RESTful Service Pattern

Cheng Wang, Jiayu Li, Rohit More, Zheng Zhan

CSE 776 Design Pattern

REST

A software architectural style that defines a set of constraints to be used for creating web services.

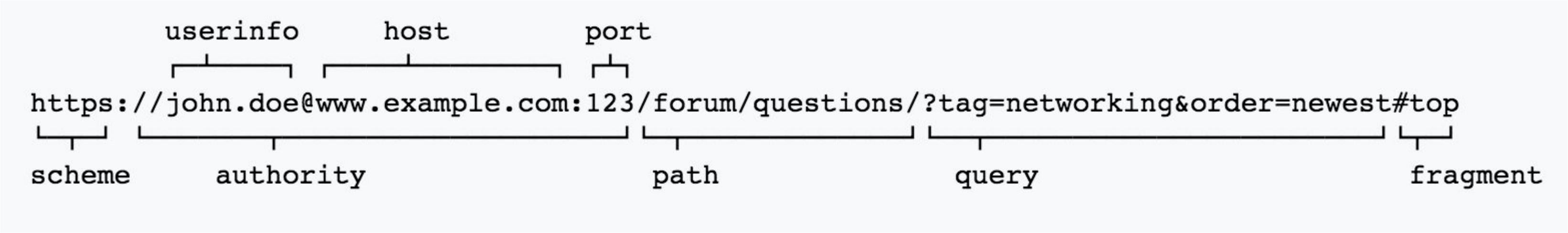
REpresentation
State
Transfer

RESTful web services allow the requesting systems to access and manipulate textual representations of web resources by using a uniform and predefined set of stateless operations.

URI

URI: A Uniform Resource Identifier (URI) is a string of characters that unambiguously identifies a particular resource.

URI = scheme:[//authority]path[?query][#fragment]



```
          userinfo       host            port
https://john.doe@www.example.com:123/forum/questions/?tag=networking&order=newest#top
scheme    authority                  path             query                       fragment
```

without REST

POST /library/book1/getBook

POST /library/createBook

POST /library/book3/updateBook

POST /library/book4/deleteBook

URI with CRUD in REST

1. **GET /library/book1/**
   Obtain book1 information
2. **POST /library**
   Create a book
3. **PUT /library/book3**
   Update book3 information
4. **DELETE /library/book4**
   Delete book4 information

Operation	**RESTful WS**
Create	POST
Read (Retrieve)	GET
Update (Modify)	PUT
Delete (Destroy)	DELETE

Six Constraints

1. Client-Server
2. Stateless
3. Cache
4. Uniform Interface
5. Layered System
6. Code-On-Demand

Client-Server

Client-server: Separation of concerns. By separating the **user interface concerns** from the **data storage concerns**

Pros: Portability, Scalability

Stateless

Stateless: Requests from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server.

Pros: Visibility, Reliability, Scalability

Cons: Decreasing network performance

Cache

Cache: Data within a response to a request be implicitly or explicitly labeled as cacheable or non-cacheable. If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests.

Pros: Efficiency, less latency

Cons: Reliability(stale data), Inconsistency

Uniform Interface

Resource identification in requests: Individual resources are identified in requests (URI).

Resource manipulation through representations: When a client holds a representation of a resource, it has enough information to modify or delete the resource.

Self-descriptive messages: Each message includes enough information to describe how to process the message.

Hypermedia as the engine of application state: a REST client should then be able to use server-provided links dynamically to discover all the available actions and resources it needs.

Layered System and Code-On-Demand

Layered System (hierarchical layers): Each component cannot "see" beyond the immediate layer.

Example: Legacy services, Legacy clients, New services, simplifying components by moving infrequently used functionality to a shared intermediary

Code-On-Demand allows client functionality to be extended by downloading and executing code in the form of applets or scripts.

