Università
della
Svizzera
italiana

**Faculty
of Informatics**

UC Berkeley
School of Information

RESTful Web Services:

Principles, Patterns, Emerging Technologies

Cesare Pautasso, Erik Wilde

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

dret@berkeley.edu
http://dret.net/netdret

Overview

9:00-10:30	1. What is REST?
11:00-12:30	2. RESTful Service Design
14:00-15:30	3. REST vs. WS-*
16:00-17:00	4. REST Composition
17:00-17:30	5. REST in Practice

Università
della
Svizzera
italiana

Faculty
of Informatics

UC Berkeley
School of Information

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

REST Design - Outline

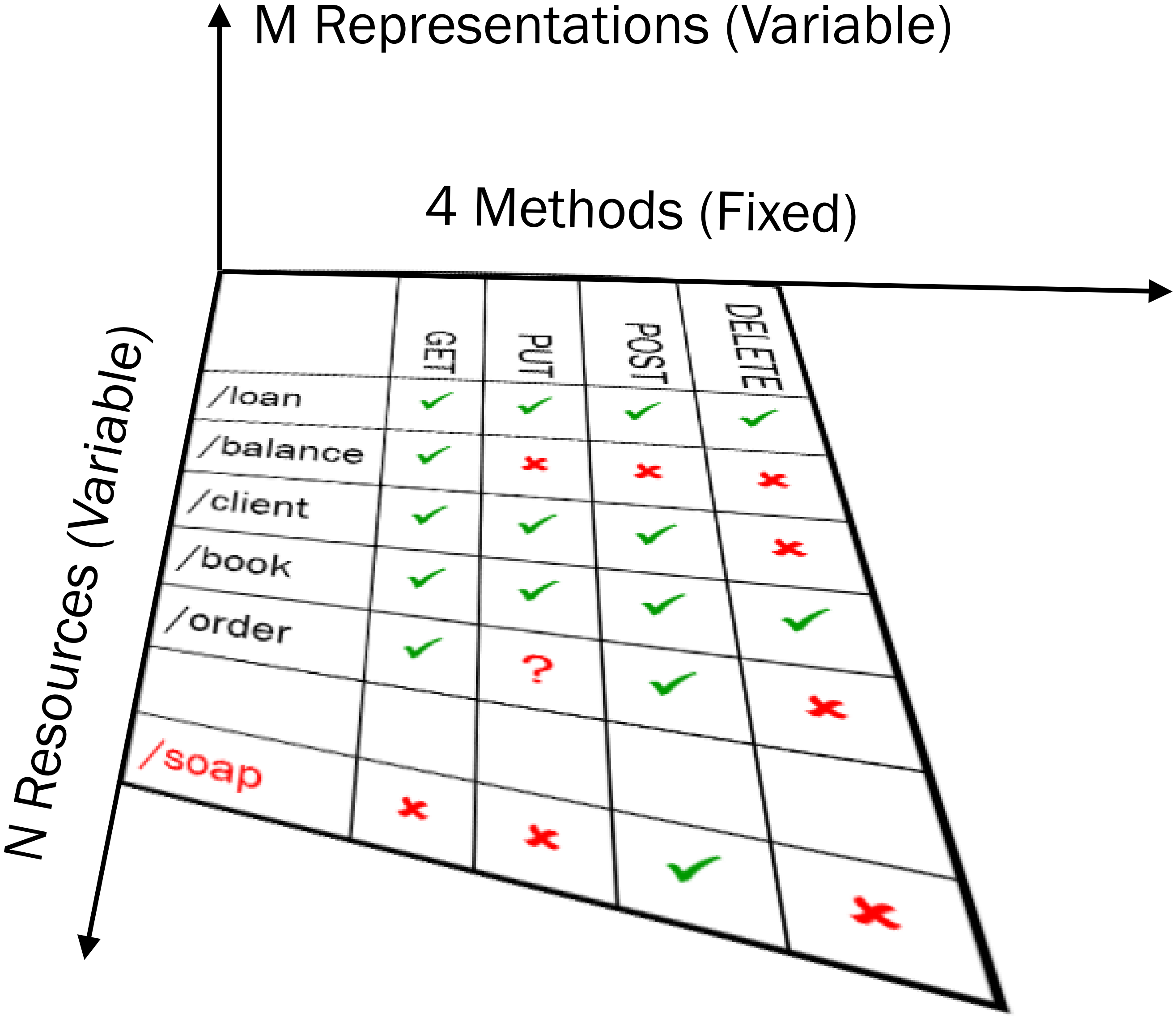
- Design Methodology
- Simple Doodle Service Example
- Design Tips
    - Is URI Design part of REST?
    - Understanding GET vs. POST vs. PUT
    - Multiple Representations
        - Content-Type Negotiation
    - Media Type Design
    - Exception Handling
        - Idempotent vs. Unsafe
        - Dealing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

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. 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



M Representations (Variable)

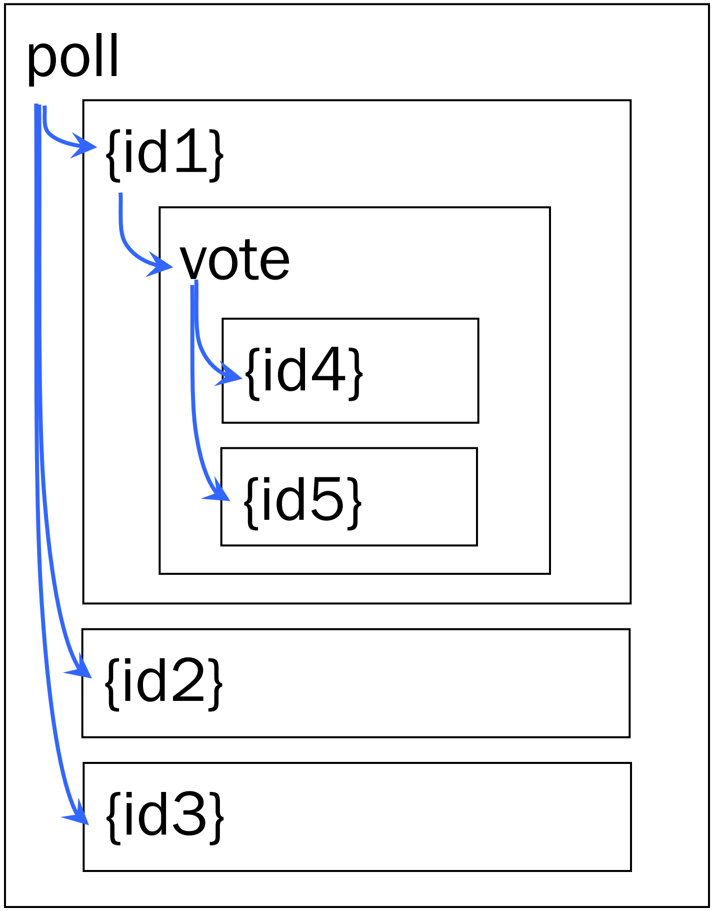
4 Methods (Fixed)

N Resources (Variable)

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

Simple Doodle API Example Design

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



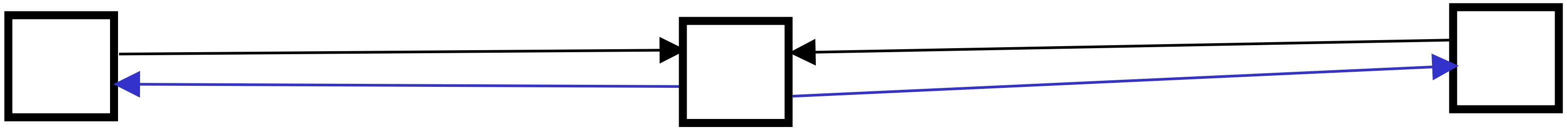
	GET	PUT	POST	DELETE
/poll	✓	✗	✓	✗
/poll/*{id}*	✓	✓	✗	✓
/poll/*{id}*/vote	✓	✗	✓	✗
/poll/*{id}*/vote/*{id}*	✓	✓	✗	?

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)

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



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

201 Created
Location: /poll/090331x
```

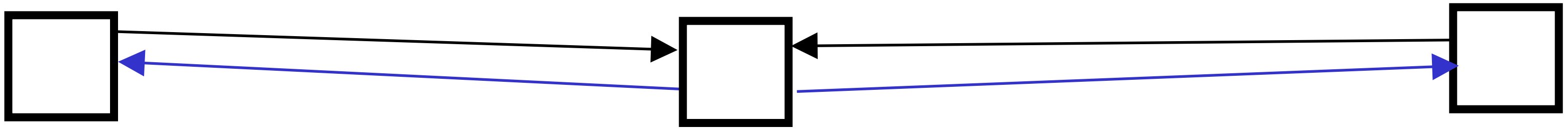
```
GET /poll/090331x

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

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

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
```



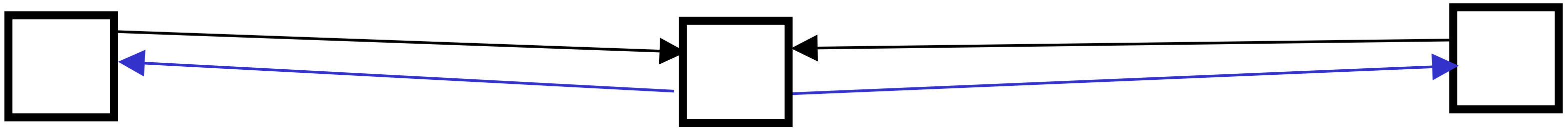
```
POST /poll/090331x/vote
<name>C. Pautasso</name>
<choice>B</choice>

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)

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



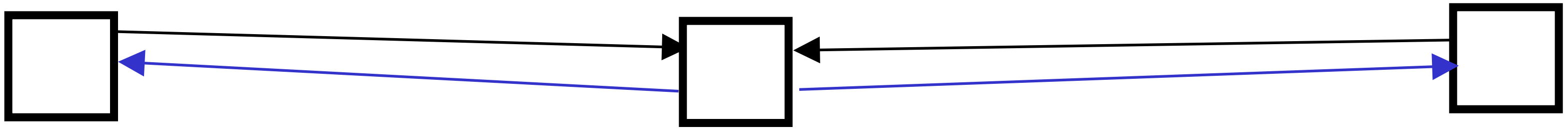
```
PUT /poll/090331x/vote/1
<name>C. Pautasso</name>
<choice>C</choice>

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>
```

Simple Doodle API Example

- Polls can be deleted once a decision has been made

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



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

200 OK                      404 Not Found
```

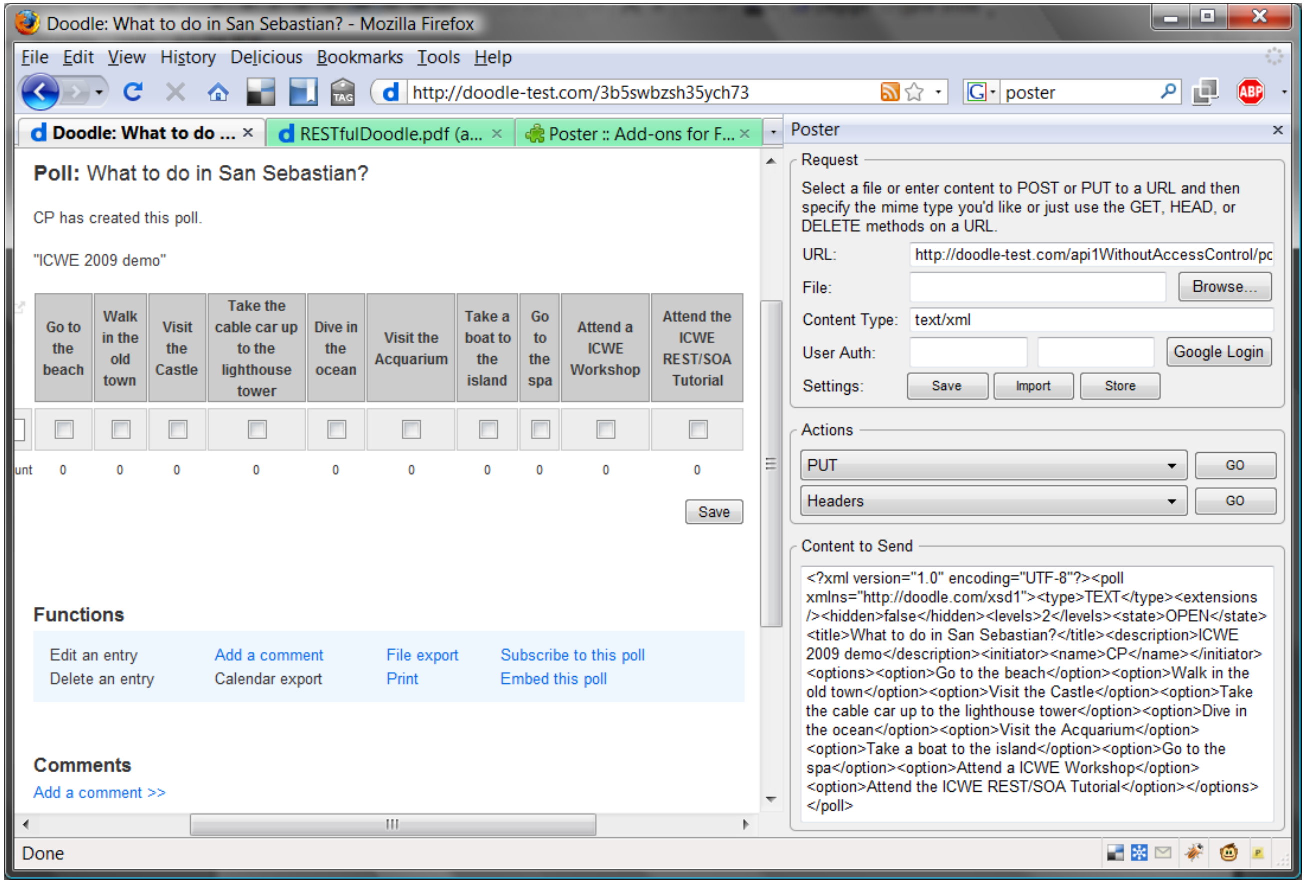
Real Doodle Demo

- 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


Doodle: What to do in San Sebastian? - Mozilla Firefox
File Edit View History Delicious Bookmarks Tools Help
http://doodle-test.com/3b5swbzsh35ych73
poster
Doodle: What to do ...
RESTfulDoodle.pdf (a...
Poster :: Add-ons for F...
Poster
Done

Poll: What to do in San Sebastian?

CP has created this poll.

"ICWE 2009 demo"

	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
□	□	□	□	□	□	□	□	□	□	□
unt	0	0	0	0	0	0	0	0	0	0

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 >>

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 GO
Headers 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

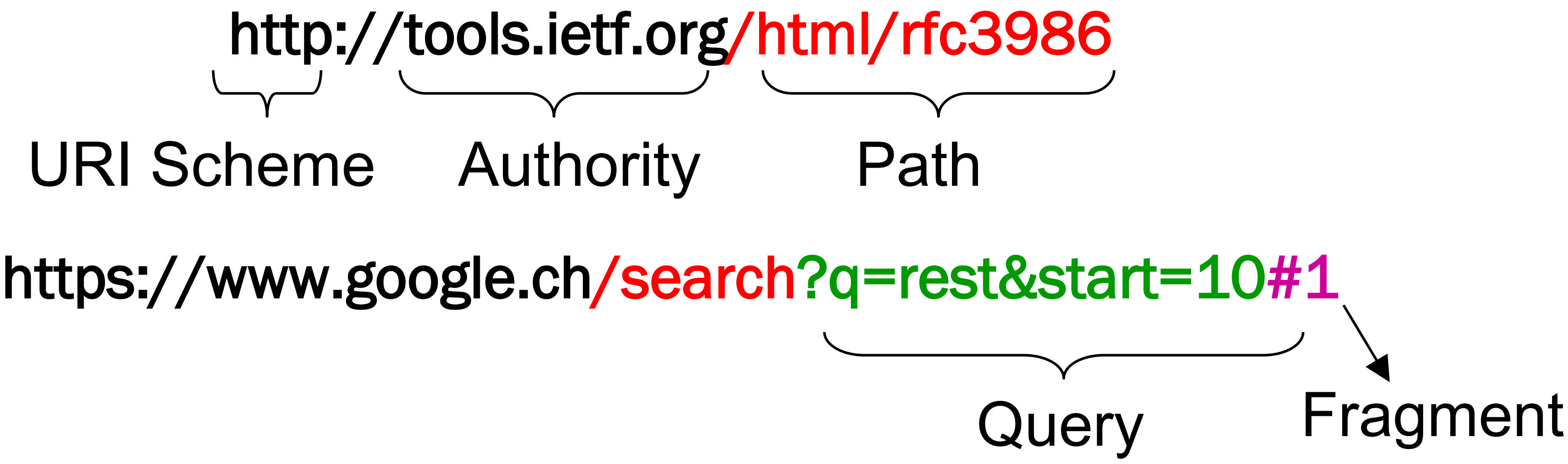
```
POST http://doodle-test.com/api1WithoutAccessControl/polls/
Content-Type: text/xml

<?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</requireEMail><requirePhone>false</requirePhone><byInvitationOnly>false</byInvitationOnly><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</eMailAddress></initiator><options><option>too fast</option><option>right speed</option><option>too slow</option></options><participants></participants><comments></comments></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: text/xml

<participant xmlns="http://doodle.com/xsd1"><name>Cesare
    Pautasso</name><preferences><option>0</option><option>1</option><
    option>0</option></preferences></participant>
```

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

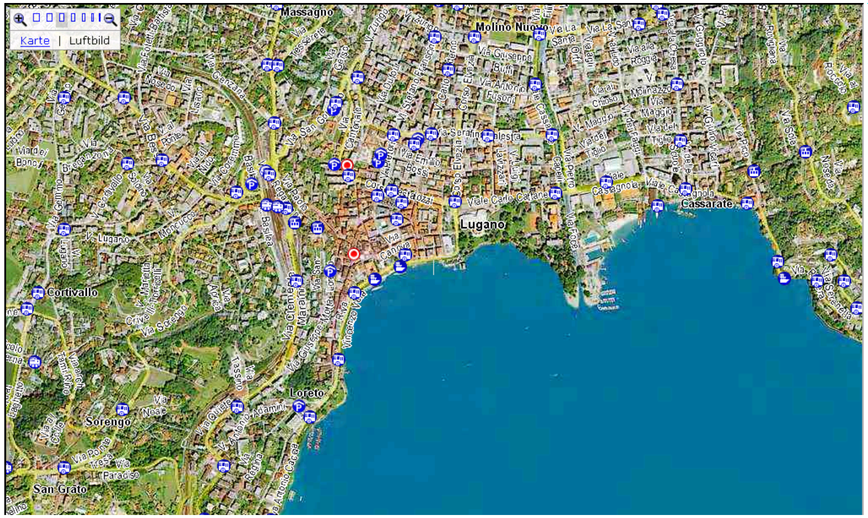
Query
Fragment

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

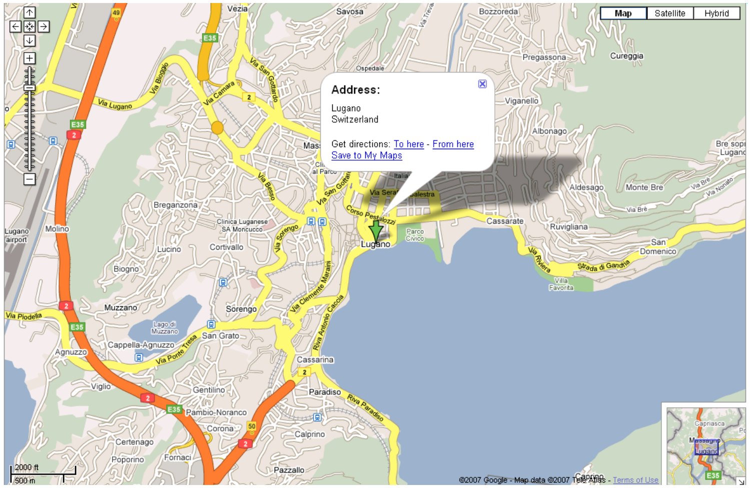
What is a “nice” URI?

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

http://map.search.ch/lugano



http://maps.google.com/lugano



Address:
Lugano
Switzerland

Get directions: To here - From here
Save to My Maps

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)

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*

- Some use URI postfixes to specify the content type

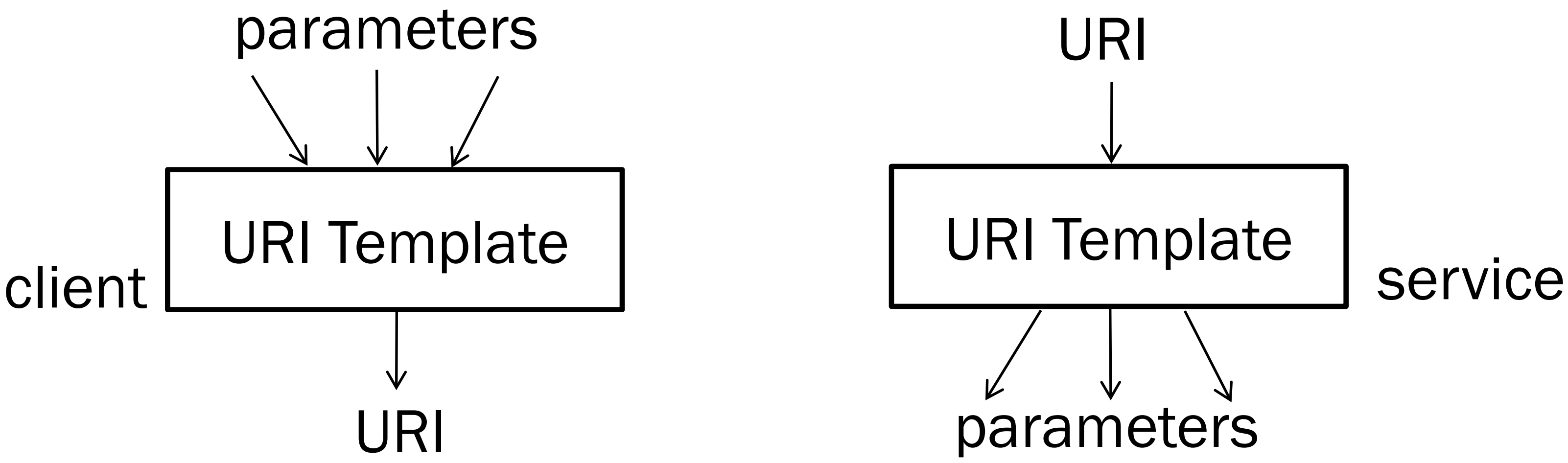
- *This may break the abstraction*

- Do not change URIs
- Use redirection if you really need to change them

- **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

Uniform Interface Constraint

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

HTML5 Forms

- HTML4/XHTML
- <form method="GET|POST">

- HTML5
- <form method="GET|POST|PUT|DELETE">

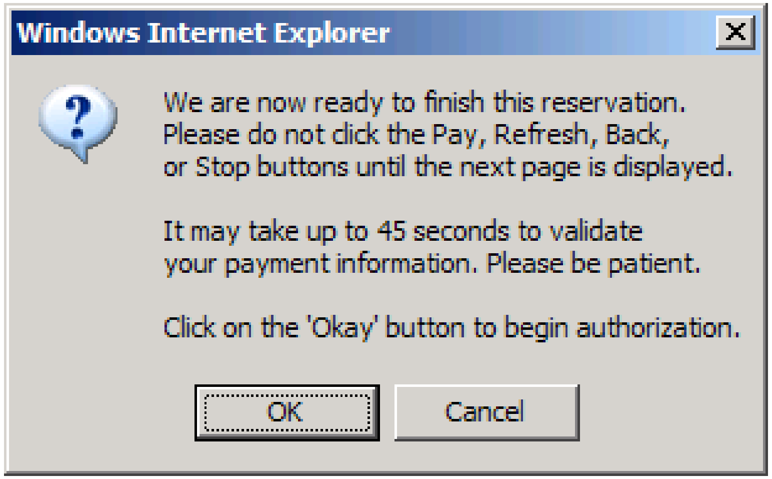
- http://www.w3.org/TR/html5/forms.html#attr-fs-method

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.

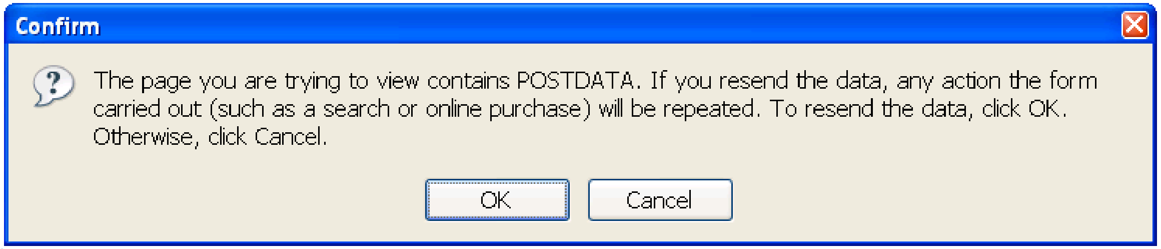


We are now ready to finish this reservation. Please do not click the Pay, Refresh, Back, or Stop buttons until the next page is displayed.

It may take up to 45 seconds to validate your payment information. Please be patient.

Click on the 'Okay' button to begin authorization.

Web browsers warn you when refreshing a page generated with POST



The page you are trying to view contains POSTDATA. If you resend the data, any action the form carried out (such as a search or online purchase) will be repeated. To resend the data, click OK. Otherwise, click Cancel.

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)

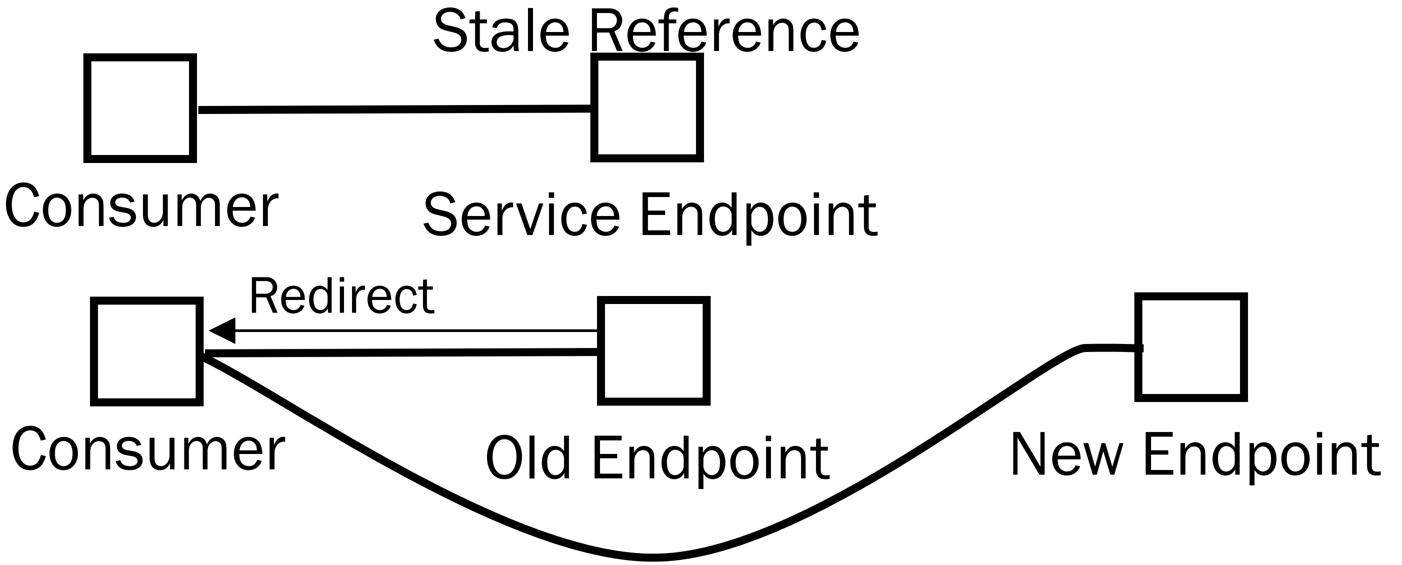
Solution 1: let the client choose a unique id (e.g., GUID)

```
POST /resource
301 Moved Permanently
Location: /resource/{id}
```

Solution 2: let the server compute the unique id

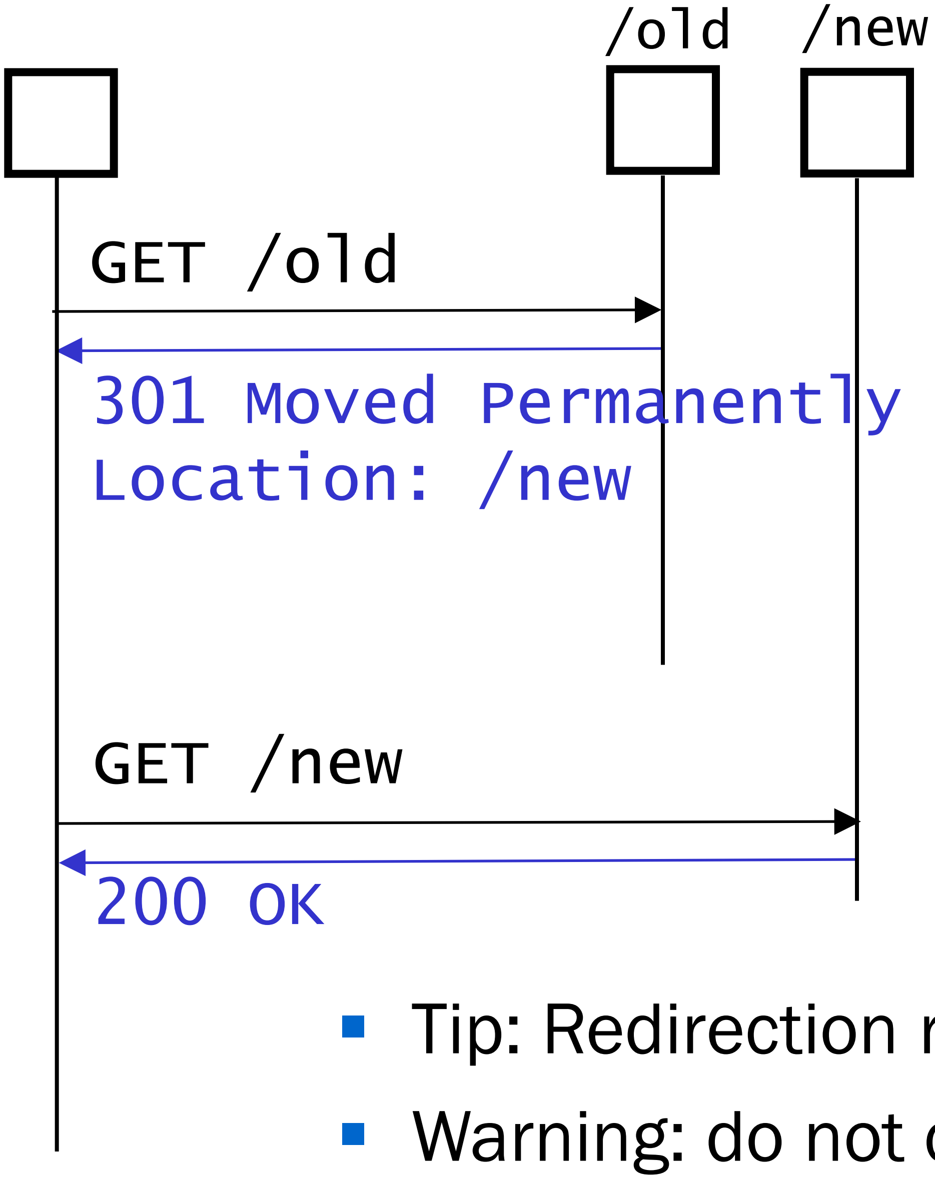
Problem: Duplicate instances may be created if requests are repeated due to unreliable communication

Redirection for Smooth Evolution



- How can consumers of a RESTful service adapt when service locations and URIs are restructured?
- Problem: Service URIs may change over time for business or technical reasons. It may not be possible to replace all references to old links simultaneously risking to introduce broken links.
- Solution: Automatically refer service consumers that access the old identifier to the current identifier.

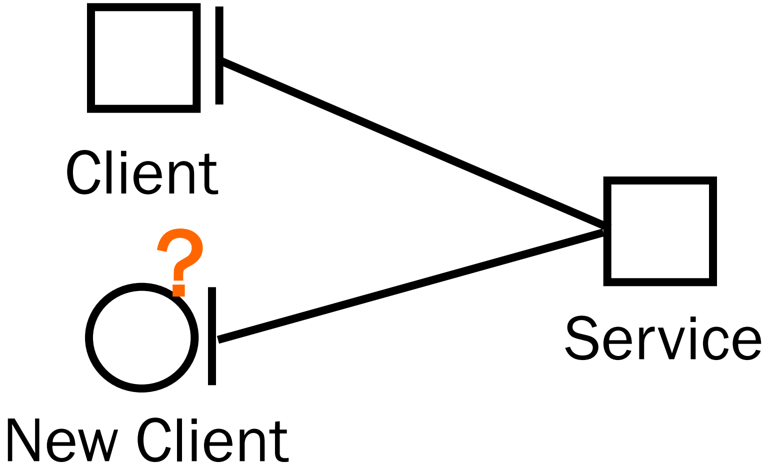
Redirection with HTTP



- HTTP natively supports redirection using a combination of 3xx status codes and standard headers:
  - 301 Moved Permanently
  - 307 Temporary Redirect
  - Location: /newURI

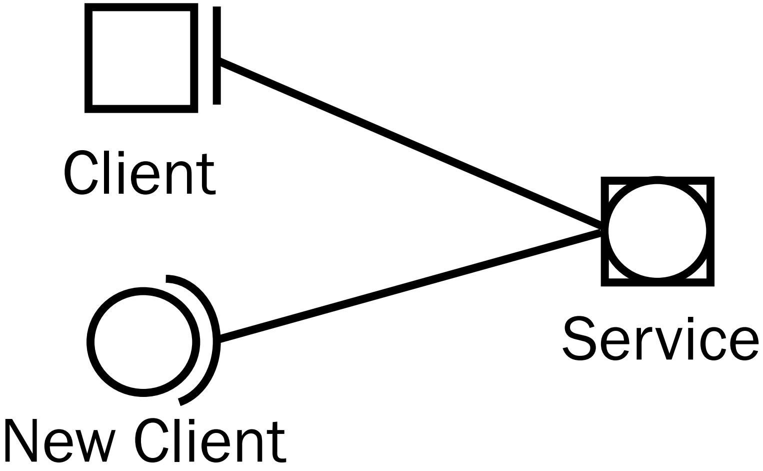
- Tip: Redirection responses can be chained.
- Warning: do not create redirection loops!

Should all agree on the same format?



- How can services support different consumers which make different assumptions about the messaging format?
- Problem: Service consumers may change their requirements in a way that is not backwards compatible. ***A service may have to support both old and new consumers*** without having to introduce a specific interface for each kind of consumer.

Solution: Content Negotiation



- Solution: specific content and data representation formats to be accepted or returned by a service capability is negotiated at runtime as part of its invocation. The service contract refers to multiple standardized "media types".
- Benefits: Loose Coupling, Increased Interoperability, Increased Organizational Agility

Content Negotiation in HTTP

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 (status 406 if none can be found)

Advanced Content Negotiation

Quality factors allow the client to indicate the relative degree of preference for each representation (or media-range).

