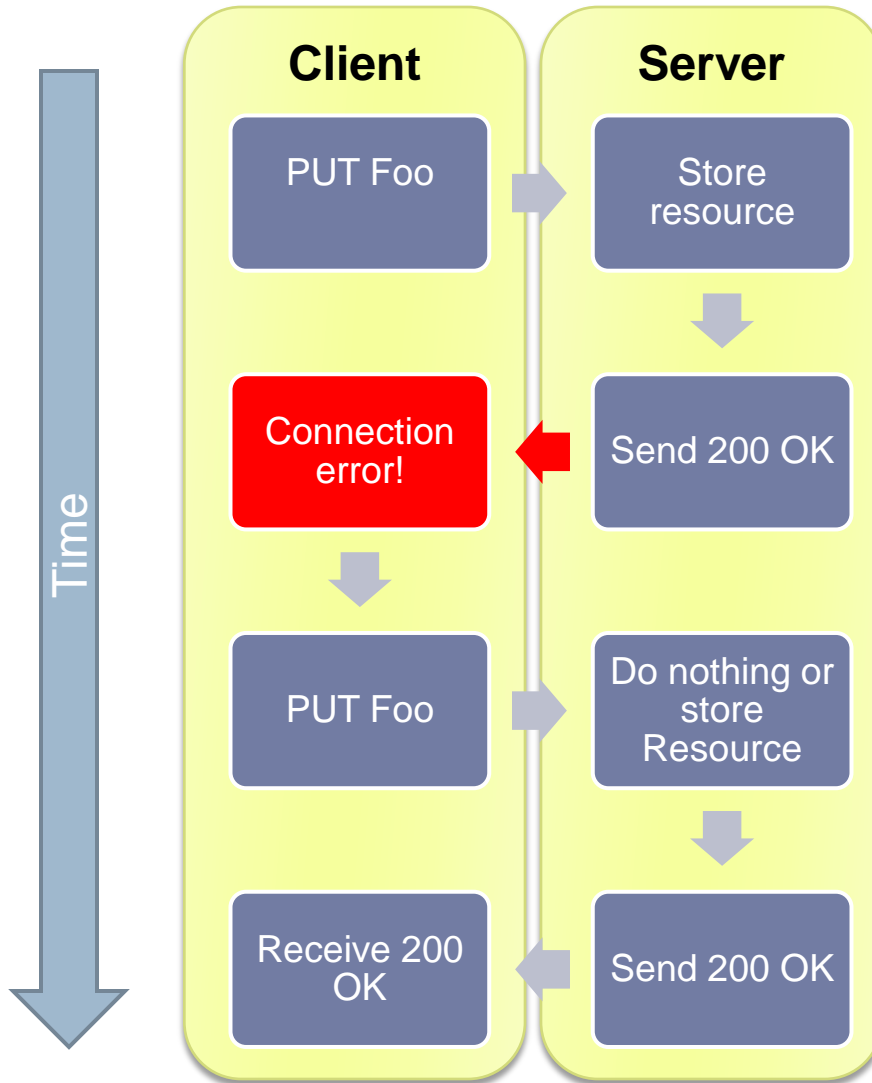


Getting a resource

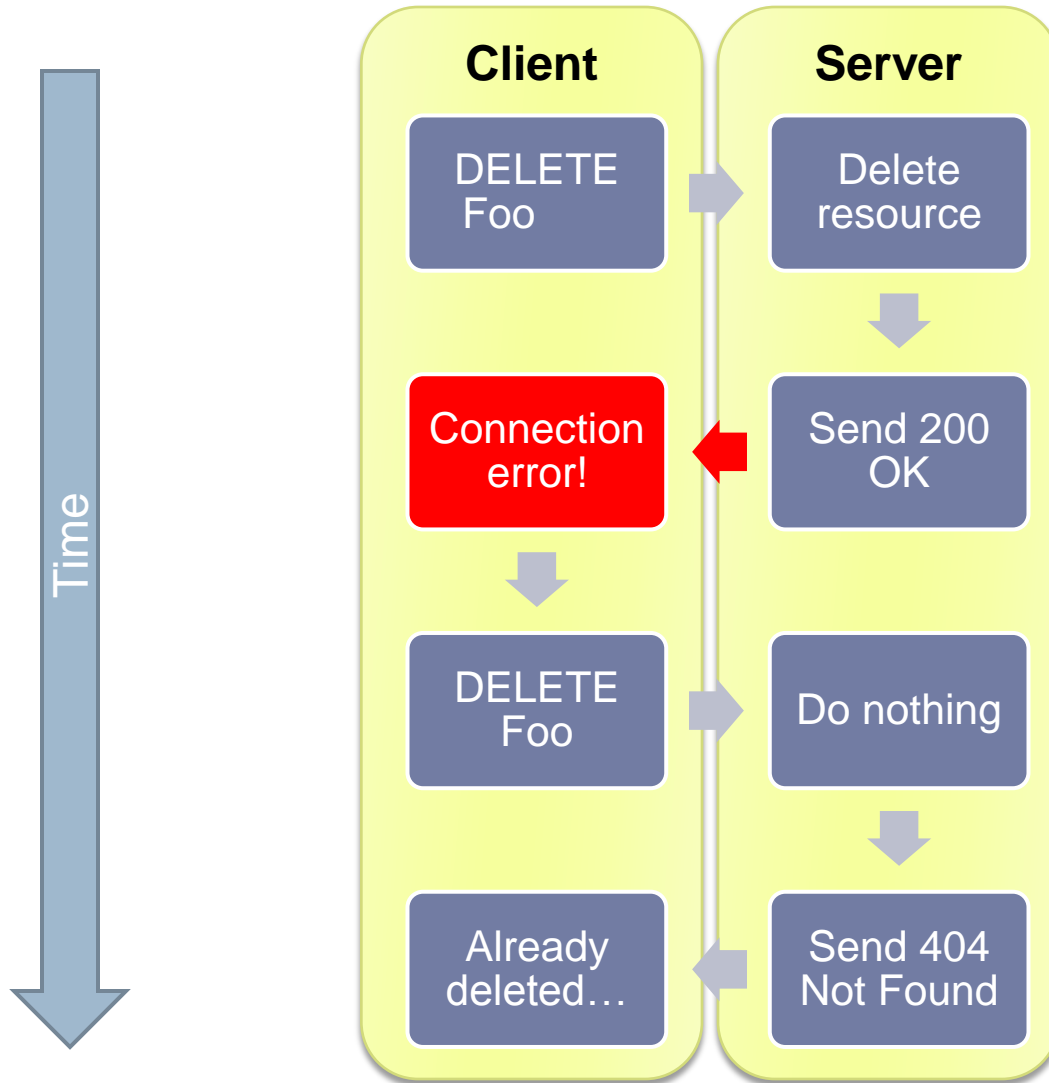
- ▶ GET is SAFE
- ▶ If original GET fails, just try, try again



Updating a resource



Deleting a resource



Creating Resources

```
POST /entries
Host: acme.com
...
```

Client



```
HTTP/1.1 201 Created
Date: ...
Content-Length: 0
Location:
  http://acme.com/entries/1
...
```

Server

```
PUT /entries/1
Host: acme.com
Content-Type: ...
Content-Length: ...
```

Some data...

Client



```
HTTP/1.1 200 OK
...
```

Server



Creating Resources

- ▶ IDs which are not used can be
 - ▶ Ignored
 - ▶ Expired
- ▶ Another option: have the client generate a unique ID and PUT to it straight away
 - ▶ They're liable to screw it up though



Problem: Firewalls

- ▶ Many firewalls do not allow PUT, DELETE
- ▶ You might want to allow other ways of specifying a header:
 - ▶ Google: `X-HTTP-Method-Override: PUT`
 - ▶ Ruby: `?method=PUT`



#2: Do use links

Bad example

```
<purchase-order>
```

```
...
```

```
<lineItem id="123">
```

```
<lineItem id="456">
```

```
</purchase-order>
```



Hypertext and linkability

- ▶ We don't want "keys", we want links!
- ▶ Resources are hypertext
 - ▶ Hypertext is just data with links to other resources
- ▶ Data model refers to other application states via links
- ▶ This is possible because of the uniform interface! No need to know different ways to get different types of entities!
- ▶ Client can simply navigate to different resources



Good example

```
<purchase-order>
```

```
...
```

```
<lineItem id="http://host/lineItems/123">
```

```
<lineItem id="http://host/lineItems/456">
```

```
</purchase-order>
```



Hypertext enables loose coupling

- ▶ No need on the client side to construct links
- ▶ No dependency on a particular link structure, client can simply navigate
- ▶ Server can change links to
 - ▶ Balance load
 - ▶ Deal with changes/relocations



#3: Do reuse dataformats

So many representations...

- ▶ XML
- ▶ JSON
- ▶ XHTML
- ▶ CSV
- ▶ Atom
- ▶ How do you decide which one to use?



