



WS-* vs. RESTful Services

Cesare Pautasso

Faculty of Informatics, USI Lugano, Switzerland

c.pautasso@ieee.org

<http://www.pautasso.info>

<http://twitter.com/pautasso>

30.6.2010

ECOWS'10

European Conference on Web Services
December 1-3 2010, Ayia Napa, Cyprus

Recent technology trends in Web Services indicate that a solution eliminating the perceived complexity of the WS-* standard technology stack may be in sight: advocates of REpresentational State Transfer (REST) have come to believe that their ideas explaining why the World Wide Web works are just as applicable to solve enterprise application integration problems and to radically simplify the plumbing required to build service-oriented architectures. In this tutorial we take a scientific look at the WS-* vs. REST debate by presenting a technical comparison based on architectural principles and decisions. We show that the two approaches differ in the number of architectural decisions that must be made and in the number of available alternatives. This discrepancy between freedom-from-choice and freedom-of-choice quantitatively explains the perceived complexity difference. We also show that there are significant differences in the consequences of certain decisions in terms of resulting development and maintenance costs. Our comparison helps technical decision makers to assess the two integration technologies more objectively and select the one that best fits their needs: REST is well suited for basic, ad hoc integration scenarios à la mashup, WS-* is more mature and addresses advanced quality of service requirements commonly found in enterprise computing.

- Assistant Professor at the [Faculty of Informatics, University of Lugano](#), Switzerland (since Sept 2007)
- Research Projects:
 - SOSOA – Self-Organizing Service Oriented Architectures
 - CLAVOS – Continuous Lifelong Analysis and Verification of Open Services
 - BPEL for REST
- Researcher at [IBM Zurich Research Lab](#) (2007)
- Post-Doc at [ETH Zürich](#)
 - Software:
[JOpera: Process Support for more than Web services](#)
<http://www.jopera.org/>
- Ph.D. at [ETH Zürich](#), Switzerland (2004)
- Laurea Politecnico di Milano (2000)
- Representations:
<http://www.pautasso.info/> (Web)
<http://twitter.com/pautasso/> (Twitter Feed)

WS-* Standards Stack

Interoperability Issues

- Basic Profile
- Attachments Profile
- Single SOAP Binding Profile
- Basic Security Profile
- WS-Tokens Profile
- SAML Token Profile
- Confidentiality Check Attachment Mechanism (CCMAM)
- Reliable Attachments Messaging Profile (RAMMP)

Business Process Specifications

- Business Process Description Language for Web Services
- Web Services Choreography Model Extension
- Web Services Choreography Description Language

Management Specifications

- WS-Service Management Framework
- WS-Events
- WS-Management
- Management Using Web Services
- Management Using SOAP

Metadata Specifications

- WS-Policy
- WS-PolicyAssertions
- WS-PolicyFramework
- WS-Discovery
- WS-Discovery Extensions
- Historical Description Discovery and Integration
- Web Service Description Language
- Web Service Description Language

Reliability Specifications

- WS-ReliableMessaging
- WS-Reliability

Security Specifications

- WS-Security
- WS-Security SOAP Message Security
- WS-Security Binary Security
- WS-Security SAML Token Profile
- WS-Security Assertions
- WS-Security Assertions
- WS-Security Assertions

Transaction Specifications

- WS-Atomic Transaction
- WS-Atomic Transaction
- WS-Coordination
- WS-Coordination Application Framework
- WS-Coordination
- WS-Coordination
- WS-Transaction Management

Resource Specifications

- Web Services Resource Framework
- WS-BaseFaults
- WS-SecurityErrors
- WS-ResourceProperties
- WS-ResourceLifecycle
- WS-Transfer
- Resource Representation SOAP Header Block (RRHSB)

Messaging Specifications

- WS-Notification
- WS-Eventing
- WS-Events
- WS-Addressing
- WS-BaseFaults
- WS-Eventing
- WS-Eventing

SOAP

- SOAP 1.1
- SOAP 1.2
- SOAP 1.2

XML Specifications

- XML
- XML
- Namespaces in XML
- XML Information Set
- XML Information Set
- XML Schema
- XML Schema
- XML Schema

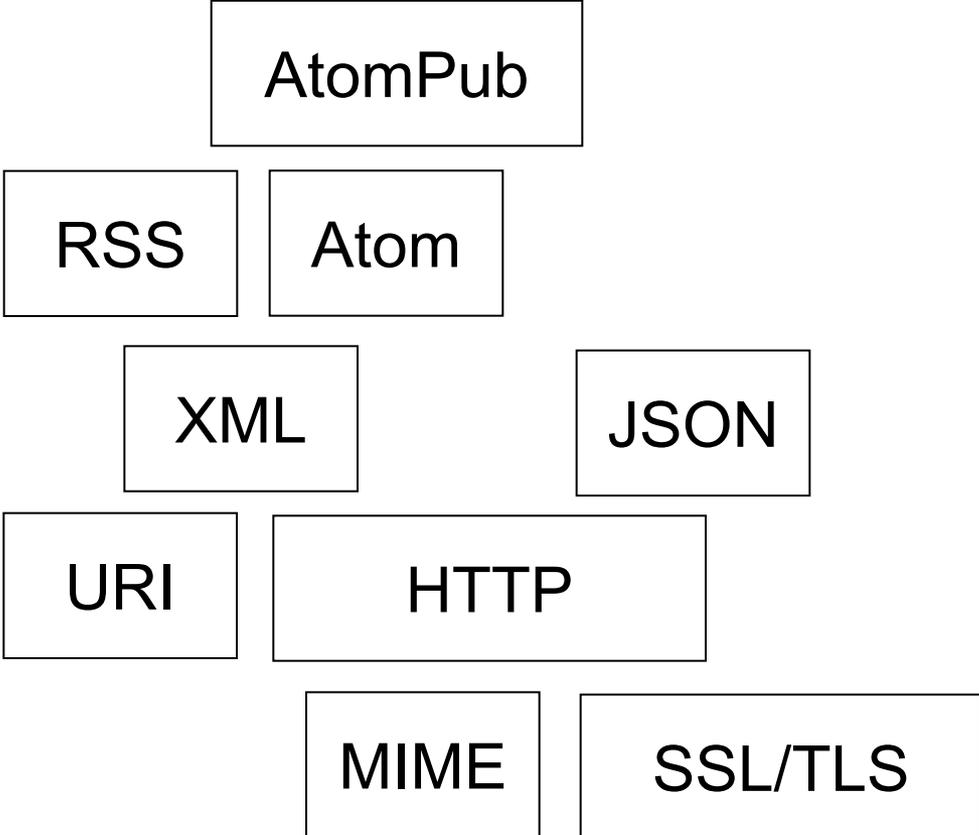
Standards Bodies

- OASIS
- W3C
- ISO



innoQ Deutschland GmbH
Hakenstraße 17
D-40880 Ratingen
Telefon +49 (0) 21 02 - 77 92 - 100
Telefax +49 (0) 21 02 - 77 92 - 01
info@innoq.com | www.innoq.com

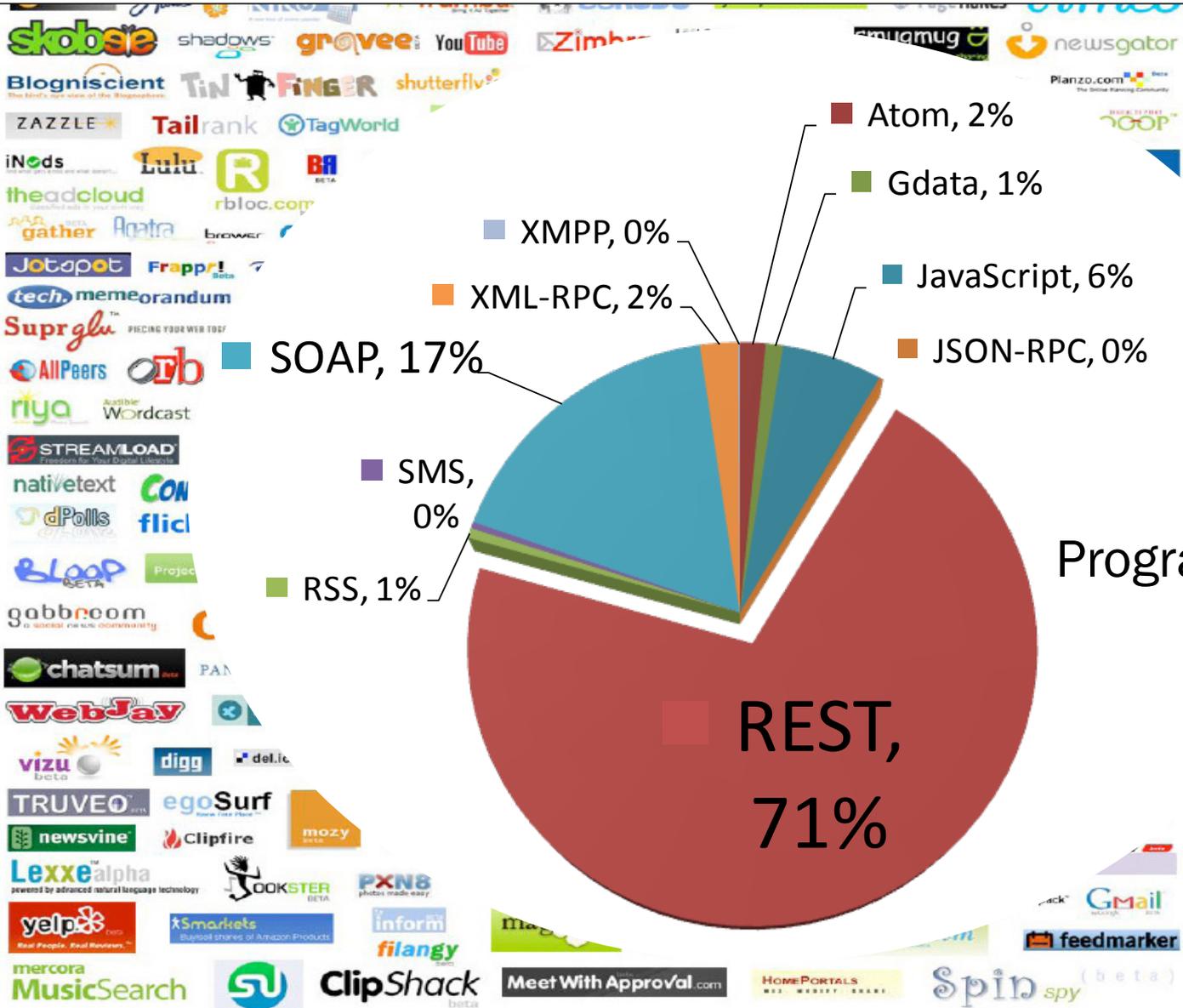
innoQ Schweiz GmbH
Gewerbestrasse 11
CH-4300 Cham
Telefon +41 (0) 41 - 743 00 10
Telefax +41 (0) 41 - 743 00 10
info@innoq.com | www.innoq.com



Is REST really used?



Is REST really used?

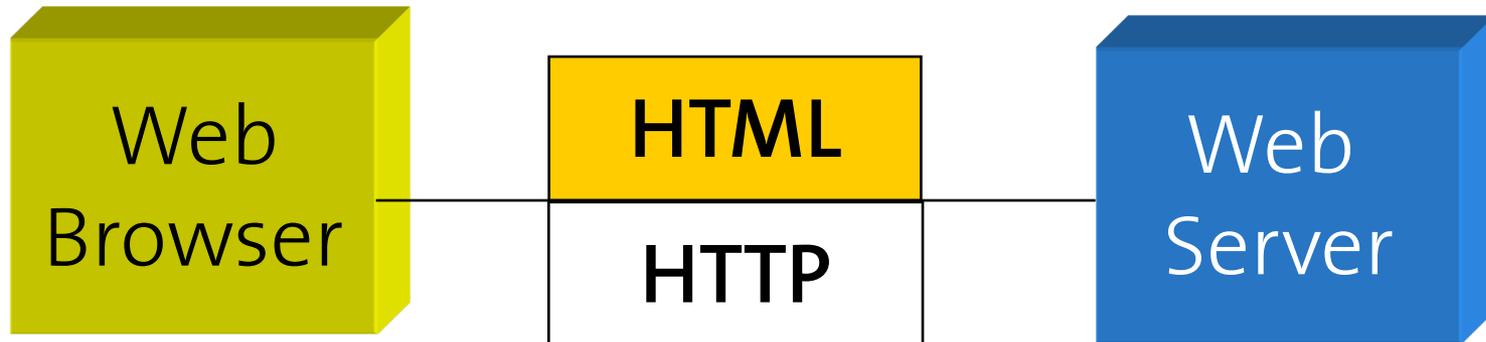


2042 APIs

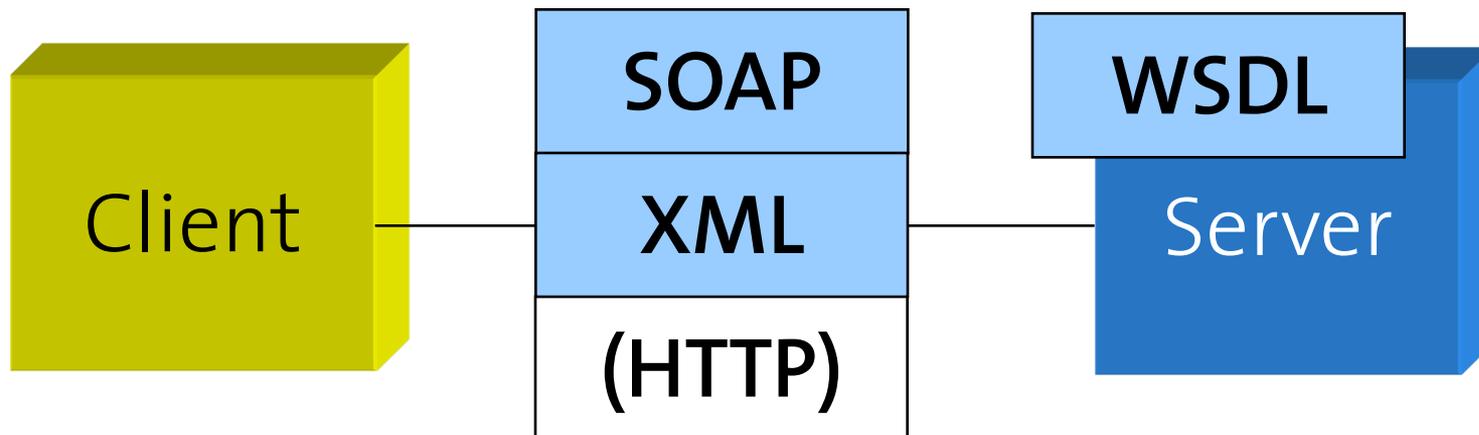
ProgrammableWeb.com

30.6.2010

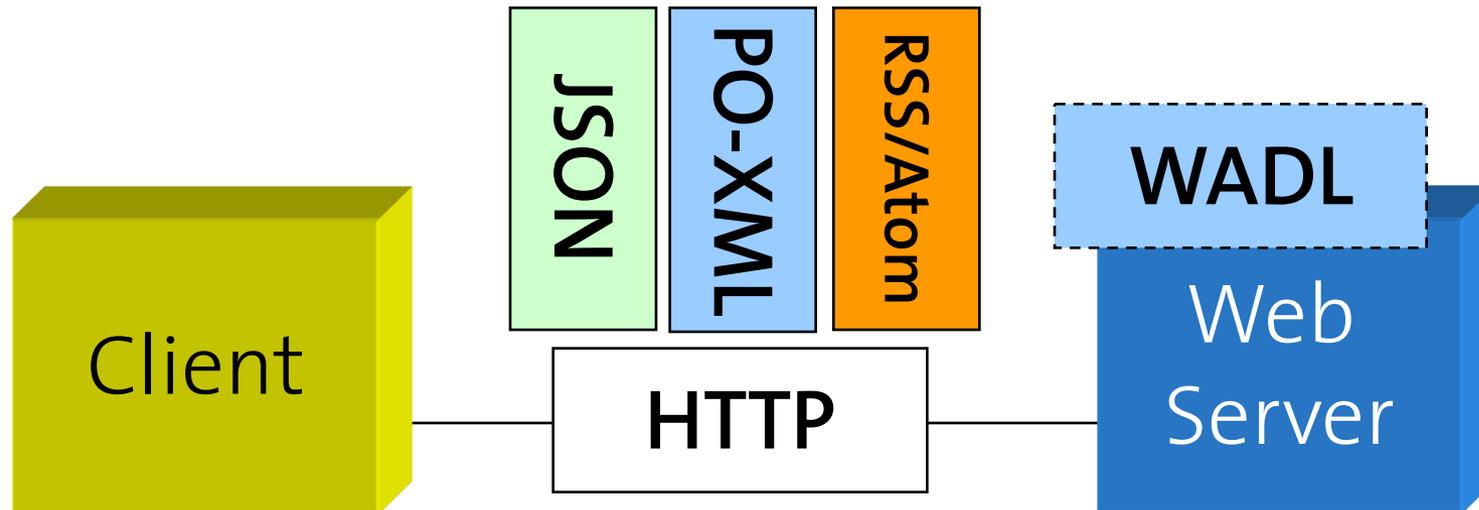
Web Sites (1992)



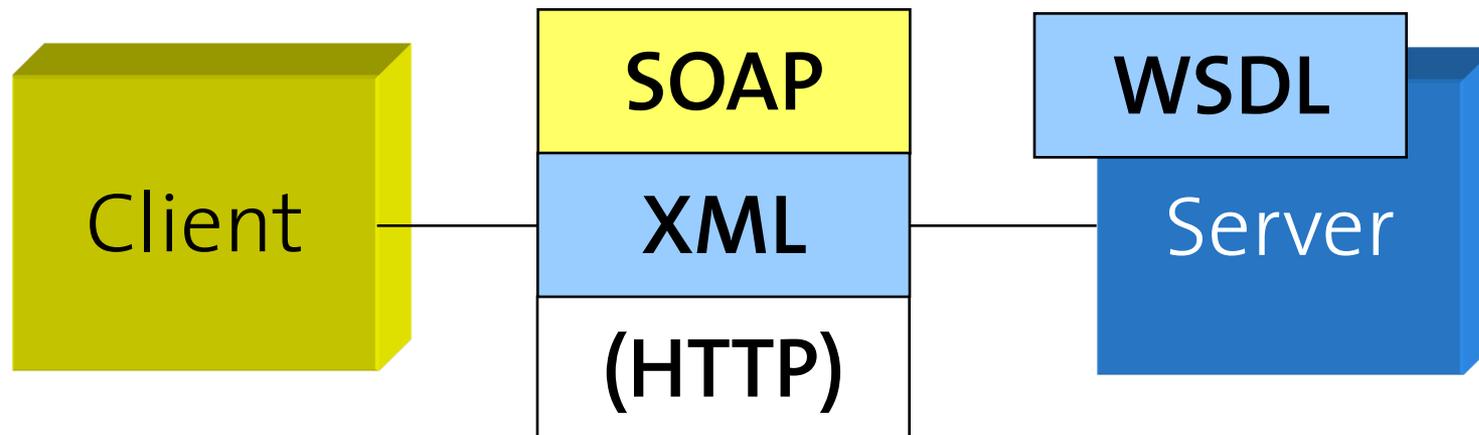
WS-* Web Services (2000)



RESTful Web Services (2007)



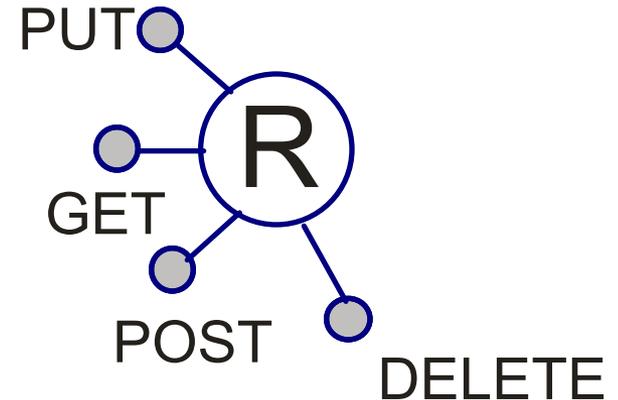
WS-* Web Services (2000)



1. Introduction to RESTful Web Services

2. Comparing REST and WS-*

- Web Services expose their data and functionality through **resources** identified by **URI**
- **Uniform Interface Principle**: Clients interact with resources through a fixed set of verbs.
Example HTTP:
GET (read), POST (create), PUT (update), DELETE
- **Multiple representations** for the same resource
- **Hyperlinks** model resource relationships and valid state transitions for dynamic protocol description and discovery



- 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`

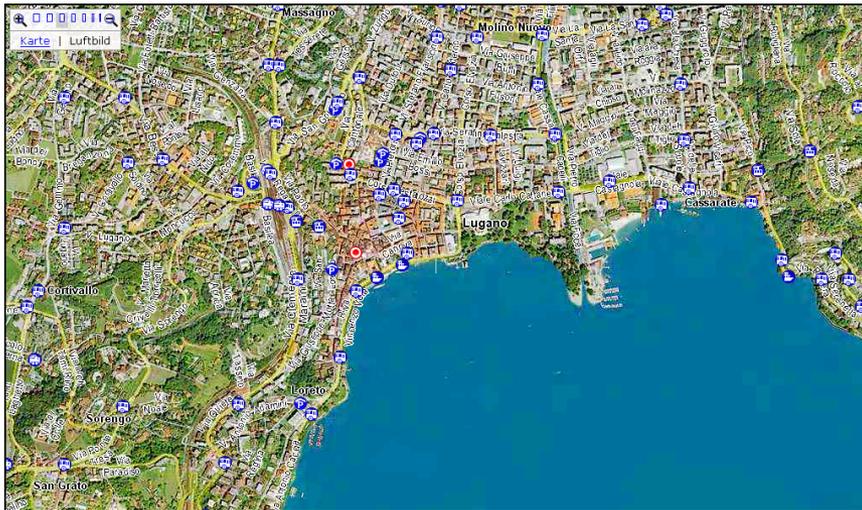
Query Fragment

- REST does **not** advocate the use of “nice” URIs
- In most HTTP stacks URIs cannot have arbitrary length (4Kb)
- #Fragments are not even sent to the server

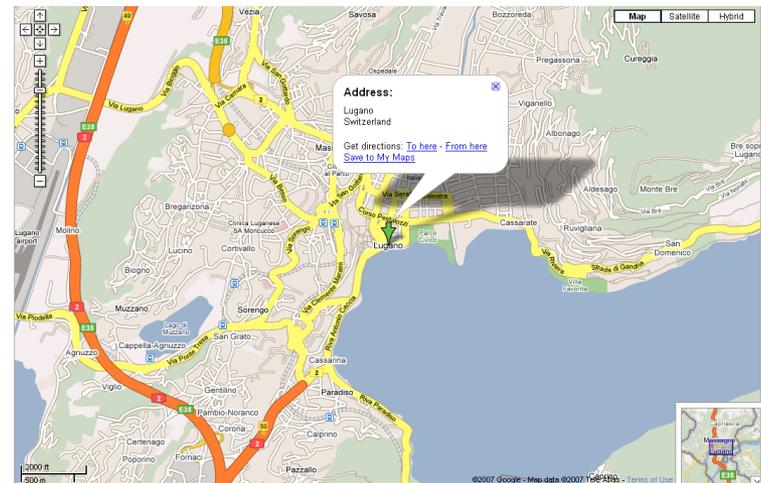
What is a “nice” URI?

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
- If possible follow a “positional” parameter-passing scheme for algorithmic resource query strings (instead of the key=value&p=v encoding)
- Some use URI postfixes 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

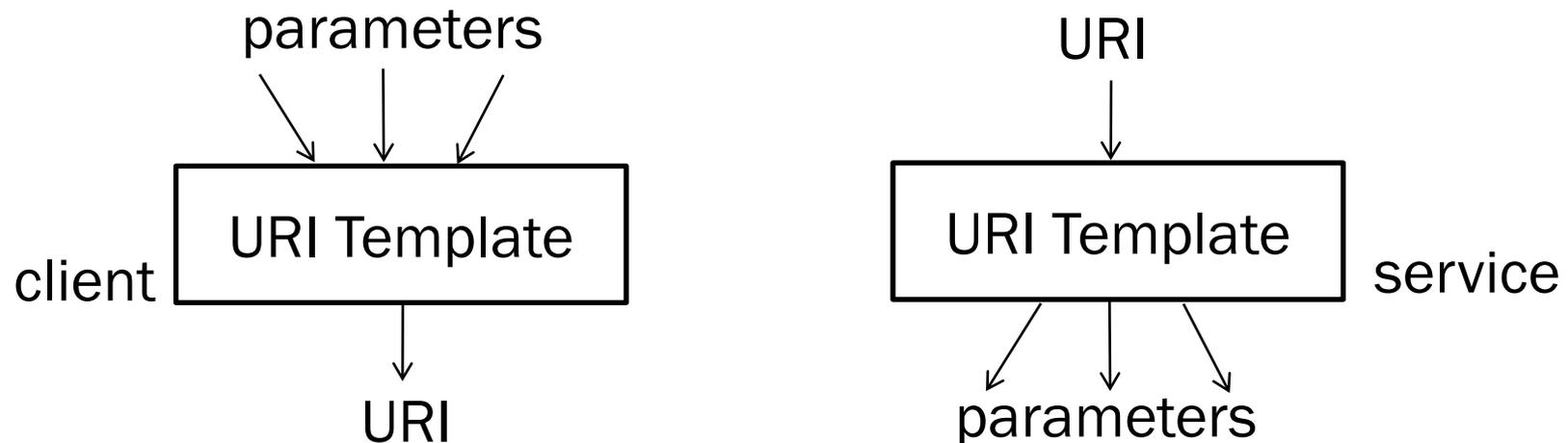
- **Note:** REST URIs are opaque identifiers that are meant to be discovered by following hyperlinks and *not constructed by the client*

- *This may break the abstraction*

- **Warning:** URI Templates introduce coupling between client and server

URI Templates

- URI Templates specify how to construct and parse parametric URIs.
 - On the service they are often used to configure “routing rules”
 - On the client they are used to instantiate URIs from local parameters



- Do not hardcode URIs in the client!
- Do not hardcode URI templates in the client!
- Reduce coupling by fetching the URI template from the service dynamically and fill them out on the client

URI Template Examples

- From <http://bitworking.org/projects/URI-Templates/>

- Template:

`http://www.myservice.com/order/{oid}/item/{iid}`

- Example URI:

`http://www.myservice.com/order/XYZ/item/12345`

- Template:

`http://www.google.com/search?{-join | & | q,num}`

- Example URI:

`http://www.google.com/search?q=REST&num=10`

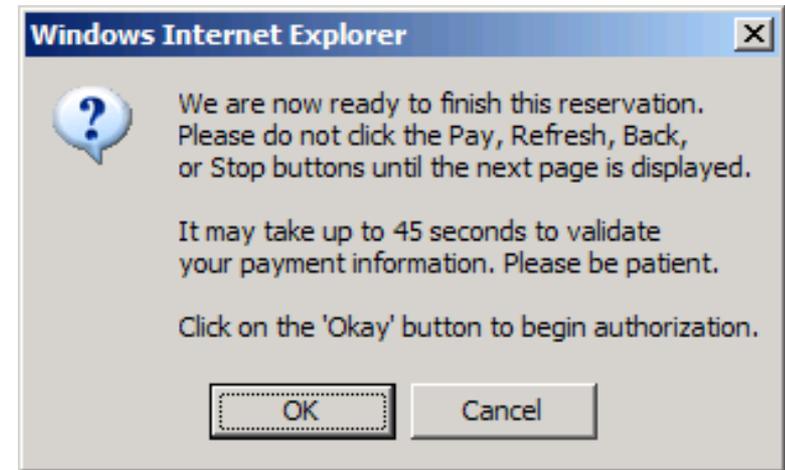
HTTP		SAFE	IDEM POTENT
POST	Create a sub resource	NO	NO
GET	Retrieve the <i>current state</i> of the resource	YES	YES
PUT	Initialize or update the state of a resource at the given URI	NO	YES
DELETE	Clear a resource, after the URI is no longer valid	NO	YES

POST vs. GET

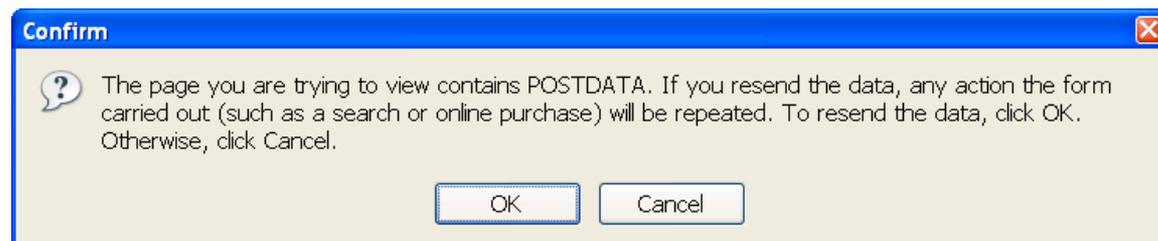
- GET is a **read-only** operation. It can be repeated without affecting the state of the resource (idempotent) and can be cached.

