



3 RESTful Service Design

Cesare Pautasso
Faculty of Informatics
University of Lugano, Switzerland

c.pautasso@ieee.org
<http://www.pautasso.info>

REST Design Constraints

1. Resource Identification URI
2. Uniform Interface
GET, PUT, DELETE, POST
(HEAD, OPTIONS...)
3. Self-Describing Messages
4. Hypermedia Driving Application State
5. Stateless Interactions

REST Design Constraints

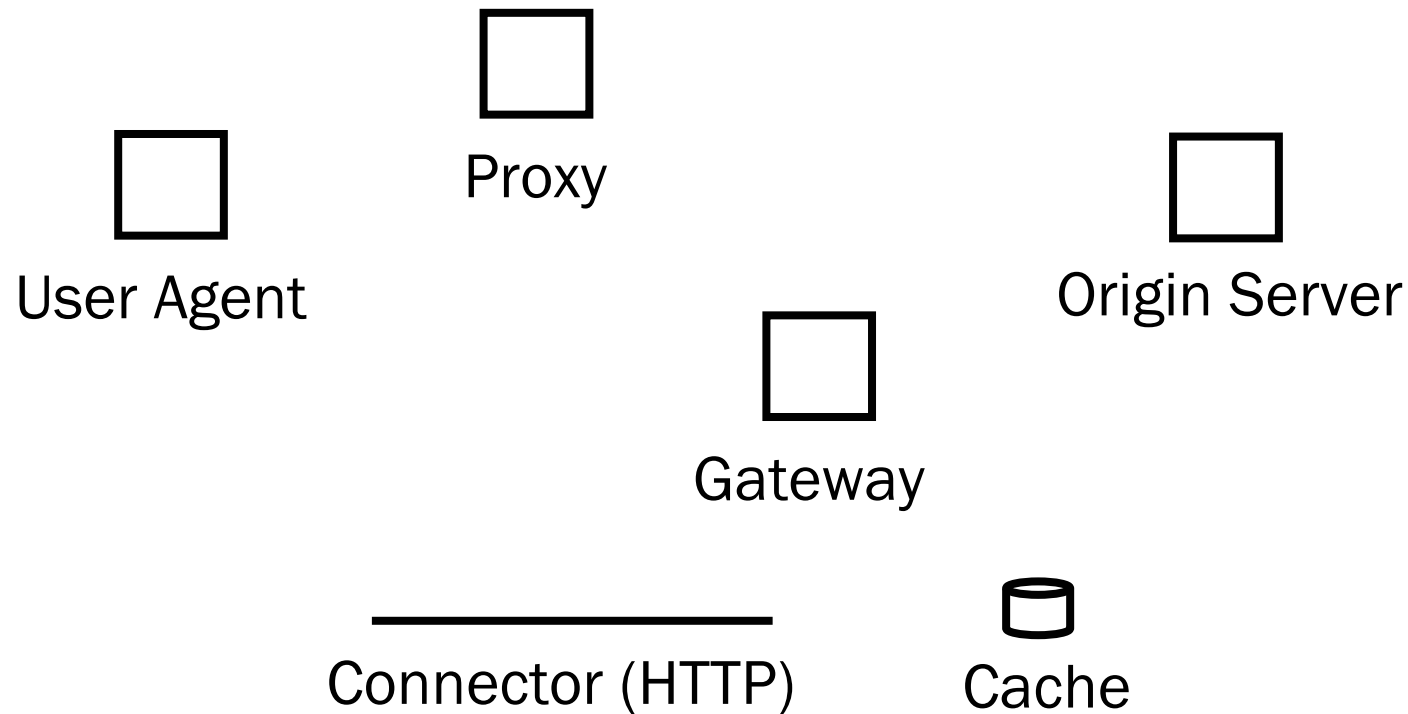
1. Resource Identification
2. Uniform Interface
GET, PUT, DELETE, POST
(HEAD, OPTIONS...)
3. Self-Describing Messages
4. Hypermedia Driving Application State
5. Stateless Interactions

REST Design - Outline

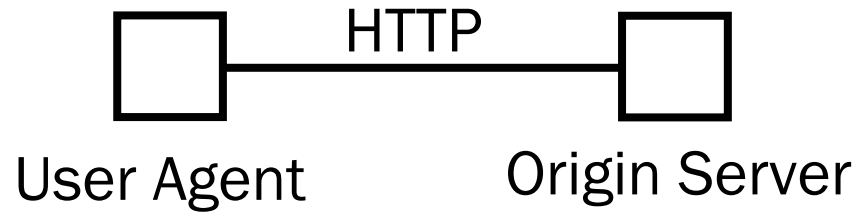
- REST Architectural Elements
- Design Methodology
 - Is URI Design part of REST?
- Simple Doodle Service Example
- Design Tips
 - Understanding GET vs. POST vs. PUT
 - Multiple Representations
 - Content-Type Negotiation
 - Exception Handling
 - Idempotent vs. Unsafe
 - Dealing with Concurrency
 - Stateful or Stateless?
- Some REST AntiPatterns

REST Architectural Elements

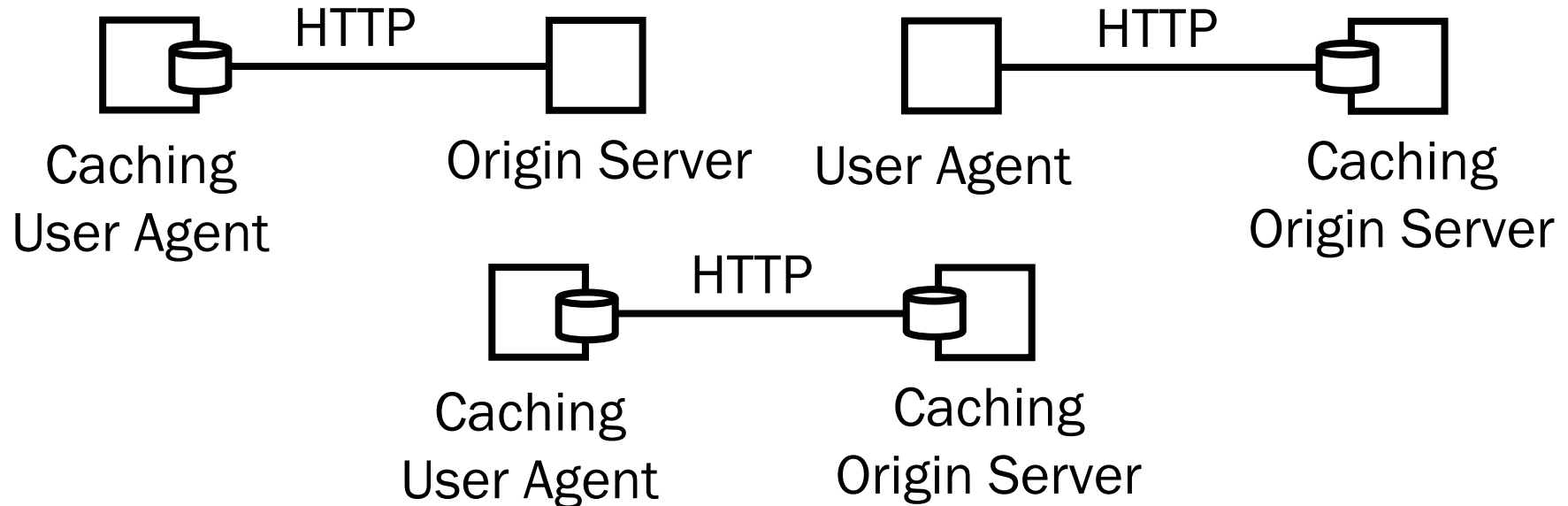
Client/Server Layered Stateless Communication Cache



