Università della Svizzera italiana

Faculty of Informatics

Some REST Design Patterns (and Anti-Patterns)

Cesare Pautasso
Faculty of Informatics
University of Lugano, Switzerland

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

Abstract

- The REST architectural style is simple to define, but understanding how to apply it to design concrete REST services in support of SOA can be more complex. The goal of this talk is to present the main design elements of a RESTful architecture and introduce a pattern-based design methodology for REST services.
- A selection of REST-inspired SOA design patterns taken from the upcoming "SOA with REST" book will be explained and further discussed to share useful solutions to recurring design problems and to also the foundational building blocks that comprise the REST framework from a patterns perspective.
- We will conclude by introducing some common SOA anti-patterns particularly relevant to the design of REST services in order to point out that not all current Web services that claim to be RESTful are indeed truly so.

Acknowledgements

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

- And all the participants, sheperds and sheeps of the SOA Patterns Workshop

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)

REST Patterns - Outline

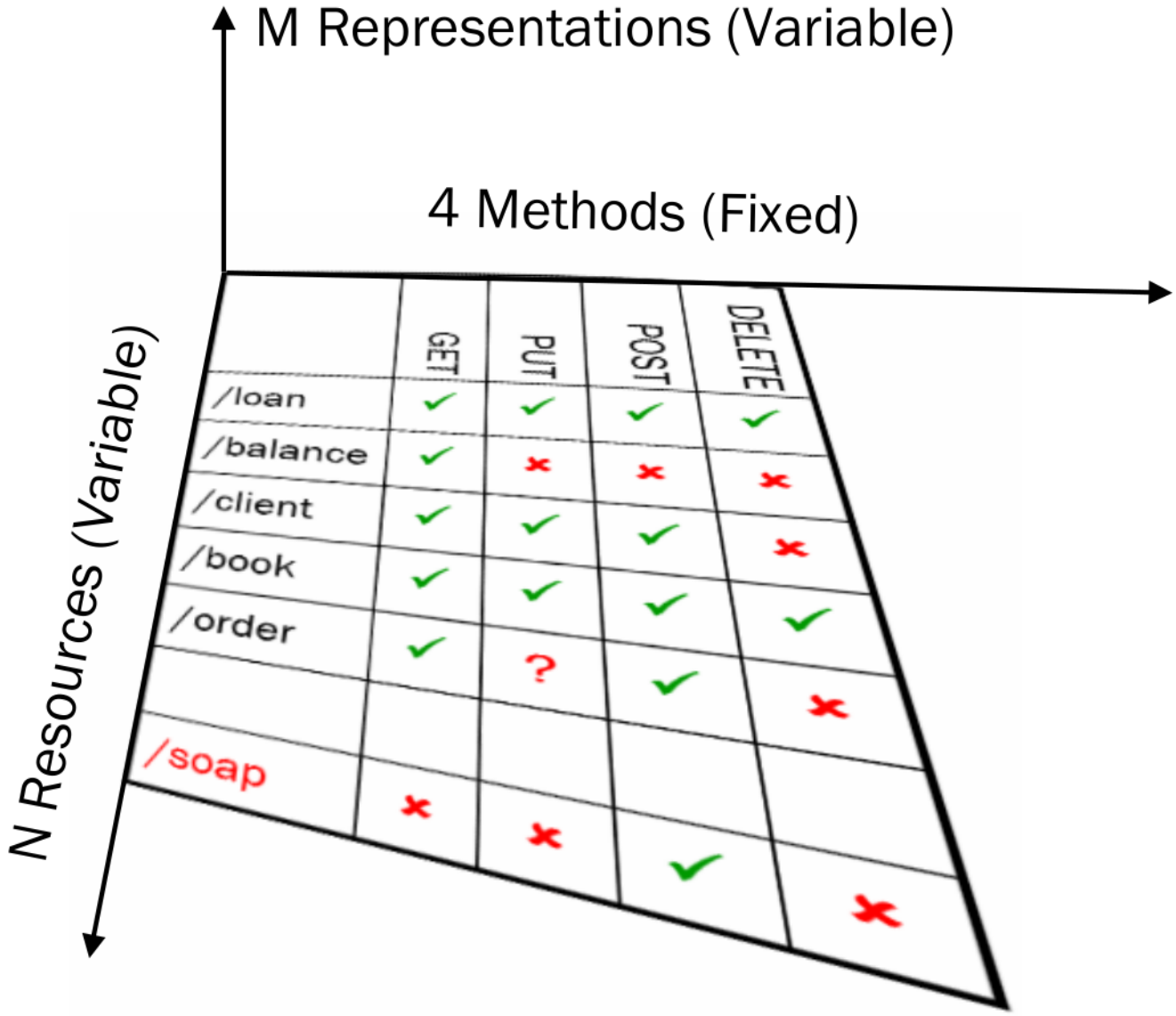
- Design Methodology
- Simple Doodle Service Example & Demo
- SOA Design Patterns
  - Entity Endpoint
  - Uniform Contract
  - Endpoint Redirection
  - Content Negotiation
  - Idempotent Capability
- AntiPatterns
  - Tunneling everything through GET
  - Tunneling everything through POST

Design Methodology for REST

1. Identify resources to be exposed as services (e.g., yearly risk report, book catalog, purchase order, open bugs, polls and votes)
2. Model relationships (e.g., containment, reference, state transitions) between resources with hyperlinks that can be followed to get more details (or perform state transitions)
3. Define “nice” URIs to address the resources
4. Understand what it means to do a GET, POST, PUT, DELETE for each resource (and whether it is allowed or not)
5. Design and document resource representations
6. Implement and deploy on Web server
7. Test with a Web browser

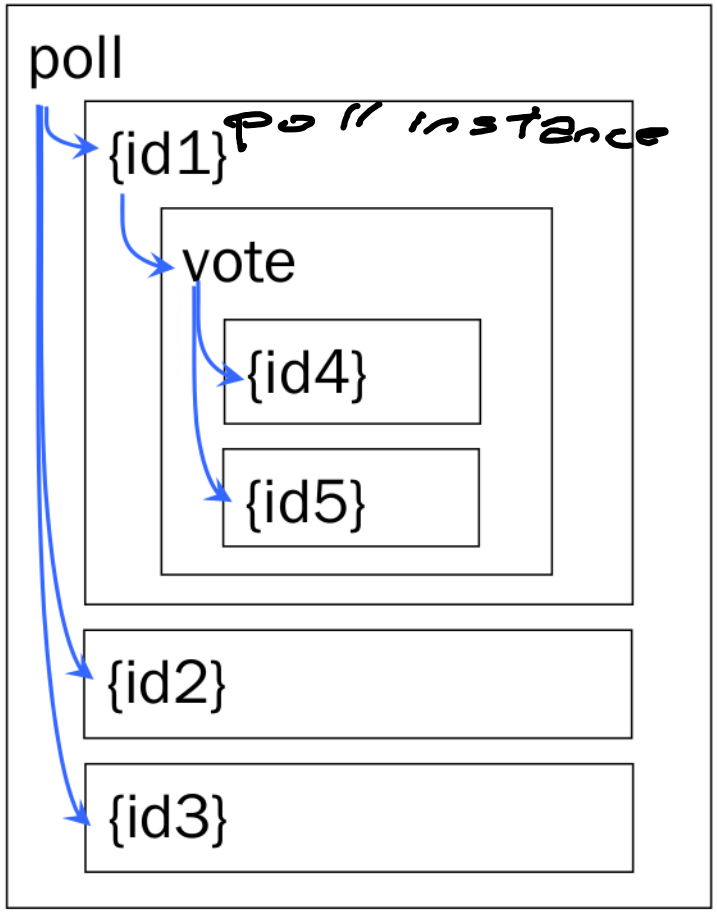
	GET	PUT	POST	DELETE
/loan	✓	✓	✓	✓
/balance	✓	✗	✗	✗
/client	✓	✓	✓	✗
/book	✓	✓	✓	✓
/order	✓	?	✓	✗
/soap	✗	✗	✓	✗

Design Space



Simple Doodle API Example Design

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



	GET	PUT	POST	DELETE	
/poll (container resource)	✓	✗	✓	✗	PATTERN
/poll/{id} (child element)	✓	✓	✗	✓	PATTERN
/poll/{id}/vote (container resource)	✓	✗	✓	✗	PATTERN
/poll/{id}/vote/{id}	✓	✓	✗	?	//

