Service Oriented Architectures for the REST of Us

Cesare Pautasso
Faculty of Informatics
University of Lugano, Switzerland

c.pautasso@ieee.org
http://www.pautasso.info
@pautasso
About Cesare Pautasso

• Assistant Professor at the Faculty of Informatics, University of Lugano, Switzerland
• Research Projects:
  • SOSOA – Self-Organizing Service Oriented Architectures
  • CLAVOS – Continuous Lifelong Analysis and Verification of Open Services
  • BPEL for REST, RESTful Atomic Transactions with TCC
  • SAW - Software Architecture Warehouse
• Researcher at IBM Zurich Research Lab (2007)
• Post-Doc at ETH Zürich
• Software: http://www.jopera.org/
  JOpera: Process Support for more than Web services
• Ph.D. at ETH Zürich, Switzerland (2004)
• Laurea Politecnico di Milano (2000)
• Representations:
  http://www.pautasso.info/ (Web)
  http://twitter.com/pautasso/ (Twitter Feed)
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 Internet works are just as applicable to solving enterprise application integration problems and to radically simplifying the "plumbing" of service-oriented architectures. In this tutorial, we give an update on how the REST architectural style has been recently rediscovered to become the foundation for so-called "RESTful Web services." After introducing REST as a set of design constraints, we will show how these can be used as a guide to the design of RESTful Web service APIs while still delivering most of the qualities required in enterprise-grade service-oriented architectures.
Outline

- Introduction
- What is REST?
- RESTful Service Design
- RESTful Web Services within the Enterprise
Outline

- Introduction
  - SOA without WS-*
  - Architectural Styles for SOA
  - REST and enterprise application integration styles
  - The Web as a software connector
  - Is REST being used?
- What is REST?
- RESTful Service Design
- RESTful Web Services within the Enterprise
SOA

REST
WS-* Standards Stack
WS-* Standards Stack
Can you do it with REST?
RESTful Services Standards

AtomPub
RSS
Atom
XML
JSON
URI
HTTP
MIME
SSL/TLS
What is REST?

1. An architectural style for building loosely coupled distributed hypermedia systems
   - The Web is an architecture which follows the REST style

When the size and complexity of a system grows, we cannot describe its whole architecture, but we can still know its style.
Service Oriented Architectural Style?

From R. Peisl, 2006
SOA Quality Attributes

- Interoperability
- Modifiability
- Integrability
- Reuse
- Portability
- Reliability
- Security
- Efficiency
- Business Flexibility/Agility
- Return on Investment
What is REST?

1. An *architectural style* for building loosely coupled distributed hypermedia systems
   - The Web is an architecture which follows the REST style

2. *The Web used correctly* to publish Web services
   - Follow Web standards (URI/HTTP/XML/JSON)
   - HTTP = application protocol

3. *The Web misused* to publish Web services
   - (As long as you do not use SOAP)
   - For example: Plain-Old-XML (POX) over HTTP
   - HTTP = transport/tunneling protocol
Web Sites (1992)

Web Browser | HTML | Web Server
-------------|------|-------------
       | HTTP |             

WS-* Web Services (2000)

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

WS-* Web Services (2000)
RESTful Web Service Example

HTTP Client (Web Browser)


POST /order

301 Location: /order/612

PUT /order/612

Web Server

Application Server

Database

SELECT *
FROM books
WHERE isbn=222

INSERT INTO orders

UPDATE orders
WHERE id=612

©2011 - Cesare Pautasso
WS-* Service Example (from REST perspective)

HTTP Client (Stub Object)

POST /soap/endpoint

return getBook(222)

POST /soap/endpoint

return new Order()

POST /soap/endpoint

order.setCustomer(x)

Web Server Application Server

Web Service Implementation

©2011 - Cesare Pautasso
“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”
WS Technology Design Space

Representations

- Many Message Formats (XML, JSON, ATOM, HTML, CSV, ...)
- 1 Message Format (SOAP)

1 Communication “Endpoint”

- Many URLs
- 4 HTTP Verbs (GET, PUT, POST, DELETE)
- Many Operations (WSDL)

Resources

WS-*

REST

©2011 - Cesare Pautasso
Software Connectors

- File Transfer
- Procedure Call
- Remote Procedure Call
- Shared Data
- Message Bus
- Events
What is your SOA connector today?

RPC

BUS

REST/HTTP
REST as a new connector

RPC

BUS

Publish/Subscribe

REST/HTTP

Get/Put/Post/Delete
Is REST really used?

