

# Unit 5: Design of Web Services (1/2)

---

Introduction to Web Services. RESTful Web Services

# Design of Web Services

---

- ▶ Introduction to Web Services
- ▶ RESTful Web Services
- ▶ SOAP or Big Web Services

# Design of Web Services: Definition

---

- ▶ A Web service is a programmatically available application logic exposed in a well-defined manner over standard Web protocols
- ▶ According to W3C Web services provide a standard means of interoperating between different software applications, running on a variety of platforms and/or frameworks



# Design of Web Services: Features

---

- ▶ Reusability
  - ◆ Turning developed functionality into new software (web service)
- ▶ Usability
  - ◆ Applications have the freedom to chose the web services that they need
- ▶ Interoperability
  - ◆ Ability to work with each other without need to know the details of how they each work
- ▶ Loosely coupled
  - ◆ Each service exists independently of the other services
- ▶ Easy of integration
  - ◆ Web Services act as glue between isolated applications

# Types of Web Services

---

## RESTful Services

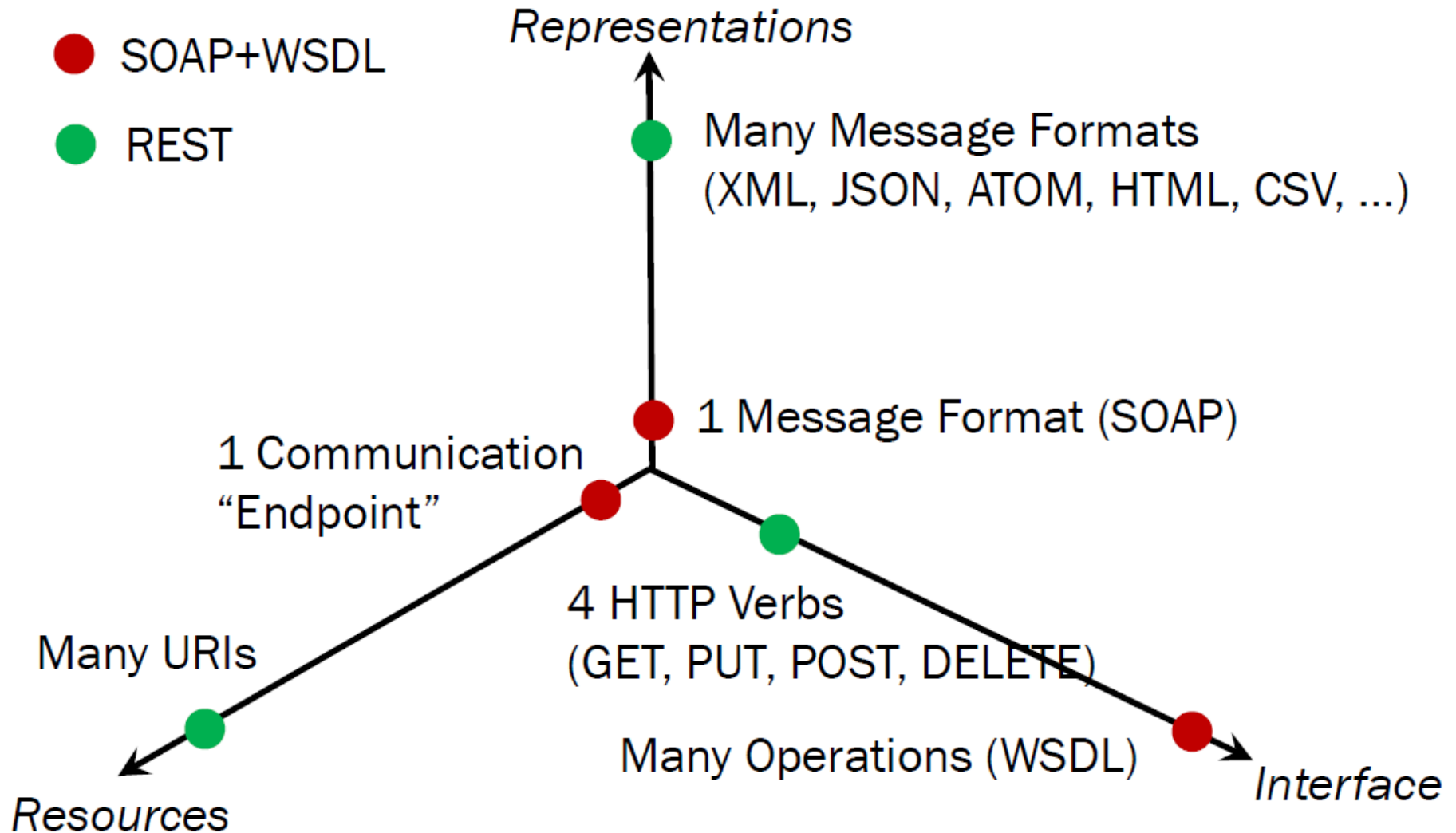
- ▶ Use the REpresentational State Transfer (REST), a stateless client-server architecture in which services are viewed as resources and can be identified by their URIs
- ▶ Exchange representations of resources by using a standardized interface and protocol
- ▶ Also known as REST Services

## SOAP or Big Services

- ▶ Sometimes defined in Service Oriented Architectures (SOA) as business logic tied to business processes
- ▶ SOAP is a standard protocol specification for message exchange (between the client and the service) based on XML
- ▶ Also known as Web Services, SOAP-based Web Services, WS-\*Services, Big Services

# SOAP+WSDL vs. RESTful

---



# Design of Web Services

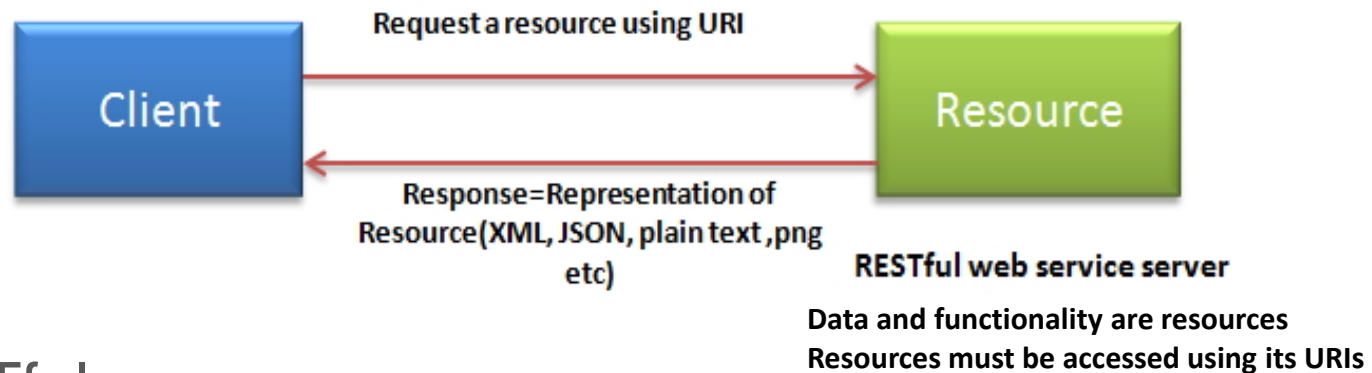
---

- ▶ Introduction Web Services
- ▶ RESTful Web Services
  - ◆ Concepts & Architectural Principles
  - ◆ Designing RESTful Web Services
    - ❖ Creating the Service API
    - ❖ Interaction between Clients and Services
  - ◆ Example
- ▶ SOAP or Big Web Services