state of the vote

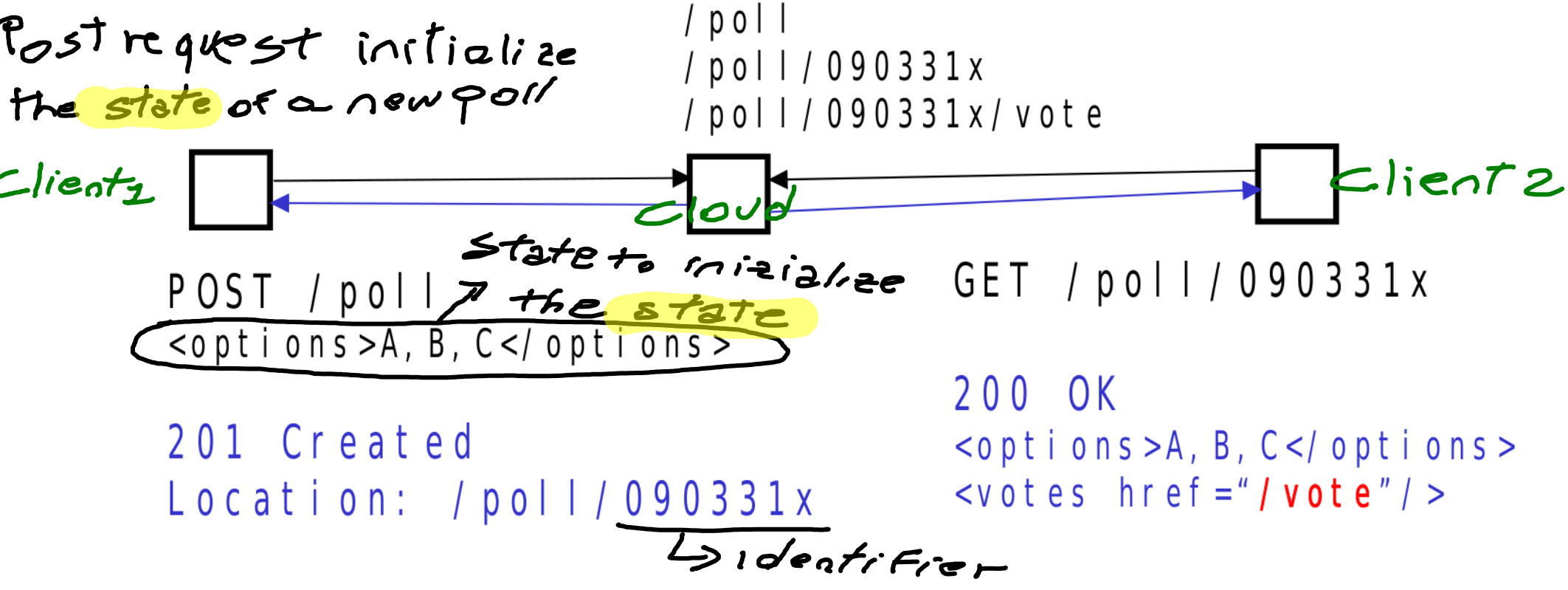
3. URIs embed IDs of "child" instance resources
4. POST on the container is used to create child resources
5. PUT/DELETE for updating and removing child resources

element of the container

We apply a uniform interface (GET, POST...) to this resources

Simple Doodle API Example

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

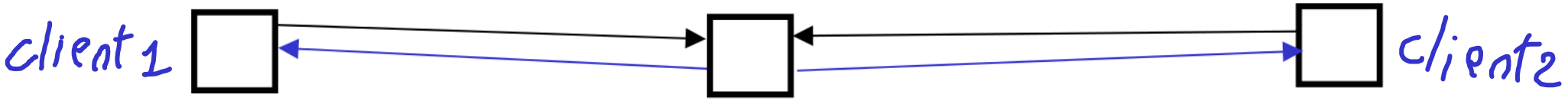


2. Reading a poll
(transfer the state of the poll from the Doodle service)

Simple Doodle API Example

- Participating in a poll by creating a new vote sub resource

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




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

201 Created
Location:
/poll/090331x/vote/1
```

if I want to update my vote

```
GET /poll/090331x

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

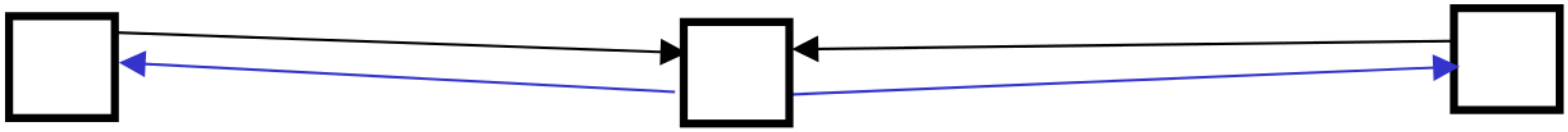
identifier

the state corresponding to the same identifier is now changed

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
```

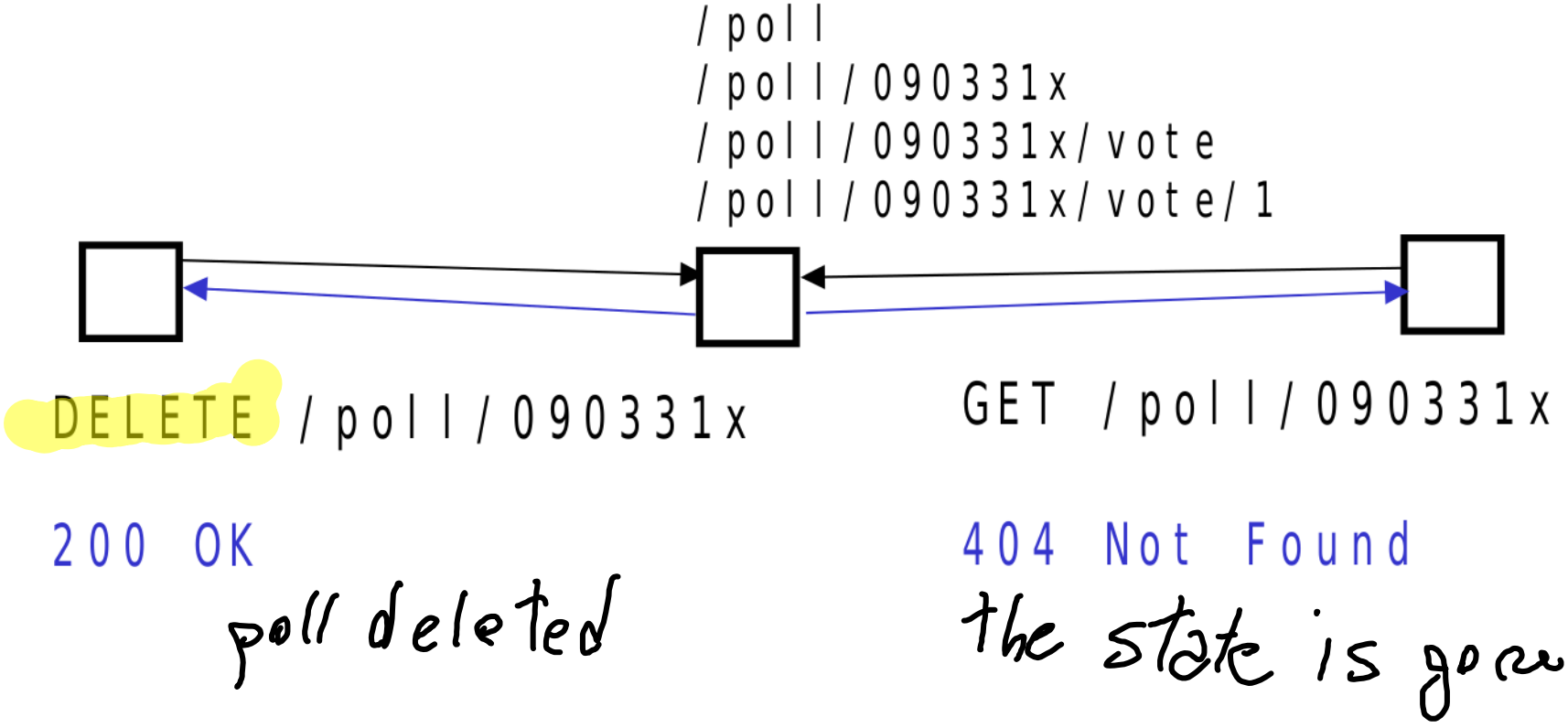
If something fails I can repeat the request how many times I want