Note: this does not mean that the same representation will be returned every time.

- 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**

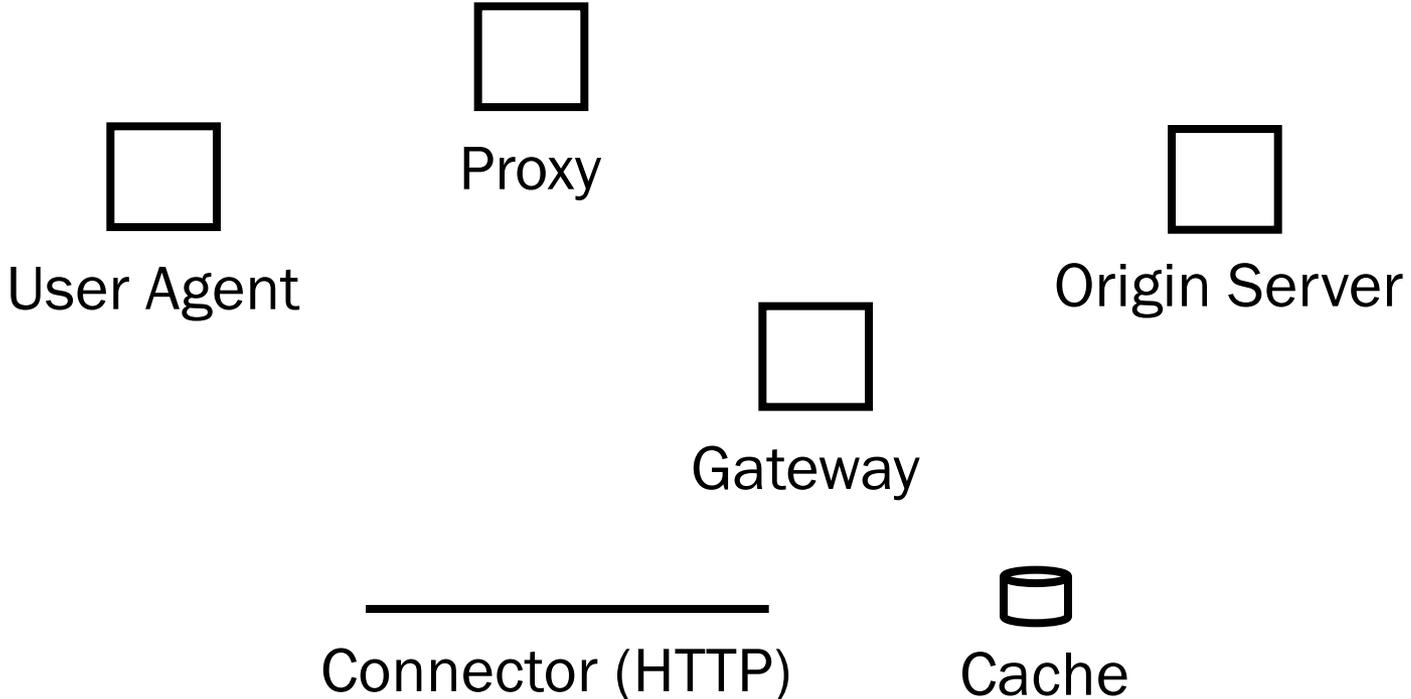
Location: /resource/{id}

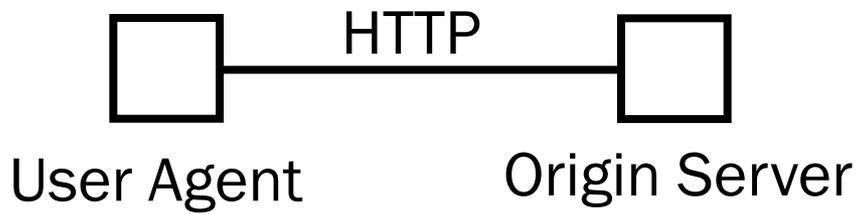
Solution 2: let the server compute the unique id

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

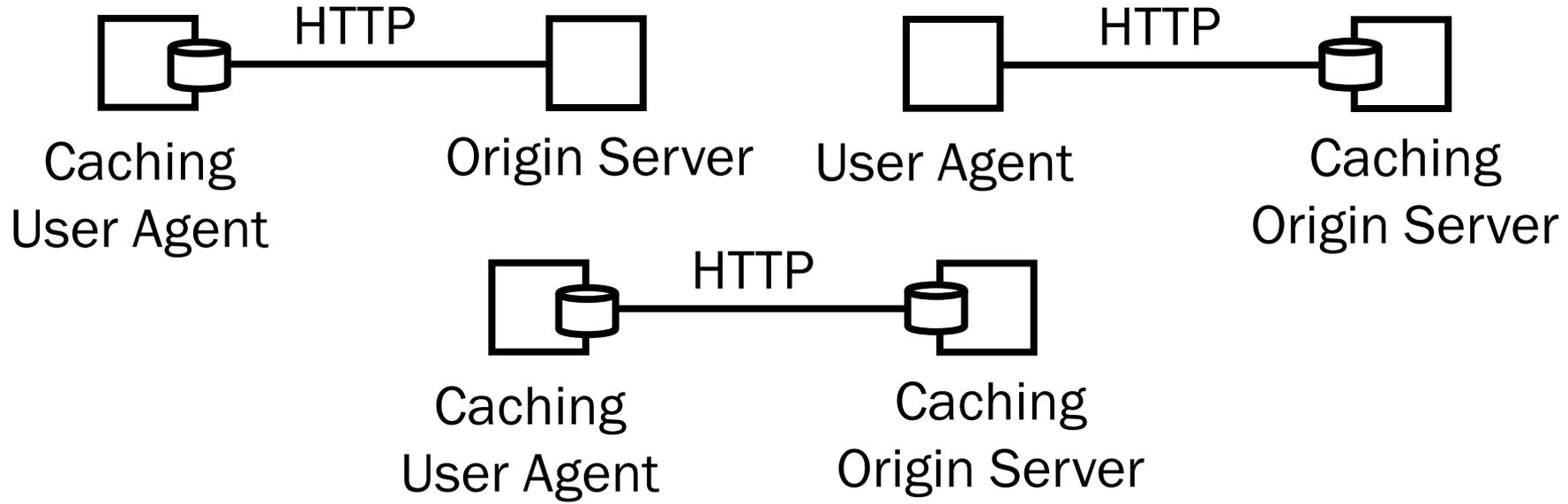
REST Architectural Elements

Client/Server Layered Stateless Communication Cache



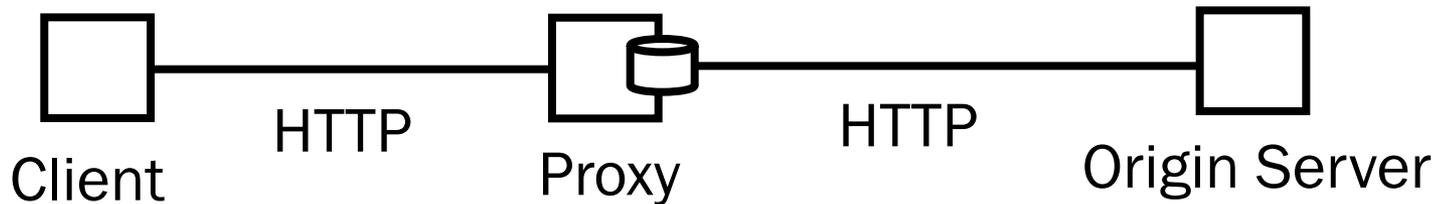


Adding Caching

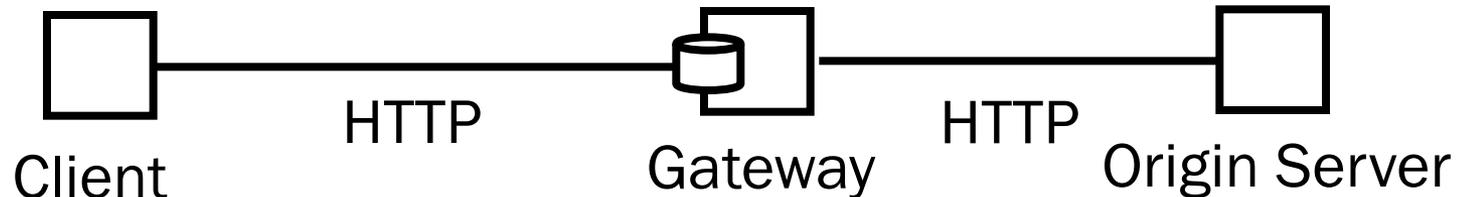


Proxy or Gateway?

Intermediaries forward (and may translate) requests and responses



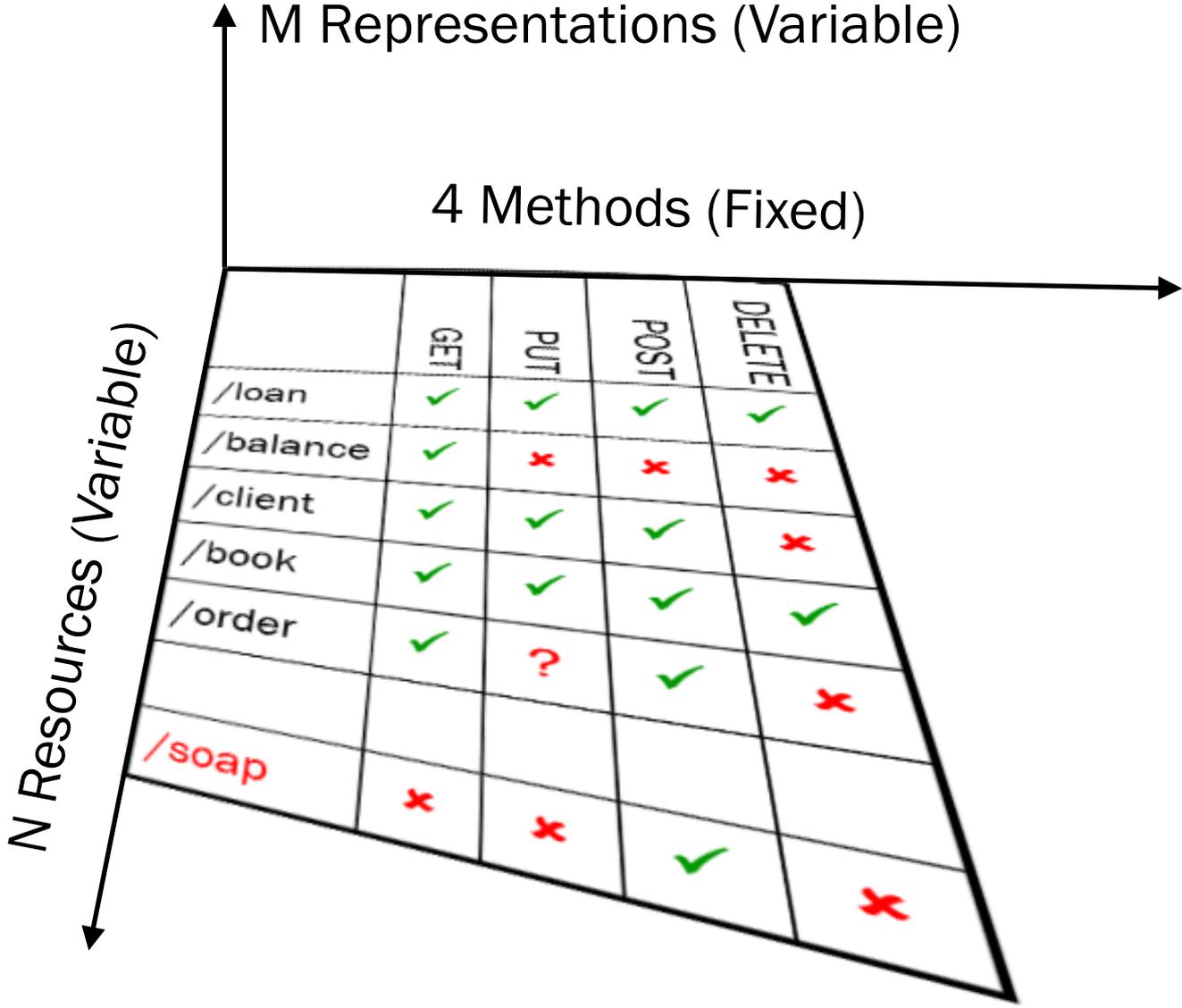
A proxy is chosen by the Client (for caching, or access control)



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

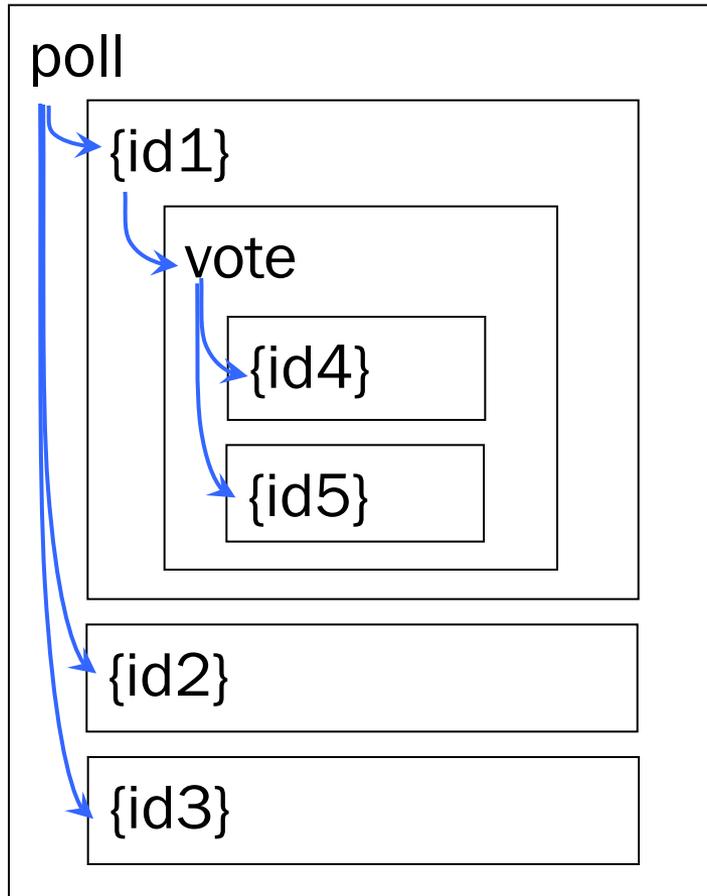
1. Identify resources to be exposed as services (e.g., yearly risk report, book catalog, purchase order, open bugs, polls and votes)
2. Model relationships (e.g., containment, reference, state transitions) between resources with hyperlinks that can be followed to get more details (or perform state transitions)
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	✗	✗	✓	✗



Simple Doodle API Example

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

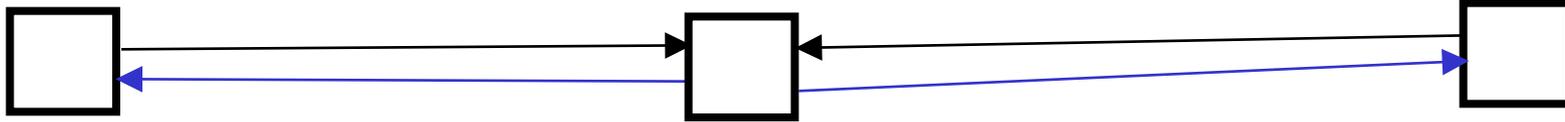


	GET	PUT	POST	DELETE
/poll	✓	✗	✓	✗
/poll/{id}	✓	✓	✗	✓
/poll/{id}/vote	✓	✗	✓	✗
/poll/{id}/vote/{id}	✓	✓	✗	?

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

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

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



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

201 Created
Location: /poll/090331x

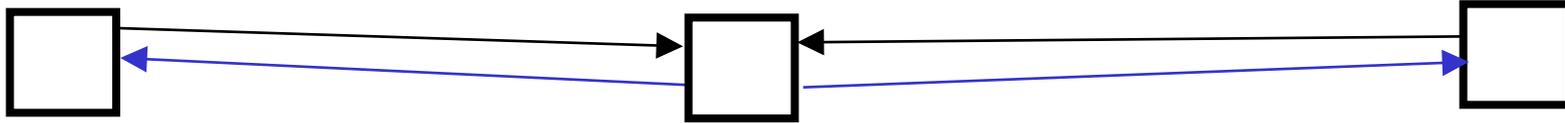
GET /poll/090331x

200 OK
<options>A,B,C</options>
<votes href="/vote"/>

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

- 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

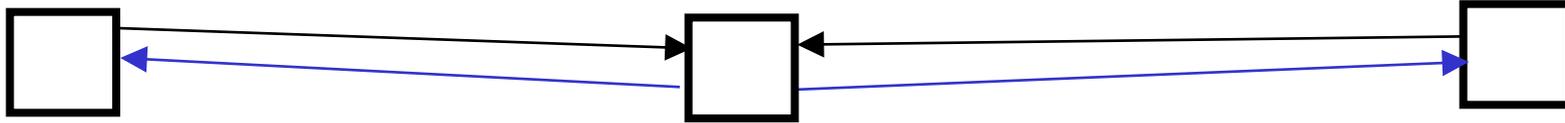
GET /poll/090331x

200 OK

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

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

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



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

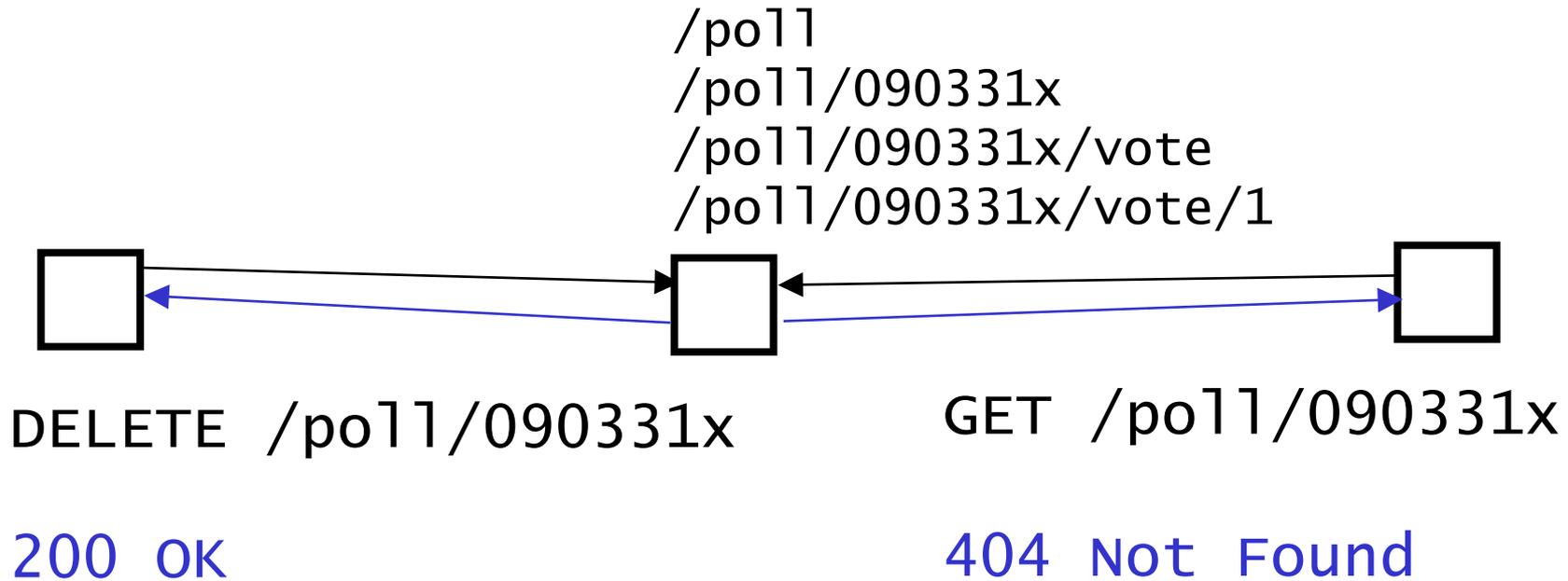
200 OK

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

200 OK

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

- Polls can be deleted once a decision has been made



Real Doodle Demo

- Info on the real Doodle API:

<http://doodle.com/xsd1/RESTfulDoodle.pdf>

- Lightweight demo with Poster Firefox Extension:

<http://addons.mozilla.org/en-US/firefox/addon/2691>

The screenshot shows a Mozilla Firefox browser window with the following elements:

- Browser Title:** Doodle: What to do in San Sebastian? - Mozilla Firefox
- Address Bar:** <http://doodle-test.com/3b5swbzh35ych73>
- Tab:** Doodle: What to do ...
- Main Content:**
 - Poll:** What to do in San Sebastian?
 - CP has created this poll.
 - "ICWE 2009 demo"
 - Options Table:**

Go to the beach	Walk in the old town	Visit the Castle	Take the cable car up to the lighthouse tower	Dive in the ocean	Visit the Acquarium	Take a boat to the island	Go to the spa	Attend a ICWE Workshop	Attend the ICWE REST/SOA Tutorial
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
0	0	0	0	0	0	0	0	0	0
 - Buttons:** Save
 - Functions:** Edit an entry, Delete an entry, Add a comment, Calendar export, File export, Print, Subscribe to this poll, Embed this poll
 - Comments:** Add a comment >>
- Poster Extension (Right Panel):**
 - Request:** Select a file or enter content to POST or PUT to a URL and then specify the mime type you'd like or just use the GET, HEAD, or DELETE methods on a URL.
 - URL:** <http://doodle-test.com/api1WithoutAccessControl/pc>
 - File:** Browse...
 - Content Type:** text/xml
 - User Auth:** Google Login
 - Settings:** Save, Import, Store
 - Actions:** PUT (dropdown), GO
 - Headers:** Headers (dropdown), GO
 - Content to Send:**

```
<?xml version="1.0" encoding="UTF-8"?><poll
xmlns="http://doodle.com/xsd1"><type>TEXT</type><extensions
/><hidden>false</hidden><levels>2</levels><state>OPEN</state>
<title>What to do in San Sebastian?</title><description>ICWE
2009 demo</description><initiator<name>CP</name></initiator>
<options><option>Go to the beach</option><option>Walk in the
old town</option><option>Visit the Castle</option><option>Take
the cable car up to the lighthouse tower</option><option>Dive in
the ocean</option><option>Visit the Acquarium</option>
<option>Take a boat to the island</option><option>Go to the
spa</option><option>Attend a ICWE Workshop</option>
<option>Attend the ICWE REST/SOA Tutorial</option></options>
</poll>
```

1. Create Poll

POST <http://doodle-test.com/api1WithoutAccessControl/polls/>
Content-Type: application/xml

```
<?xml version="1.0" encoding="UTF-8"?><poll
  xmlns="http://doodle.com/xsd1"><type>TEXT</type><extensions
  rowConstraint="1"/><hidden>>false</hidden><writeOnce>>false</writeOnce
  ><requireAddress>>false</requireAddress><requireEMail>>false</requireEM
  ail><requirePhone>>false</requirePhone><byInvitationOnly>>false</byInvitat
  ionOnly><levels>2</levels><state>OPEN</state><title>How is the tutorial
  going?</title><description></description><initiator><name>Cesare
  Pautasso</name><userId></userId><eMailAddress>test@jopera.org</eM
  ailAddress></initiator><options><option>too fast</option><option>right
  speed</option><option>too
  slow</option></options><participants></participants><comments></com
  ments></poll>
```

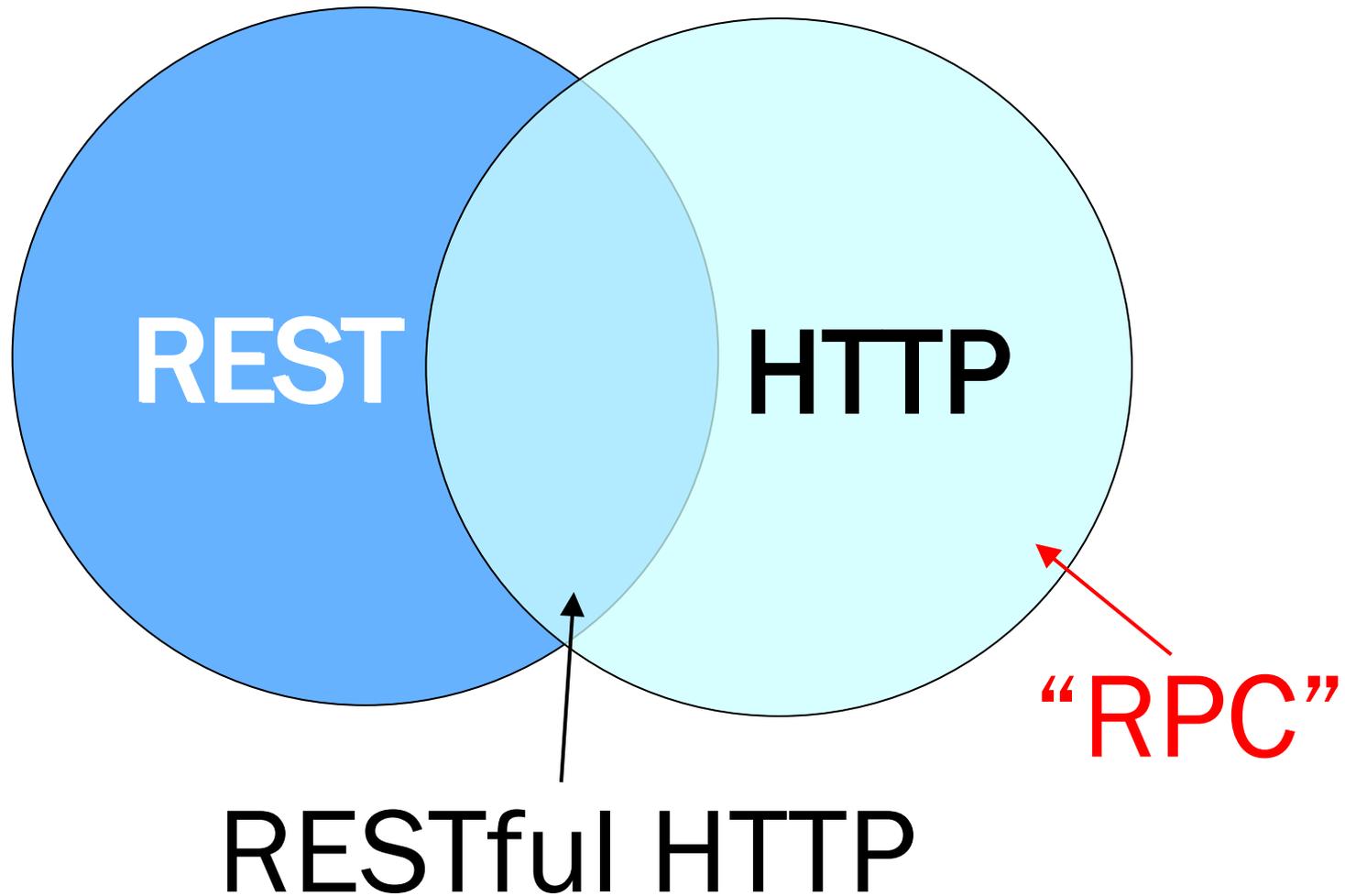
Content-Location: {id}

GET <http://doodle-test.com/api1WithoutAccessControl/polls/{id}>

2. Vote

POST http://doodle-test.com/api1WithoutAccessControl/polls/{id}/participants
Content-Type: application/xml

```
<?xml version="1.0" encoding="UTF-8"?>  
  <participant xmlns="http://doodle.com/xsd1"><name>Cesare  
Pautasso</name><preferences><option>0</option><option>1</option><  
option>0</option></preferences></participant>
```



0. HTTP as an RPC Protocol
(Tunnel POST+POX or POST+JSON)
 - I. Multiple Resource URIs
(Fine-Grained Global Addressability)
 - II. Uniform HTTP Verbs
(Contract Standardization)
 - III. Hypermedia
(Protocol Discoverability)
- A REST API needs to include levels I, II, III
 - Degrees of RESTfulness?