The Dataformat Guideline

Use the most *universal* data format possible
(unless performance concerns or awkwardness
dictate otherwise).



XHTML and Microformats

```
<div class="contact">  
  <span class="name">Dan Diephouse</span>  
  <span class="phone">+1 555 555 5555</span>  
  <span class="city">Grand Rapids</span>  
</div>
```



The XHTML Advantage

- ▶ You get both presentation and data
- ▶ Easily consumable
- ▶ Many predefined microformats out there



JSON

```
{  
  "name" : "Dan Diephouse",  
  "phone" : "+1-555-555-5555",  
  "city", : "Grand Rapids"  
}
```



#4: Do use Etag/LastModified
Headers

ETag Header

- ▶ Resources may return an ETag header when it is accessed
- ▶ On subsequent retrieval of the resource, Client sends this ETag header back
- ▶ If the resource has not changed (i.e. the ETag is the same), an empty response with a 304 code is returned
- ▶ Reduces bandwidth/latency



ETag Example

```
GET /feed.atom
Host: www.acme.com
...
```

Client



```
HTTP/1.1 200 OK
Date: ...
ETag: "3e86-410-3596fbbc"
Content-Length: 1040
Content-Type: text/html
...
```

Server

```
GET /feed.atom
If-None-Match:
  "3e86-410-3596fbbc"
Host: www.acme.com
...
```

Client



```
HTTP/1.1 304 Not Modified
Date: ...
ETag: "3e86-410-3596fbbc"
Content-Length: 0...
```

Server



LastModified Example

```
GET /feed.atom
Host: www.acme.com
...
```

```
GET /feed.atom
If-Modified-Since:
  Sat, 29 Oct 1994
  19:43:31 GMT
Host: www.acme.com
...
```

Client

```
HTTP/1.1 200 OK
Date: ...
Last-Modified: Sat, 29 Oct
  1994 19:43:31 GMT
Content-Length: 1040
Content-Type: text/html
...
```

```
HTTP/1.1 304 Not Modified
Date: ...
Last-Modified: Sat, 29 Oct
  1994 19:43:31 GMT
Content-Length: 0
```

Server

Bonus Question

When do you use LastModified and when do you use Etags?



#5: Do take advantage of caching

Scalability through Caching

- ▶ GET + ETags/LastModified header enable efficient caching
- ▶ Reduce latency, network traffic, and server load
- ▶ Types of cache:
 - ▶ Browser
 - ▶ Proxy
 - ▶ Gateway



How Caching Works

- ▶ A resource is eligible for caching if:
 - ▶ The HTTP response headers don't say not to cache it
 - ▶ The response is not authenticated or secure
 - ▶ No ETag or LastModified header is present
 - ▶ The cache representation is fresh
- ▶ From: http://www.mnot.net/cache_docs/



Is your cache fresh?

- ▶ Yes, if:
 - ▶ The expiry time has not been exceeded
 - ▶ The representation was LastModified a relatively long time ago
- ▶ If its stale, the remote server will be asked to *validate* if the representation is still fresh



#6: Do NOT create stateful applications

Stateless client/server approach

- ▶ All communication is stateless
- ▶ Session state is kept on the Client!
 - ▶ Client is responsible for transitioning to new states
 - ▶ States are represented by URIs



Stateless Benefits

- ▶ **Improves:**
 - ▶ Visibility
 - ▶ Reliability
 - ▶ Scalability



#7: Do use HTTP status codes
correctly

My Favorite Status Codes

- ▶ 100
- ▶ 200
- ▶ 201
- ▶ 204
- ▶ 301
- ▶ 400
- ▶ 401
- ▶ 403
- ▶ 404
- ▶ 405
- ▶ 409
- ▶ 500



#8: Do NOT use REST/HTTP for
everything

When should you not use REST/HTTP?

- ▶ Transactions?
- ▶ Batch?
- ▶ Security scenarios?



When should you not use REST/HTTP?

▶ Transactions

- ▶ Applicable sometimes: see SVN
- ▶ Be wary of locking resources at such a coarse grained level

▶ Batch

- ▶ What response code do you return? How do you return one for each individual resource you're creating/updating?

▶ Security scenarios

- ▶ Gateways may remove SSL info



Open Discussion

Batch? Security? WADL? OpenID/Auth?

Questions?

- ▶ Blog: <http://netzoid.com/blog>
- ▶ Email: dan.diephouse@mulesource.com
- ▶ Resources:
 - ▶ RFC2616: <http://www.faqs.org/rfcs/rfc2616.html>
 - ▶ RESTful Web Services (Richardson, Ruby, DHH)