Basic Setup

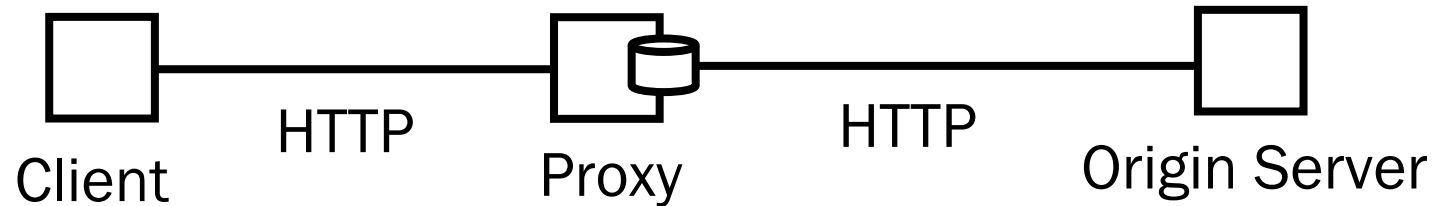


Adding Caching

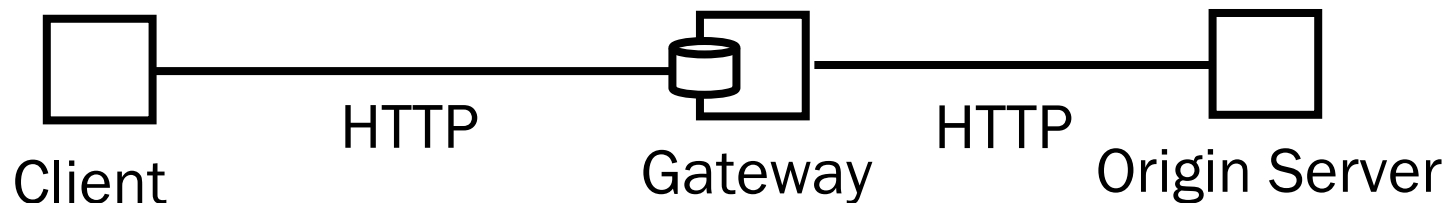


Proxy or Gateway?

Intermediaries forward (and may translate) requests and responses



A proxy is chosen by the Client (for caching, or access control)

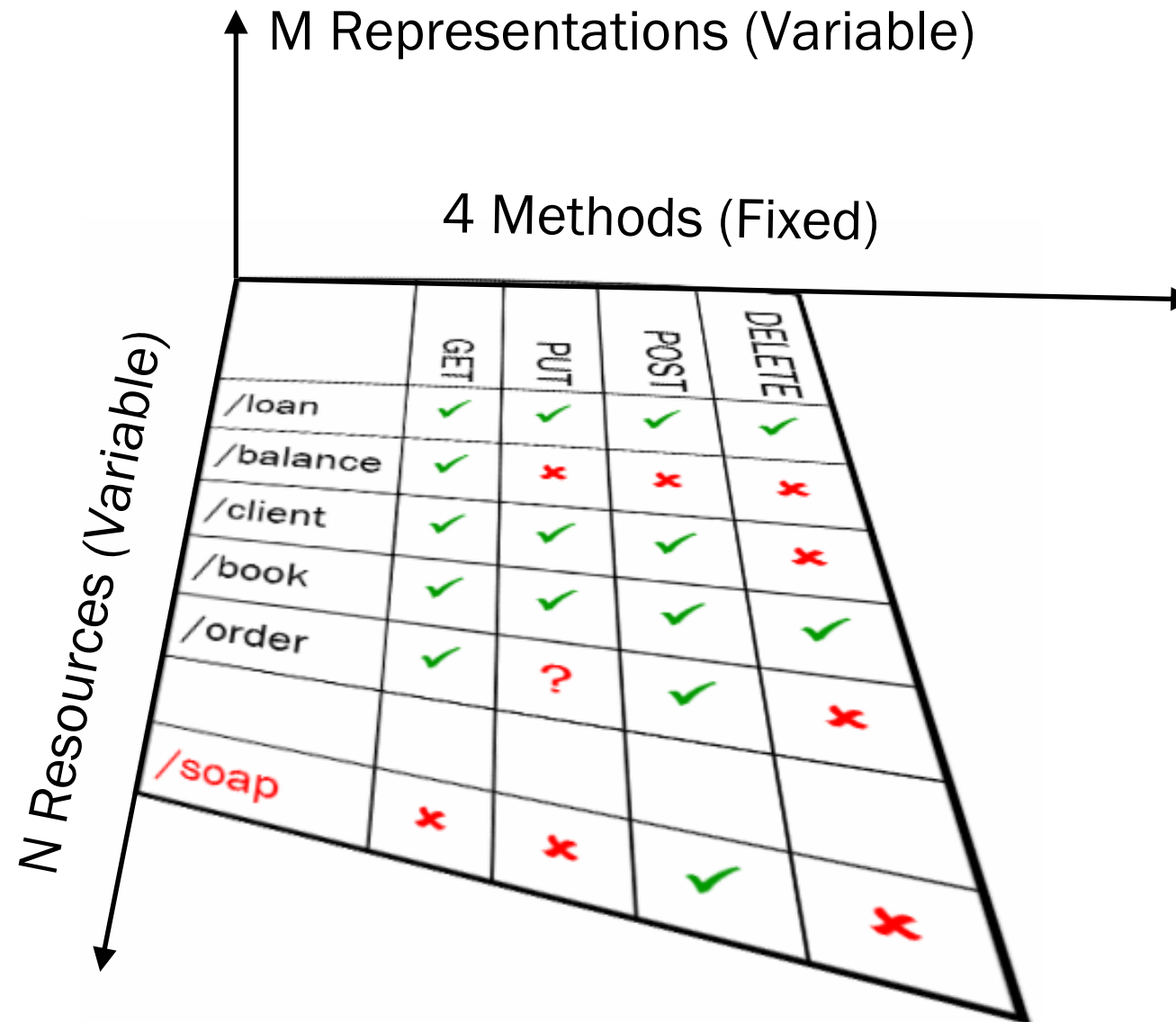


The use of a gateway (or reverse proxy) is imposed by the server

1. Identify resources to be exposed as services (e.g., yearly risk report, book catalog, purchase order, open bugs, polls and votes)
2. Model relationships (e.g., containment, reference, state transitions) between resources with hyperlinks that can be followed to get more details (or perform state transitions)
3. Define “nice” URIs to address the resources
4. Understand what it means to do a GET, POST, PUT, DELETE for each resource (and whether it is allowed or not)
5. Design and document resource representations
6. Implement and deploy on Web server
7. Test with a Web browser

