

SOA with REST

Cesare Pautasso

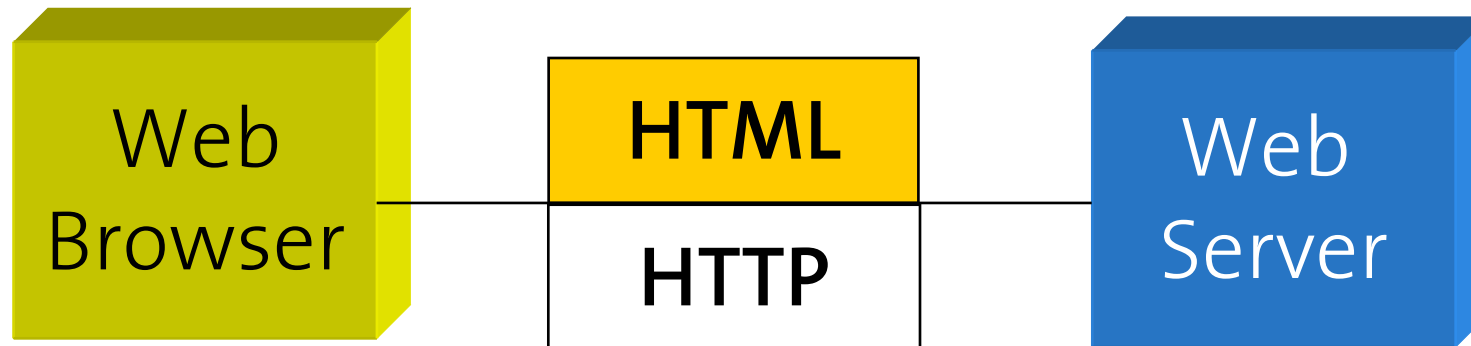
Faculty of Informatics

University of Lugano

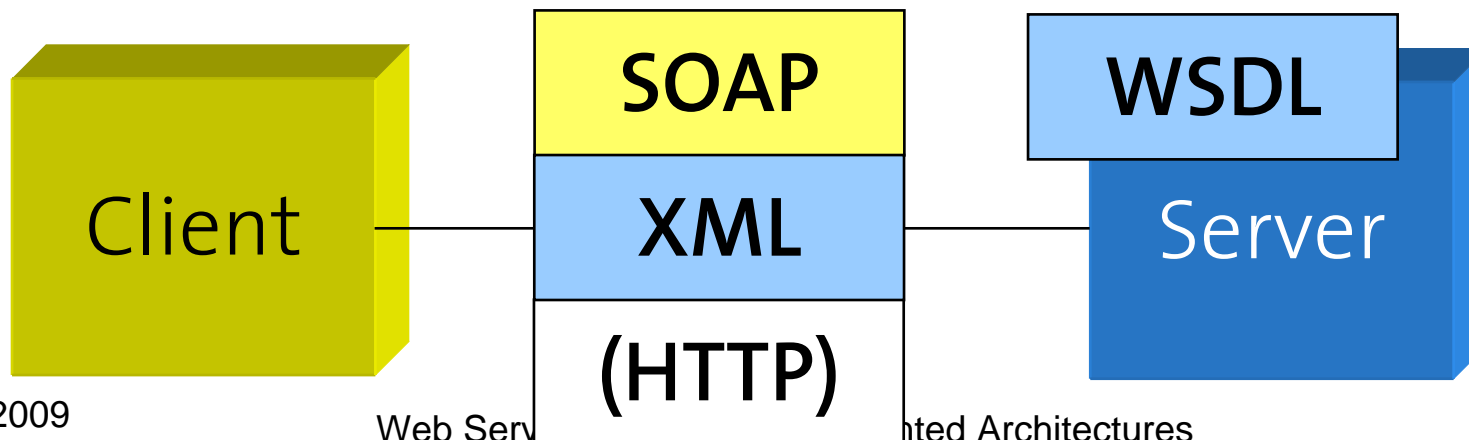
<http://www.pautasso.info>

c.pautasso@ieee.org

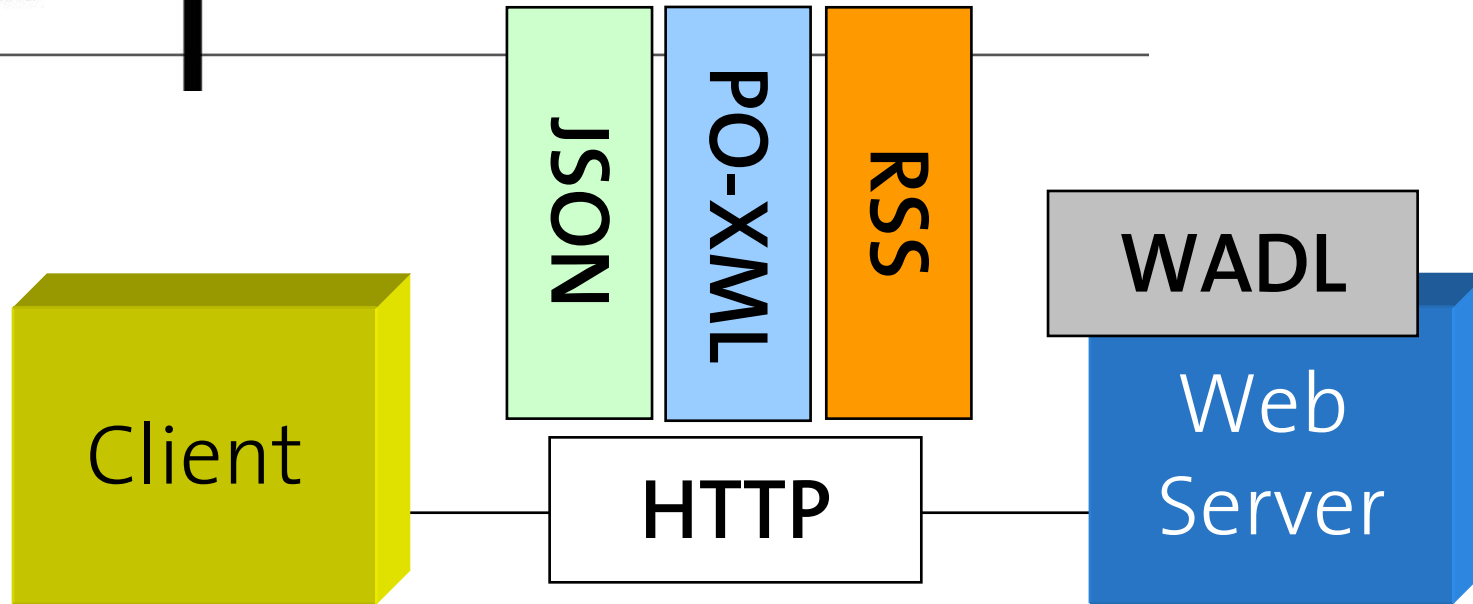
Web Sites (1992)