Pros: Extensibility

Small Example

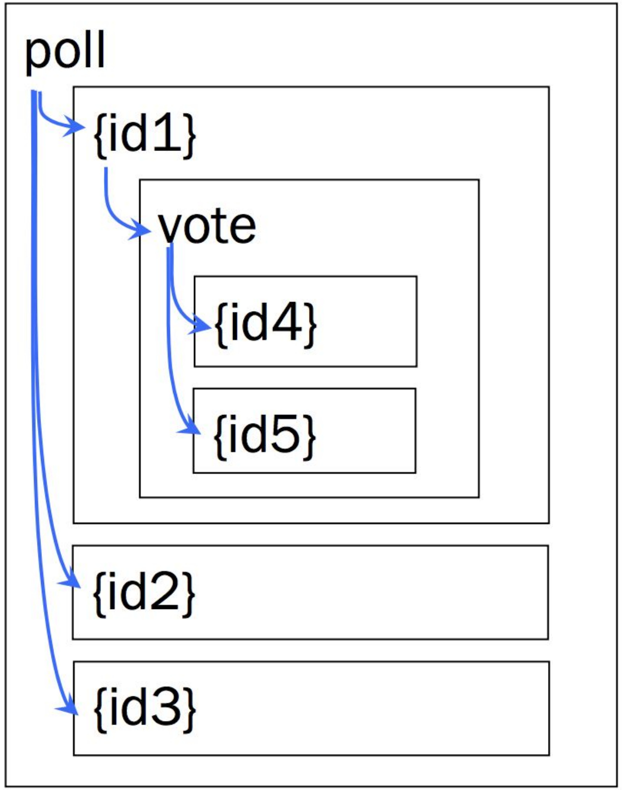
1. Resources: polls and votes

2. Containment Relationship

3. URIs embed IDs of “child” instance resources

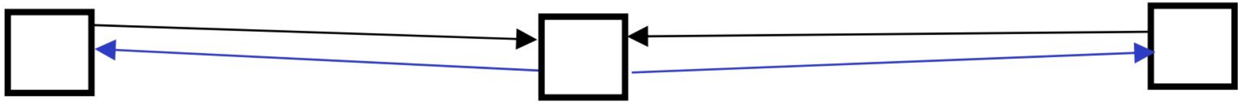
4. POST on the container is used to create child resources

5. PUT/DELETE for updating and removing child resources



Small Example

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

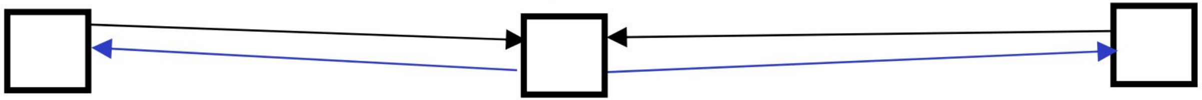


```
POST /poll/090331x/vote
<name>C. Pautasso</name>
<choice>B</choice>

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>
```

Small Example

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

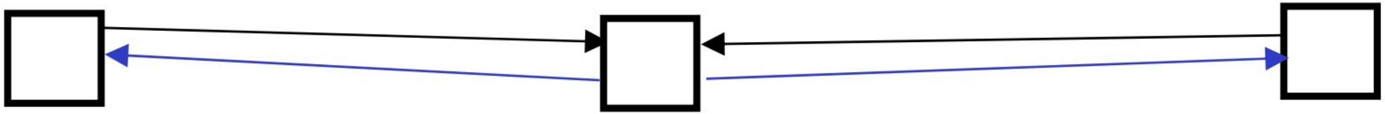


```
PUT /poll/090331x/vote/1
<name>C. Pautasso</name>
<choice>C</choice>

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>
```

Small Example

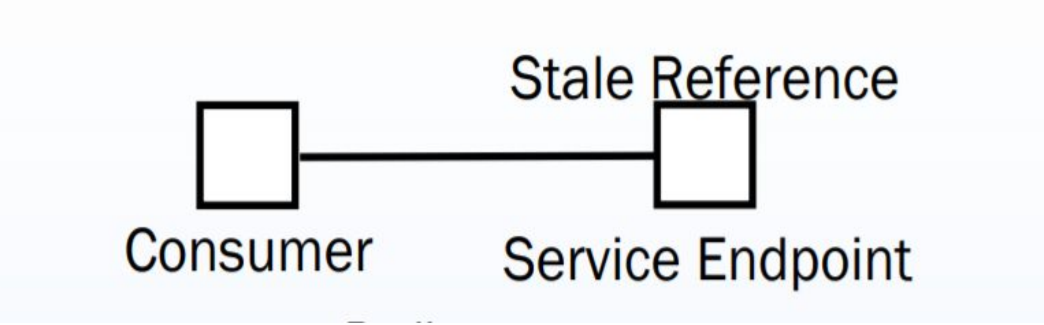
```
/poll
/poll/090331x
/poll/090331x/vote
/poll/090331x/vote/1
```



```
DELETE /poll/090331x

200 OK
```

```
GET /poll/090331x

404 Not Found
```

Endpoint Redirection



Problem:

- Service inventories may change overtime.
- Really difficult to replace references of old endpoints.