# RESTful Web Services: Concepts

---

- ▶ REST (REpresentational State Transfer)
  - ◆ A stateless client-server architecture in which the web services are viewed as resources and can be identified by their URIs
  - ◆ REST isn't protocol specific, but when people talk about REST they usually mean REST over HTTP



- ▶ RESTful
  - ◆ Typically used to refer to web services implementing REST architecture



# RESTful Web Services: Architectural Principles (1)

---

- ▶ Addressability
  - ◆ Each resource have its own specific URI
- ▶ Uniform interface
  - ◆ Resources are manipulated using the four basic methods provided by HTTP: PUT, GET, POST, and DELETE
- ▶ Client-Server
  - ◆ Separating concerns between the client and service helps to improve portability in the client and scalability of the service
- ▶ Stateless
  - ◆ Each HTTP request happens in complete isolation. The server never relies on information from previous request

# RESTful Web Services: Architectural Principles (2)

---

- ▶ Cacheable responses
  - ◆ Responses must be capable of being cached by intermediaries
- ▶ Interconnected resource representations
  - ◆ Representations of the resources are interconnected using URLs, thereby enabling a client to progress from one state to another
- ▶ Layered components
  - ◆ Intermediaries, such as proxy servers, cache servers, etc, can be inserted between clients and resources to support performance, security, etc
- ▶ Self-descriptive messages
  - ◆ Resources are decoupled from their representation so that their content can be accessed in a variety of formats

# Design of Web Services

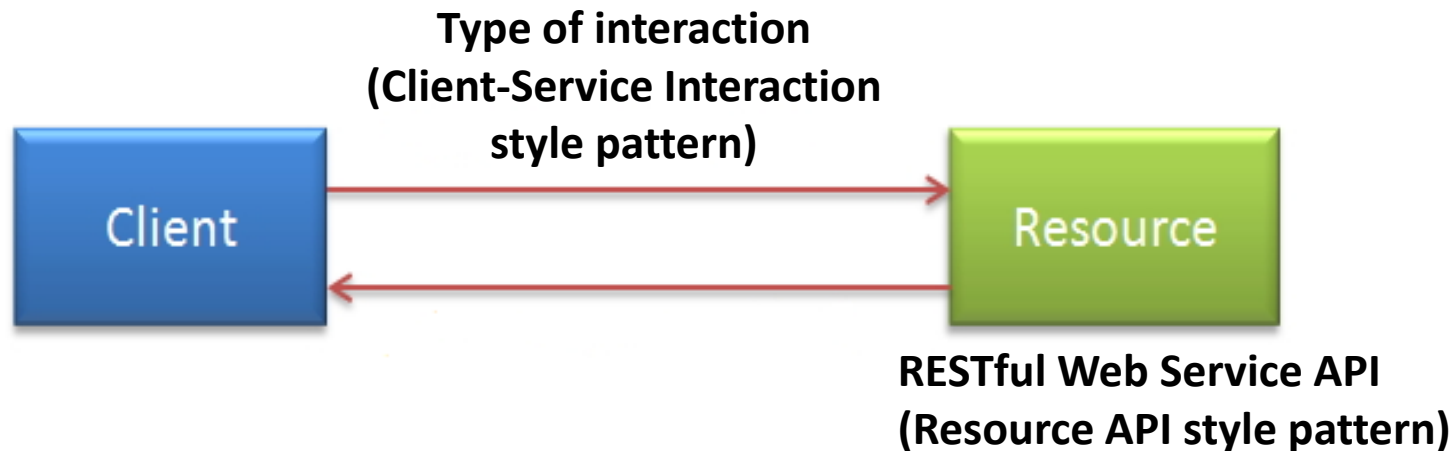
---

- ▶ Definition and Features of Web Services
- ▶ Types of Web Services
- ▶ **RESTful Web Services**
  - ◆ Concepts & Architectural Principles
  - ◆ **Designing RESTful Web Services**
    - ❖ **Creating the Service API**
    - ❖ **Interaction between Clients and Services**
  - ◆ Example
- ▶ SOAP or Big Web Services

# RESTful Web Services: Designing RESTful WS

---

- ▶ To design a RESTful Web Service we have to:
  - ◆ Create a Service API
    - ❖ Resource API style pattern
  - ◆ Define the type of interaction between the client and the service
    - ❖ Client-Service Interaction style pattern