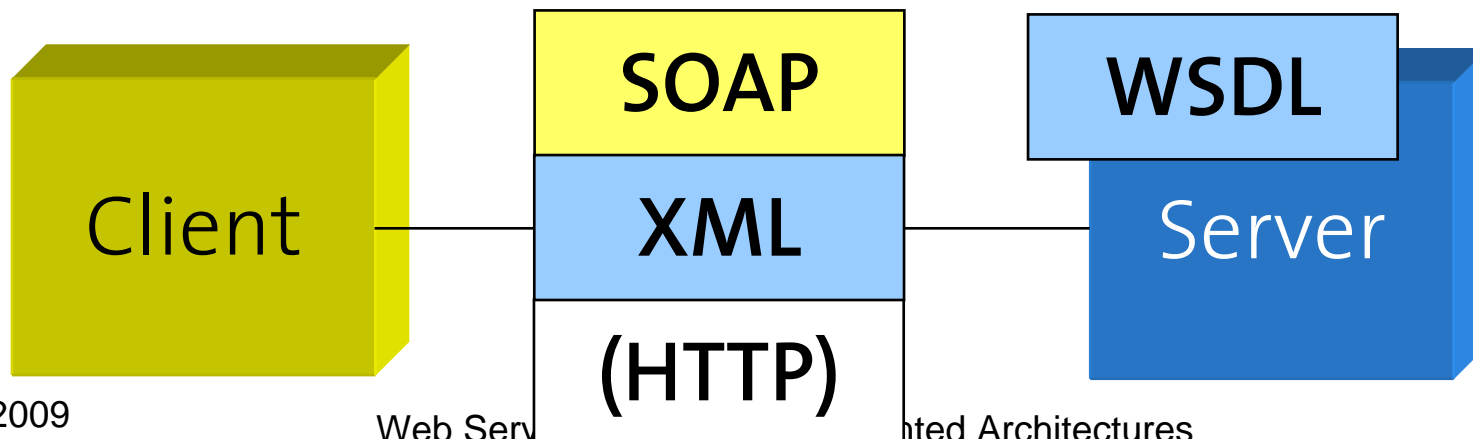
WS-* Web Services (2000)



RESTful Web Services (2007)



WS-* Web Services (2000)



Where do Web services come from?

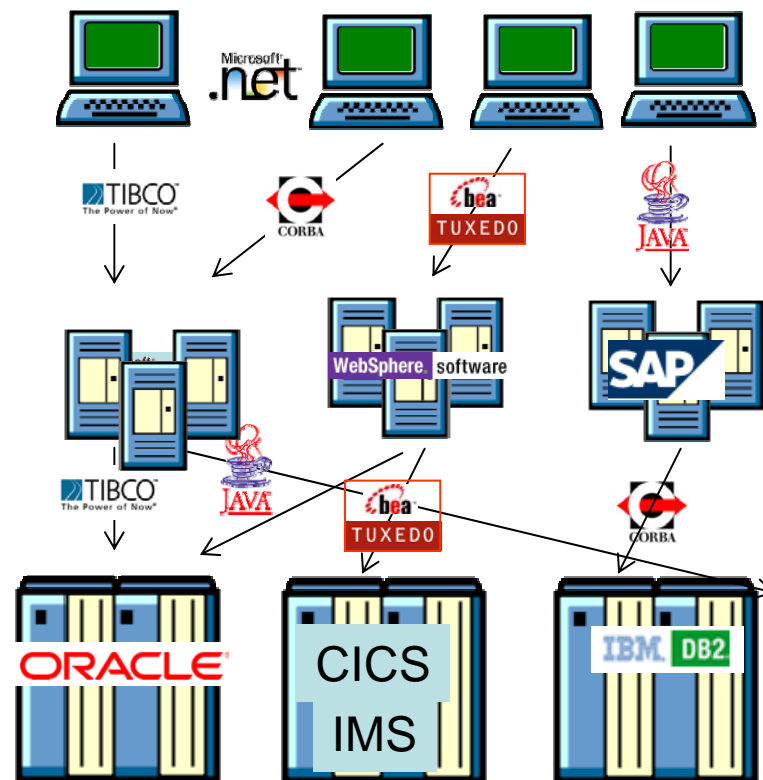
- Address the **problem of enterprise software standardization**
- Enterprise Computing Standards for Interoperability (WS started 2001)
- A layered architecture with a variety of messaging, description and discovery specifications
- Do things from the ground up, quickly, in well factored, distinct, tightly focused specifications
- Tools will hide the complexity
- REST advocates have come to believe that their ideas are just as applicable to solve application integration problems.
- Are all specifications really composable?
- “Look ma’, no tools!”

Dealing with Heterogeneity

- Web Applications



- Enterprise Computing



Picture from Eric Newcomer, IONA

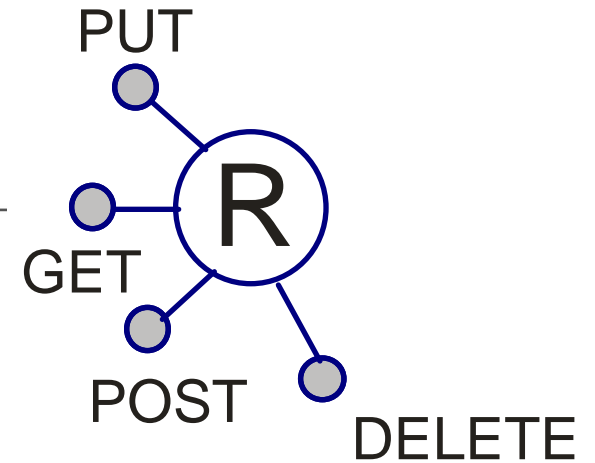
Big Web Services

- High perceived complexity
- Problematic standardization process
 - Infighting
 - Lack of architectural coherence
 - Fragmentation
 - Design by committee
 - Feature Bloat (Merge of competing specs)
 - Lack of reference implementations
 - Standardization of standards (WS-I)

WS-PageCount	
Messaging	232 pages
Metadata	111 pages
Security	230 pages
WS-BPEL	195 pages
XML/XSD	599 pages
Transactions	39 pages

Numbers from Tim Bray

- Is this starting to look like CORBA?
- When will Web services interoperability start to really work?
- Do we really have to buy XML appliances to get good performance?