Solution:

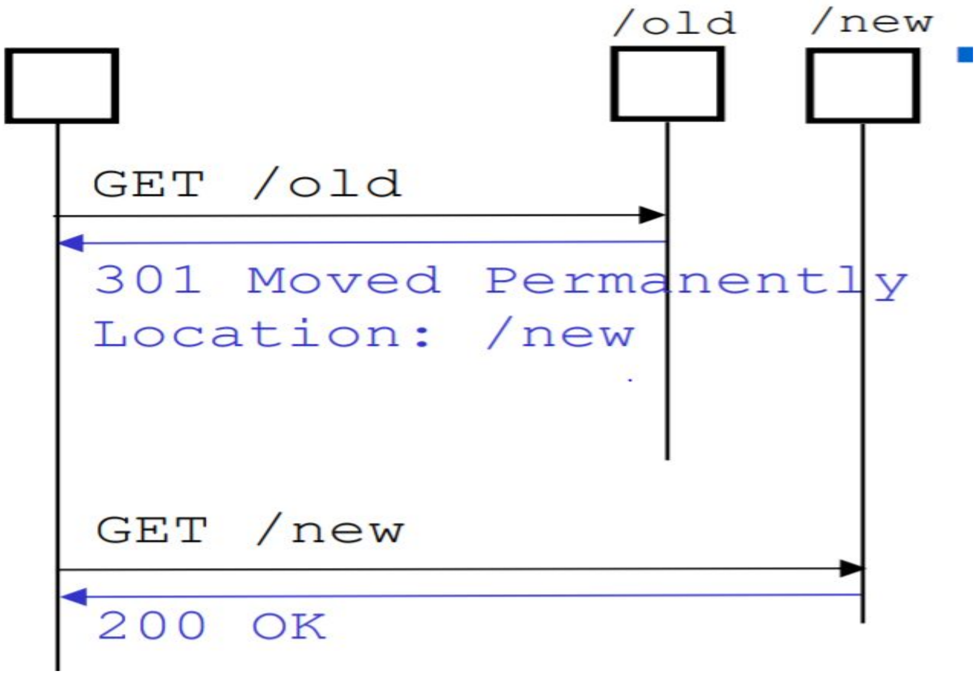
- Automatically redirect consumers when request to old consumer is made.

Endpoint Redirection

Example:

- 301- Moved Permanently
- 307-Temporary redirect

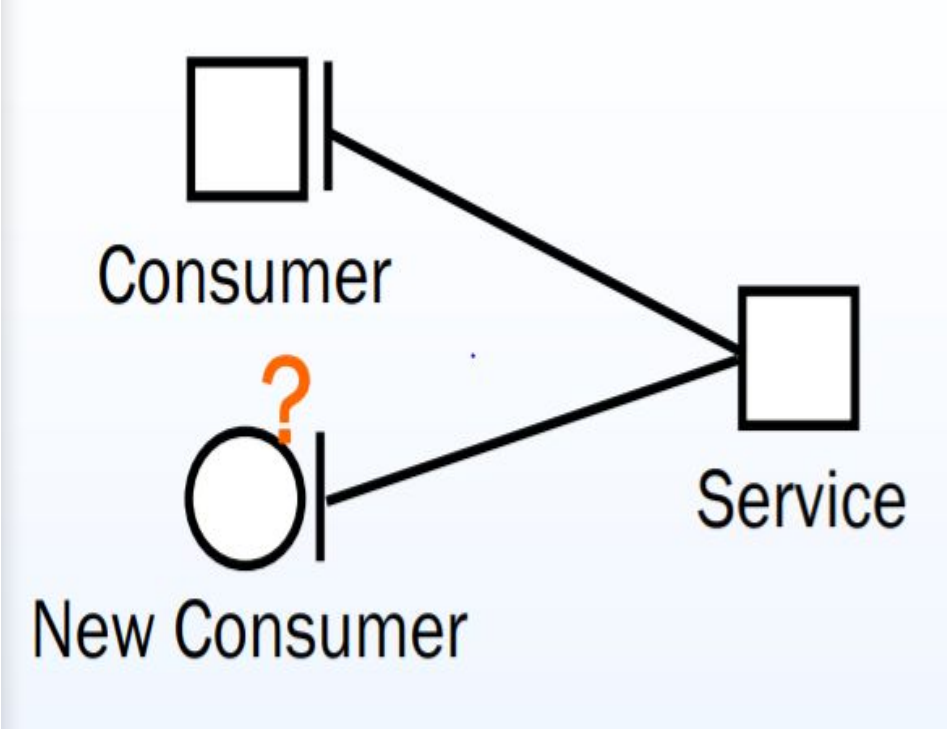
Note: Be cautious about redirection loops



Content Negotiation

Problem:

- Different consumers may accept different data format.
- Service contract may be changed frequently.
- New feature may be added to existing consumers.