3120 APIs
ProgrammableWeb.com
1.4.2011

REST, 71%

SOAP, 17%

XML-RPC, 2%

JSON-RPC, 0%

RSS, 1%

SMS, 0%

XMPP, 0%

GData, 1%

JavaScript, 6%

Atom, 2%
Outline

- Introduction
- What is REST?
  - REST as a hybrid architectural style
  - Resources and URI
  - Uniform Interface
  - Resource Representations: Beyond XML
  - Hypermedia as the engine of application state
  - Intermediaries
- RESTful Service Design
- RESTful Web Services within the Enterprise
REST Constraints in one slide

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

- 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 sent to the server
# 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
## Uniform Interface Constraint

<table>
<thead>
<tr>
<th>CRUD</th>
<th>REST</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE</td>
<td>POST</td>
<td>Create a sub resource</td>
</tr>
<tr>
<td>READ</td>
<td>GET</td>
<td>Retrieve the current state of the resource</td>
</tr>
<tr>
<td>UPDATE</td>
<td>PUT</td>
<td>Initialize or update the state of a resource at the given URI</td>
</tr>
<tr>
<td>DELETE</td>
<td>DELETE</td>
<td>Clear a resource, after the URI is no longer valid</td>
</tr>
</tbody>
</table>

©2011 - Cesare Pautasso
HTML5 Forms

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

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

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.

  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
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: 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)

\[ \text{GET /resource} \]

\[ \text{Accept: text/html, application/xml, application/json} \]

1. The client lists the set of understood formats (MIME types)

\[ \text{200 OK} \]

\[ \text{Content-Type: application/json} \]

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

\[ \text{GET /resource} \]

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

The specific URI points to a specific representation format using the postfix (extension)

\[ \text{GET /resource.html} \]
\[ \text{GET /resource.xml} \]
\[ \text{GET /resource.json} \]

**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).

<table>
<thead>
<tr>
<th>Request Header</th>
<th>Example Values</th>
<th>Response Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accept:</td>
<td>application/xml, application/json</td>
<td>Content-Type:</td>
</tr>
<tr>
<td>Accept-Language:</td>
<td>en, fr, de, es</td>
<td>Content-Language:</td>
</tr>
<tr>
<td>Accept-Charset:</td>
<td>iso-8859-5, unicode-1-1</td>
<td>Charset parameter fo the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>header</td>
</tr>
<tr>
<td>Accept-Encoding:</td>
<td>compress, gzip</td>
<td>Content-Encod ing:</td>
</tr>
</tbody>
</table>
Hypermedia

- **Problem:**
  - How to discover the URIs of a potentially infinite and dynamically changing set of resources?

- **Solution:**
  - Resource representations contain links that identify other resources
Like service discovery:
- Clients can use a service to dynamically lookup and discover other services

Unlike service discovery:
- Any resource can refer clients to any other resource (decentralized)
- Links can be embedded in any hypermedia representation format
Hypermedia (HATEOAS)

- In addition to resource discovery, hypermedia is used for: “Hypermedia as the engine of application state”

- Relationships between resources can be dynamically discovered by clients following links
  - No need to hardcode resource identifiers into clients
  - No need to statically specify resource relationships

- Links also describe valid protocol state transitions
  - The service guides the client in discovering what are the next possible valid interactions

- Clients still need to know the semantics of link traversal to choose what to do
  - *Am I going to be charged $$ if I follow this link?*
REST Architectural Elements

Client/Server  Layered  Stateless Communication  Cache

User Agent  Proxy  Gateway  Origin Server

Connector (HTTP)  Cache

©2011 - Cesare Pautasso
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
Outline

- Introduction
- What is REST?
- RESTful Service Design
  - Design Methodology
  - Design Patterns
  - Design Examples
  - Dealing with state
  - Richardson’s Maturity Model
- RESTful Web Services within the Enterprise
Design Methodology for REST

1. Identify resources to be exposed as services (e.g., yearly risk report, book catalog, purchase order, open bugs, blog entries, 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 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, document and standardize resource representations (media types)

6. Implement and deploy on Web server

7. Test with a Web browser

<table>
<thead>
<tr>
<th></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)