```
Media/Type; q=X
```

If a media type has a quality value q=0, then content with this parameter is not acceptable for the client.

```
Accept: text/html, text/*; q=0.1
```

The client prefers to receive HTML (but any other text format will do with lower priority)

```
Accept: application/xhtml+xml; q=0.9,
  text/html; q=0.5, text/plain; q=0.1
```

The client prefers to receive XHTML, or HTML if this is not available and will use Plain Text as a fall back

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?

Multi-Dimensional Negotiation

Content Negotiation is very flexible and can be performed based on different dimensions (each with a specific pair of HTTP headers).

Request Header	Example Values	Response Header

`Accept:`	`application/xml,`
	`application/json`
		`Content-Type:`

`Accept-Language:`	`en, fr, de, es`	`Content-Language:`

`Accept-Charset:`	`iso-8859-5,`
	`unicode-1-1`
		`Charset` parameter fo the `Content-Type` header

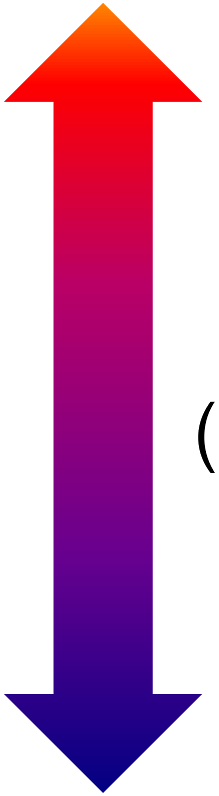
`Accept-Encoding:`	`compress,`
	`gzip`
		`Content-Encoding:`

Media Type Design

> *A REST API should spend almost all of its descriptive effort in defining the media type(s) used for representing resources and driving application state, or in defining extended relation names and/or hypertext-enabled mark-up for existing standard media types.*
>
> http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

- How to find the best media type?
- Reuse generic media types or invent custom/specific media types?
- Should you always standardize media types?

Media Type Design Trade Off



`text/xml`

(Generic, Reusable, Meaningless)

`application/atom+xml`

(Standardized, Reusable, Better Defined)

`application/vnd.my.type+xml`

(Specific, Less Reusable, Meaningful)

RFC4288 defines how to register custom media types.

List of existing standard media types:

http://www.iana.org/assignments/media-types/

Media Type Design Hints

- Reuse Existing Media Types
- Do not be afraid of inventing your own, but then standardize it and reuse it as much as possible
- Media Types capture the representation format of your resource information/data model and the implied processing model
- There is no best media type for a service, it all depends on what your clients need/support/understand
- Clients are not forced to process the media type as you expect them to

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

5xx Server's fault

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

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 (can be cached)

```
GET /book
```

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

```
Withdraw(200$) //unsafe
Deposit(200$)  //unsafe
```

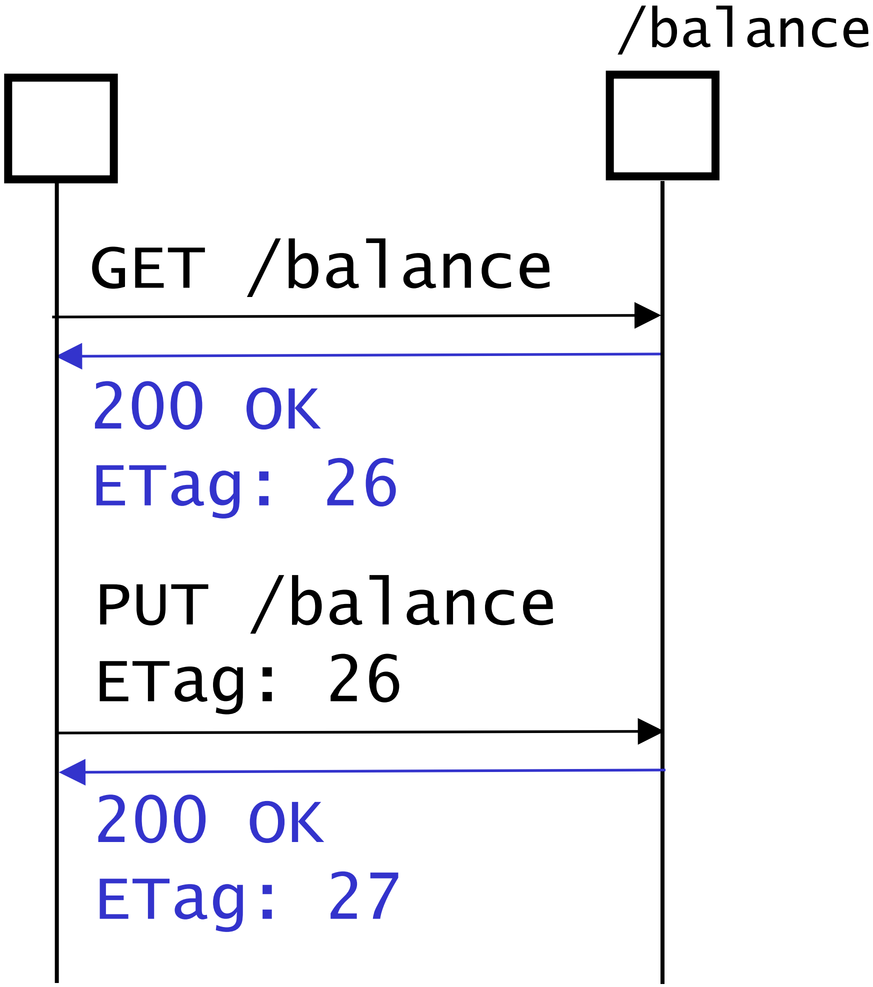
- Unsafe requests require special handling in case of exceptional situations (e.g., state reconciliation)

```
POST /order/x/payment
```

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

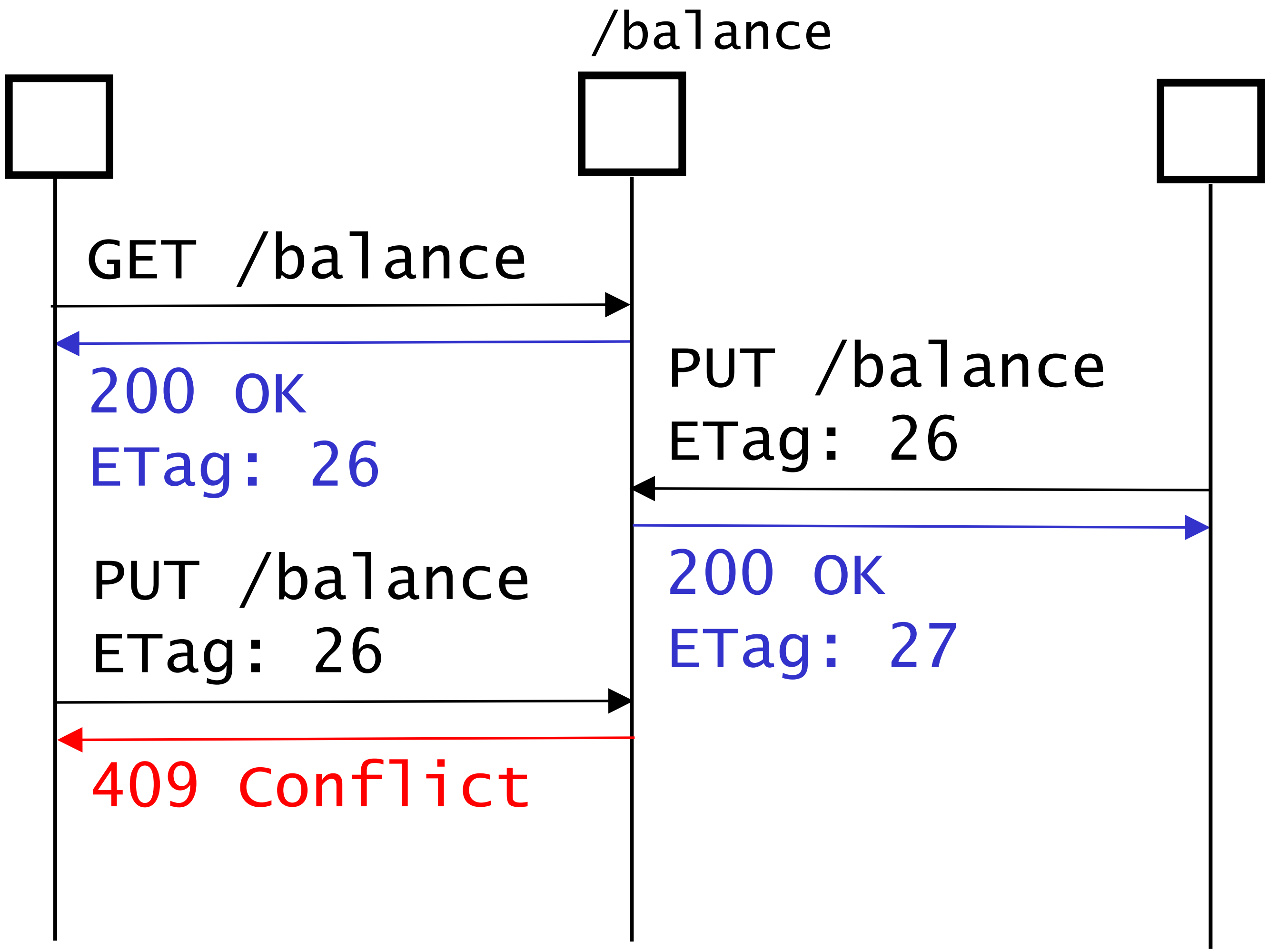
```
B = GetBalance() //safe
B = B + 200$     //local
SetBalance(B) //idempotent
```

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?
- Do we need to create an explicit `/balance/lock` resource? (Pessimistic Locking)
- Or is there an optimistic solution?

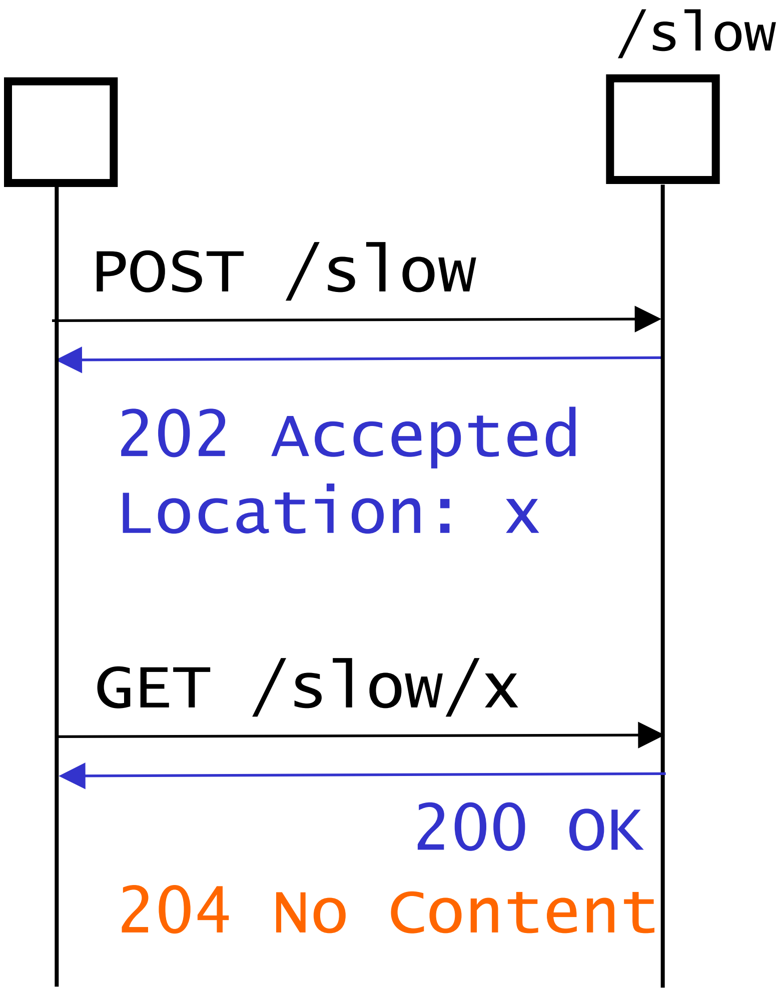
Dealing with Concurrency



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

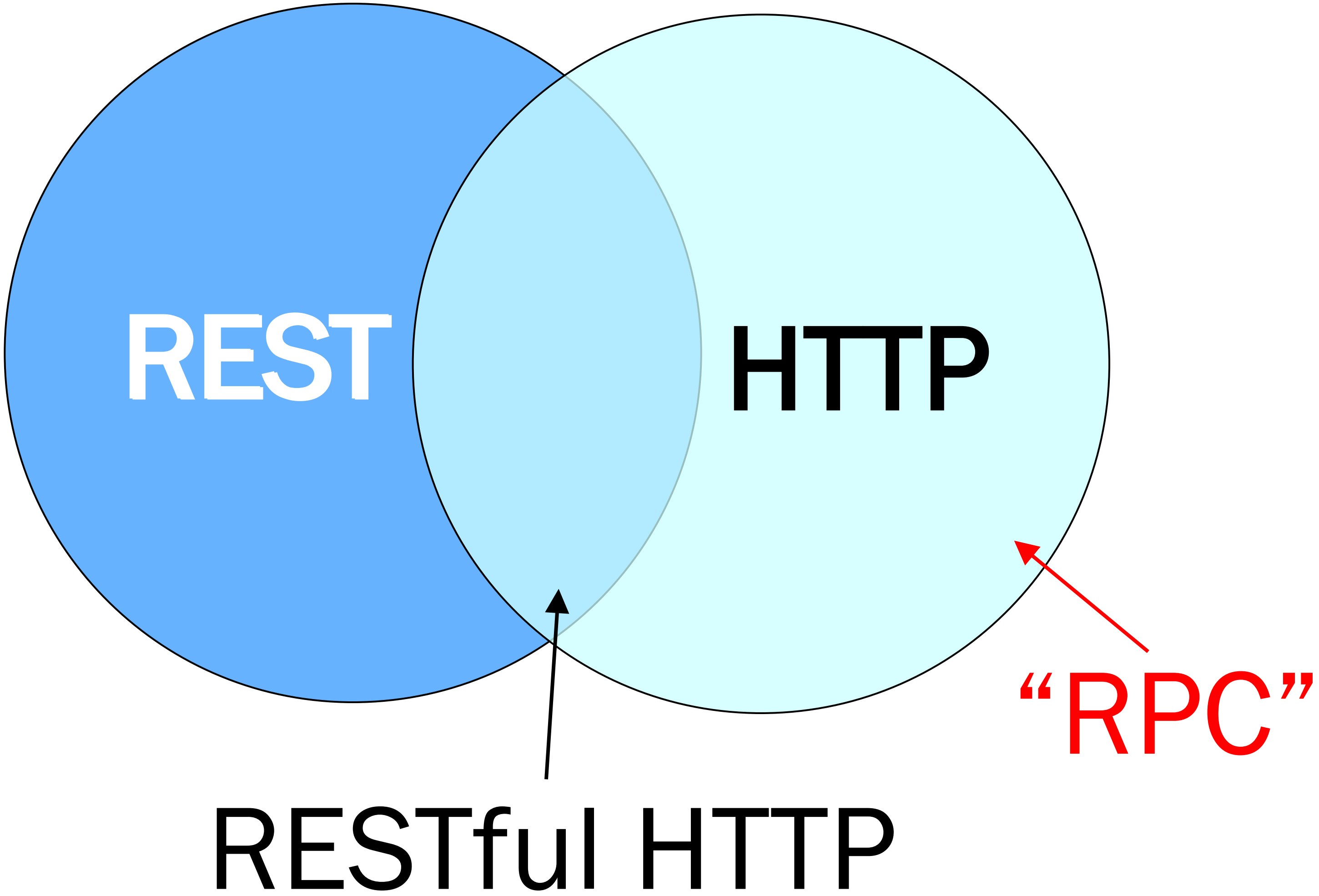
Blocking or Non-Blocking?

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

Antipatterns - REST vs. HTTP



REST Richardson Maturity Model

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?

Antipatterns – 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`)

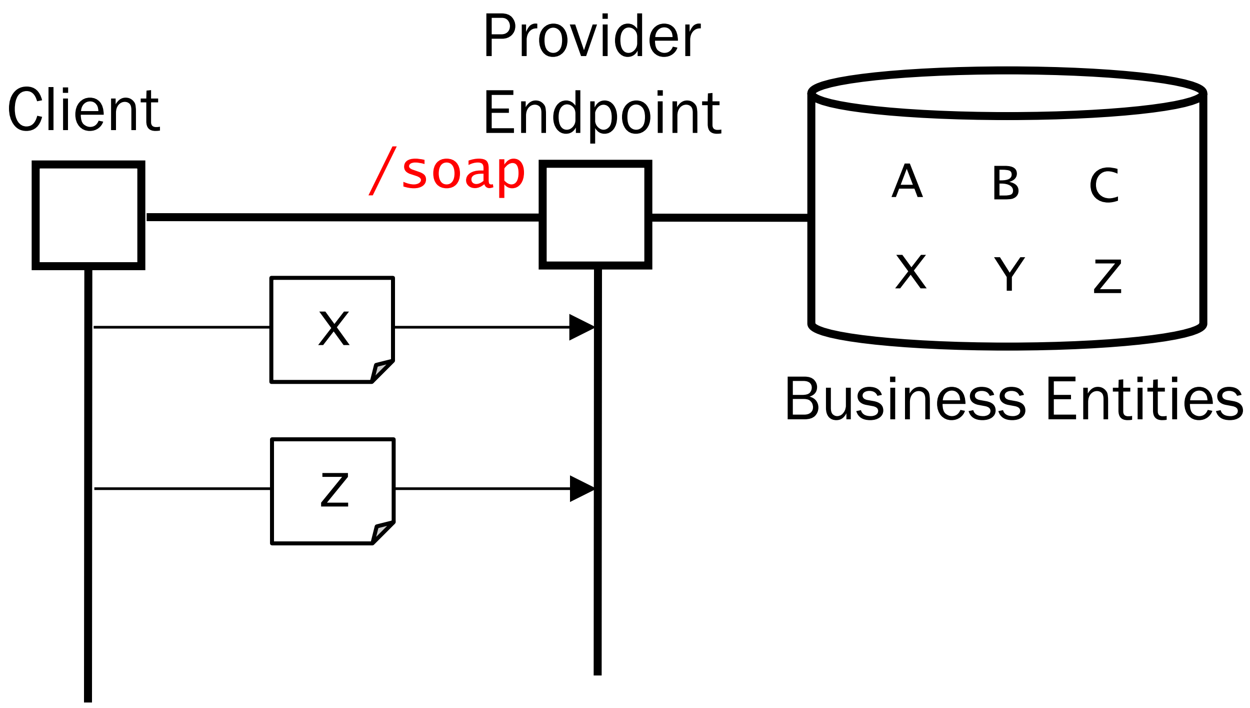
Antipatterns – HTTP as a tunnel

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

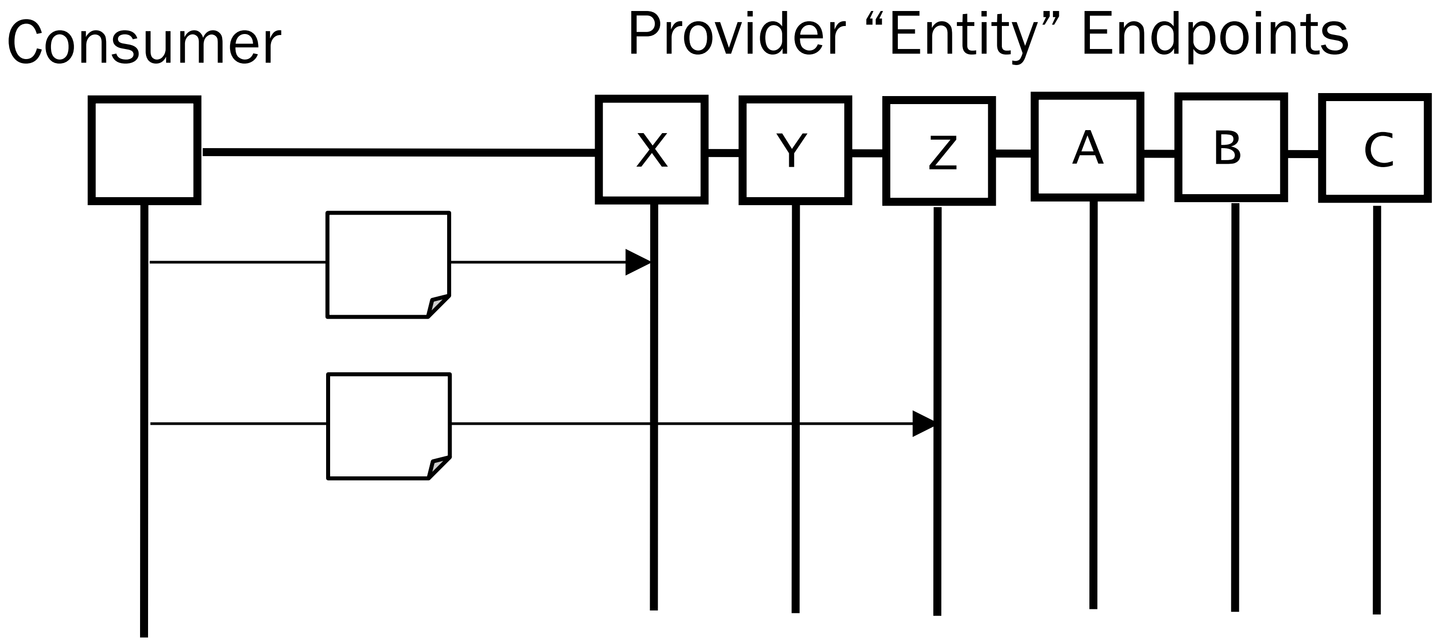
Is this a resource?

Tunneling through one endpoint



- Problem: A service with a single endpoint is too coarse-grained when its operations need to be invoked on its data entities. A client needs to work with two identifiers: a global one for the service and a local one for the entity managed by the service. Entity identifiers cannot be easily reused and shared among multiple services

Global addressability



- Solution: expose each resource entitity as individual "endpoint" of the service they reside in
- Benefits: Global addressability of service entities

Antipatterns – Cookies

- 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

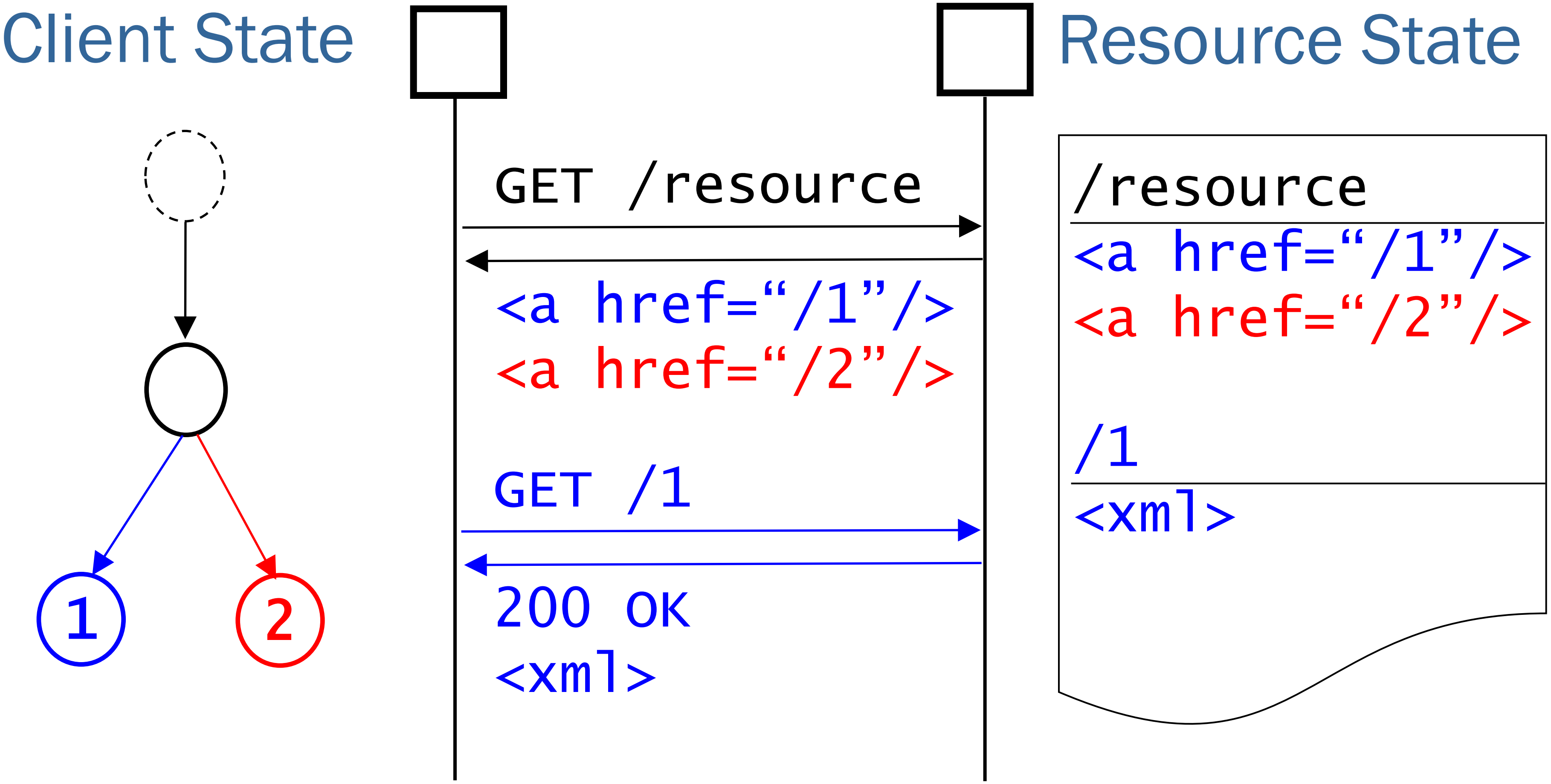
- The client interacts with resources by "navigating hyperlinks" and its state captures the current position in the hypertext.
- The server may influence the state transitions of the client by sending different representations (containing hyperlinks to be followed) in response to GET requests

Resource State

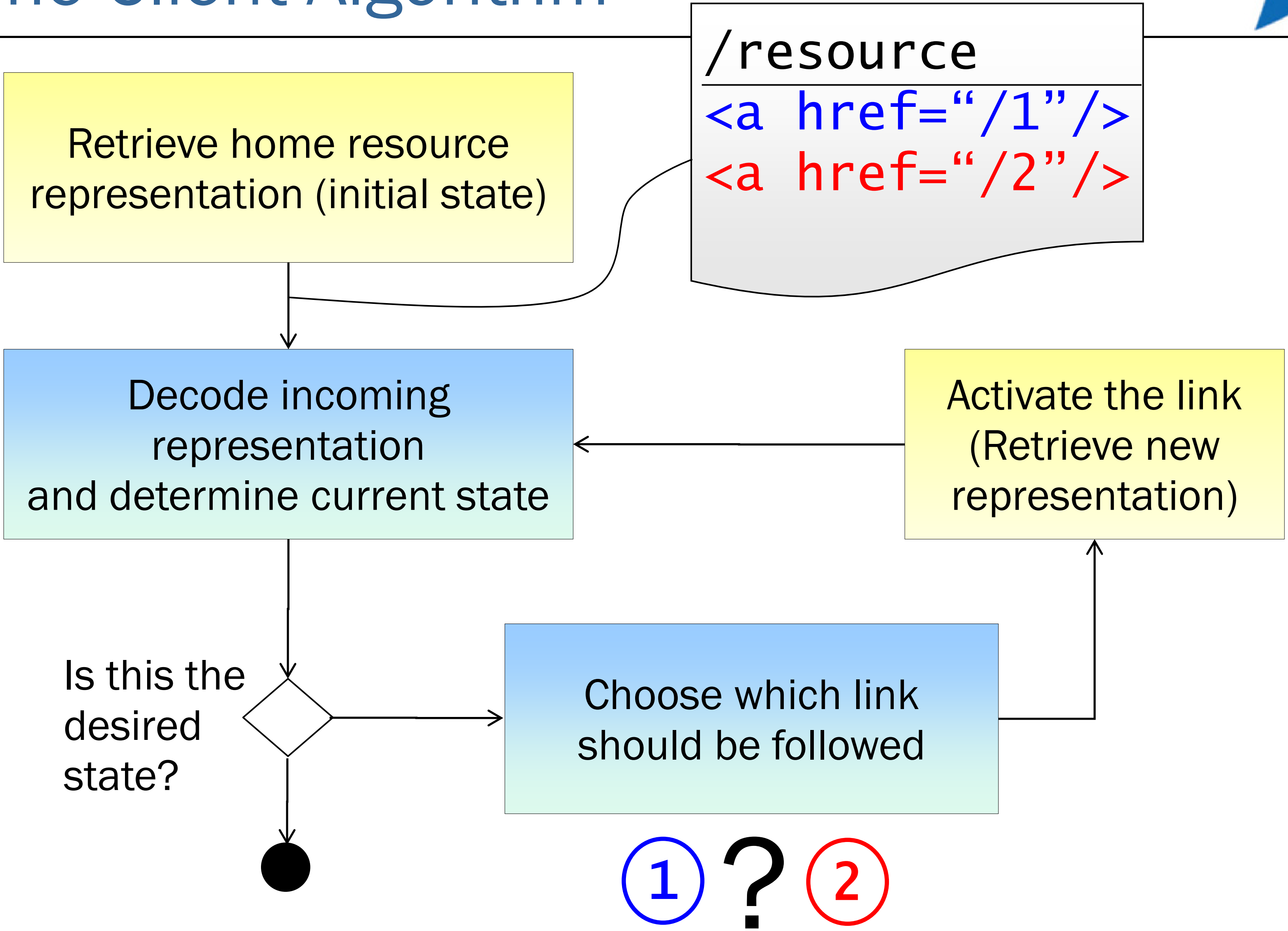
- The state of resources captures the persistent state of the service.
- This state can be accessed by clients under different representations
- The client manipulates the state of resources using the uniform interface CRUD-like semantics (PUT, DELETE, POST)

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.



The Client Algorithm