HTTP as a tunnel

- Tunnel through one HTTP Method

GET /api?method=addCustomer&name=wilde

GET /api?method=deleteCustomer&id=42

GET /api?method=getCustomerName&id=42

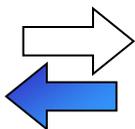
GET /api?method=findCustomers&name=wilde*

- 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?

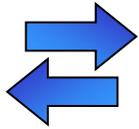


- Limitation: Requests can only send up to approx. 4KB of data (414 Request-URI Too Long)

HTTP as a tunnel

- 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)

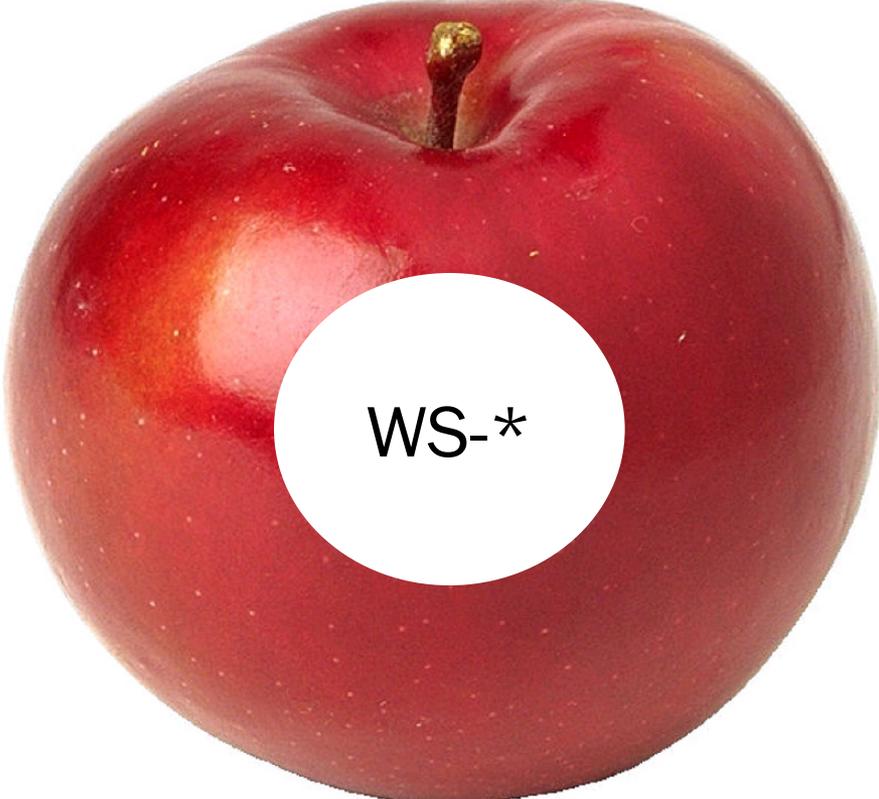
POST /service/endpoint

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

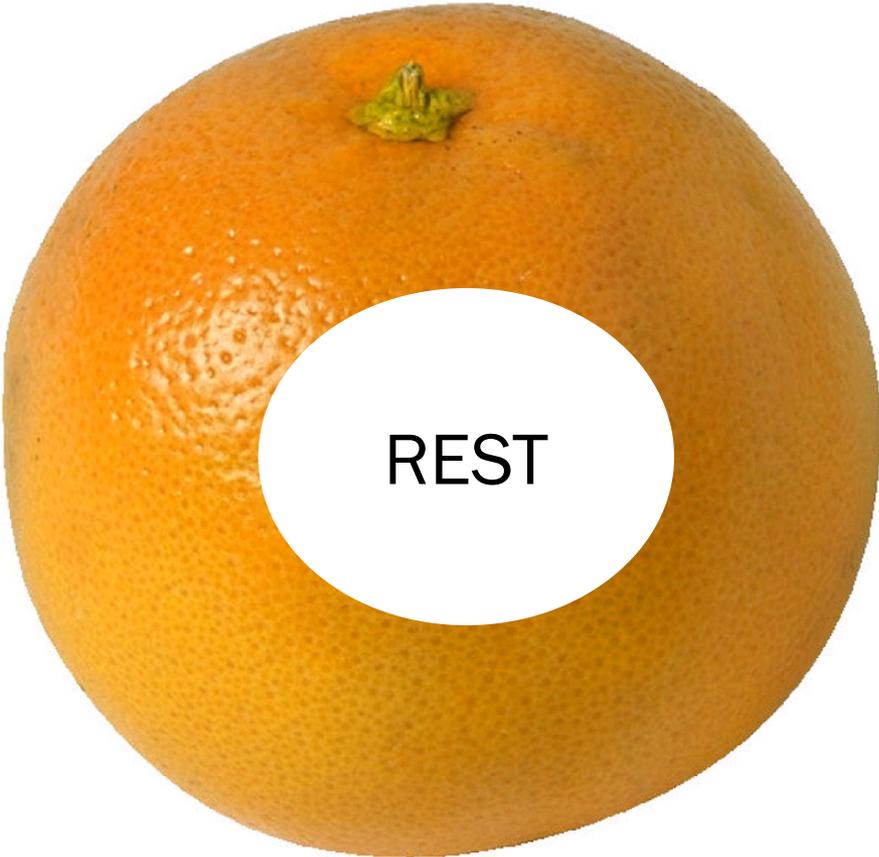


1. Introduction
to RESTful Web Services
2. Comparing REST and WS-*

Can we really compare?



WS-*



REST

Can we really compare?



WS-*

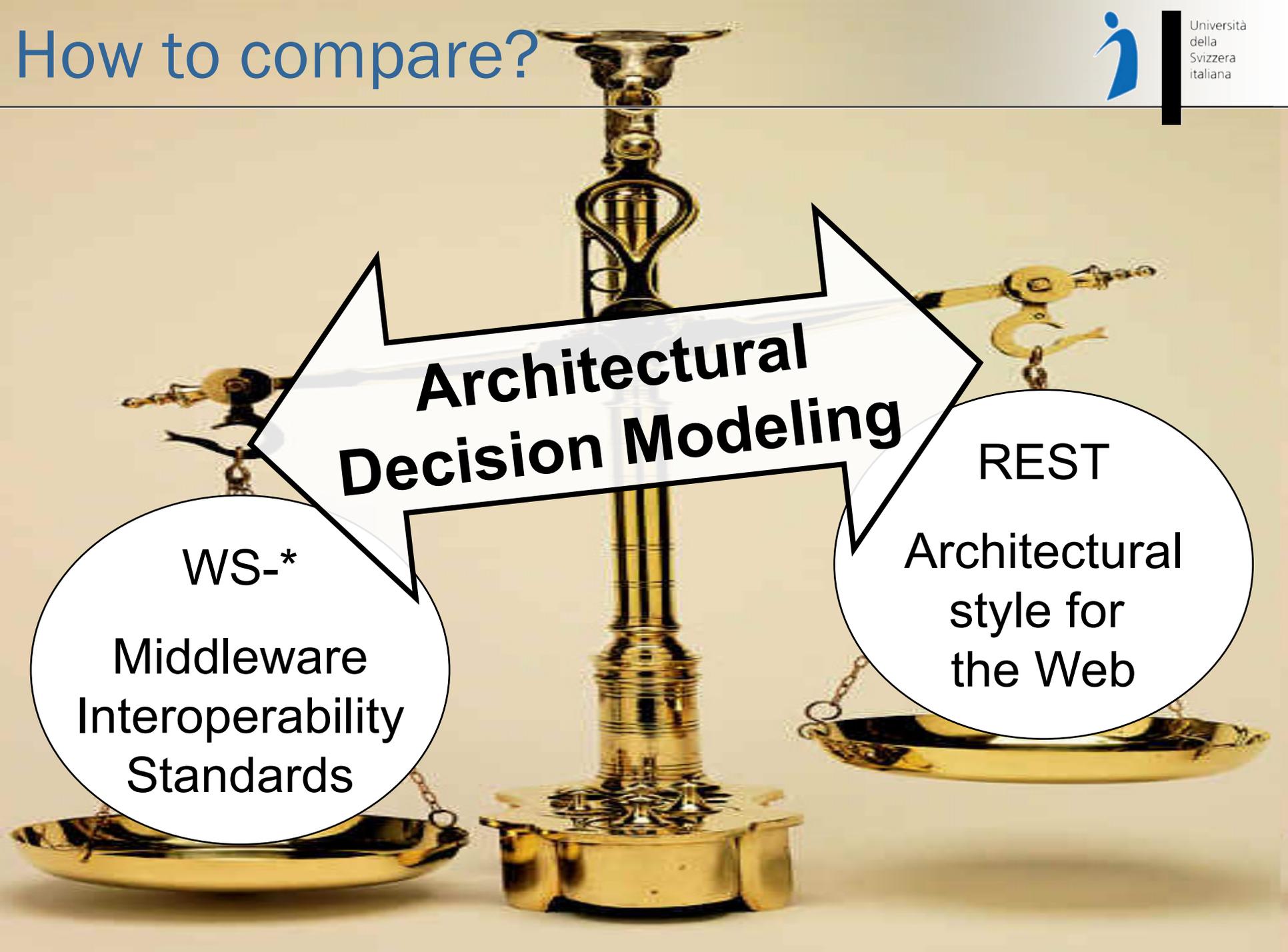
Middleware
Interoperability
Standards



REST

Architectural
style for
the Web

How to compare?



Architectural Decision Modeling

WS-*

Middleware
Interoperability
Standards

REST

Architectural
style for
the Web

- Architectural decisions capture the main design issues and the rationale behind a chosen technical solution
- **The choice between REST vs. WS-* is an important architectural decision for Web service design**
- **Architectural decisions affect one another**

Architectural Decision: **Programming Language**

Architecture Alternatives:

1. **Java**
2. **C#**
3. **C++**
4. **C**
5. **Eiffel**
6. **Ruby**
7. ...

Rationale

Decision Space Overview

Architectural Decision and AAs	REST	WS-*
Integration Style	1 AA	2 AAs
Shared Database		
File Transfer		
Remote Procedure Call	✓	✓
Messaging		✓
Contract Design	1 AA	2 AAs
Contract-first		✓
Contract-last		✓
Contract-less	✓	
Resource Identification	1 AA	n/a
Do-it-yourself	✓	
URI Design	2 AA	n/a
“Nice” URI scheme	✓	
No URI scheme	✓	
Resource Interaction Semantics	2 AAs	n/a
Lo-REST (POST, GET only)	✓	
Hi-REST (4 verbs)	✓	
Resource Relationships	1 AA	n/a
Do-it-yourself	✓	
Data Representation/Modeling	1 AA	1 AA
XML Schema	(✓) ^a	✓
Do-it-yourself	✓	
Message Exchange Patterns	1 AA	2 AAs
Request-Response	✓	✓
One-Way		✓
Service Operations Enumeration	n/a	≥3 AAs
By functional domain		✓
By non-functional properties and QoS		✓
By organizational criterion (versioning)		✓
Total Number of Decisions, AAs	8, 10	5, ≥10

^aOptional

Table 2: Conceptual Comparison Summary

Architectural Decision and AAs	REST	WS-*
Transport Protocol	1 AA	≥7 AAs
HTTP	✓	✓ ^a
waka [13]	(✓) ^b	
TCP		✓
SMTP		✓
JMS		✓
MQ		✓
BEEP		✓
IIOF		✓
Payload Format	≥6 AAs	1 AA
XML (SOAP)	✓	✓
XML (POX)	✓	
XML (RSS)	✓	
JSON [10]	✓	
YAML	✓	
MIME	✓	
Service Identification	1 AA	2 AA
URI	✓	✓
WS-Addressing		✓
Service Description	3 AAs	2 AAs
Textual Documentation	✓	
XML Schema	(✓) ^c	✓
WSDL	✓ ^d	✓
WADL [18]	✓	
Reliability	1 AA	4 AAs
HTTPR [38] ^e	(✓)	(✓)
WS-Reliability		✓
WS-ReliableMessaging		✓
Native		✓
Do-it-yourself	✓	✓
Security	1 AA	2 AAs
HTTPS	✓	✓
WS-Security		✓

Transactions	1 AA	3 AAs
WS-AT, WS-BA		✓
WS-CAF		✓
Do-it-yourself	✓	✓
Service Composition	2 AAs	2 AAs
WS-BPEL		✓
Mashups	✓	
Do-it-yourself	✓	✓
Service Discovery	1 AAs	2 AAs
UDDI		✓
Do-it-yourself	✓	✓
Implementation Technology	many	many
...	✓	✓
Total Number of Decisions, AAs	10, ≥17	10, ≥25

^aLimited to only the verb POST

^bStill under development

^cOptional

^dWSDL 2.0

^eNot standard

Table 3: Technology Comparison Summary

Architectural Principle and Aspects	REST	WS-*
Protocol Layering	yes	yes
HTTP as application-level protocol	✓	
HTTP as transport-level protocol		✓
Dealing with Heterogeneity	yes	yes
Browser Wars	✓	
Enterprise Computing Middleware		✓
Loose Coupling , aspects covered	yes, 2	yes, 3
Time/Availability		✓
Location (Dynamic Late Binding)	(✓)	✓
Service Evolution:		
Uniform Interface	✓	
XML Extensibility	✓	✓
Total Principles Supported	3	3

Table 1: Principles Comparison Summary

21 Decisions and 64 alternatives
Classified by level of abstraction:

- 3 Architectural Principles
- 9 Conceptual Decisions
- 9 Technology-level Decisions

Decisions help us to measure the complexity implied by the choice of REST or WS-*

Decision	AA1	AA2
Contract First	✓	✓
Contract First	✓	✓
Resource Identification	✓	✓
Do-it-yourself	✓	✓
URI Design	✓	✓
"Nice" URI scheme	✓	✓
No URI scheme	✓	✓
Resource Interaction Semantics	✓	✓
Lo-REST (POST/GET only)	✓	✓
Hi-REST (4 verbs)	✓	✓
Resource Relationships	✓	✓
Do-it-yourself	✓	✓
Data Representation/Media	✓	✓
XML Schema	✓	✓
Do-it-yourself	✓	✓
Message Exchange Patterns	1 AA	2 AAs
Request-Response	✓	✓
One-Way	✓	✓
Service Operations Enumeration	n/a	≥3 AAs
By functional domain	✓	✓
By non-functional properties and QoS	✓	✓
By organizational criterion (versioning)	✓	✓
Total Number of Decisions, AAs	8, 10	5, ≥10

Decision	AA1	AA2
Service Identification	1 AA	2 AA
WS-Addressing	✓	✓
XML Schema	✓	✓
Reliability	1 AA	4 AAs
WS-ReliableMessaging	✓	✓
Native	✓	✓
Do-it-yourself	✓	✓
Security	1 AA	2 AAs
HTTPS	✓	✓
WS-Security	✓	✓

Principle	AA1	AA2	AA3
Discovery	✓	✓	✓
Discovery	✓	✓	✓
Service Discovery	1 AA	2 AAs	3 AAs
URI	✓	✓	✓
Do-it-yourself	✓	✓	✓
Implementation Technology	many	many	many
...	✓	✓	✓
Total Number of Decisions, AAs	10, ≥1	10, ≥25	10, ≥25

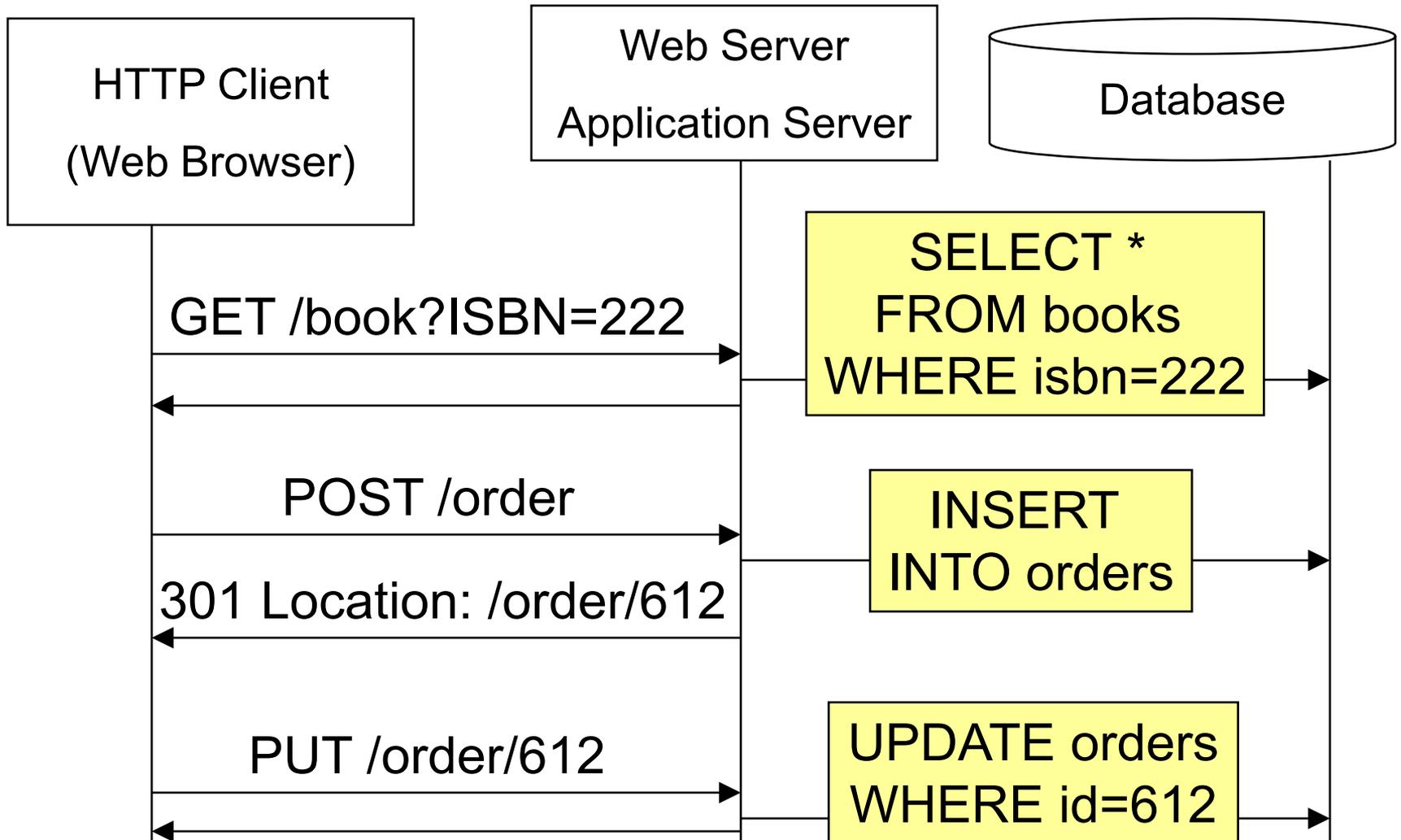
Principle	REST	WS-*
Protocol Layering	yes	yes
HTTP as application-level protocol	✓	✓
HTTP as transport-level protocol	✓	✓
Dealing with Heterogeneity	yes	yes
Enterprise Computing Middleware	✓	✓
Loose Coupling, aspects covered	yes, 2	yes, 3
Time/Availability	✓	✓
Location (Dynamic Late Binding)	(✓)	✓
Service Evolution:		
Uniform Interface	✓	✓
XML Extensibility	✓	✓
Total Principles Supported	3	3

Table 2: Conceptual Comparison Summary

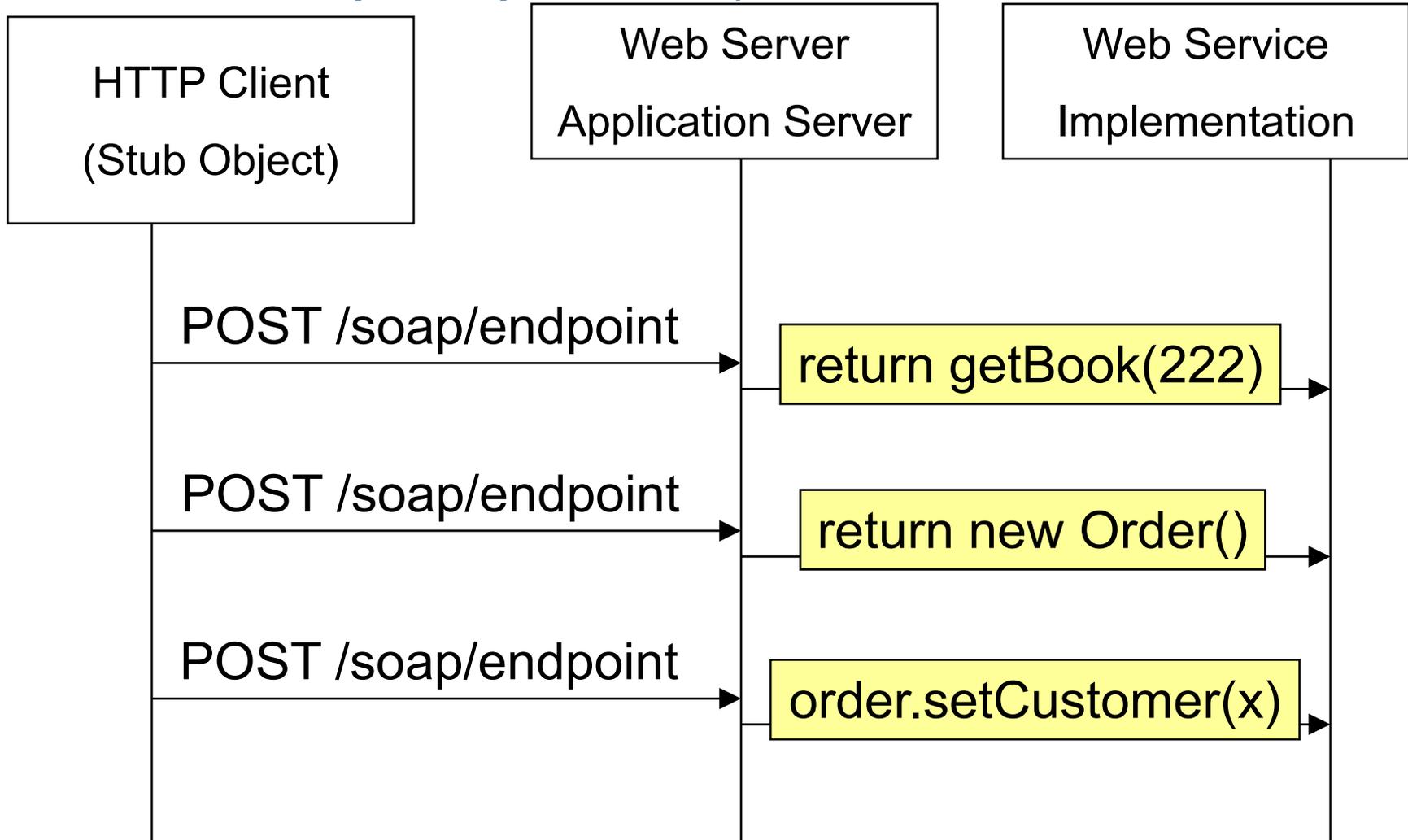
Table 1: Principles Comparison Summary

1. Protocol Layering
 - HTTP = Application-level Protocol (REST)
 - HTTP = Transport-level Protocol (WS-*)
2. Loose Coupling
3. Dealing with Heterogeneity
4. What about X?
5. (X = Composition)
6. Software Connectors for Integration

RESTful Web Service Example

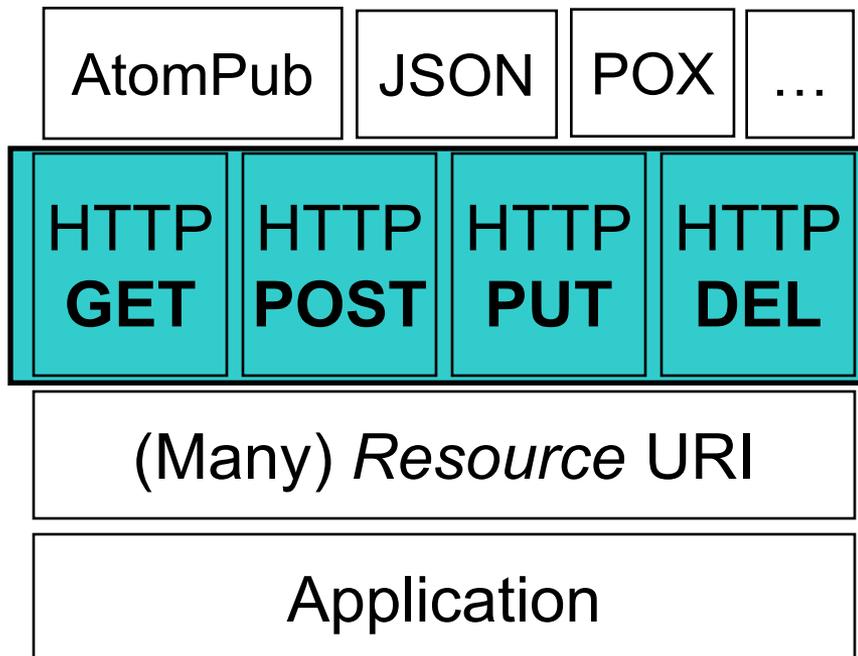


WS-* Service Example (from REST perspective)



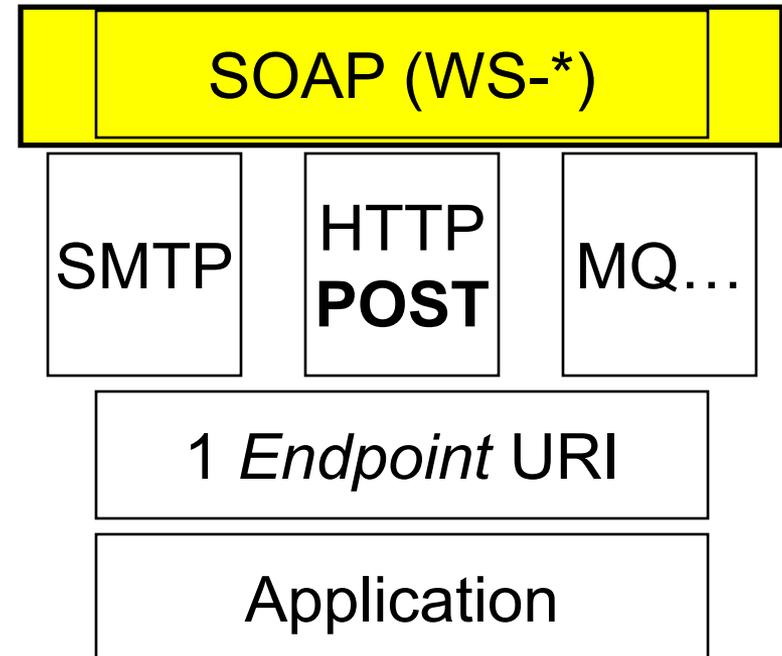
“The Web is the universe of globally accessible information”
(Tim Berners Lee)

- Applications should publish their data on the Web (through URI)



“The Web is the universal (tunneling) transport for messages”

- Applications get a chance to interact but they remain “outside of the Web”