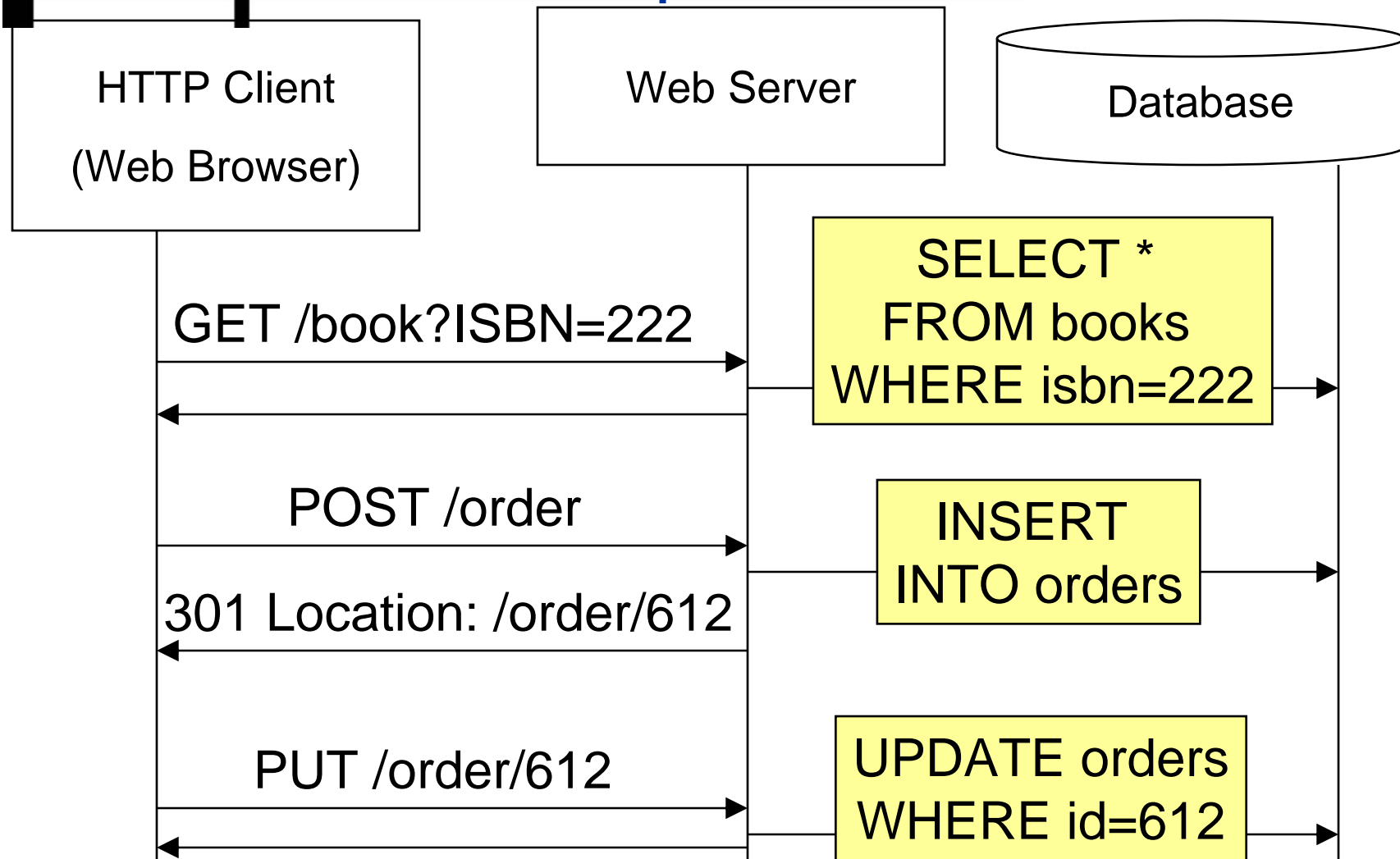


- What is REST?
- RESTful Service Design
- Comparison WS-* vs. REST
- Discussion
- Outlook


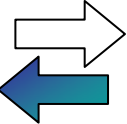
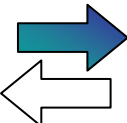
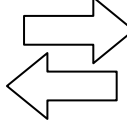
REpresentational State Transfer

- REST defines the architectural style of the Web
 - Its four principles can explain the success and the scalability of the HTTP protocol implementing them
1. **Resource Identification** through URI
 2. **Uniform Interface** for all resources:
 - GET (Query the state, idempotent, can be cached)
 - POST (Create a child resource)
 - PUT (Update, transfer a new state)
 - DELETE (Delete a resource)
 3. **“Self-Describing”** Messages through Meta-Data and multiple resource representations
 4. **Hyperlinks** to define the application state transitions and relationships between resources

RESTful Web Service Example



Uniform Interface Principle (CRUD Example)

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

URI

Uniform Resource Identifier

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

http://tools.ietf.org/html/rfc3986

URI Scheme Authority Path

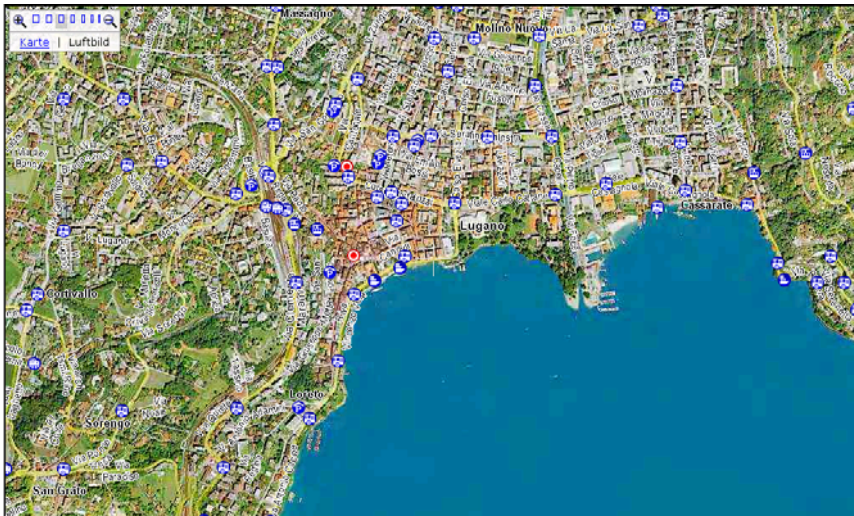
https://www.google.ch/search?q=rest&start=10#1

Query Fragment

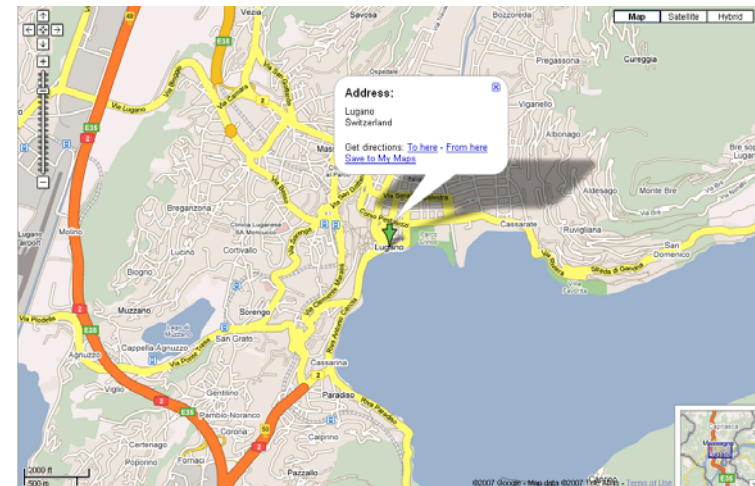
- REST does **not** advocate the use of “nice” URIs
- In most HTTP stacks URIs cannot have arbitrary length (4Kb)

What is a “nice” URI?

<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

High REST vs. Low REST

- Best practices differ:

Low REST

- HTTP GET for idempotent requests, POST for everything else
- Responses in any MIME Type (e.g., XHTML)

High REST

- Usage of “nice” URIs recommended
- Full use of the 4 verbs: GET, POST, PUT, and DELETE (*)
- Responses using Plain Old XML (**)