4 Methods (Fixed)

N Resources (Variable)
Simple Doodle API Example Design

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

<table>
<thead>
<tr>
<th>Resource Path</th>
<th>GET</th>
<th>PUT</th>
<th>POST</th>
<th>DELETE</th>
</tr>
</thead>
<tbody>
<tr>
<td>/poll</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>/poll/{id}</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>/poll/{id}/vote</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>/poll/{id}/vote/{id}</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>?</td>
</tr>
</tbody>
</table>

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>

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

   <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

```
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)

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

200 OK

GET /poll/090331x

200 OK
```

©2011 - Cesare Pautasso
Simple Doodle API Example

- Polls can be deleted once a decision has been made

```
GET /poll/090331x
404 Not Found
```

```
DELETE /poll/090331x
200 OK
```

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

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  
- Most “RESTful” APIs don’t
HTTP = Tunnel
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 entity as individual “endpoint” of the service they reside in
- Benefits: Global addressability of service entities
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
URI Design Guidelines

- Prefer Nouns to Verbs
- Keep your URIs short
- If possible follow a “positional” parameter-passing scheme for algorithmic resource query strings (instead of the key=value&p=v encoding)

Some use URI postfixes to specify the content type

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

GET /book?isbn=24&action=delete
DELETE /book/24

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

This may break the abstraction

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

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

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

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

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

- Example URI:
  http://www.myservice.com/order/XYZ/item/12345

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

- Example URI:
  http://www.google.com/search?q=REST&num=10
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.

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/xm
(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 (internally or externally) 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
- **Warning:** Clients are not forced to process the media type as you expect them to
Exception Handling

Learn to use HTTP Standard Status Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Continue</td>
</tr>
<tr>
<td>200</td>
<td>OK</td>
</tr>
<tr>
<td>201</td>
<td>Created</td>
</tr>
<tr>
<td>202</td>
<td>Accepted</td>
</tr>
<tr>
<td>203</td>
<td>Non-Authoritative</td>
</tr>
<tr>
<td>204</td>
<td>No Content</td>
</tr>
<tr>
<td>205</td>
<td>Reset Content</td>
</tr>
<tr>
<td>206</td>
<td>Partial Content</td>
</tr>
<tr>
<td>300</td>
<td>Multiple Choices</td>
</tr>
<tr>
<td>301</td>
<td>Moved Permanently</td>
</tr>
<tr>
<td>302</td>
<td>Found</td>
</tr>
<tr>
<td>303</td>
<td>See Other</td>
</tr>
<tr>
<td>304</td>
<td>Not Modified</td>
</tr>
<tr>
<td>305</td>
<td>Use Proxy</td>
</tr>
<tr>
<td>307</td>
<td>Temporary Redirect</td>
</tr>
<tr>
<td>400</td>
<td>Bad Request</td>
</tr>
<tr>
<td>401</td>
<td>Unauthorized</td>
</tr>
<tr>
<td>402</td>
<td>Payment Required</td>
</tr>
<tr>
<td>403</td>
<td>Forbidden</td>
</tr>
<tr>
<td>404</td>
<td>Not Found</td>
</tr>
<tr>
<td>405</td>
<td>Method Not Allowed</td>
</tr>
<tr>
<td>406</td>
<td>Not Acceptable</td>
</tr>
<tr>
<td>407</td>
<td>Proxy Authentication Required</td>
</tr>
<tr>
<td>408</td>
<td>Request Timeout</td>
</tr>
<tr>
<td>409</td>
<td>Conflict</td>
</tr>
<tr>
<td>410</td>
<td>Gone</td>
</tr>
<tr>
<td>411</td>
<td>Length Required</td>
</tr>
<tr>
<td>412</td>
<td>Precondition Failed</td>
</tr>
<tr>
<td>413</td>
<td>Request Entity Too Large</td>
</tr>
<tr>
<td>414</td>
<td>Request-URI Too Long</td>
</tr>
<tr>
<td>415</td>
<td>Unsupported Media Type</td>
</tr>
<tr>
<td>416</td>
<td>Requested Range Not Satisfiable</td>
</tr>
<tr>
<td>417</td>
<td>Expectation Failed</td>
</tr>
<tr>
<td>500</td>
<td>Internal Server Error</td>
</tr>
<tr>
<td>501</td>
<td>Not Implemented</td>
</tr>
<tr>
<td>502</td>
<td>Bad Gateway</td>
</tr>
<tr>
<td>503</td>
<td>Service Unavailable</td>
</tr>
<tr>
<td>504</td>
<td>Gateway Timeout</td>
</tr>
<tr>
<td>505</td>
<td>HTTP Version Not Supported</td>
</tr>
</tbody>
</table>