Facets

REST

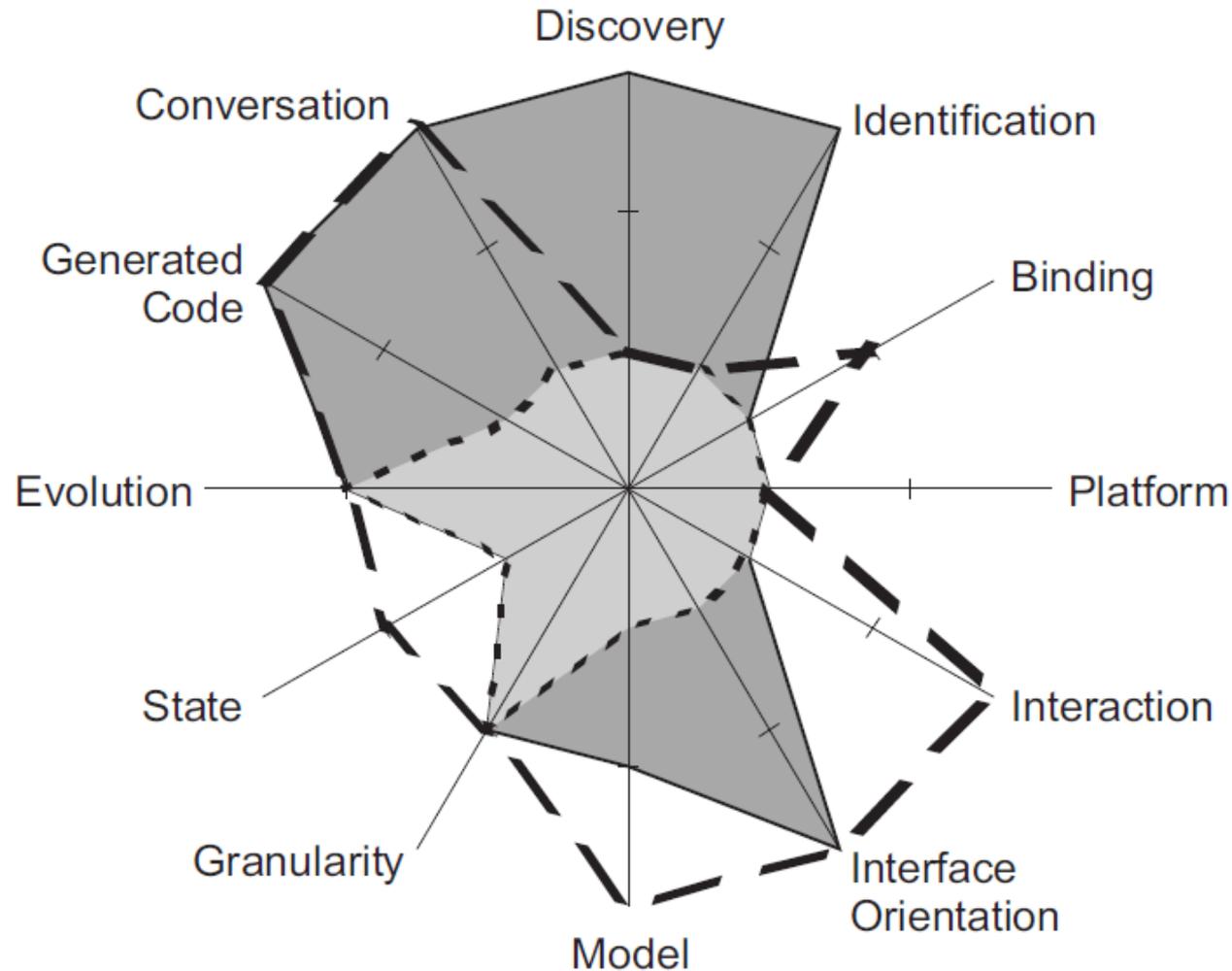
WS-*

- | | | |
|------------------|-------------------|-----------------|
| ■ Discovery | ■ Referral | ■ Centralized |
| ■ Identification | ■ Global | ■ Context-Based |
| ■ Binding | ■ Late | ■ Late |
| ■ Platform | ■ Independent | ■ Independent |
| ■ Interaction | ■ Asynchronous | ■ Asynchronous |
| ■ Model | ■ Self-Describing | ■ Shared Model |
| ■ State | ■ Stateless | ■ Stateless |
| ■ Generated Code | ■ None/Dynamic | ■ Static |
| ■ Conversation | ■ Reflective | ■ Explicit |

More Info on <http://dret.net/netdret/docs/loosely-coupled-www2009/>

Coupling Comparison

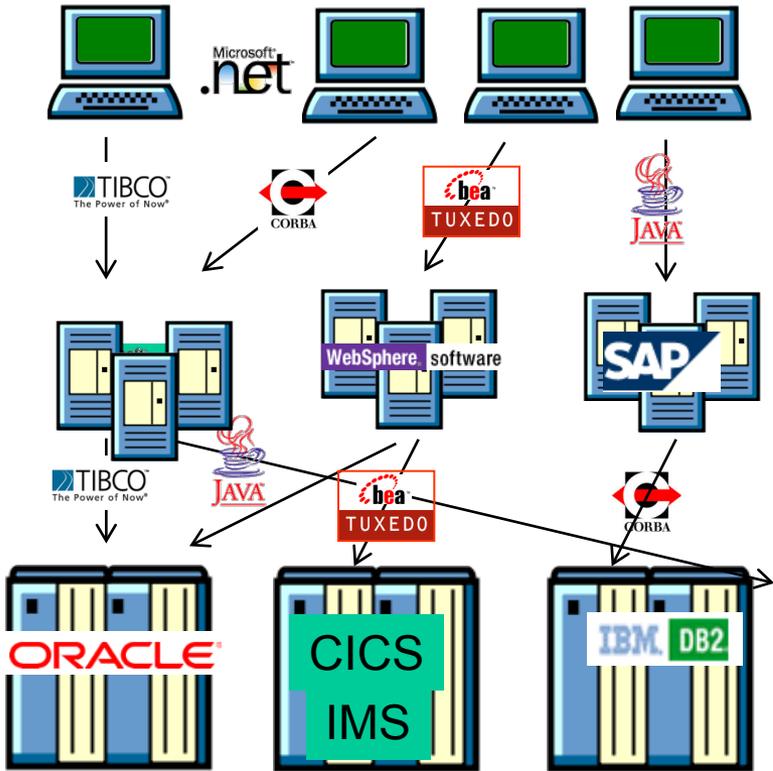
▣ RESTful HTTP ▣ RPC over HTTP ▣ WS-*/ESB



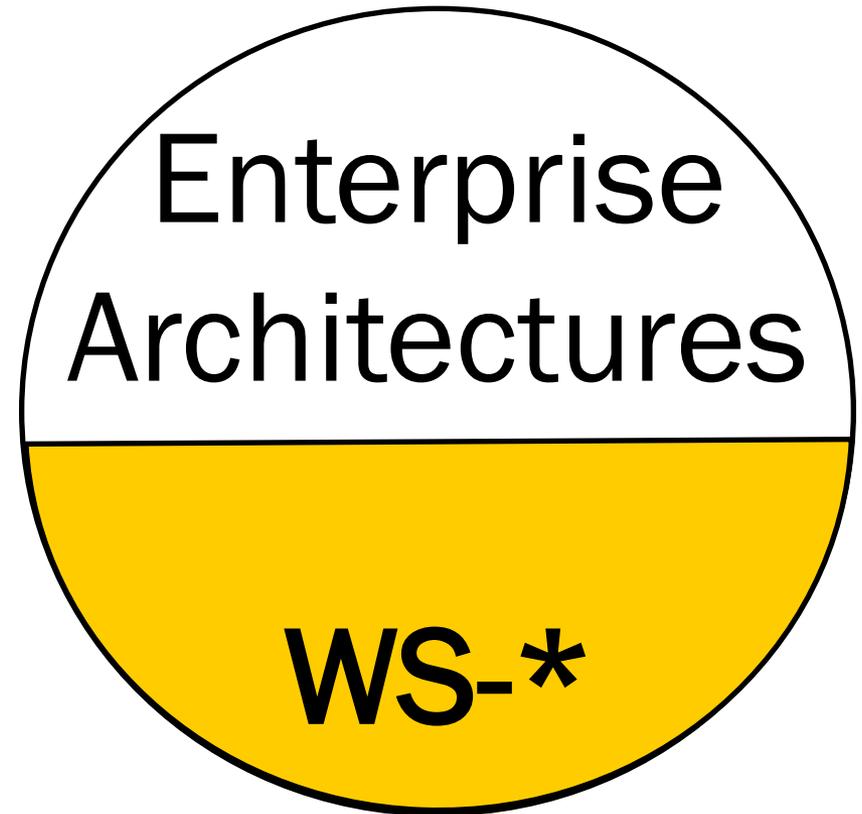
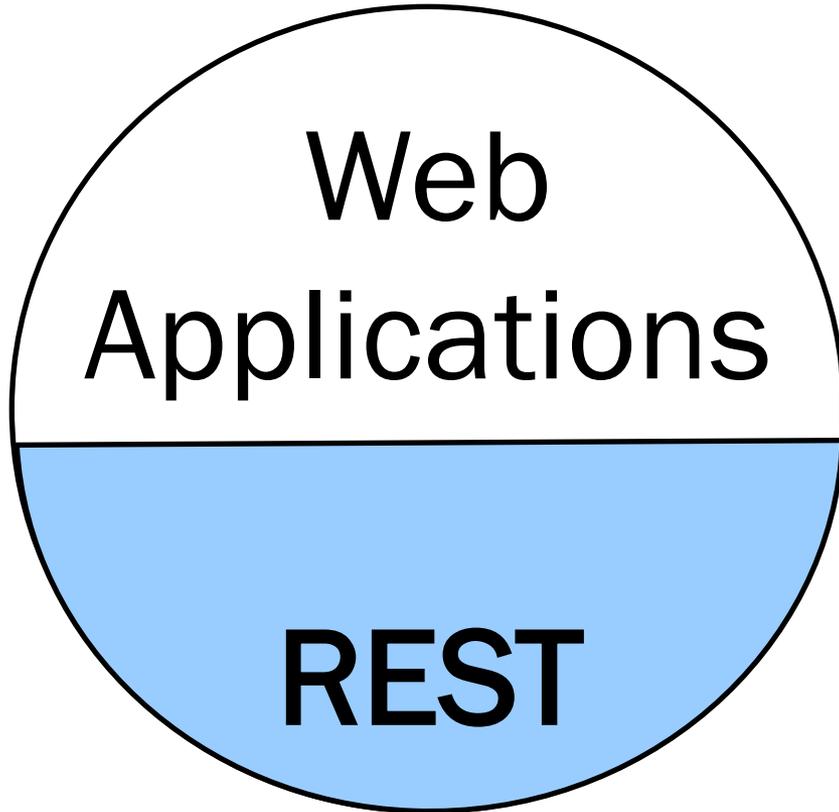
- Enable Cooperation
- Web Applications

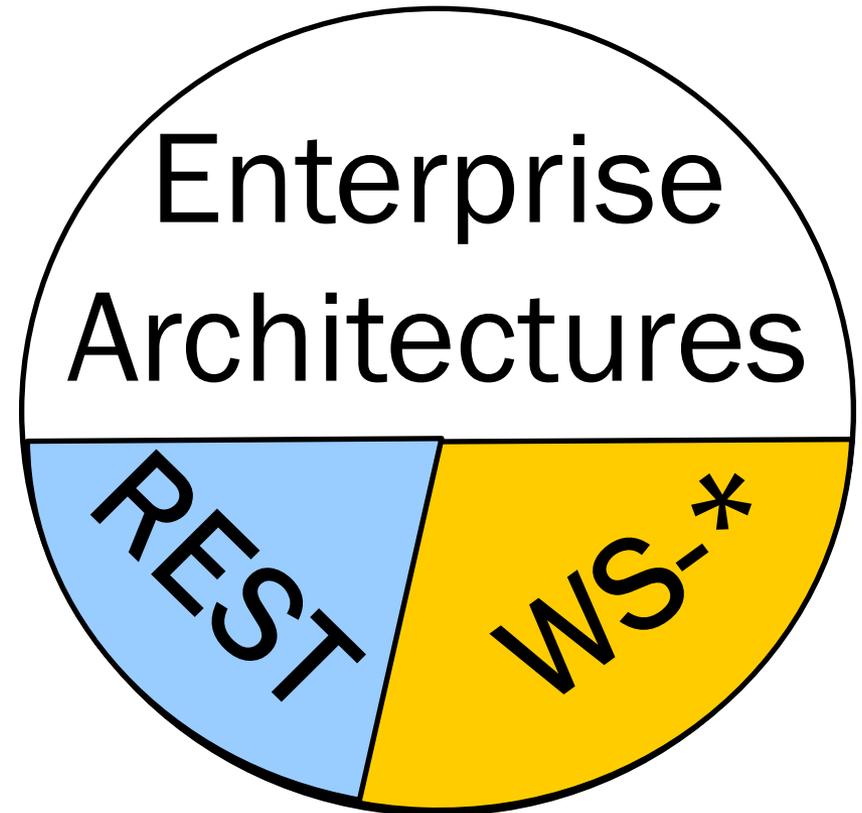
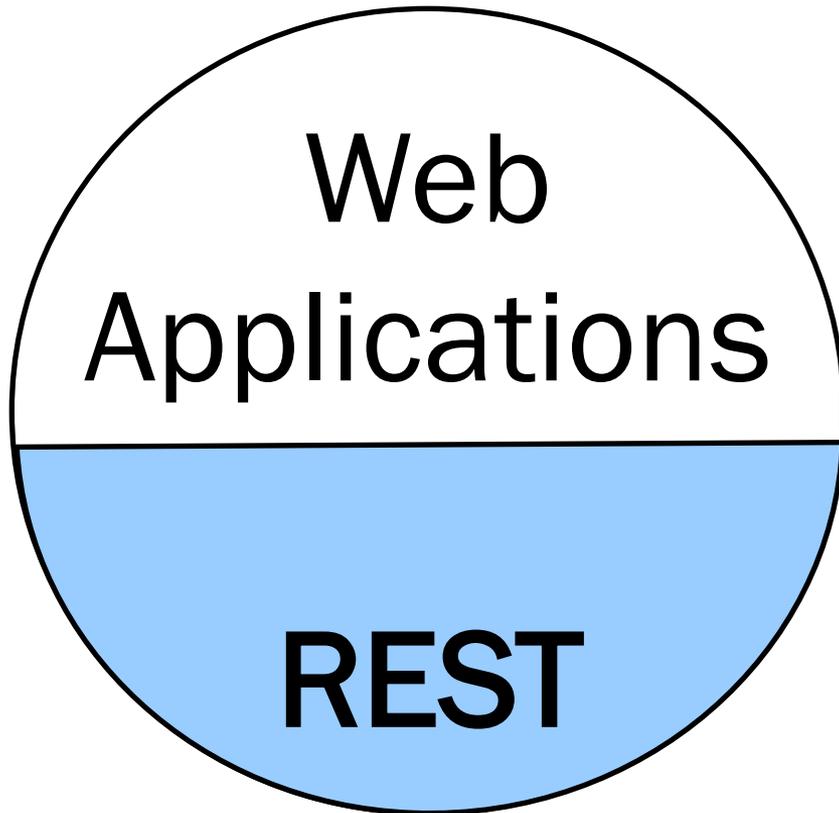


- Enable Integration
- Enterprise Architectures

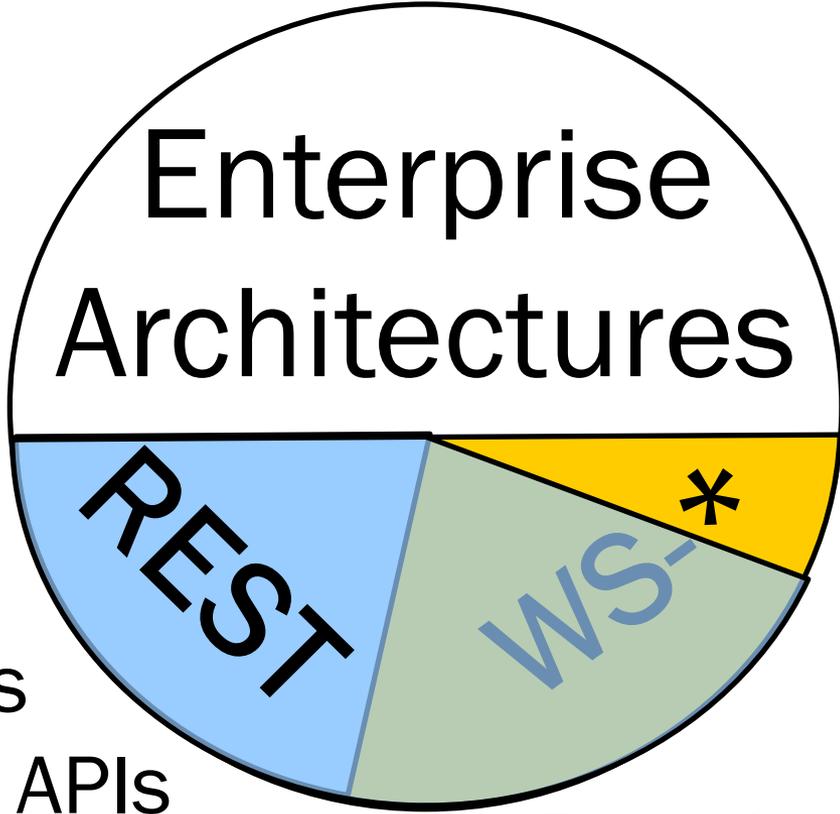


Picture from Eric Newcomer, IONA



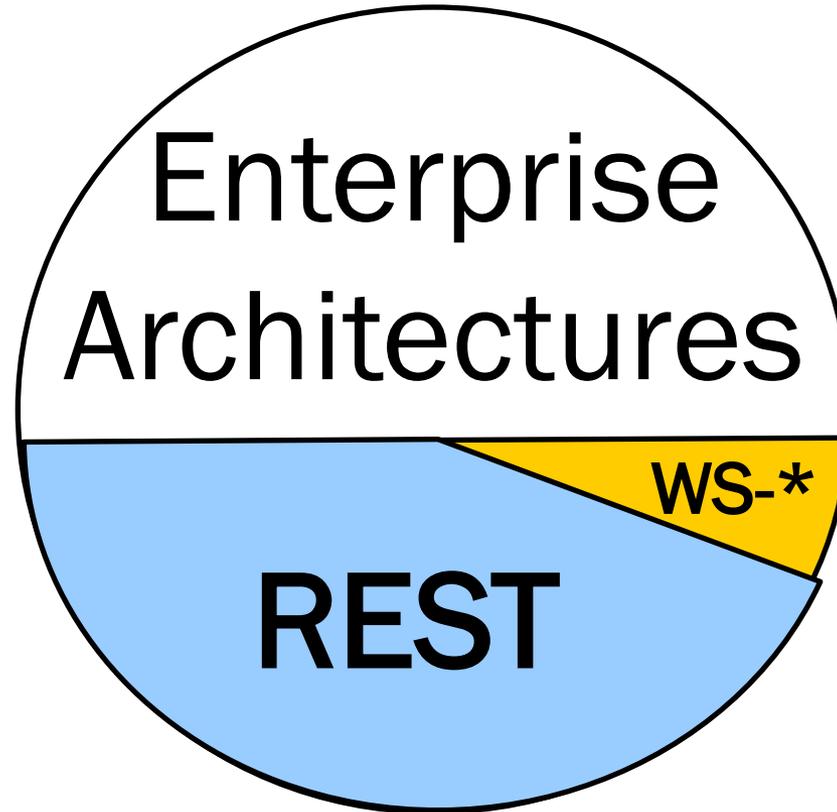


Claim: REST can also be successfully used to design integrated enterprise applications



CRUD Services
Web-friendly APIs
Mobile Services

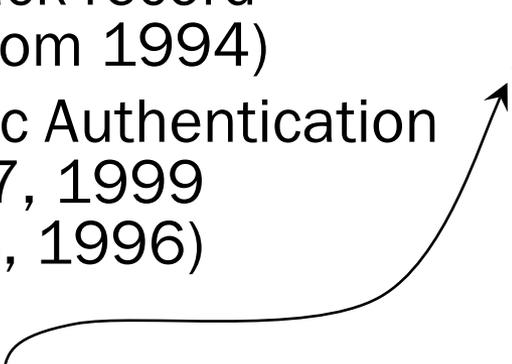
Real-time Services
Transactional Services
Composite Services



Part of the debate is about how many “enterprise” use cases can be covered with REST as opposed to WS-*

- Service Description
- Security
- Asynch Messaging
- Reliable Messaging
- Stateful Services
- Service Composition
- Transactions
- Semantics
- SLAs
- Governance
- ...

- REST relies on human readable documentation that defines requests URIs templates and responses (XML, JSON media types)
- 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
- XForms 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) provides a new HTTP binding

- REST security is all about HTTPS (HTTP + SSL/TLS)
 - Proven track record (SSL1.0 from 1994)
 - HTTP Basic Authentication (RFC 2617, 1999
RFC 1945, 1996)
 - Note: These are also applicable with REST when using XML content
 - Secure, point to point communication (Authentication, Integrity and Encryption)
 - SOAP security extensions defined by WS-Security (from 2004)
 - XML Encryption (2002)
 - XML Signature (2001)
 - Secure, end-to-end communication – Self-protecting SOAP messages (does not require HTTPS)
- 

What about asynchronous messaging?

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

POST /queue

202 Accepted

Location:

/queue/message/1230213

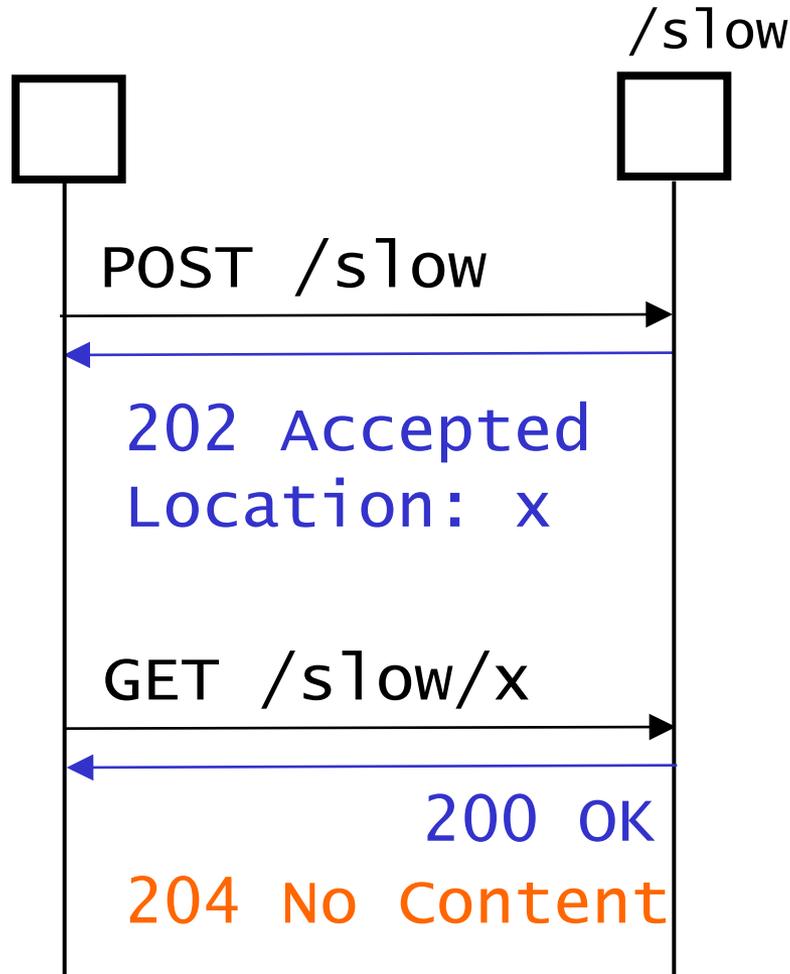
GET /queue/message/1230213

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

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? /slow/x could include an estimate of the finishing time if not yet completed

What about reliable messaging?

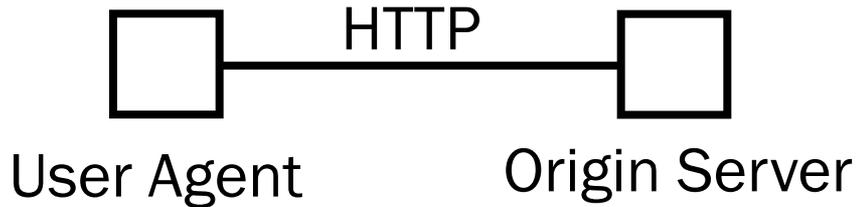
- The HTTP uniform interface defines clear exception handling semantics
- If a failure occurs it is enough to **retry idempotent** methods (GET, PUT, DELETE)
- With POST, recovery requires an additional reconciliation step (usually done with GET) before the request can be retried
- POE (POST-Once-Exactly) has been proposed to also make POST reliable
- WS-ReliableMessaging (or WS-Reliability) define a protocol for reliable message delivery based on SOAP headers for message identification and acknowledgement
- WS-* middleware can ensure guaranteed in-order, exactly once message delivery semantics
- Hint: Reliable Messaging does **not** imply reliable applications!

- REST provides explicit state transitions
 - Communication is stateless*
 - Resources contain data **and hyperlinks** representing valid state transitions
 - Clients maintain application state correctly by navigating hyperlinks
- Techniques for adding session to HTTP:
 - Cookies (HTTP Headers)
 - URI 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)
 - WS-Resource Framework (HTTP on top of SOAP on top of HTTP)

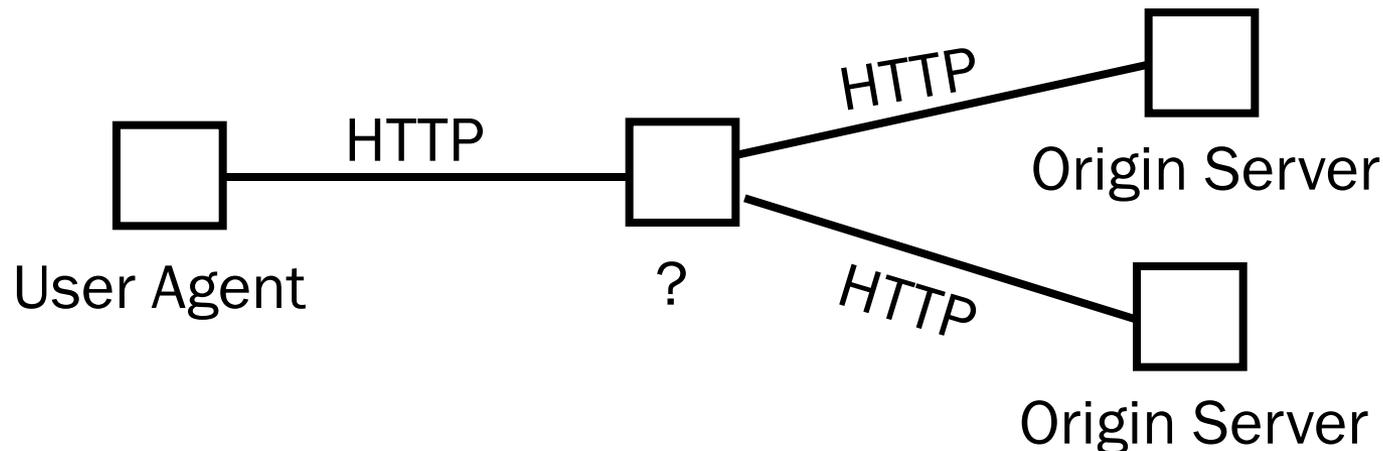
(*) Each client request to the server must contain all information needed to understand the request, without referring to any stored context on the server. Of course the server stores the state of its resources, shared by all clients.

