Lesson 14 – SOA with REST (Part I)

Service Oriented Architectures Security

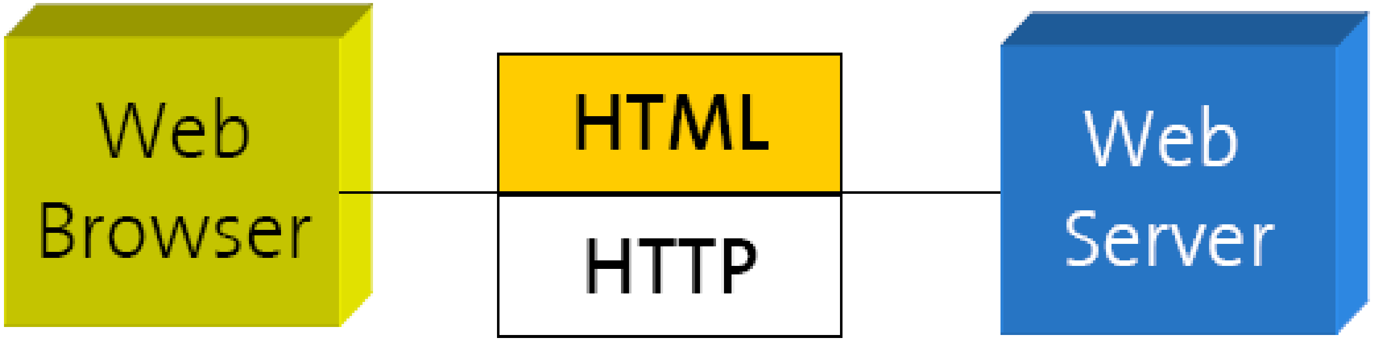
Module 3 - Resource-oriented services

Unit 1 – REST

Ernesto Damiani

Università di Milano

Web Sites (1992)