# RESTful Web Services: Creating a Resource API

---

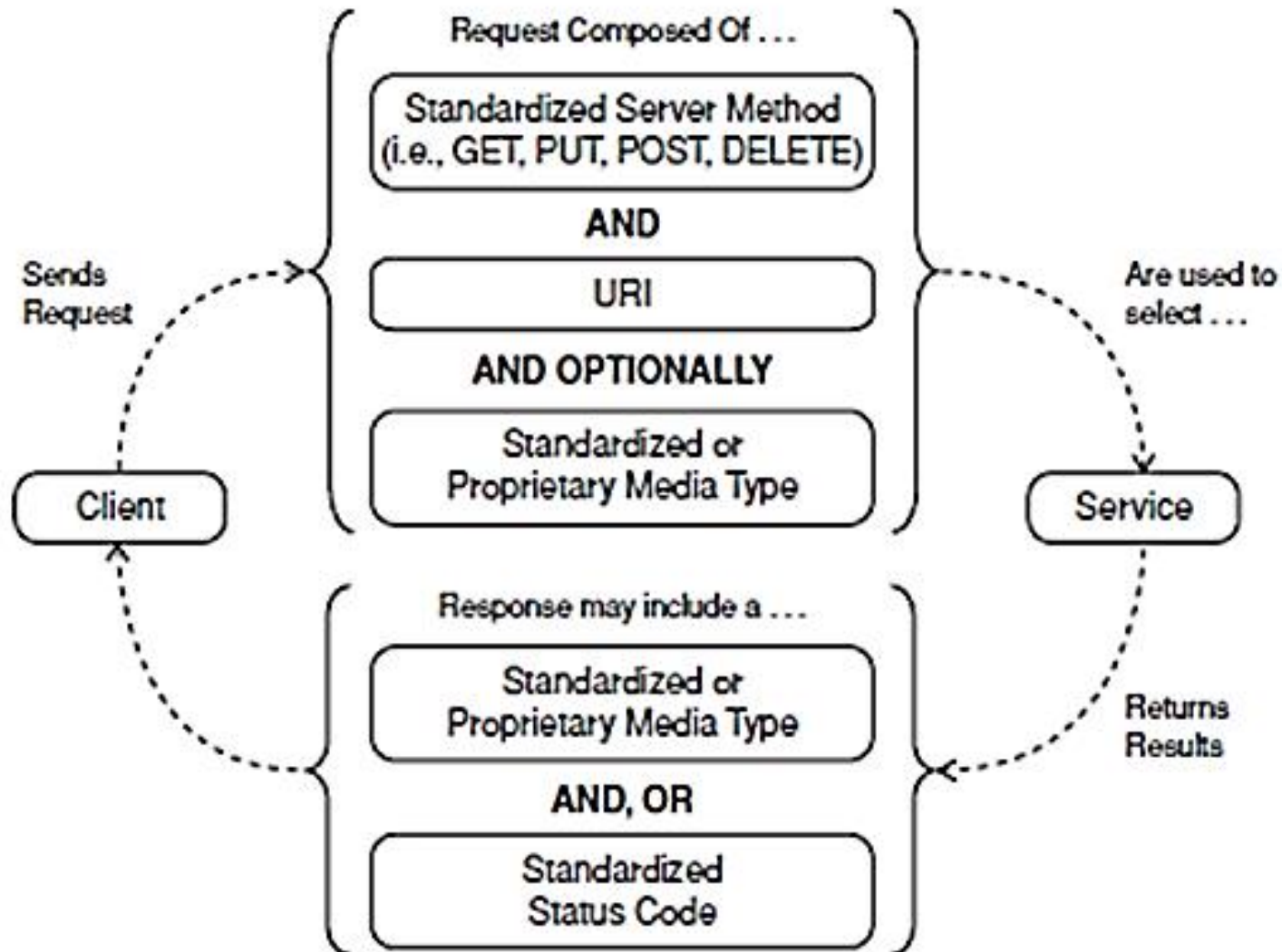
Pattern Name	Problem	Description	WS Tech.
<b><i>RPC API</i></b>	How can clients execute remote procedures over HTTP?	Define messages that identify the remote procedures to execute and also include a fixed set of elements that map directly into the parameters of remote procedures	WS-*
<b><i>Message API</i></b>	How can clients send commands, notifications, or other information to remote systems over HTTP while avoiding direct coupling to remote procedures?	Define messages that are not derived from the signatures of remote procedures. These messages may carry information on specific topics, tasks to execute, and events	WS-*
<b><i>Resource API</i></b>	How can a client manipulate data managed by a remote system, avoid direct coupling to remote procedures, and minimize the need for domain-specific APIs?	Assign all procedures, instances of domain data, and files a URI. Leverage HTTP as a complete application protocol to define standard service behaviors	REST

# RESTful Web Services: Creating a Resource API

---

- ▶ Services that follow *Resource API* pattern use the requested URI and HTTP methods issued by the client along with the submitted or requested media type to determine the client's intent
- ▶ *Resource API* provides access to **resources** through their **representations**
- ▶ Resources may be accessed by the basic **HTTP methods**
- ▶ A resource may be a text file, a media file, a specific row in a database table, a downloadable program
- ▶ Representations capture the current state of a resource
- ▶ Clients manipulate the state of resources through representations (XHTML, JSON, XML, etc...).

# RESTful Web Services: Creating a Resource API



\* Extracted from: Robert Daigneau. Service Design Patterns. Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services. Addison Wesley, 2012

## Creating a Resource API: A Blueprint for designing Resource APIs

---

1. Identify resources to be exposed as services
2. Define “nice” URIs to address the resources
3. Understand what it means to do a GET, POST, PUT, DELETE for each resource (and whether it is allowed or not)
4. Design and document resource representations
5. Consider error conditions: use status codes meaningfully

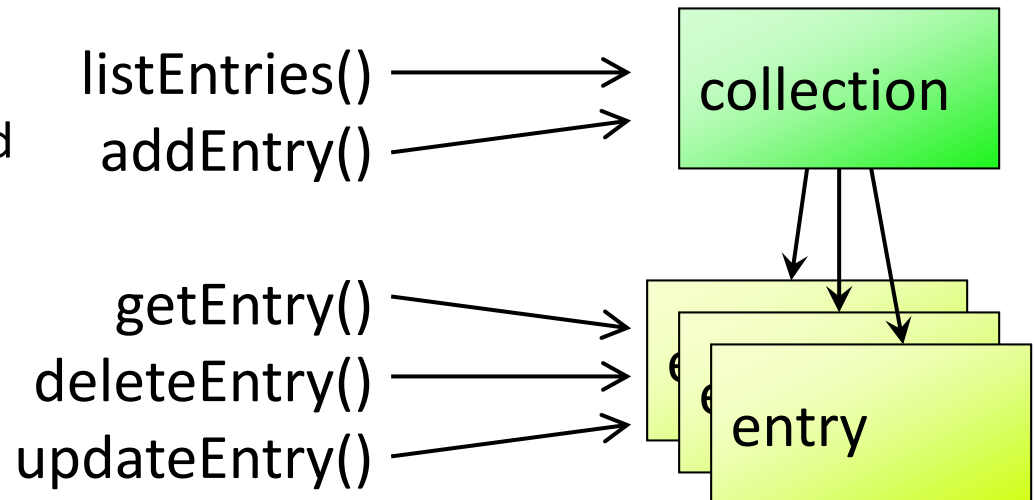


# Creating a Resource API: Identify resources to be exposed

---

- ▶ A **resource** is anything that is important enough to be referenced as a thing itself
- ▶ Usually we have 2 levels of abstraction:
  - ◆ A *collection*: is a resource representing a whole class and/or set of individuals
  - ◆ An *entry*: is a resource representing an instance/individual within a class/set

Operations can be distributed onto the resources (the collection, each entry) and mapped to a small uniform set of operations

