

Middleware Architectures 1

Lecture 4: HATEOAS, Caching and Concurrency

doc. Ing. Tomáš Vitvar, Ph.D.

tomas@vitvar.com • @TomasVitvar • <http://vitvar.com>



Czech Technical University in Prague

Faculty of Information Technologies • Software and Web Engineering • <http://vitvar.com/courses/mdw>



Modified: Mon Dec 14 2020, 12:56:16
Humla v0.3

Overview

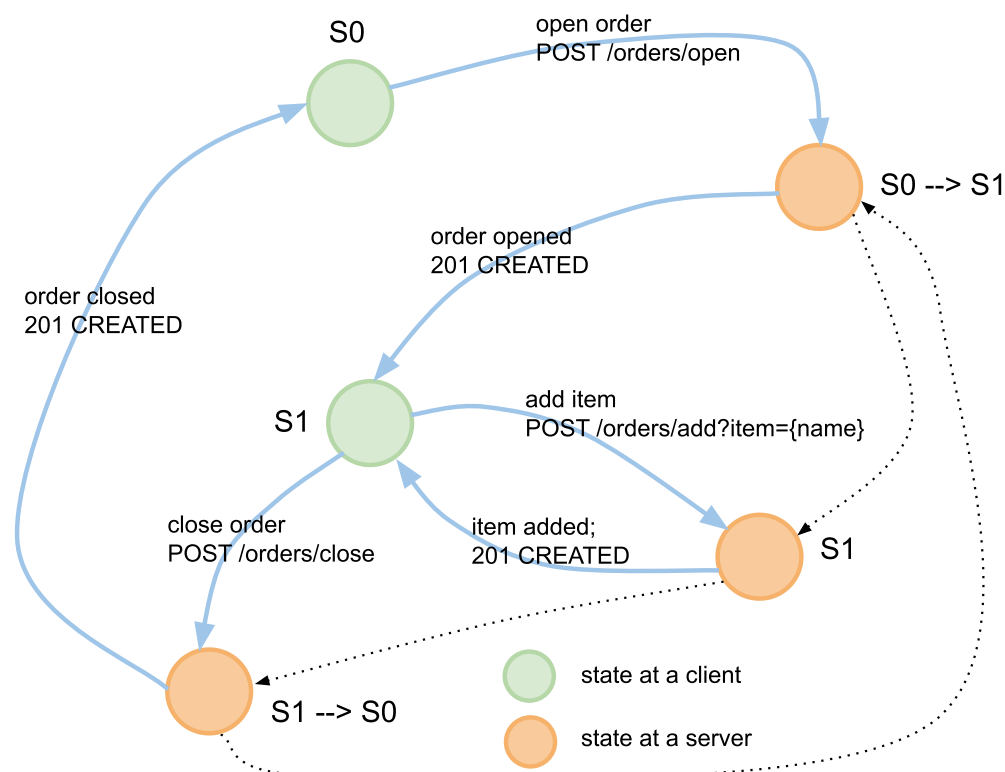
- HATEOAS
- Caching, Revalidation, Concurrency Control
- Richardson Maturity Model

HATEOAS

- HATEOAS = Hypertext as the Engine for Application State
 - *The REST core principle*
 - ***Hypertext***
 - *Hypertext is a representation of a resource state with **links***
 - *A link is an URI of a resource*
 - *Applying an access (PUT, POST, DELETE) to a resource via its link = state transition*
- Statelessness
 - *A service does not use a session memory to remember a state*
 - *HATEOAS enables stateless implementation of services*