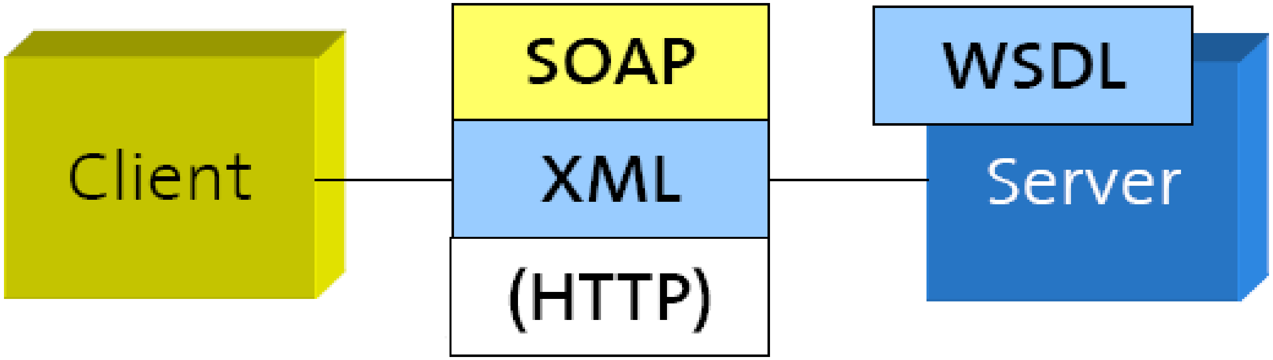
Web Browser

HTML

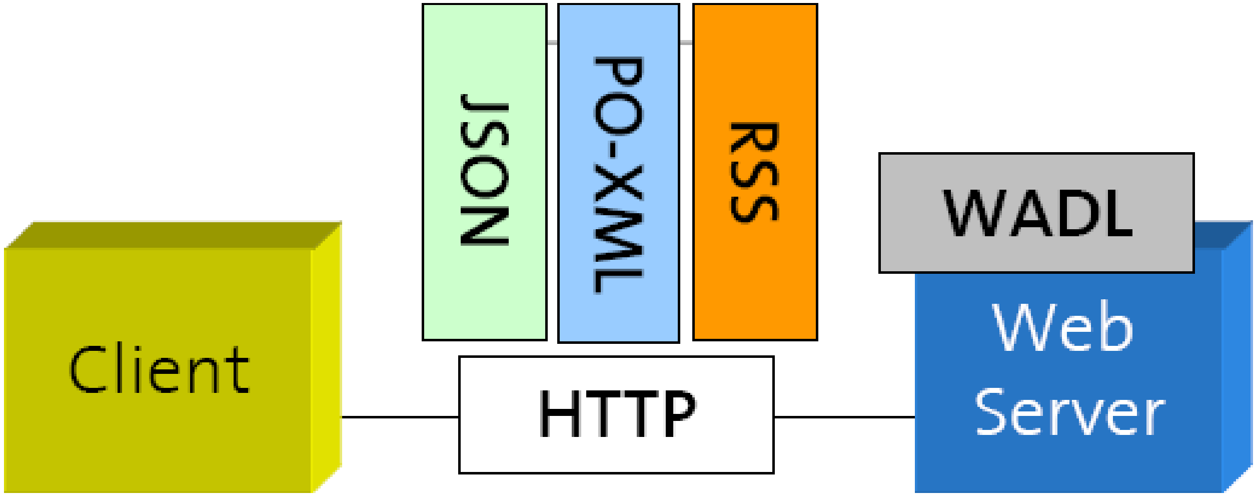
HTTP

Web Server

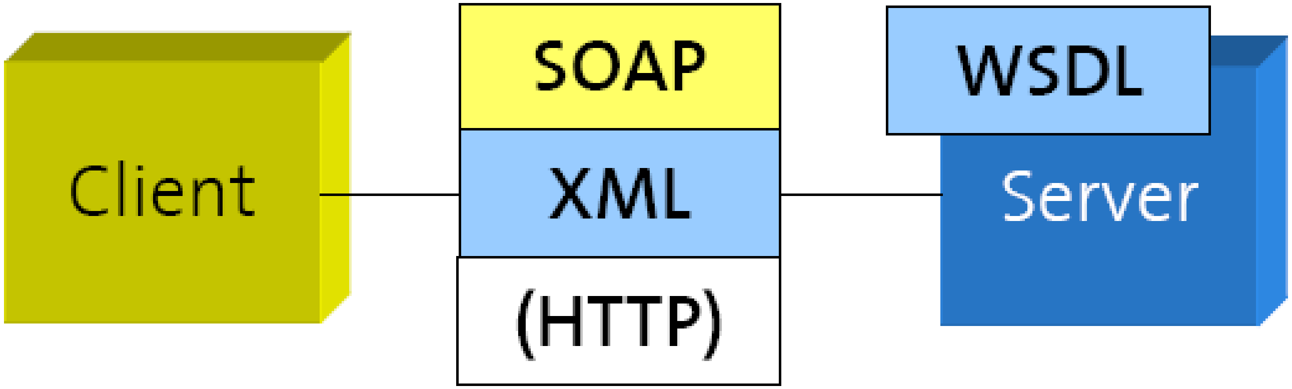
WS-* Web Services (2000)



RESTful Web Services (2007)



WS-* Web Services (2000)