Content Negotiation

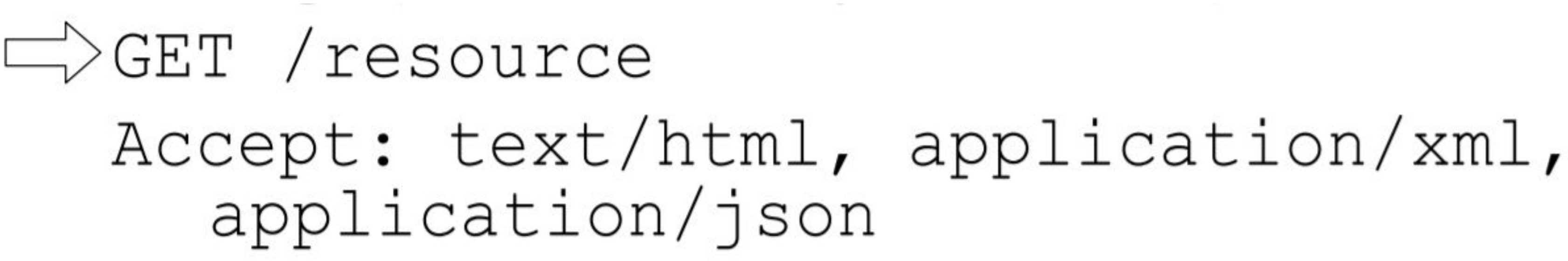
Solution:

- Include multiple standardized types in contract.
- Data format is negotiated at run time

Content Negotiation

Example

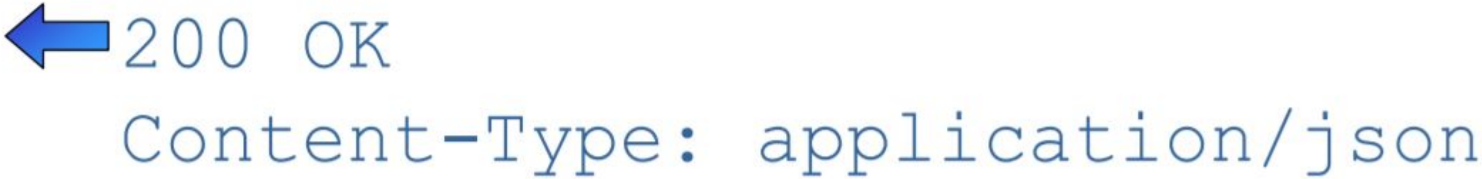
:Client’s request:



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

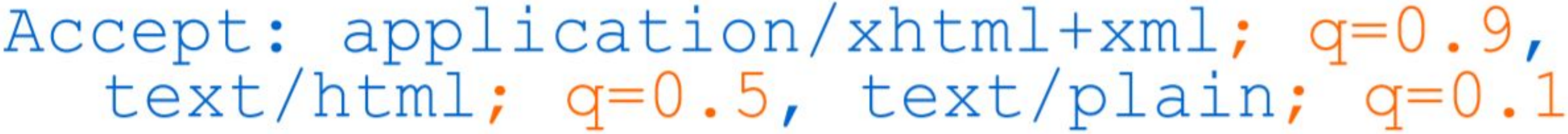
Content Negotiation

Response from server:



```
200 OK
Content-Type: application/json
```

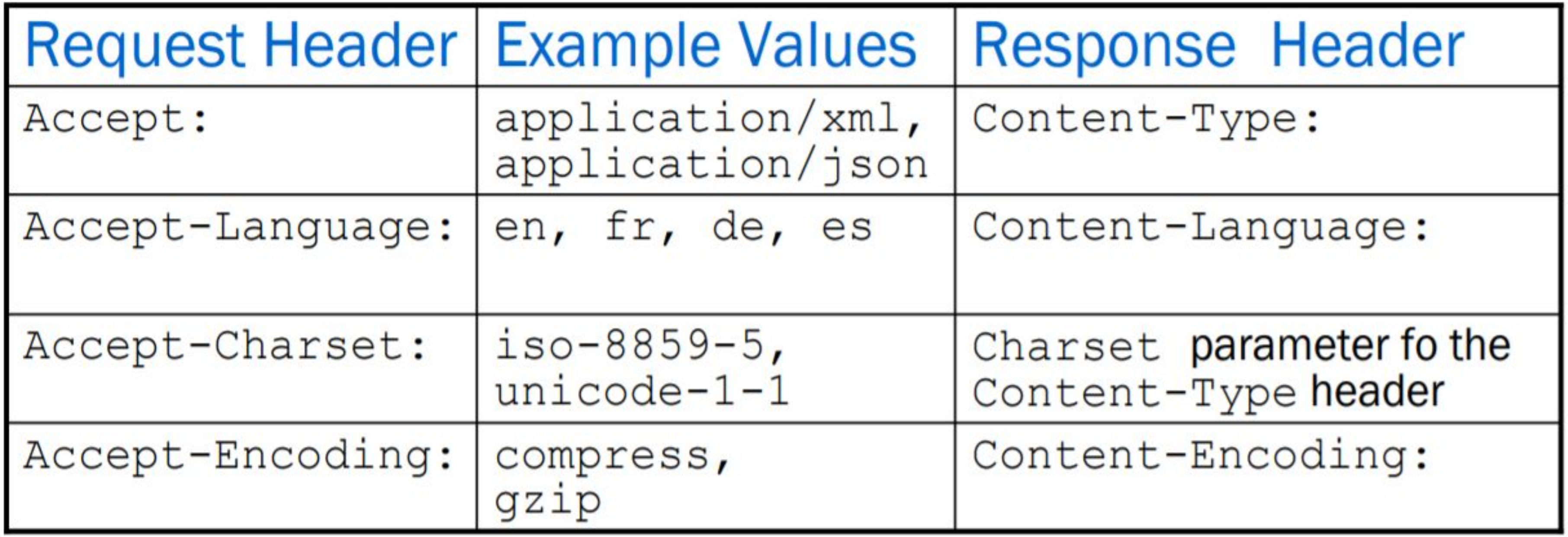
Advanced content negotiation:



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

Content Negotiation

Multi dimensional negotiation is also possible:



Request Header	Example Values	Response Header

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

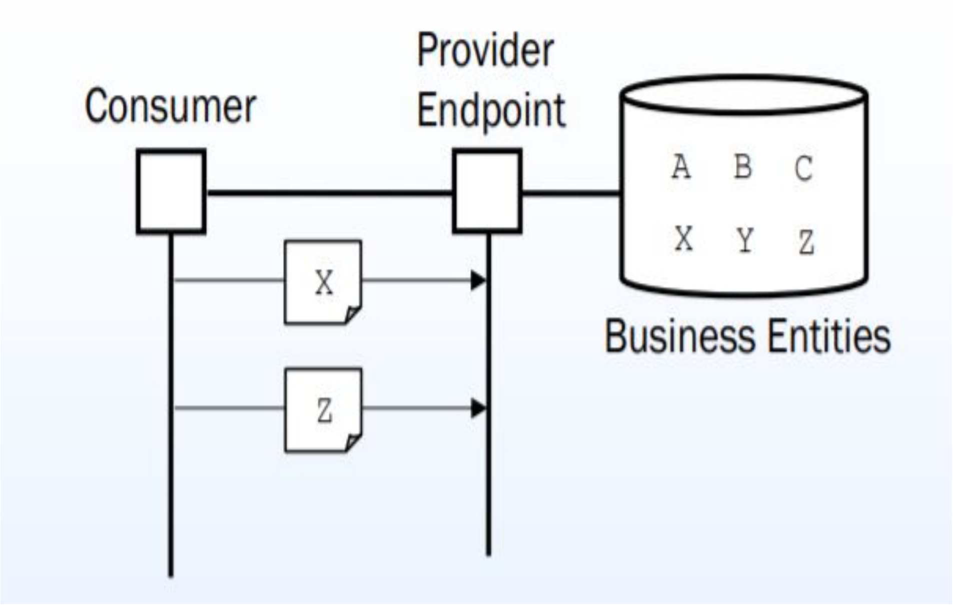
`Accept-Language:`	`en, fr, de, es`	`Content-Language:`

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

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

Entity Endpoint

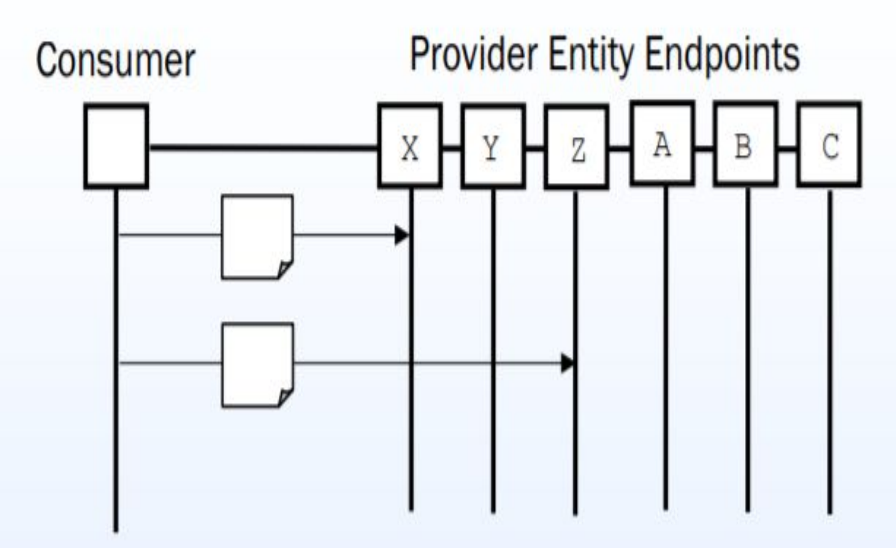
- Access to end points requires two identifiers.
- Entity identifier will vary from service to service.