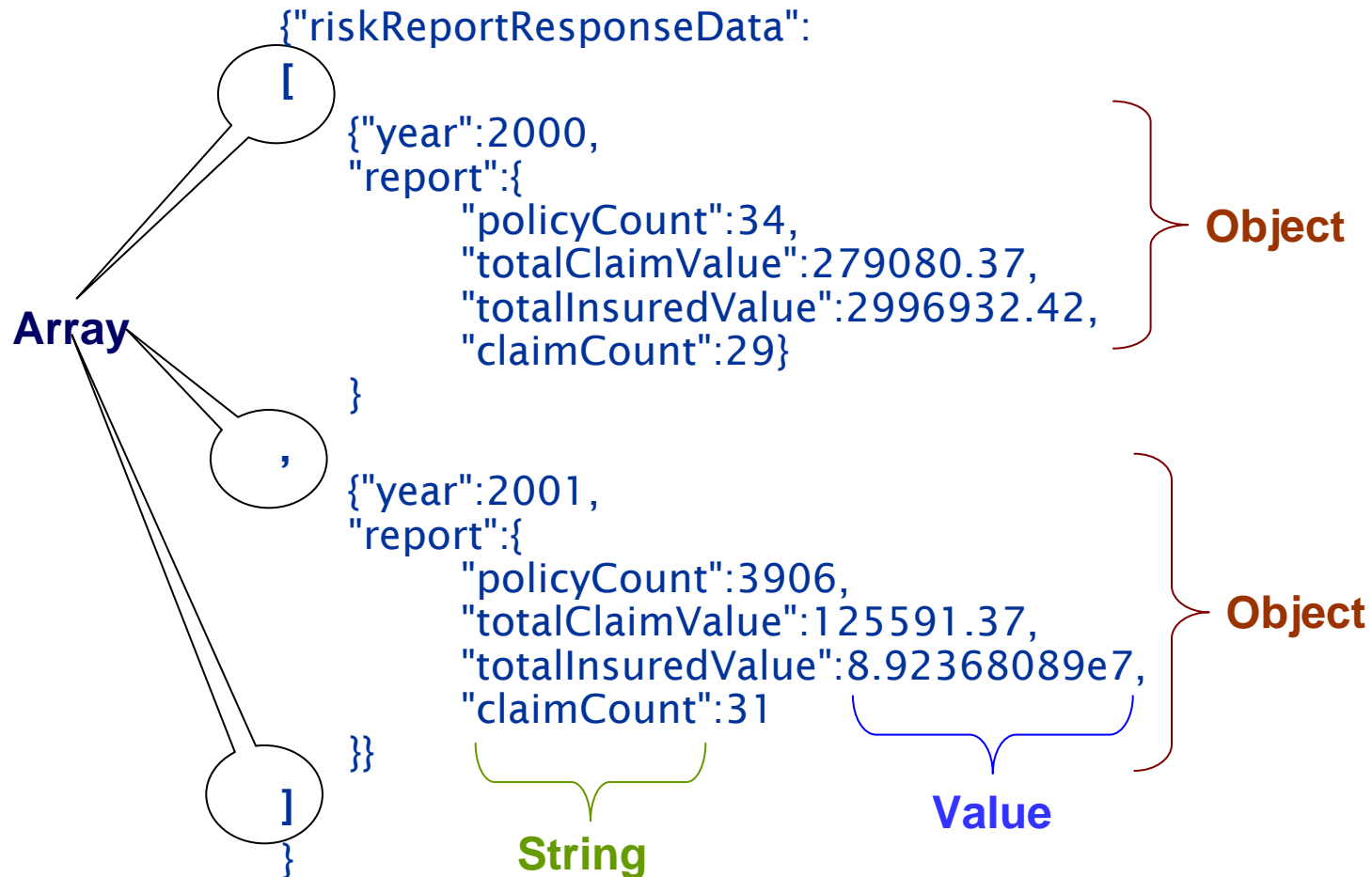
(*) Some firewalls or HTTP Proxies may drop PUT/DELETE requests.

(**) POX could be replaced with RDF, JSON, YAML, or ATOM (highly debated)

Resource Representation Formats: XML vs. JSON

- XML
 - PO-XML
 - SOAP (WS-*)
 - RSS, ATOM
- Standard textual syntax for semi-structured data
- Many tools available: XML Schema, DOM, SAX, XPath, XSLT, XQuery
- Everyone can parse it (not necessarily understand it)
- Slow and Verbose
- JavaScript Object Notation (JSON)
- Wire format introduced for AJAX Web applications (Browser-Web Server communication)
- Textual syntax for serialization of non-recurrent data structures
- Supported in most languages (not only JavaScript)
- Not extensible (does not need to be)
- “JSON has become the X in Ajax”

JSON Example



REST Strengths

- Simplicity
 - Uniform interface is **immutable** (no problem of breaking clients)
- HTTP/POX is ubiquitous (goes through firewalls)
- Stateless/Synchronous interaction
- Proven scalability
 - “after all the Web works”, **caching**, clustered server farms for QoS
- Perceived ease of adoption (light infrastructure)
 - just need a browser to get started - no need to buy WS-* middleware
- Grassroots approach
- Leveraged by all major Web 2.0 applications
 - 85% clients prefer Amazon RESTful API (*)
 - Google does no longer support its SOAP/WSDL API

(*) <http://www.oreilynet.com/pub/wlg/3005>

Is REST being used?



Slide from Paul Downey, BT

REST Weaknesses

- Confusion (high REST vs. low REST)
 - Is it really 4 verbs? (HTTP 1.1. has 8 verbs: HEAD, **GET**, **POST**, PUT, DELETE, TRACE, OPTIONS, and CONNECT)
- Mapping REST-style synchronous semantics on top of back end systems creates design mismatches (when they are based on asynchronous messaging or event driven interaction)
- Cannot deliver enterprise-style “-ilities” beyond HTTP/SSL
- Challenging to identify and locate resources appropriately in all applications
- Apparent lack of standards (other than URI, HTTP, XML, MIME, HTML)
- Semantics/Syntax description very informal (user/human oriented)

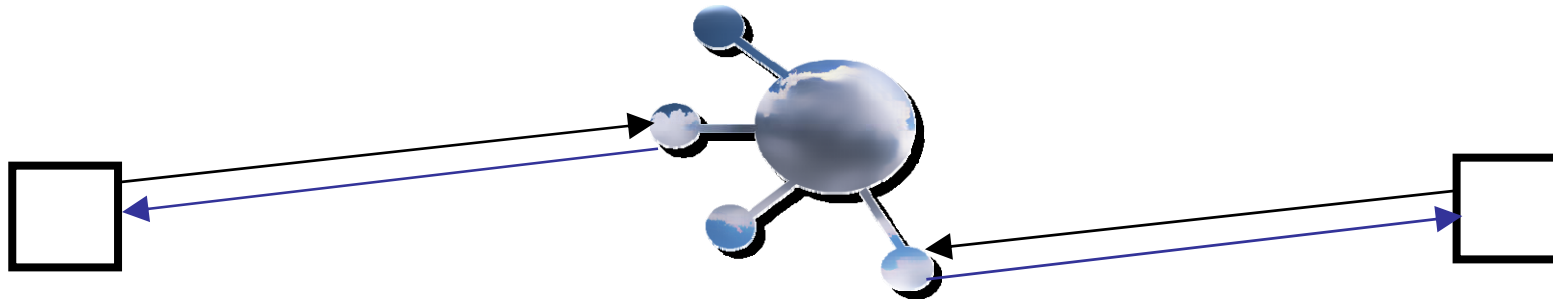
RESTful Web Services Design Methodology

1. Identify resources to be exposed as services (e.g., yearly risk report, book catalog, purchase order, open bugs, polls and votes)
2. Define “nice” URLs to address them
3. Understand what it means to do a GET, POST, PUT, DELETE on a given resource URI
4. Design and document resource representations
5. Model relationships (e.g., containment, reference, state transitions) between resources with hyperlinks that can be followed to get more details (or perform state transitions)
6. Implement and deploy on Web server
7. Test with a Web browser