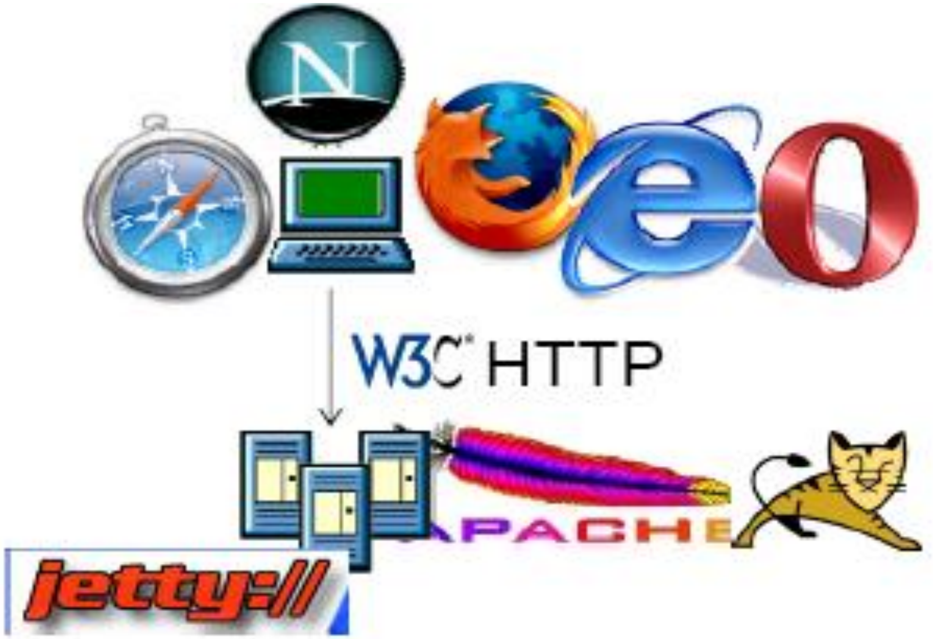


Where do Web services come from?

- 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

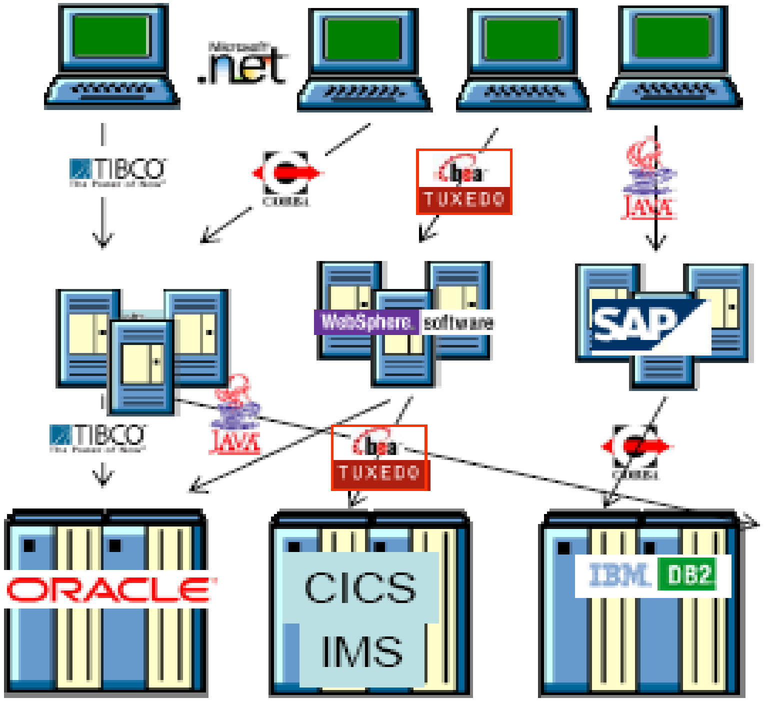
Dealing with Heterogeneity (1)

- Web Applications



Dealing with Heterogeneity (2)

- Enterprise Computing

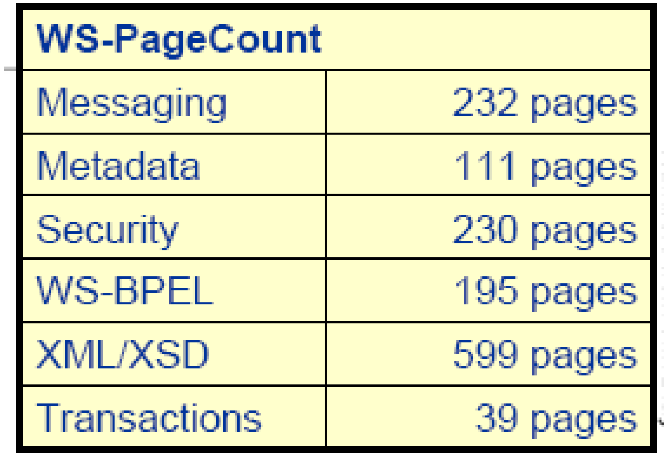


Big Web Services (1)

- 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)

Big Web Services (2)

- 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?



