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
Abstract

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.
About Cesare Pautasso

- 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)
- Representations:
  - http://www.pautasso.info/ (Web)
  - http://twitter.com/pautasso/ (Twitter Feed)
WS-* Standards Stack
RESTful Services Standards
Is REST really used?
Is REST really used?

- REST, 71%
- SOAP, 17%
- JSON-RPC, 0%
- XML-RPC, 2%
- RSS, 1%
- SMS, 0%
- XMPP, 0%
- Atom, 2%
- Gdata, 1%
- JavaScript, 6%

2042 APIs
ProgrammableWeb.com
30.6.2010
Web Sites (1992)

Web Browser  HTML  Web Server

WS-* Web Services (2000)

Client  SOAP  WSDL
XML  (HTTP) Server
RESTful Web Services (2007)

WS-* Web Services (2000)
Outline

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 fix 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:


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

- 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/maps?f=q&hl=en&q=lugano,+switzerland&layer=&ie=UTF8&z=12&om=1&iwloc=addr

http://maps.google.com/maps?f=q&hl=en&q=lugano

http://maps.google.com/maps?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 suffixes 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
# Uniform Interface Constraint

<table>
<thead>
<tr>
<th>HTTP</th>
<th>SAFE</th>
<th>IDEM POTENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>GET</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>PUT</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>DELETE</td>
<td>NO</td>
<td>YES</td>
</tr>
</tbody>
</table>

- **POST**: Create a sub resource
- **GET**: Retrieve the current state of the resource
- **PUT**: Initialize or update the state of a resource at the given URI
- **DELETE**: Clear a resource, after the URI is no longer valid
POST vs. GET

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

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

User Agent  Proxy  Gateway

Connector (HTTP)  Cache
Basic Setup

User Agent Origin Server

HTTP

Adding Caching

Caching User Agent

Caching Origin Server

Origin Server

User Agent

Caching Origin Server

Caching User Agent
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
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

<table>
<thead>
<tr>
<th>Resource</th>
<th>GET</th>
<th>PUT</th>
<th>POST</th>
<th>DELETE</th>
</tr>
</thead>
<tbody>
<tr>
<td>/loan</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>/balance</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>/client</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>/book</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>/order</td>
<td>✓</td>
<td>?</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>/soap</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
</tr>
</tbody>
</table>
### Design Space

#### M Representations (Variable)

#### N Resources (Variable)

<table>
<thead>
<tr>
<th>Resource</th>
<th>GET</th>
<th>PUT</th>
<th>POST</th>
<th>DELETE</th>
</tr>
</thead>
<tbody>
<tr>
<td>/loan</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>/balance</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>/client</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>/book</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>/order</td>
<td>✔</td>
<td>?</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>/soap</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
</tr>
</tbody>
</table>

*4 Methods (Fixed)*
1. Resources: **polls and votes**

2. Containment Relationship:

   - GET
   - PUT
   - POST
   - DELETE

   ![Diagram](image)

   - /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)

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

   201 Created
   Location: /poll/090331x

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

   GET /poll/090331x

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

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

200 OK
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
The resource acts as a communication medium that allows services to exchange representations of their state.

This is not equivalent to sending and receiving messages from a bus.
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
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</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/>
<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: 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>
Antipatterns - REST vs. HTTP