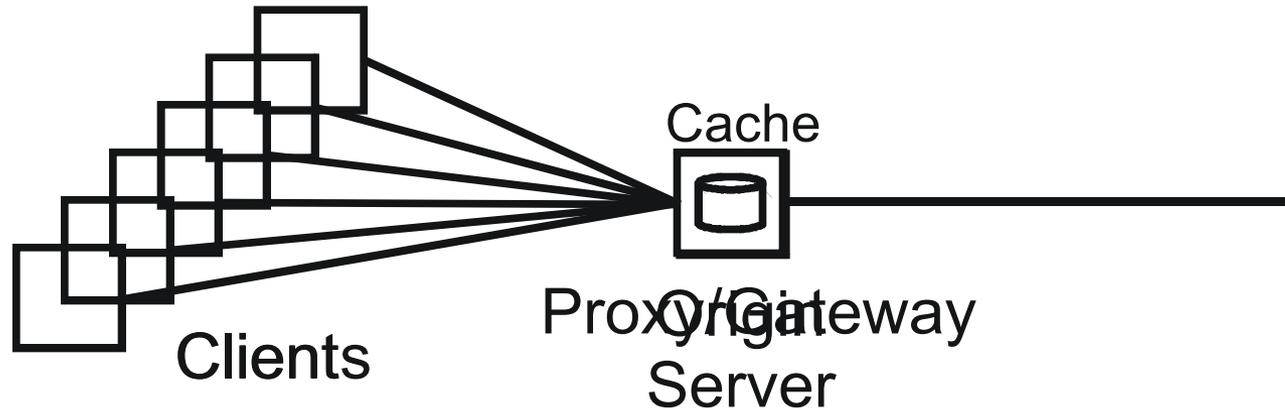
What about composition?

- The basic REST design elements do not take composition into account

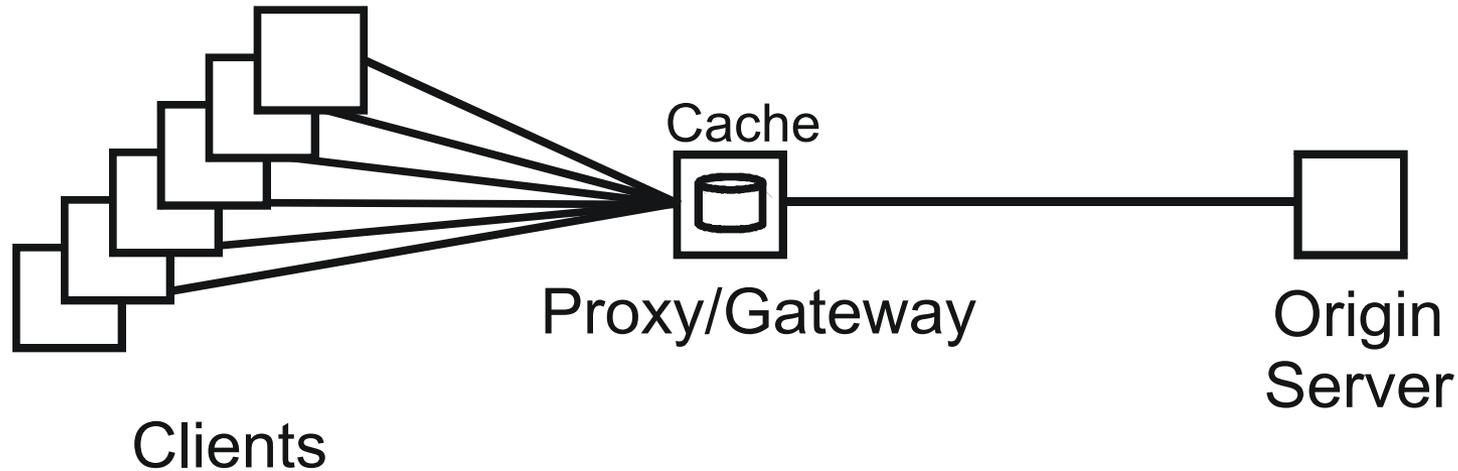


- WS-BPEL is the standard Web service composition language. Business process models are used to specify how a collection of services is orchestrated into a composite service
- Can we apply WS-BPEL to RESTful services?

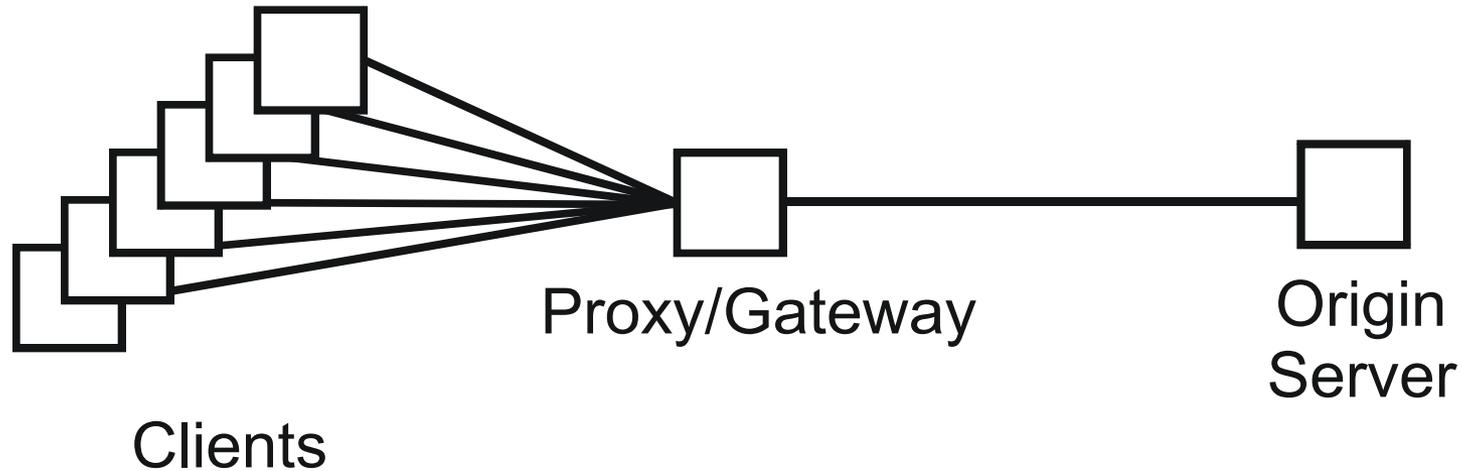




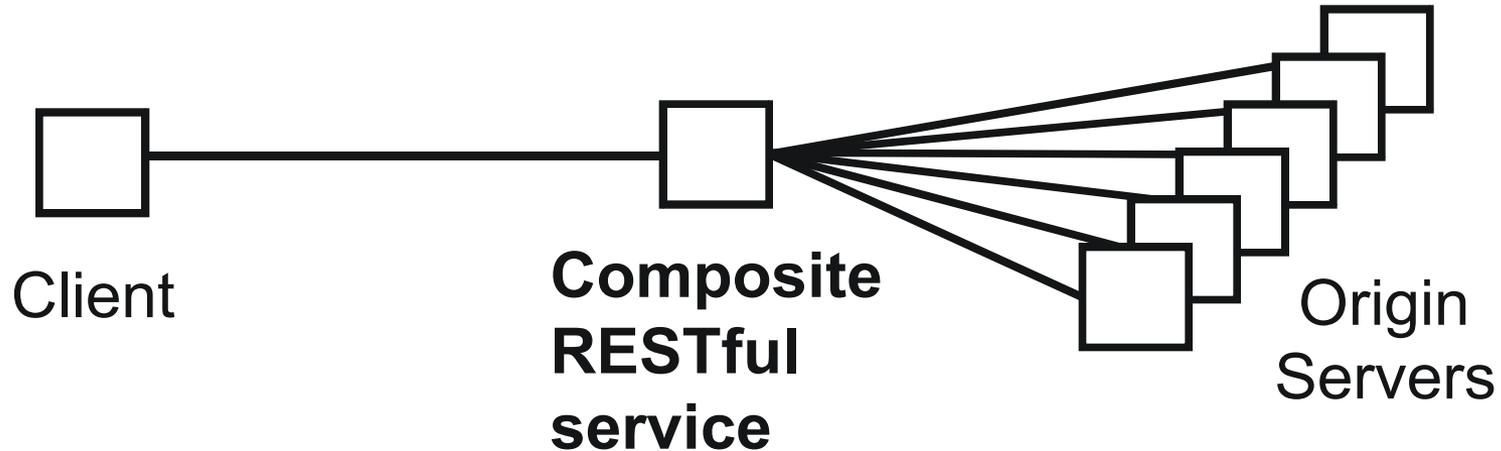
- One example of REST middleware is to help with the scalability of a server, which may need to service a very large number of clients



- One example of REST middleware is to help with the scalability of a server, which may need to service a very large number of clients

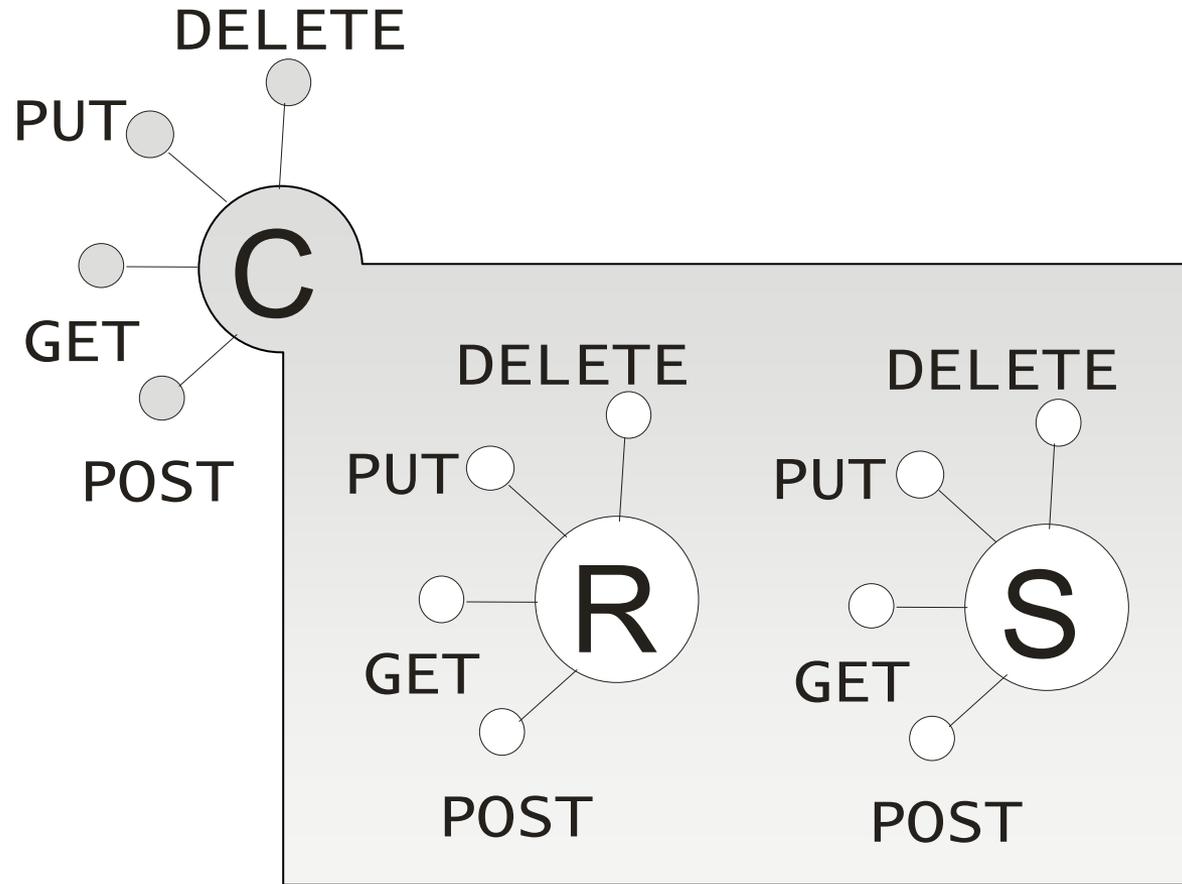


- Composition shifts the attention to the client which should consume and aggregate from many servers

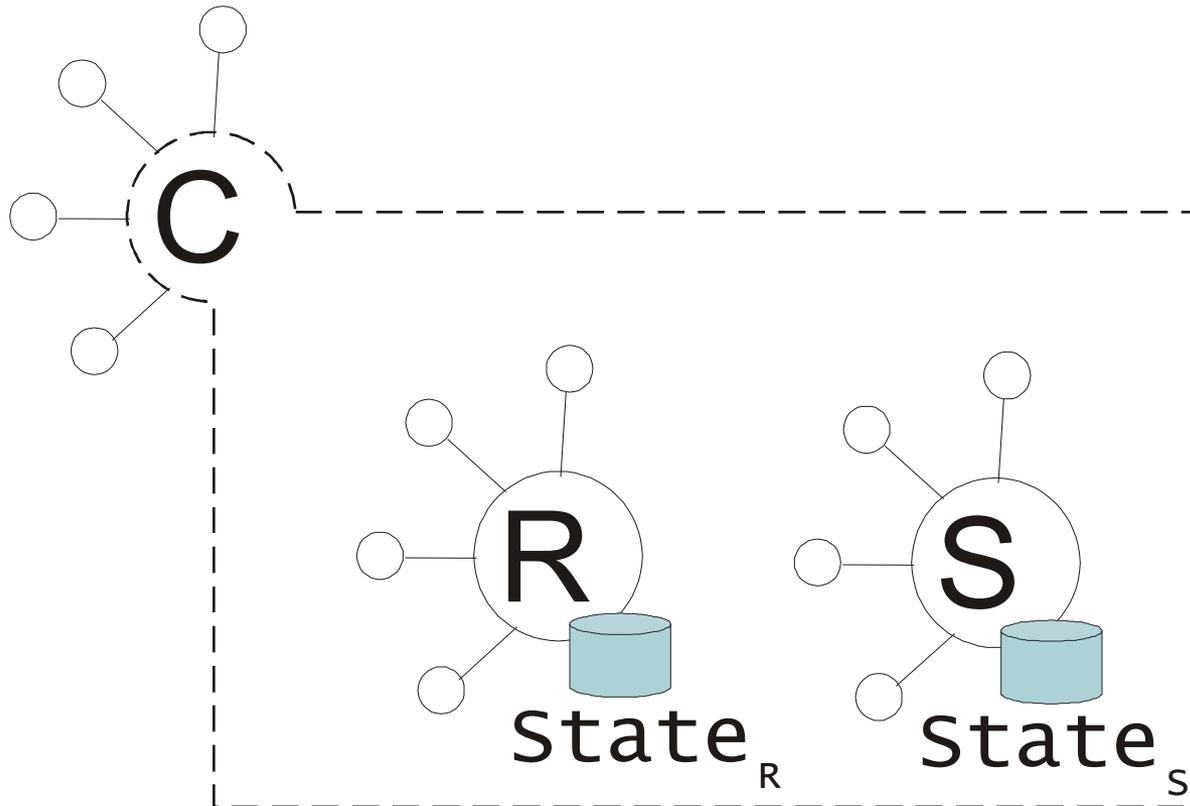


- The “proxy” intermediate element which aggregates the resources provided by multiple servers plays the role of a composite RESTful service
- Can/Should we implement it with BPM?

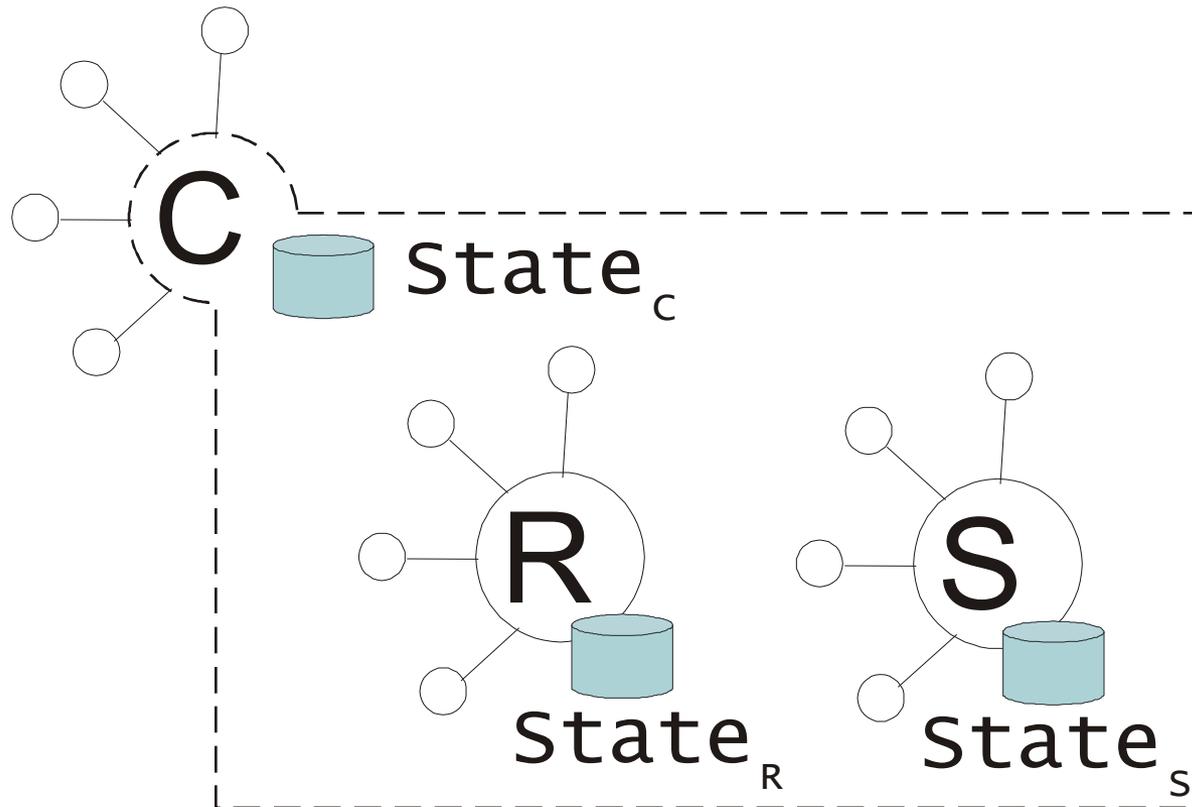
Composite Resources



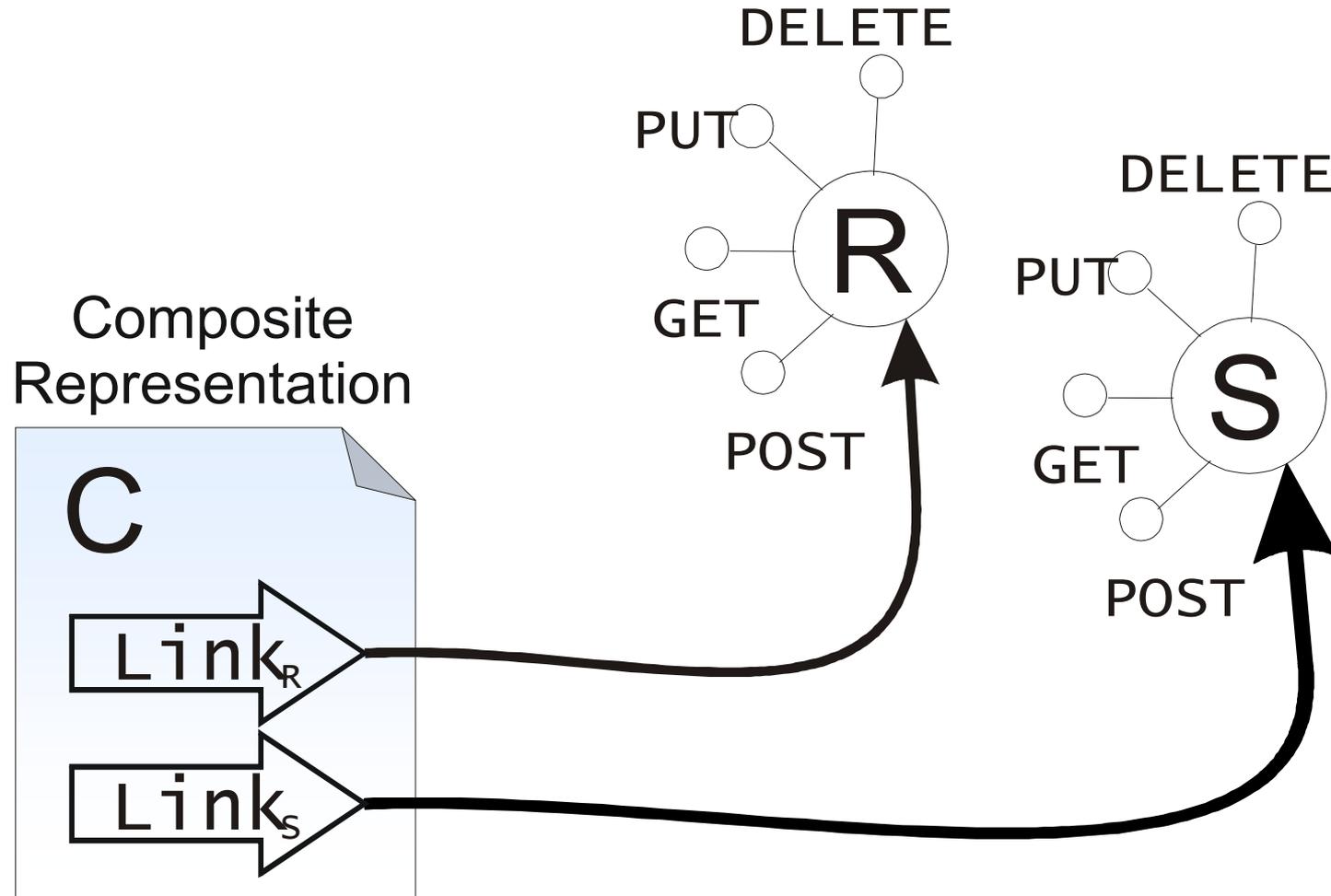
- The composite resource only aggregates the state of its component resources

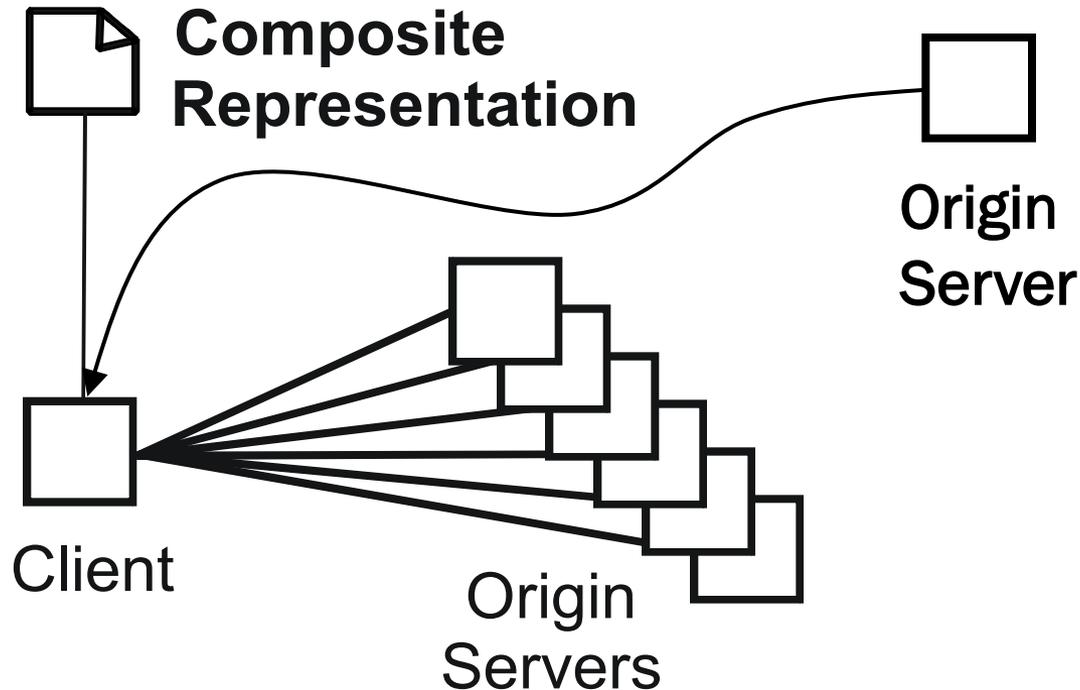


- The composite resource augments (or caches) the state of its component resources



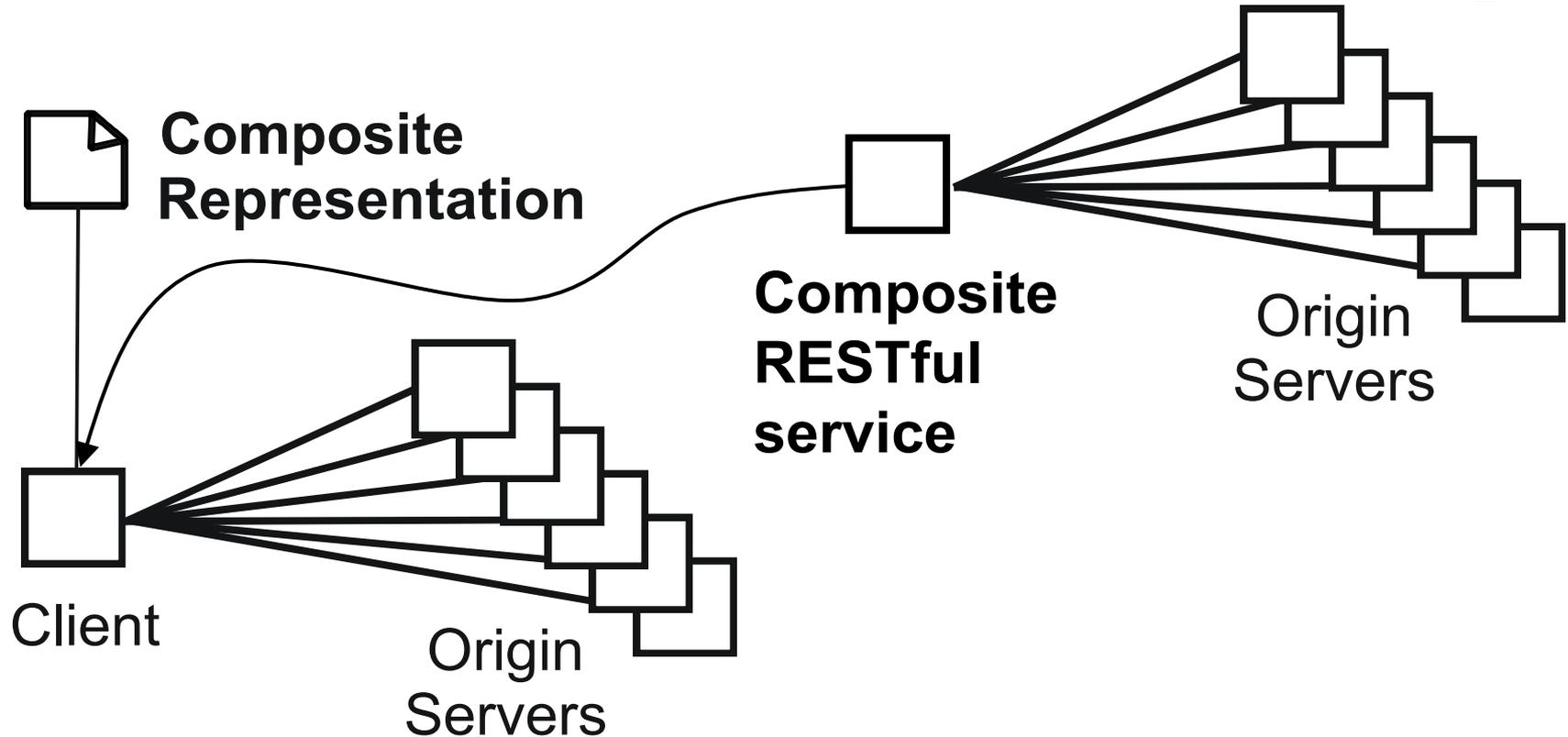
Composite Representation





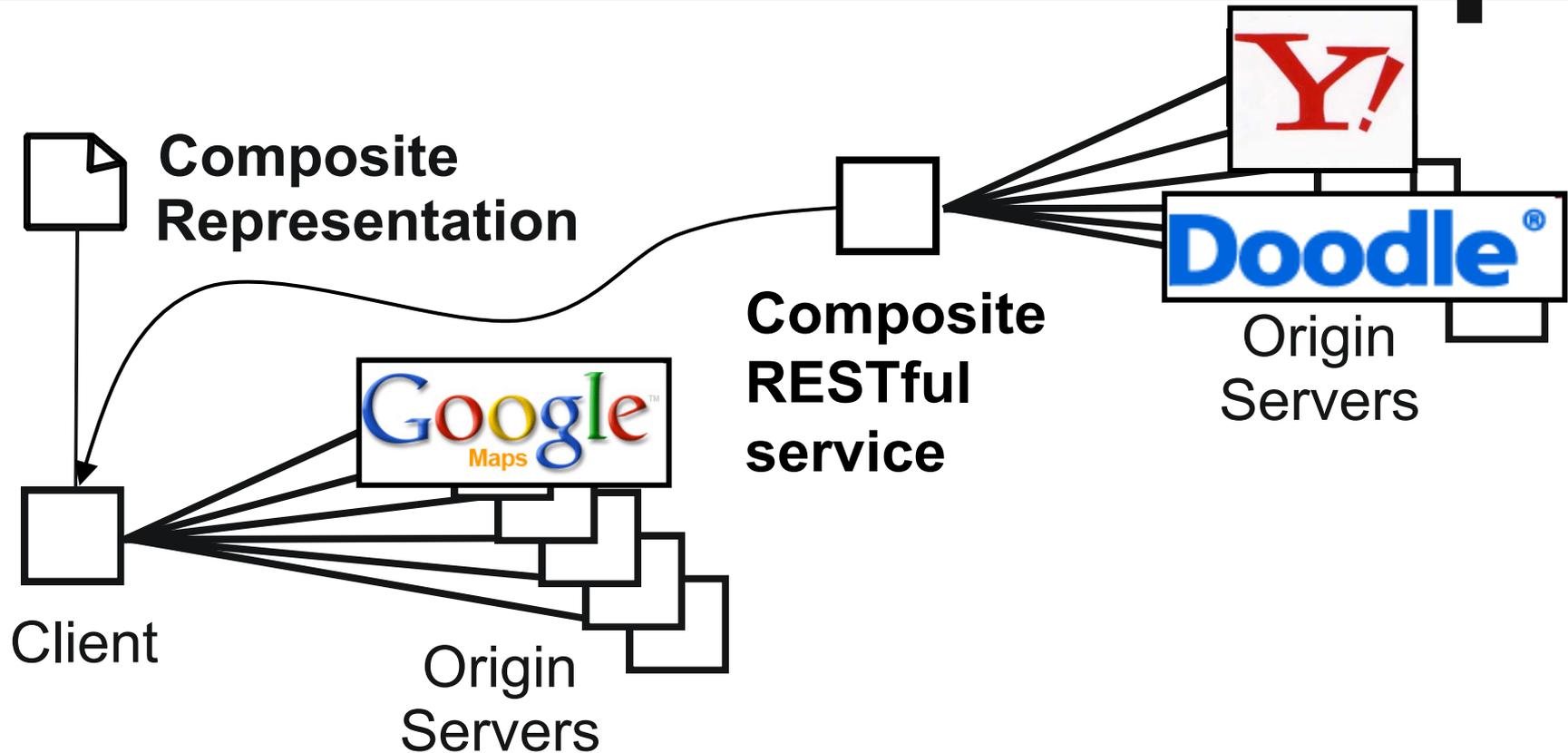
- A composite representation is interpreted by the client that follows its hyperlinks and aggregates the state of the referenced component resources