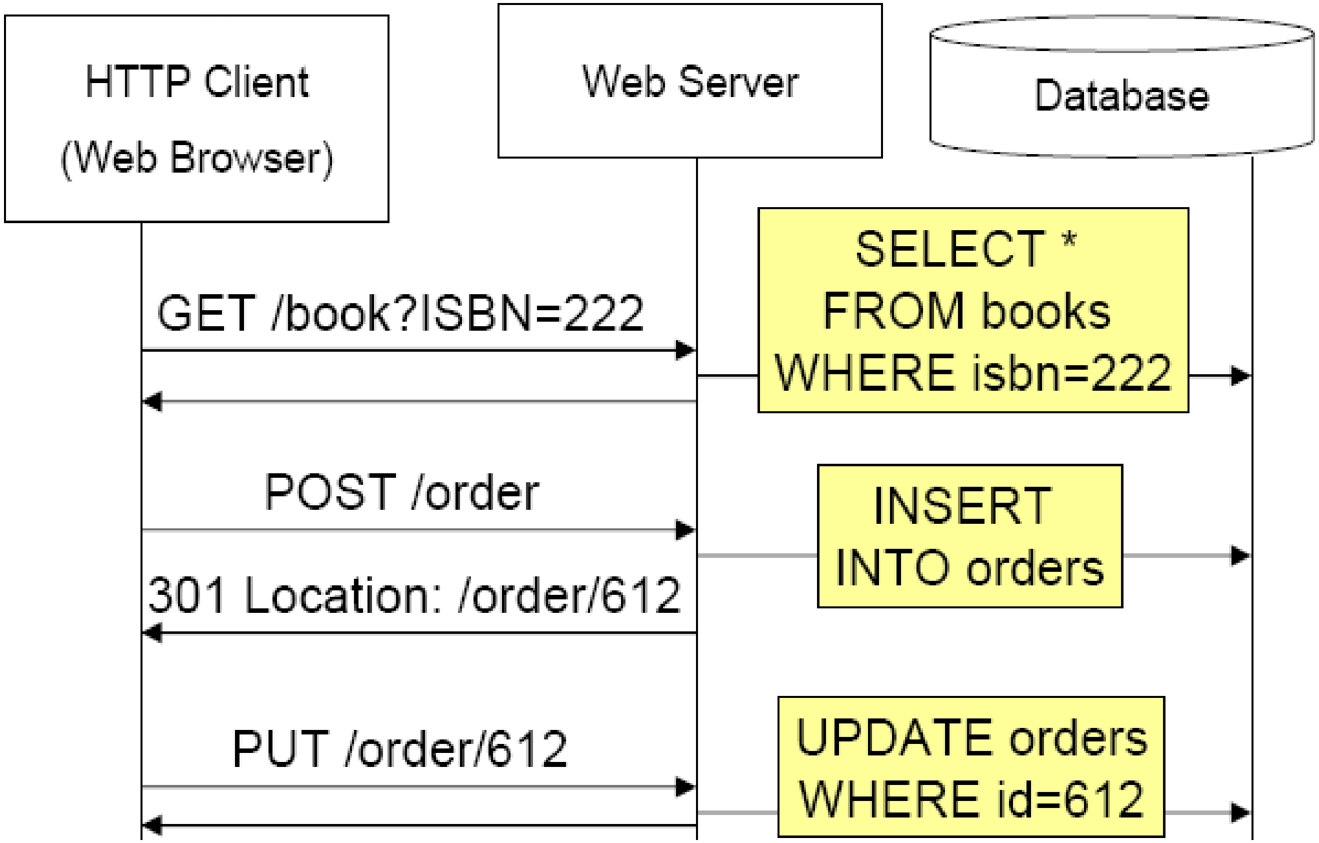
WS-PageCount	
Messaging	232 pages
Metadata	111 pages
Security	230 pages
WS-BPEL	195 pages
XML/XSD	599 pages
Transactions	39 pages

REpresentational State Transfer

- REST (REepresentational State Transfer) 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