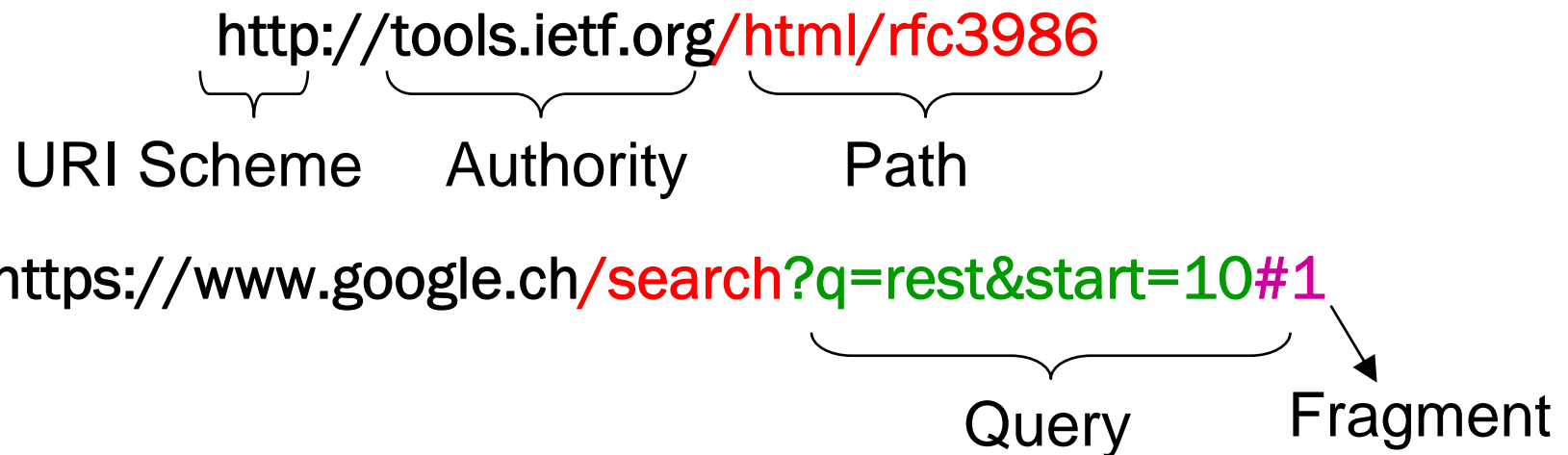
	GET	PUT	POST	DELETE
/loan	✓	✓	✓	✓
/balance	✓	✗	✗	✗
/client	✓	✓	✓	✗
/book	✓	✓	✓	✓
/order	✓	?	✓	✗
/soap	✗	✗	✓	✗

Design Space



URI - Uniform Resource Identifier

- Internet Standard for resource naming and identification (originally from 1994, revised until 2005)
- Examples:

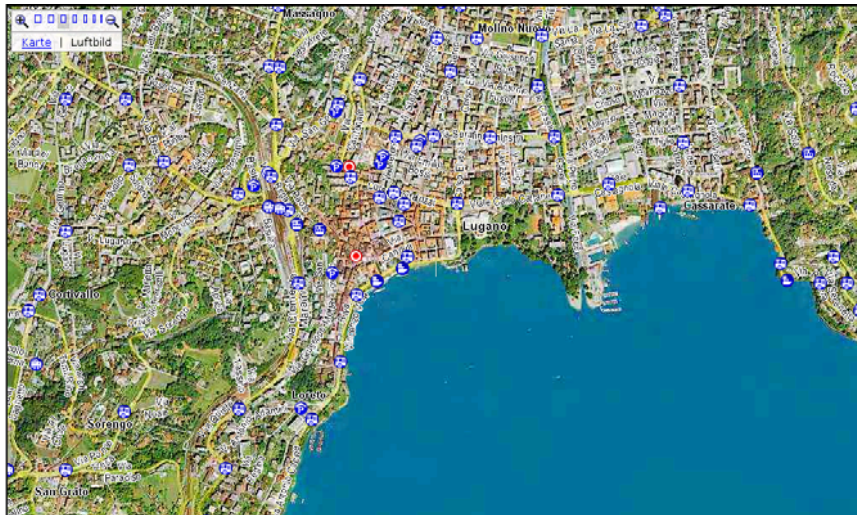


- REST does not advocate the use of “nice” URIs
- In most HTTP stacks URIs cannot have arbitrary length (4Kb)

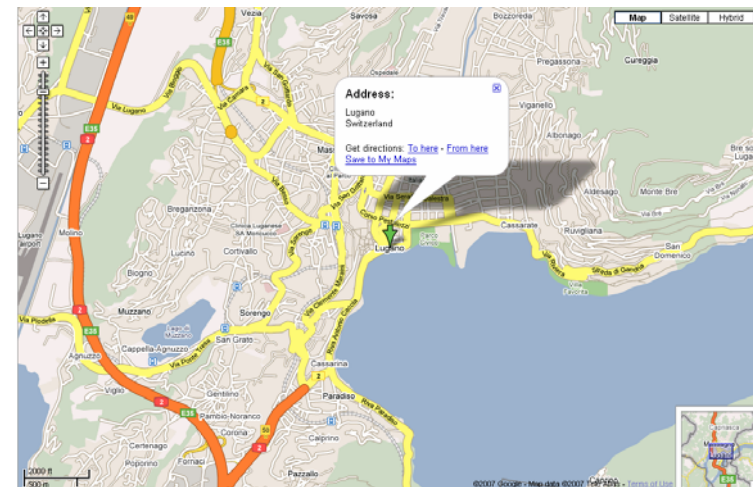
What is a “nice” URI?

A RESTful service is much more than just a set of nice URIs

<http://map.search.ch/lugano>



<http://maps.google.com/lugano>



<http://maps.google.com/maps?f=q&hl=en&q=lugano,+switzerland&layer=&ie=UTF8&z=12&om=1&iwloc=addr>

URI Design Guidelines

- Prefer Nouns to Verbs
- Keep your URIs short
- Follow a “positional” parameter-passing scheme (instead of the key=value&p=v encoding)
- URI postfixes can be used to specify the content type
- Do not change URIs
- Use redirection if you really need to change them

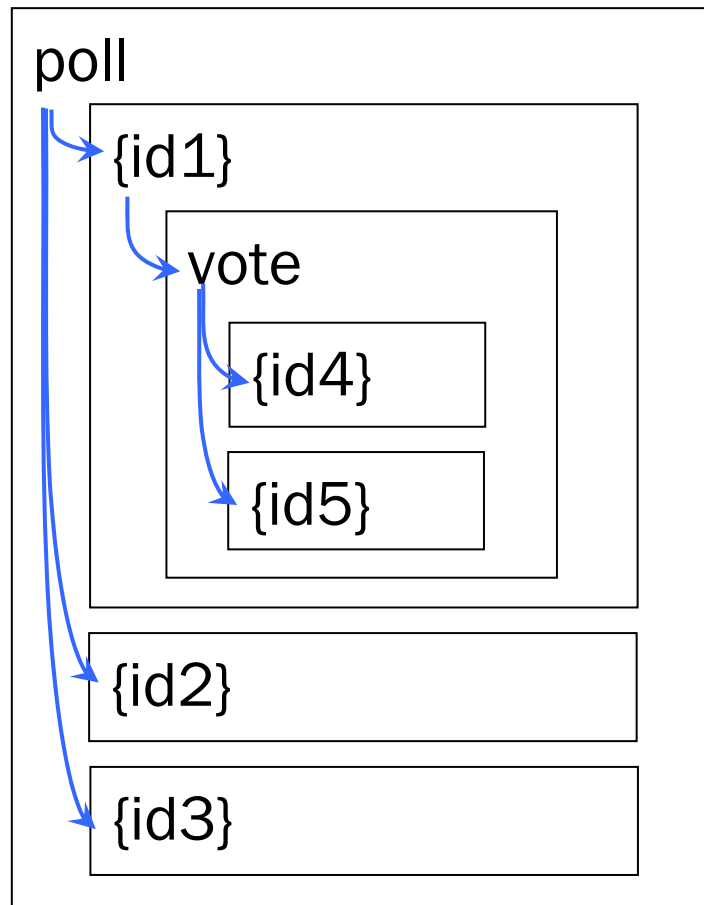
GET /book?isbn=24&action=delete
DELETE /book/24

- **Note:** REST URIs are opaque identifiers that are meant to be discovered by following hyperlinks and *not constructed by the client*

