SOA with REST

Cesare Pautasso
Faculty of Informatics
University of Lugano, Switzerland

c.pautasso@ieee.org
http://www.pautasso.info
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. In this talk we take a close look at the potential for convergence of service orientation and the REST architectural style. We highlight the benefits in terms of simplicity, loose coupling, and performance of a RESTful approach to SOA and discuss the most important SOA design patterns that become available once REST is introduced.
Acknowledgements

- The following distinguished individuals have contributed to the patterns, ideas and reviewed some of the material presented in this talk:
  - Raj Balasubramanian
  - Benjamin Carlyle
  - Thomas Erl
  - Stefan Tilkov
  - Erik Wilde
  - Herbjorn Wilhelmsen
  - Jim Webber
SOA with REST - Outline

• Introduction
• RESTful Service Design
• Simple Doodle Service Example & Demo
• Some REST-inspired SOA Design Patterns
  • Entity Endpoint
  • Uniform Contract
  • Endpoint Redirection
  • Content Negotiation
• Discussion
Web Sites (1992)

Web Browser ➔ HTML ➔ HTTP ➔ Web Server

WS-* Web Services (2000)

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

WS-* Web Services (2000)
Is REST being used?

Slide from Paul Downey, BT
RESTful Service Design

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>✗</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>
### Design Space

#### M Representations (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>
A new kind of Service

- From Service Capabilities to Resources
- From Service Contracts to the Uniform Contract

- GetInvoice
- ReportYearEnd
- UpdateClient

GET /invoices/{id}
GET /reports/{year}
PUT /clients/{id}
Simple Doodle REST API Example

1. Resources: **polls and votes**

2. Containment Relationship:

   - **poll**
     - {id1}
     - vote
       - {id4}
       - {id5}
     - {id2}
     - {id3}

   - **GET**
   - **PUT**
   - **POST**
   - **DELETE**

<table>
<thead>
<tr>
<th>URI</th>
<th>GET</th>
<th>PUT</th>
<th>POST</th>
<th>DELETE</th>
</tr>
</thead>
<tbody>
<tr>
<td>/poll</td>
<td>✓</td>
<td></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></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>

   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

```plaintext
GET /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)

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

```
GET /poll/090331x
<options>A,B,C</options>
<votes><vote id="/1"
<option>name</option>C. Pautasso</name>
<option>choice</option>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
```
What is your SOA Connector today?

RPC

ESB

WWW
Design Patterns

Content Negotiation

Entity Endpoint

Endpoint Redirect

M Representations (Variable)

4 Methods (Fixed)

<table>
<thead>
<tr>
<th>/loan</th>
<th>GET</th>
<th>PUT</th>
<th>POST</th>
<th>DELETE</th>
</tr>
</thead>
<tbody>
<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>

Uniform Contract
Pattern: Uniform Contract

- How can consumers take advantage of multiple evolving service endpoints?
- **Problem:** Accessing similar services requires consumers to access capabilities expressed in service-specific contracts. *The consumer needs to be kept up to date* with respect to many evolving individual contracts.
Pattern: Uniform Contract

- Solution: **Standardize** a uniform contract across alternative service endpoints that is abstracted from the specific capabilities of individual services.
- Benefits: Service Abstraction, Loose Coupling, Reusability, Discoverability, Composability.
## Example Uniform Contract

<table>
<thead>
<tr>
<th>CRUD</th>
<th>HTTP</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>
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}

$201 \text{ Created}$

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

$301 \text{ Moved Permanently}$

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
Pattern: Endpoint Redirection

- How can consumers of a service endpoint adapt when service inventories are restructured?
- Problem: Service inventories may change over time for business or technical reasons. It may not be possible to replace all references to old endpoints simultaneously.
- Solution: Automatically refer service consumers that access the stale endpoint identifier to the current identifier.
Endpoint Redirection with HTTP

- HTTP natively supports the Endpoint redirection pattern 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!
How can entities be positioned as reusable enterprise resources?

Problem: A service with a single endpoint is too coarse-grained when its capabilities need to be invoked on its data entities. A consumer 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 reused and shared among multiple services.
Pattern: Entity Endpoint

- Solution: expose each entity as individual lightweight endpoints of the service they reside in
- Benefits: Global addressability of service entities
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)
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/lugano
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
Pattern: Content Negotiation

- How can services support different consumers without changing their contract?
- 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 capability for each kind of consumer.
Pattern: 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
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).

\textbf{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.

\textbf{Accept: text/html, text/*; q=0.1}

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

\textbf{Accept: application/xhtml+xml, 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?
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,</td>
<td>Content-Type:</td>
</tr>
<tr>
<td></td>
<td>application/json</td>
<td></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 for the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Content-Type header</td>
</tr>
<tr>
<td>Accept-Encoding:</td>
<td>compress, gzip</td>
<td>Content-Encoding:</td>
</tr>
</tbody>
</table>
More SOA with REST Patterns

1. Uniform Contract
2. Entity Endpoint
3. Entity Linking*
4. Content Negotiation
5. Distributed Response Caching*
6. Endpoint Redirection
7. Idempotent Capability*
8. Message-based State Deferral*
9. Message-based Logic Deferral*
10. Consumer-Processed Composition*

*Not Included in this talk
Conclusion

- GetInvoice
- ReportYearEnd
- UpdateClient

GET /invoices/{id}
GET /reports/{year}
PUT /clients/{id}
Conclusion

- 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