```
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

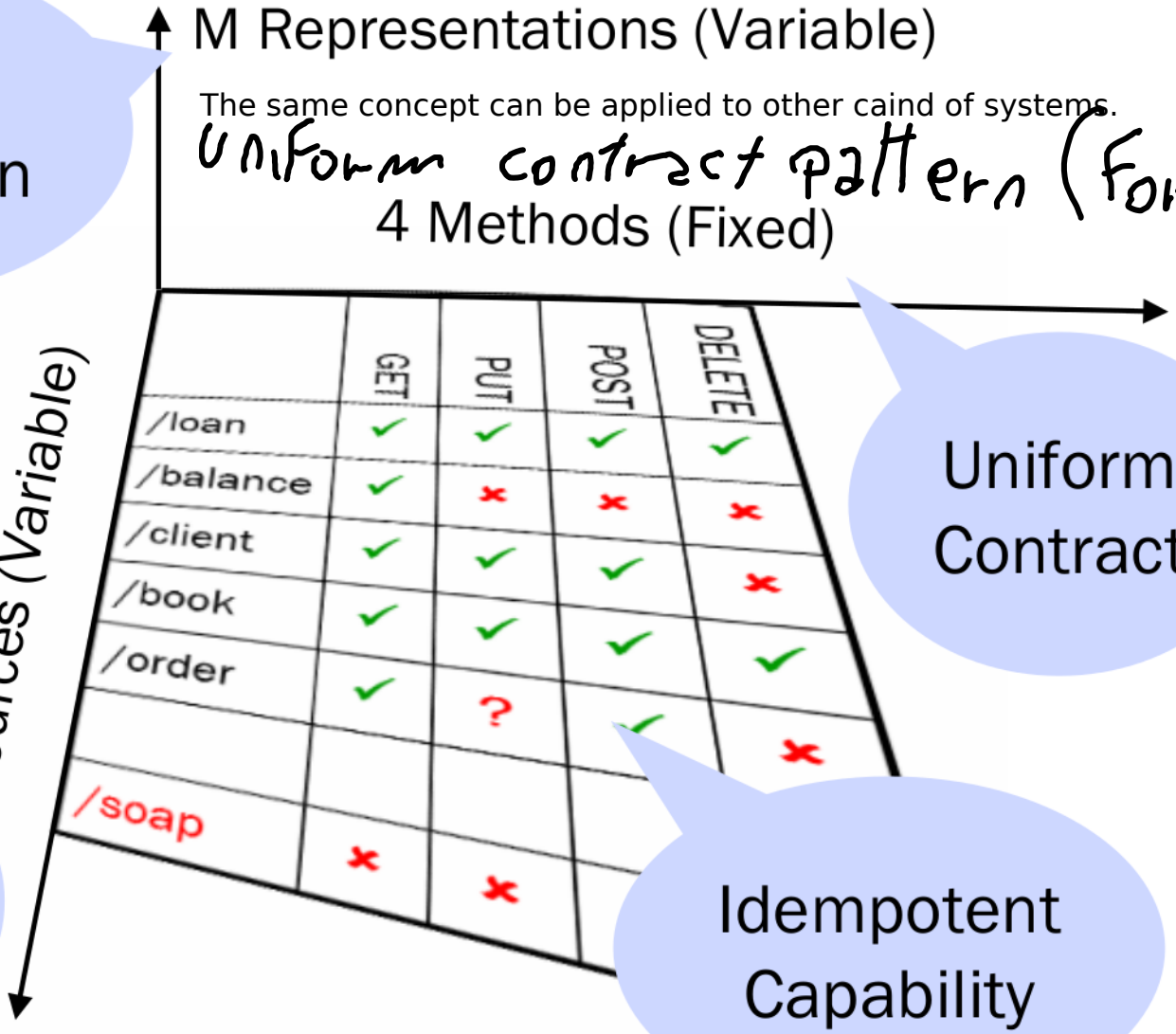


Design Patterns

Content Negotiation

Entity Endpoint

Endpoint Redirect



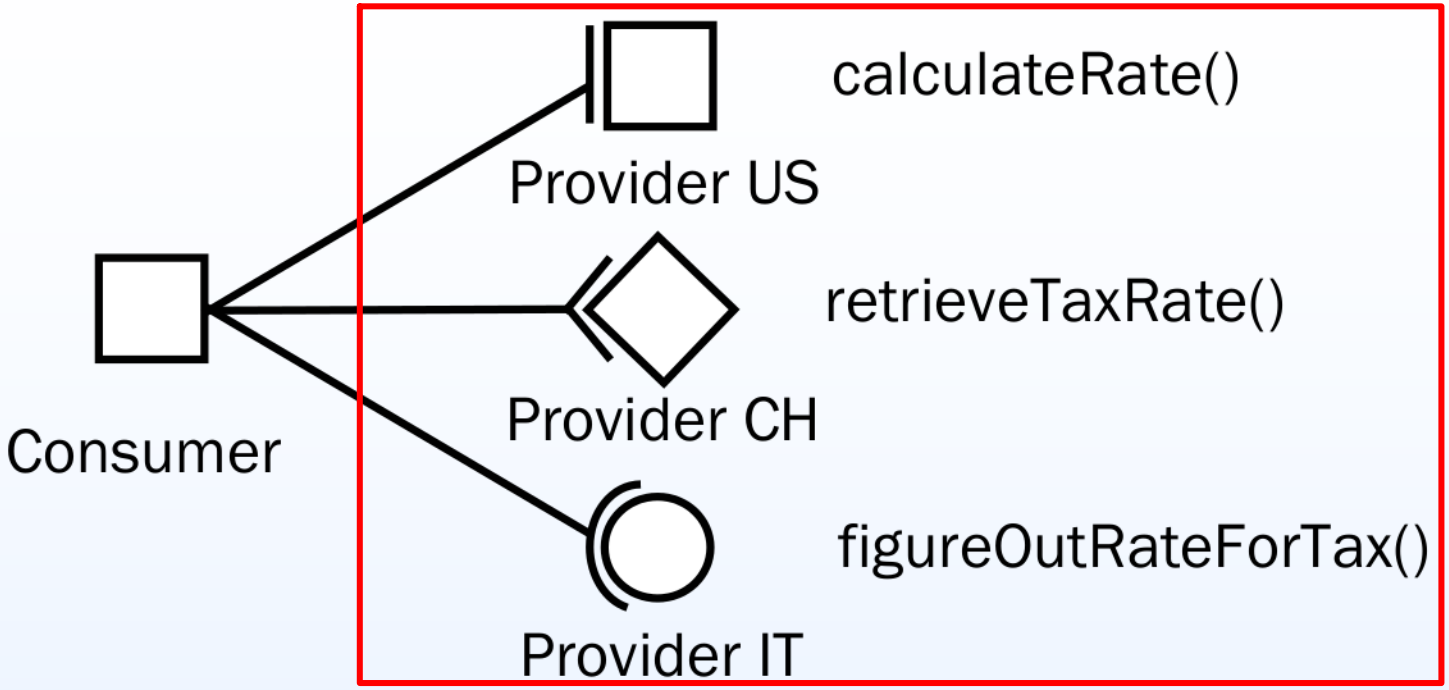
Uniform Contract

Idempotent Capability

The same concept can be applied to other caind of systems.

Uniform contract pattern (for HTTP)

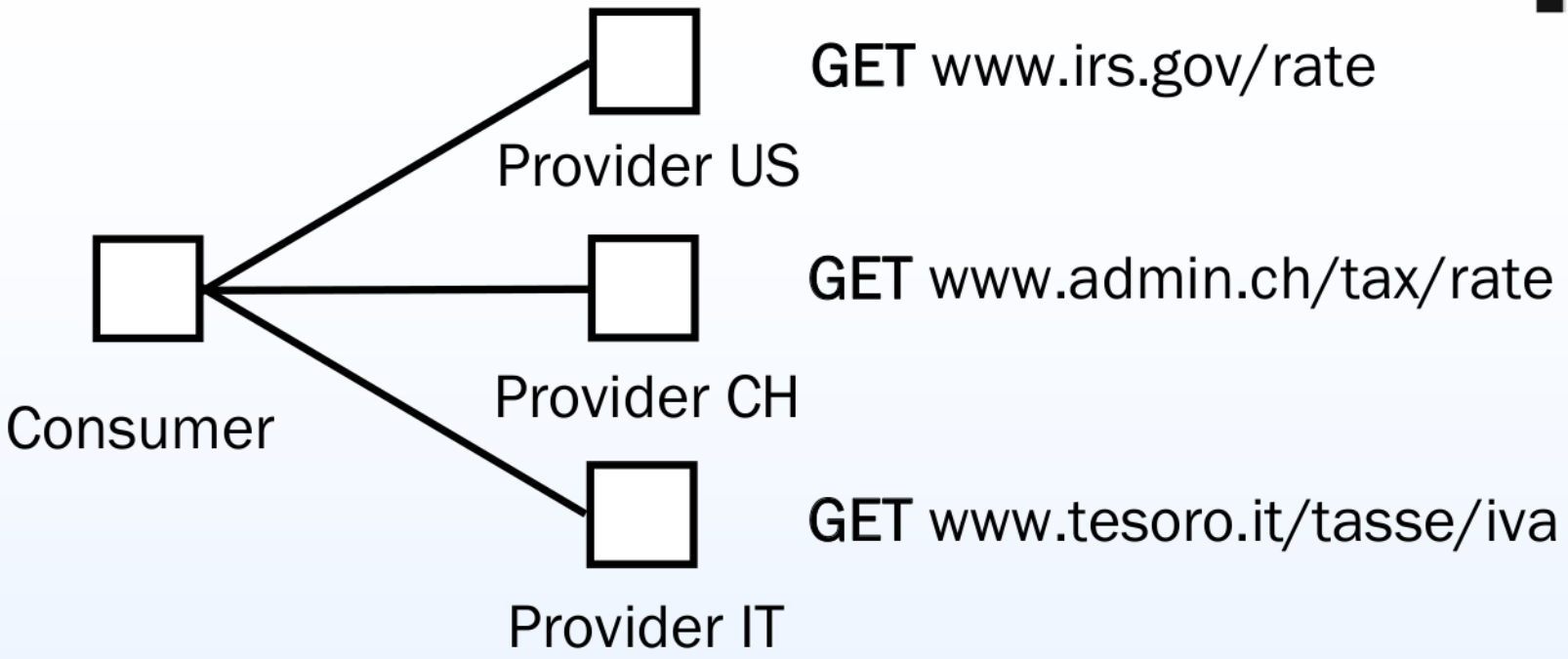
Pattern: Uniform Contract



services endpoints

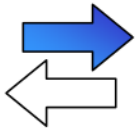
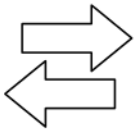
- How can consumers take advantage of multiple evolving service endpoints?
- **Problem:** Accessing similar services requires consumers to access capabilities expressed in service specific contracts. *The consumer needs to be kept up to date* with respect to many evolving individual contracts.