Warning: URI Templates introduce coupling between client and server

Simple Doodle API Example Design

1. Resources:
polls and votes
2. Containment Relationship:

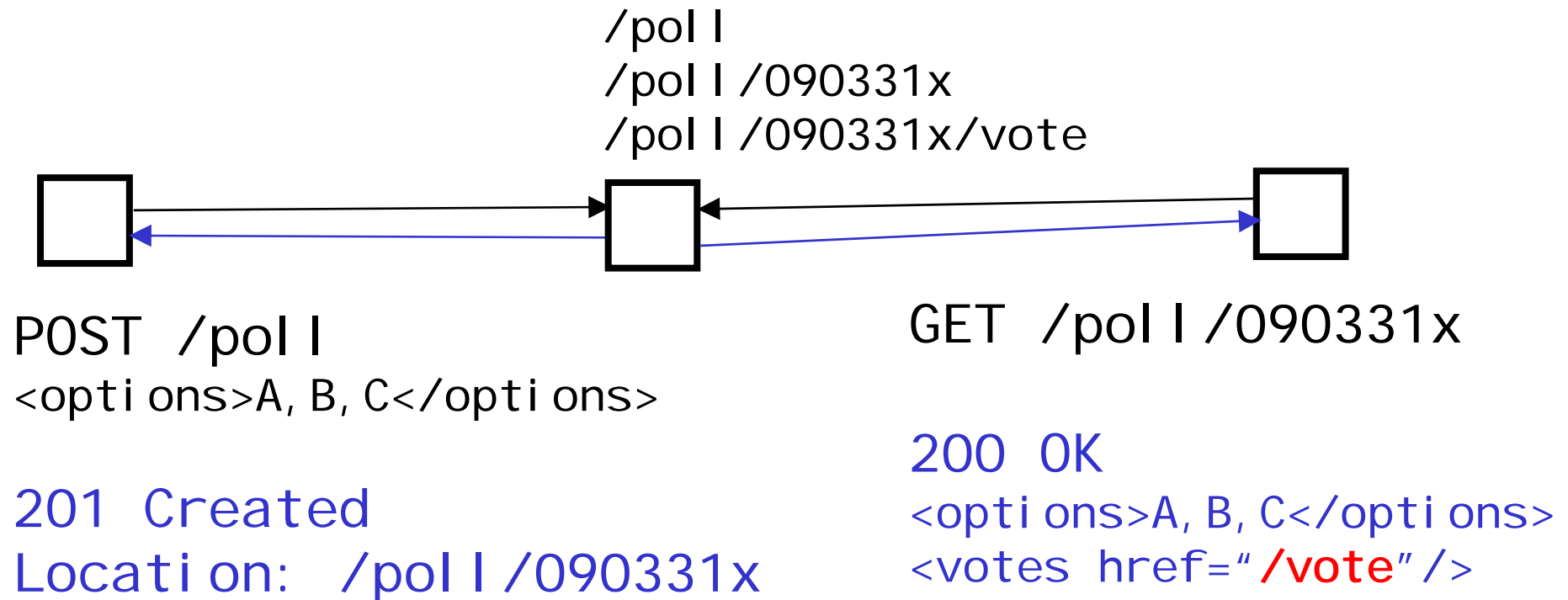


	GET	PUT	POST	DELETE
/poll	✓	✗	✓	✗
/poll/{id}	✓	✓	✗	✓
/poll/{id}/vote	✓	✗	✓	✗
/poll/{id}/vote/{id}	✓	✓	✗	?

3. URIs embed IDs of “child” instance resources
4. POST on the container is used to create child resources
5. PUT/DELETE for updating and removing child resources

Simple Doodle API Example

1. Creating a poll
(transfer the state of a new poll on the Doodle service)

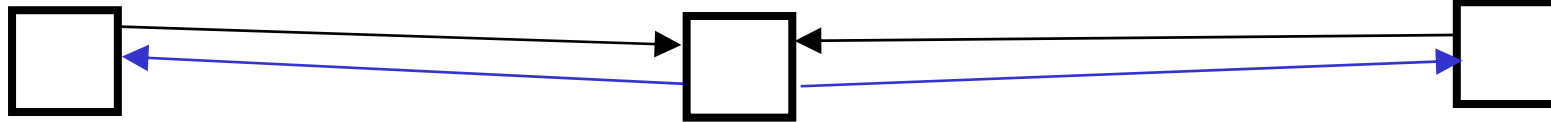


2. Reading a poll
(transfer the state of the poll from the Doodle service)

Simple Doodle API Example

- Participating in a poll by creating a new vote sub-resource

/poll
/poll /090331x
/poll /090331x/vote
/poll /090331x/vote/1



POST /poll /090331x/vote
<name>C. Pautasso</name>
<choice>B</choice>

201 Created

Location:

/poll /090331x/vote/1

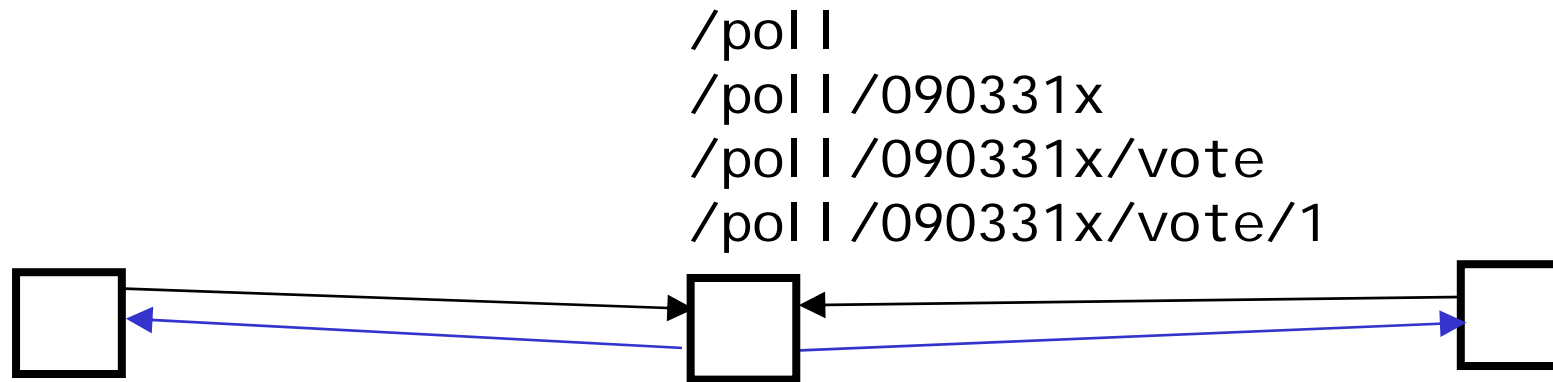
GET /poll /090331x

200 OK

<options>A, B, C</options>
<votes><vote id="1">
<name>C. Pautasso</name>
<choice>B</choice>
</vote></votes>

Simple Doodle API Example

- Existing votes can be updated (access control headers not shown)