Bringing it all together



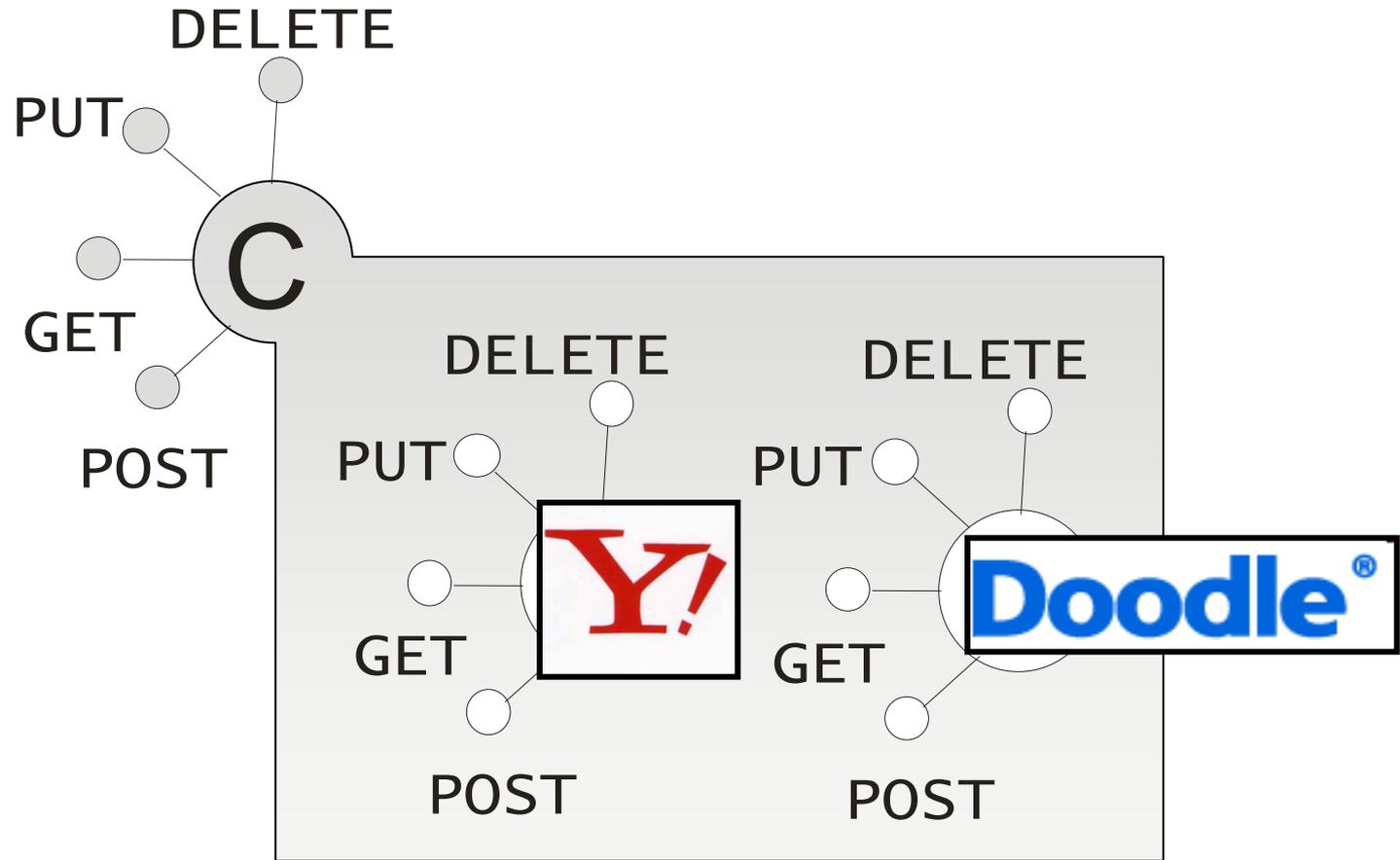
- A composite representation can be produced by a composite service too

Doodle Map Example

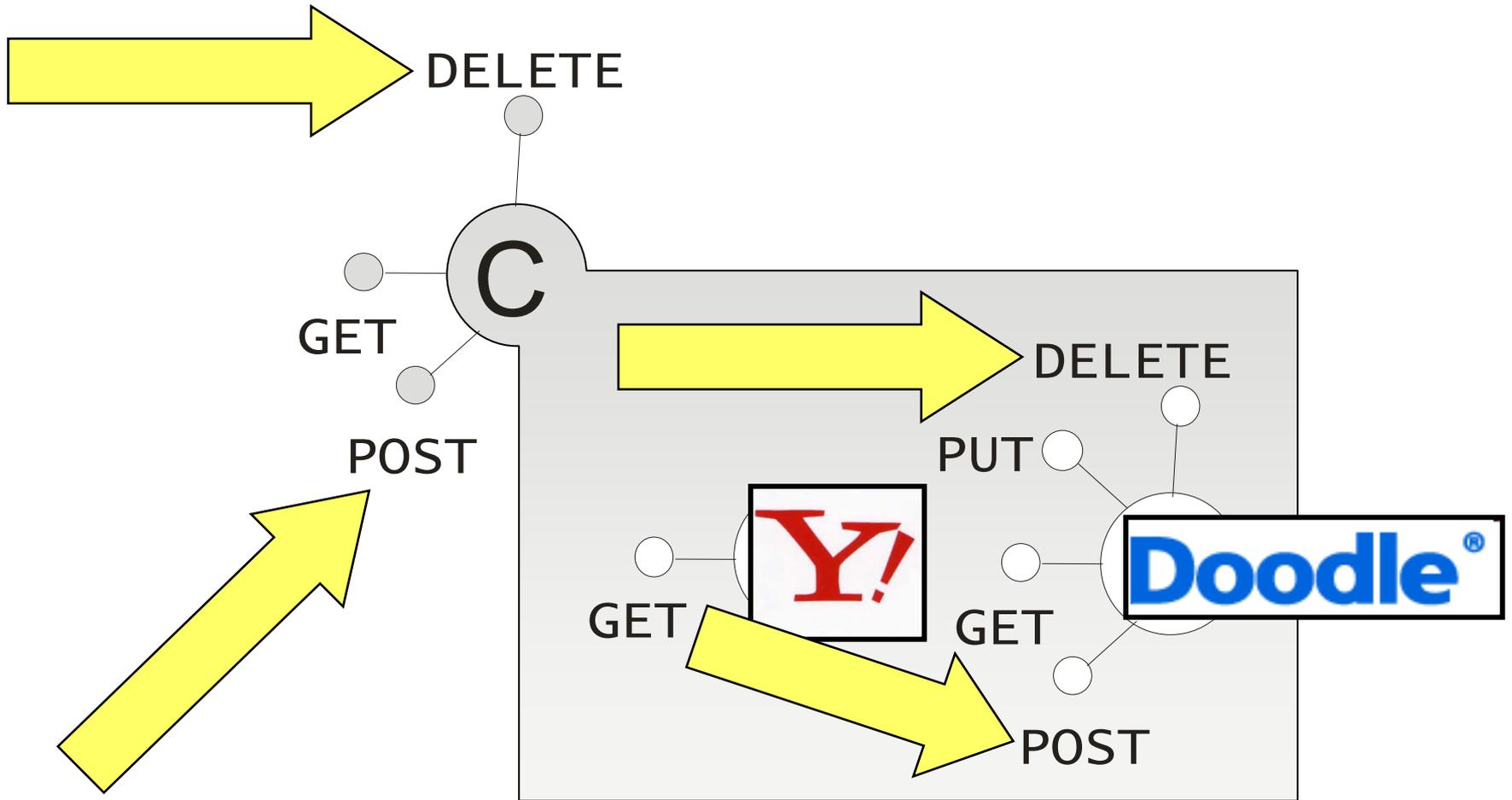


- Vote on a meeting place based on its geographic location

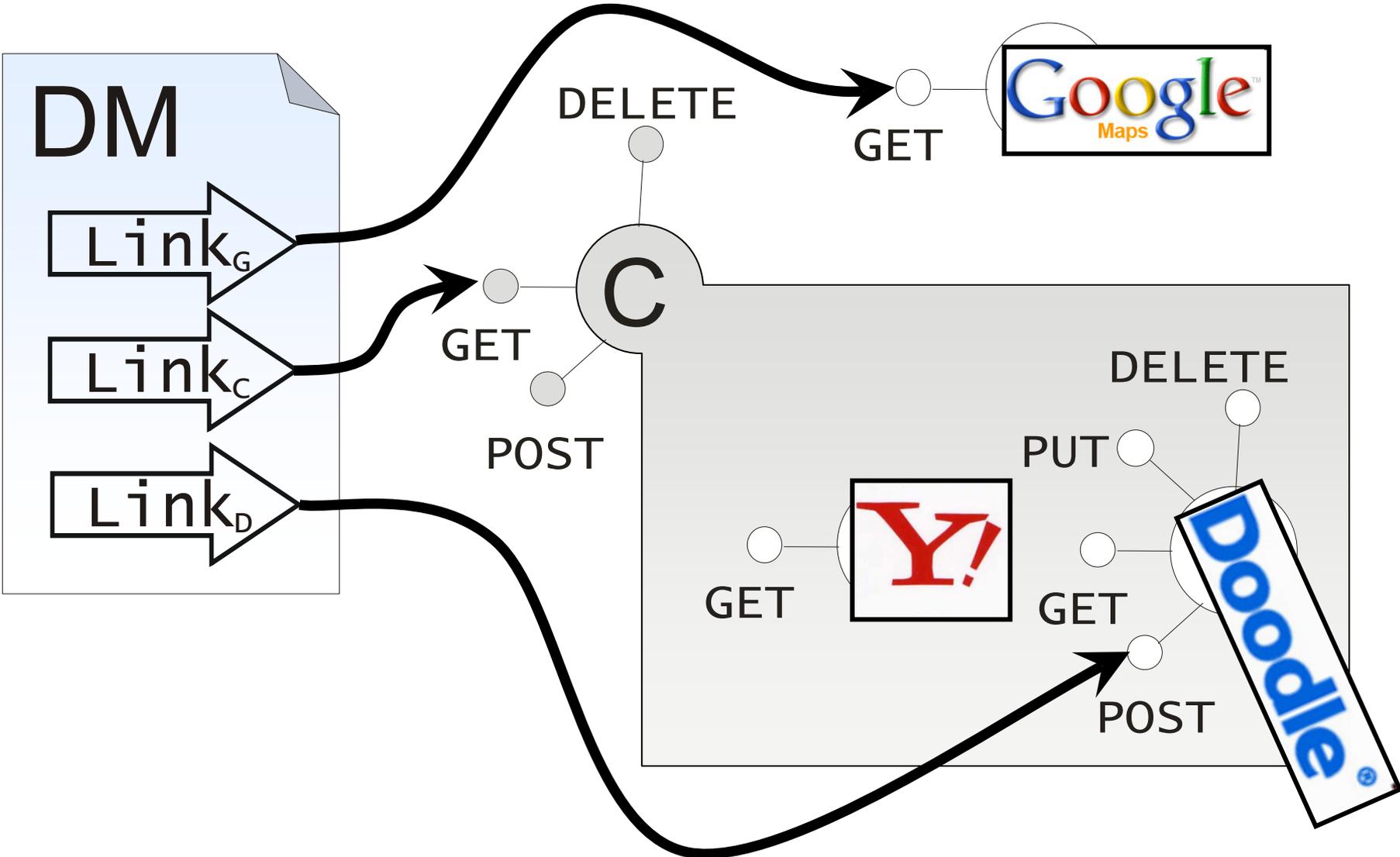
Composite Resource

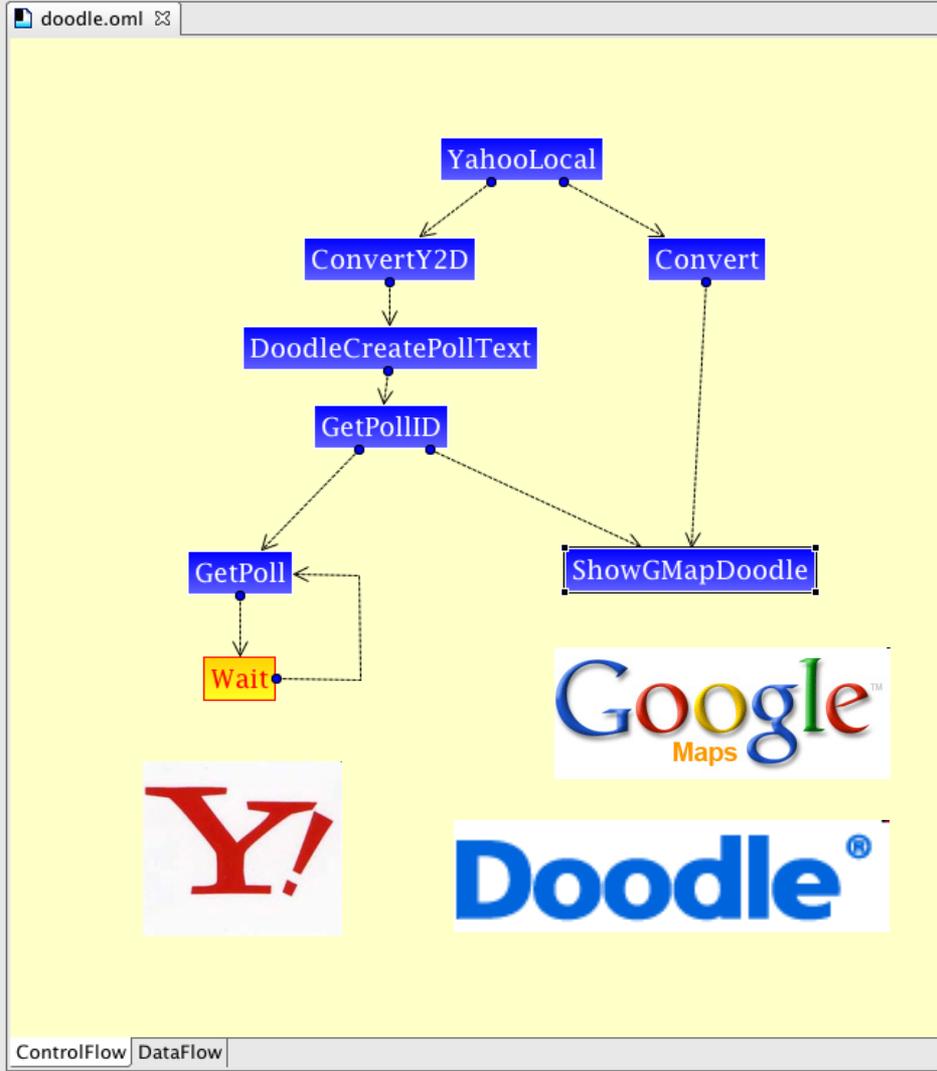


Composite Resource



Composite Representation





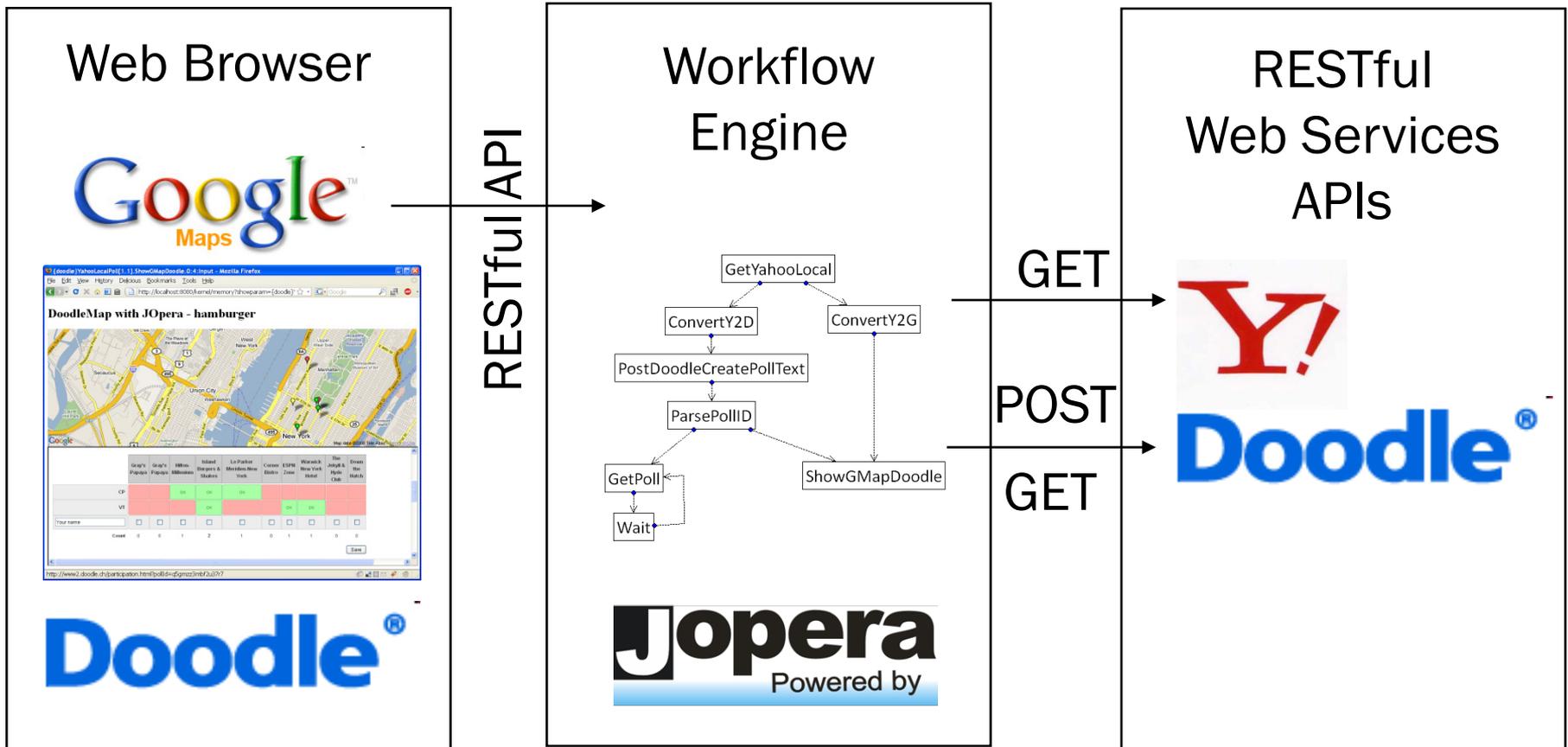
DoodleMap with JOpera

The browser window shows a Google Map of New York City. A popup window titled "Island Burgers & Shakes Preferences:2" is displayed over the map. Below the map, there is a poll titled "Poll: hamburger" created by "CP". The poll shows a list of restaurant names and a grid of responses from users "CP" and "PA".

Poll: hamburger
 CP has created this poll.
 "10001"

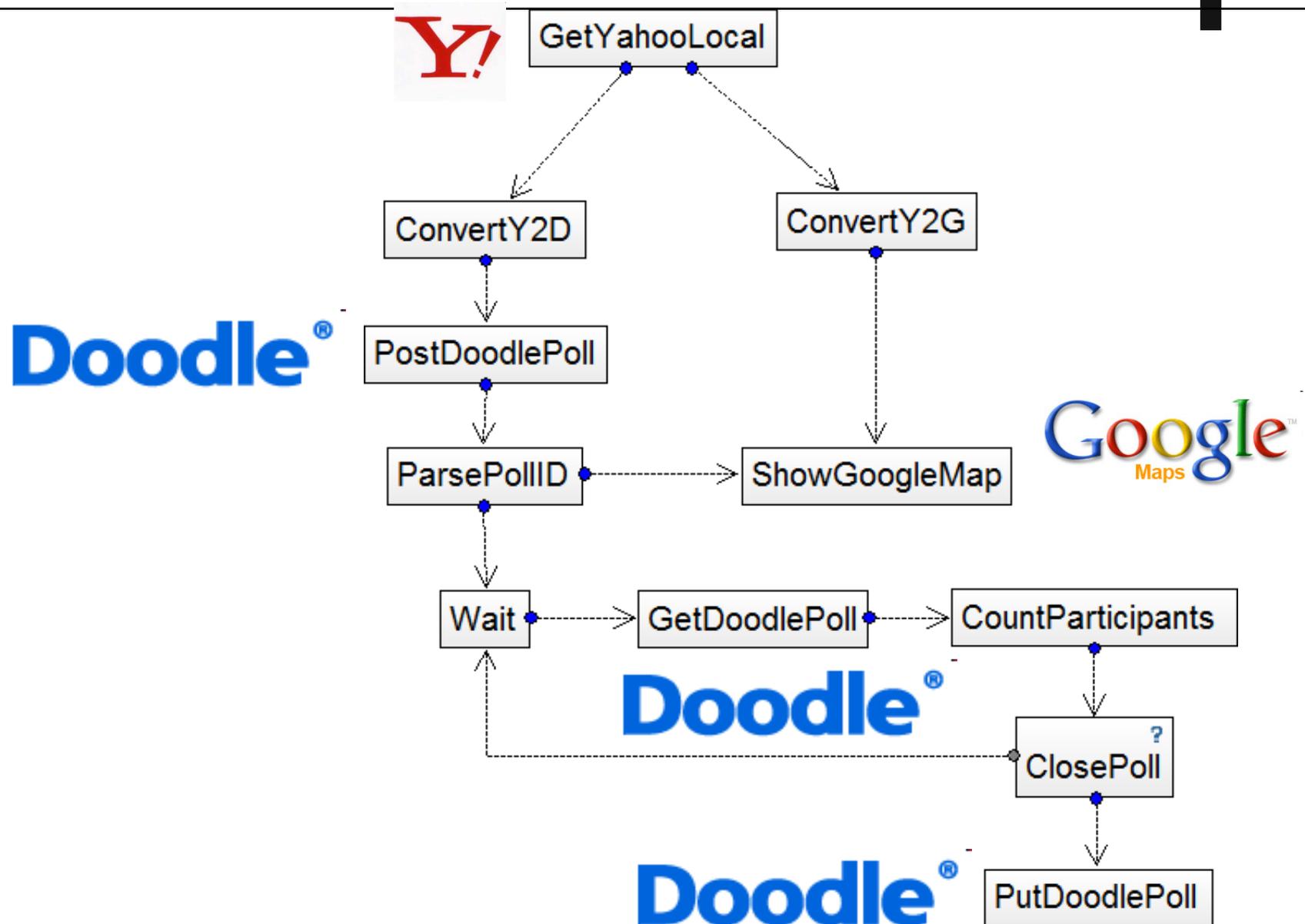
	Gray's Papaya	Gray's Papaya	Hilton-Millennium	Island Burgers & Shakes	Le Parker Meridien - New York	Corner Bistro	ESPN Zone	Warwick New York Hotel	The Jekyll & Hyde Club	Dow the Hat
CP				OK		OK				
PA				OK	OK					

