[ASW] Unit 5: Design of Web Services

Introduction to 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

Jeff Bezzos' API Mandate (circa 2002)

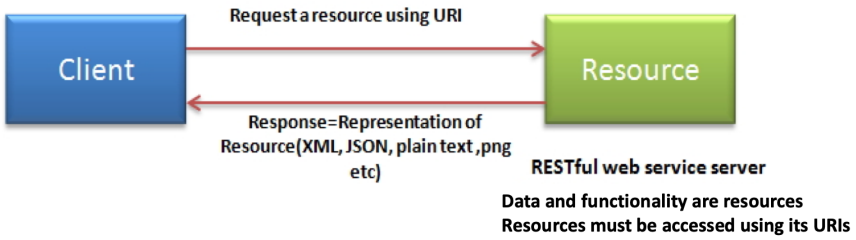
- All teams will henceforth expose their data and functionality through service interfaces.

- Teams must communicate with each other through these interfaces.
- There will be no other form of inter-process communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network.
- It doesn't matter what technology you use.
- All service interfaces, without exception, must be designed from the ground up to be externalize-able. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world. No exceptions.
- The mandate closed with:
  - **Anyone who doesn't do this will be fired. Thank you; have a nice day!**

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

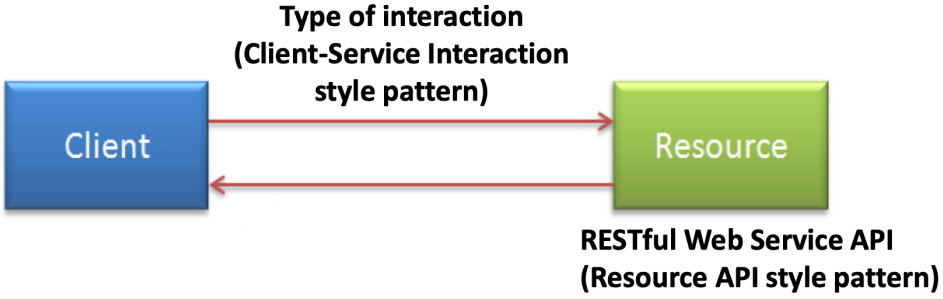
Architectural Principles

- **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
- **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

Designing RESTful Web Services

To design a RESTful Web Service we have to:

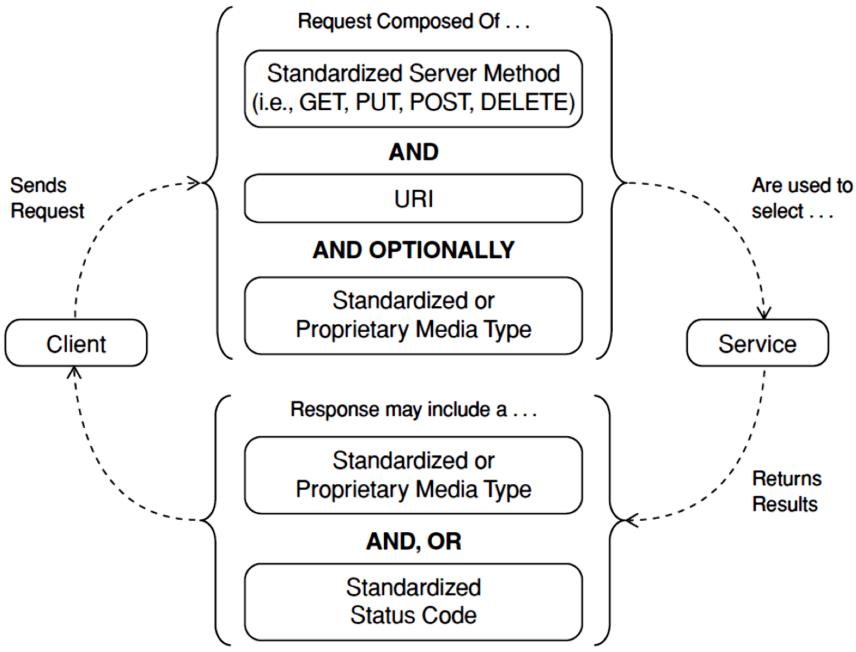
- Create a Service API
  - Resource API style pattern
- Define the type of interaction between the client and the service
  - Client-Service Interaction style pattern



Creating the Service API

Pattern name	Problem	Description	WS Tech.
RPC API	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-*
Message API	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-*
Resource API	How can a client manipulate data managed by a remote system, avoid direct coupling to remote procedures, and minimise 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 behaviours	REST

- 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...).