PUT /pol I /090331x/vote/1
<name>C. Pautasso</name>
<choi ce>C</choi ce>

200 OK

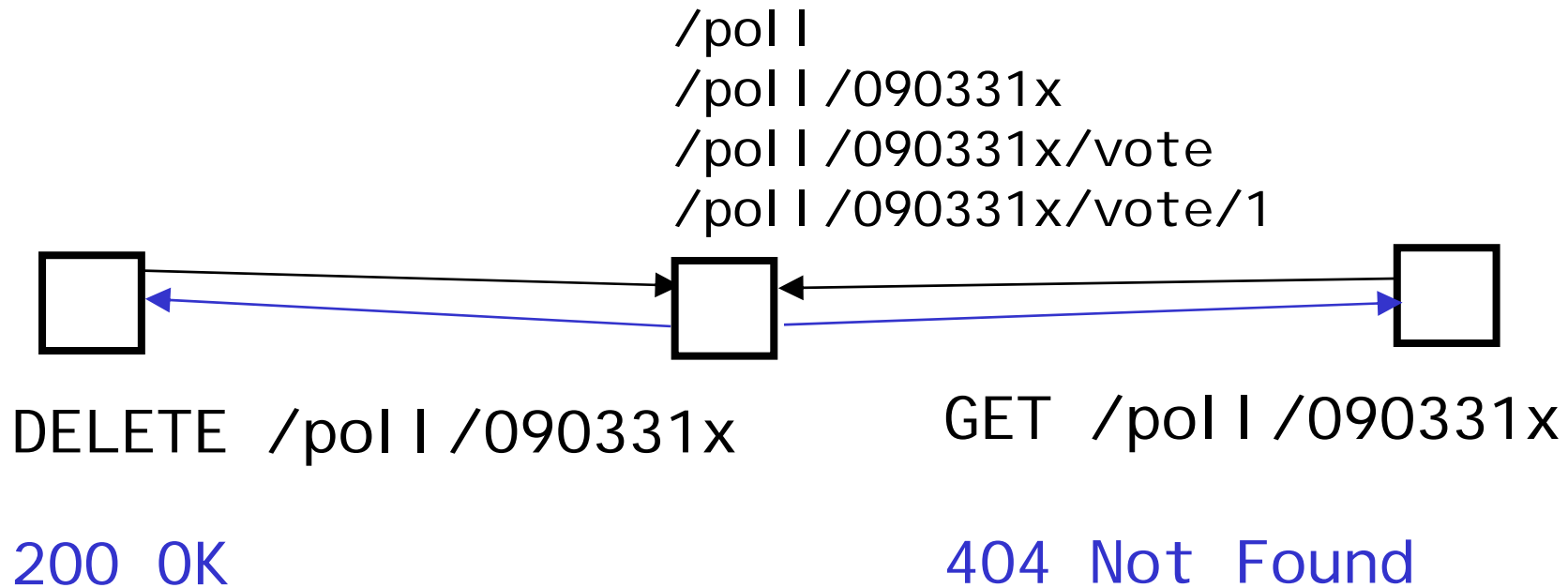
GET /pol I /090331x

200 OK

<opti ons>A, B, C</opti ons>
<votes><vote id="/1">
<name>C. Pautasso</name>
<choi ce>C</choi ce>
</vote></votes>

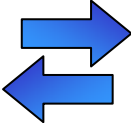
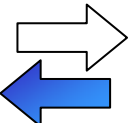
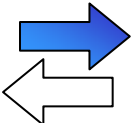
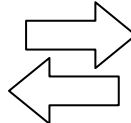
Simple Doodle API Example

- Polls can be deleted once a decision has been made



More info on the real Doodle API: <http://doodle.com/xsd1/RESTfulDoodle.pdf>

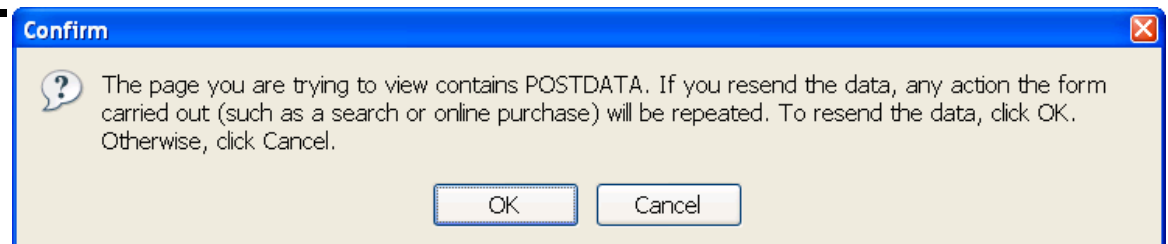
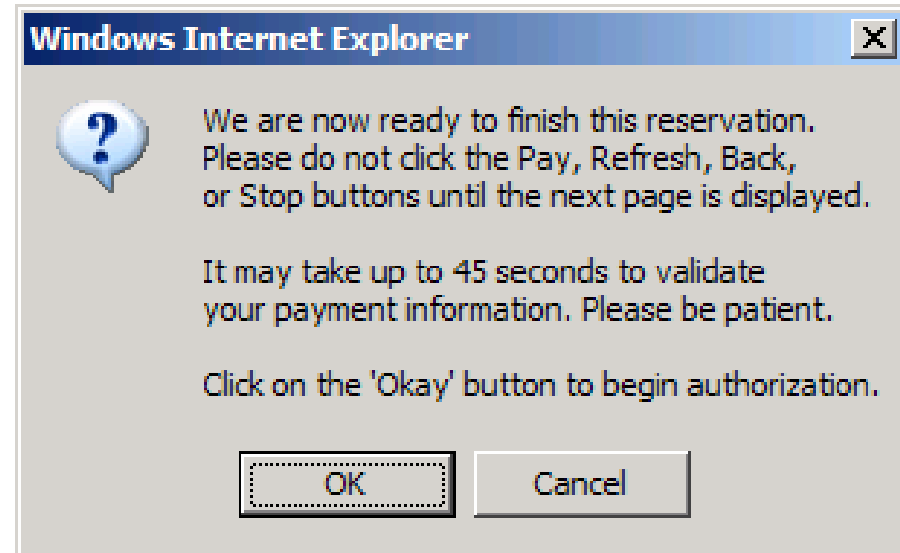
Uniform Interface Principle

CRUD	REST	
CREATE	POST 	Create a sub resource
READ	GET 	Retrieve the current state of the resource
UPDATE	PUT 	Initialize or update the state of a resource at the given URI
DELETE	DELETE 	Clear a resource, after the URI is no longer valid

POST vs. GET

- GET is a **read-only** operation. It can be repeated without affecting the state of the resource (idempotent) and can be cached
- POST is a **read-write** operation and may change the state of the resource and provoke side effects on the server.

Web browsers warn you when refreshing a page generated with POST



POST vs. PUT

What is the right way of creating resources (initialize their state)?

→ PUT /resource/{id}

← 201 Created

Problem: How to ensure resource {id} is unique?
(Resources can be created by multiple clients concurrently)

Solution 1: let the client choose a unique id (e.g., GUID)

→ POST /resource

← 301 Moved Permanently

Location: /resource/{id}

Solution 2: let the server compute the unique id

Problem: Duplicate instances may be created if requests are repeated due to unreliable communication

Content Negotiation (Conneg)

Negotiating the message format does not require to send more messages (the added flexibility comes for free)

⇒ GET /resource

Accept: text/html , application/xml ,
application/json

1. The client lists the set of understood formats (MIME types)

← 200 OK

Content-Type: application/json

2. The server chooses the most appropriate one for the reply

Forced Content Negotiation

The generic URI supports content negotiation

GET /resource

Accept: text/html , application/xml ,
application/json

The specific URI points to a specific representation format using the postfix (extension)

GET /resource.html

GET /resource.xml

GET /resource.json

Warning: This is a conventional practice, not a standard.

What happens if the resource cannot be represented in the requested format?