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

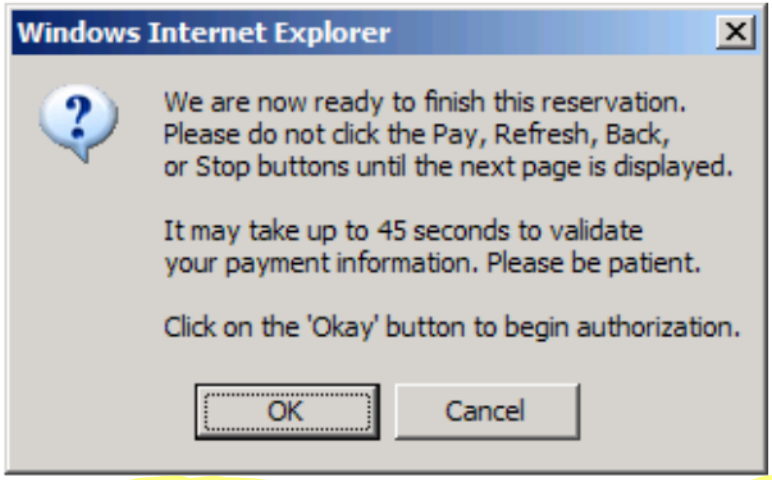
CRUD	REST	
CREATE	POST 	Create a sub resource
READ	GET 	Retrieve the *current state* of the resource
UPDATE	PUT 	Initialize or update the state of a resource at the given URI
DELETE	DELETE 	Clear a resource, after the URI is no longer valid

POST vs. GET

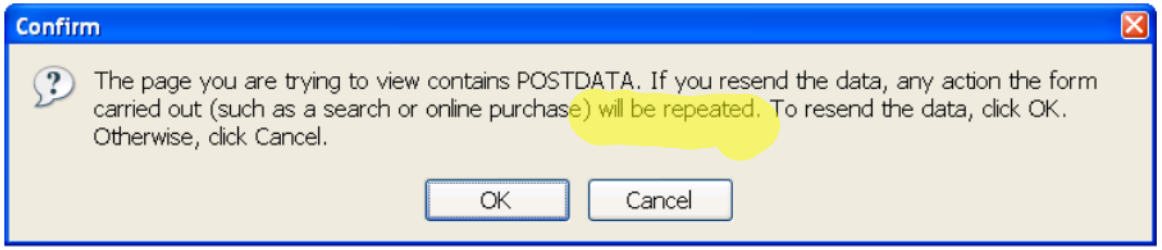
- GET is a **read-only** operation. It can be repeated without affecting the state of the resource (idempotent) and can be cached.

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.



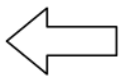
Web browsers warn you when refreshing a page generated with POST



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)

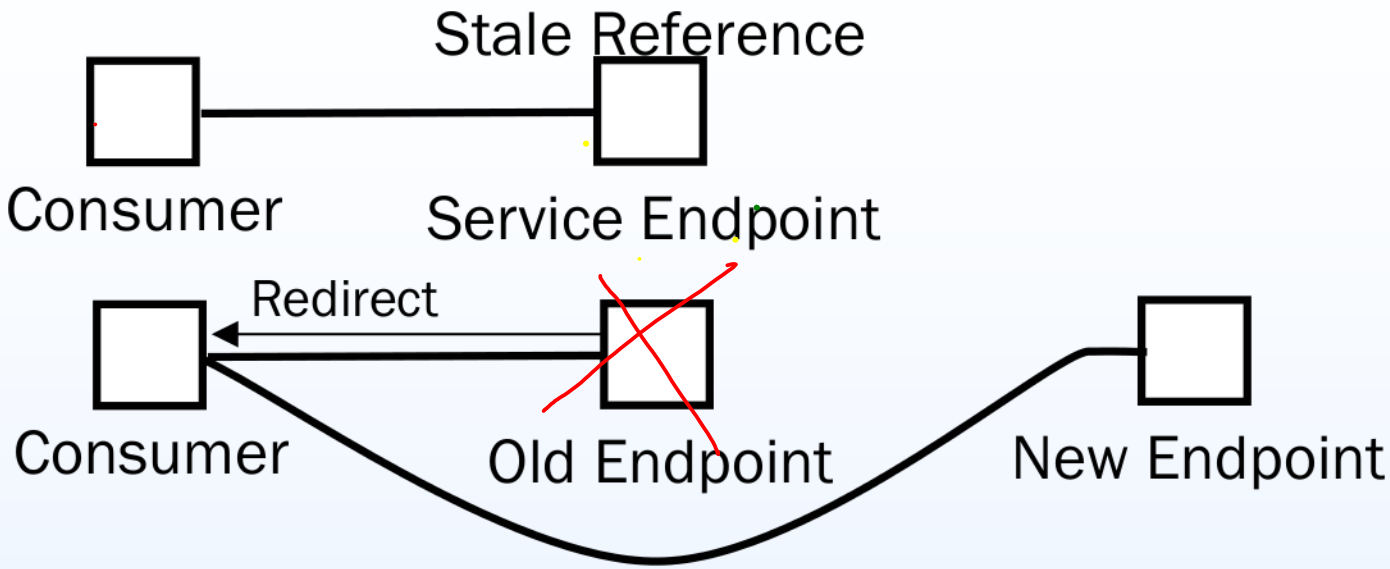
```
POST /resource
```

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

Solution 2: let the server compute the unique id

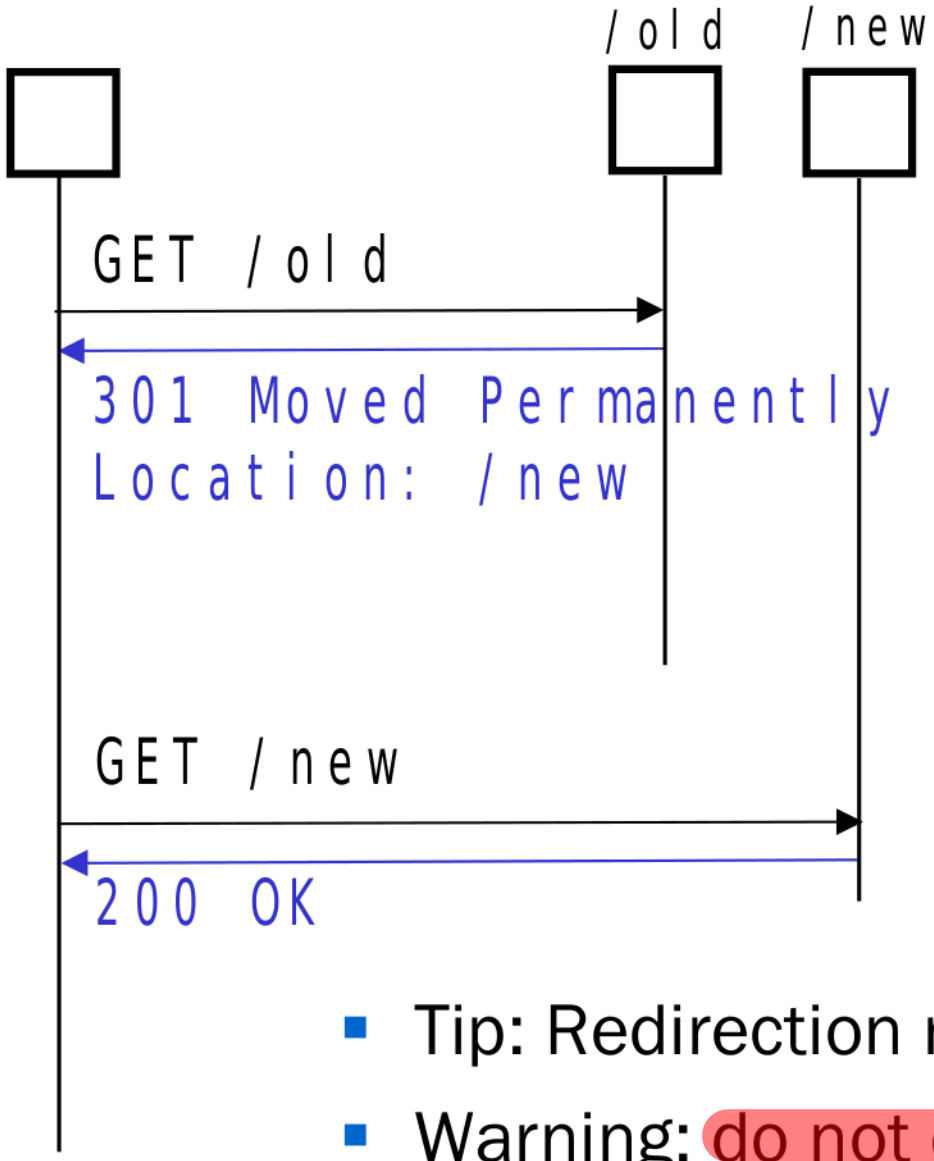
Problem: Duplicate instances may be created if requests are repeated due to unreliable communication

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~~ old endpoint identifier to the current identifier.