```
/resource
<a href="/1"/>
<a href="/2"/>
```

Retrieve home resource representation (initial state)

Decode incoming representation and determine current state

Is this the desired state?

Choose which link should be followed

Activate the link (Retrieve new representation)

Adapted from Leonard Richardson's WS-REST 2010 slides

Università
della
Svizzera
italiana

Faculty
of Informatics

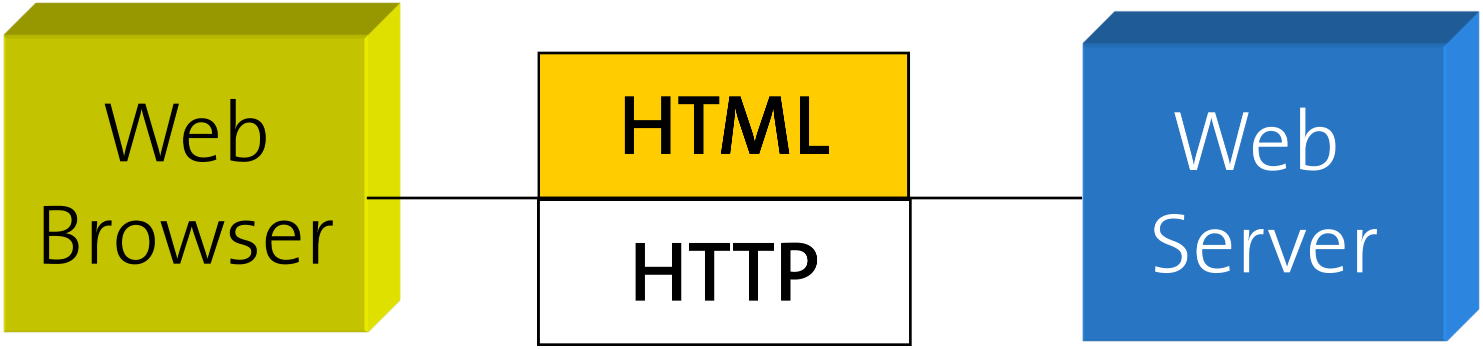
UC Berkeley
School of Information

3 REST vs WS-* Comparison

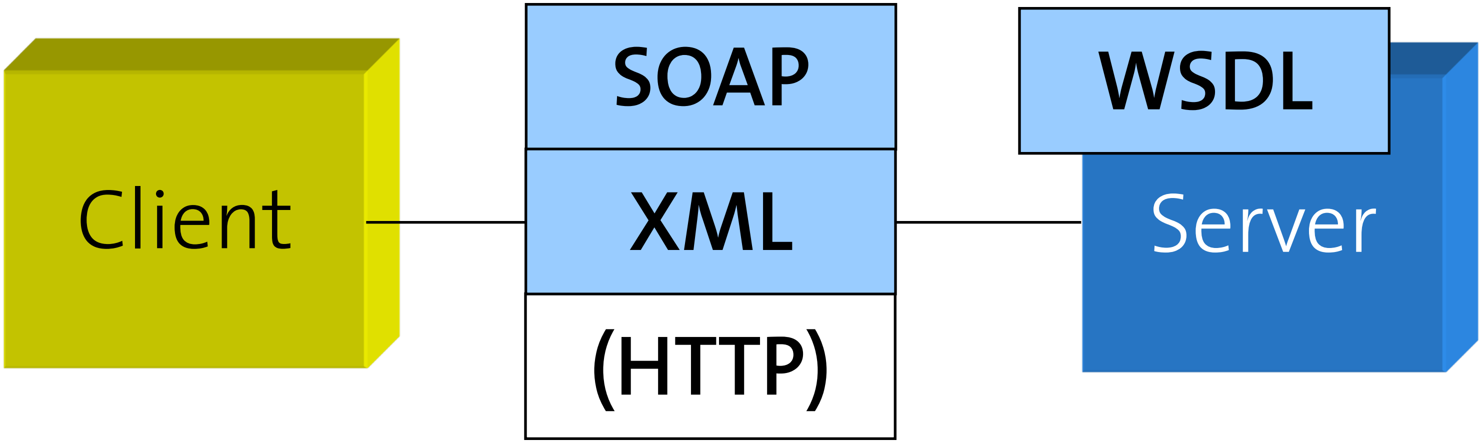
Cesare Pautasso
Faculty of Informatics
University of Lugano, Switzerland

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

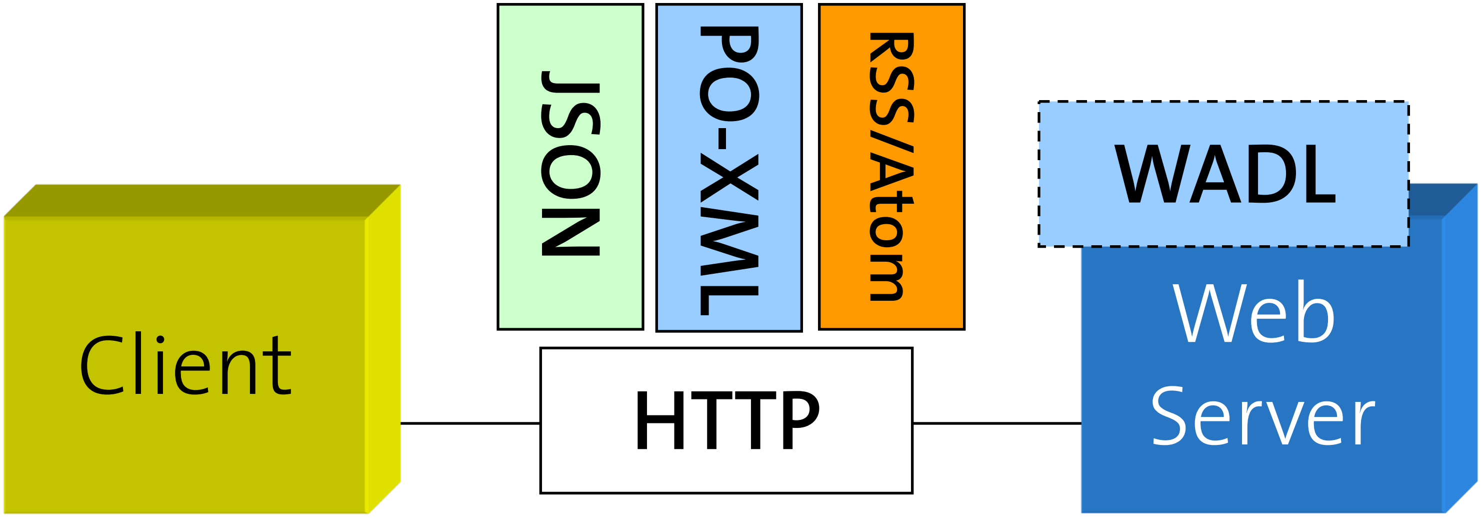
Web Sites (1992)



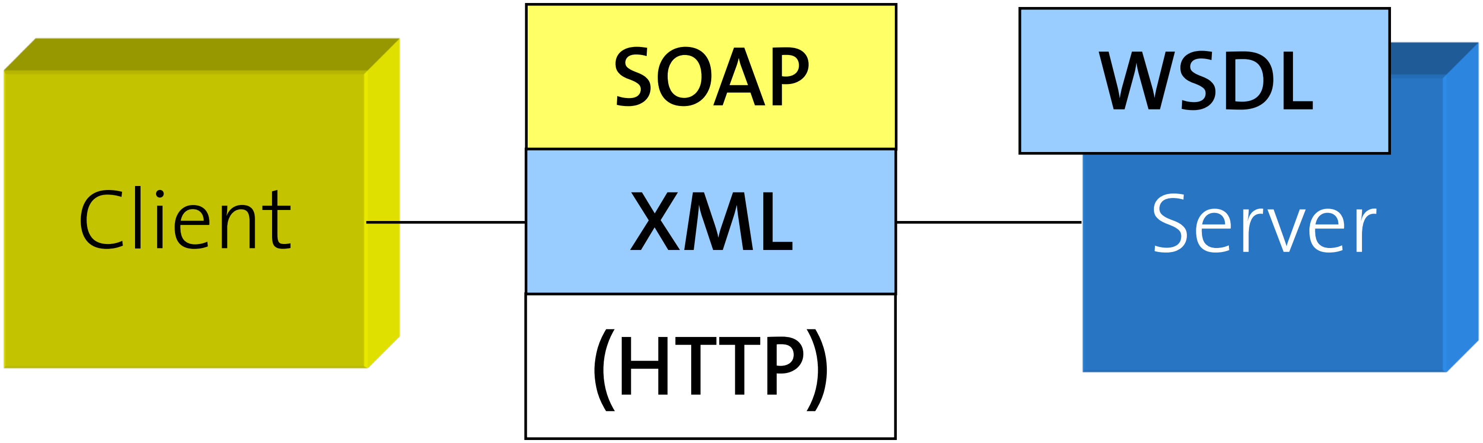
WS-* Web Services (2000)



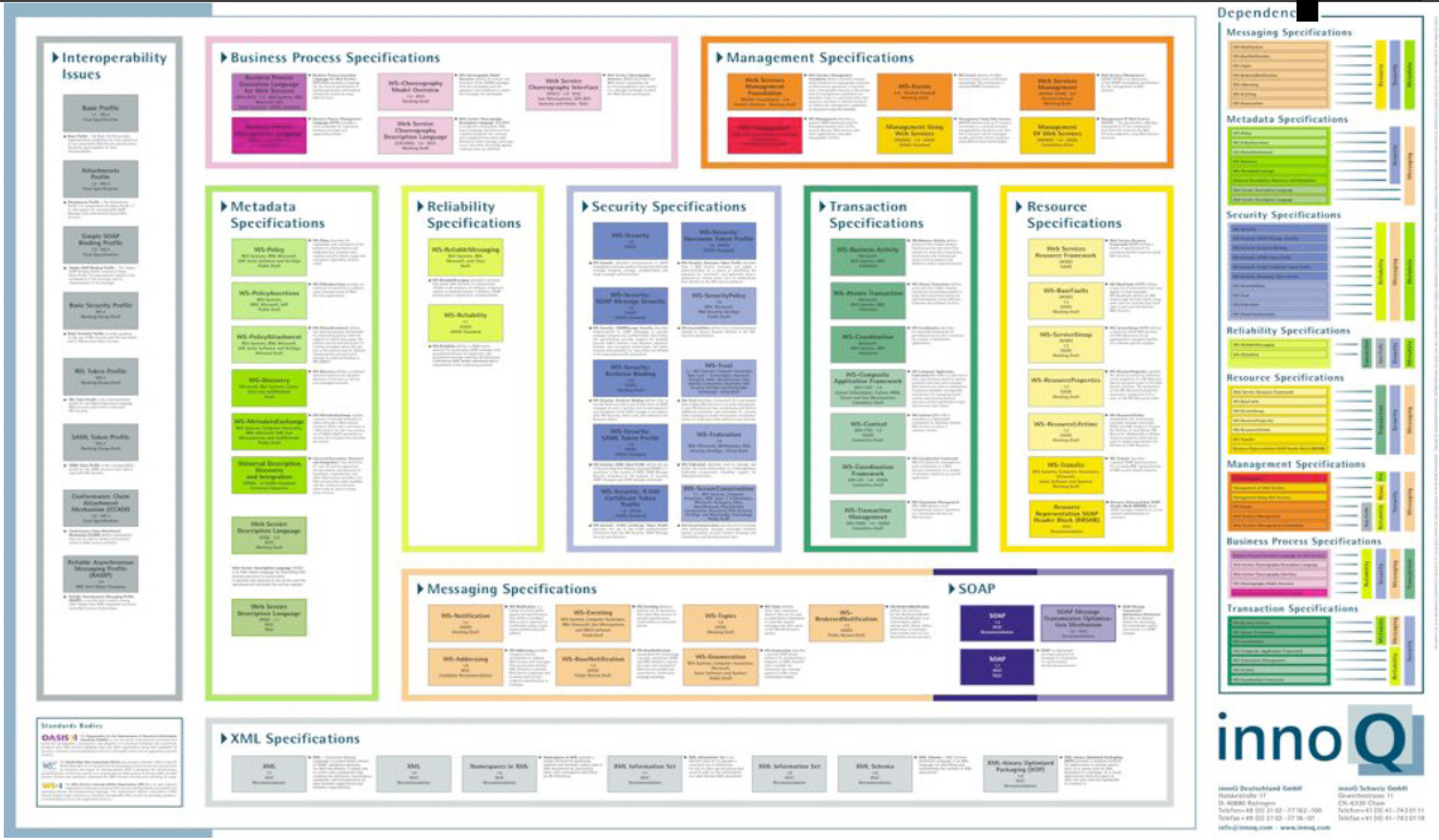
RESTful Web Services (2007)



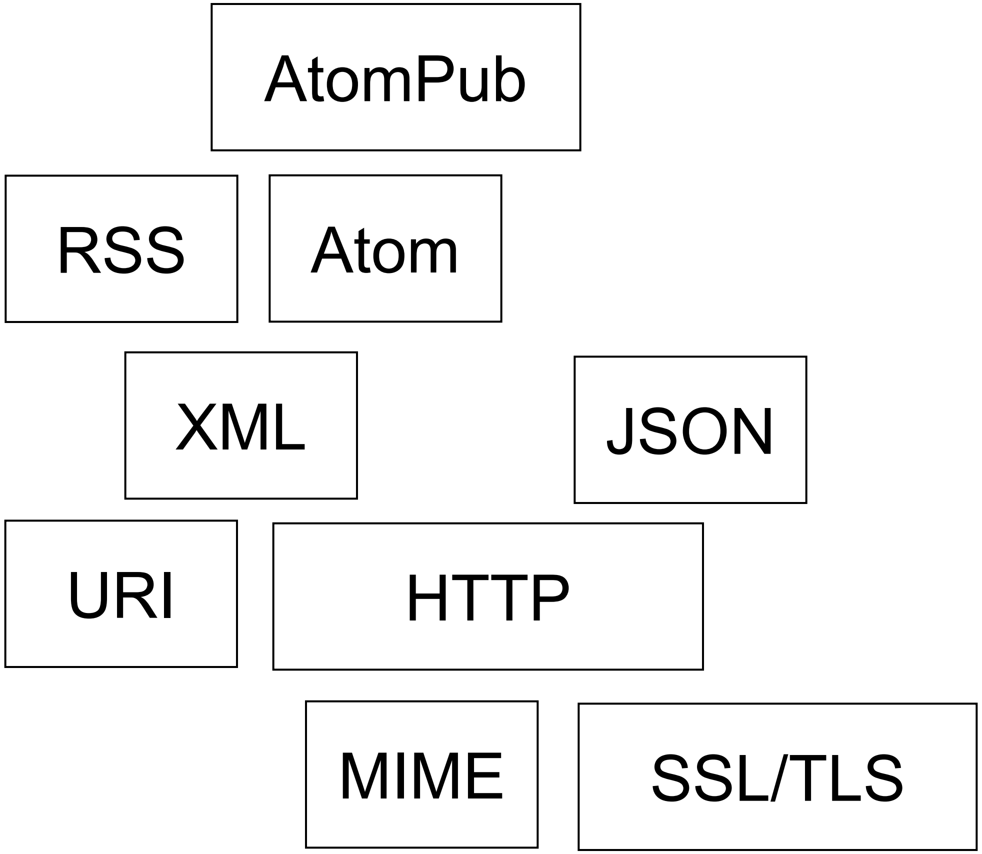
WS-* Web Services (2000)



WS-* Standards Stack


Interoperability Issues
Basic Profile
Attachments Profile
Simple SOAP Binding Profile
Basic Security Profile
REL Token Profile
SAML Token Profile
Conformance Claim Attachment Mechanism (CCAM)
Reliable Asynchronous Messaging Profile (RAMP)
Business Process Specifications
Business Process Execution Language for Web Services
WS-Choreography Model Overview
Web Service Choreography Interface
Business Process Management Language
Web Service Choreography Description Language
Management Specifications
Web Services Management Foundation
WS-Events
Web Services Management
WS-Management
Management Using Web Services
Management Of Web Services
Metadata Specifications
WS-Policy
WS-PolicyAssertions
WS-PolicyAttachment
WS-Discovery
WS-MetadataExchange
Universal Description, Discovery and Integration
Web Service Description Language
Reliability Specifications
WS-ReliableMessaging
WS-Reliability
Security Specifications
WS-Security
WS-Security: Kerberos Token Profile
WS-Security: SOAP Message Security
WS-SecurityPolicy
WS-Security: Kerberos Binding
WS-Trust
WS-Security: SAML Token Profile
WS-Federation
WS-Security: X.509 Certificate Token Profile
WS-SecureConversation
Transaction Specifications
WS-Business Activity
WS-Atomic Transaction
WS-Coordination
WS-Composite Application Framework
WS-Context
WS-Coordination Framework
WS-Transaction Management
Resource Specifications
Web Services Resource Framework
WS-BaseFaults
WS-ServiceGroup
WS-ResourceProperties
WS-ResourceLifetime
WS-Transfer
Resource Representation SOAP Header Block (RRSHB)
Messaging Specifications
WS-Notification
WS-Eventing
WS-Topics
WS-BrokeredNotification
WS-Addressing
WS-BaseNotification
WS-Enumeration
SOAP
SOAP
SOAP Message Transmission Optimization Mechanism
SOAP
Standards Bodies
OASIS
W3C
WS-I
XML Specifications
XML
XML
Namespaces in XML
XML Information Set
XML Information Set
XML Schema
XML-binary Optimized Packaging (XOP)
Dependencies
Messaging Specifications
Metadata Specifications
Security Specifications
Reliability Specifications
Resource Specifications
Management Specifications
Business Process Specifications
Transaction Specifications
innoQ

RESTful Web Services Standards Stack



AtomPub

RSS

Atom

XML

JSON

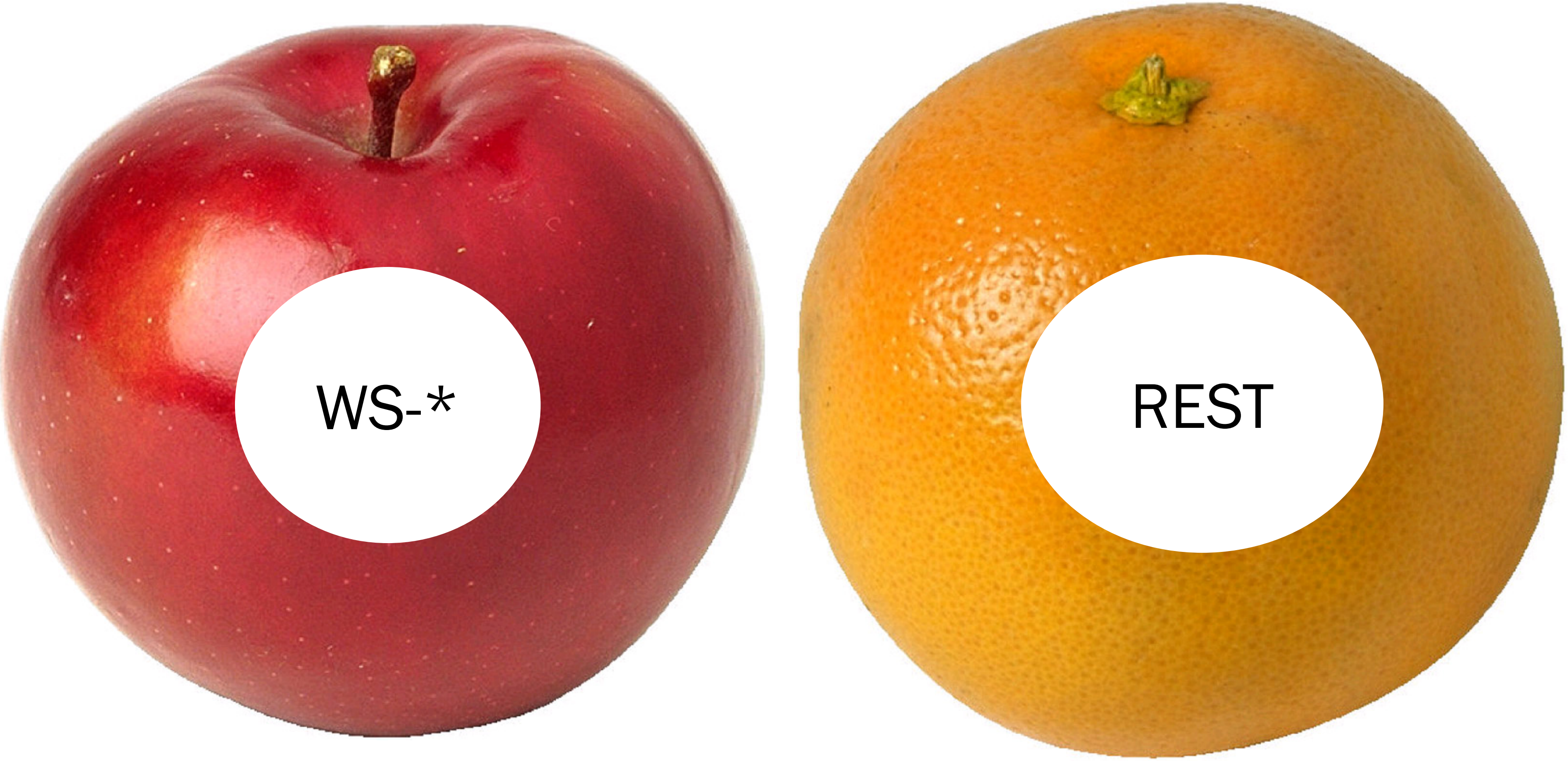
URI

HTTP

MIME

SSL/TLS

Can we really compare WS-* vs. REST?



Can we really compare WS-* vs. REST?

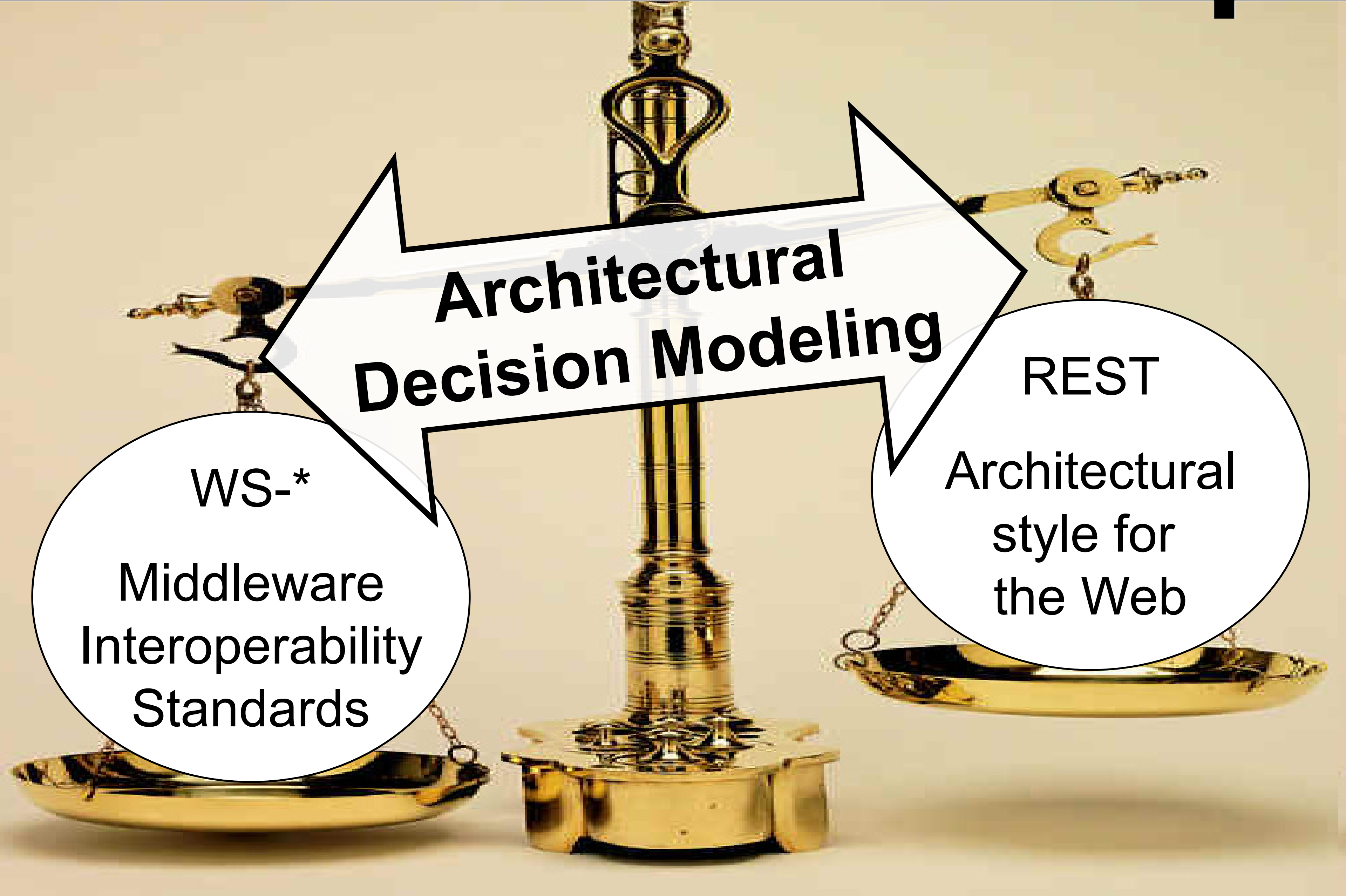
WS-*

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

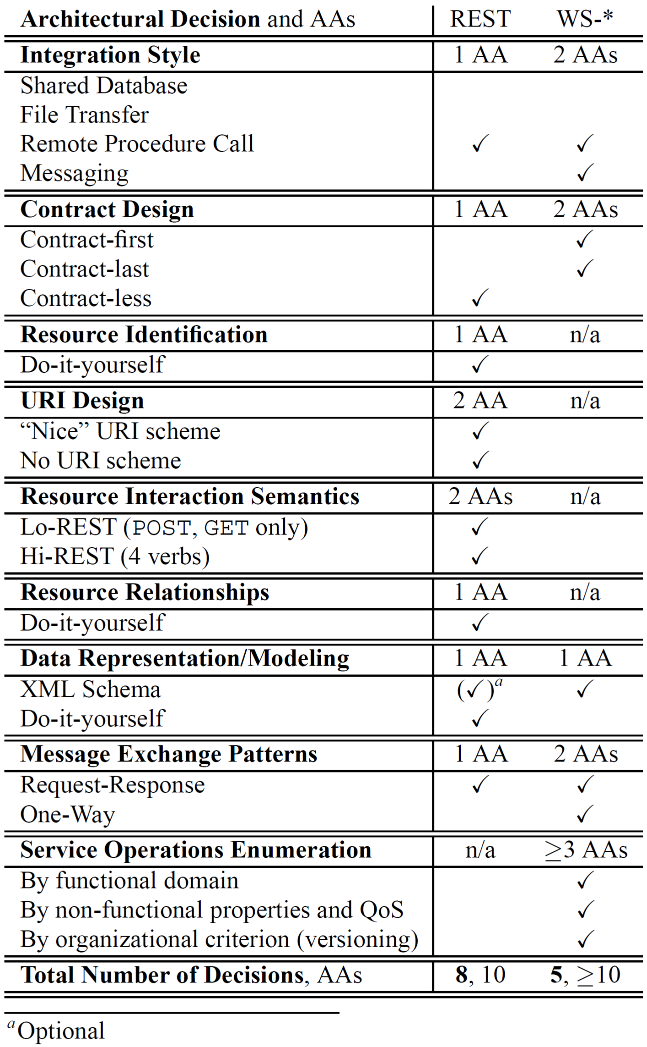
- Architectural decisions capture the main design issues and the rationale behind a chosen technical solution
- The choice between REST vs. WS-* is an important architectural decision for Web service design
- Architectural decisions affect one another

Architectural Decision:
Programming Language

Architecture Alternatives:
1. Java
2. C#
3. C++
4. C
5. Eiffel
6. Ruby
7. …

Rationale

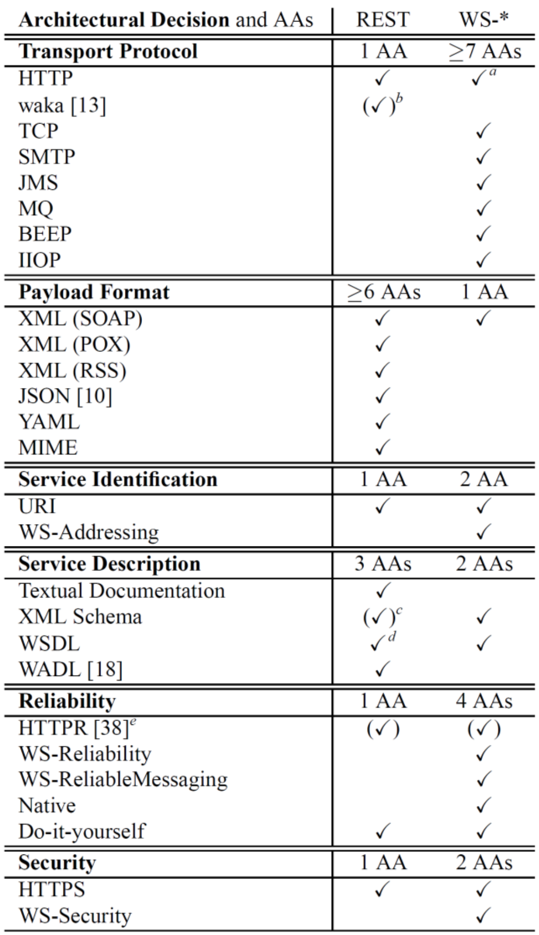
Decision Space Overview

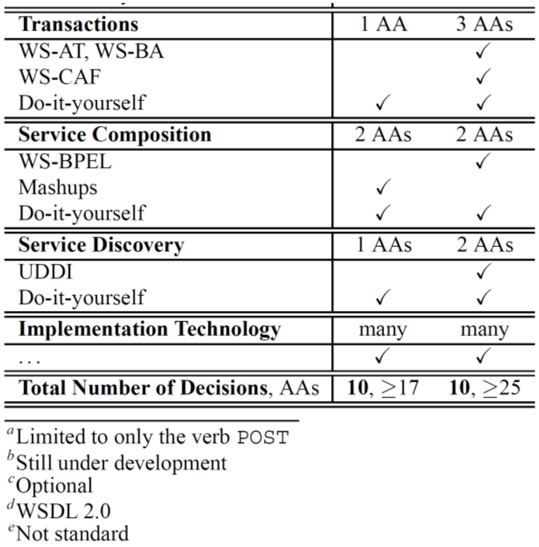


Architectural Decision and AAs	REST	WS-*
Integration Style	1 AA	2 AAs
Shared Database		
File Transfer		
Remote Procedure Call	✓	✓
Messaging		✓
Contract Design	1 AA	2 AAs
Contract-first		✓
Contract-last		✓
Contract-less	✓	
Resource Identification	1 AA	n/a
Do-it-yourself	✓	
URI Design	2 AA	n/a
"Nice" URI 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

[a] Optional

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		✓
IIOP		✓
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

[a] Limited to only the verb `POST`

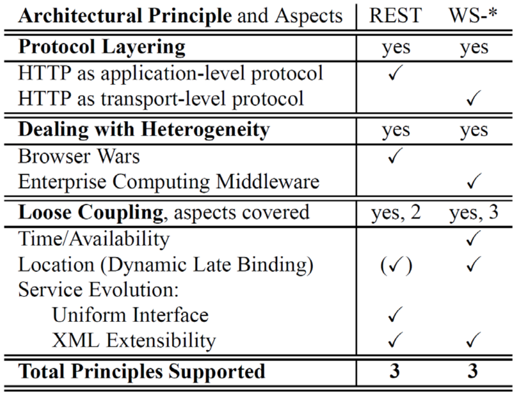
[b] Still under development

[c] Optional

[d] WSDL 2.0

[e] Not 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

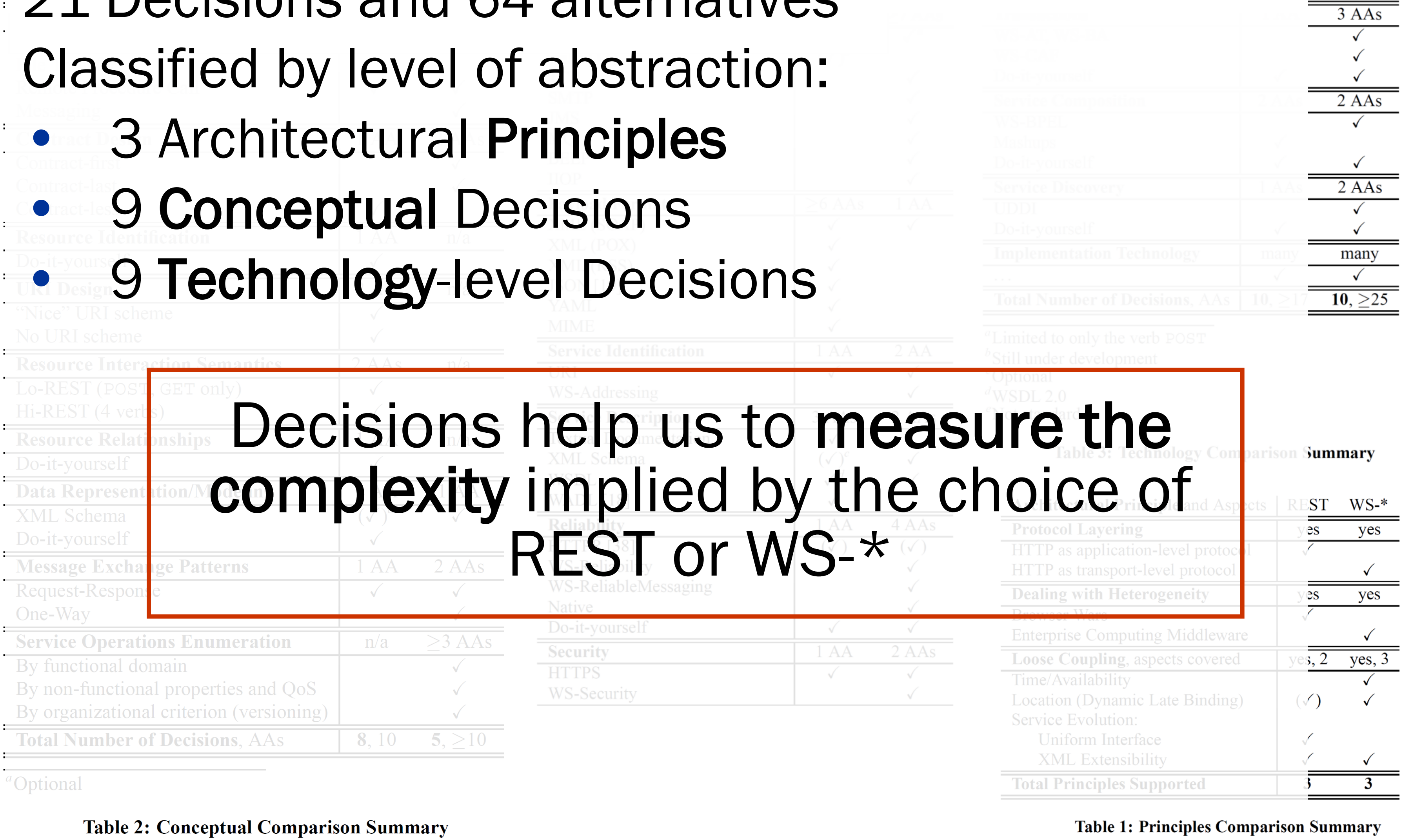
Decision Space Overview

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-*



Resource Identification	1 AA	n/a
Do-it-yourself	✓	
URI Design		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	2 AAs	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

[a] Optional

Table 2: Conceptual Comparison Summary

[a] Limited to only the verb POST

[b] Still under development

[c] Optional

[d] WSDL 2.0

[e] Not 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**

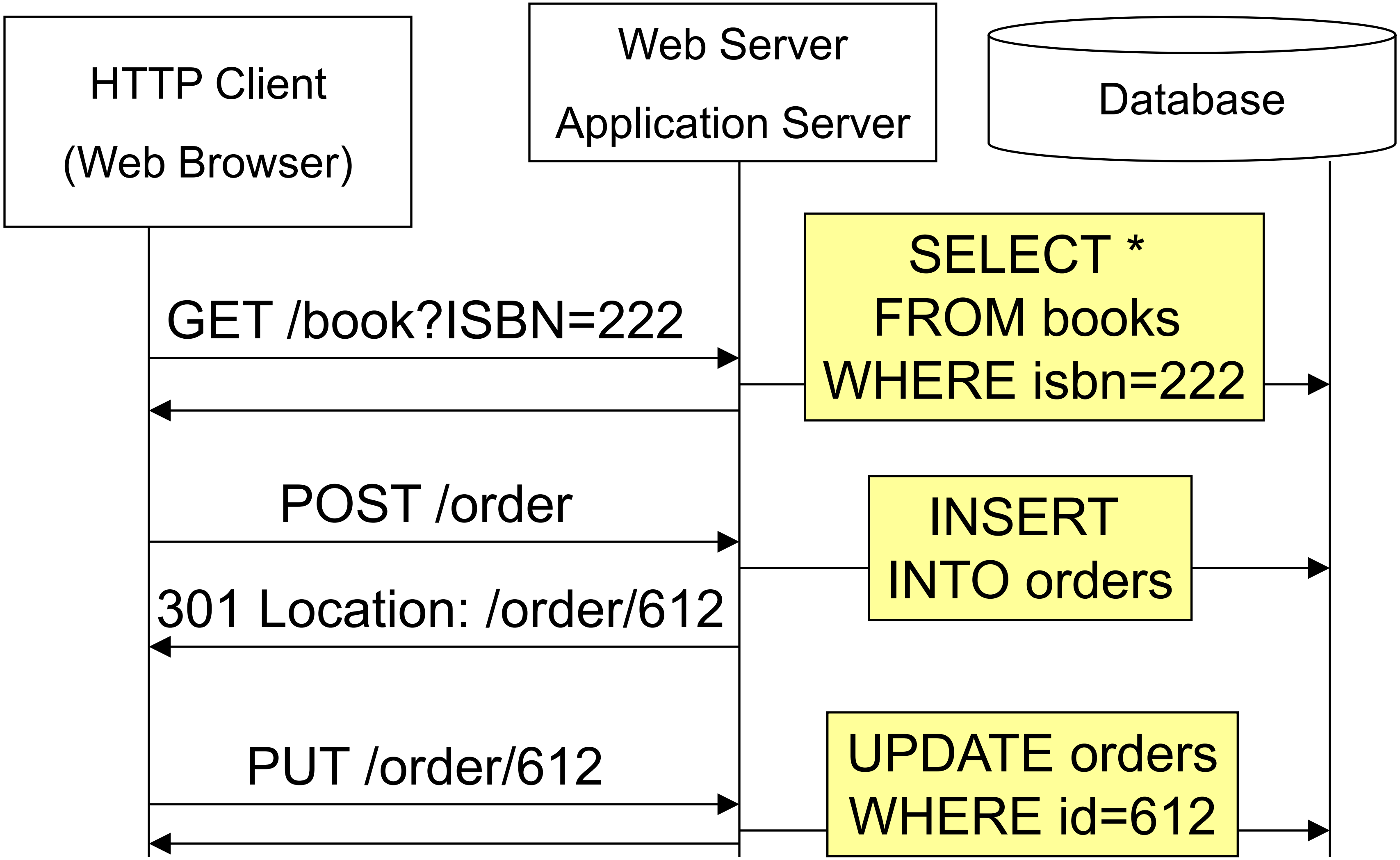
Table 1: Principles Comparison Summary

Architectural Principles

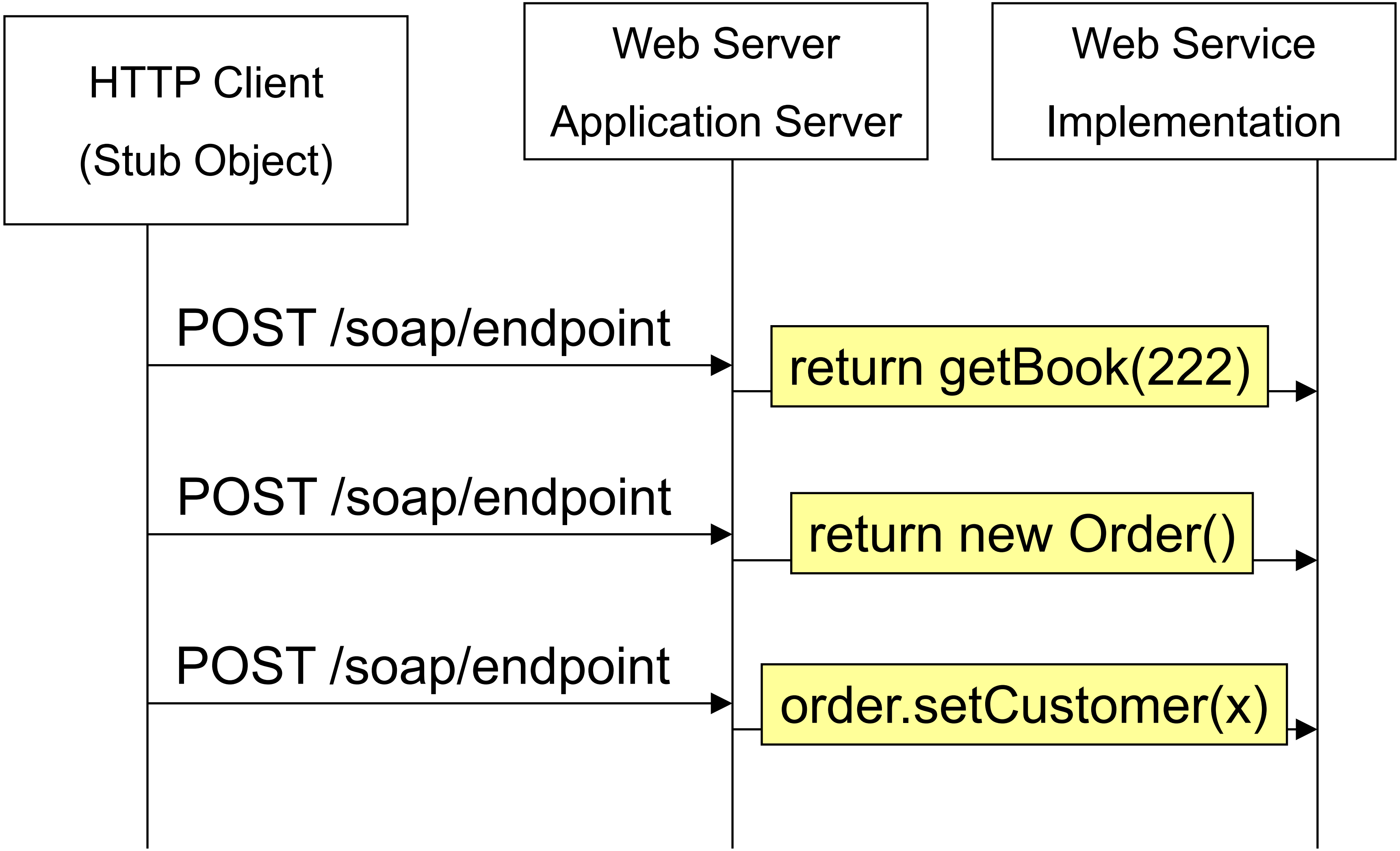
1. Protocol Layering
   - HTTP = Application-level Protocol (REST)
   - HTTP = Transport-level Protocol (WS-*)
2. Dealing with Heterogeneity
3. Loose Coupling*

* http://dret.net/netdret/docs/loosely-coupled-www2009/

RESTful Web Service Example



WS-* Service Example (from REST perspective)



HTTP Client

(Stub Object)

Web Server

Application Server

Web Service

Implementation

POST /soap/endpoint

return getBook(222)

POST /soap/endpoint

return new Order()

POST /soap/endpoint

order.setCustomer(x)