Entity Endpoint

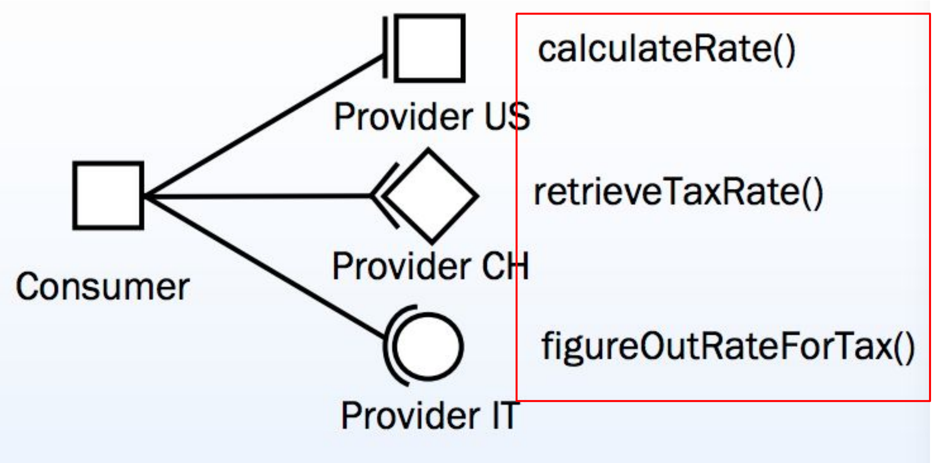
Solution:

- Expose each entity as individual lightweight endpoints of the service.
- Provides global addressability of entities



Pattern: Uniform Contract

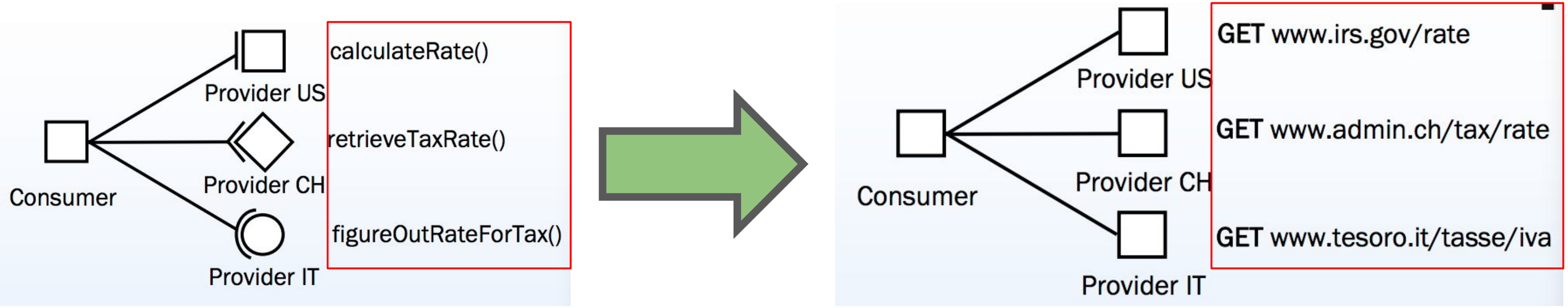
How can consumers take advantage of multiple evolving service endpoints?



Problem:

1. Accessing similar services requires consumers to access capabilities expressed in **service-specific** contracts.
2. **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.

Pros: 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

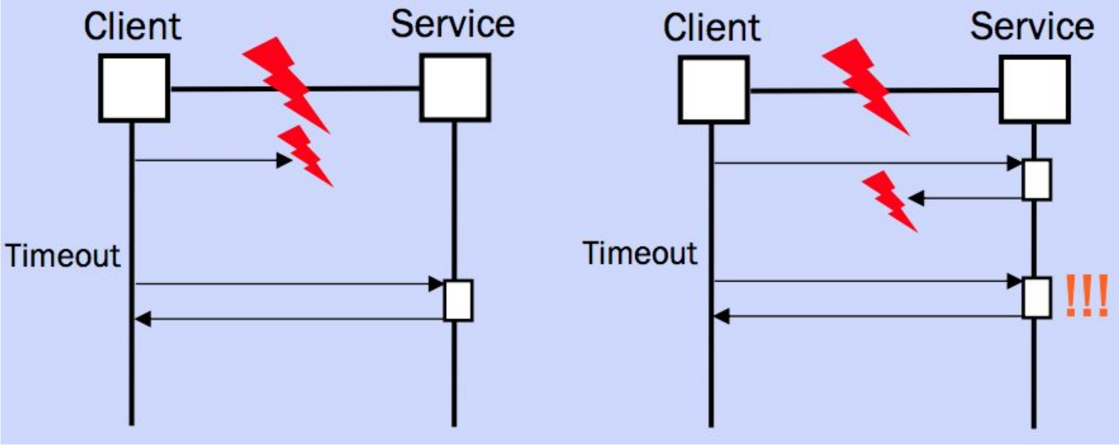
WHY?