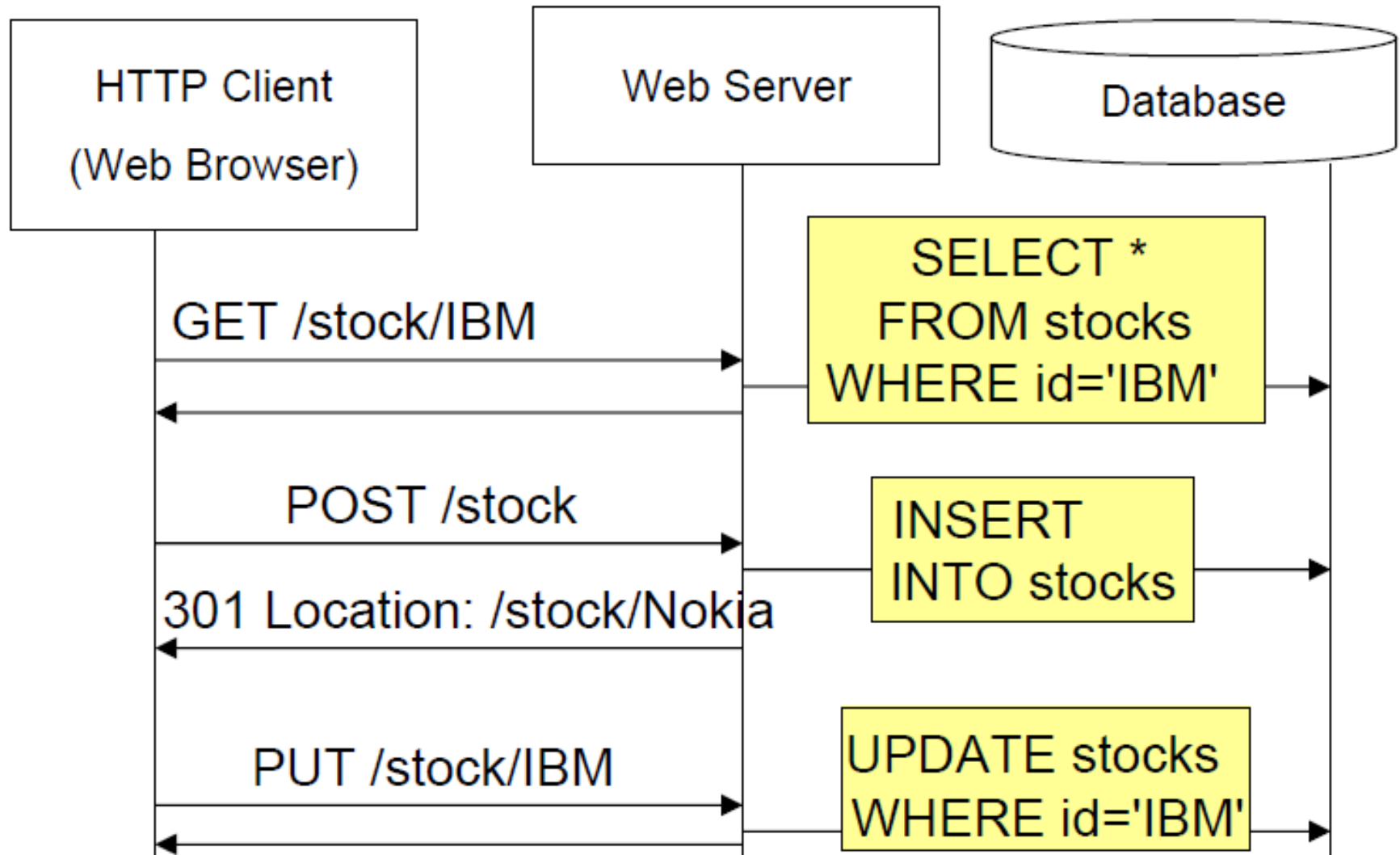


# Creating a Resource API: Name resources with URIs

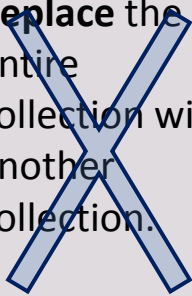

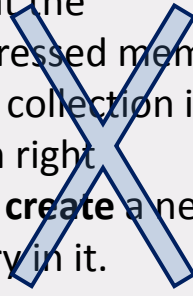
---

- ▶ Only two base URIs per resource:
  - ◆ Collection: `/stocks` (plural noun)
  - ◆ Entry: `/stocks/:stock_id` (e.g. `/stocks/IBM` )
- ▶ Query with specific attributes:
  - ◆ `/dogs?color=red&state=running&location=park`
- ▶ Versioning:
  - ◆ `/v1/stocks`
- ▶ Paging:
  - ◆ `/stocks?limit=25&offset=50`
- ▶ Non a resource response (e.g. calculate, convert, ...):
  - ◆ `/convert?from=EUR&to=CNY&amount=100` (verbs, not nouns)

# Creating a Resource API: Operations (Example)



# Creating a Resource API: Operations (Example)

Resource	GET	PUT	POST	DELETE
<b>http://www.stock.org/stocks</b>	<b>List</b> the members (URIs and perhaps other details) of the collection. For example list all the stocks.	<b>Replace</b> the entire collection with another collection. 	<b>Create</b> a new entry in the collection. The new entry's ID is assigned automatically and is usually returned by the operation.	<b>Delete</b> the entire collection. 
<b>http://www.stock.org/stocks/IBM</b>	<b>Retrieve</b> a representation of the addressed member of the collection, expressed in an appropriate Internet media type.	<b>Update</b> the addressed member of the collection, or if it doesn't exist, <b>create</b> it.	Treat the addressed member as a collection in its own right and <b>create</b> a new entry in it. 	<b>Delete</b> the addressed member of the collection.

# Creating a Resource API: Safeness & Idempotence

---

- ▶ Method GET is **safe**, which means it is intended only for information retrieval and should not change the state of the server. In other words, it should not have side effects, beyond relatively harmless effects such as logging, caching, etc.
- ▶ Methods PUT and DELETE are defined to be **idempotent**, meaning that multiple identical requests should have the same effect as a single request.
- ▶ Method GET, being prescribed as safe, should also be idempotent, as HTTP is a stateless protocol.
- ▶ Method POST is not necessarily idempotent, and therefore sending an identical POST request multiple times may further affect state or cause further side effects

## Creating a Resource API: Representation accepted from Client

---

- ▶ Simple representations are usually key-value pairs: set this item of resource state to that value:  
`username=leonard`
  - ◆ There are lots of representations for key-value pairs, form-encoding being the most popular
- ▶ Complex representations should be in the same format it uses to serve outgoing representations (served to the client)

## Creating a Resource API: Representation served to Client

---

- ▶ Representations should be human-readable, but computer-oriented
- ▶ Representations should be useful, they should expose interesting data.
  - ◆ A single representation should contain all relevant information necessary to fulfil a need
  - ◆ A client should not have to get several representations of the same resource to perform a single operation
- ▶ Usually JSON and XML formats are used