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

Simple Doodle API Example

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



POST /poll
<options>A, B, C</options>

201 Created
Location: /poll/090331x

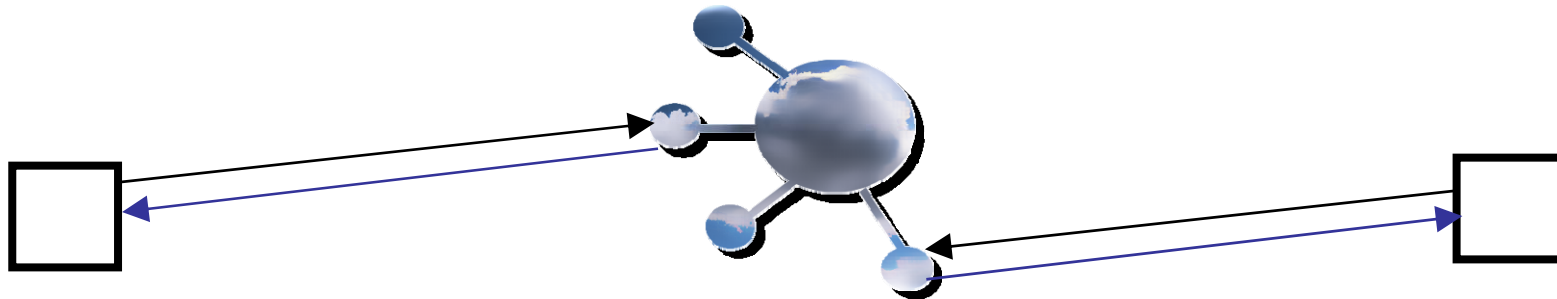
GET /poll/090331x

200 OK
<options>A, B, C</options>

- 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



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

201 Created

Location:

</poll/090331x/vote/1>

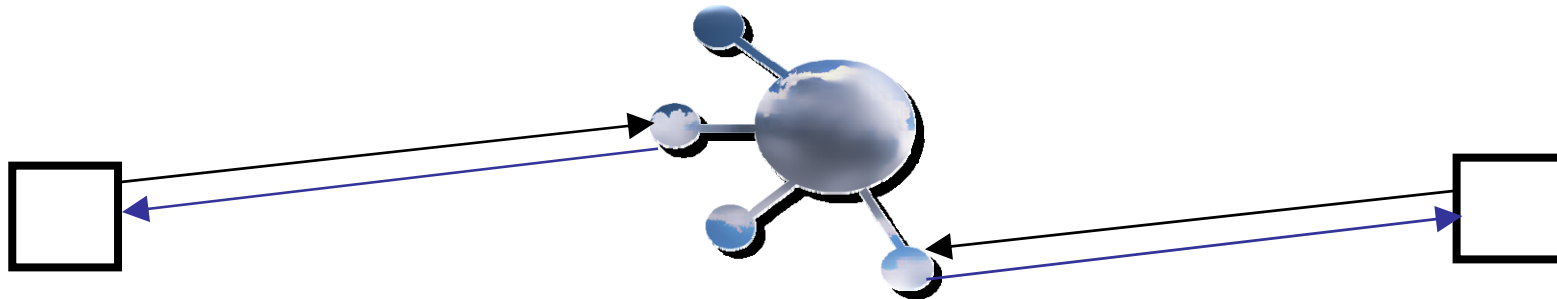
```
GET /poll/090331x
```

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 /poll/090331x/vote/1  
<name>C. Pautasso</name>  
<choice>C</choice>
```

200 OK

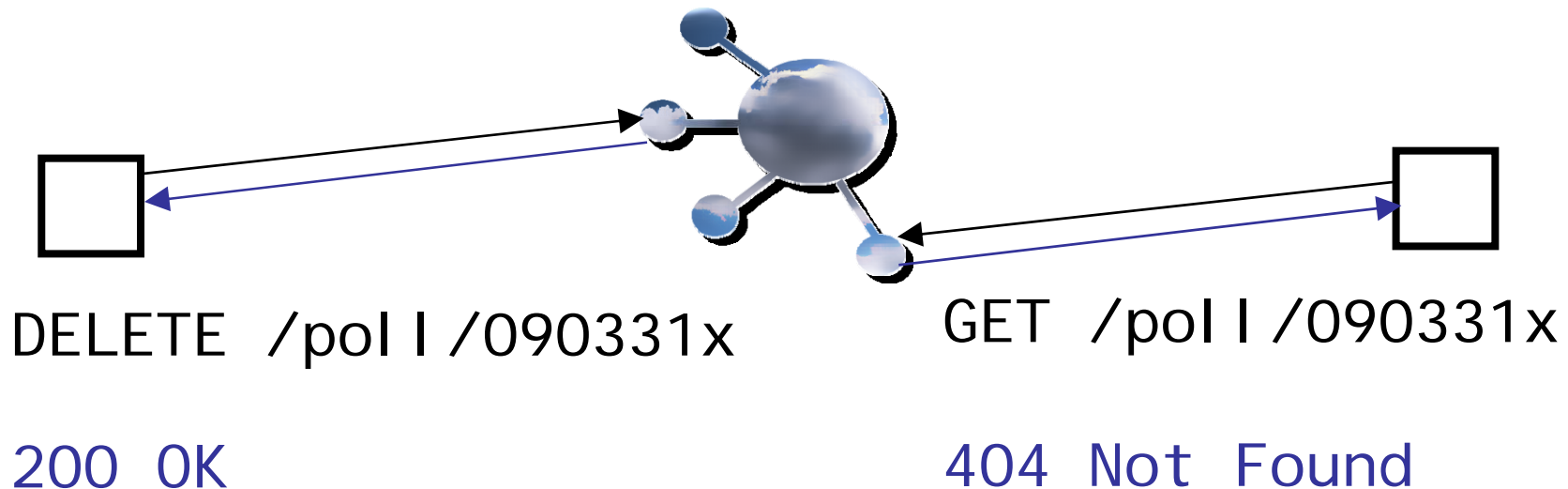
```
GET /poll/090331x
```

200 OK

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

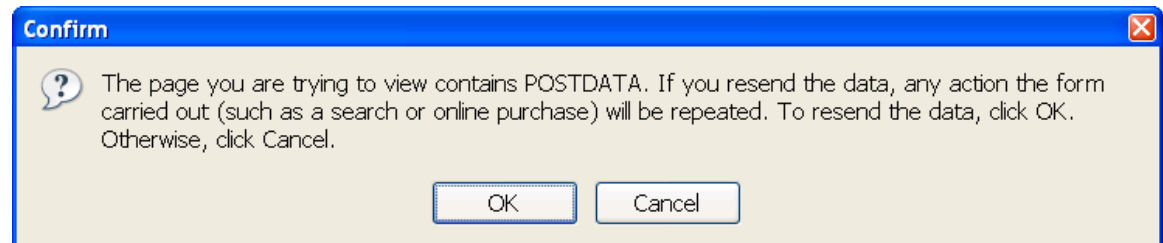
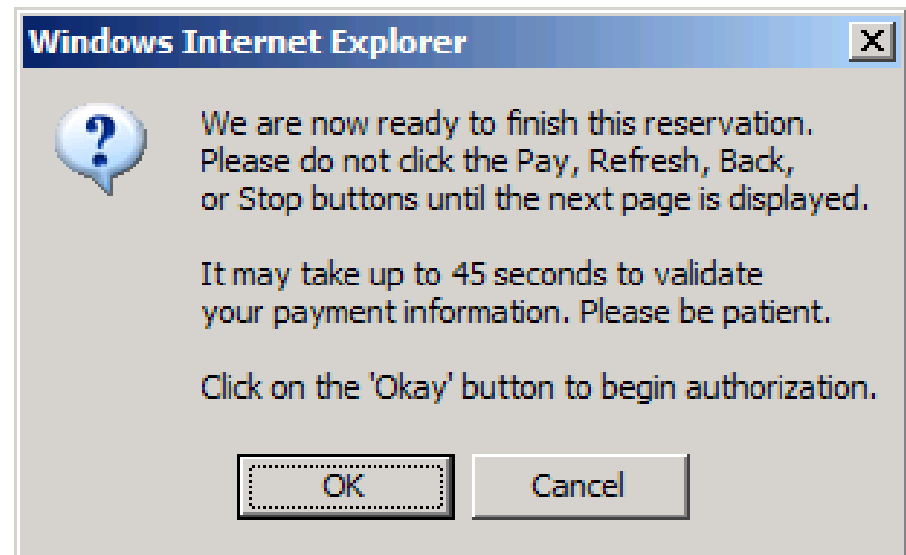
REST Design Tips

1. Understanding GET vs. POST vs. PUT
2. Multiple Representations
 - Content-Type Negotiation
3. Exception Handling
 - Idempotent vs. Unsafe

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}`

Problem: How to ensure resource {id} is unique?

(Resources can be created by multiple clients concurrently)

POST `/resource`

201 Created

Location: `/resource/{id}`

Solution: let the server compute the unique id

Content Negotiation (Conneg)

Negotiating the message format does not require to send more messages

GET /resource

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

1. The client lists the set of format (MIME types) that it understands

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

GET /resource.html

GET /resource.xml

GET /resource.json

Warning: This is a conventional “best practice” (not a standard)

Exception Handling

Learn to use HTTP Standard Status Codes

100 Continue
200 OK
201 Created
202 Accepted
203 Non-Authoritative
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 (the state of the server does not change)

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

- Unsafe requests modify the state of the server and cannot be repeated without further 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) //safe

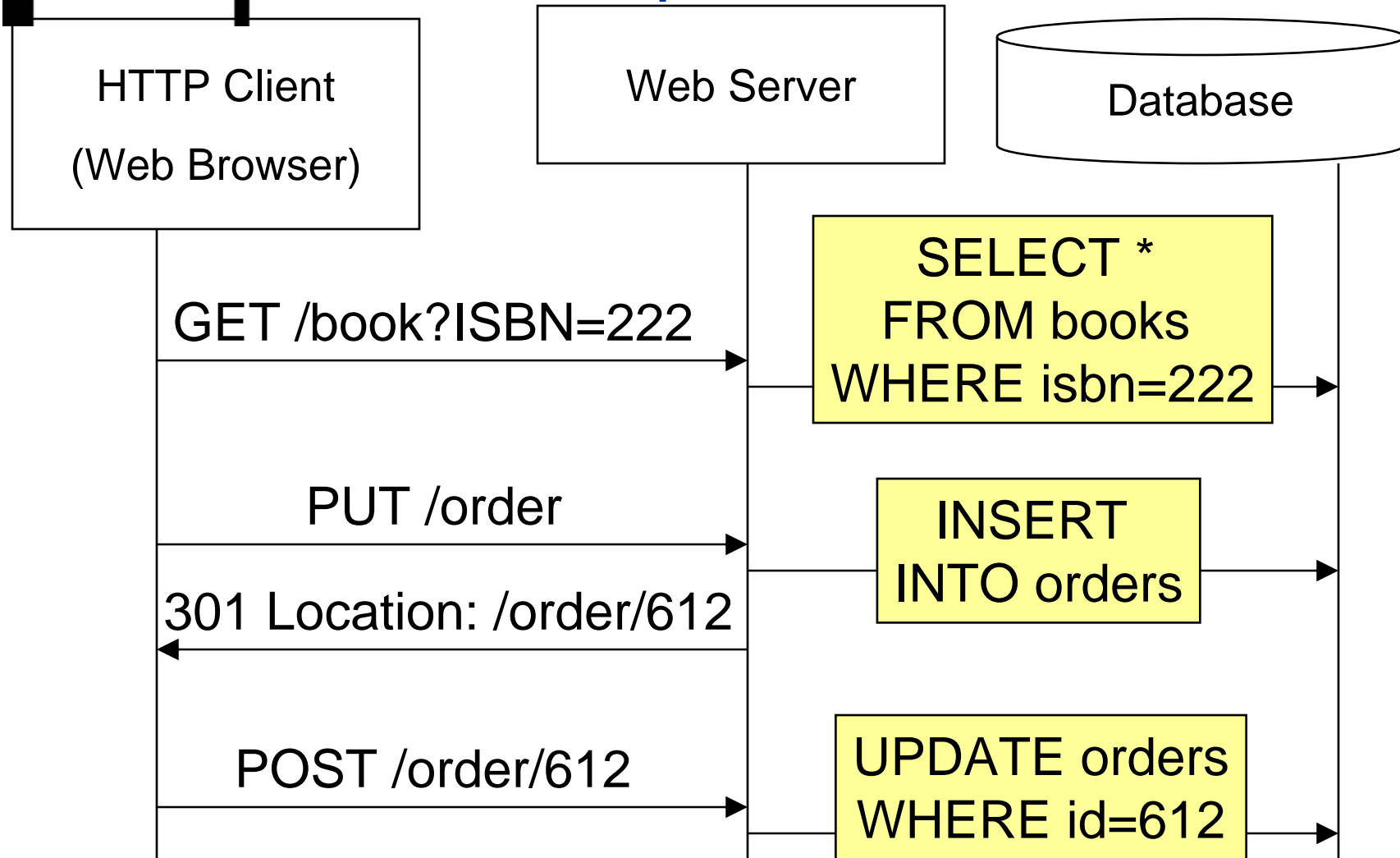
Comparing WS-* vs. REST?



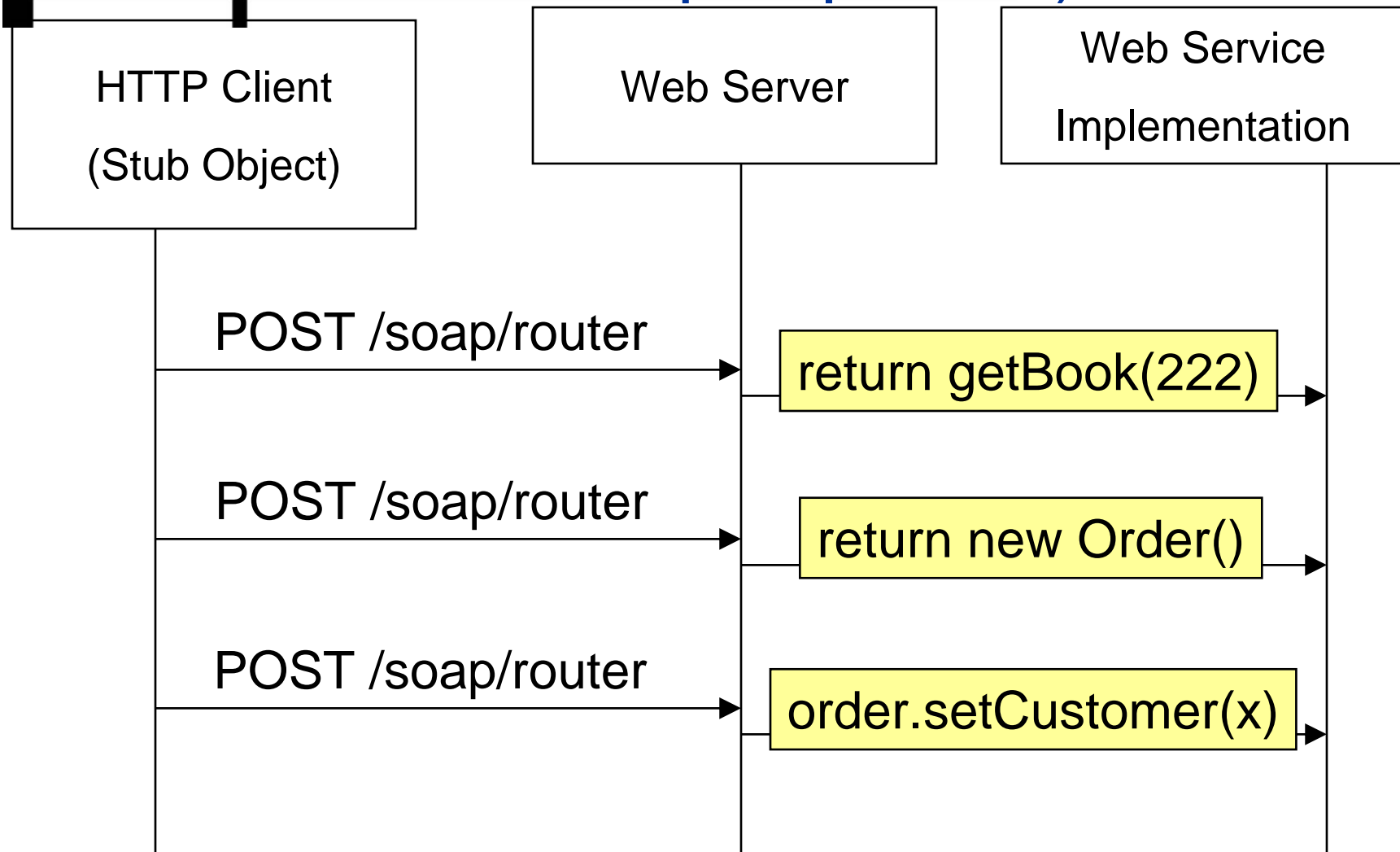
SOAP vs. REST Comparison

- Example
- Conceptual Comparison
 - REST as a connector
- Technology Comparison
 - What about service description?
 - What about security?
 - What about state management?
 - What about asynchronous messaging?

RESTful Web Application Example



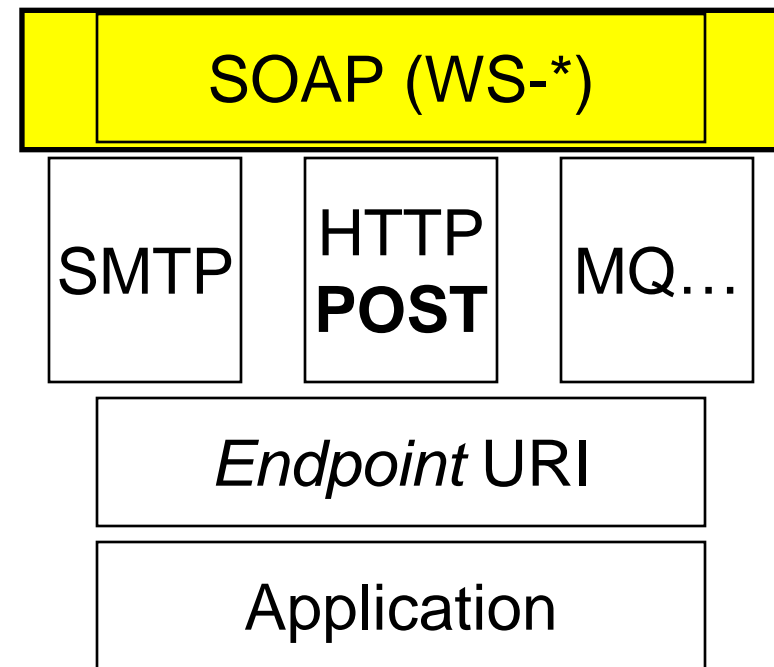
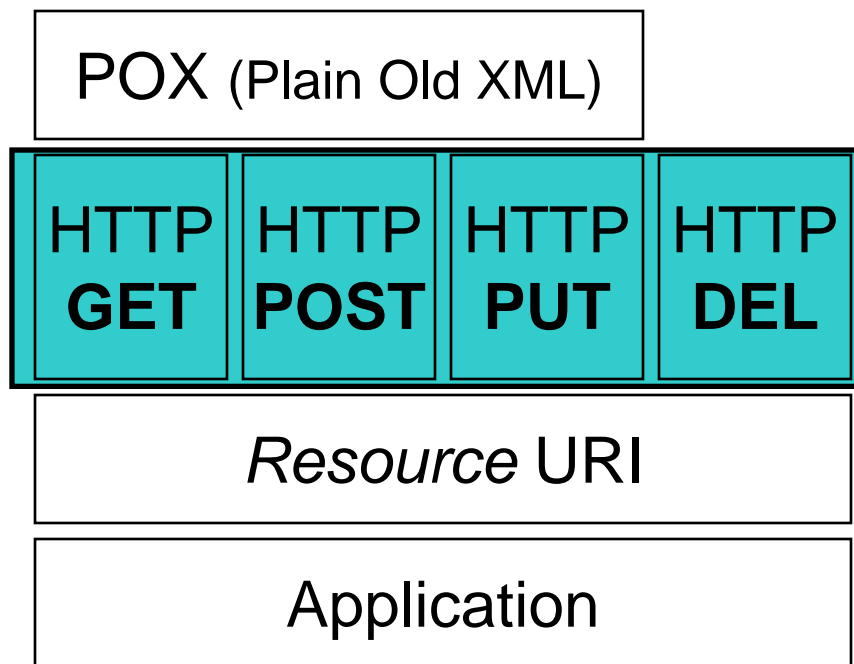
Web Service Example (from REST perspective)



Main difference: REST vs. SOAP

- “The Web is the universe of globally accessible information”
(Tim Berners Lee)
 - Applications should publish their data on the Web (through URI)

- “The Web is the universal transport for messages”
 - Applications get a chance to interact but they remain “outside of the Web”



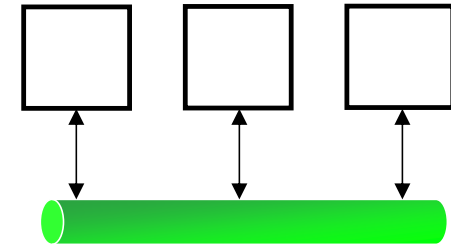
REST as a connector



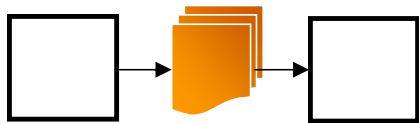
Stream



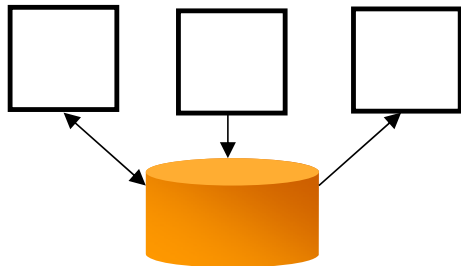
Remote Procedure Call



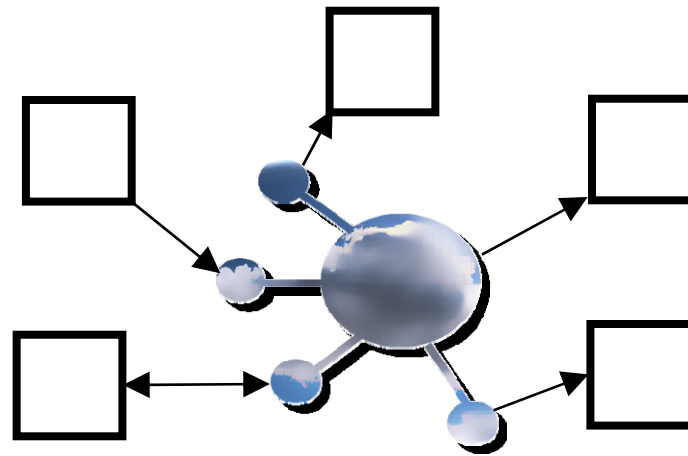
Message Bus
Events



File Transfer



Shared Data



REpresentational
State Transfer

Stateless or Stateful?

- REST provides explicit state transitions
 - Communication is stateless*
 - Resources contain data **and links** representing valid state transitions
 - Clients maintain state correctly by following links in generic manner
- Techniques for adding session to HTTP:
 - Cookies (HTTP Headers)
 - URL Re-writing
 - Hidden Form Fields
- SOAP services have implicit state transitions
 - Servers may maintain conversation state across multiple message exchanges