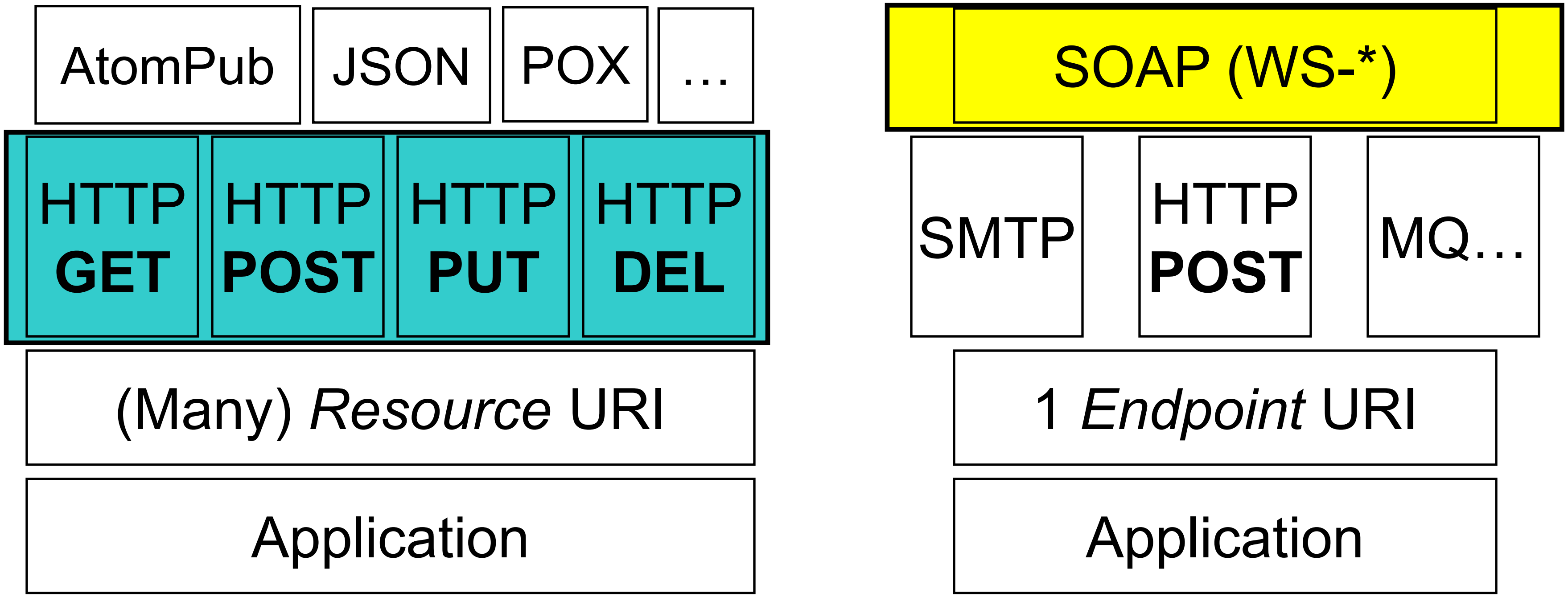
Protocol Layering

"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"

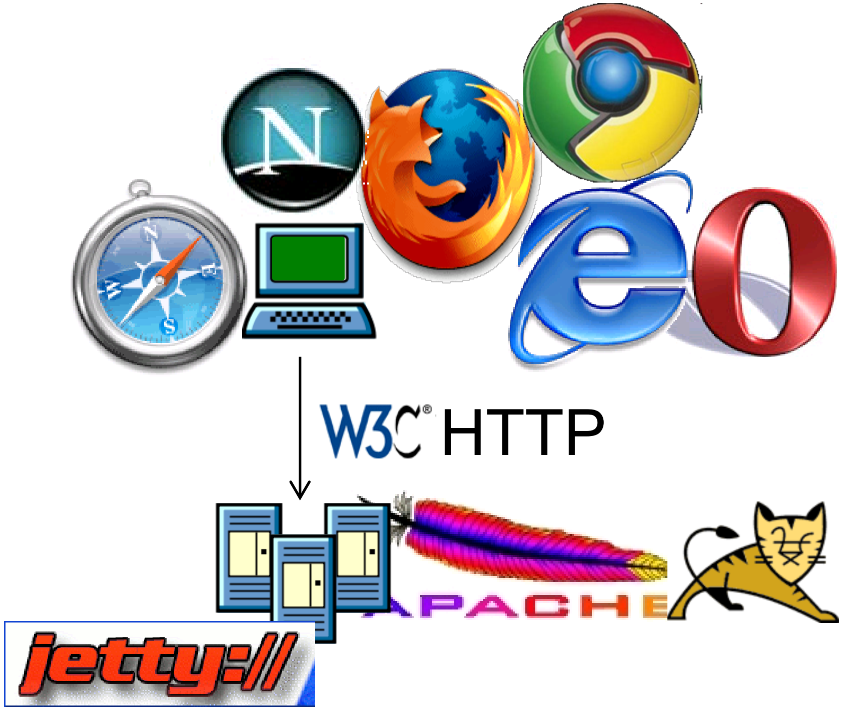
- Applications get a chance to interact but they remain "outside of the Web"

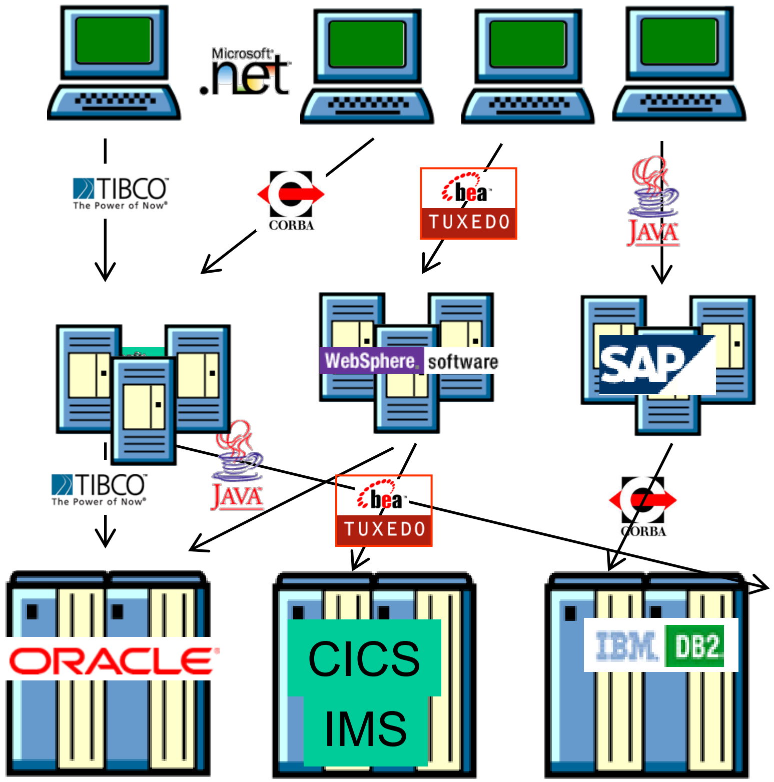


Dealing with Heterogeneity

- Enable Cooperation
- Web Applications

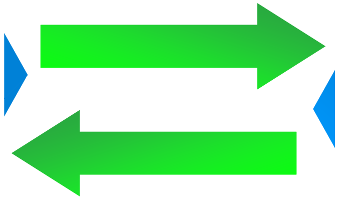
- Enable Integration
- Enterprise Computing





Picture from Eric Newcomer, IONA

Different software connectors



Managing State

- 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 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 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)

What about security?

- 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)
- 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

Measuring Complexity

- Why is REST perceived to be simpler?
- Architectural Decisions give a **quantitative measure** of the complexity of an architectural design space:
  - Total number of decisions
  - For each decision, number of alternative options
  - For each alternative option, estimate the effort

	REST	WS-*
Decisions	17	14
Alternatives	27	35

Decisions with *1 or more* alternative options

Measuring Complexity

	REST	WS-*
Decisions	5	12
Alternatives	16	32

Decisions with *more than 1* alternative options

	REST	WS-*
Decisions	17	14
Alternatives	27	35

Decisions with *1 or more* alternative options

Measuring Complexity

	REST	WS-*
Decisions	5	12
Alternatives	16	32

Decisions with *more than 1* alternative options

- URI Design
- Resource Interaction Semantics
- Payload Format
- Service Description
- Service Composition

Measuring Complexity

	REST	WS-*
Decisions	5	12
Alternatives	16	32

Decisions with *more than 1* alternative options

	REST	WS-*
Decisions	12	2

Decisions with *only 1* alternative option

Measuring Complexity

- Payload Format
- Data Representation Modeling

	REST	WS-*
Decisions	12	2

Decisions with *only 1* alternative option

Measuring Effort

	REST	WS-*
Do-it-yourself Alternatives	5	0

Decisions with **only** *do-it-yourself* alternatives

	REST	WS-*
Decisions	12	2

Decisions with *only 1* alternative option

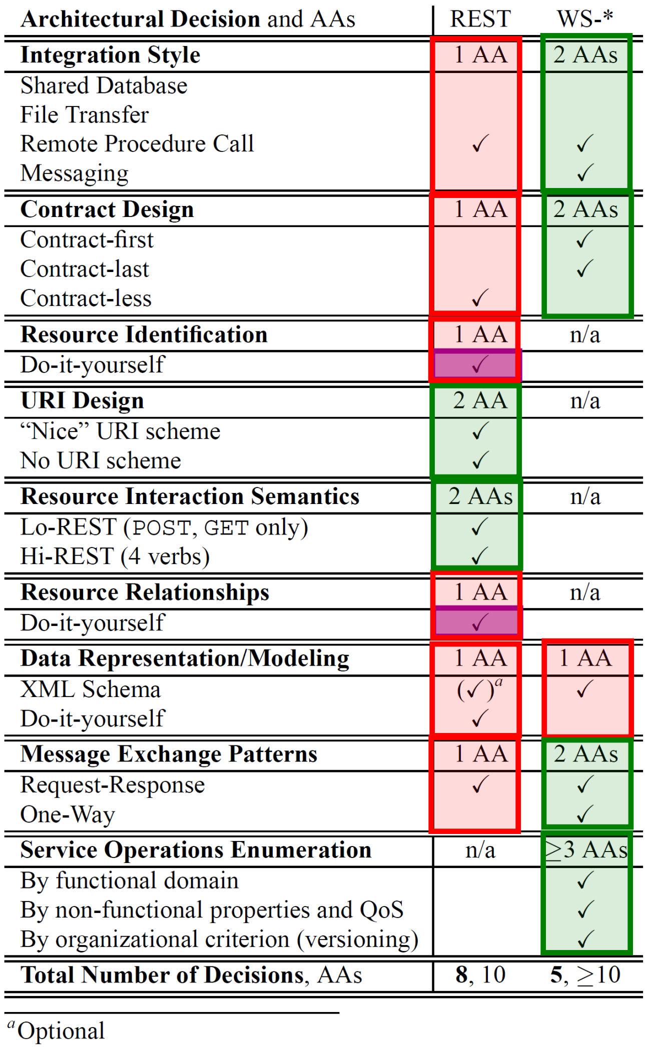
Measuring Effort

	REST	WS-*
Do-it-yourself Alternatives	5	0

Decisions with **only** *do-it-yourself* alternatives

- Resource Identification
- Resource Relationship
- Reliability
- Transactions
- Service Discovery

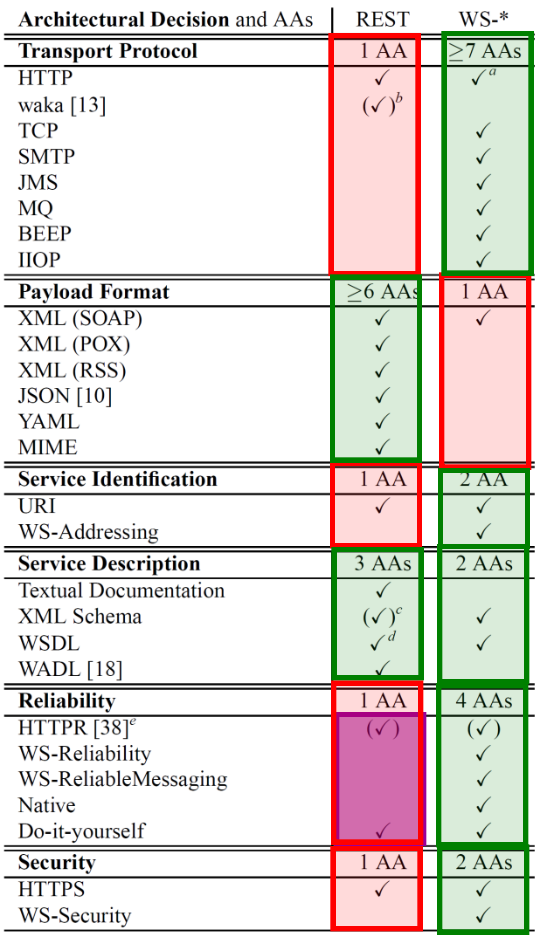
Freedom of Choice (>1 Alternative)
Freedom from Choice (=1 Alternative)



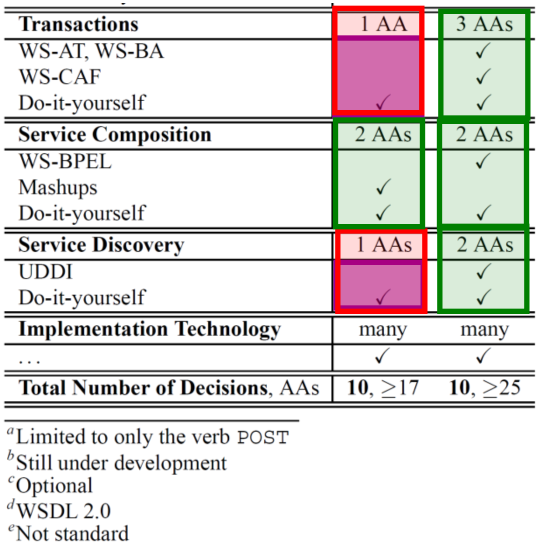
Architectural Decision and AAs	REST	WS-*
Integration Style	1 AA	2 AAs
Shared Database		
File Transfer		
Remote Procedure Call	✓	✓
Messaging		✓
Contract Design	1 AA	2 AAs
Contract-first		✓
Contract-last		✓
Contract-less	✓	
Resource Identification	1 AA	n/a
Do-it-yourself	✓	
URI Design	2 AA	n/a
“Nice” URI 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

[a] Optional

Table 2: Conceptual Comparison Summary



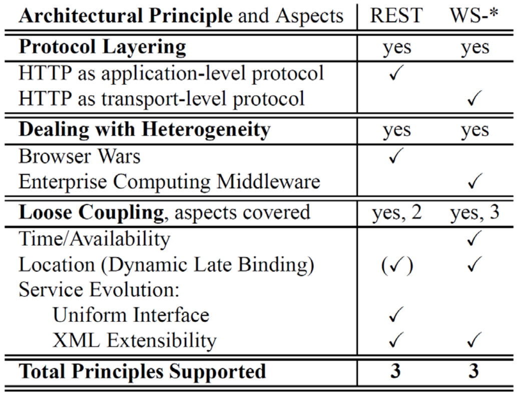
Architectural Decision and AAs	REST	WS-*
Transport Protocol	1 AA	≥7 AAs
HTTP	✓	✓[a]
waka [13]	(✓)[b]	
TCP		✓
SMTP		✓
JMS		✓
MQ		✓
BEEP		✓
IIOP		✓
Payload Format	≥6 AAs	1 AA
XML (SOAP)	✓	✓
XML (POX)	✓	
XML (RSS)	✓	
JSON [10]	✓	
YAML	✓	
MIME	✓	
Service Identification	1 AA	2 AA
URI	✓	✓
WS-Addressing		✓
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

[a] Limited to only the verb POST
[b] Still under development
[c] Optional
[d] WSDL 2.0
[e] Not 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

Comparison Summary

- Architectural Decisions measure complexity implied by alternative technologies

- **REST simplicity** = freedom from choice
  - 5 decisions require to choose among 16 alternatives
  - 12 decisions are already taken (but 5 are *do-it-yourself*)

- **WS-* complexity** = freedom of choice
  - 12 decisions require to choose among 32 alternatives
  - 2 decisions are already taken (SOAP, WSDL+XSD)

Comparison Conclusion

- 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.

Università
della
Svizzera
italiana

Faculty
of Informatics

UC Berkeley
School of Information

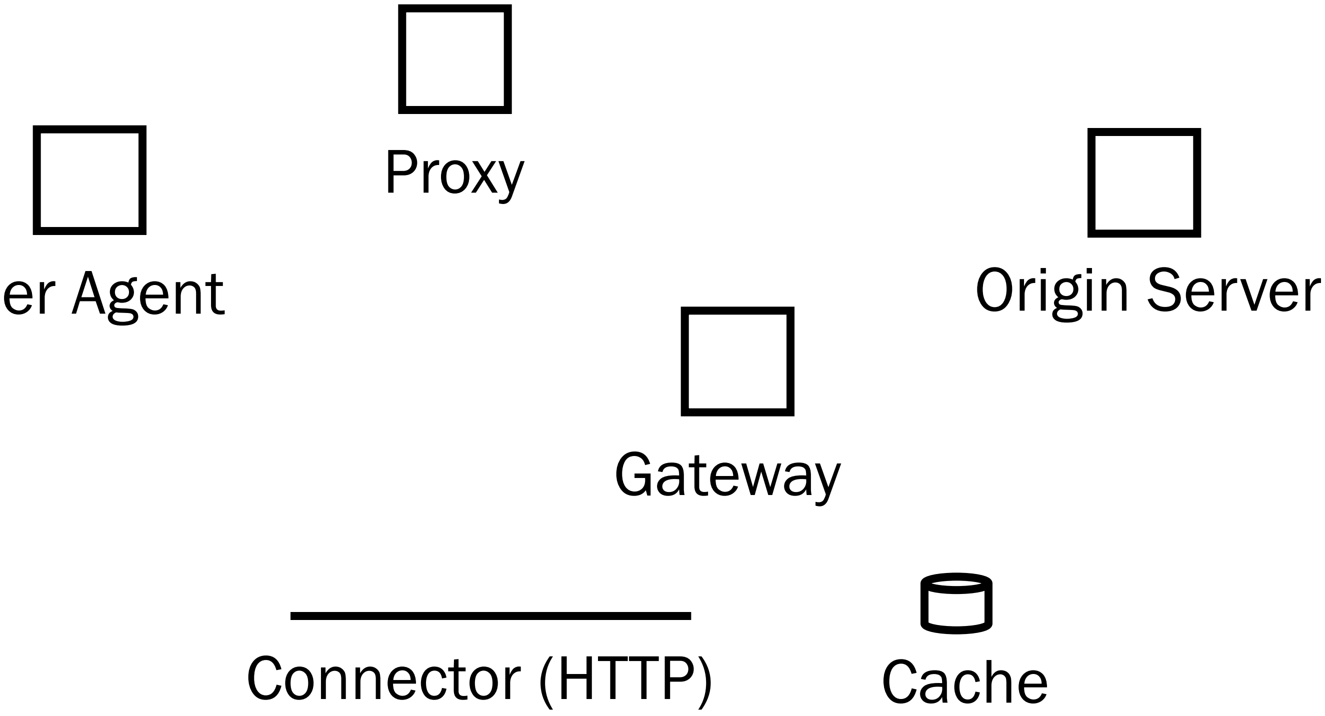
4 RESTful Service Composition

Cesare Pautasso
Faculty of Informatics
University of Lugano, Switzerland

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

REST Architectural Elements

Client/Server Layered Stateless Communication Cache



Proxy

User Agent

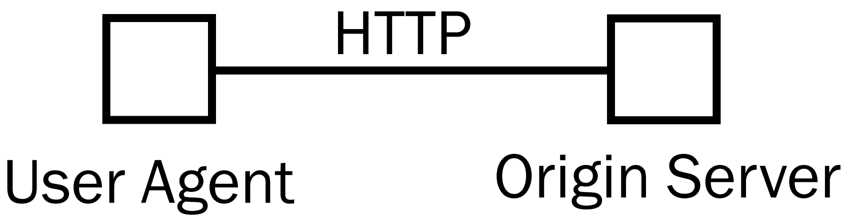
Origin Server

Gateway

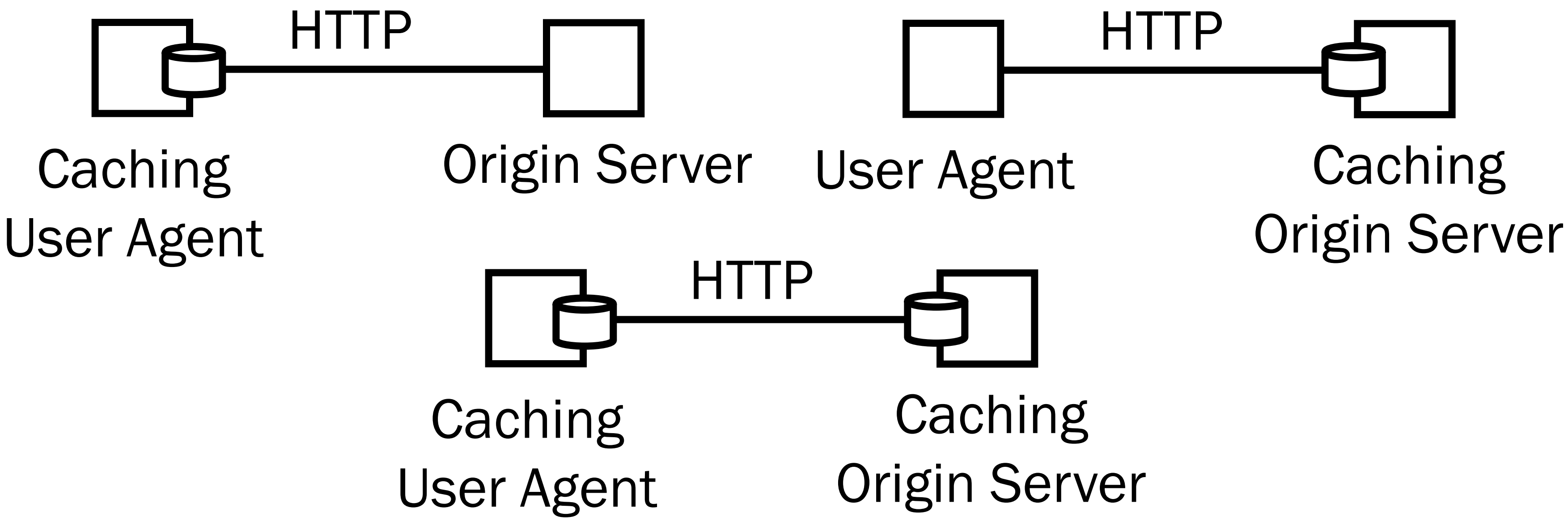
Connector (HTTP)

Cache

Basic Setup

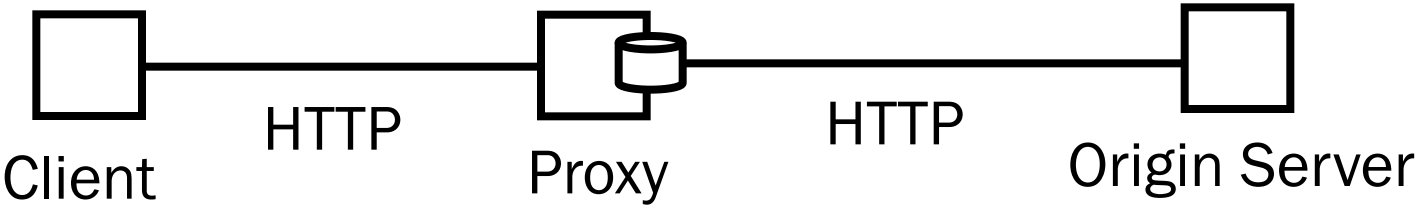


Adding Caching

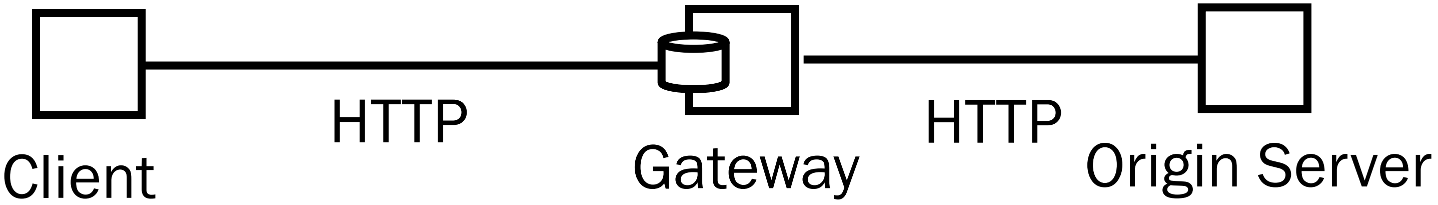


Proxy or Gateway?

Intermediaries forward (and may translate) requests and responses



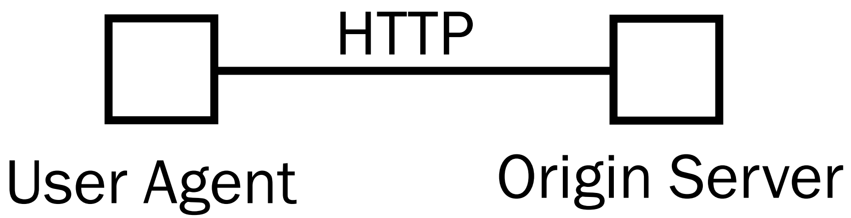
A proxy is chosen by the Client (for caching, or access control)



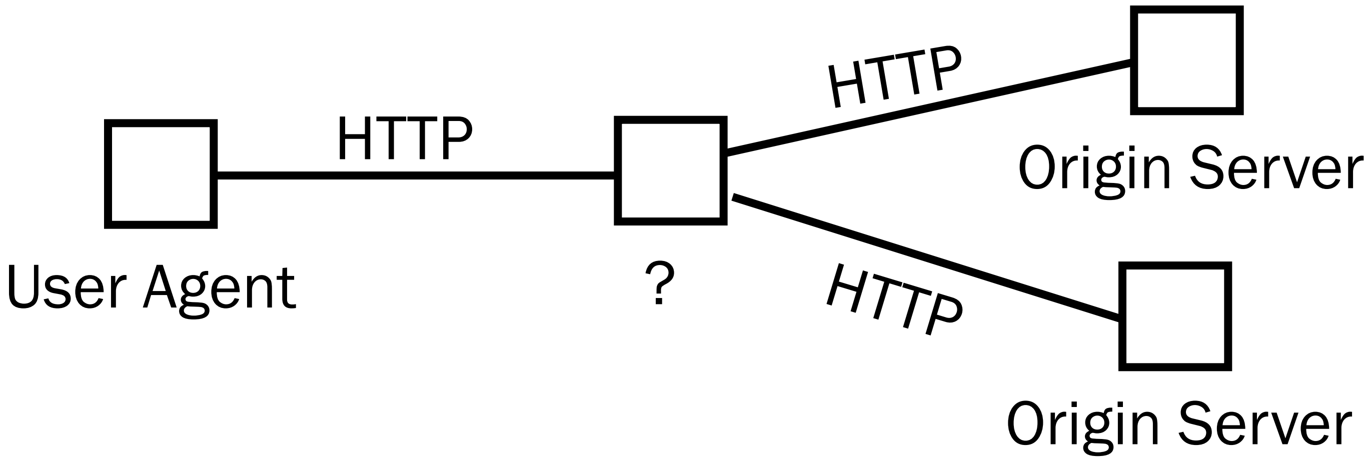
The use of a gateway (or reverse proxy) is imposed by the server

What about composition?

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



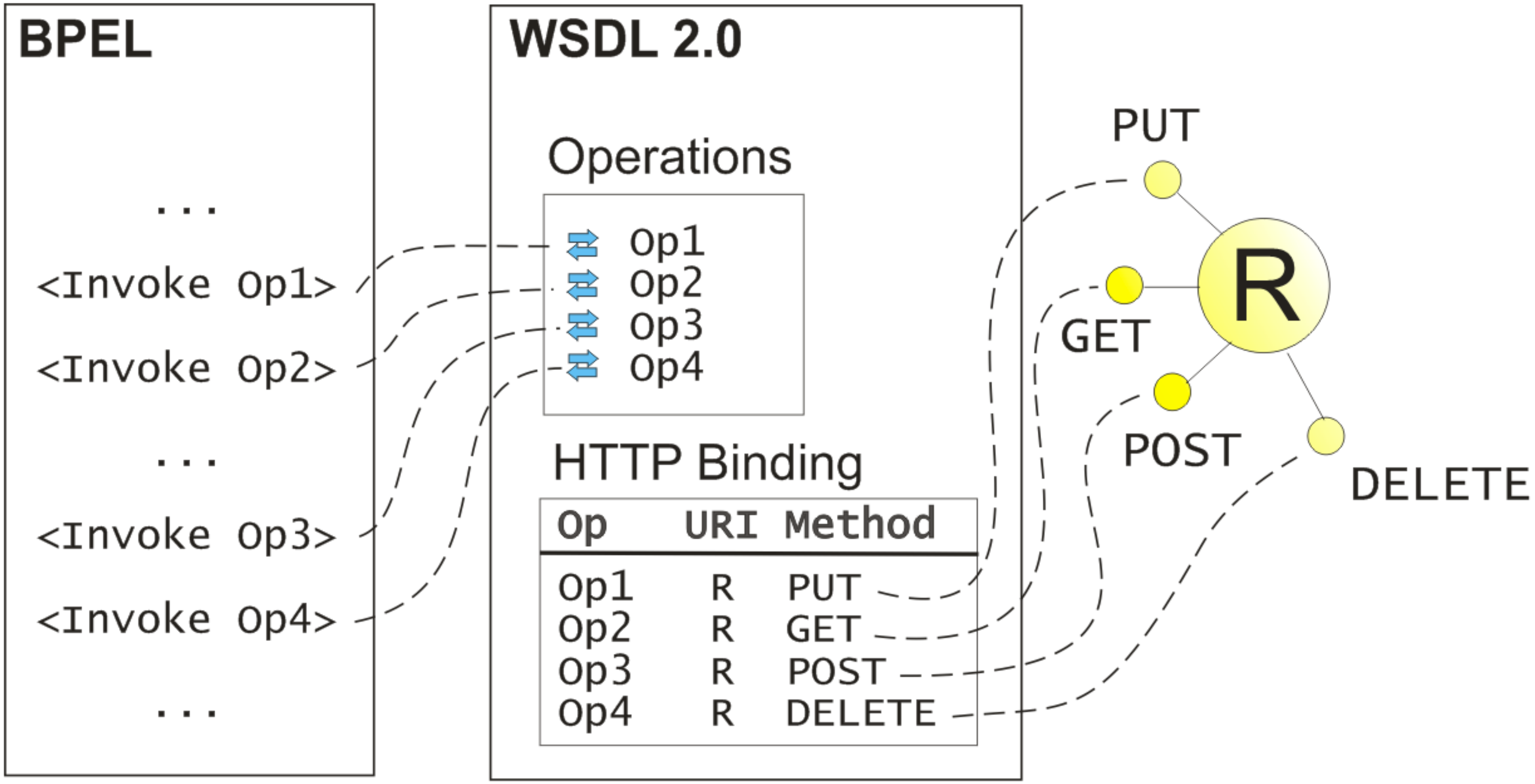
- WS-BPEL is the standard Web service composition language. Business process models are used to specify how a collection of services is orchestrated into a composite service
- Can we apply WS-BPEL to RESTful services?



BPEL and WSDL 2.0

WSDL 2.0 HTTP Binding can wrap RESTful Web Services

(WS-BPEL 2.0 does not support WSDL 2.0)



BPEL

...
<Invoke Op1>
<Invoke Op2>
...
<Invoke Op3>
<Invoke Op4>
...

WSDL 2.0

Operations

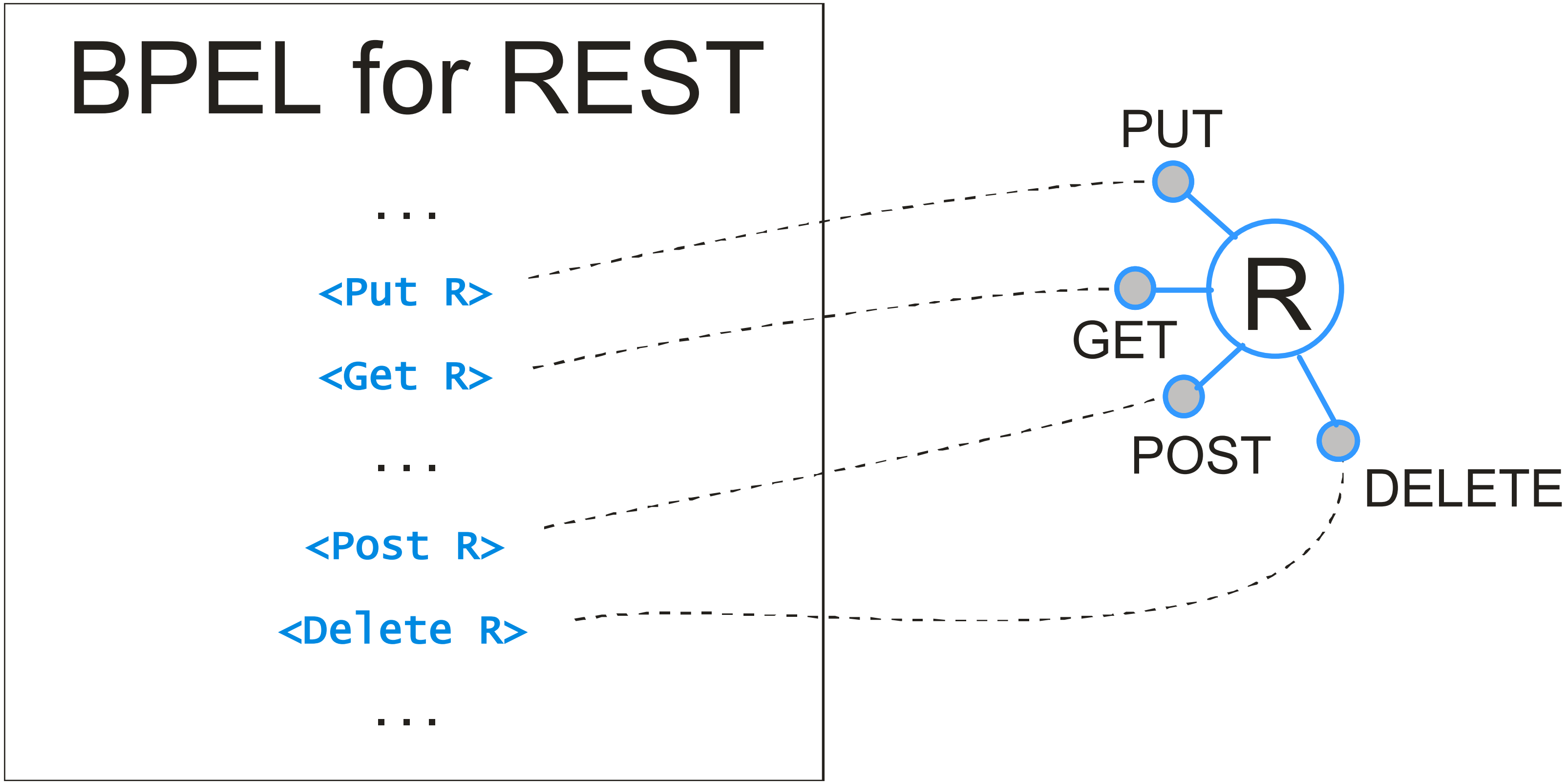
- Op1
- Op2
- Op3
- Op4

HTTP Binding

Op	URI	Method
Op1	R	PUT
Op2	R	GET
Op3	R	POST
Op4	R	DELETE

BPEL for REST

Make REST interaction primitives first-class language constructs of BPEL

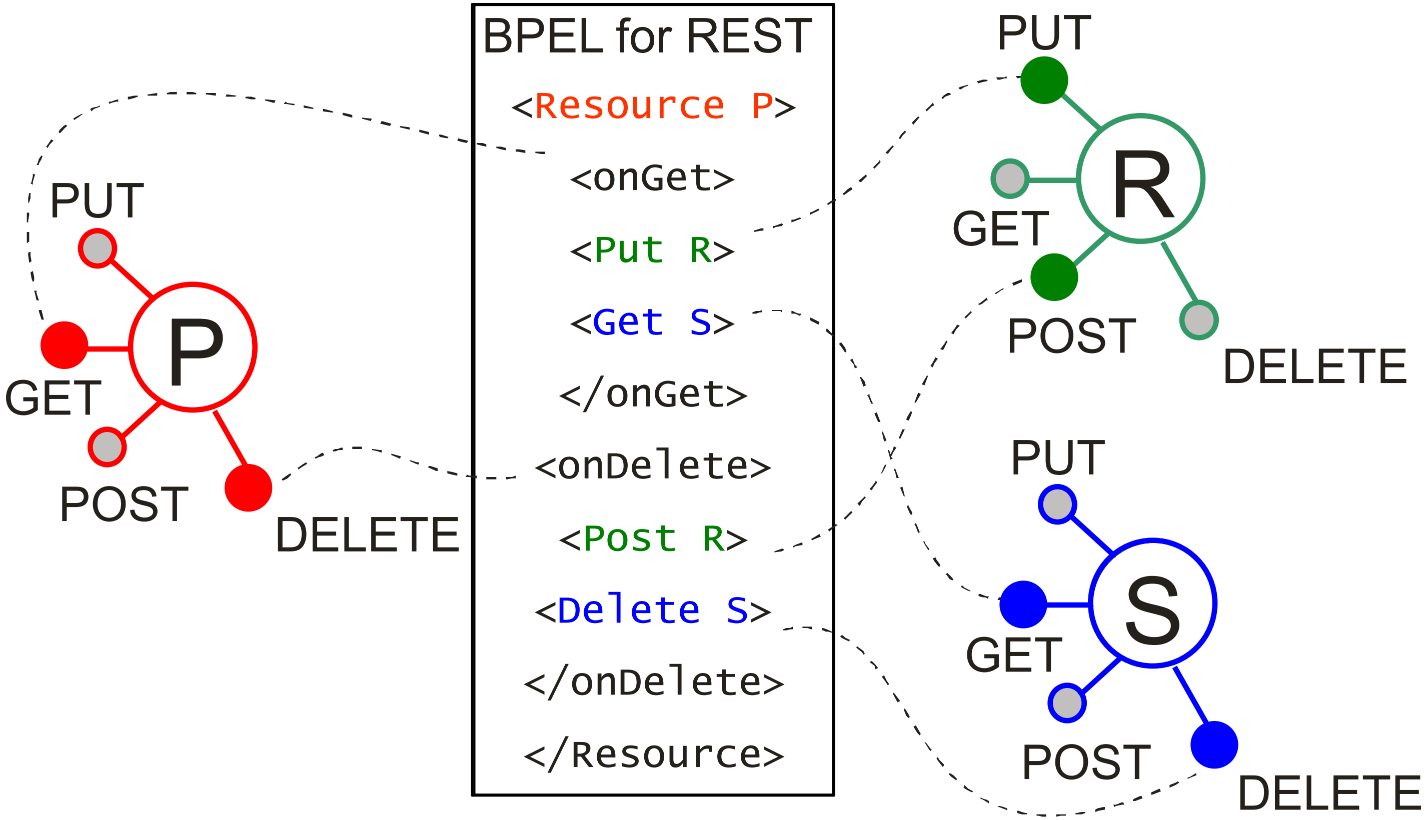


BPEL for REST