Endpoint Redirection with HTTP

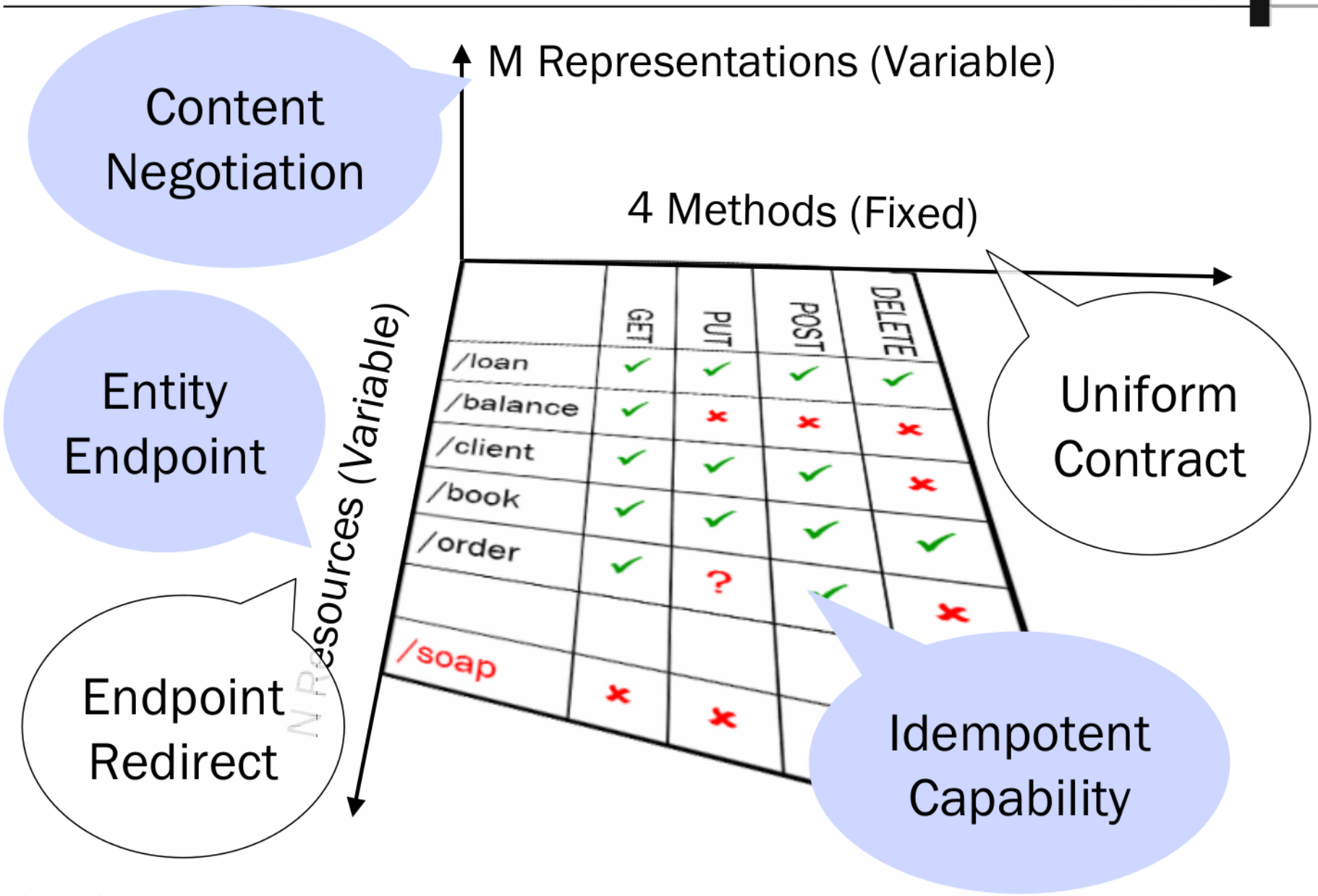


- HTTP natively supports the Endpoint redirection pattern using a combination of 3xx status codes and standard headers:
  - 301 Moved Permanently
  - 307 Temporary Redirect
  - Location: /newURI

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

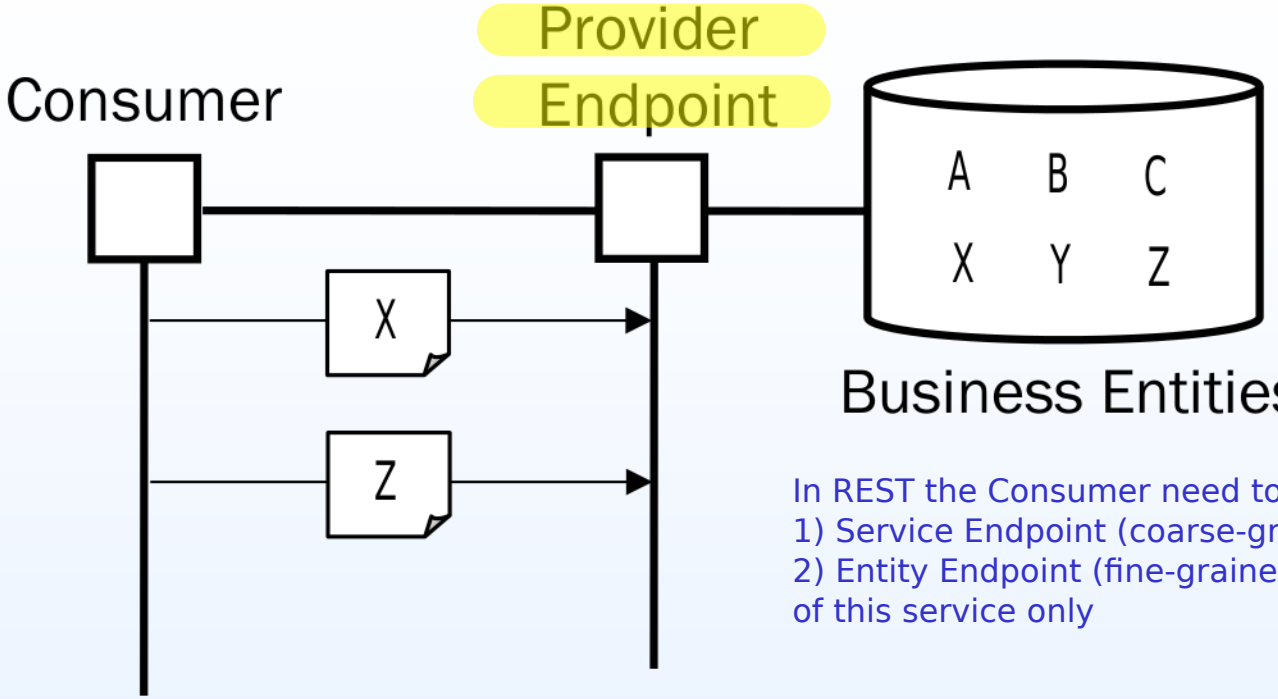
to avoid loops is a good idea to use a library that does that. It is not reccomanded to redirect by hand

Design Patterns



NOTE: In SOAP there is one EndPoint for the entire service

Pattern: Entity Endpoint

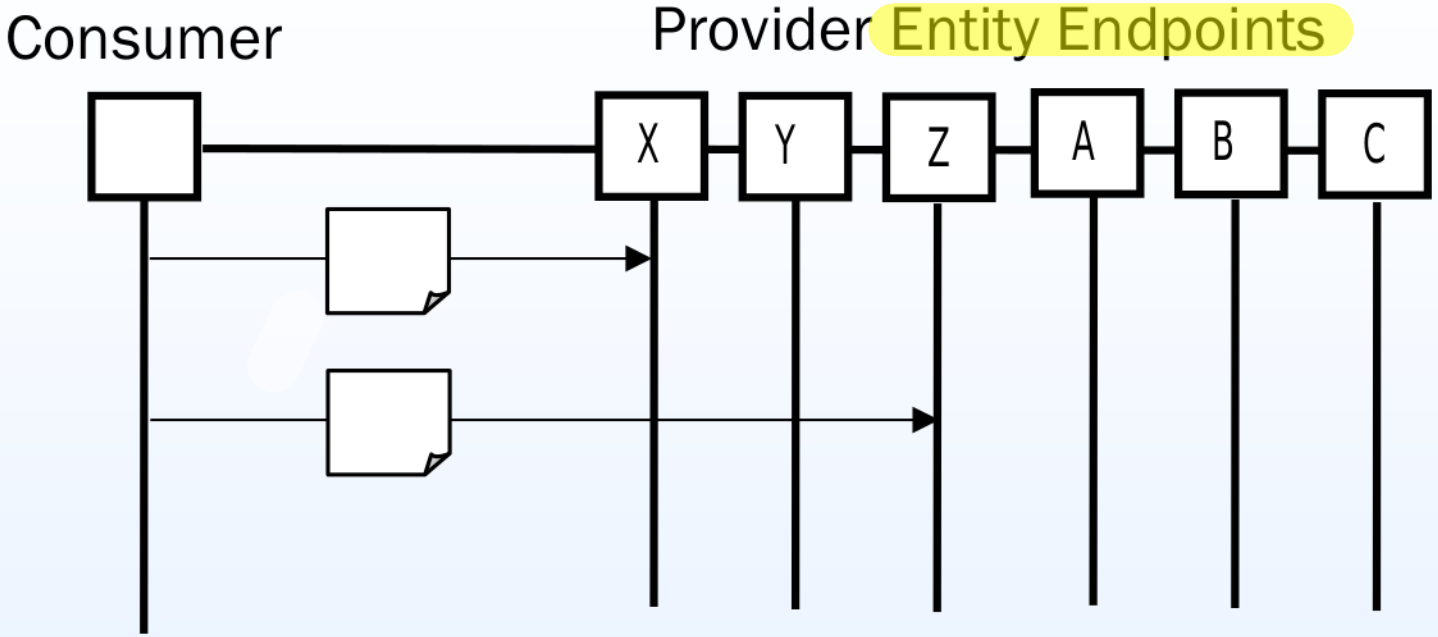


In REST the Consumer need to remember to things:
1) Service Endpoint (coarse-grained)
2) Entity Endpoint (fine-grained) valid on the content of this service only