4xx Client’s fault

5xx Server’s fault
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 = Get Balance()  // safe
  B = B + 200$        // local
  Set Balance( 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?
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.
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!
Antipatterns - REST vs. HTTP

RESTful HTTP

“RPC”
Antipatterns – HTTP as a tunnel

- Tunnel through one HTTP Method

  ```
  GET /api?method=addCustomer\&name=Pautasso
  GET /api?method=deleteCustomer\&id=42
  GET /api?method=getCustomerName\&id=42
  GET /api?method=findCustomers\&name=Pautasso*
  ```

- 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)

```
POST /service/endpoint

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

- Enable Cooperation
  - Web Applications
- Enable Integration
  - Enterprise Architectures

©2011 - Cesare Pautasso
Heterogeneity

Web Applications

REST

Enterprise Architectures

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

REST

WS-*
Enterprise “Use Cases”

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

- Introduction
- What is REST?
- RESTful Service Design
- RESTful Web Services within the Enterprise
  - What about service description?
  - What about security?
  - What about reliability?
  - Composing RESTful Web Services
  - Mashups vs. Composite RESTful Web Services
  - RESTful Business Process Management
  - Transactional RESTful Web Services
What about service description?

- REST relies on human readable documentation that defines requests URIs and responses (XML, JSON)
- Interacting with the service means hours of testing and debugging URIs manually built as parameter combinations. (Is it really that simpler building URIs by hand?)
- Why do we need strongly typed SOAP messages if both sides already agree on the content?
  - WADL proposed Nov. 2006
  - 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 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
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 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?
Business Process Management

Process Model
- Act 1
- Act 2
- Act 3
- Act 4
- Act 5
- Act 6
- Act 7

Users/Clients

Workflow Participants

RESTful Workflow Management Engine

Adapters
- Applications
- Databases

PUT
- GET
- POST
- DELETE

Bus
Web Services

©2011 - Cesare Pautasso
BPM with REST

RESTful Workflow Management Engine

Publishing Processes as Resources

RESTful Service Composition

©2011 - Cesare Pautasso
WSDL 2.0 HTTP Binding can wrap RESTful Web Services

(WS-BPEL 2.0 does not support WSDL 2.0)
BPEL for REST

Make REST interaction primitives first-class language constructs of BPEL
BPEL for REST – Resource Block

- Dynamically publish resources from BPEL processes and handle client requests

BPEL for REST

```xml
<Resource P>
  <onGet>
    <Put R>
    <Get S>
    </onGet>
  <onDelete>
    <Post R>
    <Delete S>
    </onDelete>
  </Resource>
```
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
Composite Resources

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

Composite Representation

C

\[ \text{Link}_R \]

\[ \text{Link}_S \]

R

DELETE

PUT

GET

POST

S

DELETE

PUT

GET

POST
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
1. Composite Resource
1. Composite Resource
2. Composite Representation
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

Web Browser

Workflow Engine

RESTful Web Services APIs

Composite Representations

Composite Resources

Resources

©2011 - Cesare Pautasso
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?
Security Policies on the client may not always allow it to aggregate data from multiple different sources.
BPM with REST

Process Model

Act 1
Act 2
Act 3
Act 4
Act 5
Act 6
Act 7

RESTful Workflow Management Engine

Publishing Processes as Resources

RESTful Service Composition

©2011 - Cesare Pautasso
BPM

- Processes
- Tasks
- Control Flow
- Data Flow
- ...

REST

- Resources/URIs
- Uniform Interface
- Representations
- Hypermedia

©2011 - Cesare Pautasso
Everything is a resource

/process/X

Process

/Task

Process Instance

/task/Y

/task/Y/1

Task Instance

©2011 - Cesare Pautasso
Hypermedia

GET /process

GET /process/name

GET /process/name/instance

GET /process/name/instance/taskname

Follow links to discover the processes deployed as resources
Representations

- Web page with form to start a new process instance
- List of process input parameters
- Content Type: text/html
- Content Type: application/xml
- GET /process/name
- Content Type: text/plain
- Content Type: application/json
- Basic textual description of the process
- Image of the process metadata in JSON

©2011 - Cesare Pautasso
Uniform Interface and Hypermedia

- GET /process: List the deployed processes
- GET /process/name: Get a form describing how to start the process
- POST /process/name: Start a new process instance
- GET /process/name/instance: Check what is the state of the instance
- DELETE /process/name/instance: Clean up (once it is done)
Starting or Running a new process?

- Should the client be kept waiting for the process to run until completion?

- Clients may want to block until the whole process has completed its execution (or it decides to reply to them)
Starting or Running a new process?

- The client starting a long running process is redirected to a location $x$ representing the newly started process instance.
- The process and the client run asynchronously.
- The client may retrieve the current state of the process instance at any time.

POST /process

202 Accepted
Location: $x$

GET /process/$x$

200 OK
Uniform Interface and Hypermedia

- **GET /task**
  - List the active tasks

- **GET /task/name/instance**
  - Get a form describing how to perform the task

- **POST /task/name/instance**
  - Finish the task

- **GET /task/name/instance**
  - Get the final representation of the completed task

...
POST or PUT?

GET /task

GET /task/name/instance

POST /task/name/instance

GET /task/name/instance

...
RESTbucks
Example
taken from

REST in Practice

Jim Webber, Savas Parastatidis, Ian Robinson
Simple RESTBucks Example

/rest/restbucks/order/1.0/
/rest/restbucks/order/1.0/{id}
/tasks/restbucks/order/1.0/{id}/payment
/receipt/{uuid}
Simple RESTBucks Example

Hypermedia Centric Service Design done with a business process model
Instantiating a process

GET /rest/restbucks/order/1.0/

Retrieve a form which describes how to instantiate a new process
Instantiating a process

POST /rest/restbucks/order/1.0/

Start = non blocking
(redirect to URI of the new instance)

Run = blocking
(client waits until the process replies)
Interacting with a task

GET

/task/restbucks/order/1.0/0/payment

Task restbucks.order [1.0].payment.0

State: Waiting

Input Parameters

- item: Latte
- instance: 1
- price: 19.0
- size: XXL
- id: a7b963b5-1bca-46b8-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

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