 - Messages contain only data (but do not include information about valid state transitions)
 - Clients maintain state by guessing the state machine of the service
- Techniques for adding session to SOAP:
 - Session Headers (non standard)

(*) Each client request to the server must contain all information needed to understand the request, without referring to any stored context on the server. Of course the server stores the state of its resources, shared by all clients.

What about service description?

- REST relies on human readable documentation that defines requests URIs and responses (XML, JSON)
- Interacting with the service means hours of testing and debugging URIs manually built as parameter combinations. (Is it really that simpler building URIs by hand?)
- Why do we need strongly typed SOAP messages if both sides already agree on the content?
- WADL proposed Nov. 2006
- XML Forms enough?
- Client stubs can be built from WSDL descriptions in most programming languages
- Strong typing
- Each service publishes its own interface with different semantics
- WSDL 1.1 (entire port type can be bound to HTTP GET or HTTP POST or SOAP/HTTP POST or other protocols)
- WSDL 2.0 (more flexible, each operation can choose whether to use GET or POST)

What about security?

- REST security is all about HTTPS
- Proven track record (SSL1.0 from 1994)
- Secure, point to point communication (Authentication, Integrity and Encryption)
- SOAP security extensions defined by WS-Security (from 2004)
- XML Encryption (2002)
- XML Signature (2001)
- Implementations are starting to appear now
 - Full interoperability moot
 - Performance?
- Secure, end-to-end communication – Self-protecting SOAP messages (does not require HTTPS)

What about asynchronous reliable messaging?

- Although HTTP is a synchronous protocol, it can be used to “simulate” a message queue.

POST /queue

202 Accepted

Location:

/queue/message/1230213

GET /queue/message/1230213

DELETE

/queue/message/1230213

- SOAP messages can be transferred using asynchronous transport protocols and APIs (like JMS, MQ, ...)
- WS-Addressing can be used to define transport-independent endpoint references
- WS-ReliableExchange defines a protocol for reliable message delivery based on SOAP headers for message identification and acknowledgement

SOAP and REST

Some slides from:

Noah Mendelsohn
Christopher Ferris
James Snell
IBM

- Bringing the Web back into Web services
- Conclusion and Outlook

Putting the Web back into Web services

- RESTafarians would like Web services to use and not to abuse the architecture of the Web
- Web Services more valuable when accessible from the Web
- Web more valuable when Web Services are a part of it
- **W3C Workshop on Web of Services for Enterprise Computing**, 27-28 February 2007 – with IBM, HP, BEA, IONA, Yahoo, Sonic, Redhat/JBoss, WSO2, Xerox, BT, Coactus Consulting, Progress Software, and others.

REST and SOAP/WS*

Similarities

- XML
 - XML Schema
 - HTTP
 - Loose coupling important for scalability
- O/XML binding problematic
 - Alternatives to XML starting to appear (e.g., JSON, YAML, RDF)
 - Data contract needs to be defined even with uniform interface
 - Although REST claims SOAP misuses the POST verb
 - Interpretations differ
 - Synchronous vs. Asynchronous
 - Uniform Interface never changes