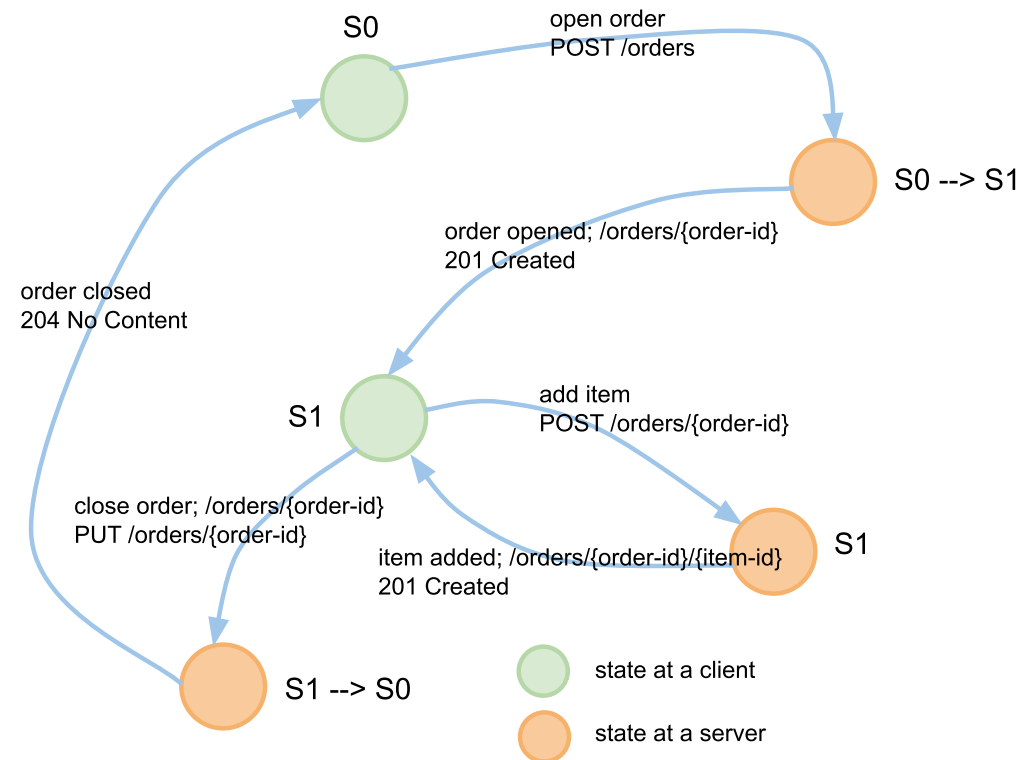
Stateful server

- Sessions to store the application state
 - *The app uses a server memory to remember the state*
 - *When the server restarts, the app state is lost*



Stateless server

- HTTP and hypermedia to transfer the app state
 - *Does not use a server memory to remember the app state*
 - *State transferred between a client and a service via HTTP metadata and resources' representations*

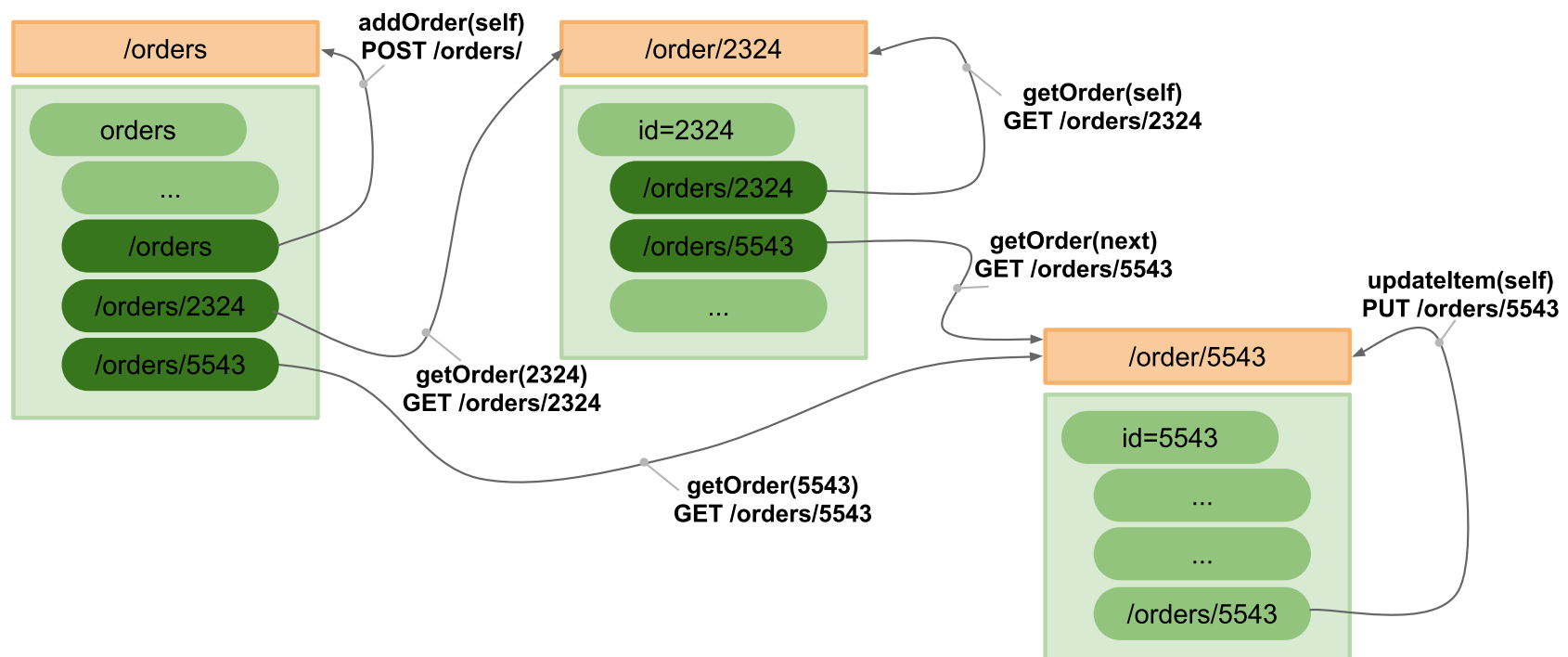


Persistent Storage and Session Memory

- Persistent Storage
 - *Contains the app data*
 - *Data is serialized into resource representation formats*
 - *All sessions may access the data via resource IDs*
- Session Memory
 - *Server memory that contains a state of the app*
 - *A session may only access its session memory*
 - *Access through cookies*
 - *Note*
 - *A session memory may be implemented via a persistent storage (such as in Google AppEngine)*