- 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

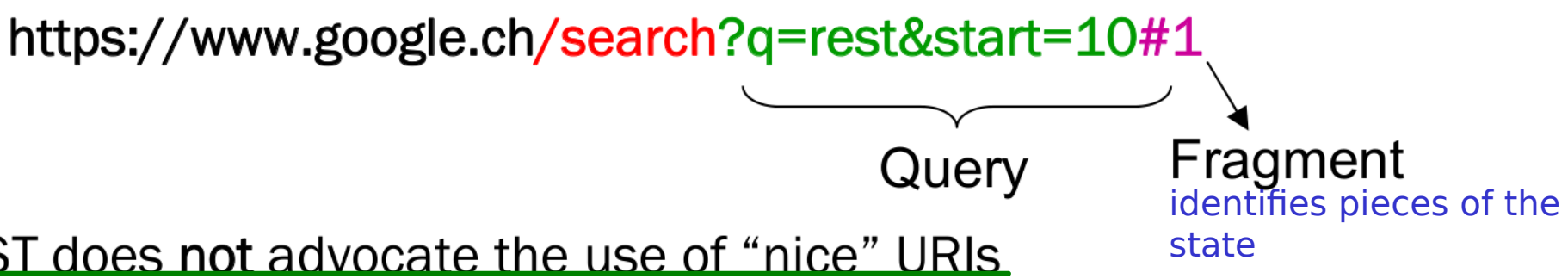
The alternative way to do this is:
we talk directly with our entities without going through a central Endpoint



- Solution: expose each entitity as individual lightweight endpoints of the service they reside in
- Benefits: Global addressability of service entities from everybody else

URI - Uniform Resource Identifier

- Internet Standard for resource naming and identification (originally from 1994, revised until 2005)
- Examples:



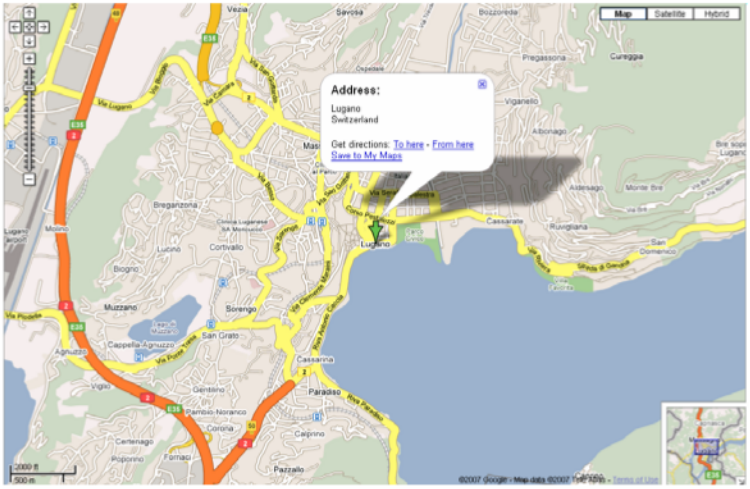
- REST does **not** advocate the use of "nice" URIs
- In most HTTP stacks URIs cannot have arbitrary length (4Kb)

What is a “nice” URI?

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

http://map.search.ch/lugano

http://maps.google.com/lugano



http://maps.google.com/maps?f=q&hl=en&q=lugano,+switzerland&layer=&ie=UTF8&z=12&om=1&iwloc=addr

NOTE that all this link can rappresent RESTful services or not

Redirect is a temporary solution

URI Design Guidelines

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

GET /book?isbn=24&action=delete

DELETE /book/24

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

- Some use URI postfixes to specify the content type

- *This may break the abstraction*

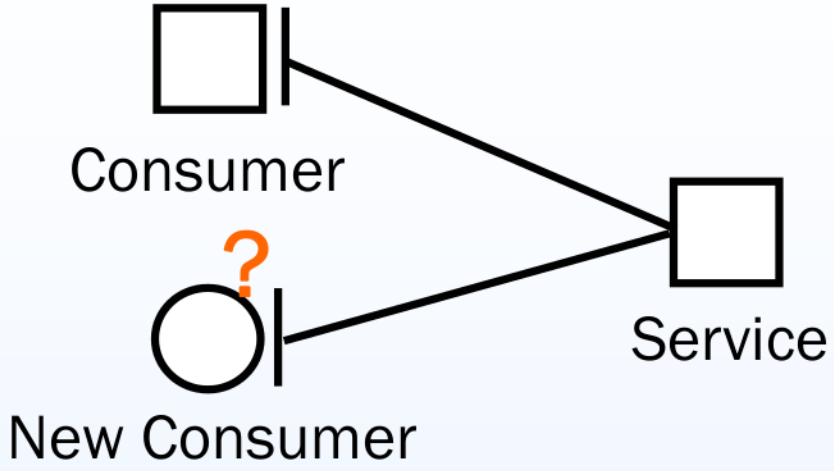
- Do not change URIs
- Use redirection if you really need to change them

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

In order to support different "requirement" or different "consumers of services"
==> evoiding to introduce a new contract for a new service

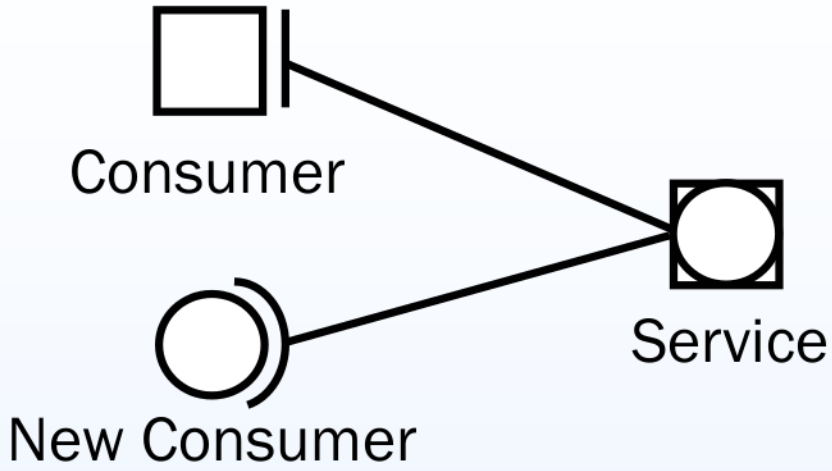
Pattern: Content Negotiation

E.g. I want to go from XML to JSON and the contract service's contract remain the same



- 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



The solution is to Adapt on the fly at the need of the 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

==> support multiple multiple format agreed at runtime

Content Negotiation in HTTP

Negotiating the message format does not require to send more messages (the added flexibility comes for free)

⇨ `GET /resource`

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

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

⇦ `200 OK`

`Content-Type: application/json`

2. The server chooses the most appropriate one for the reply (status 406 if none can be found)

Advanced Content Negotiation

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

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

⇒ Priority for the Formats request

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

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

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

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

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

Forced Content Negotiation

The generic URI supports content negotiation

```
GET /resource
Accept: text/html, application/xml,
    application/json
```

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

```
GET /resource.html
GET /resource.xml
GET /resource.json
```

This postfix points to the specific representation

This is not necessarily the best practice: instead to send an additional header we send the extension

Warning: This is a conventional practice, not a standard.

What happens if the resource cannot be represented in the requested format?

The representation and identification of the resources are mixed together ☹

Multi-Dimensional Negotiation

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

Request Header	Example Values	Response Header
Accept:	application/xml, application/json	Content-Type:
Accept-Language:	en, fr, de, es	Content-Language:
Accept-Charset:	iso-8859-5, unicode-1-1	Charset parameter fo the Content-Type header
Accept-Encoding:	compress, gzip	Content-Encoding:

internationalization

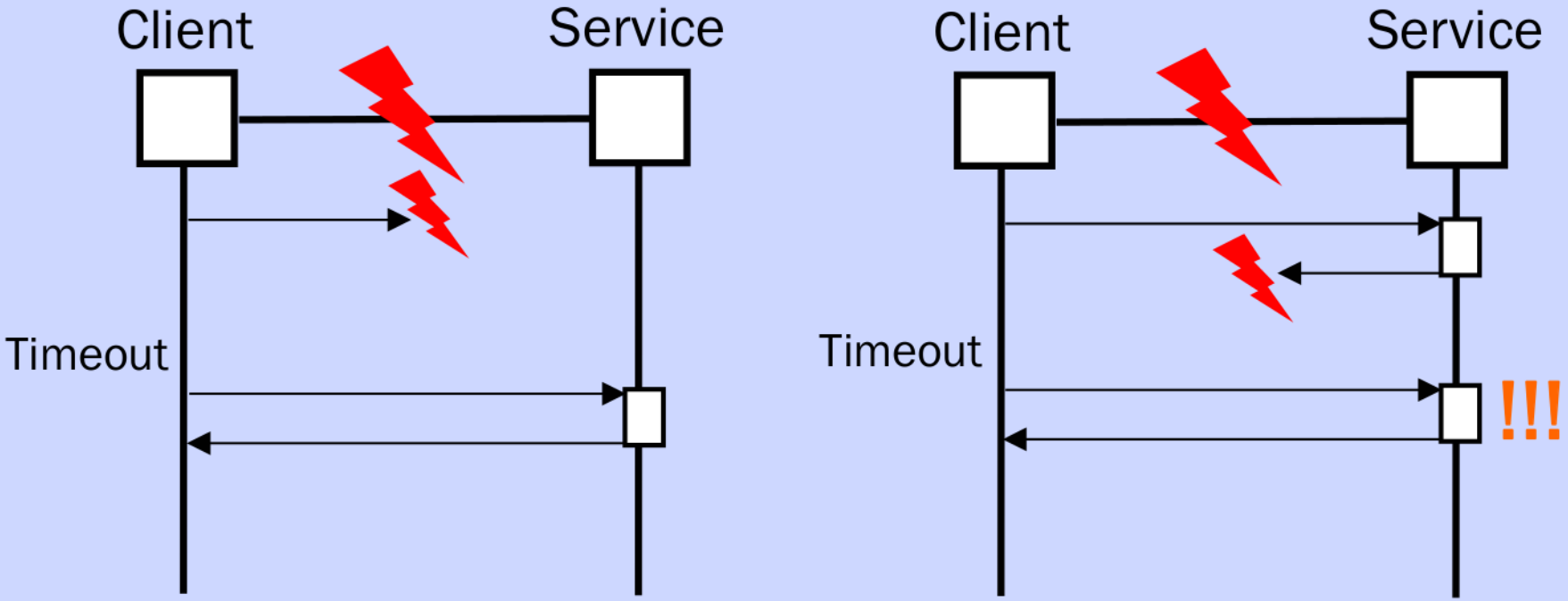
negoziate the 5

allows to compress the data exchange on the fly

everything is already supported by the http protocol

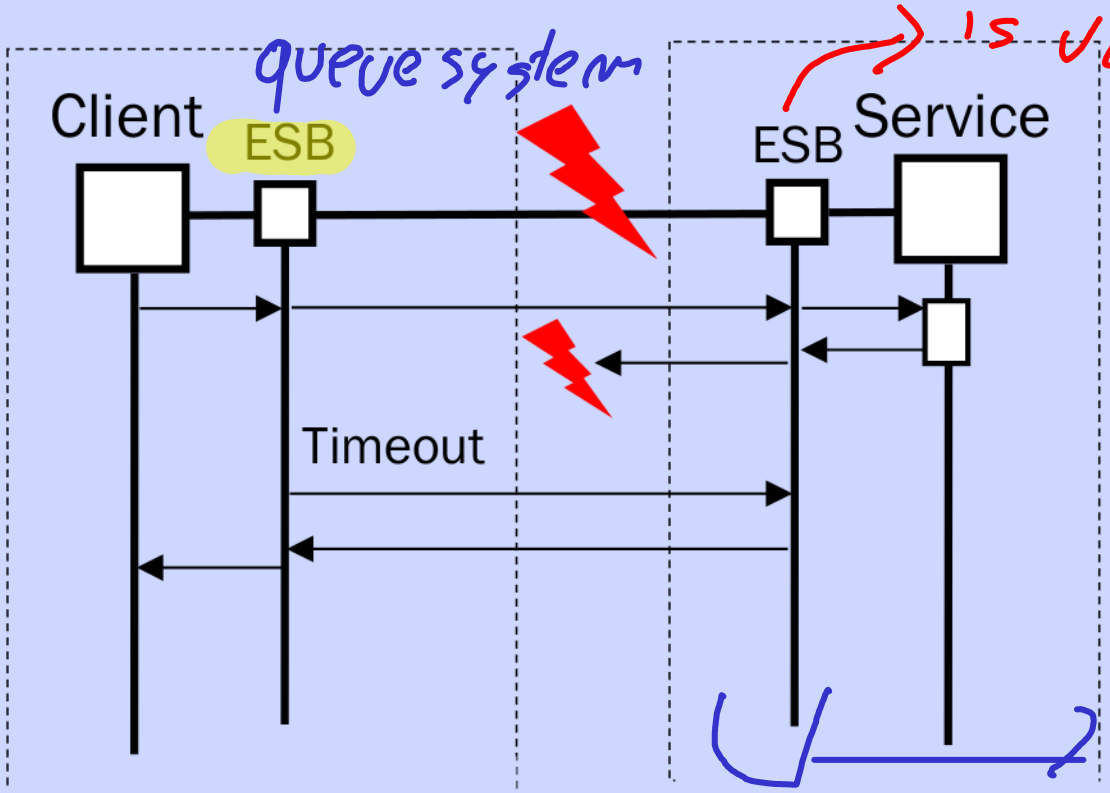
Idempotent cap. came to play when a failure occurs

Pattern: Idempotent Capability



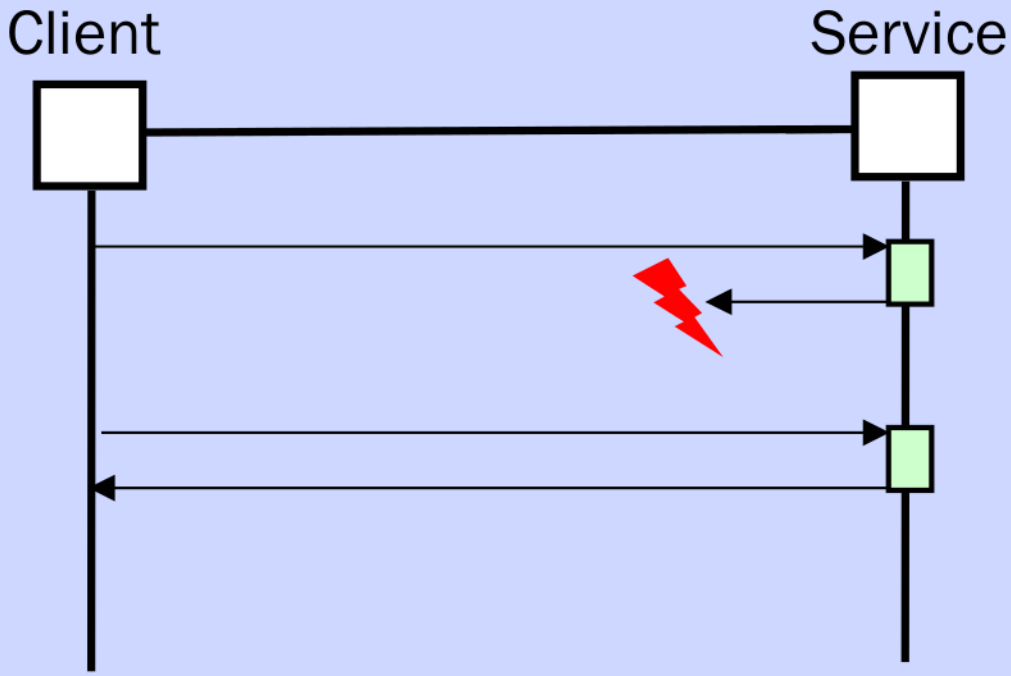
- How can a service consumer recover from lost messages after network disruption or server failure within a service cluster?
- Problem: Service oriented architectures are distributed systems. Failures (such as the loss of messages) may occur during service capability invocation. A lost request should be retried, but a lost response may cause unintended side-effects if retried automatically.

Pattern: Idempotent Capability



- Solution: use an ESB, with support for reliable messaging.
- Problem: do we always need this? Are there some messages more critical than others?

Pattern: Idempotent Capability



- **Simpler Solution:** if possible use idempotent service capabilities, whereby services provide a guarantee that capability invocations are safe to repeat in the case of failures that could lead to a response message being lost

Idempotent vs. Unsafe

- Idempotent requests can be processed multiple times without side-effects

```
GET /book
PUT /order/x
DELETE /order/y
```

- If something goes wrong (server down, server internal error), the request can be simply replayed until the server is back up again
- Safe requests are idempotent requests which do not modify the state of the server (can be cached)

```
GET /book
```

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

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

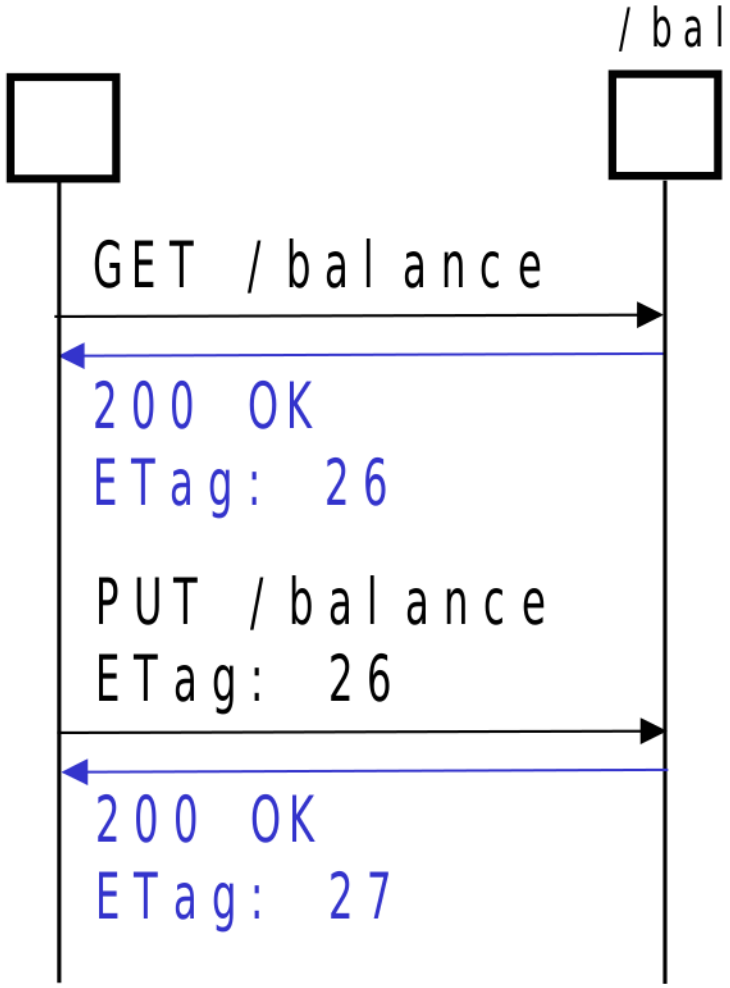
- Unsafe requests require special handling in case of exceptional situations (e.g., state reconciliation)