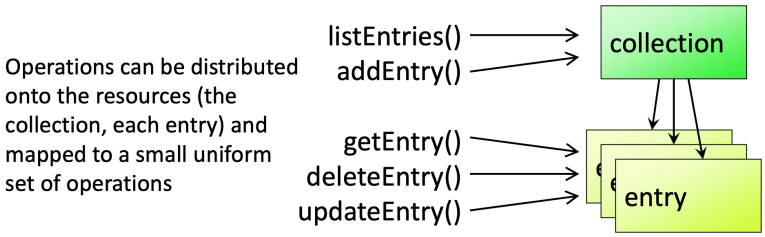


A Blueprint for designing Resource APIs

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
4. Design and document ingoing/outgoing resource representations
5. Use status codes and headers meaningfully in server responses

(1) Identify resources to be exposed as services

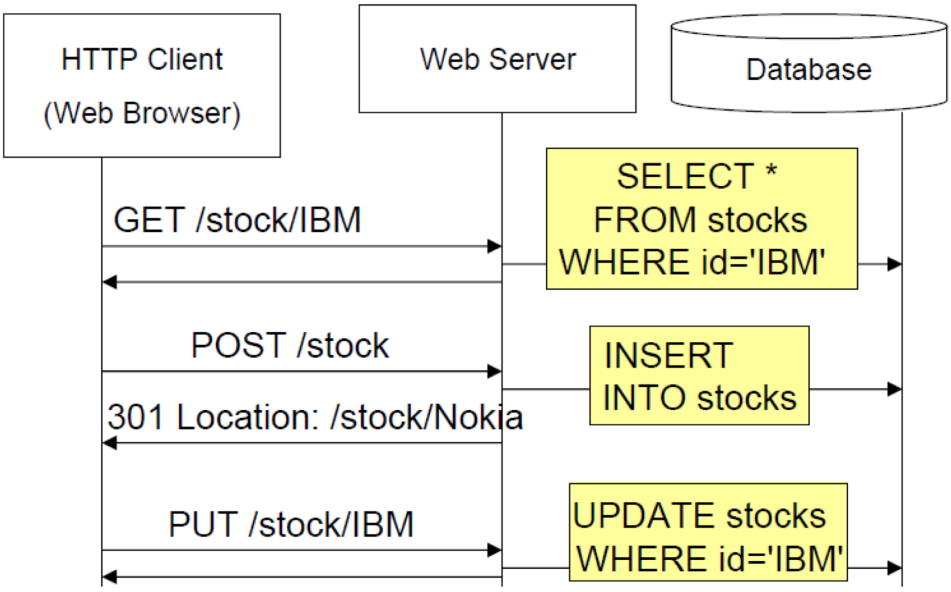
- A resource is anything that is important enough to be referenced as a thing itself
- Usually we have 2 levels of abstraction:
  - A collection: is a resource representing a whole class and/or set of individuals
  - An entry: is a resource representing an instance/individual within a class/set



(2) Define “nice” URIs to address the resources

- 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)

(3) Understand what it means to do a `GET`, `POST`, `PUT`, `DELETE` for each resource



3. Understanding HTTP methods (Example)

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

- 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

(4) Design and document `ingoing` resource representations

- Ingoing representations = data sent by the Client to the Server
- Simple representations are usually key-value pairs:
  - `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 from the Server)

(4) Design and document `outgoing` resource representations

- Outgoing representations = data sent back by the Server to the 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
- Nowadays, JSON is the most used format.

(5) Use status codes and headers meaningfully in server responses

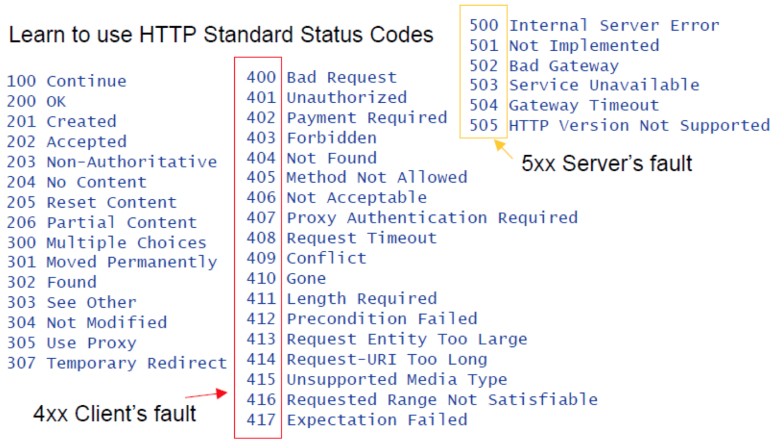
Successes

- 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)