Exception Handling

Learn to use HTTP Standard Status Codes

100 Continue
200 OK
201 Created
202 Accepted
203 Non-Authentic
204 No Content
205 Reset Content
206 Partial Content
300 Multiple Choices
301 Moved Permanently
302 Found
303 See Other
304 Not Modified
305 Use Proxy
307 Temporary Redirect

4xx Client's fault

400 Bad Request
401 Unauthorized
402 Payment Required
403 Forbidden
404 Not Found
405 Method Not Allowed
406 Not Acceptable
407 Proxy Authentication Required
408 Request Timeout
409 Conflict
410 Gone
411 Length Required
412 Precondition Failed
413 Request Entity Too Large
414 Request-URI Too Long
415 Unsupported Media Type
416 Requested Range Not Satisfiable
417 Expectation Failed

500 Internal Server Error
501 Not Implemented
502 Bad Gateway
503 Service Unavailable
504 Gateway Timeout
505 HTTP Version Not Supported

5xx Server's fault

Idempotent vs. Unsafe

- Idempotent requests can be processed multiple times without side-effects

GET /book

PUT /order/x

DELETE /order/y

- If something goes wrong (server down, server internal error), the request can be simply replayed until the server is back up again
- Safe requests are idempotent requests which do not modify the state of the server

GET /book

- Unsafe requests modify the state of the server and cannot be repeated without additional (unwanted) effects:

Withdraw(200\$) //unsafe

Deposit(200\$) //unsafe

- Unsafe requests require special handling in case of exceptional situations (e.g., state reconciliation)

POST /order/x/payment

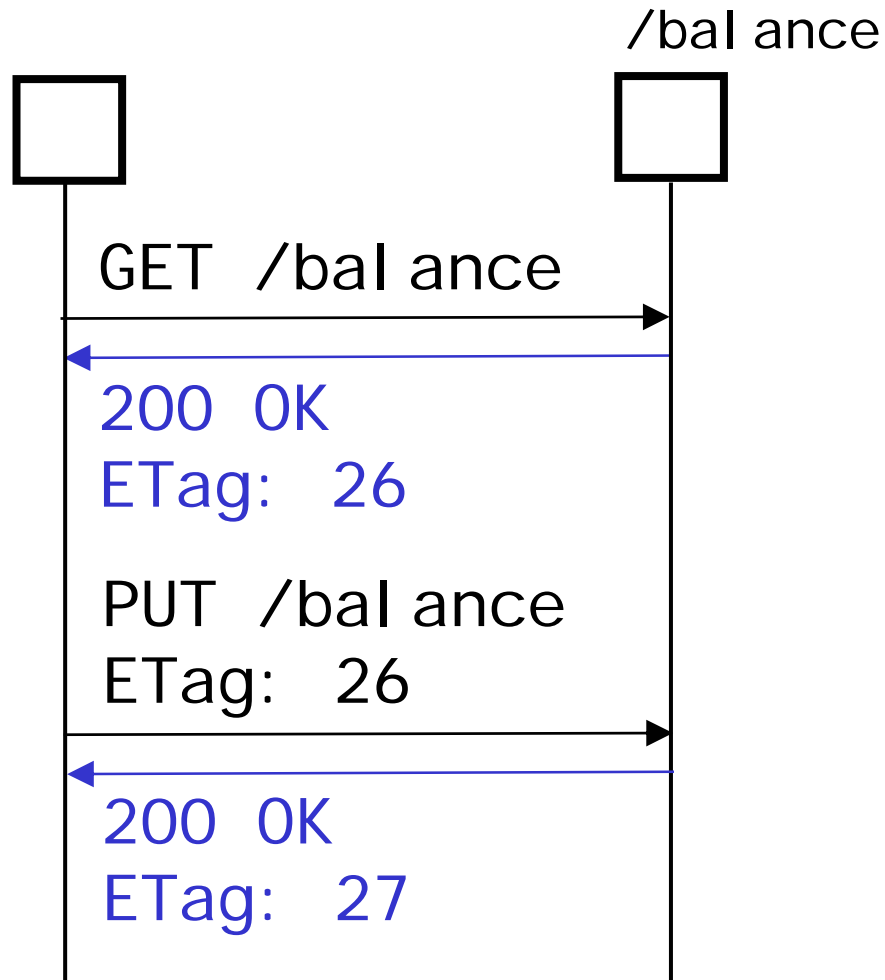
- In some cases the API can be redesigned to use idempotent operations:

B = GetBalance() //safe

B = B + 200\$ //local

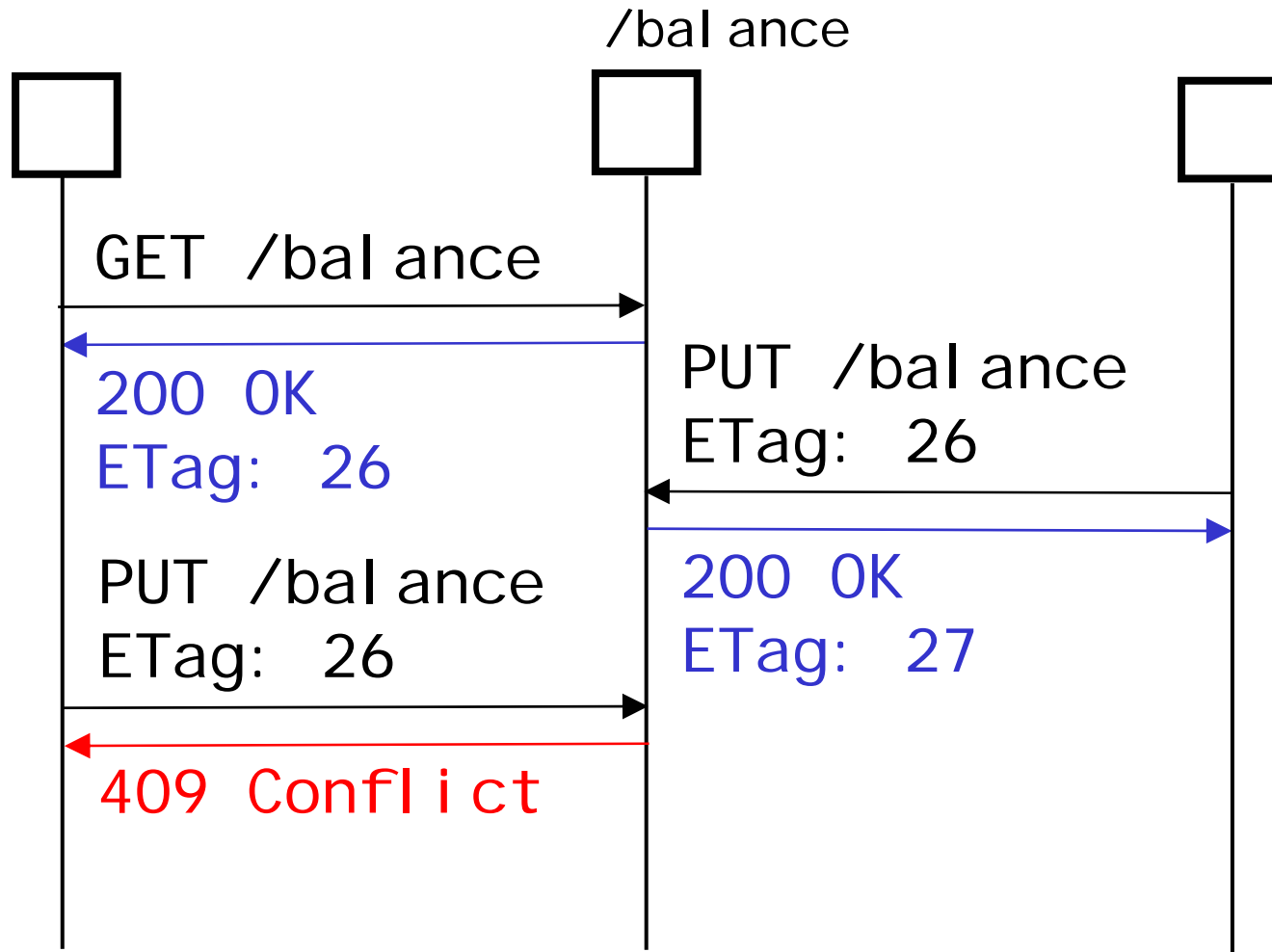
SetBalance(B) //idempotent

Dealing with Concurrency



- Breaking down the API into a set of idempotent requests helps to deal with temporary failures.
- But what about if another client concurrently modifies the state of the resource we are about to update?