REST

HTTP

RESTful HTTP

“RPC”
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?
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>
```
Outline

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

- 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.

<table>
<thead>
<tr>
<th>Architectural Decision: Programming Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture Alternatives:</td>
</tr>
<tr>
<td>1. Java</td>
</tr>
<tr>
<td>2. C#</td>
</tr>
<tr>
<td>3. C++</td>
</tr>
<tr>
<td>4. C</td>
</tr>
<tr>
<td>5. Eiffel</td>
</tr>
<tr>
<td>6. Ruby</td>
</tr>
<tr>
<td>7. ...</td>
</tr>
</tbody>
</table>

Rationale
### Decision Space Overview

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

<table>
<thead>
<tr>
<th>Architectural Decision and AAs</th>
<th>REST</th>
<th>WS-*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transport Protocol</td>
<td>1 AA</td>
<td>≥7 AA</td>
</tr>
<tr>
<td>HTTP</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>waka [13]</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>TCP</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SMTP</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>JMS</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>MQ</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>BEEP</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>IIOP</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Payload Format</td>
<td>≥6 AA</td>
<td>1 AA</td>
</tr>
<tr>
<td>XML (SOAP)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>XML (POX)</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>XML (RSS)</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>JSON [10]</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>YAML</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>MIME</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Service Identification</td>
<td>1 AA</td>
<td>2 AA</td>
</tr>
<tr>
<td>URI</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>WS-Addressing</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Service Description</td>
<td>3 AA</td>
<td>2 AA</td>
</tr>
<tr>
<td>Textual Documentation</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>XML Schema</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>WSDL</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>WADL [18]</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Reliability</td>
<td>1 AA</td>
<td>4 AA</td>
</tr>
<tr>
<td>HTTPR [38]</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>WS-Reliability</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>WS-ReliableMessaging</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Native</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Do-it-yourself</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Security</td>
<td>1 AA</td>
<td>2 AA</td>
</tr>
<tr>
<td>HTTPS</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>WS-Security</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Transactions</th>
<th>REST</th>
<th>WS-*</th>
</tr>
</thead>
<tbody>
<tr>
<td>WS-AT, WS-BA</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>WS-CAF</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Do-it-yourself</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Service Composition</td>
<td>2 AA</td>
<td>2 AA</td>
</tr>
<tr>
<td>WS-BPEL</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Mashups</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Do-it-yourself</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Service Discovery</td>
<td>1 AA</td>
<td>2 AA</td>
</tr>
<tr>
<td>UDDI</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Do-it-yourself</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Implementation Technology</td>
<td>many</td>
<td>many</td>
</tr>
</tbody>
</table>

| Total Number of Decisions, AAs | 10, ≥17 | 10, ≥25 |

*Optional

---

**Table 2: Conceptual Comparison Summary**

**Table 1: Principles Comparison Summary**

**Table 3: Technology 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-*
Comparison Overview

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

HTTP Client (Web Browser)

Web Server
Application Server

Database


SELECT *
FROM books
WHERE isbn=222

POST /order

INSERT
INTO orders

301 Location: /order/612

PUT /order/612

UPDATE orders
WHERE id=612

©2009-2010 - Cesare Pautasso - 30.6.2010
WS-* Service Example
(from REST perspective)

HTTP Client
(Stub Object)

Web Server
Application Server

Web Service
Implementation

POST /soap/endpoint

return getBook(222)

return new Order()

order.setCustomer(x)
“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”
Coupling Facets

<table>
<thead>
<tr>
<th>Facets</th>
<th>REST</th>
<th>WS-*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discovery</td>
<td>Referral</td>
<td>Centralized</td>
</tr>
<tr>
<td>Identification</td>
<td>Global</td>
<td>Context-Based</td>
</tr>
<tr>
<td>Binding</td>
<td>Late</td>
<td>Late</td>
</tr>
<tr>
<td>Platform</td>
<td>Independent</td>
<td>Independent</td>
</tr>
<tr>
<td>Interaction</td>
<td>Asynchronous</td>
<td>Asynchronous</td>
</tr>
<tr>
<td>Model</td>
<td>Self-Describing</td>
<td>Shared Model</td>
</tr>
<tr>
<td>State</td>
<td>Stateless</td>
<td>Stateless</td>
</tr>
<tr>
<td>Generated Code</td>
<td>None/Dynamic</td>
<td>Static</td>
</tr>
<tr>
<td>Conversation</td>
<td>Reflective</td>
<td>Explicit</td>
</tr>
</tbody>
</table>

Coupling Comparison

- RESTful HTTP
- RPC over HTTP
- WS-*/ESB
Dealing with Heterogeneity

- Enable Cooperation
- Web Applications

- Enable Integration
- Enterprise Architectures

Picture from Eric Newcomer, IONA
Heterogeneity

Web Applications

Enterprise Architectures

REST

WS-*
Claim: REST can also be successfully used to design integrated enterprise applications
Enterprise “Use Cases”

Enterprise Architectures

- CRUD Services
- Web-friendly APIs
- Mobile Services
- Real-time Services
- Transactional Services
- Composite Services

©2009-2010 - Cesare Pautasso - 30.6.2010
Part of the debate is about how many “enterprise” use cases can be covered with REST as opposed to WS-*
What about...

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

©2009-2010 - Cesare Pautasso - 30.6.2010
What about service description?

- 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 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
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 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

- 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!
What about stateful services?

- 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?
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.
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
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

$C$

$\text{Link}_R$

$\text{Link}_S$

DELETE

PUT

GET

POST

R

DELETE

PUT

GET

POST

S
A composite representation is interpreted by the client that follows its hyperlinks and aggregates the state of the referenced component resources.
A composite representation can be produced by a composite service too.
- Vote on a meeting place based on its geographic location
Composite Resource
Composite Resource
Demo

DoodleMap with JOpera

Poll: hamburger

CP has created this poll.

"10001"
Doodle Map Architecture

Web Browser

Workflow Engine

RESTful Web Services APIs

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

Control Flow

Data Flow

Service Bindings

JAVA  XPATH  XSLT  HTML  HTTP  ...

©2010 - Cesare Pautasso
Control Flow

- GetYahooLocal
- ConvertY2D
- ConvertY2G
- PostDoodlePoll
- ParsePollID
- ShowGoogleMap
- Wait
- GetDoodlePoll
- CountParticipants
- ClosePoll
- PutDoodlePoll

Control Flow Dependency

Task1

Task2
Data Flow

Data Flow (Copy)
Was it just a mashup?

(Mashup)

(REST)

(Composition)

(It depends on the definition of Mashup)
Moving state around

- Read-only vs. Read/Write
Simply aggregating data

- **Read-only** vs. Read/write
Is your composition reusable?

- **UI vs. API Composition**

  ![Diagram showing UI vs. API Composition]

- **Reusable services vs. Reusable Widgets**

  ![Diagram showing the comparison between UI and API compositions]
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.
Complementary

Read-Only

UI

Situational Sandboxed

Mashup

Composition

REST

Read/Write

API

Reusable Service
Towards REST Composition

- 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/
Software Connectors

- File Transfer
- Procedure Call
- Remote Procedure Call
- Shared Data
- Message Bus
- Events
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
Hot Folder

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
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 handle concurrency.
• 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
Different software connectors

- RPC
- ESB
- WS-*
- WWW
- REST
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.
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.
References

- 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
Web References

- 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
Self-References


- 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
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