REST and SOAP Similarities

- Existing Web applications can gracefully support both traditional Web clients (HTML/POX) and SOAP clients in a RESTful manner
- MIME Type: **application/soap+xml**
- SOAP with document/literal style not so different from REST (or at least HTTP/POX) - apart from the GET/POST misuse and the extra `<envelope><header><body>` tags in the payload

Debunking the Myth

- Many “RESTafarians” have taken the position that REST and Web services are somehow incompatible with one another
- **Fact:** recent versions of SOAP and WSDL have been designed specifically to enable more RESTful use of Web services
- SOAP1.2
 - SOAP1.2 Response MEP
 - Web Method
- WSDL2.0
 - HTTP binding permits assigning verbs (GET, POST, etc.) on a per-operation basis
 - Attribute to mark an operation as safe (*and thus cacheable*)
- Unfortunately, the implementations of Web services runtimes and tooling have made RESTful use of Web services difficult



Conclusion and Outlook



- Service-Oriented Architecture can be implemented in different ways.
- You should generally focus on whatever architecture gets the job done and recognize the significant value of open standards but try to avoid being religious about any specific architectures or technologies.
- SOAP and the family of WS-* specifications have their strengths and weaknesses and will be highly suitable to some applications and positively terrible for others. Likewise with REST. The decision of which to use depends entirely on the circumstances of the application.
- In the near future there will be a single scalable middleware stack, offering the best of the Web in simple scenarios, and scaling gracefully with the addition of optional extensions when more robust quality of service features are required.
- The right steps have been taken in the development of some of the more recent WS-* specifications to enable this vision to become reality

References

- Leonard Richardson, Sam Ruby, **RESTful Web Services**, O'Reilly, May 2007
- Roy Fielding, Architectural Styles and the Design of Network-based Software Architectures, University of California, Irvine, 2000, Chapter 5
http://roy.gbiv.com/pubs/dissertation/fielding_dissertation.pdf
- **W3C Workshop on Web of Services for Enterprise Computing, 27-28 February 2007** <http://www.w3.org/2007/01/wos-ec-program.html>
- Sun, JSR311 - **Java API for RESTful Web Services**
<http://jcp.org/en/jsr/detail?id=311>
- Marc J. Hadley (Sun), Web Application Description Language (**WADL**)
<https://wadl.dev.java.net/>
- Thomas Bayer, **REST Web Services** – Eine Einfuehrung (November 2002)
<http://www.oio.de/public/xml/rest-webservices.pdf>
- Michi Henning, **The Rise and Fall of CORBA**, Component Technologies, Vol. 4. No. 5, June 2006
- Stefan Tilkov, REST Patterns and Antipatterns, JAOO 2008
- Cesare Pautasso, Erik Wilde, **From SOA to REST**, WWW 2009 Tutorial
- Jacob Nielsen, **URI are UI**, <http://www.useit.com/alertbox/990321.html>
- Douglas Crockford, **JSON - the fat-free alternative to XML**, XML 2006.
- Cesare Pautasso, Olaf Zimmermann, Frank Leymann, **RESTful Web Services vs. Big Web Services: Making the Right Architectural Decision**, 17th International World Wide Web Conference (WWW2008), Beijing, China, April 2008.
<http://www.jopera.org/docs/publications/2008/restws>