```
...
<Put R>
<Get R>
...
<Post R>
<Delete R>
...
```

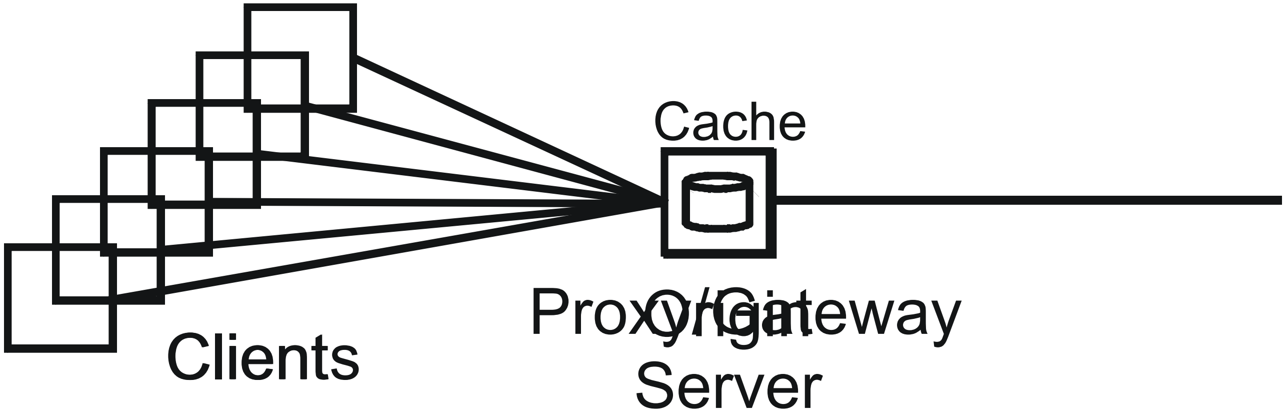
BPEL for REST – Resource Block

- Dynamically publish resources from BPEL processes and handle client requests



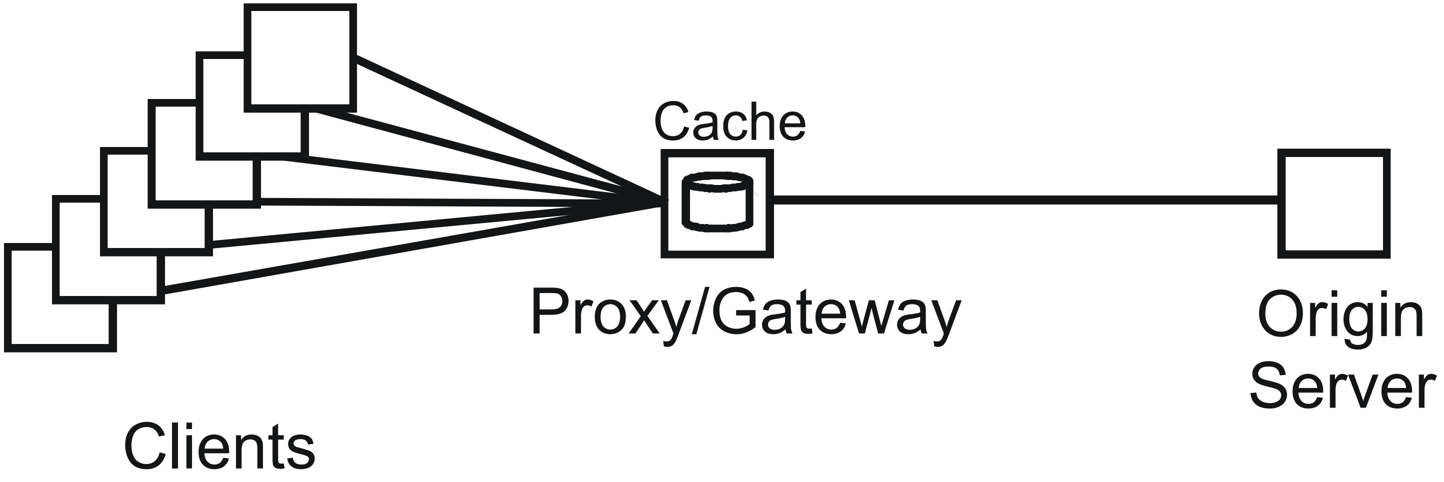
```
BPEL for REST
<Resource P>
<onGet>
<Put R>
<Get S>
</onGet>
<onDelete>
<Post R>
<Delete S>
</onDelete>
</Resource>
```

REST Scalability



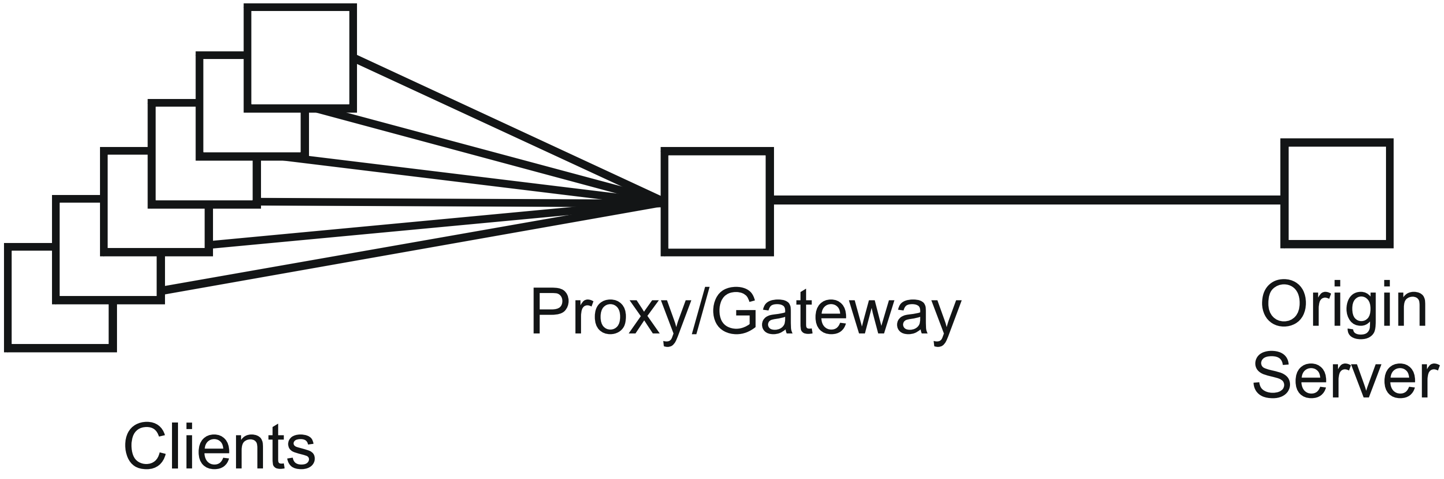
- One example of REST middleware is to help with the scalability of a server, which may need to service a very large number of clients

REST Scalability



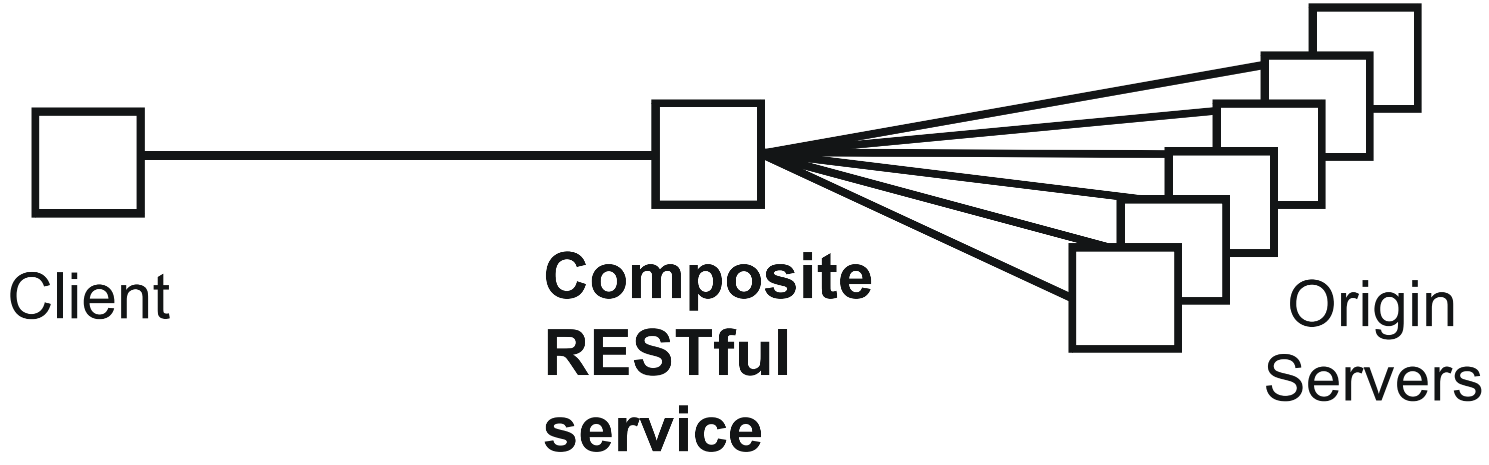
- One example of REST middleware is to help with the scalability of a server, which may need to service a very large number of clients

REST Composition



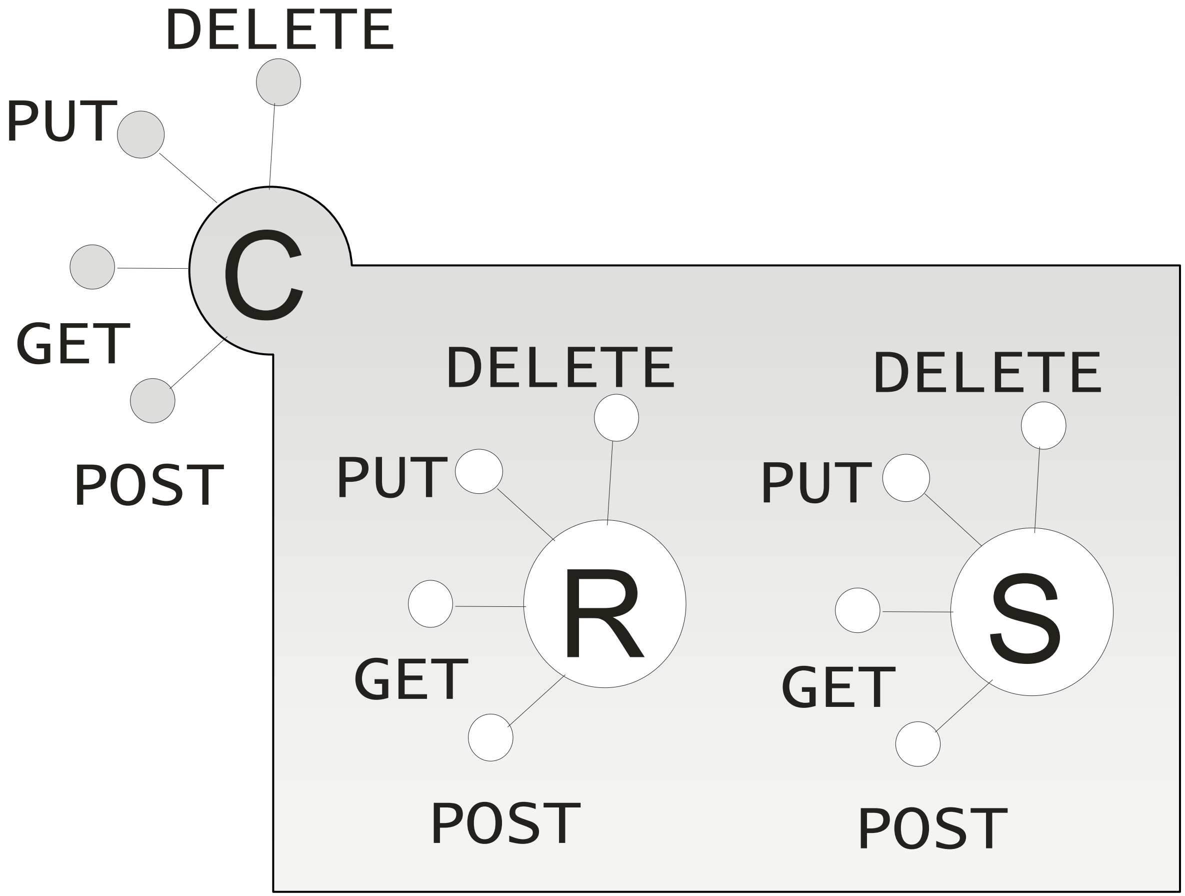
- Composition shifts the attention to the client which should consume and aggregate from many servers

REST Composition



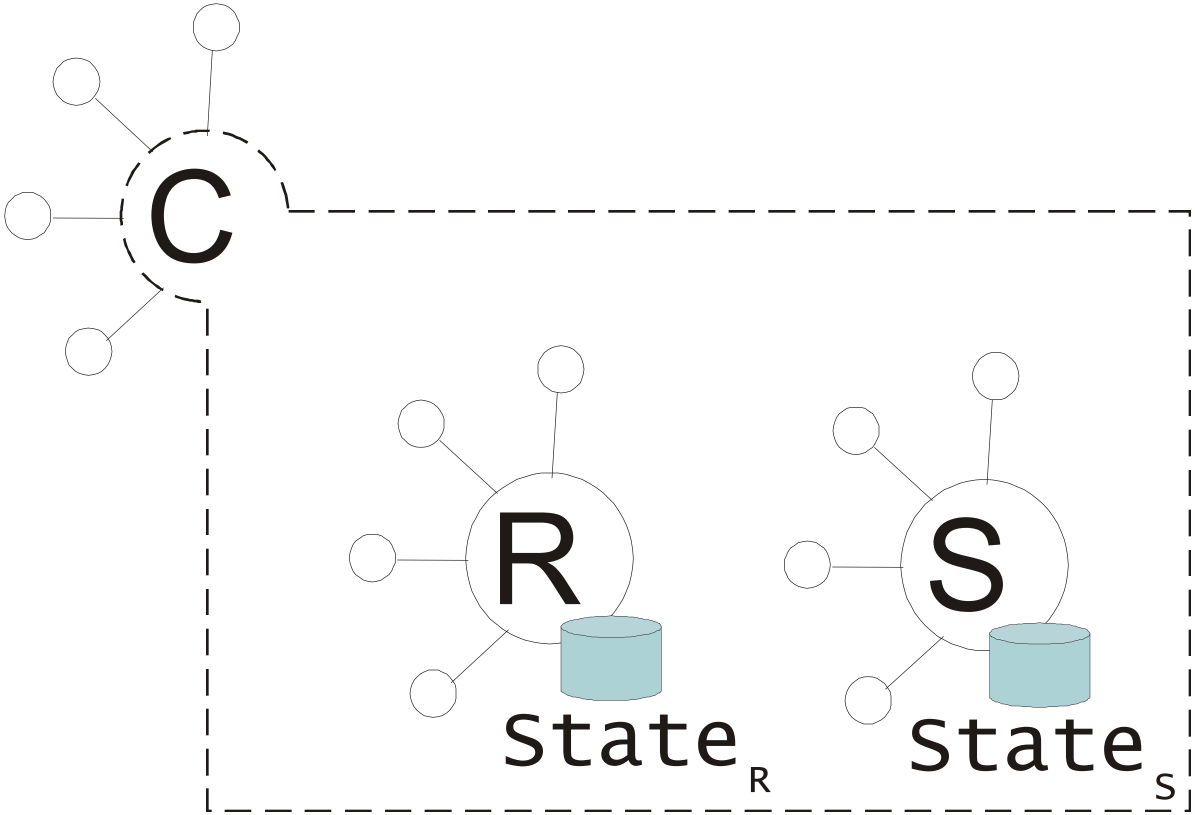
- The “proxy” intermediate element which aggregates the resources provided by multiple servers plays the role of a composite RESTful service
- Can/Should we implement it with BPM?

Composite Resources



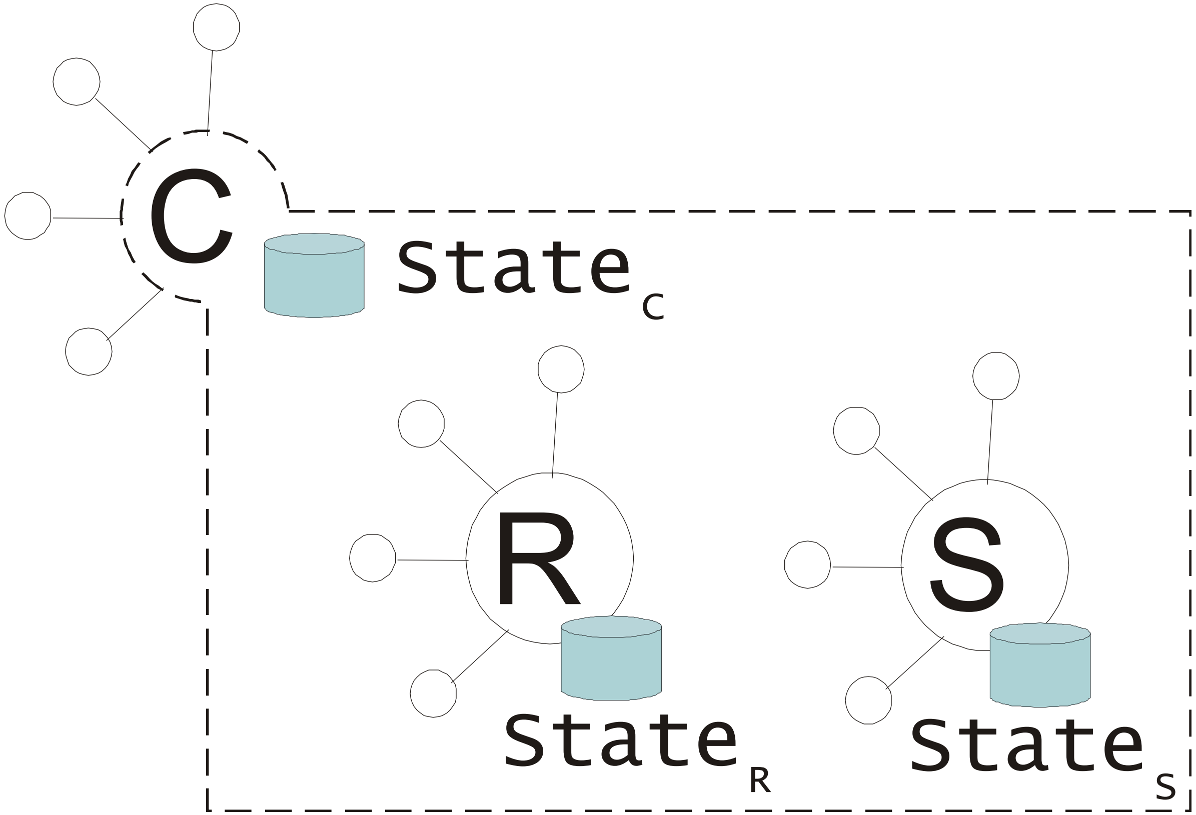
Composite Resources

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

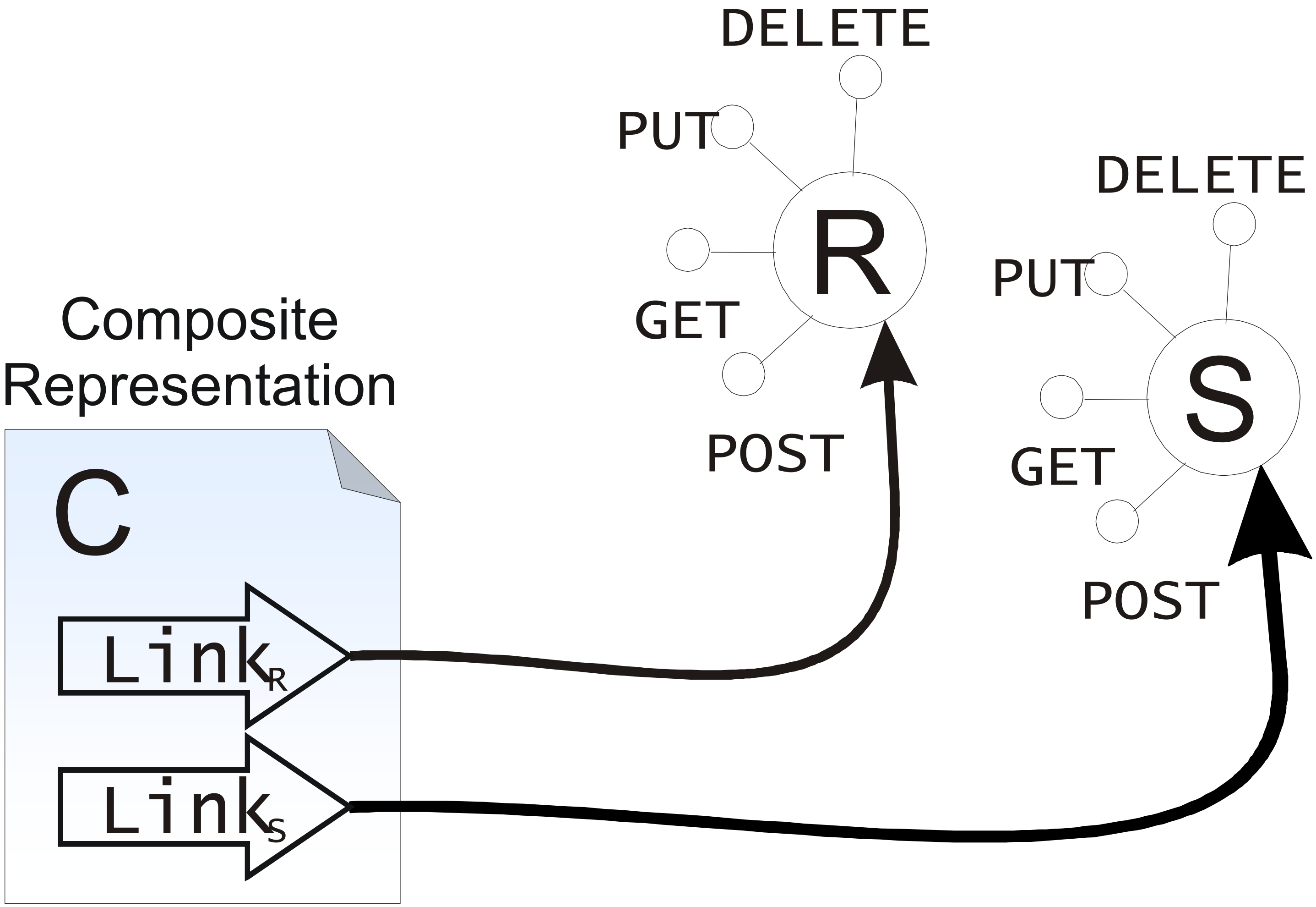


Composite Resources

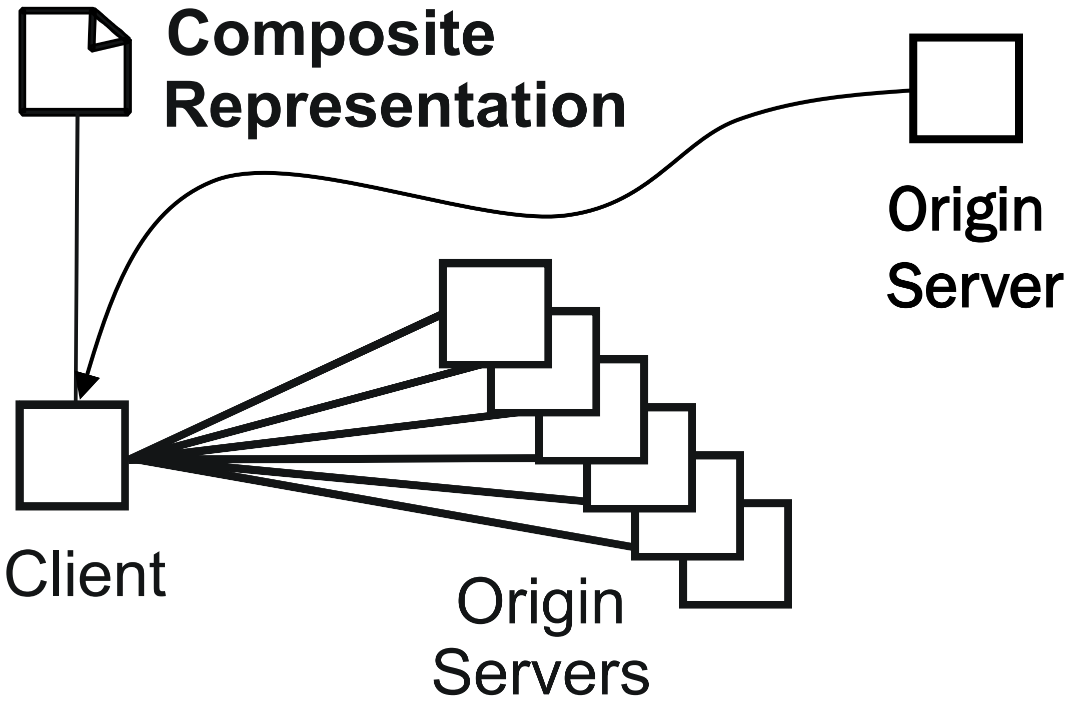
- The composite resource augments (or caches) the state of its component resources



Composite Representations

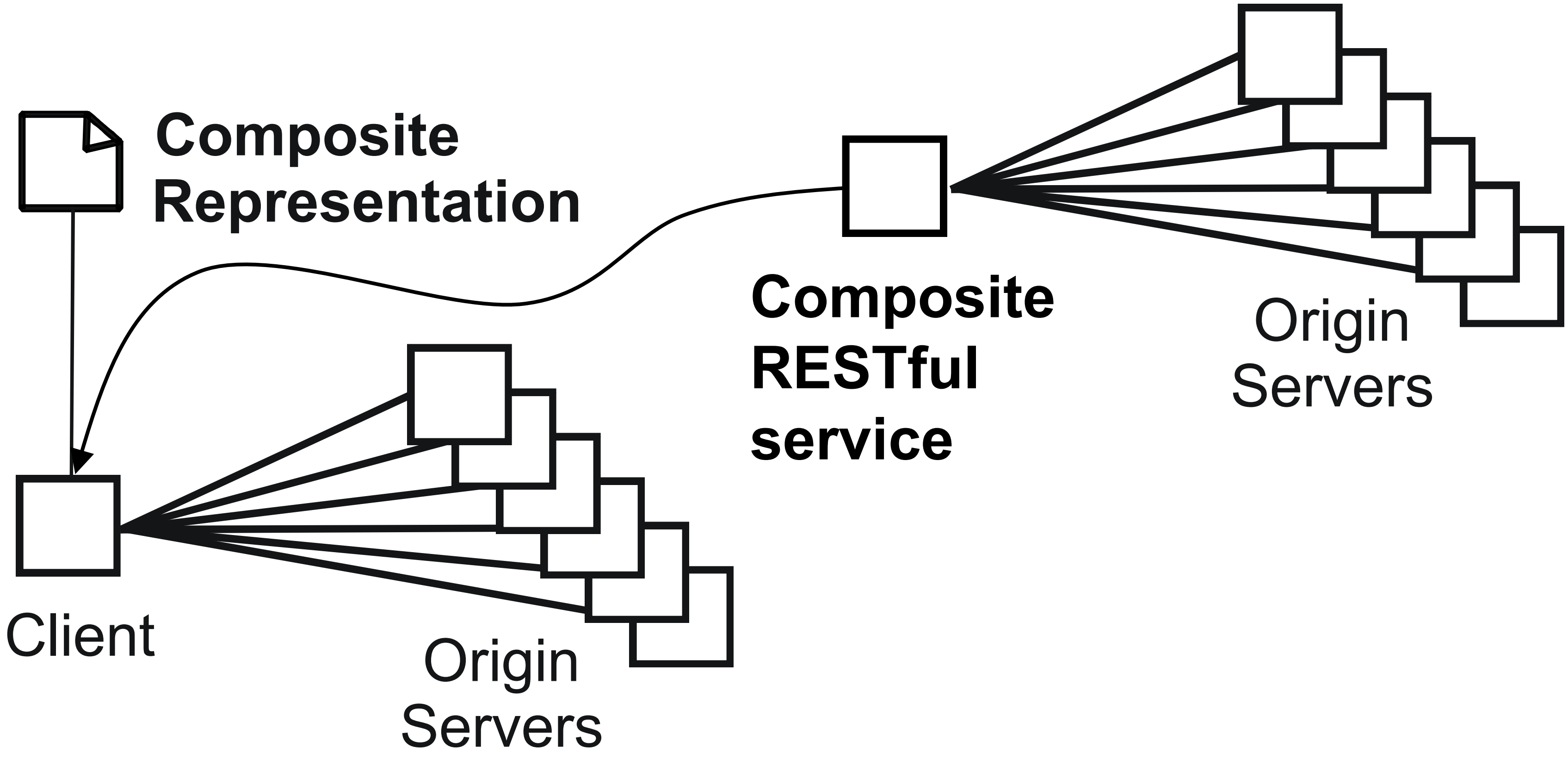


Composite Representation



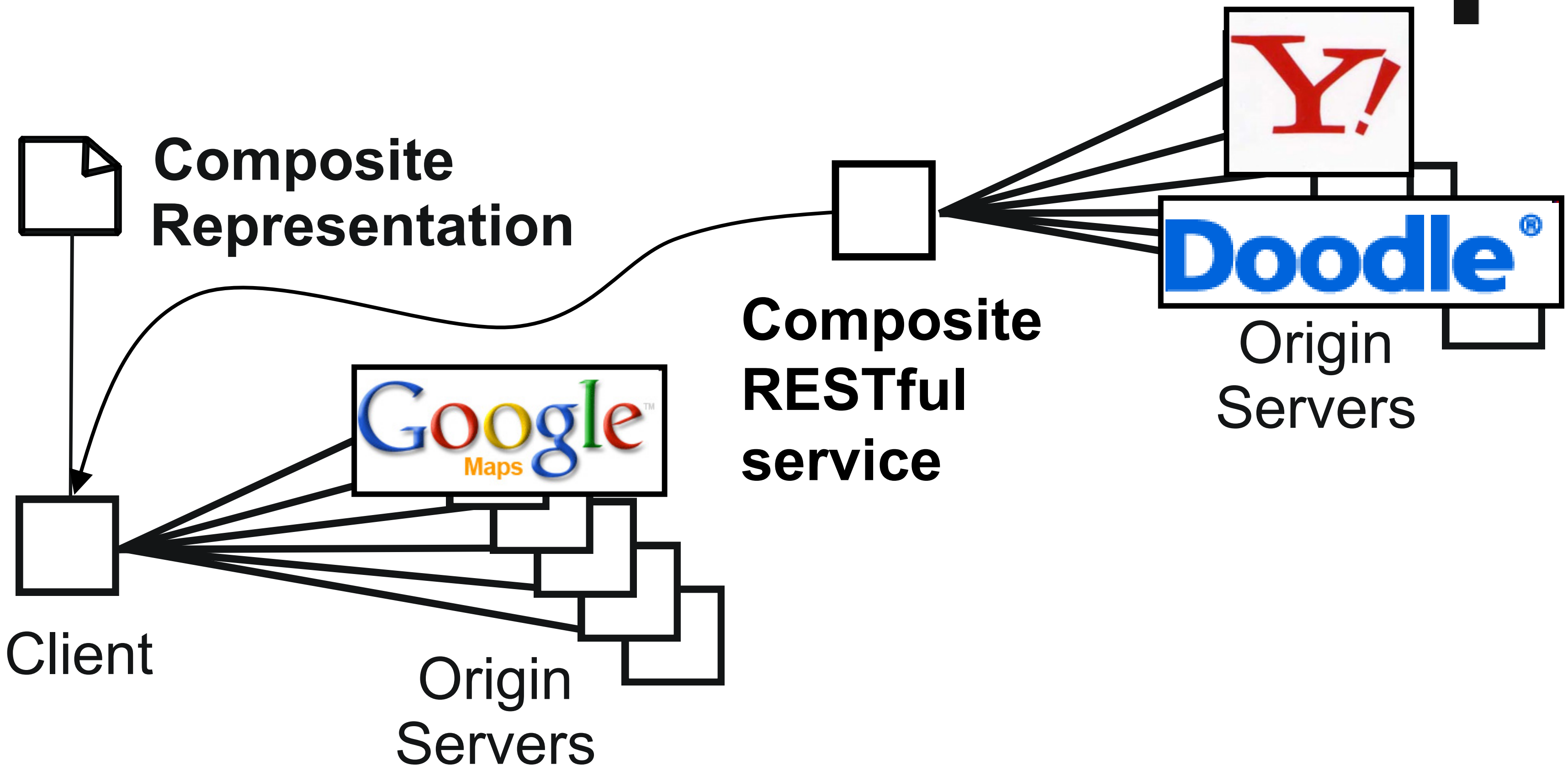
- A composite representation is interpreted by the client that follows its hyperlinks and aggregates the state of the referenced component resources

Bringing it all together



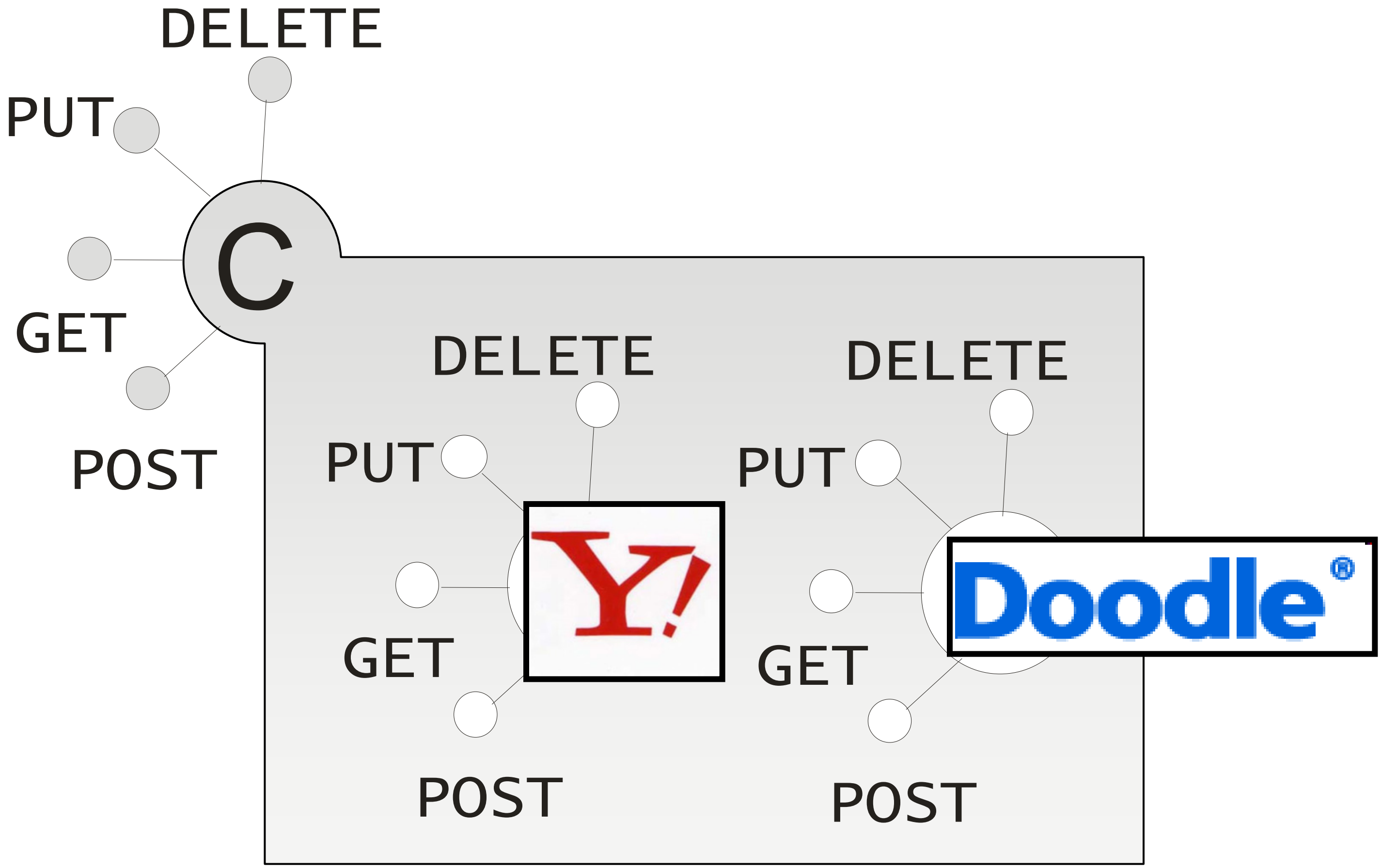
- A composite representation can be produced by a composite service too

Doodle Map Example

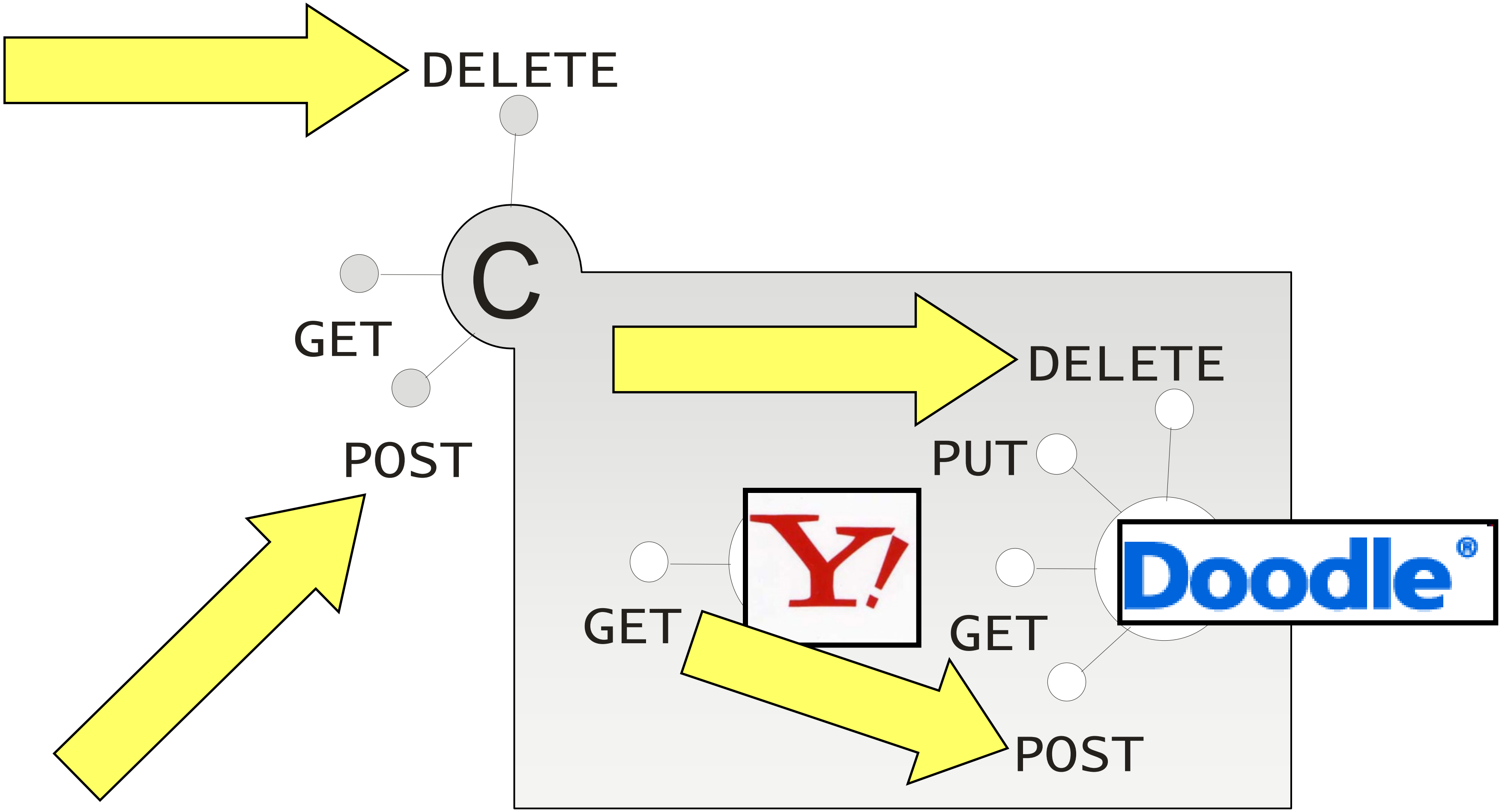


- Vote on a meeting place based on its geographic location

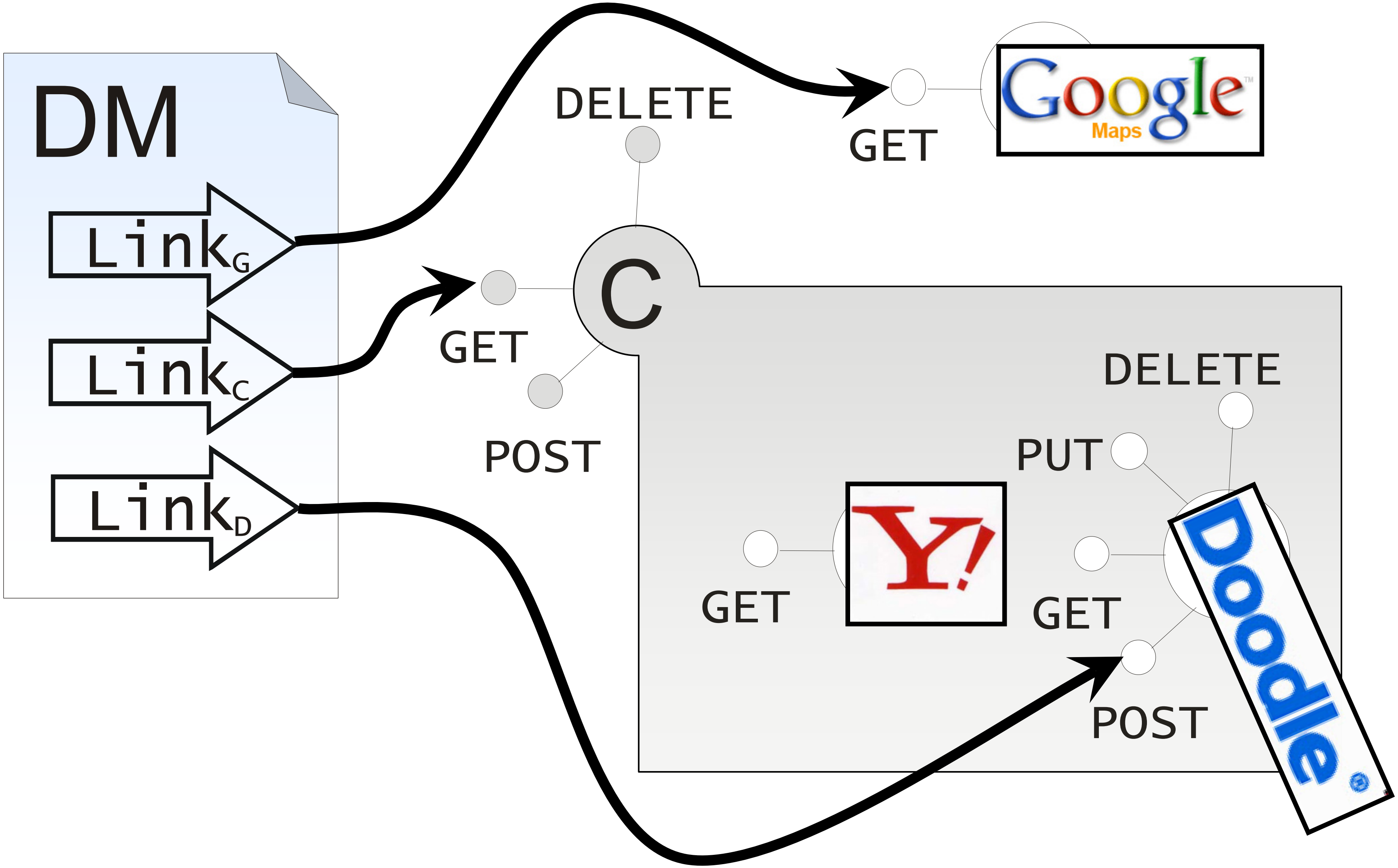
1. Composite Resource



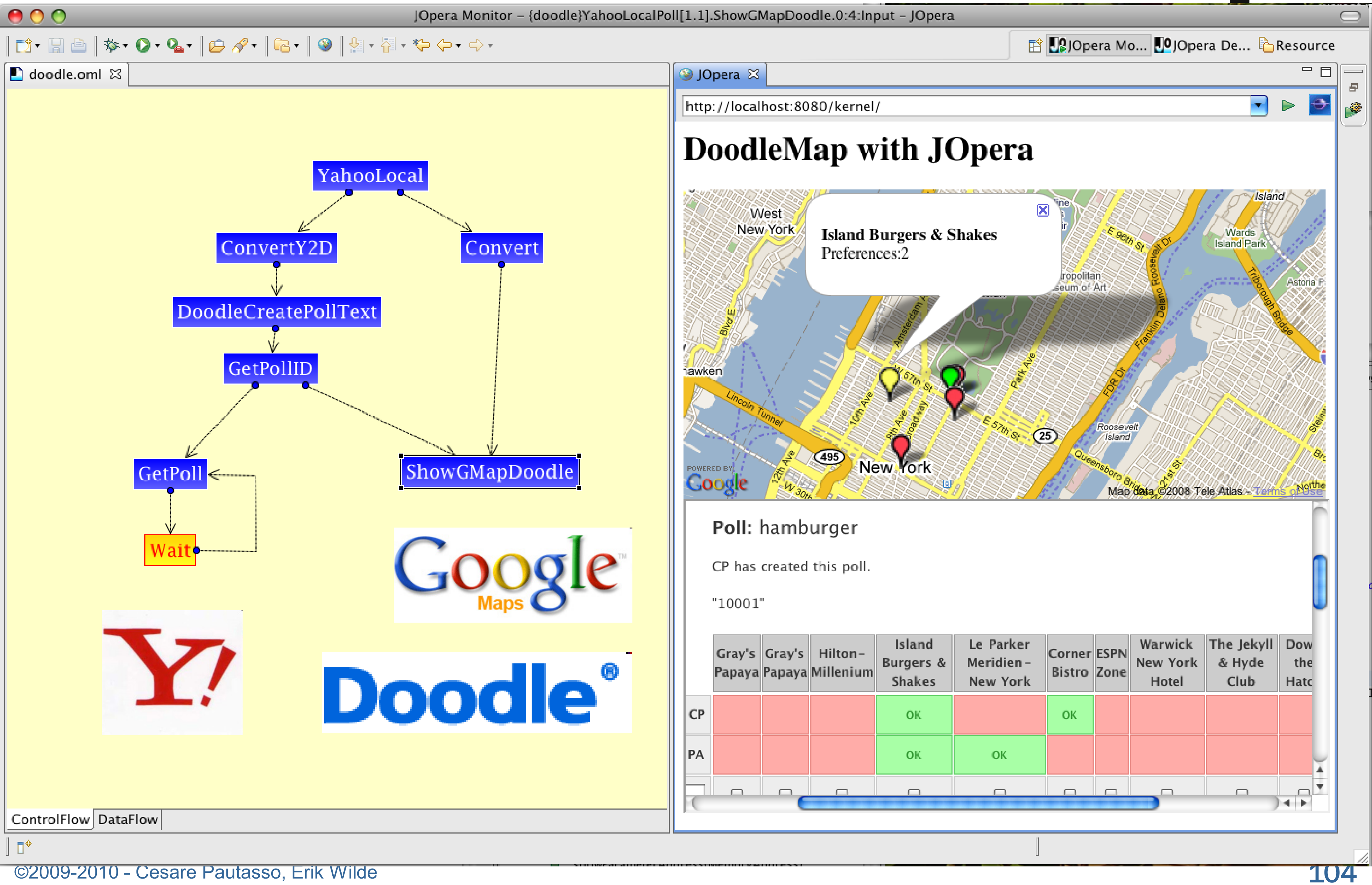
1. Composite Resource



2. Composite Representation



RESTful Composition Example



DoodleMap with JOpera

Island Burgers & Shakes
Preferences:2

Poll: hamburger

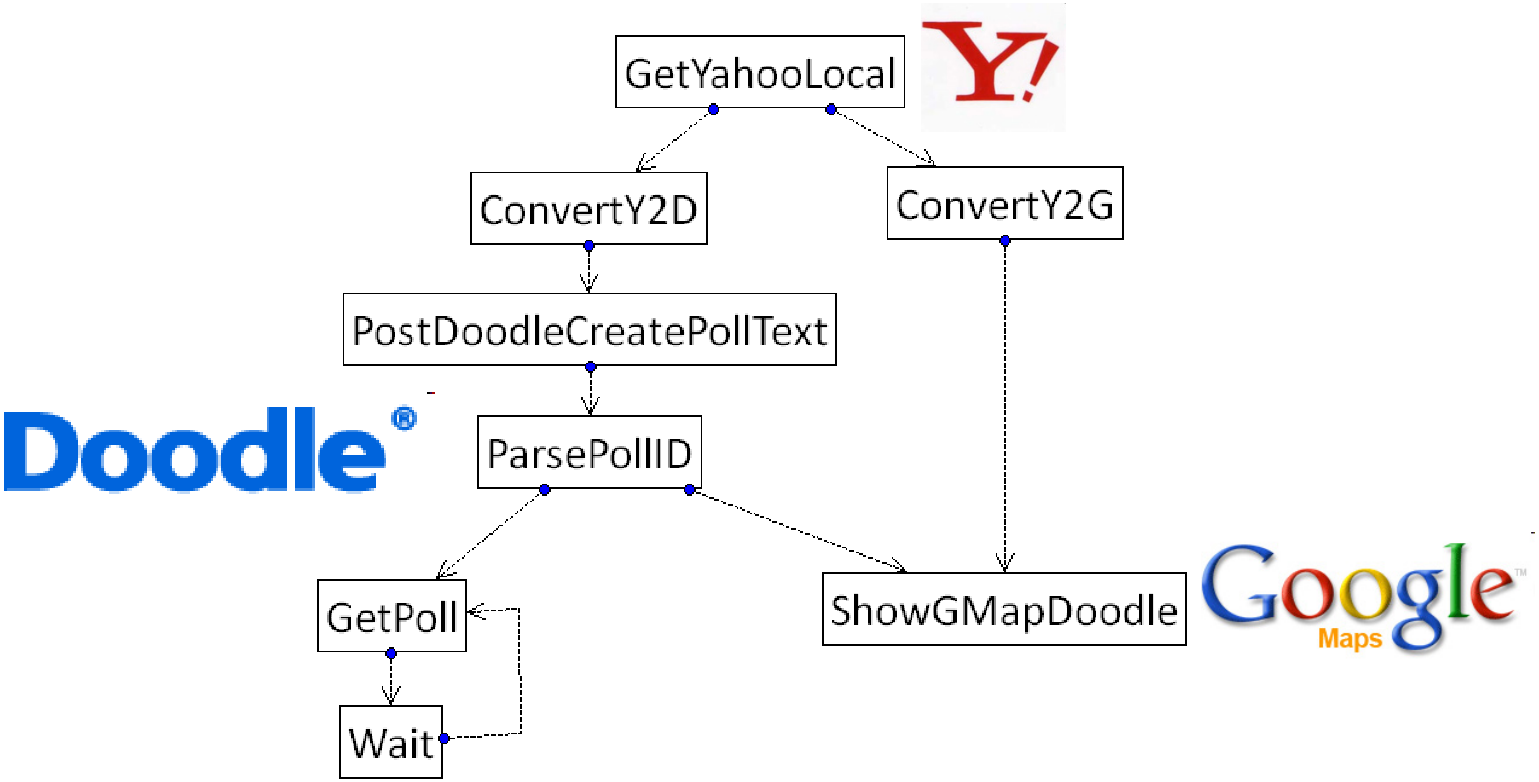
CP has created this poll.

"10001"

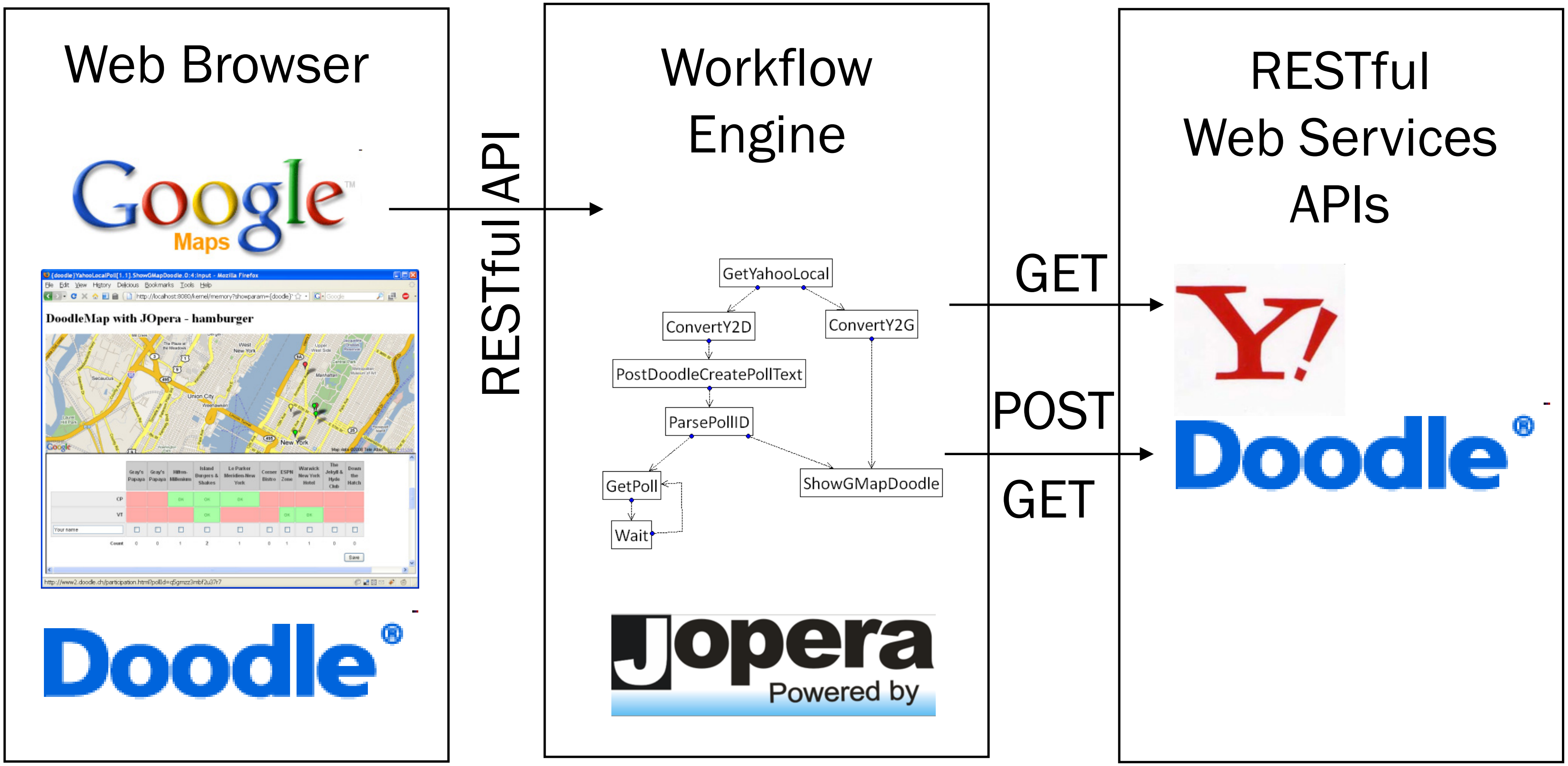
	Gray's Papaya	Gray's Papaya	Hilton-Millenium	Island Burgers & Shakes	Le Parker Meridien-New York	Corner Bistro	ESPN Zone	Warwick New York Hotel	The Jekyll & Hyde Club	Dow the Hatc
CP				OK		OK				