Link

- Service operation
 - Applying an access to a link (*GET, PUT, POST, DELETE*)
 - Link: *HTTP method + resource URI + optional link semantics*
- Example: **getOrder**, **addOrder**, and **updateItem**



Atom Links

- Atom Syndication Format
 - *XML-based document format; Atom feeds*
 - *Atom links becoming popular for RESTful applications*

```
1 <order a:xmlns="http://www.w3.org/2005/Atom" xmlns="...">
2   <a:link
3     rel="next"
4     href="http://company.com/orders/5543"
5     type="application/xml"/>
6   <customer>Tomas</customer>
7   <items>...</items>
8 </order>
```

- *Link structure*
 - rel** – *name of the link*
~ *semantics of an operation behind the link*
 - href** – *URI to the resource described by the link*
 - type** – *media type of the resource the link points to*

Link Semantics

- Standard **rel** values
 - *Navigation: next, previous, self*
 - *Does not reflect a HTTP method you can use*
- Extension **rel** values
 - *You can use **rel** to indicate a semantics of an operation*
 - *Example: add item, delete order, update order, etc.*
 - *A client associates this semantics with an operation it may apply at a particular state*
 - *The semantics should be defined by using an URI*

```
1 <order a:xmlns="http://www.w3.org/2005/Atom" xmlns="...">
2   <id>2324</id>
3   <a:link rel="http://company.com/op/addItem"
4     href="http://company.com/orders/2324"/>
5   <a:link rel="http://company.com/op/deleteOrder"
6     href="http://company.com/orders/2324"/>
7 </order>
```

Link Headers

- An alternative to Atom links in resource representations
 - *links defined in HTTP Link header, Web Linking IETF spec* [!\[\]\(467d80e979964f7f8c752fb22248b5b7_img.jpg\)](#)
 - *They have the same semantics as Atom Links*
 - *Example:*

```
> HEAD /orders HTTP/1.1
```

```
< Content-Type: application/xml
```

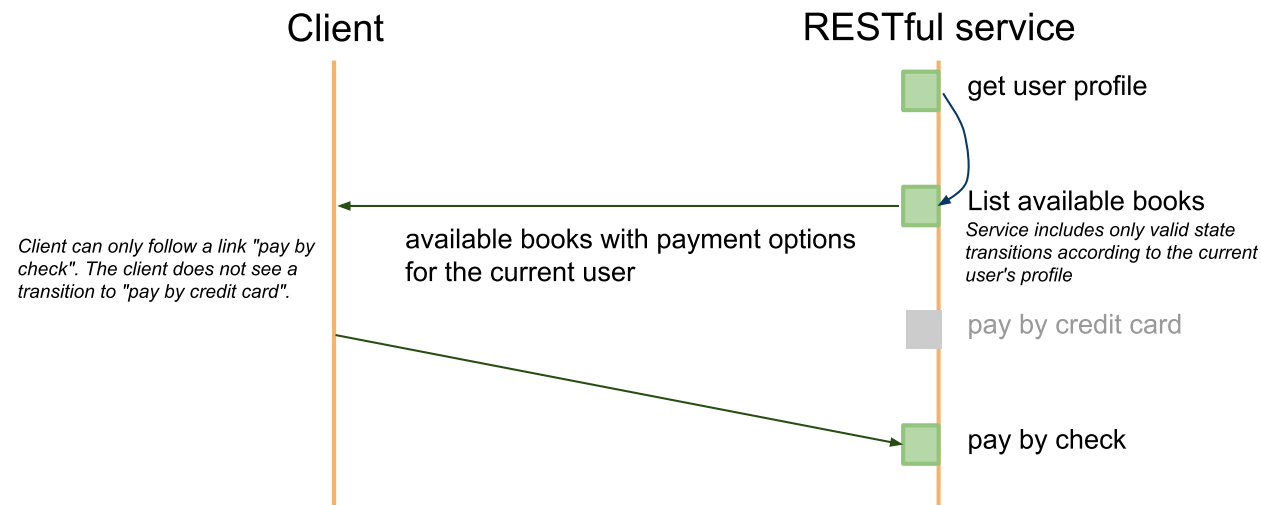
```
< Link: <http://company.com/orders/?page=2&size=10>; rel="next"
```

```
< Link: <http://company.com/orders/?page=10&size=10>; rel="last"
```