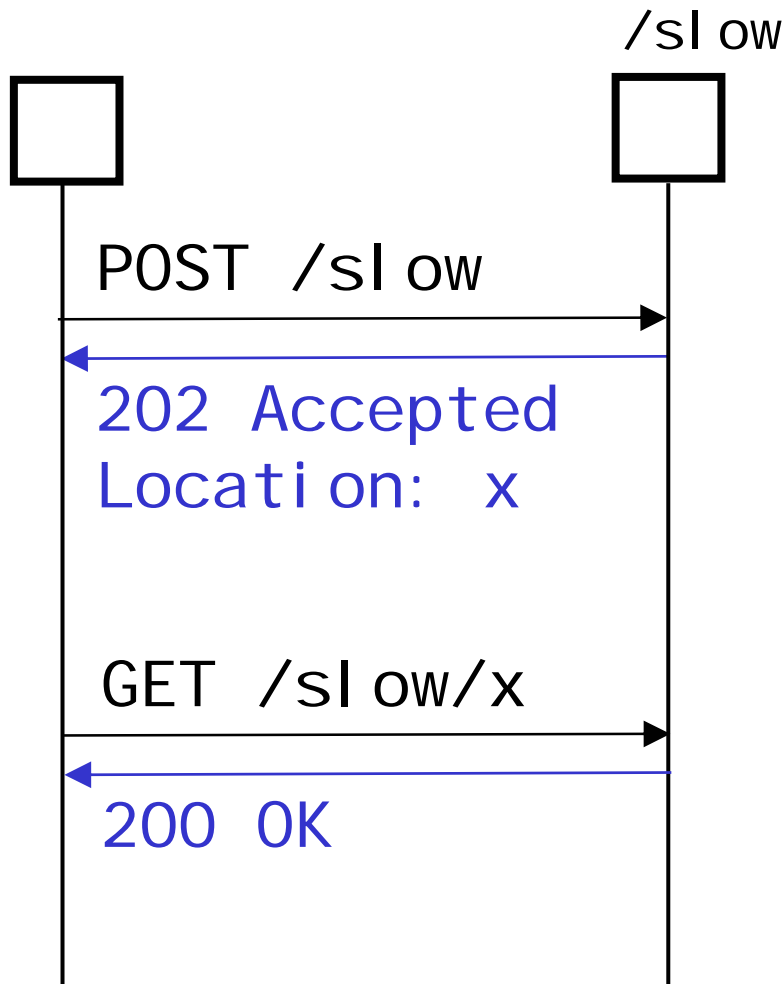
Dealing with Concurrency



The 409 status code can be used to inform a client that his request would render the state of the resource inconsistent

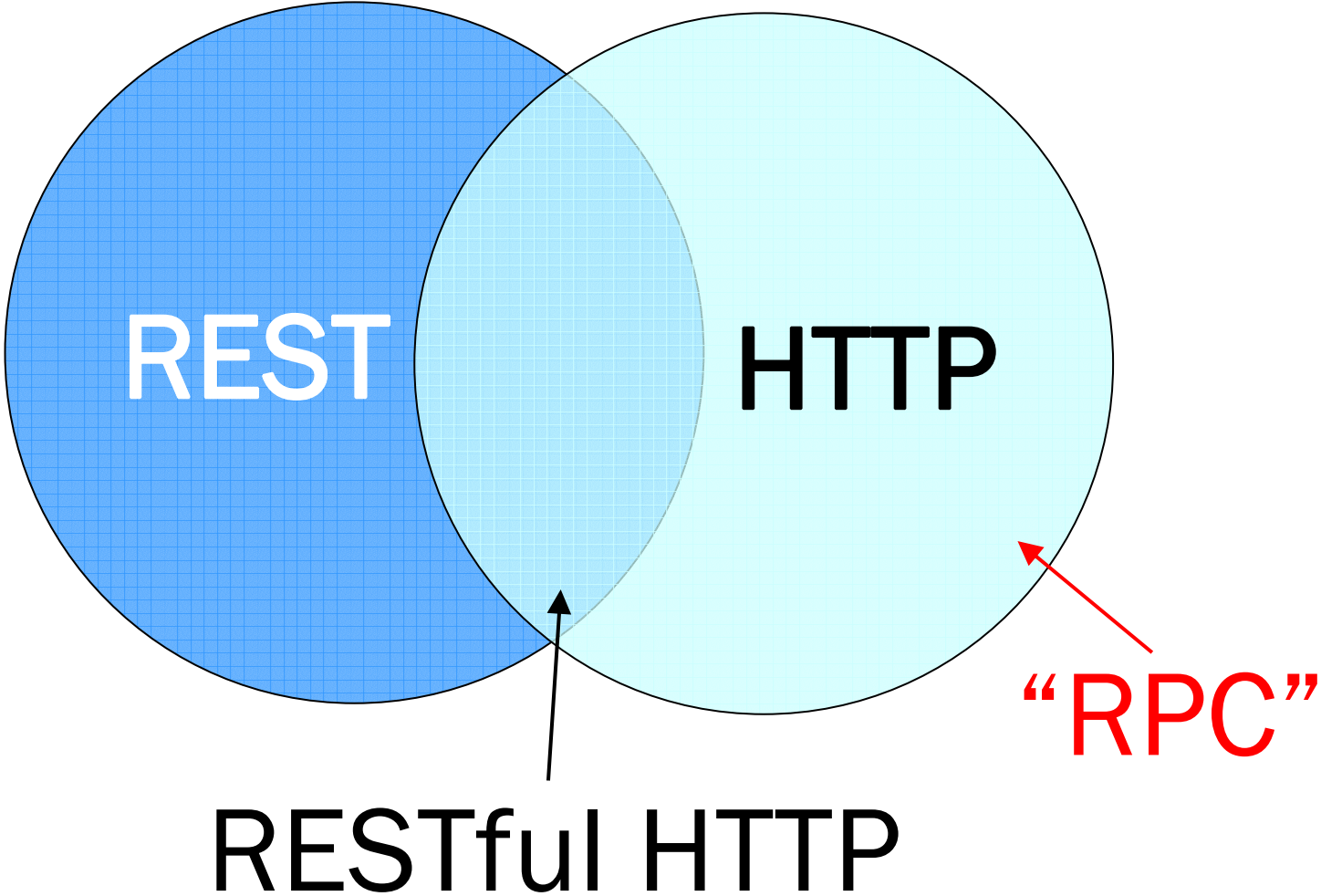
Blocking or Non-Blocking?

- HTTP is a synchronous interaction protocol. However, it does not need to be blocking.



- A Long running request may time out.
- The server may answer it with 202 Accepted providing a URI from which the response can be retrieved later.
- Problem: how often should the client do the polling?

Antipatterns - REST vs. HTTP



Antipatterns – HTTP as a tunnel

- Tunnel through one HTTP Method

GET /api ?method=addCustomer&name=Wi I de

GET /api ?method=deleteCustomer&i d=42

GET /api ?method=getCustomerName&i d=42

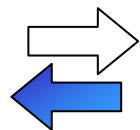
GET /api ?method=findCustomers&name=Wi I de*

- Everything through GET

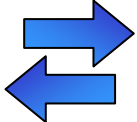
- Advantage: Easy to test from a Browser address bar (the “action” is represented in the resource URI)

- **Problem: GET should only be used for read-only (= idempotent and safe) requests.**

What happens if you bookmark one of those links?