PA				OK	OK					
	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐

Example: Doodle Map Mashup

- Setup a Doodle with Yahoo! Local search and visualize the results of the poll on Google Maps

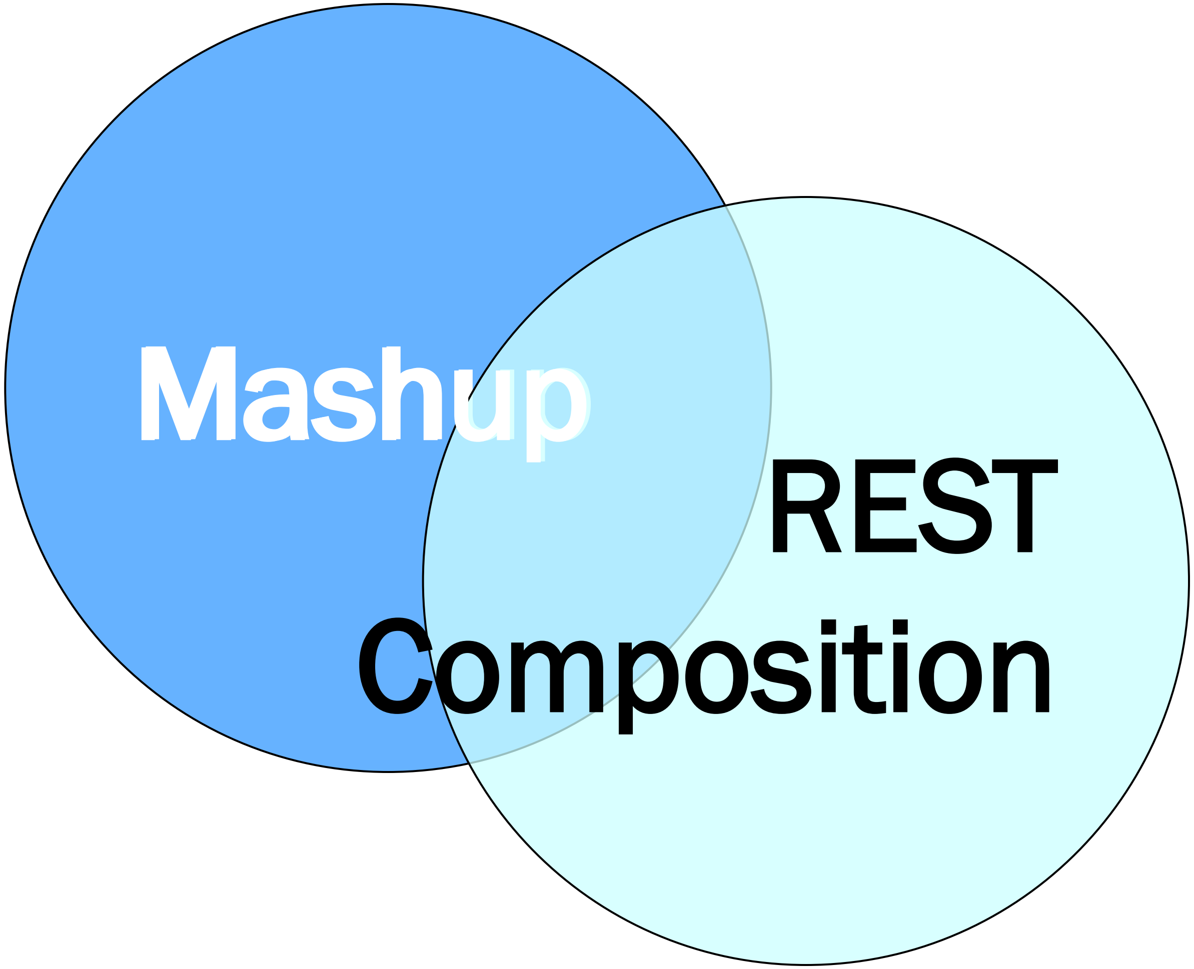


Doodle Map Mashup Architecture



Cesare Pautasso, RESTful Web Service Composition with JOpera, Proc. of the International Conference on Software Composition (SC 2009), Zurich, Switzerland, July 2009.

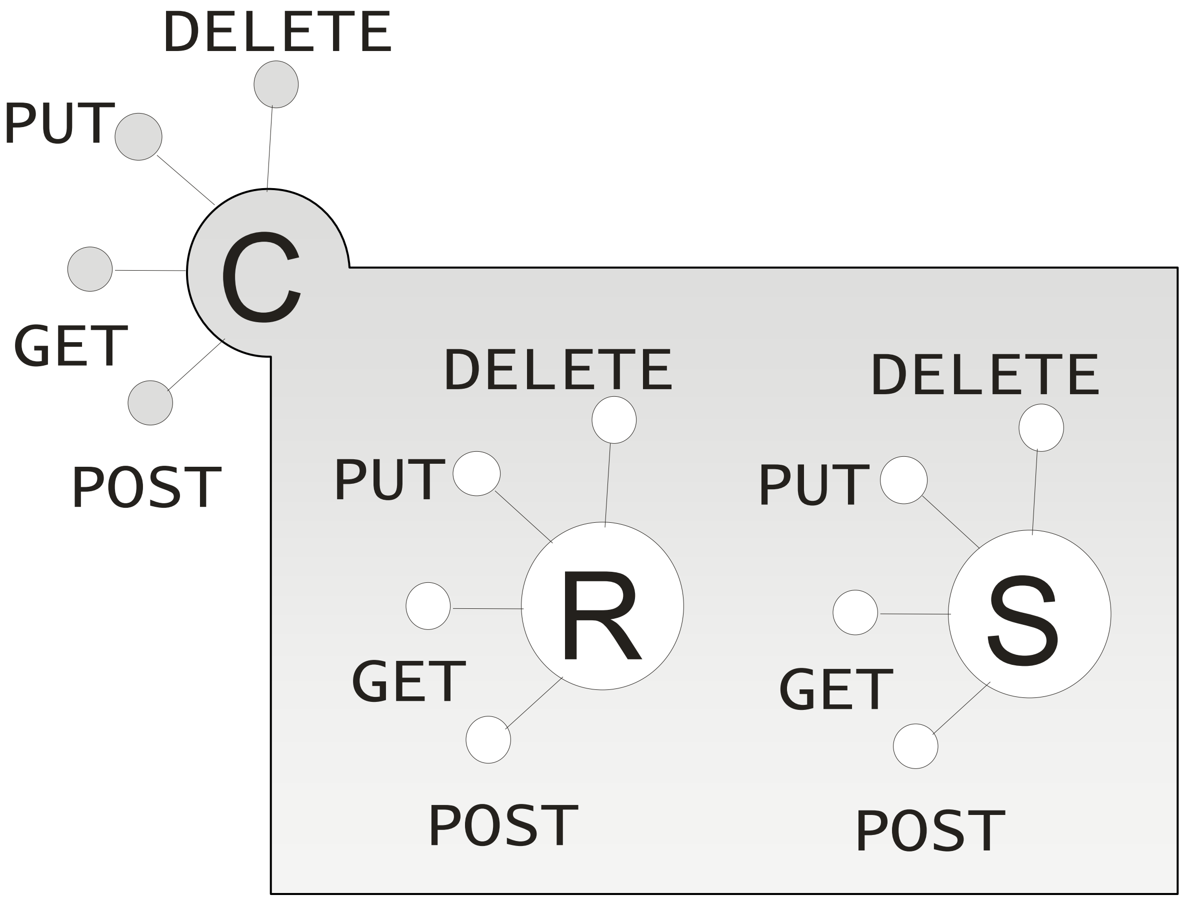
Was it just a mashup?



(It depends on the definition of Mashup)

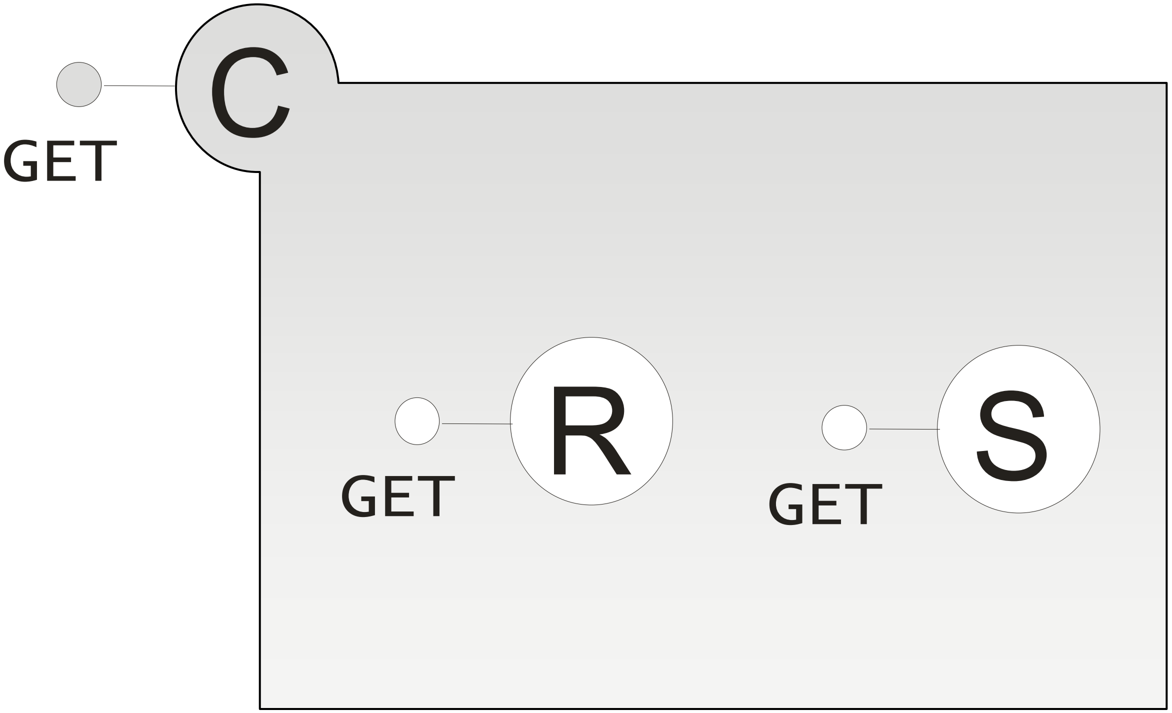
Moving state around

- Read-only vs. Read/Write



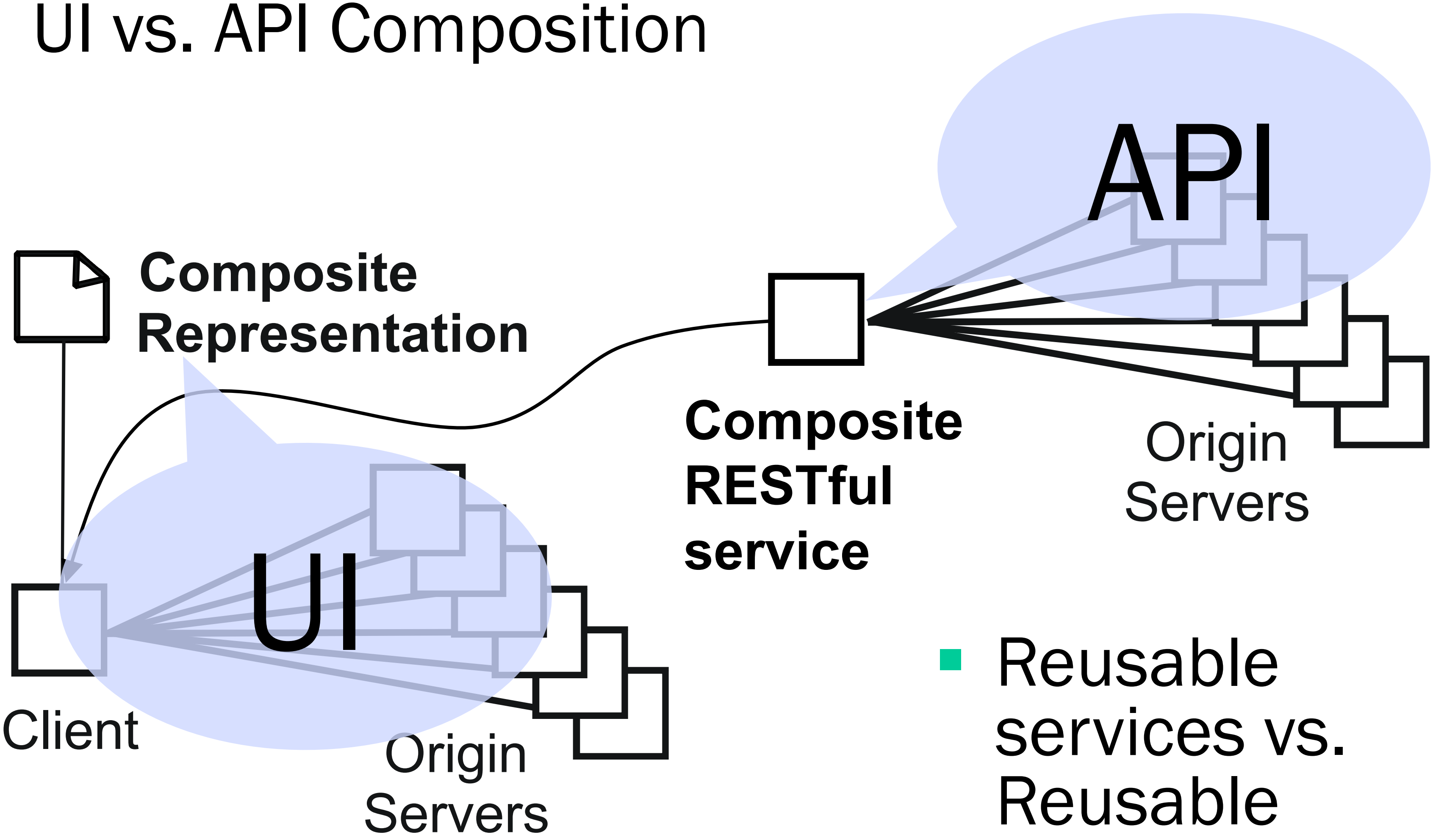
Simply aggregating data (feeds)

- Read-only vs. Read/write



Is your composition reusable?

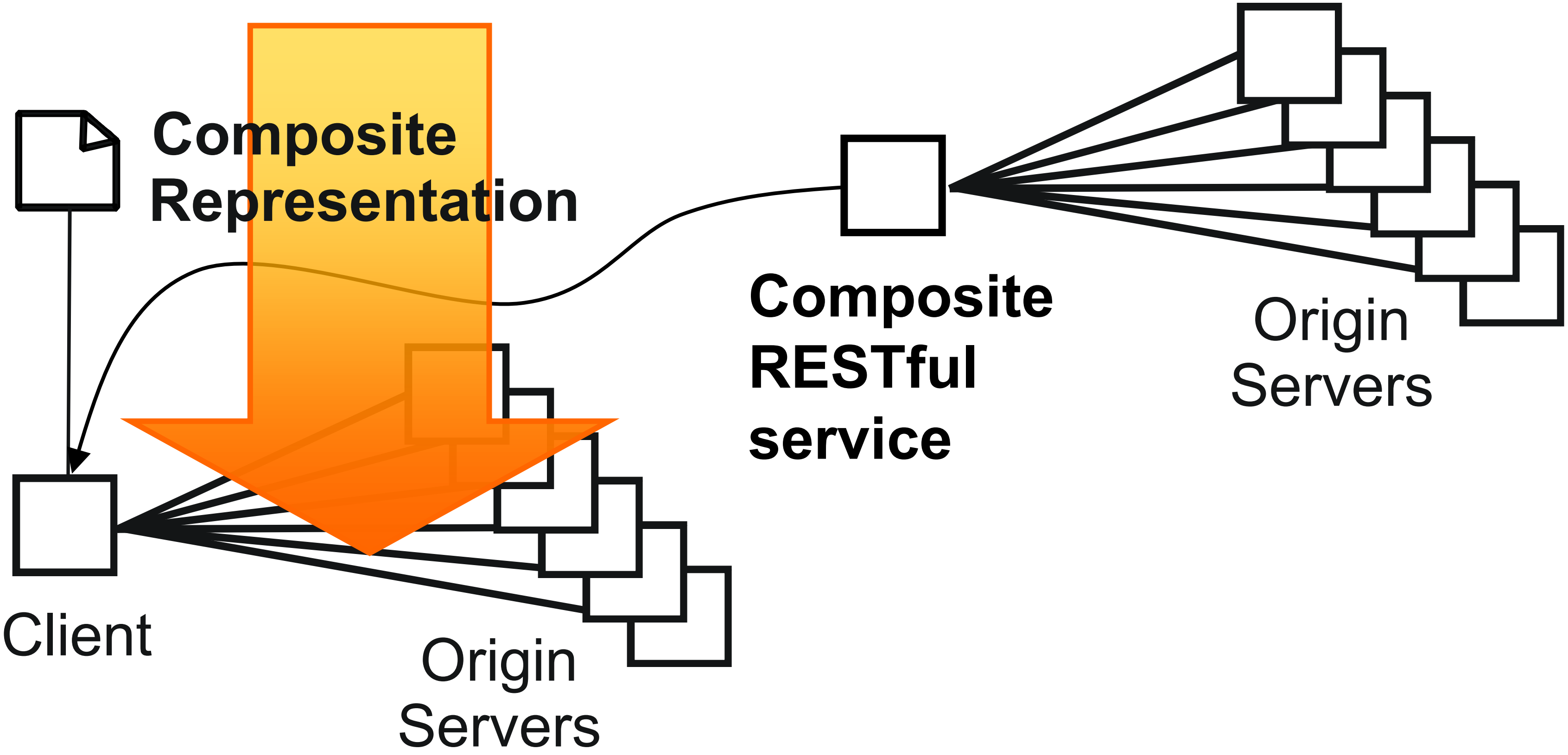
- UI vs. API Composition



- Reusable services vs. Reusable Widgets

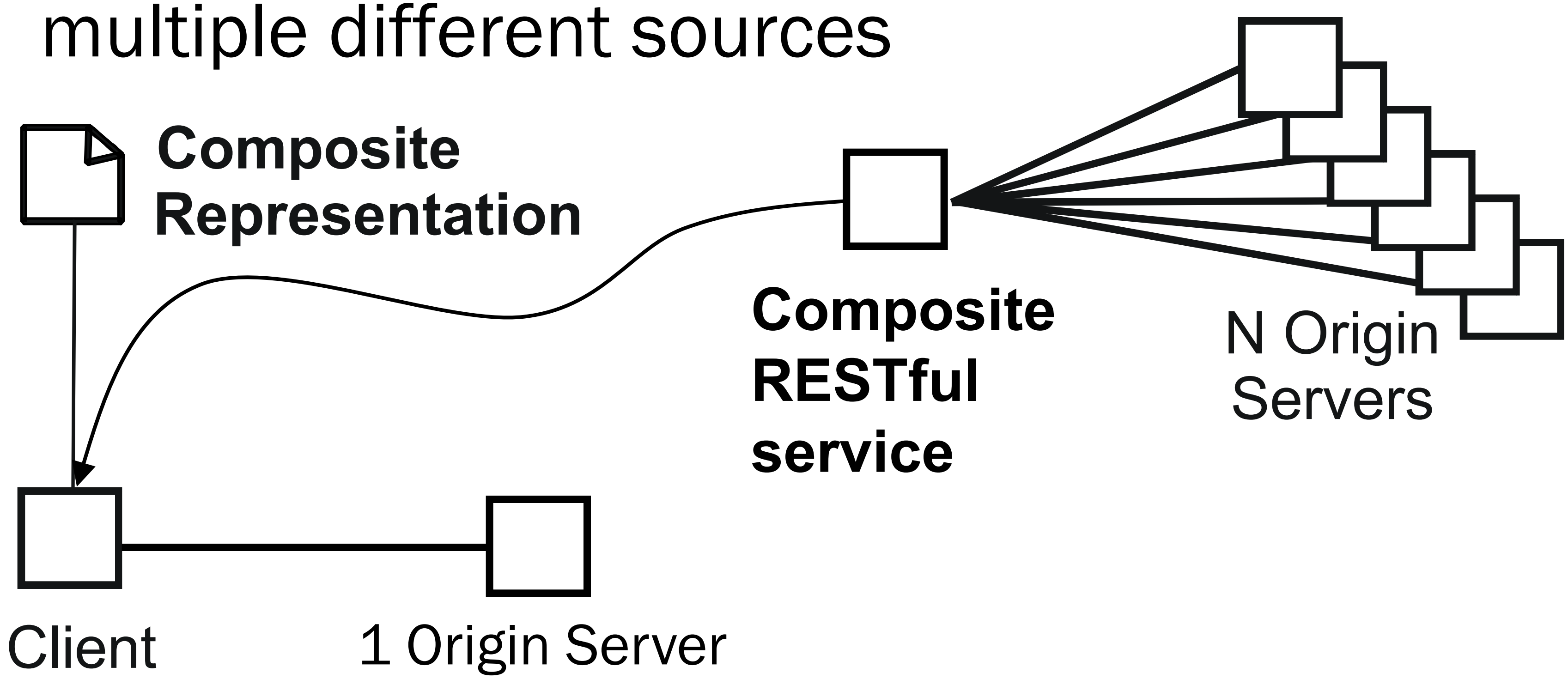
Single-Origin Sandbox

- Can you always do this from a web browser?

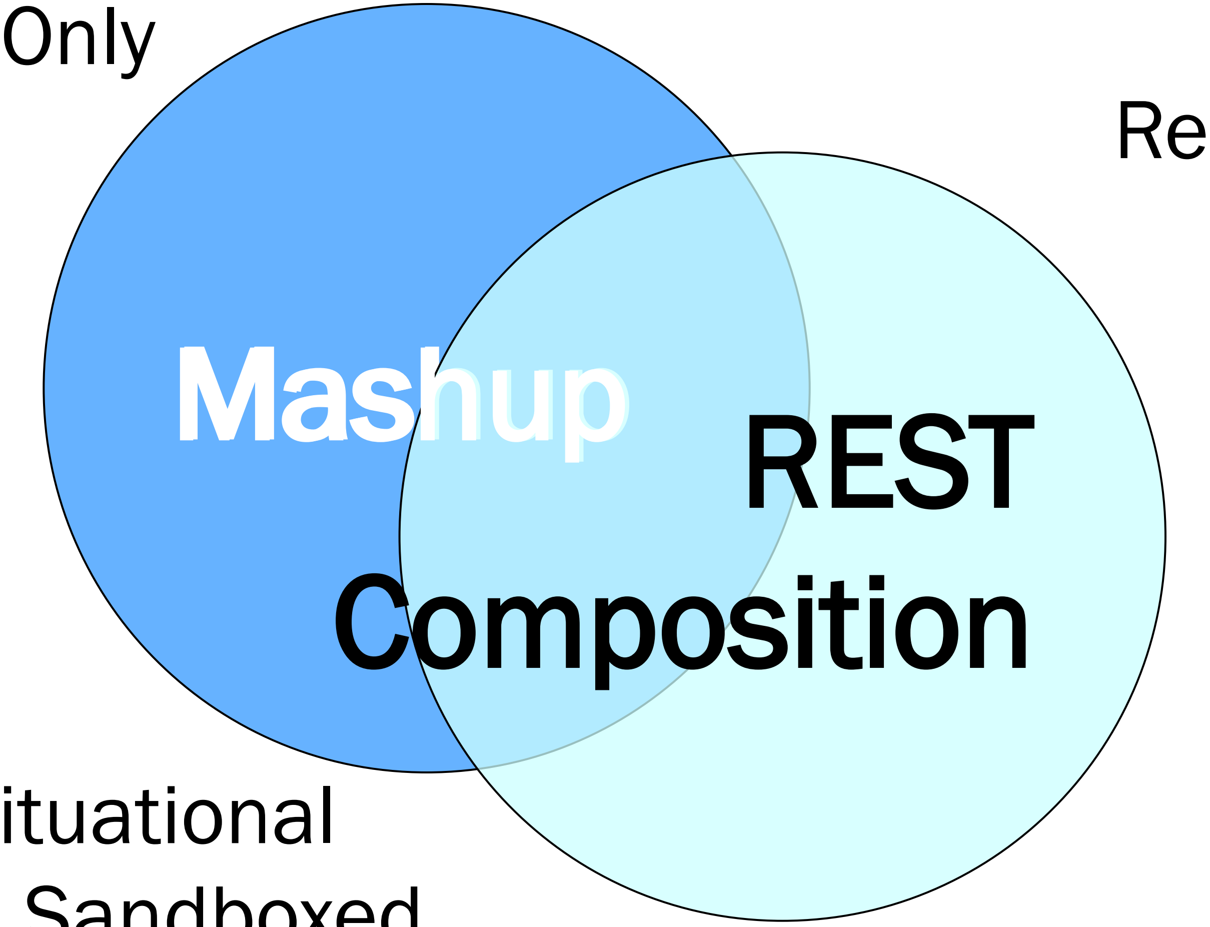


Single-Origin Sandbox

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



Complementary



Read-Only

Read/Write

UI

Mashup

REST

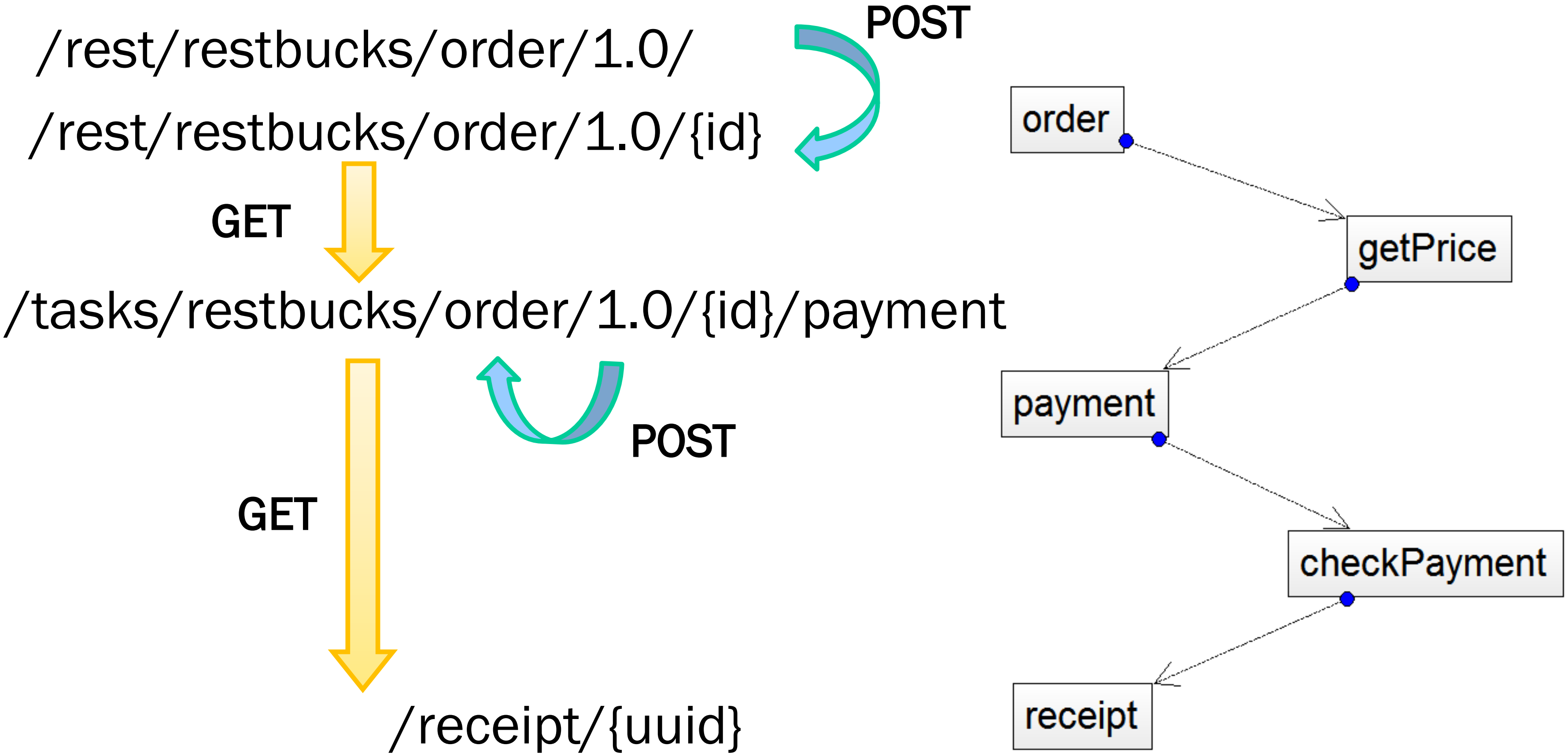
Composition

API

Reusable Service

Situational Sandboxed

TinyRESTBucks Example



/rest/restbucks/order/1.0/

POST

/rest/restbucks/order/1.0/{id}

GET

/tasks/restbucks/order/1.0/{id}/payment

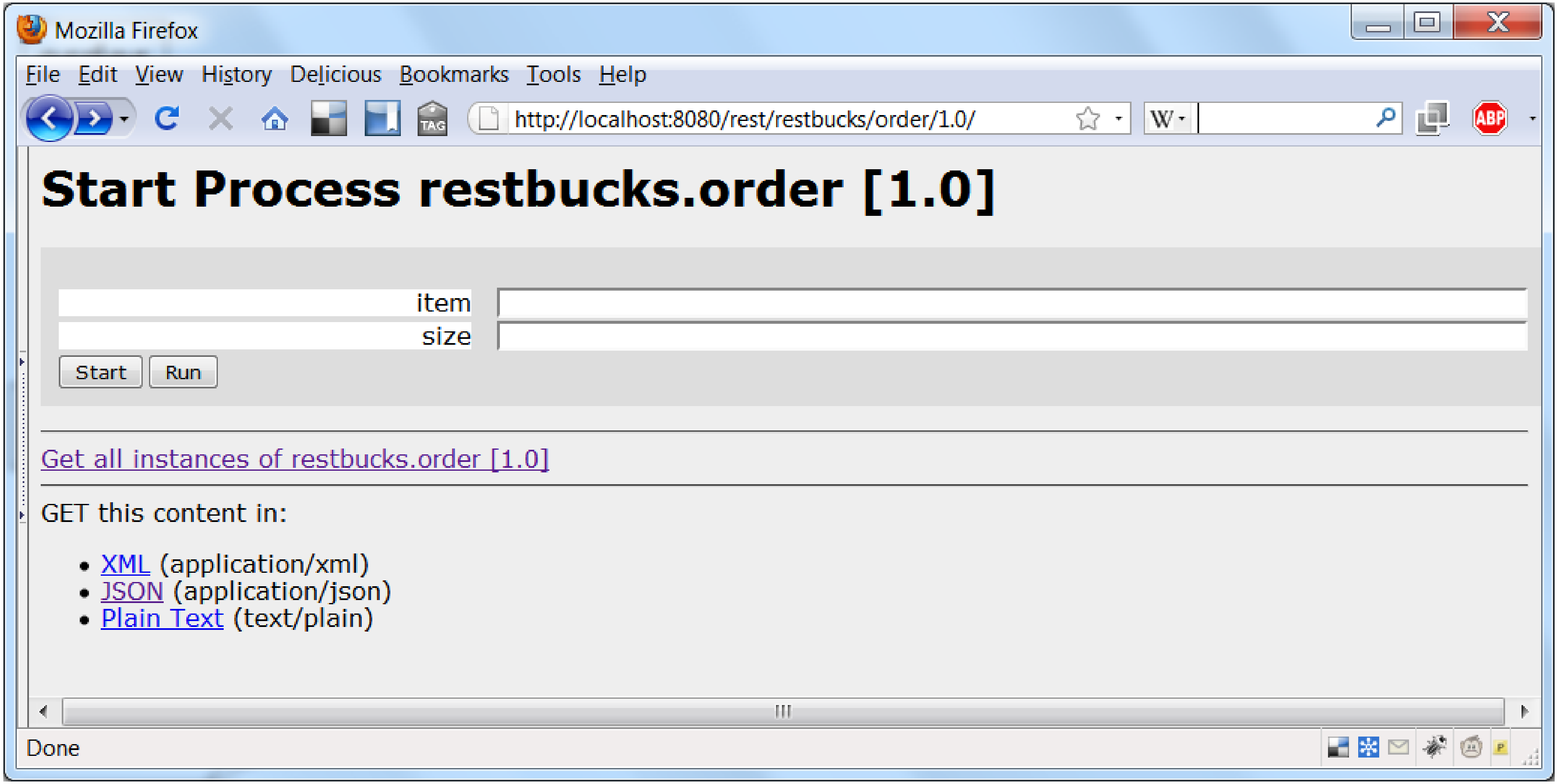
POST

GET

/receipt/{uuid}

Instantiating a process

GET /rest/restbucks/order/1.0/



http://localhost:8080/rest/restbucks/order/1.0/

Start Process restbucks.order [1.0]

item	
size	

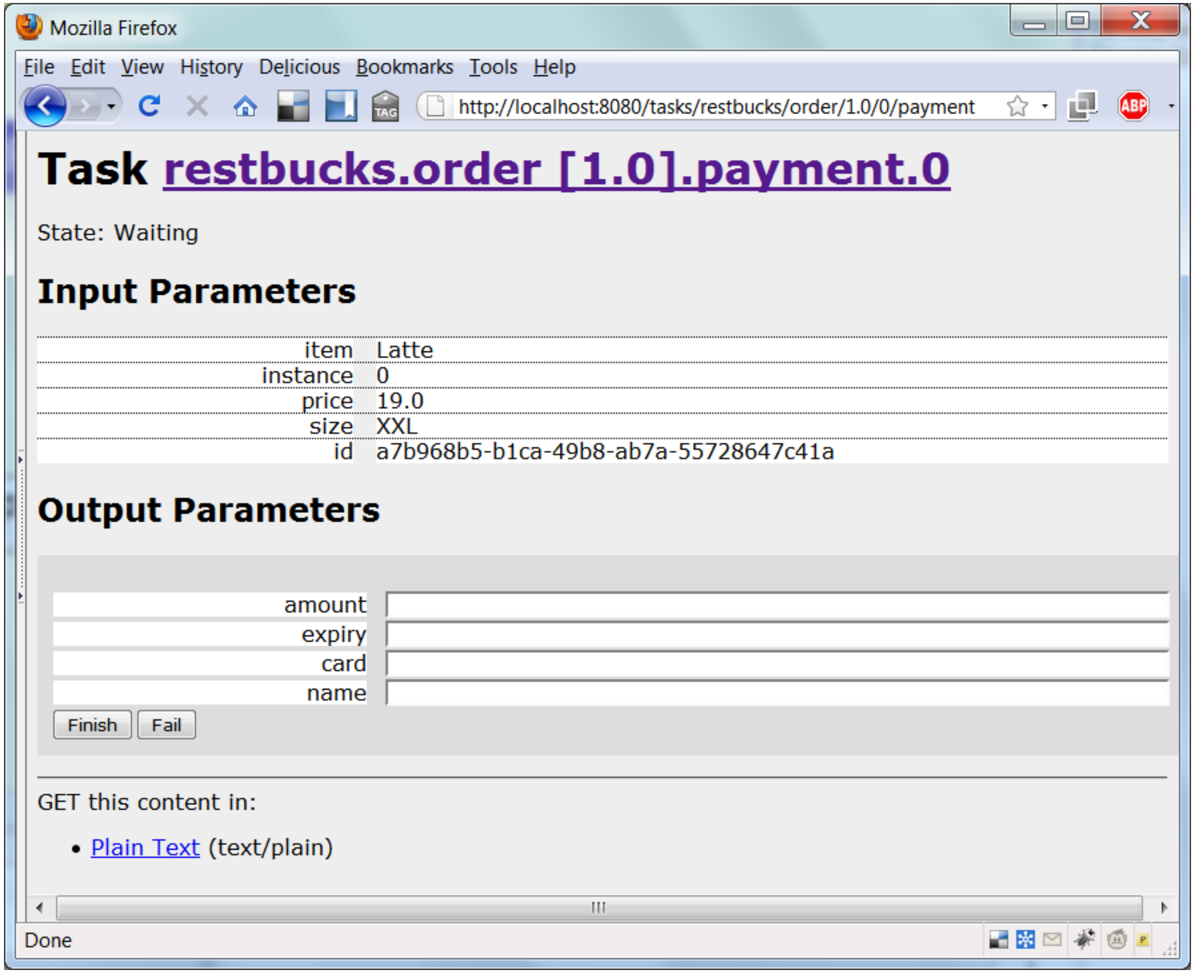
Get all instances of restbucks.order [1.0]

GET this content in:

- XML (application/xml)
- JSON (application/json)
- Plain Text (text/plain)

Interacting with a task

GET /rest/restbucks/order/1.0/0/payment



http://localhost:8080/tasks/restbucks/order/1.0/0/payment

Task restbucks.order [1.0].payment.0

State: Waiting

Input Parameters

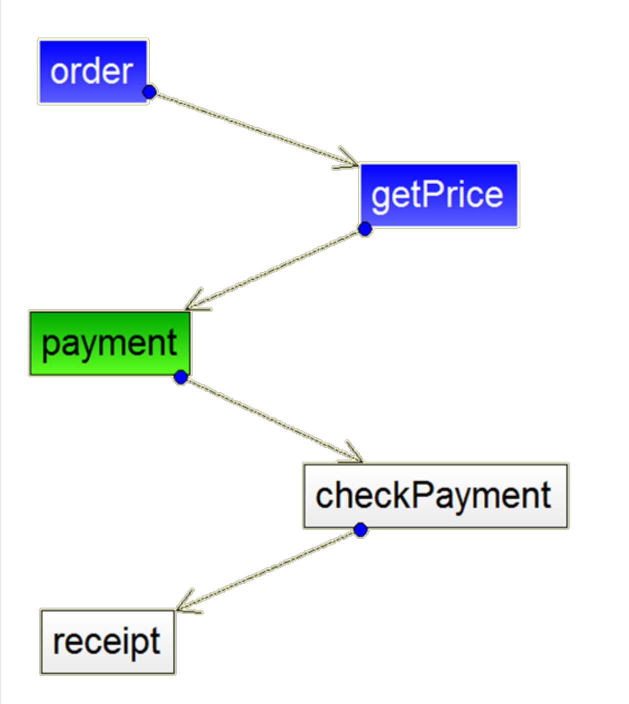
item	Latte
instance	0
price	19.0
size	XXL
id	a7b968b5-b1ca-49b8-ab7a-55728647c41a

Output Parameters

amount	
expiry	
card	
name	

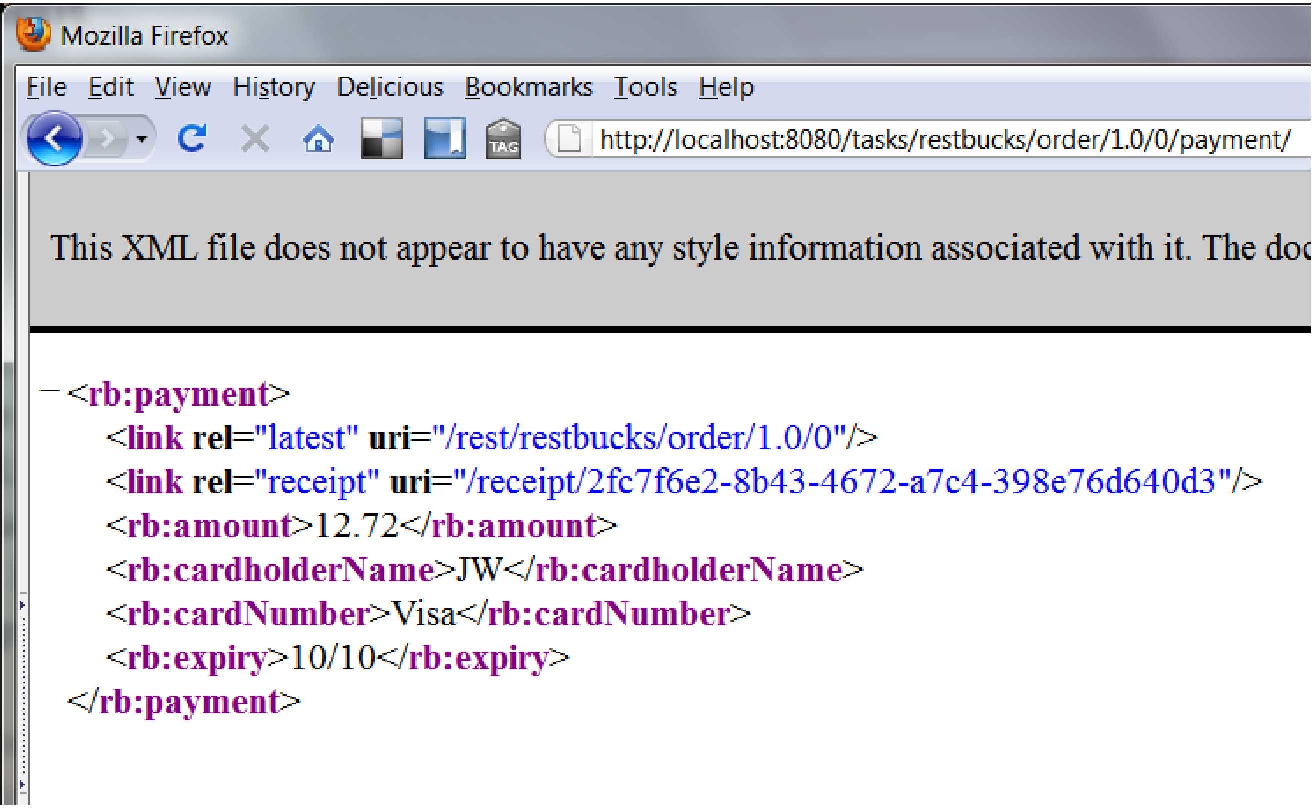
GET this content in:

- Plain Text (text/plain)



Interacting with a task

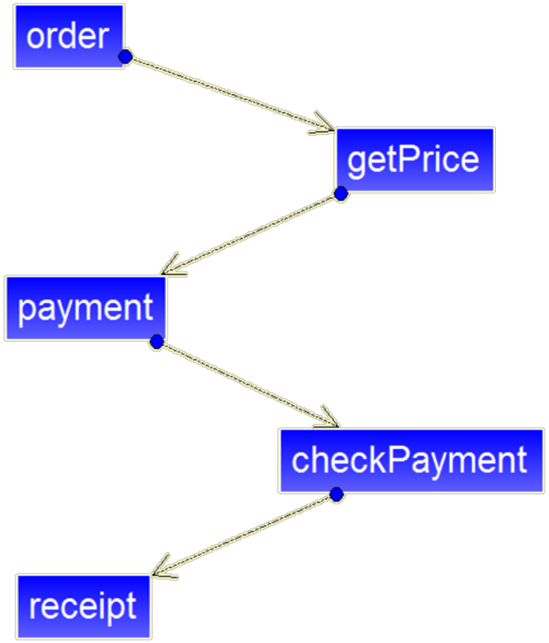
POST /rest/restbucks/order/1.0/0/payment



Mozilla Firefox
File Edit View History Delicious Bookmarks Tools Help
http://localhost:8080/tasks/restbucks/order/1.0/0/payment/

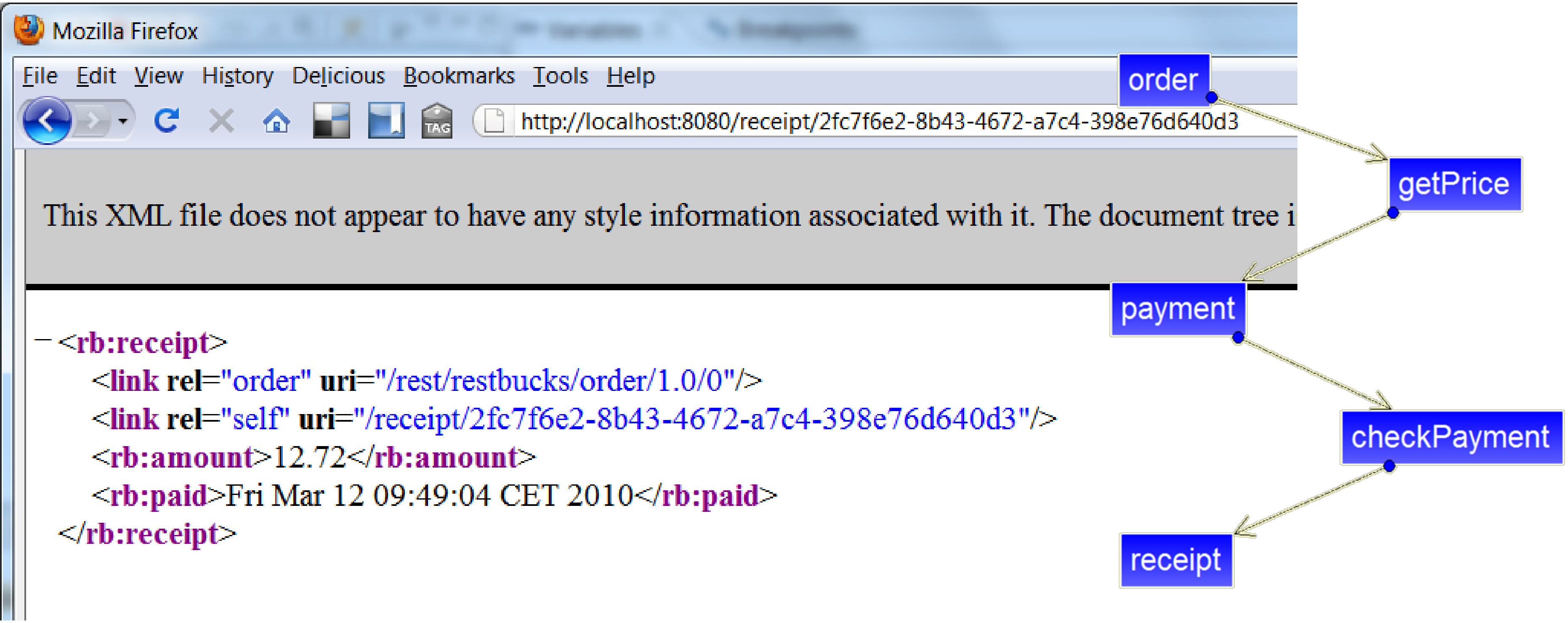
This XML file does not appear to have any style information associated with it. The doc