# Creating a Resource API: Server response

---

- ▶ Make sure that clients requests are correctly turned into responses
  - ◆ Response code
    - ❖ Examples: **200** (“OK”), **201** (“Created”)
  - ◆ Some HTTP headers
    - ❖ Example: **Content-Type** is **image/png** for a map image
  - ◆ Representation
- ▶ A set of headers should be considered to save client and server time and bandwidth
  - ◆ Conditional HTTP GET for requesting resources that are frequently requested and invariable (**Last-Modified** and **If-Modified-Since** headers)



# Creating a Resource API: Errors

---

- ▶ Make sure that clients receive clear information in case of error
  - ◆ Response code
    - ❖ Examples: **3xx**, **4xx**, **5xx**
  - ◆ Some HTTP headers
  - ◆ A document describing an error condition

# Creating a Resource API: Errors

---

## Learn to use HTTP Standard Status Codes

100 Continue  
200 OK  
201 Created  
202 Accepted  
203 Non-Authoritative  
204 No Content  
205 Reset Content  
206 Partial Content  
300 Multiple Choices  
301 Moved Permanently  
302 Found  
303 See Other  
304 Not Modified  
305 Use Proxy  
307 Temporary Redirect

4xx Client's fault

400 Bad Request  
401 Unauthorized  
402 Payment Required  
403 Forbidden  
404 Not Found  
405 Method Not Allowed  
406 Not Acceptable  
407 Proxy Authentication Required  
408 Request Timeout  
409 Conflict  
410 Gone  
411 Length Required  
412 Precondition Failed  
413 Request Entity Too Large  
414 Request-URI Too Long  
415 Unsupported Media Type  
416 Requested Range Not Satisfiable  
417 Expectation Failed

500 Internal Server Error  
501 Not Implemented  
502 Bad Gateway  
503 Service Unavailable  
504 Gateway Timeout  
505 HTTP Version Not Supported

5xx Server's fault

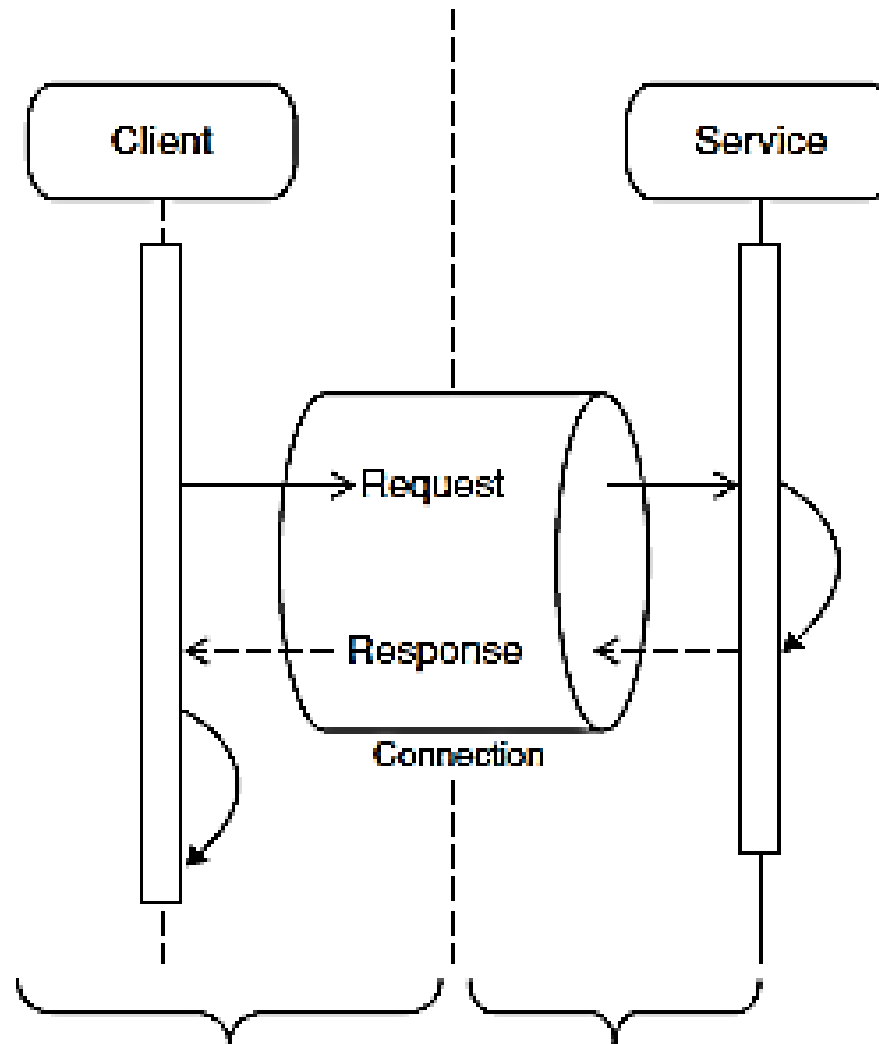
# RESTful Web Services: Defining the interaction

---

Pattern Name	Problem	Description
<b><i>Request/Response</i></b>	What's the simplest way for a web service to process a request and provide a result?	Process requests when they're received and return results over the same client connection.
<b><i>Request/Acknowledge</i></b>	How can a web service safeguard systems from spikes in request load and ensure that requests are processed even when the underlying systems are unavailable?	When a service receives a request, forward it to a background process, then return an acknowledgment containing a unique request identifier.
<b><i>Media Type Negotiation</i></b>	How can a web service provide multiple representations of the same logical resource while minimizing the number of distinct URIs for that resource?	Allow clients to indicate one or more media type preferences in HTTP request headers. Send requests to services capable of producing responses in the desired format.
<b><i>Linked Service</i></b>	Once a service has processed a request, how can a client discover the related services that may be called, and also be insulated from changing service locations and URI patterns?	Only publish the addresses of a few root web services. Include the addresses of related services in each response. Let clients parse responses to discover subsequent service URIs.