Objective: an internet size network of REST services

Solution: have to enforce global concepts, like standards to make them understand each other.

Pattern: Idempotent Capability

How can a service consumer recover from Failures?



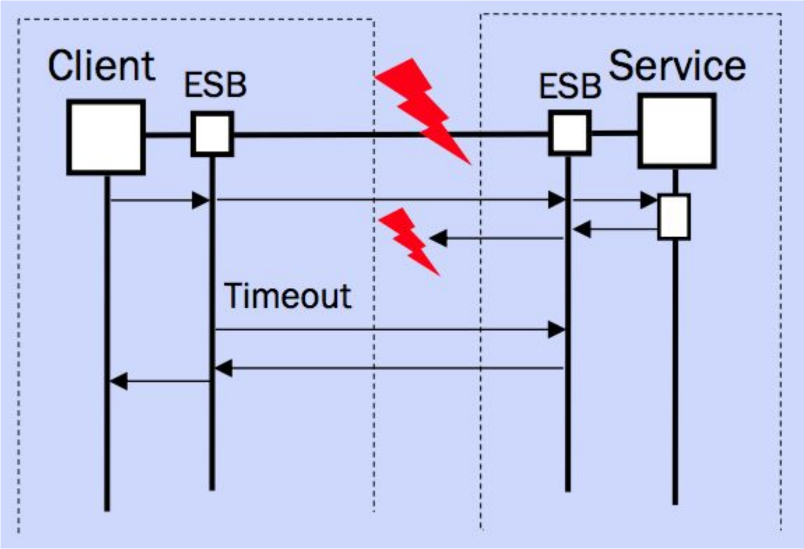
Problem:

1. Failures (such as the loss of messages) may occur during service capability invocation.
2. 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 (Enterprise Service Bus), with support for reliable messaging.

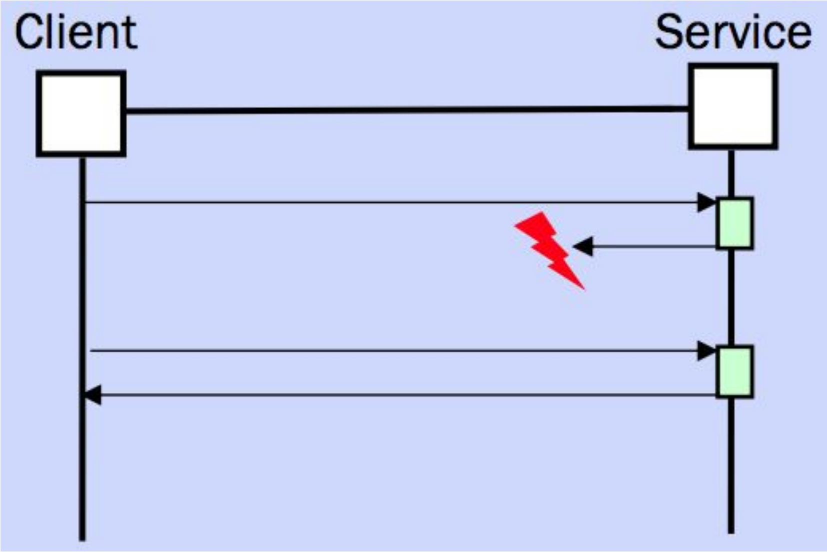
Problem: do we always need this? Are there some messages more critical than others?