- Limitation: Requests can only send up to approx. 4KB of data (414 Request-URI Too Long)

- Tunnel through one HTTP Method
 - Everything through POST
- 
- Advantage: Can upload/download an arbitrary amount of data (this is what SOAP or XML-RPC do)
 - Problem: POST is not idempotent and is unsafe (cannot cache and should only be used for “dangerous” requests)

POST /service/endpoint

```
<soap:Envelope>
  <soap:Body>
    <findCustomers>
      <name>Wilde* </name>
    </findCustomers>
  </soap:Body>
</soap:Envelope>
```



- Are Cookies RESTful or not?
 - It depends. REST is about stateless communication (without establishing any session between the client and the server)
- 1. Cookies can also be self-contained
 - carry all the information required to interpret them with every request/response
- 2. Cookies contain references to the application state (not maintained as a resource)
 - they only carry the so-called “session-key”
 - Advantage: less data to transfer
 - Disadvantage: the request messages are no longer self-contained as they refer to some context that the server needs to maintain. Also, some garbage collection mechanism for cleaning up inactive sessions is required. More expensive to scale-up the server.

Stateless or Stateful?

- RESTful Web services are not stateless. The very name of “Representational State Transfer” is centered around how to deal with state in a distributed system.

Client State

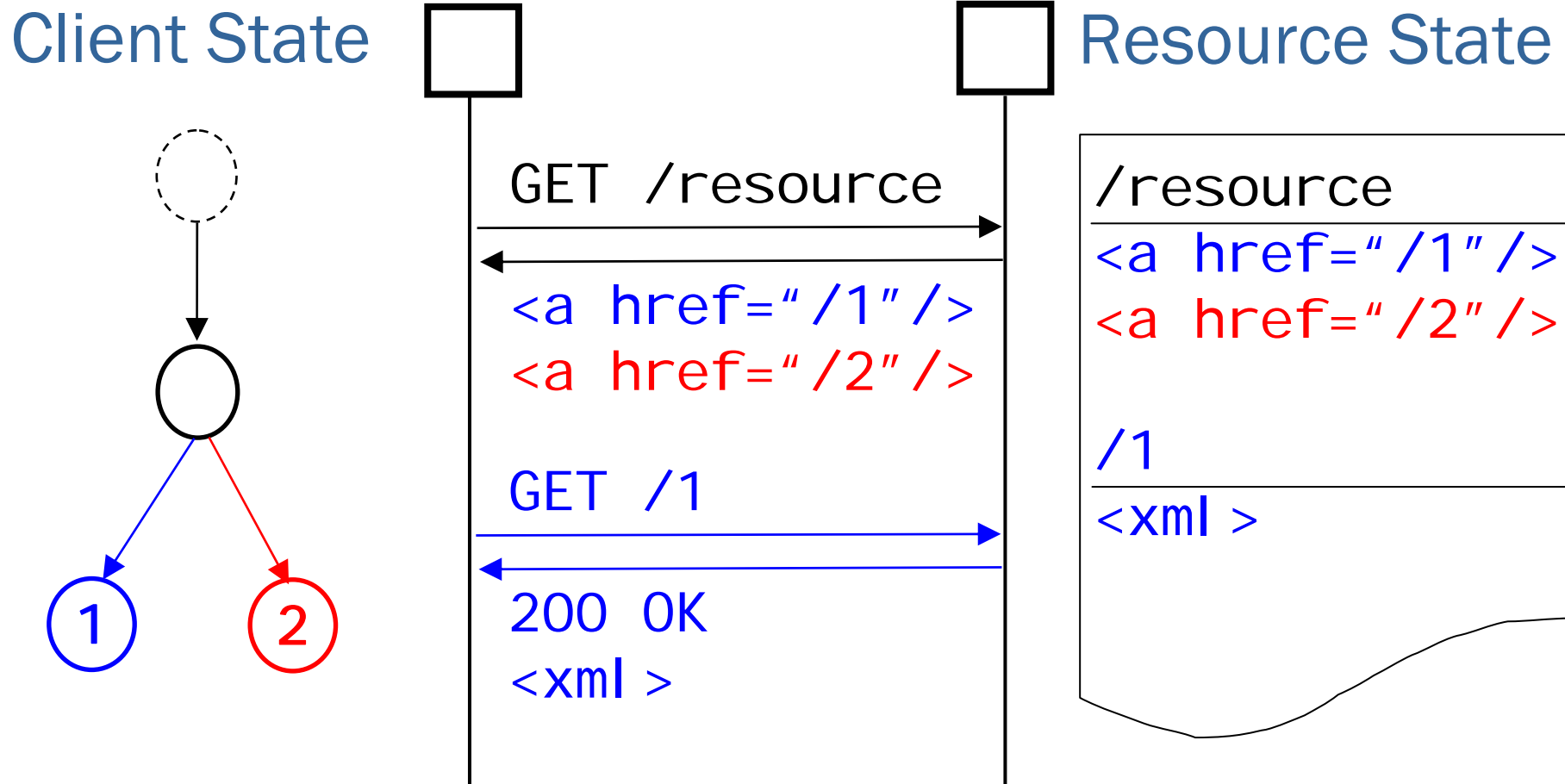
- The client interacts with resources by “navigating hyperlinks” and its state captures the current position in the hypertext.
- The server may influence the state transitions of the client by sending different representations (containing hyperlinks to be followed) in response to GET requests

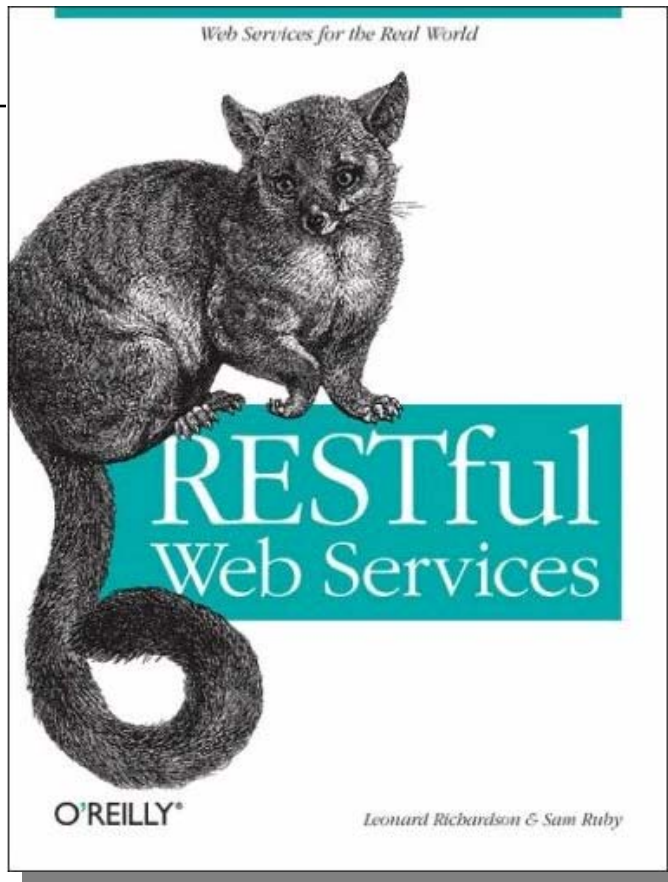
Resource State

- The state of resources captures the persistent state of the service.
- This state can be accessed by clients under different representations
- The client manipulates the state of resources using the uniform interface CRUD-like semantics (PUT, DELETE, POST)

Stateless or Stateful?

- RESTful Web services are not stateless. The very name of “Representational State Transfer” is centered around how to deal with state in a distributed system.





Leonard Richardson,
Sam Ruby,
RESTful Web Services,
O'Reilly, May 2007



Raj Balasubramanians, Benjamin
Carlyle, Thomas Erl, Cesare Pautasso,
SOA with REST,
Prentice Hall, End of 2009