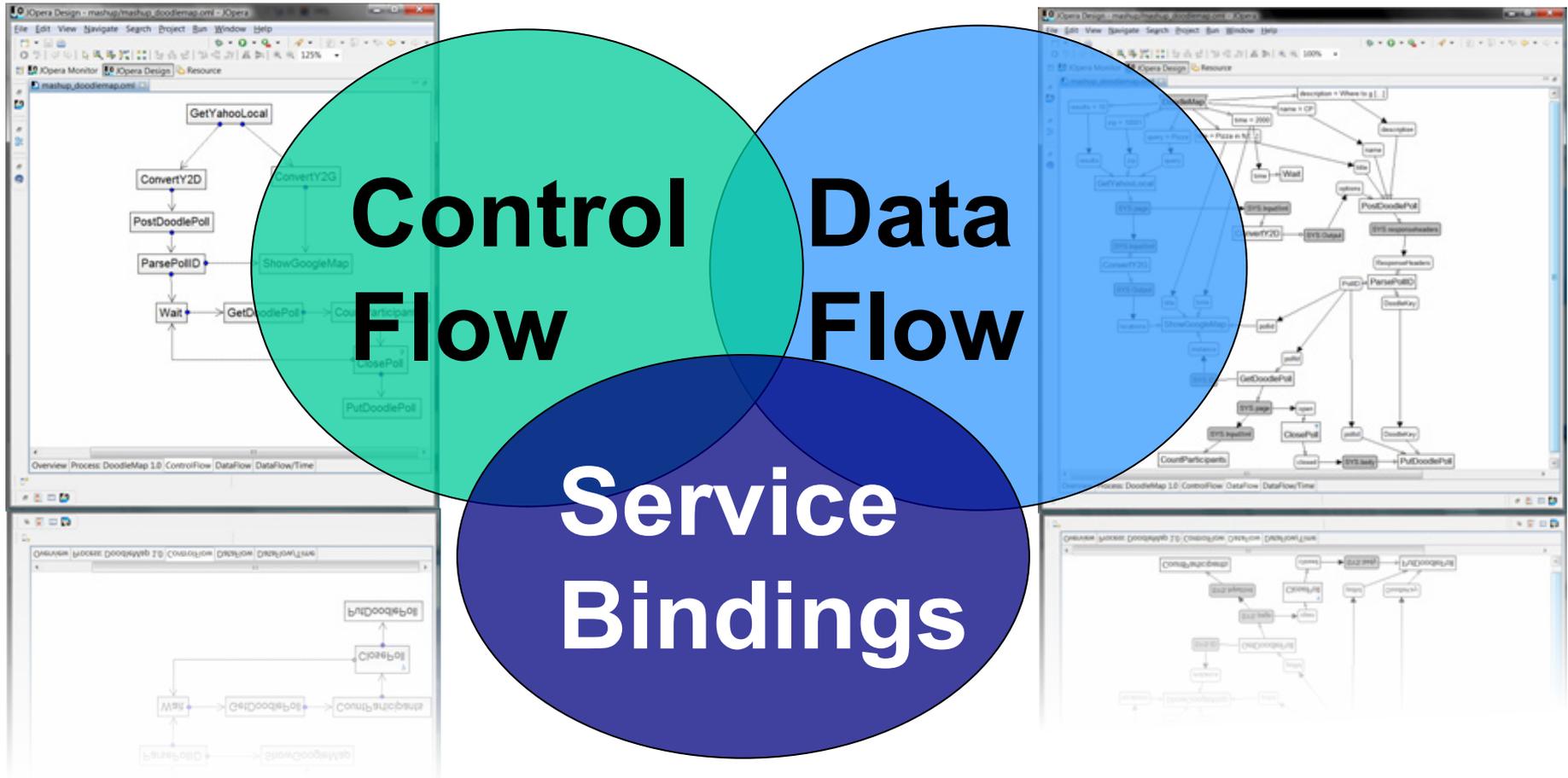
Doodle Map Architecture



Watch it on <http://www.jopera.org/docs/videos/doodlemap>

DoodleMap Model





JAVA

XPATH

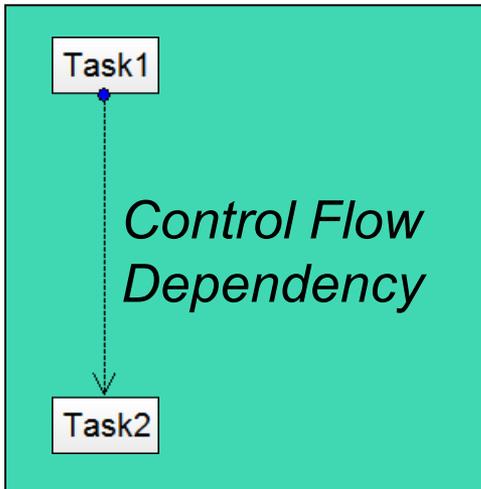
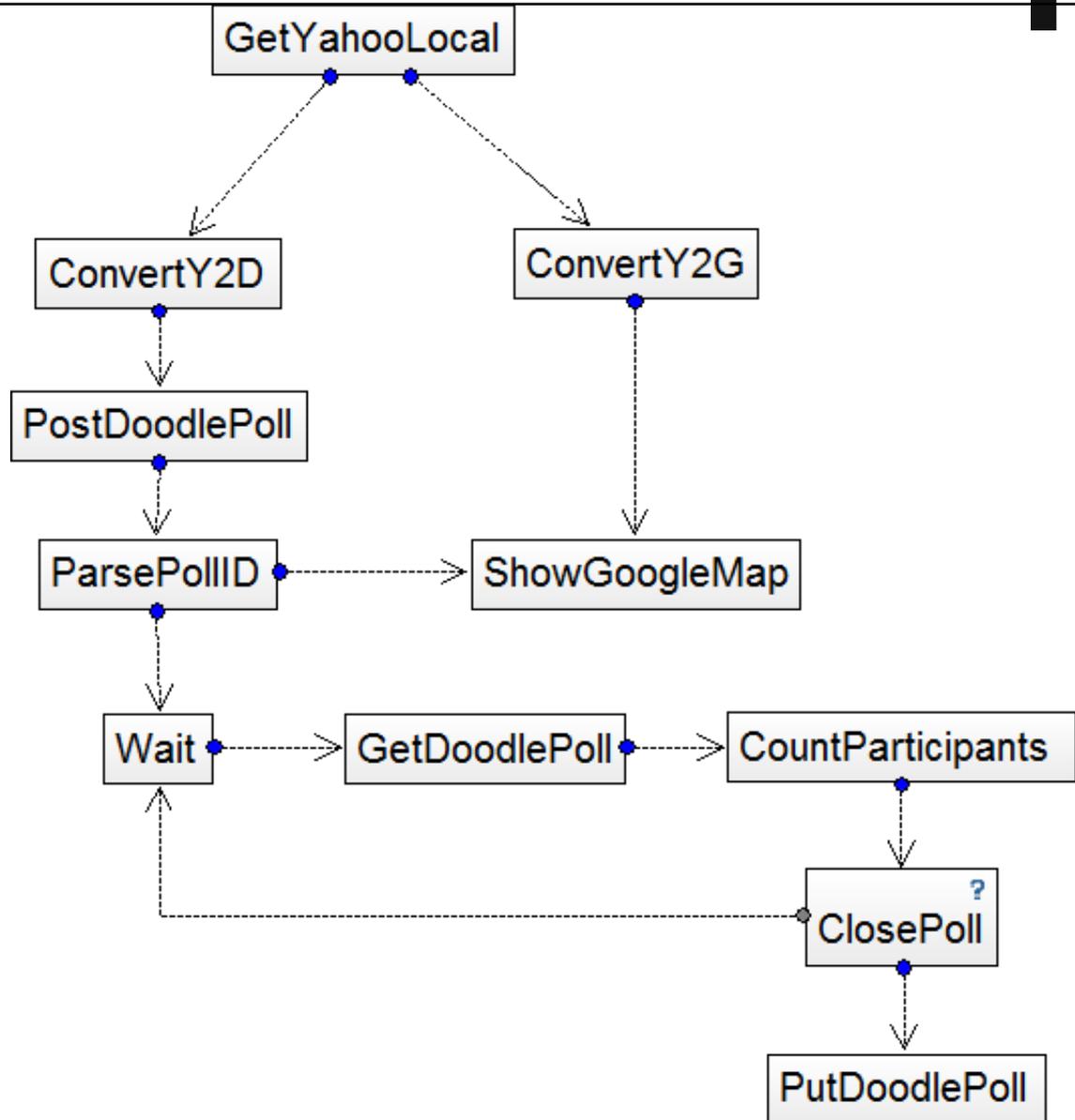
XSLT

HTML

HTTP

...

Control Flow



Service Bindings

HTTP

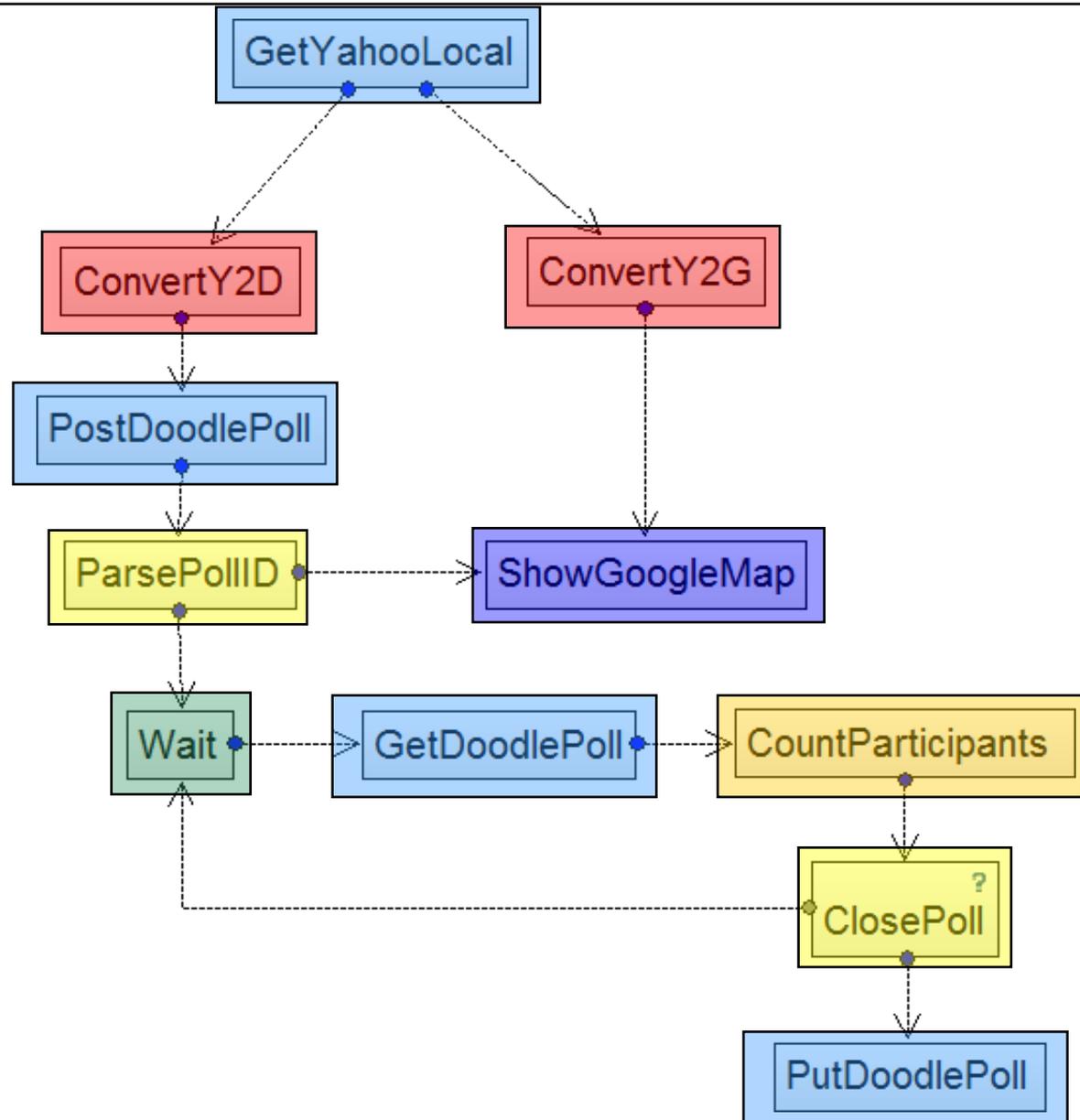
HTML

XSLT

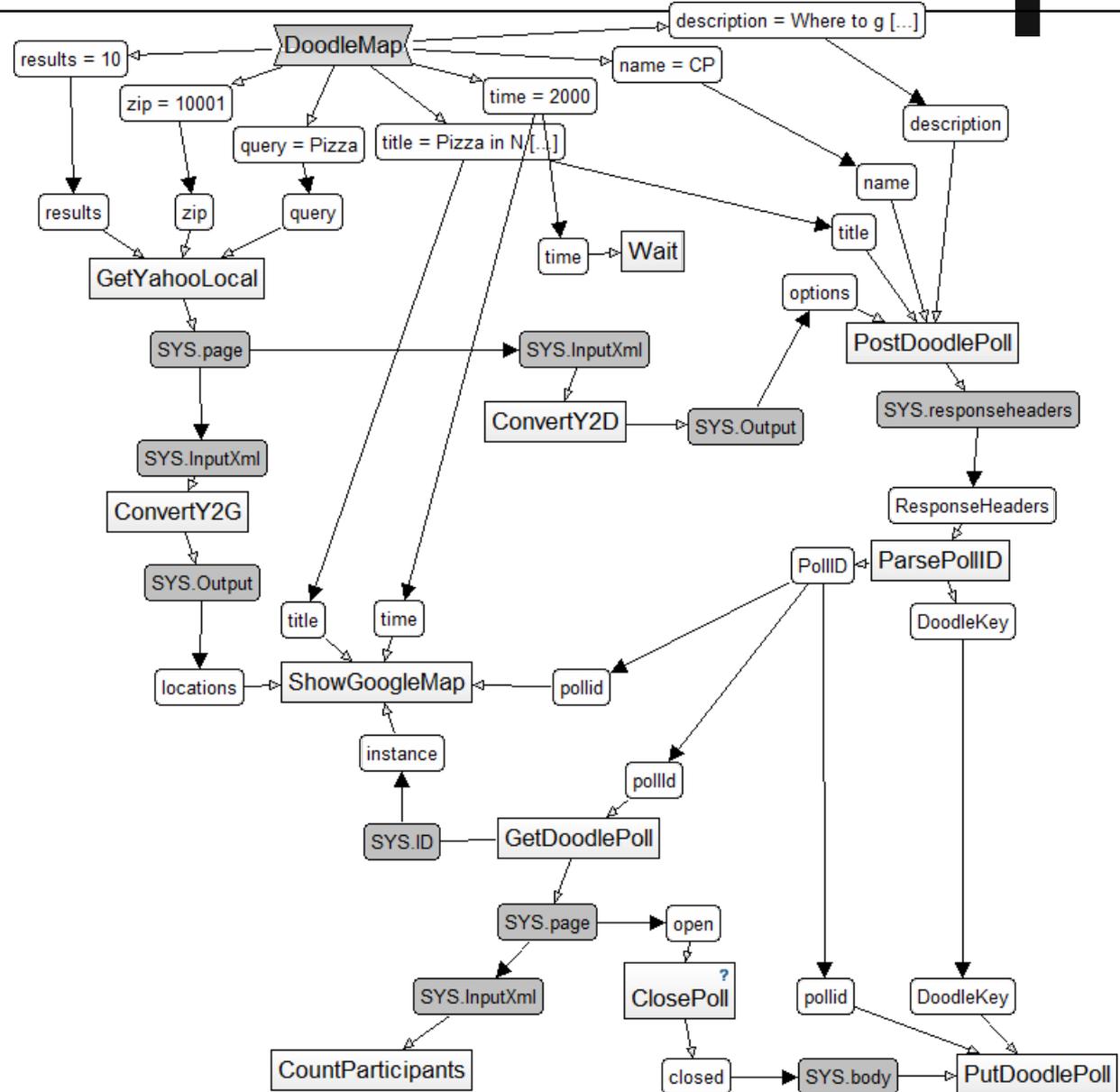
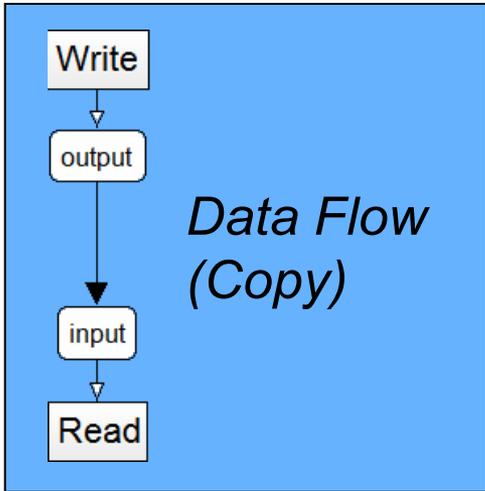
XPATH

JAVA

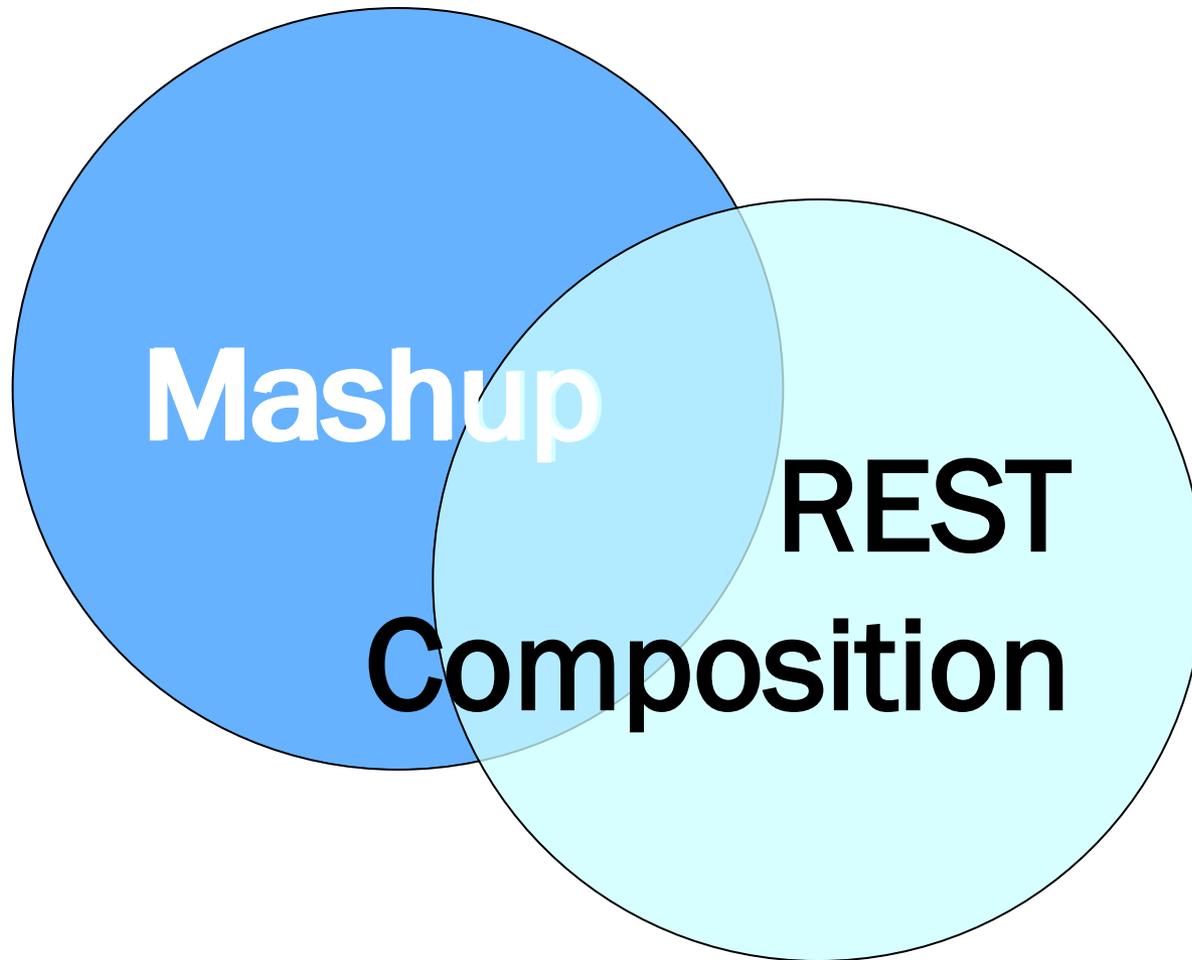
...



Data Flow

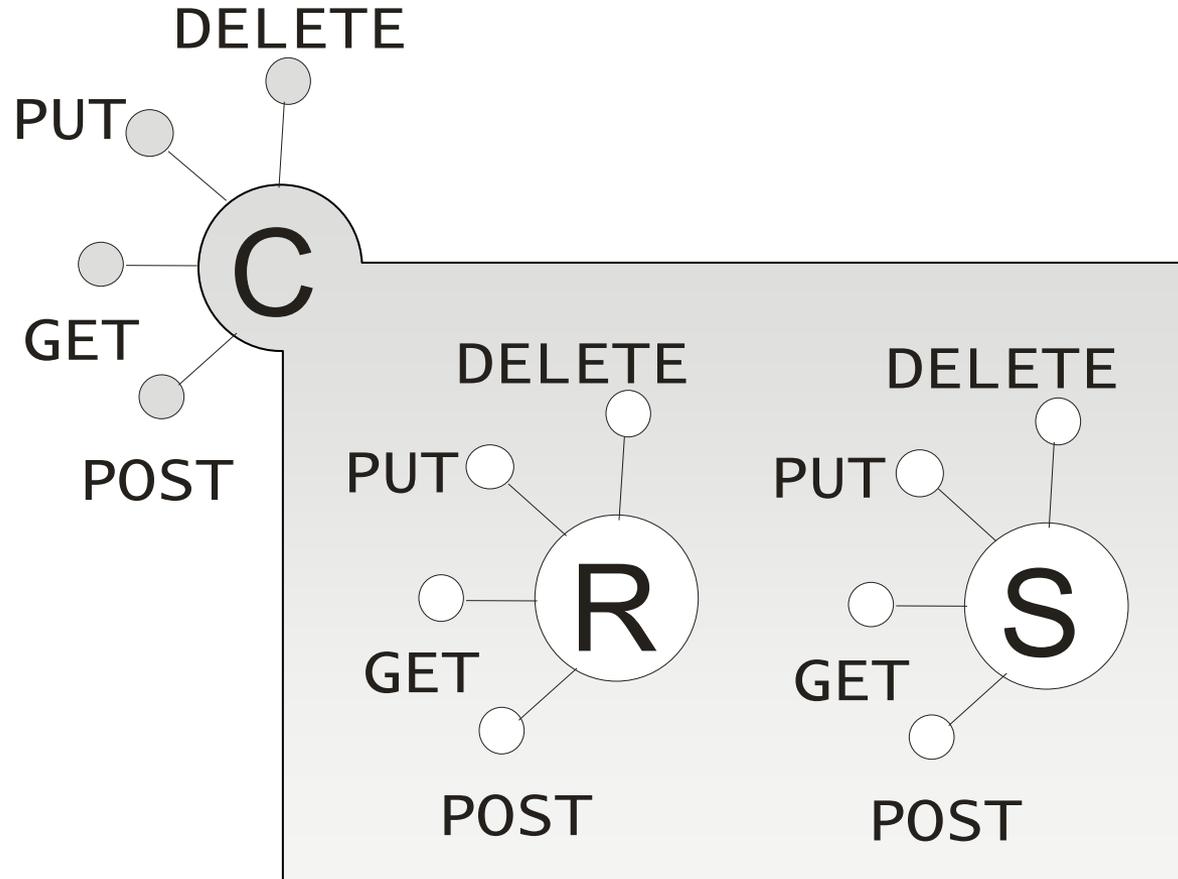


Was it just a mashup?

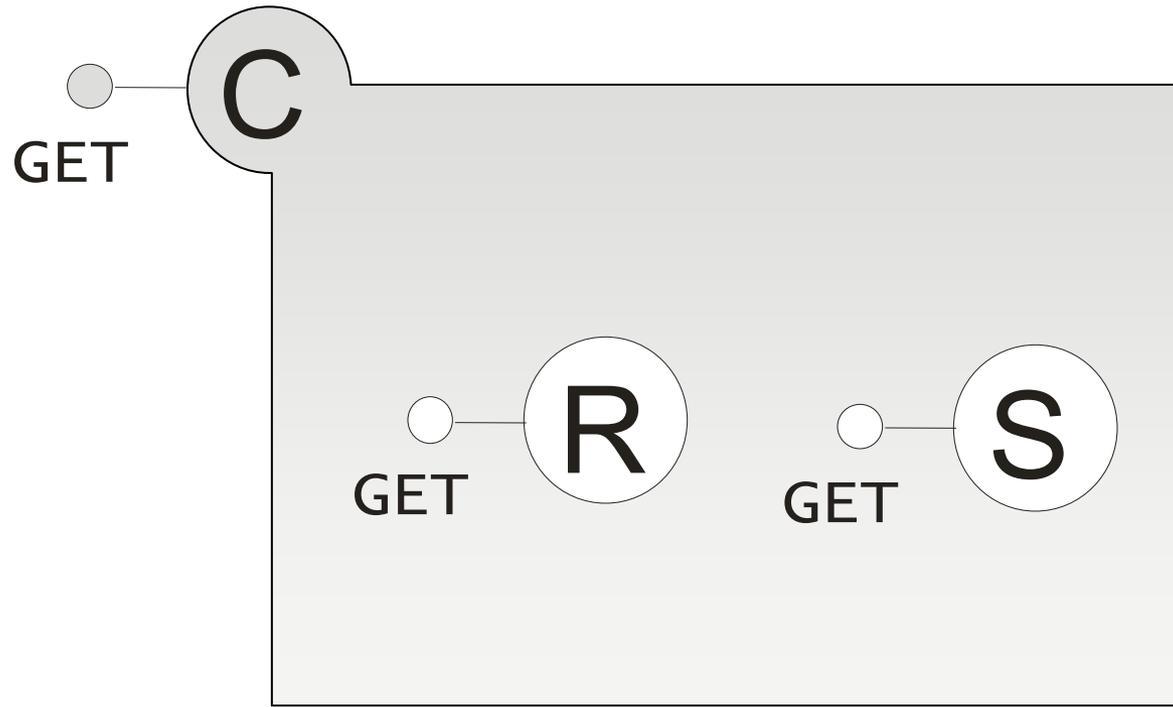


(It depends on the definition of Mashup)