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



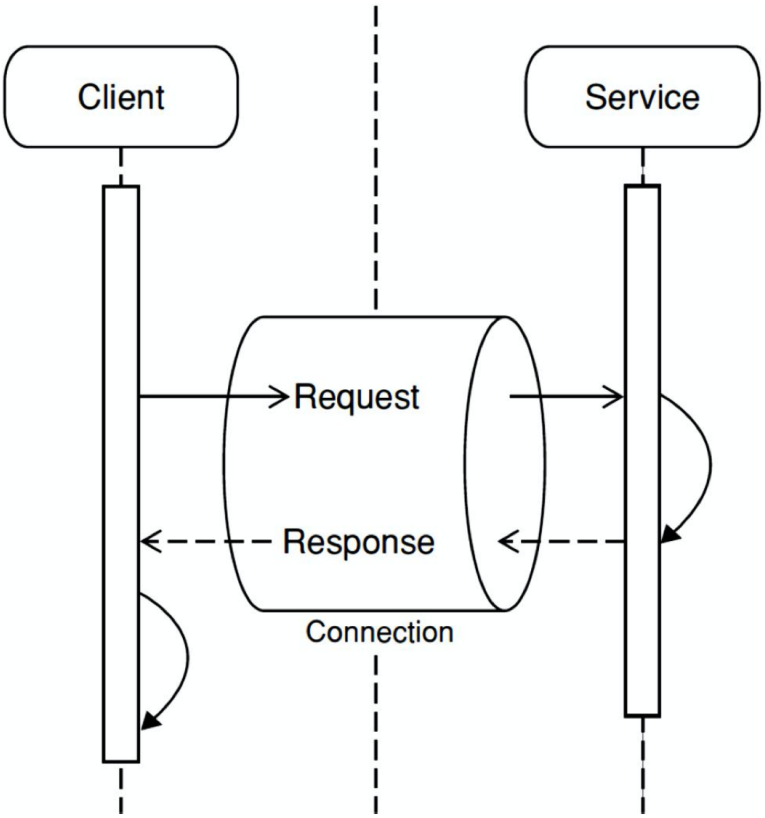
Interaction between Clients and Services

Defining the interaction

Aa Pattern name	Problem	Description

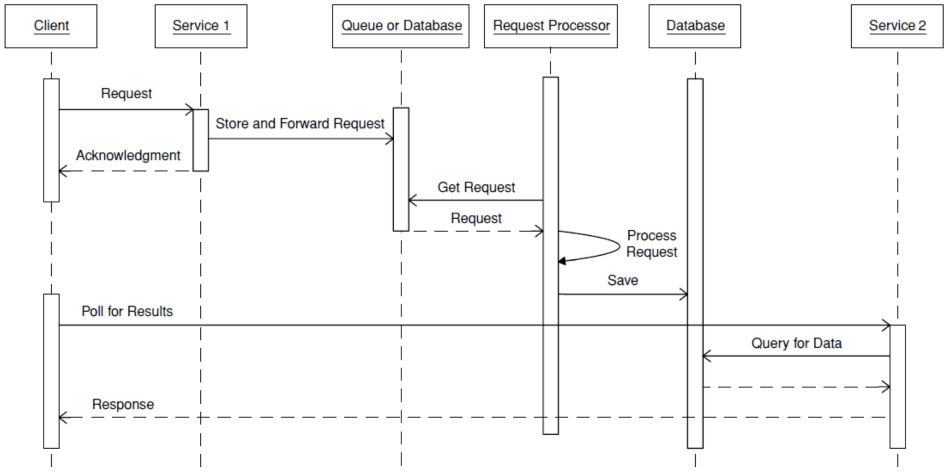
Aa Pattern name	Problem	Description
Request/ Response	What's the simplest way for a web service to process a request and provide a result?	Process requests when they're received and return results over the same client connection.
Request/ Acknowledge	How can a web service safeguard systems from spikes 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.
Media Type Negotiation	How can a web service provide multiple representations of the same logical resource while minimising 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.
Linked Service	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.