```xml
  <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...

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

```
<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>
```
DELETE /rest/restbucks/order/1.0/0
Deleting a process resource

DELETE /rest/restbucks/order/1.0/0
Static vs. Dynamic Typing

- **Myth:** RESTful Web services cannot be composed (with BPEL) because they do not give a static contract description.

- **Reality:** RESTful Web services can dynamically negotiate the most suitable representation format with their clients.

- **Challenge:** How to support dynamic typing and content type negotiation in a BPM composition language?
Active Resources

- **Myth:** Processes cannot be mapped to resources because they can change their state (independently of their clients)

- **Reality:** REST Resources do not have to be passive “CRUD” services but can be active and have a life of their own.

- **Challenge:** How to best let clients control an active resource backed up by a process instance through the uniform interface?
Synchronous vs. Asynchronous

- **Myth**: Processes run for a long time and need to interact asynchronously with their clients. This cannot be done with HTTP.
- **Reality**: HTTP supports non-blocking interactions. Each process instance is mapped to a resource URI, which can be used by clients throughout its lifetime.
- **Challenge**: How to let processes send notifications back to their clients?
BPM with REST

- REST resources are a good abstraction to publish processes on the Web.
- RESTful HTTP is good enough to interact without any extension with process execution engines and drive the execution of process and task instances.
- If done right, BPM can be a great modeling tool for Hypermedia-centric service design (and implementation!)

©2011 - Cesare Pautasso
REST Transactions: The Problem

Adapted from Stefan Tilkov, Using REST for SOA, QCon SFO 2010
Does REST need transactions?

- If you find yourself in need of a distributed transaction protocol, then how can you possibly say that your architecture is based on REST? I simply cannot see how you can get from one situation (of using RESTful application state on the client and hypermedia to determine all state transitions) to the next situation of needing distributed agreement of transaction semantics wherein the client has to tell the server how to manage its own resources.

- ...for now I consider "rest transaction" to be an oxymoron.

Roy Fielding, REST discuss, June 9th, 2009
Does REST need transactions?

- The typical conversation thread, real or virtual, about transactions over HTTP goes something like this (elided for brevity):
  - "You don't want transactions over HTTP"
  - But I need to organize number of steps into a single unit I can deal with easily.
  - "OK, but you don't need transactions over HTTP"
  - But I need the ability to back out changes in multiple locations safely and consistently.
  - "OK, but you can't do transactions over HTTP!"
  - Really?
- And here the topic usually dies or descends into a heated debate.