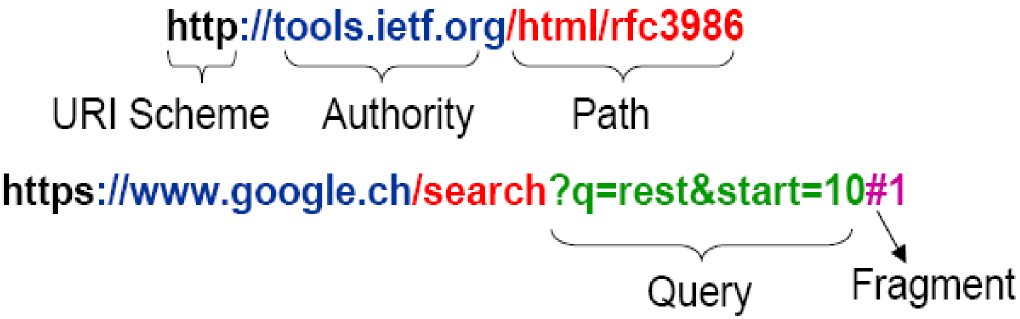


Uniform Interface Principle (CRUD Example)

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

Uniform Resource Identifier

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



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

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

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

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

High REST vs. Low REST

Best practices differ:

- High REST
  - Usage of “nice” URIs recommended
  - Full use of the 4 verbs: GET, POST, PUT, and DELETE
  - Responses using Plain Old XML
- Low REST
  - HTTP GET for idempotent requests, POST for everything else
  - Responses in any MIME Type (e.g., XHTML)

Resource Representation Formats: XML vs. JSON (1)

XML

- PO-XML
- SOAP (WS-*)
- RSS, ATOM

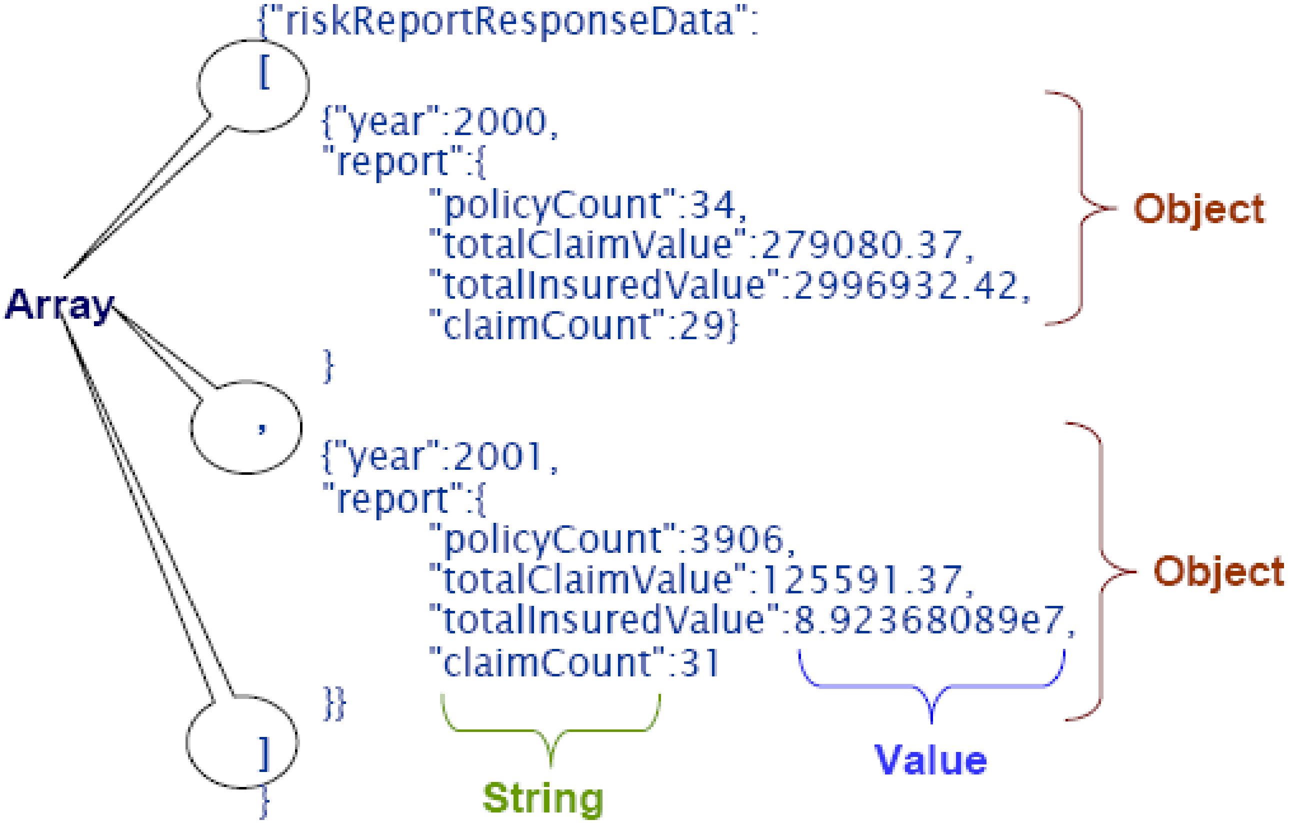
- Standard textual syntax for semi-structured data
- Many tools available:
  - XML Schema, DOM, SAX, XPath, XSLT, XQuery
- Everyone can parse it (not necessarily understand it)
- Slow and Verbose