# Defining the interaction: Request/Response Pattern

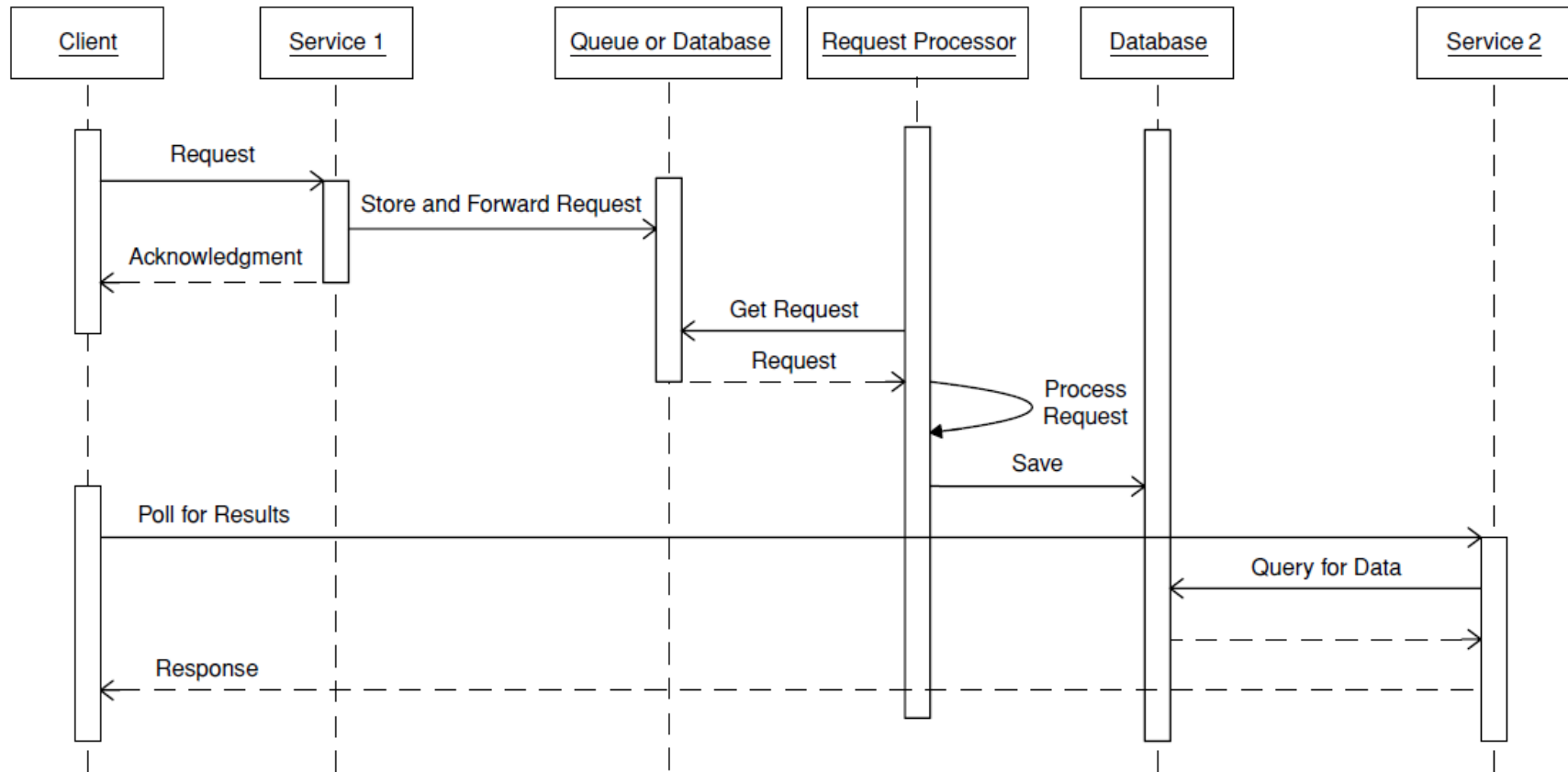
---



\* Extracted from: Robert Daigneau. Service Design Patterns. Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services. Addison Wesley, 2012

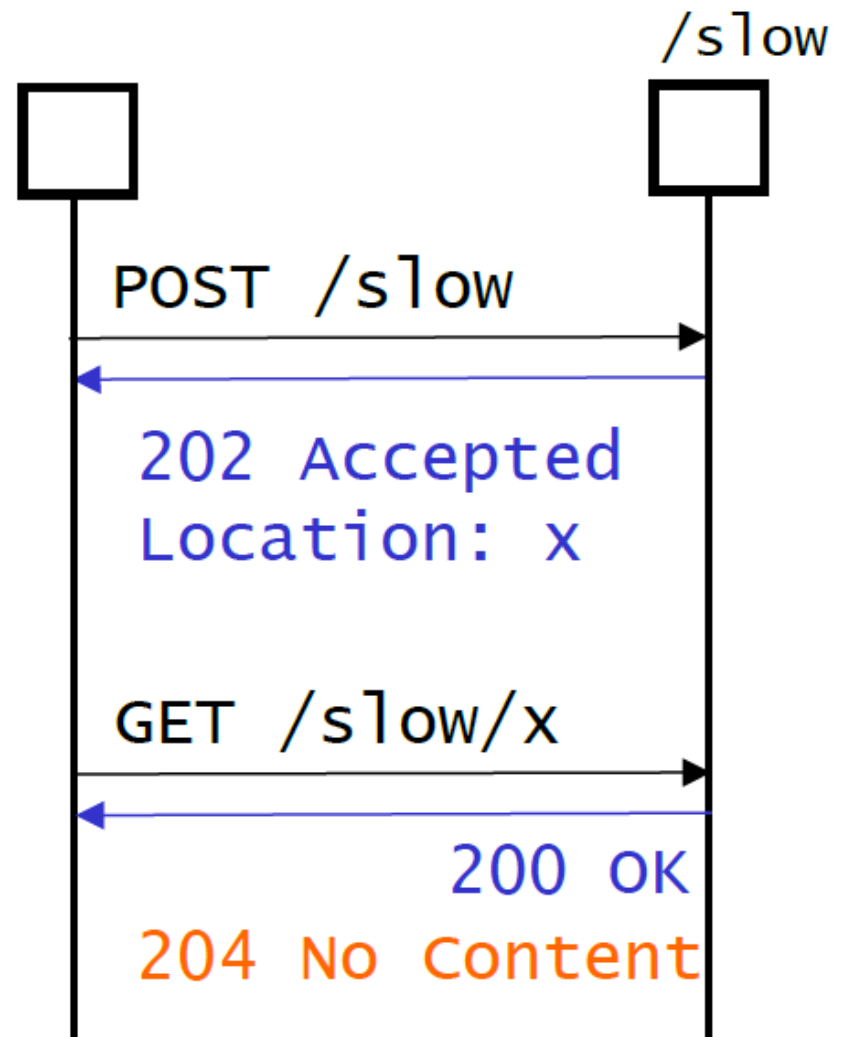
# Defining the interaction: Request/Acknowledge Pattern