```
- <rb:payment>
    <link rel="latest" uri="/rest/restbucks/order/1.0/0"/>
    <link rel="receipt" uri="/receipt/2fc7f6e2-8b43-4672-a7c4-398e76d640d3"/>
    <rb:amount>12.72</rb:amount>
    <rb:cardholderName>JW</rb:cardholderName>
    <rb:cardNumber>Visa</rb:cardNumber>
    <rb:expiry>10/10</rb:expiry>
  </rb:payment>
```



Interacting with a resource

GET /receipt/2fc7f6e2-8b43-4672-a7c4...

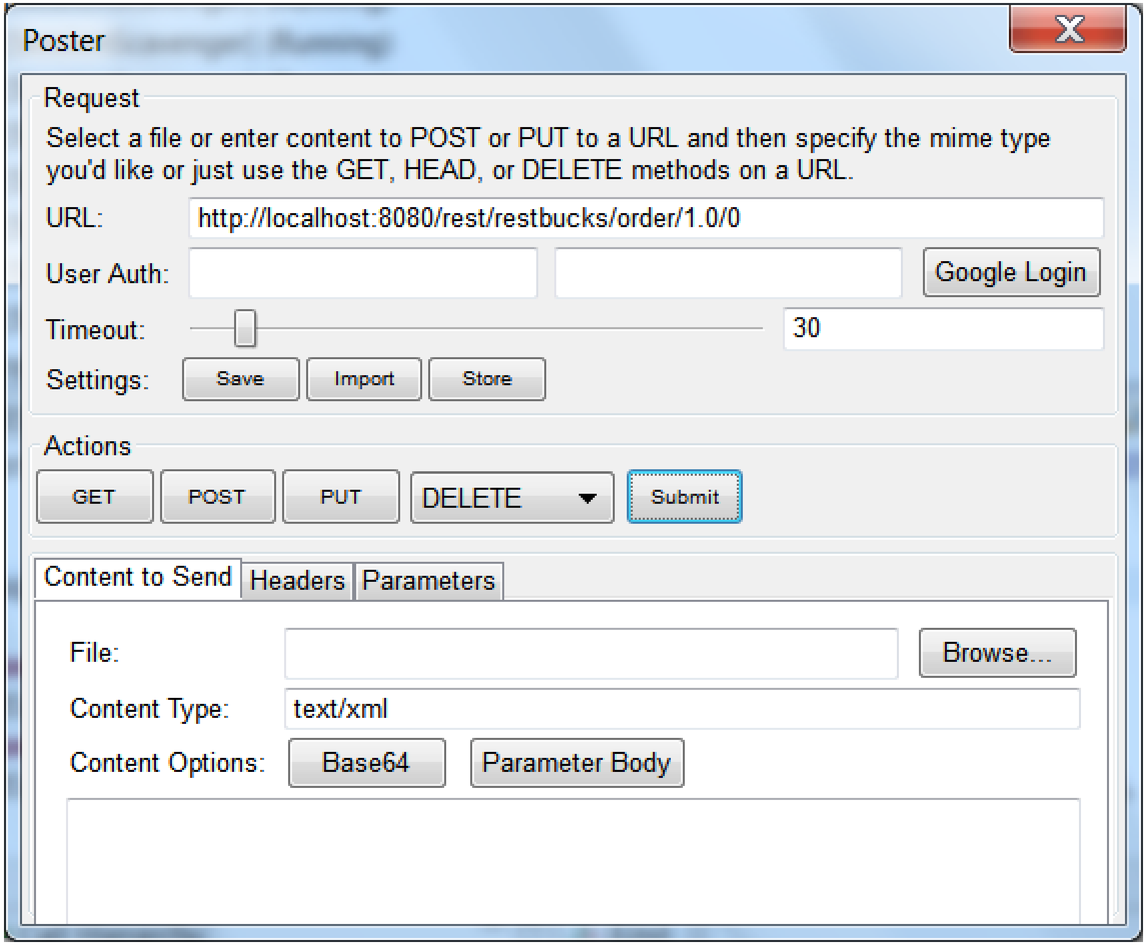

Mozilla Firefox
File Edit View History Delicious Bookmarks Tools Help
TAG
http://localhost:8080/receipt/2fc7f6e2-8b43-4672-a7c4-398e76d640d3
order
getPrice
payment
checkPayment
receipt

This XML file does not appear to have any style information associated with it. The document tree i