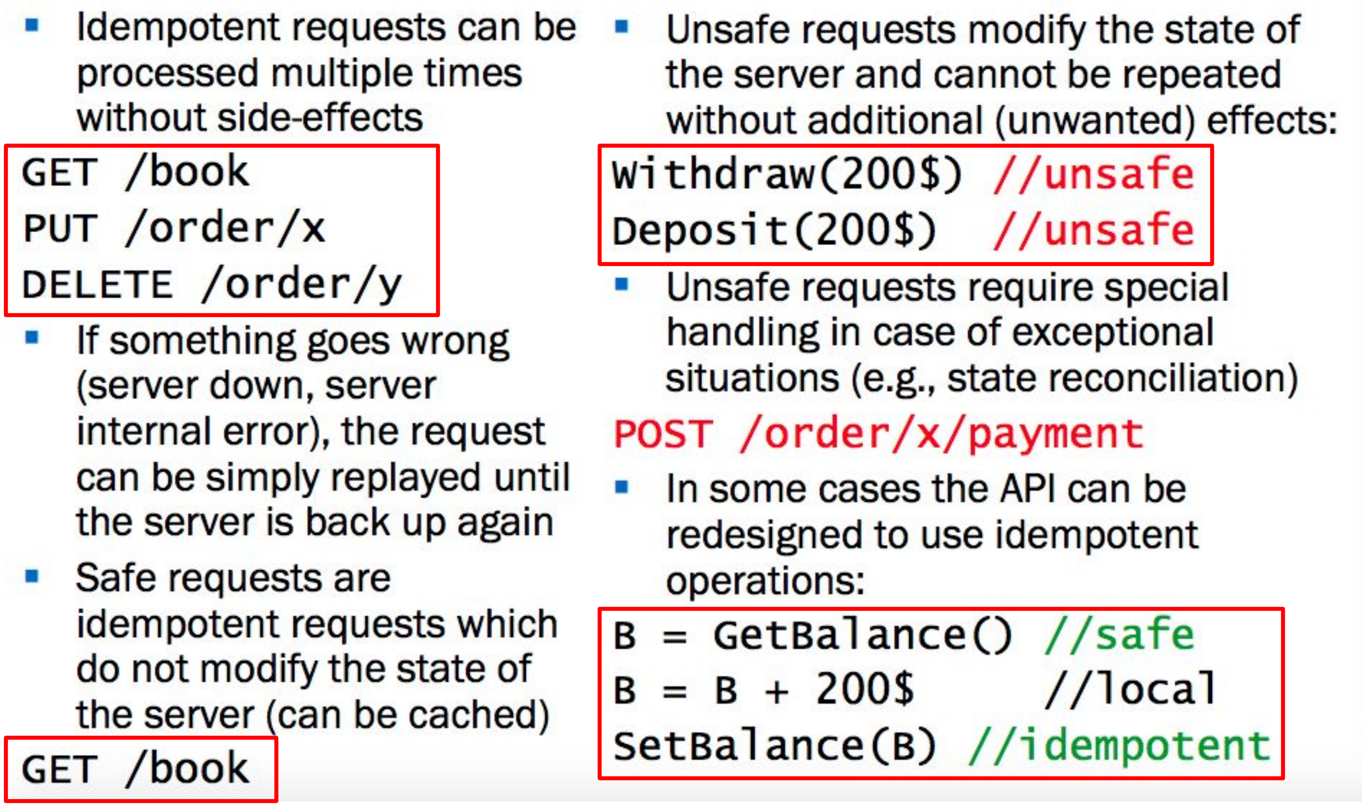
Pattern: Idempotent Capability

An **idempotent** method means that the result of a successful performed request is independent of the number of times it is executed.

Simpler Solution: use idempotent service capabilities to 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
```

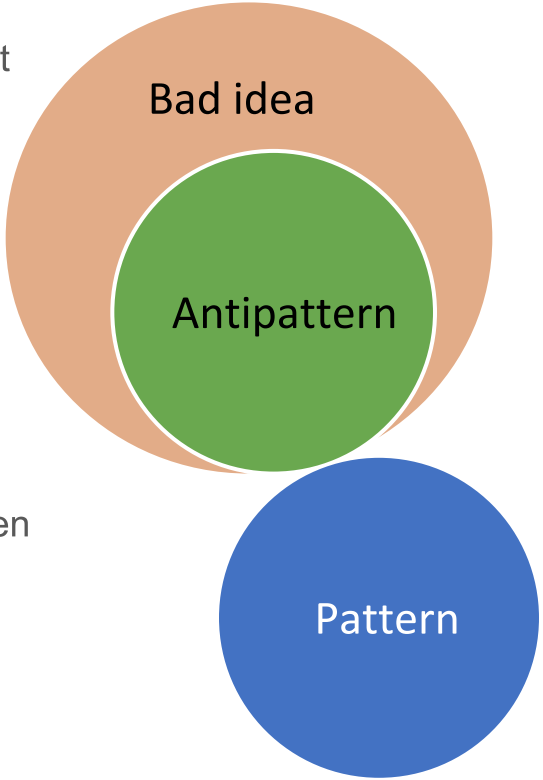
Antipatterns

- An **anti-pattern** is a **common response** to a recurring problem that is usually **ineffective** and **risks** being highly **counterproductive**.

- there must be at least **two key elements** present to formally distinguish an actual anti-pattern from a simple bad habit, bad practice, or bad idea:

1. A commonly used process, structure, or pattern of action that despite initially appearing to be an appropriate and effective response to a problem, has more bad consequences than good ones.

2. Another solution exists that is documented, repeatable, and proven to be effective.



Tunneling everything through GET

- 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)

Tunneling everything through POST

- 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)

Demo

1. A Nodejs Project
2. Google Calendar API