## Poll variation



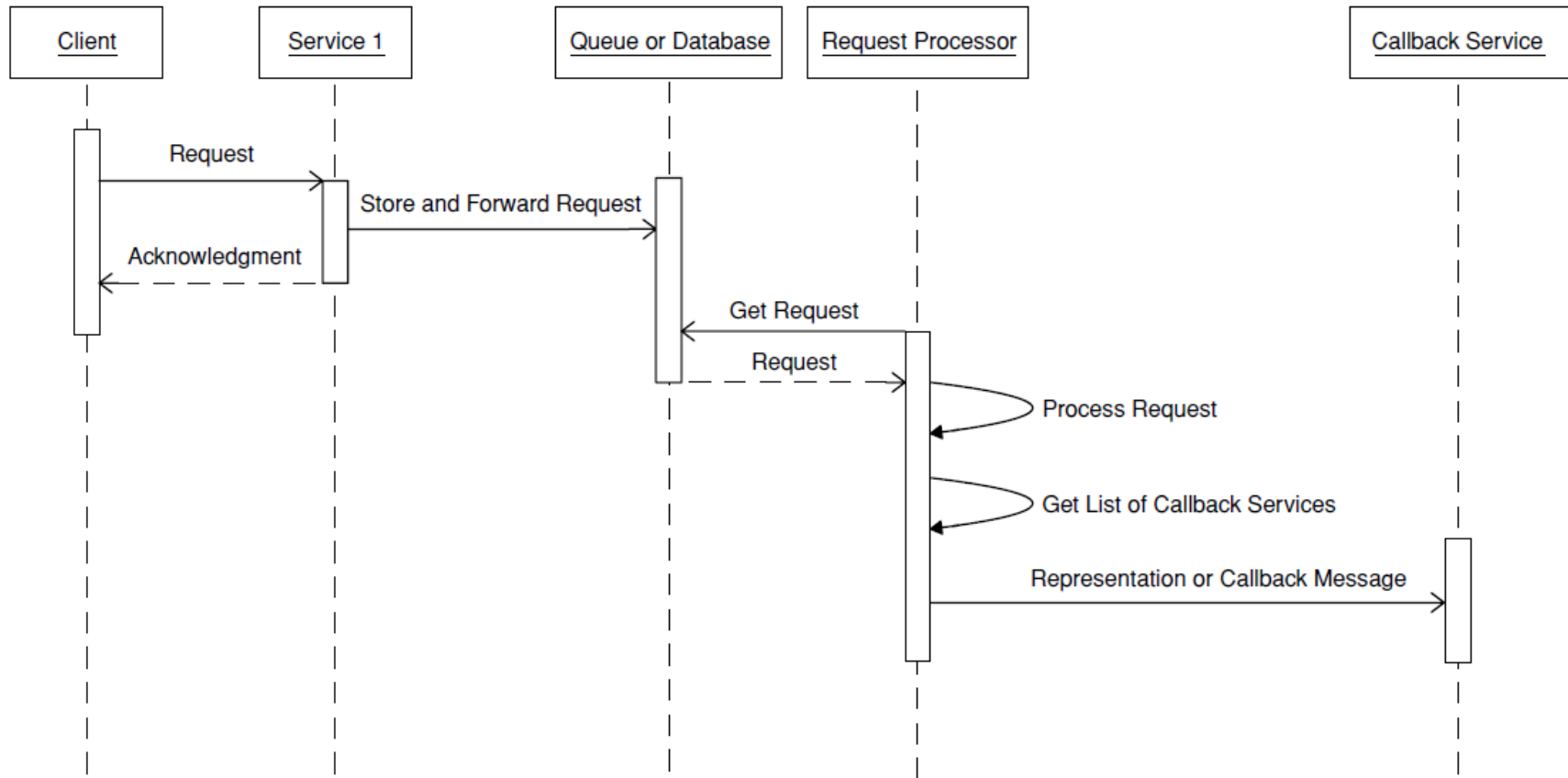
## Defining the interaction: Request/Acknowledge Pattern

- ▶ 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



# Defining the interaction: Request/Acknowledge Pattern

## Callback variation



# Defining the interaction: Media Type Negotiation

---

- ▶ Web services that are used by large and diverse client populations must often accommodate different media type preferences.
- ▶ There are many ways for a client to convey its preferences
  - ◆ Forced Media Type Negotiation
  - ◆ Media Type Negotiation
  - ◆ Advanced Media Type Negotiation



## Defining the interaction: Forced Media Type Negotiation Pattern

---

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?
- ▶ A URI should be used to represent distinct resources (not distinct formats of the same resources).

## Defining the interaction: Media Type Negotiation Pattern

---

- ▶ Negotiating the message format does not require to send more messages
- ▶ The client lists the set of understood formats (MIME types)
- ▶ The server chooses the most appropriate one for the reply (status 406 if none can be found)

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

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

## Defining the interaction: Advanced Media Type Negotiation Pat.

---

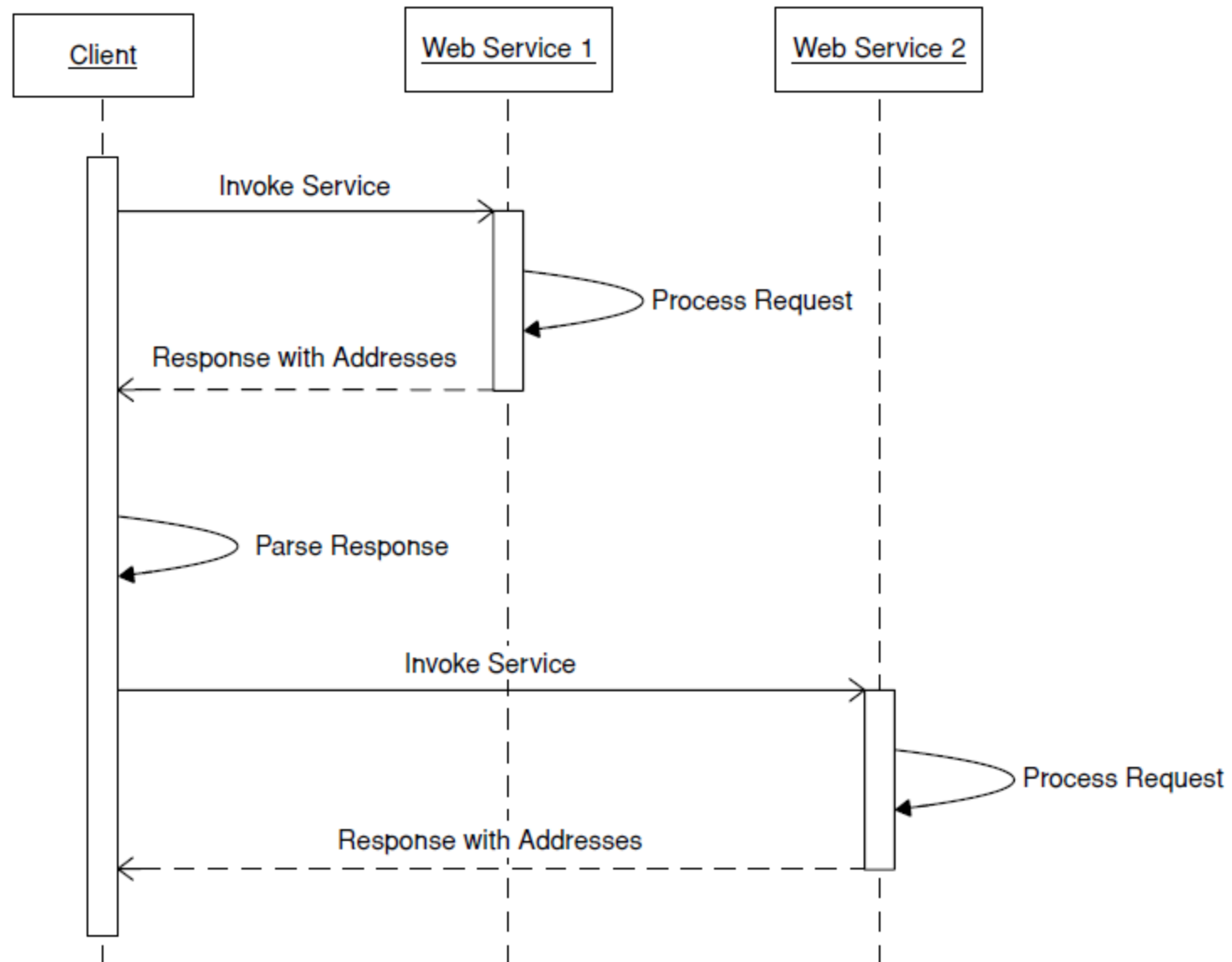
- ▶ Quality factors allow the client to indicate the relative degree of preference for each representation. If  $q=0$ , then content with this parameter is not acceptable for the client.
- ▶ **Ex1:** The client prefers to receive HTML (but any other text format will do with lower priority)
- ▶ **Ex2:** The client prefers to receive XHTML, or HTML if this is not available and will use Plain Text as a fallback