Mike Amundsen,
http://amundsen.com/blog/archives/1024
Adapted from Stefan Tilkov, Using REST for SOA, QCon SFO 2010
The problem

Thanks to the idempotency of GET/PUT, each individual state transfer is reliable and atomic.
The problem

How to we make both interactions atomic?
Constraints

- Interoperability:
  - No changes/extensions to HTTP
    - No additional verbs
    - No special/custom headers

- Loose Coupling:
  - REST shifts all the “work” to the client
  - RESTful Web services *should remain unaware* they are participating in a transaction

- Simplicity:
  - Transactions will not be adopted in practice unless they can be made simple enough
Assumption: Try-Confirm/Cancel

- Resource state transitions follow the TCC pattern

- Before they are made permanent state transitions go through an intermediate "reserved" state which either will be confirmed or canceled by a client within a given time

- Hint: Cancel/Confirm are idempotent
Example: Flight Booking Resource

Try
- POST /booking
- 302 Location: /booking/X

Confirm
- PUT /booking/X
- 200

Cancel
- DELETE /booking/X
- 200

Reserve the flight
URI of the reserved state
Pay and confirm the flight
Cancel the reservation
Protocol

1. A client interacts with multiple RESTful Web services. Interactions may lead to state transitions (the intermediate state is identified by a URI known to the client)

2. Once the client has completed all interactions, it uses the URIs identifying the intermediate states to confirm the state transitions (and thus commit the transaction)

Note: If the client stops before step 2, the state transitions will eventually be undone by the services themselves (after a timeout). As an optimization, the client can use the same URIs to cancel the state transitions (and thus explicitly rollback the transaction).
Simple Example

1. POST www.swiss.ch/booking
   302 Location: /booking/1

   POST www.ezyj.com/booking
   302 Location:/booking/X

2. PUT www.swiss.ch/booking/1
   200

   PUT www.ezyj.com/booking/X
   200
What if something fails?

1. POST www.swiss.ch/booking
   302 Location:/booking/1
   Whatever happens, these state transitions are temporary.
   If something fails, stop before moving to phase 2

2. PUT www.swiss.ch/booking/1
   200
   Only idempotent methods are allowed in the confirmation phase.
   If something fails, retry as many times as necessary
A matter of timing

Time

Agreement is reached if the confirmation phase ends before the resources undo the state transitions because of the timeouts.
A matter of timing

If the confirmation runs longer than the earliest timeout we cannot guarantee agreement.
Timeouts and heuristics

- **Bad News:** As with every distributed agreement protocol, it is impossible to avoid heuristics.

- **Good News:**
  - thanks to the REST uniform interface we can always do a GET on the URI of the reserved resource to see how much time we have left before it cancels.
  - Avoid starting phase 2 if there is not enough time left to confirm with every service.

- More complex preparation: if the resource allows it, extend the reservation time (also idempotent) before starting phase 2.

- In any case, use a lightweight transaction coordinator to log everything for recovery and human diagnosis of heuristics.
Architecture (Client-side Transaction)
Architecture (Service Composition)

Client → Workflow Engine → Transaction Coordinator

1. Try
2. Confirm
3. Try
4. Confirm
5. Confirm
6. Try

Composite RESTful Service → TCC Resources

©2011 - Cesare Pautasso
Atomic Transactions for REST

- The protocol guarantees atomicity in the event of failures among multiple interactions with RESTful Web services that comply with the TCC pattern.
- (We are not interested in isolation)
- No HTTP extension is required
- Fits very nicely with the REST uniform interface and the idempotency of the PUT/DELETE methods
- Hypermedia can be easily built in to guide the discovery of the cancellation/confirmation URIs (e.g., with HTTP Link Headers)
Conclusions

- SOA comes from the business IT domain, while REST comes from the World Wide Web.
- REST is more at home with HTTP and HTML, while SOA is more at home with SOAP and WSDL.
- Some REST advocates see the Web Services stack both as begin synonymous with SOA and as an invader in the architecture of the "real" Web. Some SOA advocates see REST as an unnecessary diversion from ensuring connectivity between enterprise service bus technologies supplied by different vendors.
- Despite their different histories, REST and SOA can learn a lot from each other.
- SOA with REST aims to forge an effective architectural model both for enterprise computing and for computing on the World Wide Web that brings the best of both worlds together.
References

Service Oriented Architectures for the REST of Us

Cesare Pautasso
Faculty of Informatics
University of Lugano, Switzerland

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