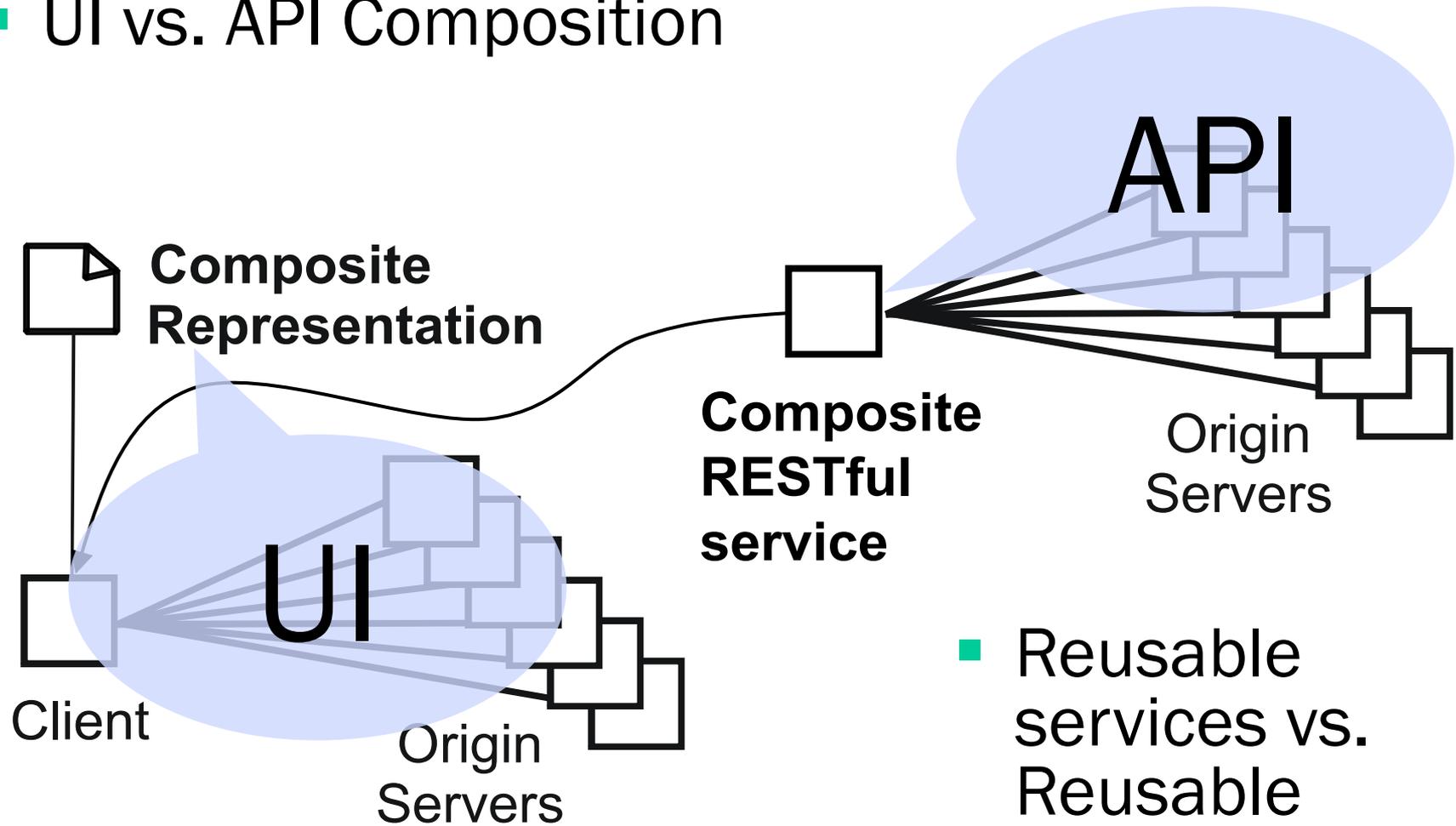
- Read-only vs. Read/Write



- Read-only vs. Read/write

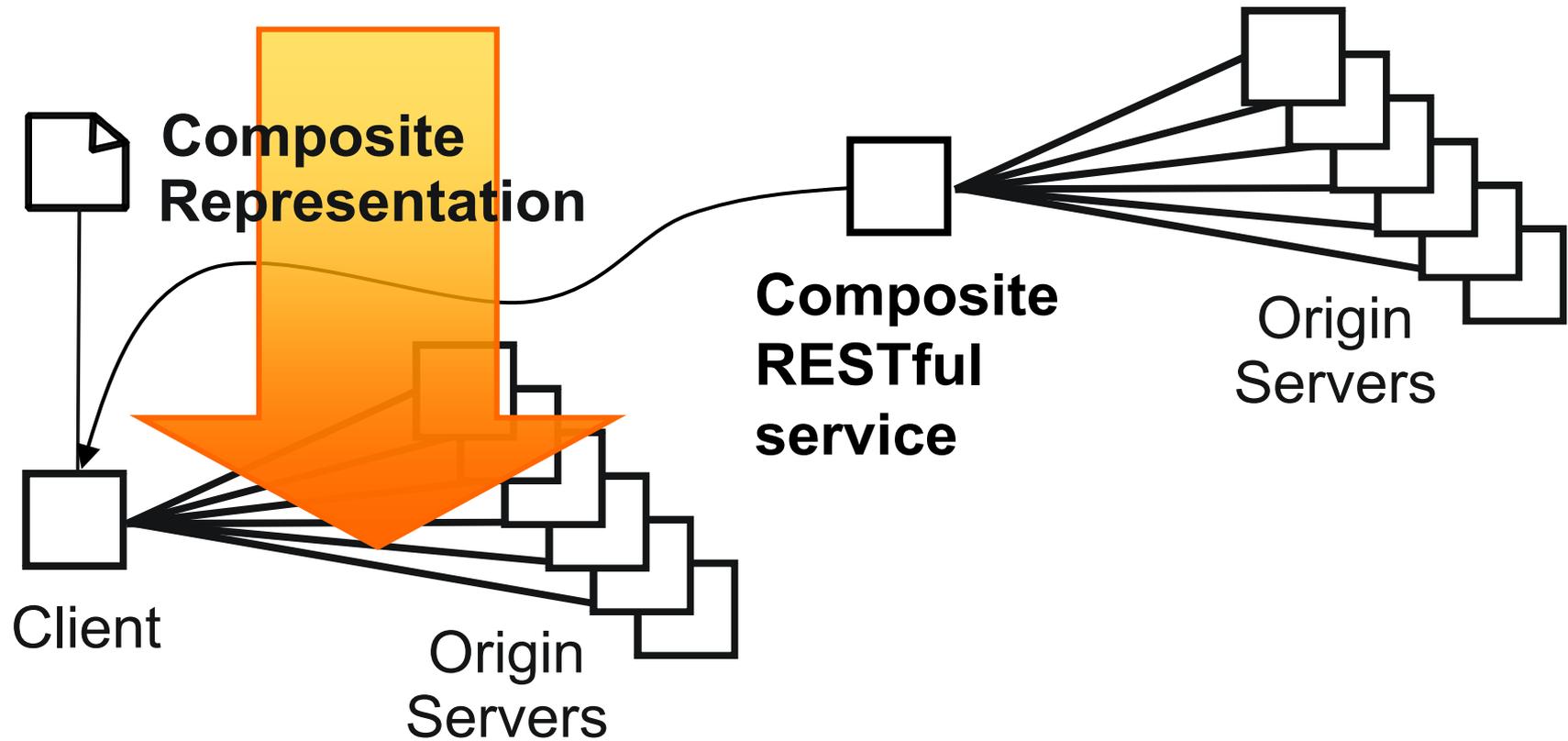


- UI vs. API Composition

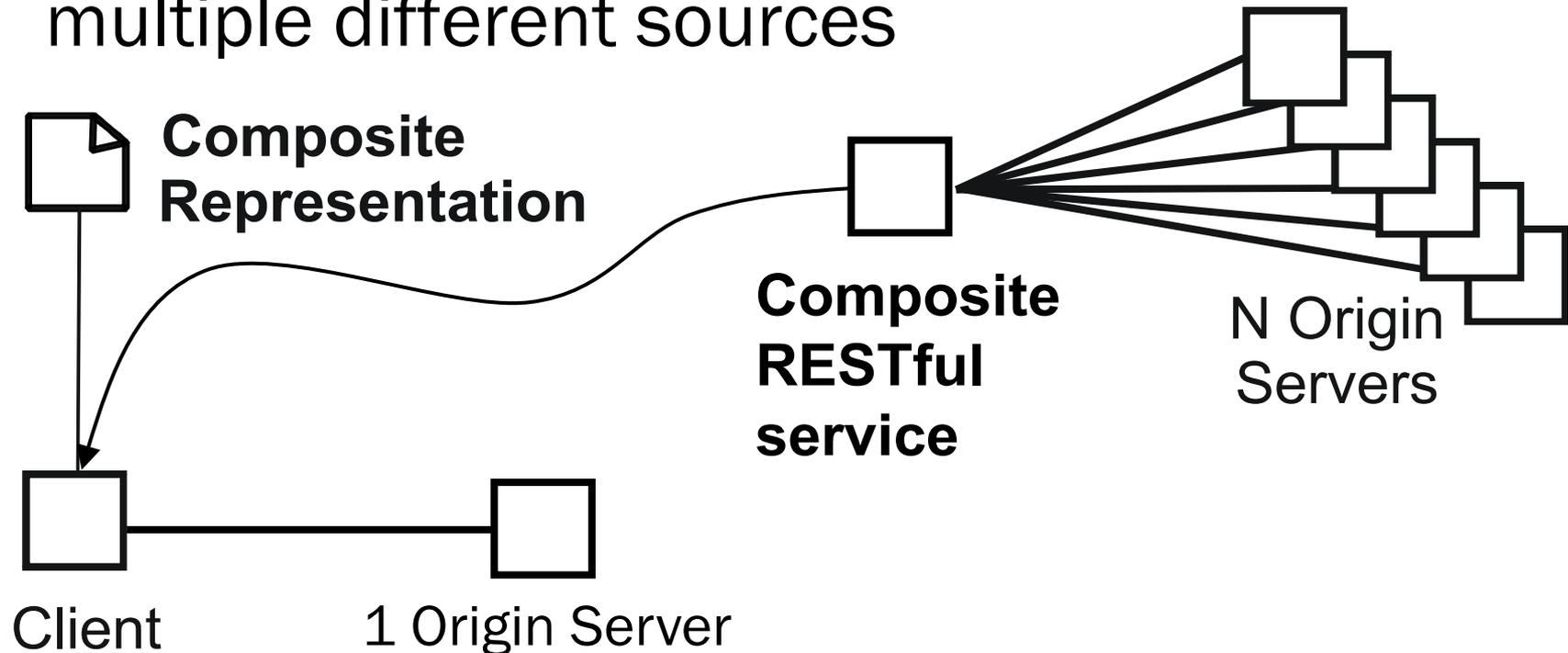


- Reusable services vs. Reusable Widgets

- Can you always do this from a web browser?



- Security Policies on the client may not always allow it to aggregate data from multiple different sources



- This will change very soon with HTML5

Read-Only

Read/Write

UI

Mashup

REST

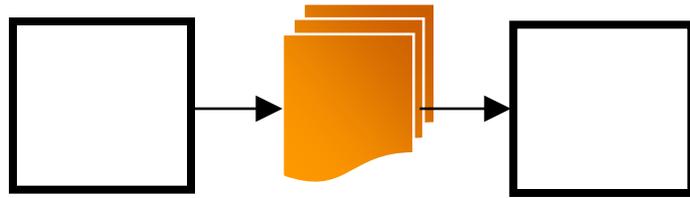
API

Composition

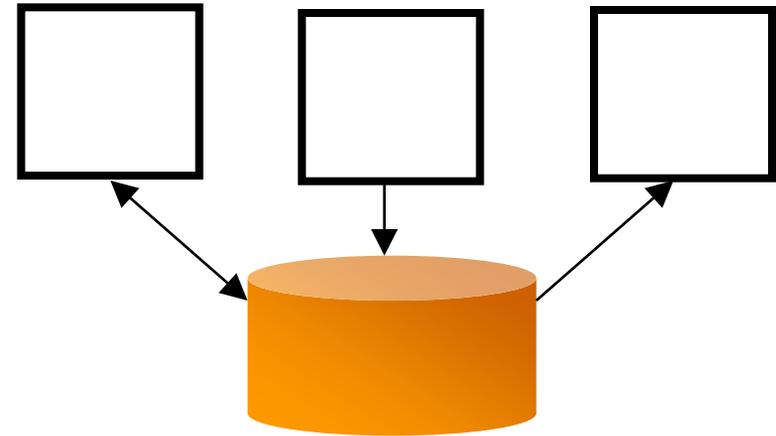
Situational
Sandboxed

Reusable
Service

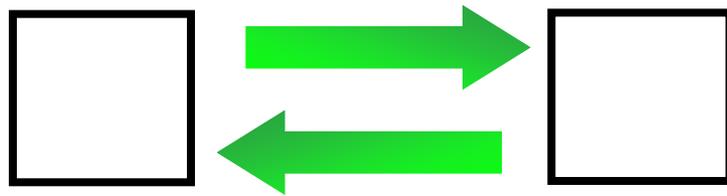
- REST brings a new perspective and new problems to service composition
- RESTful services can be composed on the server by defining composite resources and on the client with composite representations
- Composing RESTful services helps to put the integration logic of a mashup into a reusable service API and keep it separate from its UI made out of reusable widgets
- Business processes can be published on the Web as RESTful Services
- RESTful Web service composition is different than mashups, but both can be built using BPM tools like JOpera
- GET <http://www.jopera.org/>



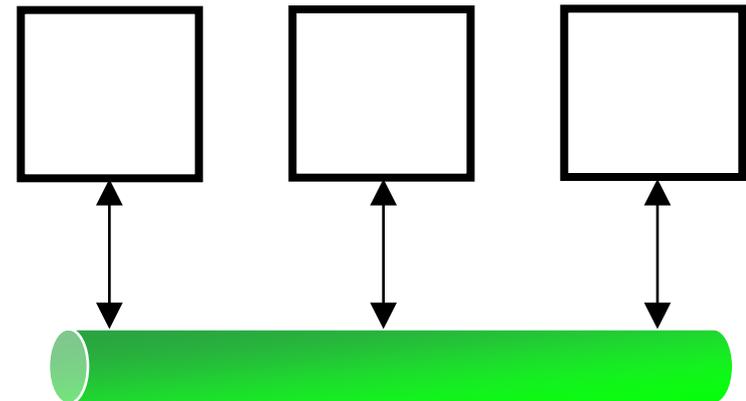
File Transfer



Shared Data

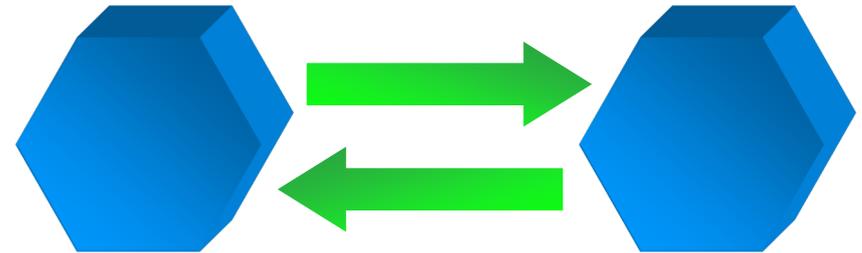


Procedure Call
Remote Procedure Call



Message Bus
Events

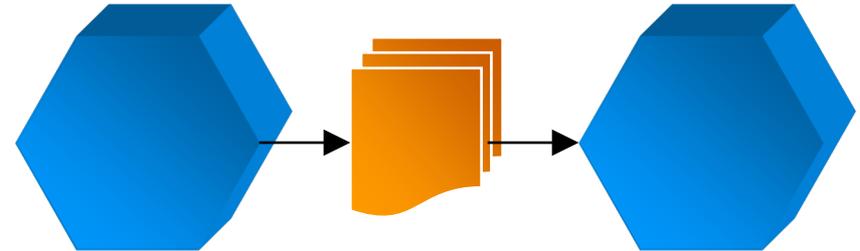
Call



Remote Procedure Call

- Procedure/Function Calls are the easiest to program with.
- They take a basic programming language construct and make it available across the network (Remote Procedure Call) to connect distributed components
- Remote calls are often used within the client/server architectural style, but call-backs are also used in event-oriented styles for notifications

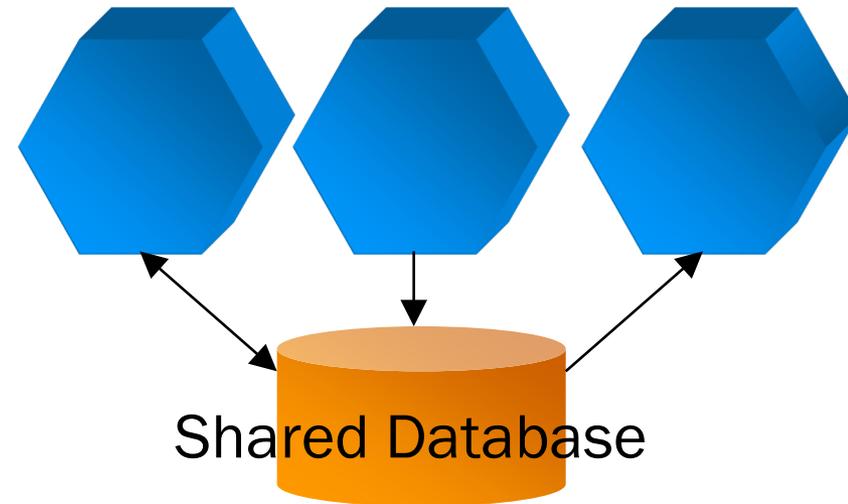
Write
Copy
Watch
Read



File Transfer
(Hot Folder)

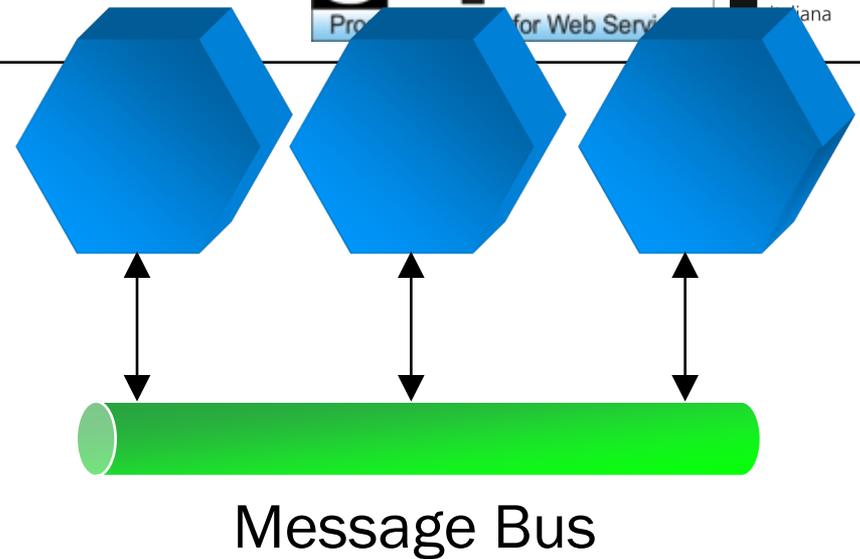
- Transferring files does not require to modify components
- A component writes a file, which is then copied on a different host, and fed as input into a different component
- The transfers can be batched with a certain frequency

Create
Read
Update
Delete

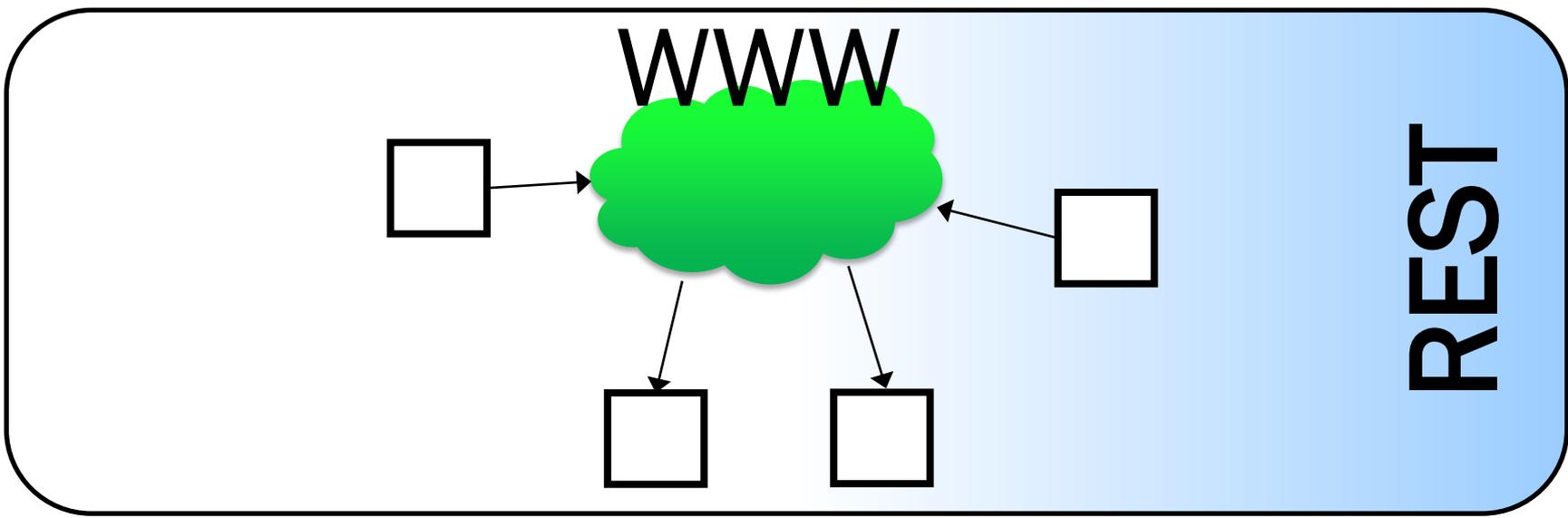
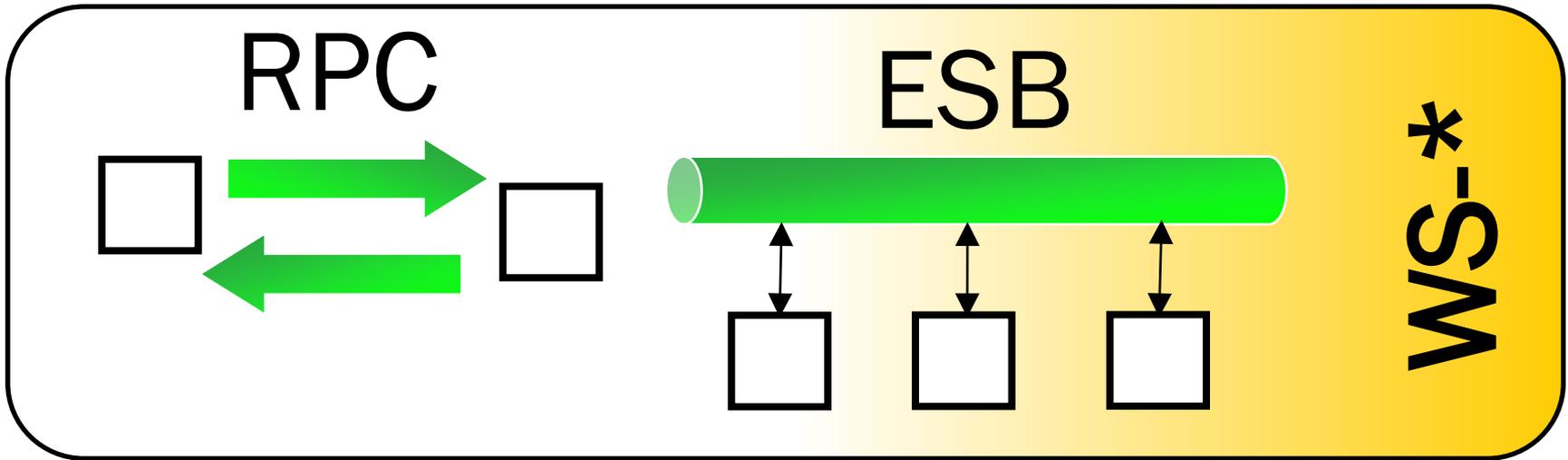


- Sharing a common database does not require to modify components, if they all can support the same schema
- Components can communicate by creating, updating and reading entries in the database, which can safely handles the concurrency

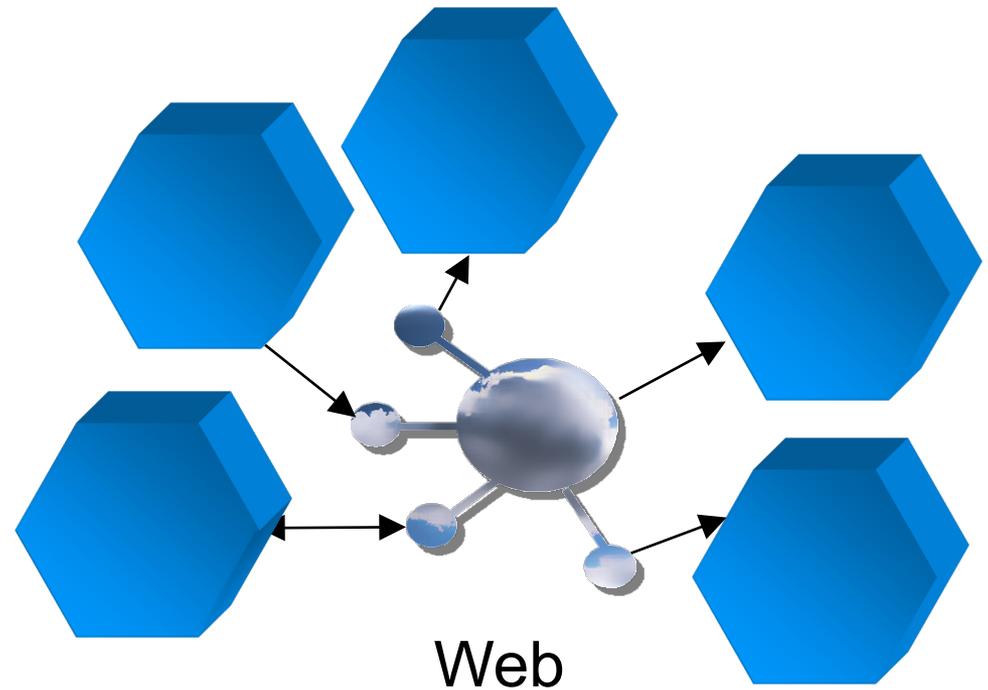
Publish Subscribe



- A message bus connects a variable number of components, which are decoupled from one another.
- Components act as message sources by publishing messages into the bus; Components act as message sinks by subscribing to message types (or properties based on the actual content)
- The bus can route, queue, buffer, transform and deliver messages to one or more recipients
- The “enterprise” service bus is used to implement the SOA style



Get Put Delete Post



- The Web is the connector used in the REST (Representational State Transfer) architectural style
- Components may reliably transfer state among themselves using the GET, PUT, DELETE primitives. POST is used for unsafe interactions.

- You should focus on whatever solution gets the job done and try to **avoid being religious** about any specific architectures or technologies.
- WS-* has 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 application requirements and constraints.
- We hope this comparison will help you make the right choice.

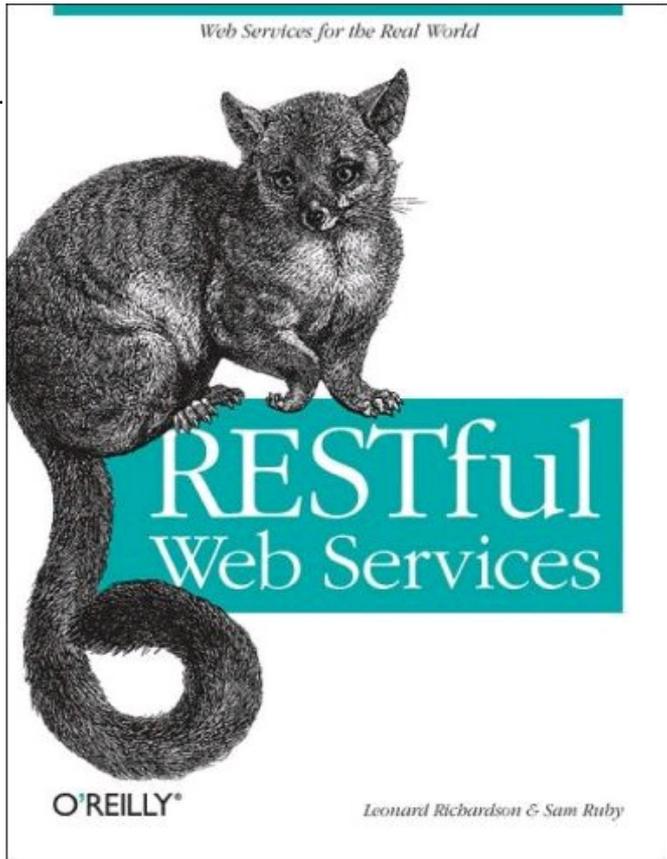
- Roy Fielding, [Architectural Styles and the Design of Network-based Software Architectures](#), PhD Thesis, University of California, Irvine, 2000
- Leonard Richardson, Sam Ruby, **RESTful Web Services**, O'Reilly, May 2007
- Jim Webber, Savas Parastatidis, Ian Robison, **REST in Practice: Hypermedia and Systems Architecture**, O'Reilly, 2010
- Subbu Allamaraju, **RESTful Web Services Cookbook: Solutions for Improving Scalability and Simplicity**, O'Reilly, 2010
- Stevan Tilkov, **HTTP und REST**, dpunkt Verlag, 2009, <http://rest-http.info/>

- Martin Fowler,
Richardson Maturity Model: steps toward the glory of REST,
<http://martinfowler.com/articles/richardsonMaturityModel.html>
- My Constantly Updated Feed or REST-related material:
<http://delicious.com/cesare.pautasso/rest>
- This week in REST
<http://thisweekinrest.wordpress.com/>

ws://rest.2010

First International Workshop on RESTful Design

- Cesare Pautasso, Olaf Zimmermann, Frank Leymann, [RESTful Web Services vs. Big Web Services: Making the Right Architectural Decision](#), Proc. of the 17th International World Wide Web Conference ([WWW2008](#)), Beijing, China, April 2008.
- Cesare Pautasso and Erik Wilde. [Why is the Web Loosely Coupled? A Multi-Faceted Metric for Service Design](#), Proc of the 18th International World Wide Web Conference ([WWW2009](#)), Madrid, Spain, April 2009.
- Cesare Pautasso, [BPEL for REST](#), Proc. of the 6th International Conference on Business Process Management ([BPM 2008](#)), Milan, Italy, September 2008.
- Cesare Pautasso, [RESTful Web Service Composition with JOpera](#), Proc. Of the International Conference on Software Composition (SC 2009), Zurich, Switzerland, July 2009.
- Cesare Pautasso, Gustavo Alonso: **From Web Service Composition to Megaprogramming** In: Proceedings of the 5th VLDB Workshop on Technologies for E-Services (TES-04), Toronto, Canada, August 2004
- Thomas Erl, Raj Balasubramanians, Cesare Pautasso, Benjamin Carlyle, **SOA with REST**, Prentice Hall, end of 2010



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



Thomas Erl, Raj Balasubramanians,
Cesare Pautasso, Benjamin Carlyle,
SOA with REST,
Prentice Hall, end of 2010



ECOWS'10

8th European Conference on Web Services
Ayia Napa, Cyprus
December 1-3, 2010

<http://www.cs.ucy.ac.cy/ecows10>
<http://twitter.com/ecows2010>

Abstract Submission: Friday, July **16**, 2010