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

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

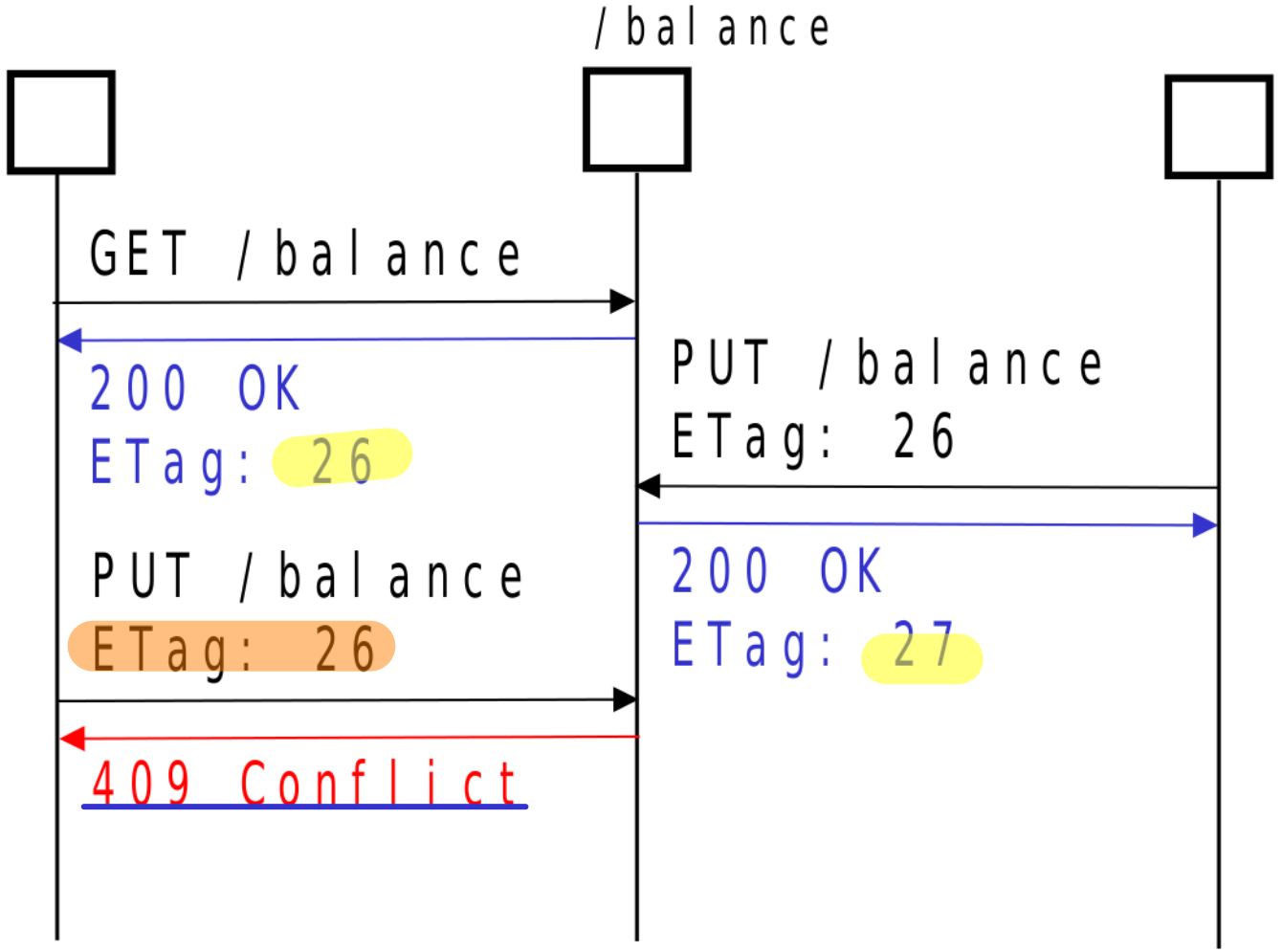
```
B = GetBalance()  //safe
B = B + 200$      //local
SetBalance(B)     //idempotent
```

Dealing with Concurrency



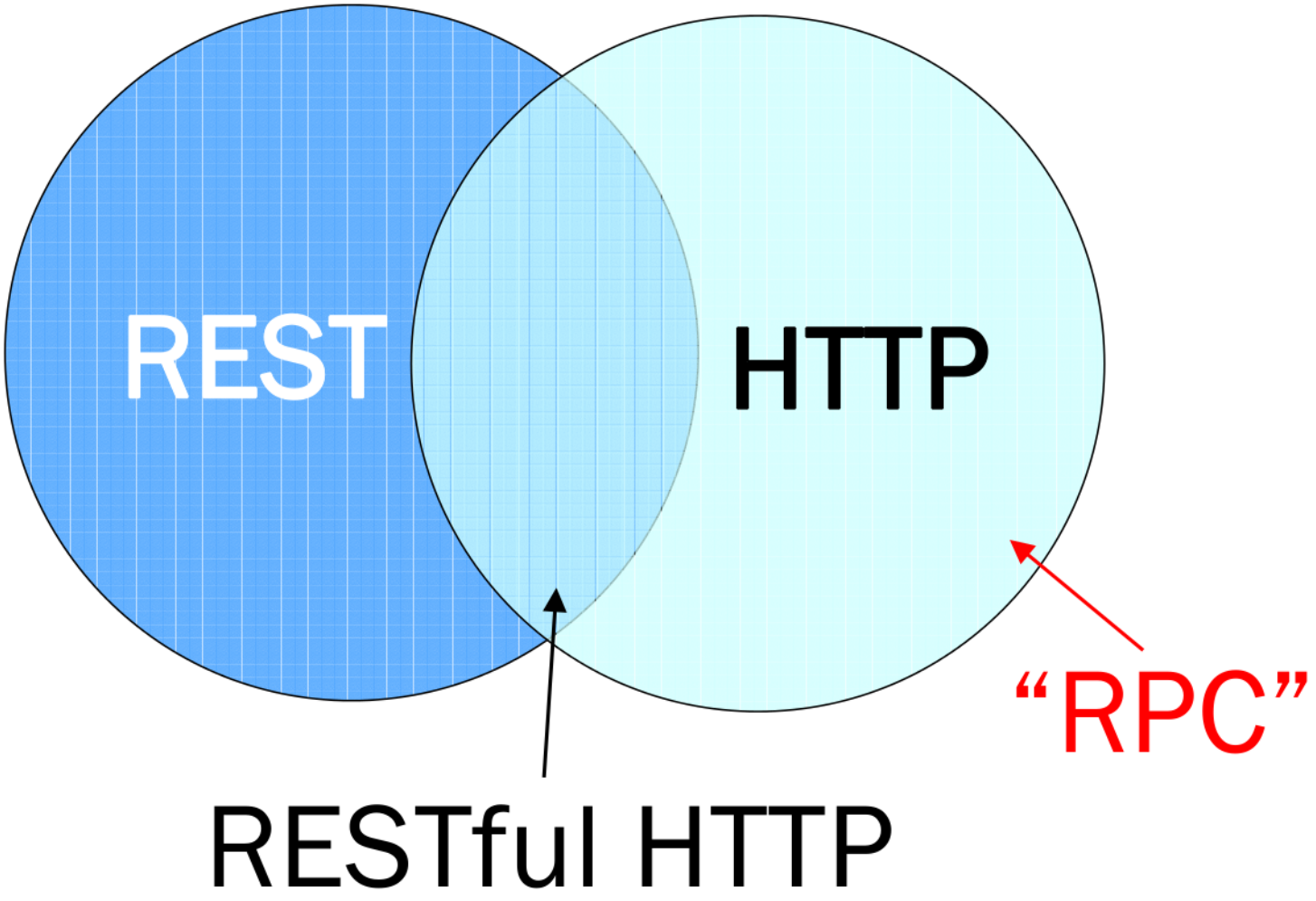
- Breaking down the API into a set of idempotent requests helps to deal with temporary failures.
- But what about if another client concurrently modifies the state of the resource we are about to update?
- Do we need to create an explicit /balance/lock resource? (Pessimistic Locking)
- Or is there an optimistic solution?

Dealing with Concurrency



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

Antipatterns - REST vs. HTTP



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 and should only be used for "dangerous" requests)

```
POST /service/endpoint

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

REST Design 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

References

- R. Fielding, Architectural Styles and the Design of Network-based Software Architectures, PhD Thesis, University of California, Irvine, 2000
- C. Pautasso, O. Zimmermann, F. Leymann, RESTful Web Services vs. Big Web Services: Making the Right Architectural Decision, Proc. of the 17th International World Wide Web Conference (WWW2008), Bejing, China, April 2008
- C. Pautasso, BPEL for REST, Proc. of the 7th International Conference on Business Process Management (BPM 2008), Milano, Italy, September 2008
- C. Pautasso, Composing RESTful Services with JOpera, In: Proc. of the International Conference on Software Composition (SC2009), July 2009, Zurich, Switzerland.

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

SOA with REST,

Prentice Hall,
to appear in 2010