Resource Representation Formats: XML vs. JSON (2)

JSON (JavaScript Object Notation)

- 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



```
{"riskReportResponseData":
[
    {"year":2000,
    "report":{
            "policyCount":34,
            "totalClaimValue":279080.37,
            "totalInsuredValue":2996932.42,
            "claimCount":29}
    }
,
    {"year":2001,
    "report":{
            "policyCount":3906,
            "totalClaimValue":125591.37,
            "totalInsuredValue":8.92368089e7,
            "claimCount":31
    }}
]
}
```

REST Strengths (1)

- 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

REST Strengths (2)

- 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

REST Weaknesses (1)

- 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

REST Weaknesses (2)

- 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)

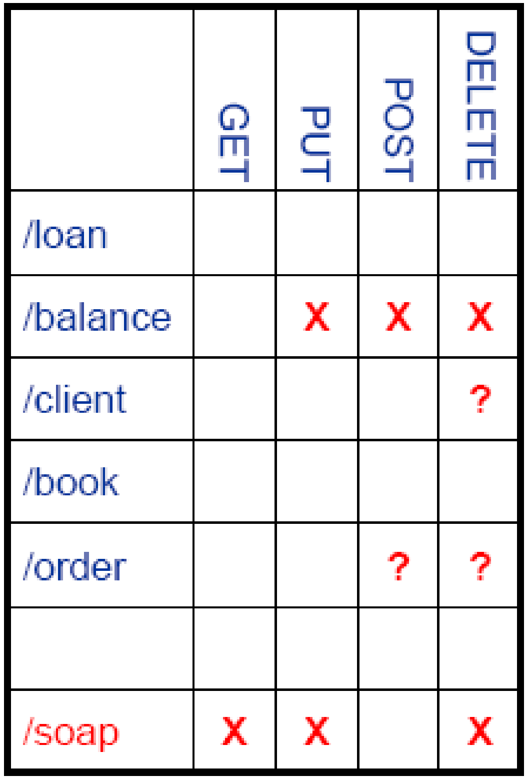
1. Identify resources to be exposed as services (e.g., yearly risk report, book catalog, purchase order, open bugs, polls and votes)
2. Define “nice” URLs to address them
3. Understand what it means to do a GET, POST, PUT, DELETE on a given resource URI
4. Design and document resource representations

RESTful Web Services Design Methodology (2)

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

RESTful Web Services Design Methodology (3)