```
− <rb:receipt>
    <link rel="order" uri="/rest/restbucks/order/1.0/0"/>
    <link rel="self" uri="/receipt/2fc7f6e2-8b43-4672-a7c4-398e76d640d3"/>
    <rb:amount>12.72</rb:amount>
    <rb:paid>Fri Mar 12 09:49:04 CET 2010</rb:paid>
  </rb:receipt>
```

Interacting with a resource

DELETE /rest/restbucks/order/1.0/0



Poster

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://localhost:8080/rest/restbucks/order/1.0/0

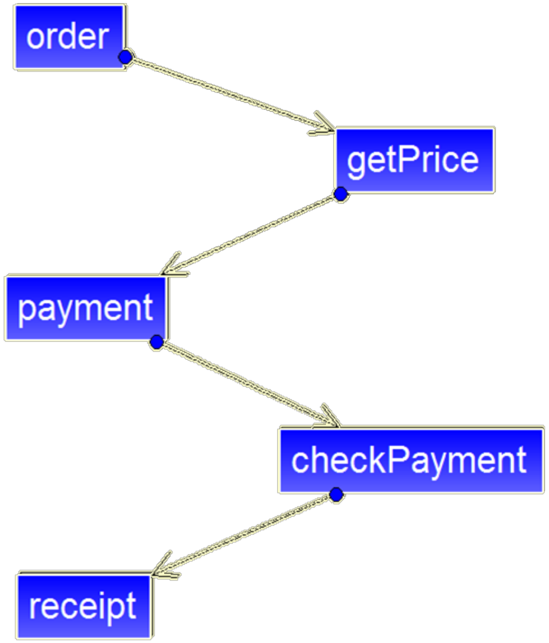
User Auth:

Timeout: 30

Actions

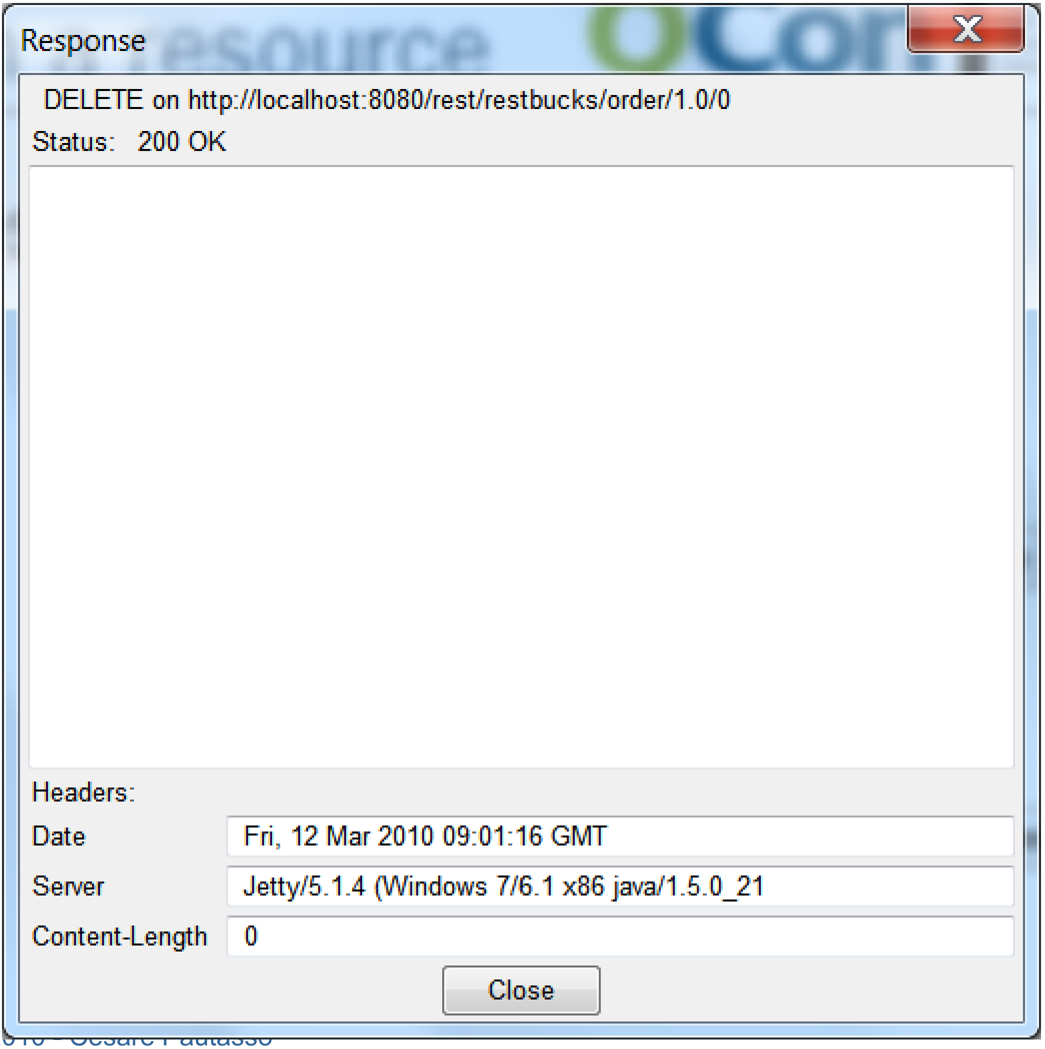
DELETE

Content Type: text/xml



Deleting a process resource

DELETE /rest/restbucks/order/1.0/0

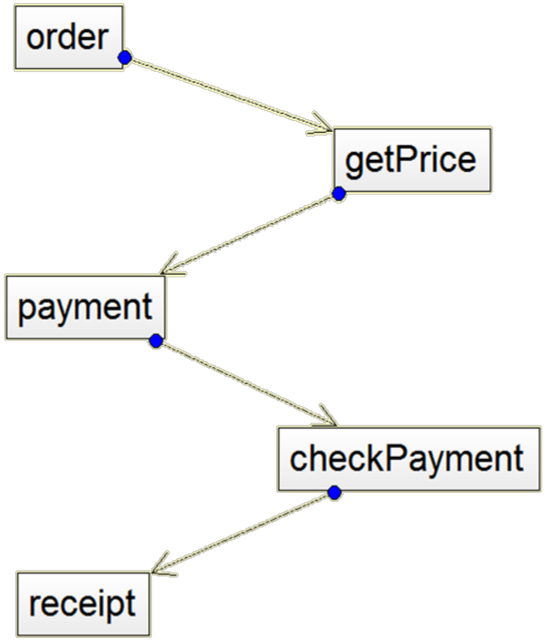


DELETE on http://localhost:8080/rest/restbucks/order/1.0/0

Status: 200 OK

Headers:

Date	Fri, 12 Mar 2010 09:01:16 GMT
Server	Jetty/5.1.4 (Windows 7/6.1 x86 java/1.5.0_21
Content-Length	0



Service Bindings

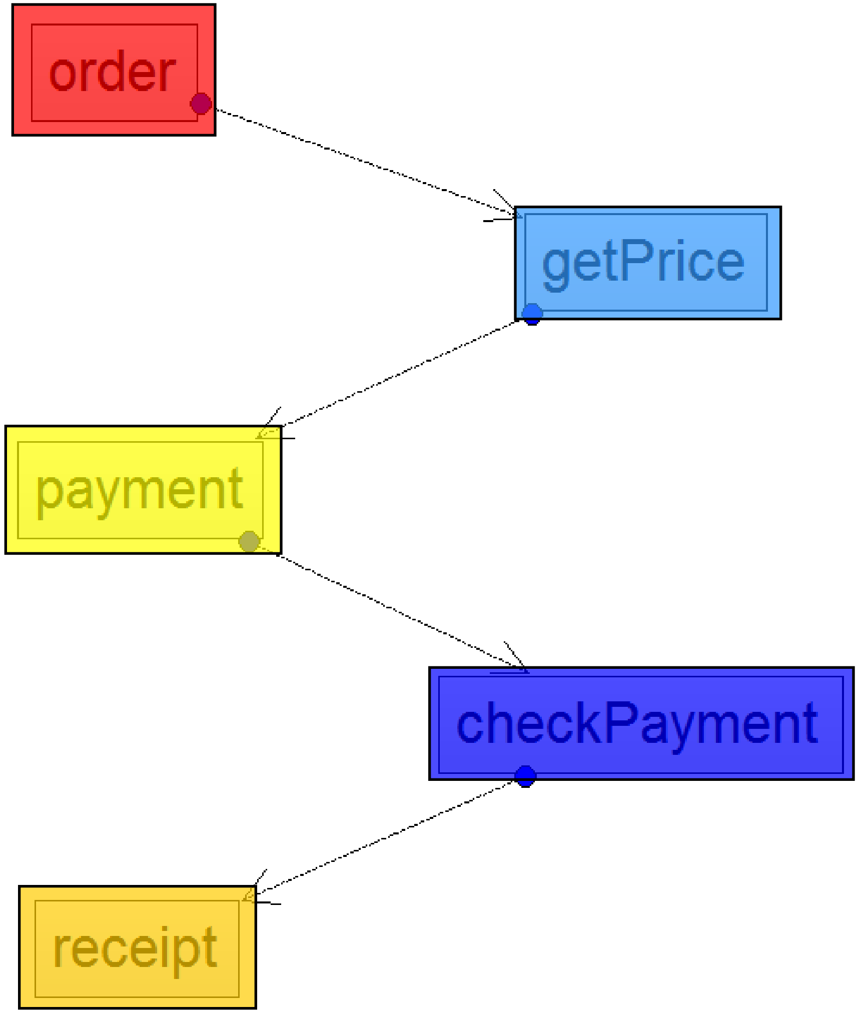
SQL

WS-*

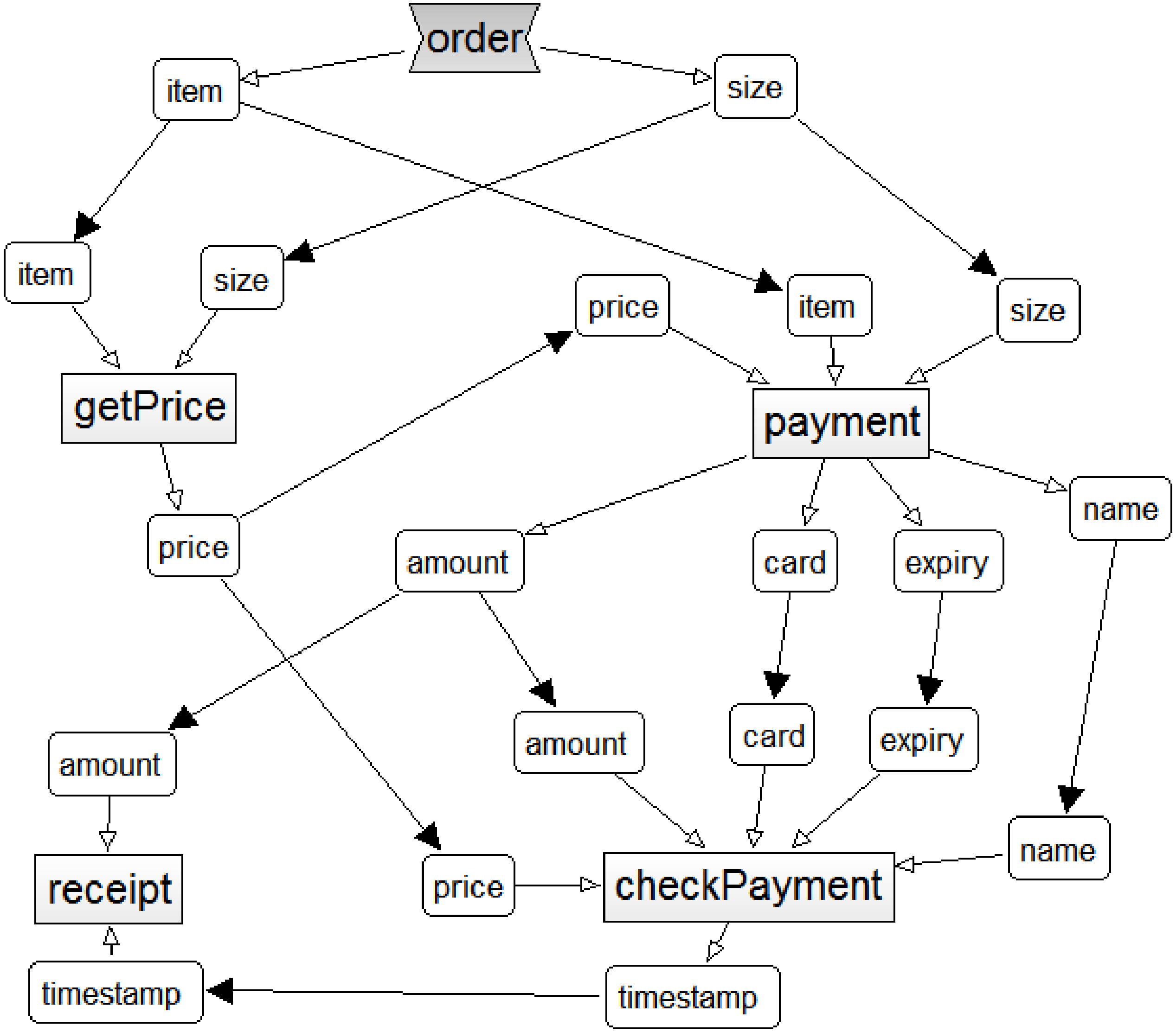
REST

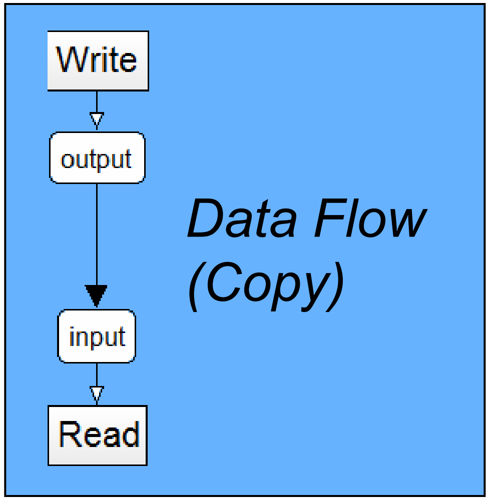
REST.URI

REST.TASK



Data Flow





Data Flow (Copy)

Conclusions

- RESTful HTTP is good enough to interact **without any extension** with process execution engines and their processes and tasks published as resources
- RESTful Web service composition is different than mashups, but both can be built using BPM
- If done right, BPM can be a great modeling tool for Hypermedia-centric service design **(and implementation!)**
- GET http://www.jopera.org/

References

- 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 Robinson, **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
- Raj Balasubramanians, Benjamin Carlyle,Thomas Erl, Cesare Pautasso, **SOA with REST**, Prentice Hall, end of 2010
- Martin Fowler, **Richardson Maturity Model: steps toward the glory of REST,**

http://martinfowler.com/articles/richardsonMaturityModel.html

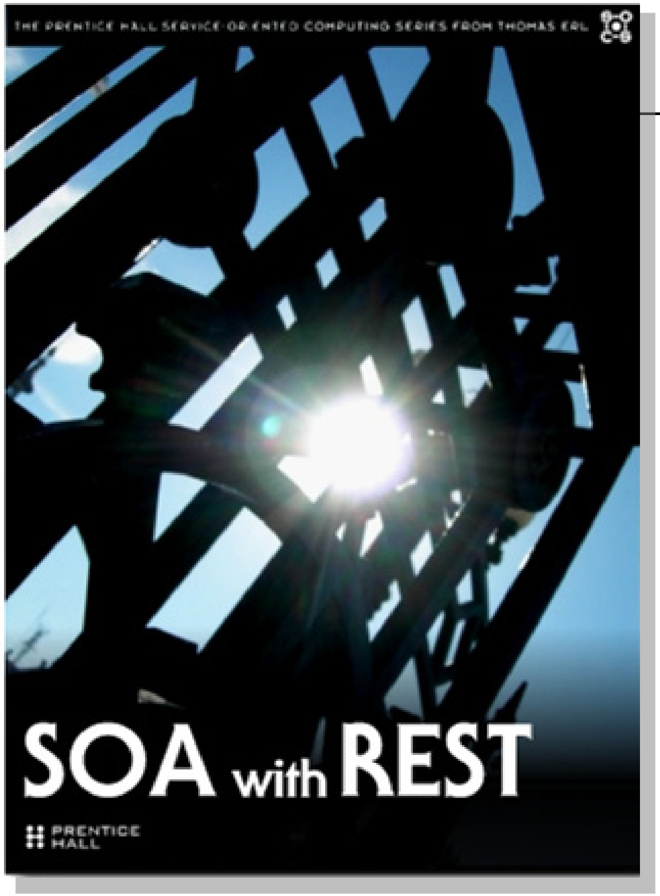
Self-References

- 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), Bejing, 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.

Web Services for the Real World
RESTful
Web Services
O'REILLY
Leonard Richardson & Sam Ruby

Leonard Richardson,
Sam Ruby,

RESTful Web Services,
O'Reilly, May 2007


THE PRENTICE HALL SERVICE-ORIENTED COMPUTING SERIES FROM THOMAS ERL
SOA with REST
PRENTICE HALL

Raj Balasubramanians, Benjamin
Carlyle,Thomas Erl, Cesare Pautasso,

SOA with REST,

Prentice Hall, end of 2010

ECOWS 2010

8th European Conference on Web Services

Cyprus

http://www.cs.ucy.ac.cy/ecows10

http://www.twitter.com/ecows2010