Request/Response Pattern



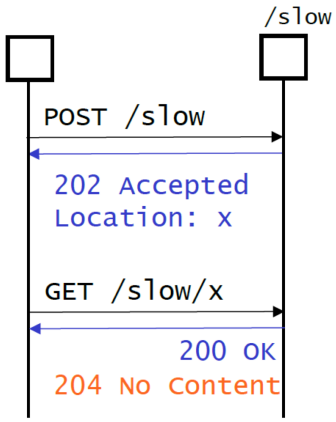
Request/Acknowledge Pattern

Poll variation

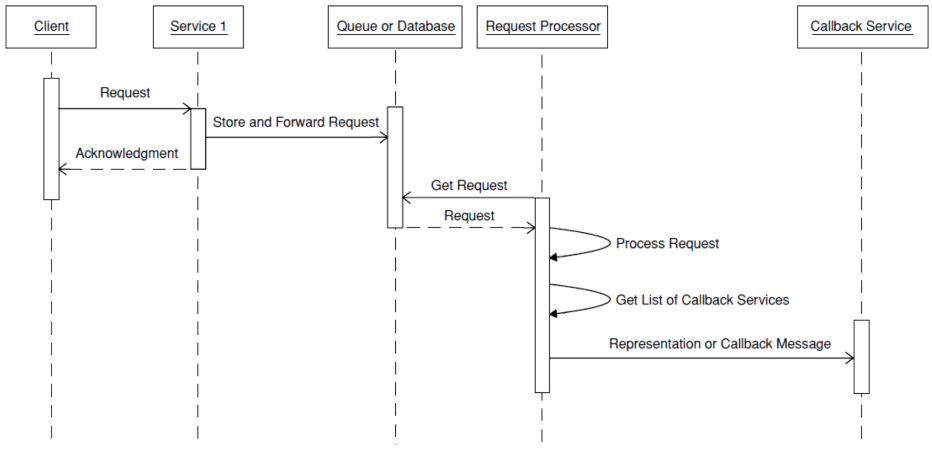


- The server may answer it with 202 Accepted providing a URI from which the response can be retrieved later
- Problem: how often should the client do the polling? /slow/x could include an estimate of

the finishing time if not yet completed



Callback variation



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

▶ Forced Media Type Negotiation Pattern

- The specific URI points to a specific representation format using the postfix

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

(extension)

- 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).

▶ Media Type Negotiation Pattern

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

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

- 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)

▶ Advanced Media Type Negotiation Pattern

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

- 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.

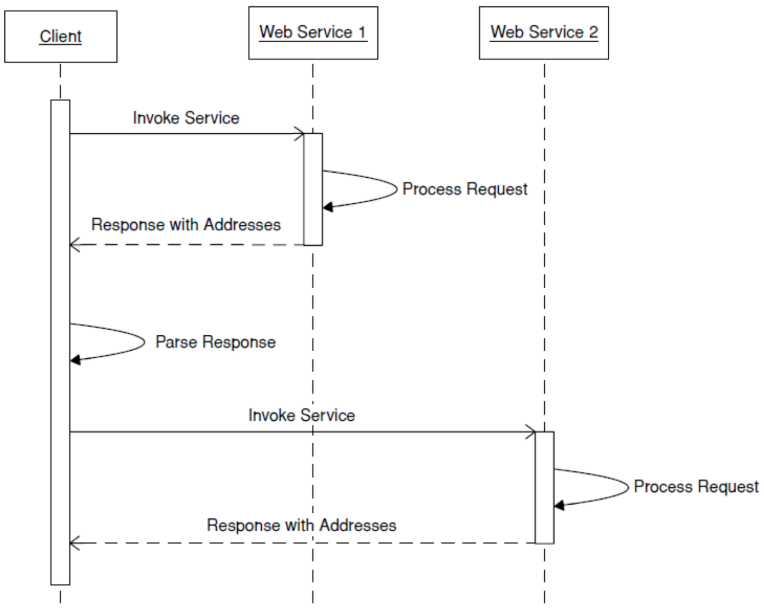
- **Ex1**: The client prefers to receive HTML (but any other text format will do with lower priority)

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

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

- **Ex2**: The client prefers to receive XHTML, or HTML if this is not available and will use Plain Text as a fallback

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

▶ 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"
  }
  , ... ], ...
}
```

▶ 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: *HTPP Tunneling***
  - Single URI (endpoint). Variants:
    - `POST` + Payload: SOAP, XML RPC, POX
    - `GET uri?method=method_name&par1=val1&par2=val2` (Flickr)
- **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

SOAP or Big Web Services