Media/Type;  $q=X$

Accept: text/html,  
text/\*;  $q=0.1$

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

# Defining the interaction: Linked Service Pattern



# Defining the interaction: Linked Service Pattern

---

- ▶ Model relationships (e.g., containment, reference, etc) between resources with hyperlinks in their representations that can be followed to get more details
- ▶ **H**ypertext **A**s **T**he **E**ngine **O**f **A**pplication **S**tate (HATEOAS): representations may contain links to potential next application states, including direction on how to transition to those states when a transition is selected

# Defining the interaction: HATEOAS Example

---

GET /tweets/391678195147079681

Accept: application/json

200 OK

Content-Type: application/json

```
{"created_at": "Sat Oct 19 21:32:13 +0000 2013",  
  "id": 391678195147079700,  
  "text": "bla, bla, bla",  
  "actions": [  
    { "action": "Retweet",  
      "method": "post",  
      "href": "/tweets/391678195147079681/retweets" } , ...  
  ], ... }
```

## Defining the interaction: Linked Service Pattern Benefits

---

- ▶ Provides relevant links for each request
  - ◆ Responses may be constrained to only provide service links that make sense given the context of the most recent request.
- ▶ Ensures correctly formatted links
- ▶ Protects clients from changing URI patterns and service locations
  - ◆ Clients can trust that each response contains only the most current URIs for a set of related services.
  - ◆ This makes it easy for service owners to change their URI patterns, or even move a service to an entirely different domain without breaking clients.

# Richardson Maturity Model

---

## ▶ **Level 0: *HTTP Tunneling***

- ◆ Single URI (endpoint). Variants:

- ❖ POST + Payload: SOAP, XML RPC, POX

- ([Flickr](#)) ❖ GET uri?method=method\_name&par1=val1&par2=val2

## ▶ **Level 1: *URI Tunneling*** ([Twitter](#))

- ◆ Many URIs
- ◆ Few verbs: GET, POST

## ▶ **Level 2: *CRUD API***

- ◆ Many URIs: collections and individuals
- ◆ Many verbs: GET, POST, PUT, DELETE

## ▶ **Level 3: *Hypermedia API***

- ◆ Level 2 + HATEOAS



# Design of Web Services

---

- ▶ Definition and Features of Web Services
- ▶ Types of Web Services
- ▶ **RESTful Web Services**
  - ◆ Concepts
  - ◆ Architectural Principles
  - ◆ Designing RESTful Web Services
    - ❖ Creating the Service API
    - ❖ Interaction between Clients and Services
  - ◆ **Example**
- ▶ SOAP or Big Web Services

# RESTful Web Services: Example

---

- ▶ **Delicious** is a social bookmarking web service for storing, sharing, and discovering web bookmarks. Users can tag each of their bookmarks with freely chosen index terms
- ▶ Delicious has a simple RESTful API that allows:
  - ◆ Get a list of all our bookmarks and to filter that list by tag or date or limit by number
  - ◆ Get the number of bookmarks created on different dates
  - ◆ Get the last time we updated our bookmarks
  - ◆ Get a list of all our tags
  - ◆ Add a bookmark
  - ◆ Edit a bookmark
  - ◆ Delete a bookmark
  - ◆ Rename a tag

# RESTful Web Services: Example

---

- ▶ Bookmarks and tags are exposed as resources at the URL paths of `http://del.icio.us/api/[username]/bookmarks` and `http://del.icio.us/api/[username]/tags` where `[username]` is the username of the user's bookmarks
- ▶ It is necessary to define some XML document formats to represent bookmarks and tags. A proprietary XML is used
- ▶ Getting Tags: it allows to get tag names
- ▶ The complete example may be found at:  
<http://www.peej.co.uk/articles/restfully-delicious.html>

# RESTful Web Services: Example

---

Title	Getting Tags	
URI	http://del.icio.us/api/:username/tags/	
Method	GET	
URI Params	start= [Integer]	The number of the first tag to return
	end= [Integer]	The number of the last tag to return
Returns	Success Response	200 OK & XML (delicious/tags+xml)
	Error Response	401 Unauthorized
		404 Not Found

```
GET http://del.icio.us/api/peej/tags/?start=1&end=2
<?xml version="1.0"?>
<tags start="1" end="2" next="http://del.icio.us/api/peej/tags?start=3&end=4">
  <tag name="example" count="2" href="http://del.icio.us/api/peej/tags/example"/>
  <tag name="test" count="2" href="http://del.icio.us/api/peej/tags/test"/>
</tags>
```

\* Extracted from: <http://www.peej.co.uk/articles/restfully-delicious.html>

# References

---

- ▶ Leonard Richardson and Sam Ruby. **RESTful Web Services**. O'Reilly, 2007
- ▶ Robert Daigneau. **Service Design Patterns. Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services**. Addison Wesley, 2012
- ▶ Thomas Erl. **Service-Oriented Architecture. Concepts, Technology, and Design**. Prentice Hall, 2005
- ▶ <http://www.w3.org/TR/ws-arch/#concepts>
- ▶ <http://www.java2blog.com/2013/03/web-service-tutorial.html>
- ▶ <http://www.peej.co.uk/articles/restfully-delicious.html>