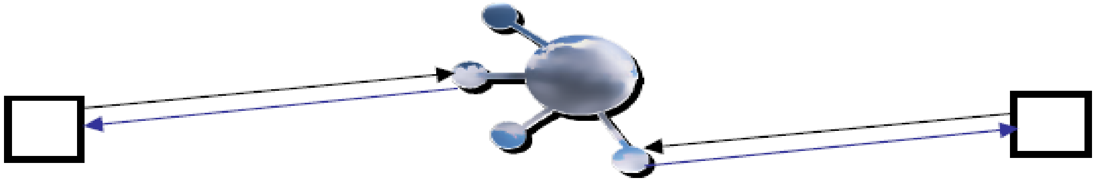
7. Test with a Web browser



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

Simple Doodle API Example (1)

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



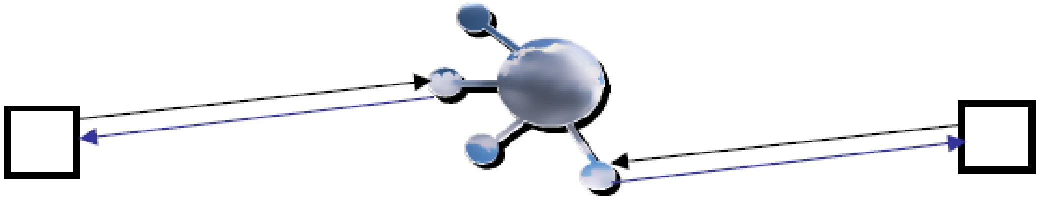
```
POST /poll
<options>A,B,C</options>

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 (2)

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

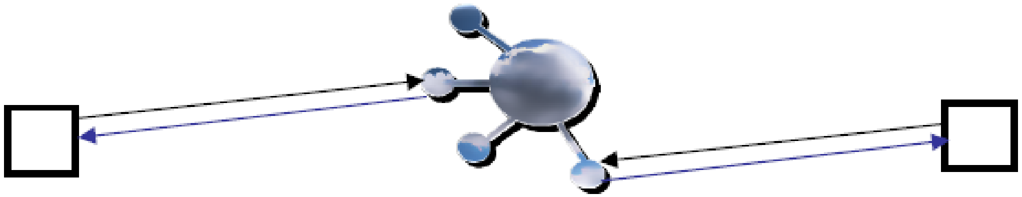


```
POST /poll/090331x/vote
<name>C. Pautasso</name>
<choice>B</choice>

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 (3)

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

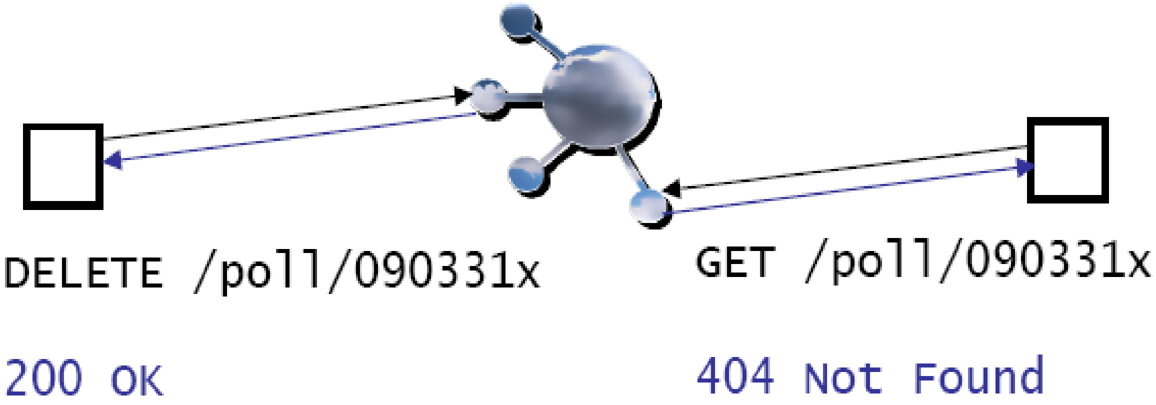


```
PUT /poll/090331x/vote/1
<name>C. Pautasso</name>
<choice>C</choice>

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 (4)

- Polls can be deleted once a decision has been made



DELETE /poll/090331x

200 OK

GET /poll/090331x

404 Not Found