- Advantages
 - *no need to get the entire document*
 - *no need to parse the document to retrieve links*
 - *use HTTP HEAD only*

Preconditions and HATEOAS

- Preconditions in HATEOAS
 - *Service in a current state generates only valid transitions that it includes in the representation of the resource.*
 - *Transition logic is realized at the server-side*



Advantages

- Location transparency
 - *only "entry-level" links published to the World*
 - *other links within documents can change without changing client's logic*
 - *Hypertext represents the current user's view, i.e. rights or other context*
- Loose coupling
 - *no need for a logic to construct the links*
 - *Clients know to which states they can move via links*
- Statelessness and Cloud
 - *Better implementation of scalability*

Overview

- HATEOAS
- Caching, Revalidation, Concurrency Control
- Richardson Maturity Model

Scalability

- Need for scalability
 - *Huge amount of requests on the Web every day*
 - *Huge amount of data downloaded*
- Some examples
 - *Google, Facebook: 5 billion API calls/day*
 - *Twitter: 3 billions of API calls/day (75% of all the traffic)*
 - *50 million tweets a day*
 - *eBay: 8 billion API calls/month*
 - *Bing: 3 billion API calls/month*
 - *Amazon WS: over 100 billion objects stored in S3*
- Scalability in REST
 - *Caching and revalidation*
 - *Concurrency control*

Caching



- Your service should cache:
 - *anytime there is a static resource*
 - *even there is a dynamic resource*
 - *with chances it updates often*
 - *you can force clients to always revalidate*
- three steps:
 - *client GETs the resource representation*
 - *server controls how it should cache through **Cache-Control** header*
 - *client revalidates the content via conditional GET*

Cache Headers

- **Cache-Control** response header
 - *controls over local and proxy caches*
 - **private** – *no proxy should cache, only clients can*
 - **public** – *any intermediary can cache (proxies and clients)*
 - **no-cache** – *the response should not be cached. If it is cached, the content should always be revalidated.*
 - **no-store** – *must not store persistently (this turns off caching)*
 - **no-transform** – *no transformation of cached data; e.g. compressions*
 - **max-age**, **s-maxage** *a time in seconds how long the cache is valid; s-maxage for proxies*
- **Last-Modified** and **ETag** response headers
 - *Content last modified date and a content entity tag*
- **If-Modified-Since** and **If-None-Match** request headers
 - *Content revalidation (conditional GET)*

Example Date Revalidation

- Cache control example:

```
> GET /orders HTTP/1.1
> ...

< HTTP/1.1 200 OK
< Content-Type: application/xml
< Cache-Control: private, max-age=200
< Last-Modified: Sun, 7 Nov 2011, 09:40 CET
<
< ...data...
```

— *only client can cache, the cache is valid for 200 seconds.*

- Revalidation (conditional GET) example:

— *A client revalidates the cache after 200 seconds.*

```
> GET /orders HTTP/1.1
> If-Modified-Since: Sun, 7 Nov 2011, 09:40 CET

< HTTP/1.1 304 Not Modified
< Cache-Control: private, max-age=200
< Last-Modified: Sun, 7 Nov 2011, 09:40 CET
```

Entity Tags

- Signature of the response body
 - *A hash such as MD5*
 - *A sequence number that changes with any modification of the content*
- Types of tag
 - *Strong ETag: reflects the content bit by bit*
 - *Weak ETag: reflects the content "semantically"*
 - *The app defines the meaning of its weak tags*
- Example content revalidation with ETag

```
< HTTP/1.1 200 OK
< Cache-Control: private, max-age=200
< Last-Modified: Sun, 7 Nov 2011, 09:40 CET
< ETag: "4354a5f6423b43a54d"
```

```
> GET /orders HTTP/1.1
> If-None-Match: "4354a5f6423b43a54d"
```

```
< HTTP/1.1 304 Not Modified
< Cache-Control: private, max-age=200
< Last-Modified: Sun, 7 Nov 2011, 09:40 CET
< ETag: "4354a5f6423b43a54d"
```

Design Suggestions

- Composed resources use weak ETags
 - For example */orders*
 - a composed resource that contains a summary information
 - changes to an order's items will not change semantics of */orders*
 - It is usually not possible to perform updates on these resources
- Non-composed resources use strong ETags
 - For example */orders/{order-id}*
 - They can be updated
- Further notes
 - Server should send both *Last-Modified* and *ETag* headers
 - If client sends both *If-Modified-Since* and *If-None-Match*, *ETag* validation takes preference

Weak ETag Example

- App specific, `/orders` resource example

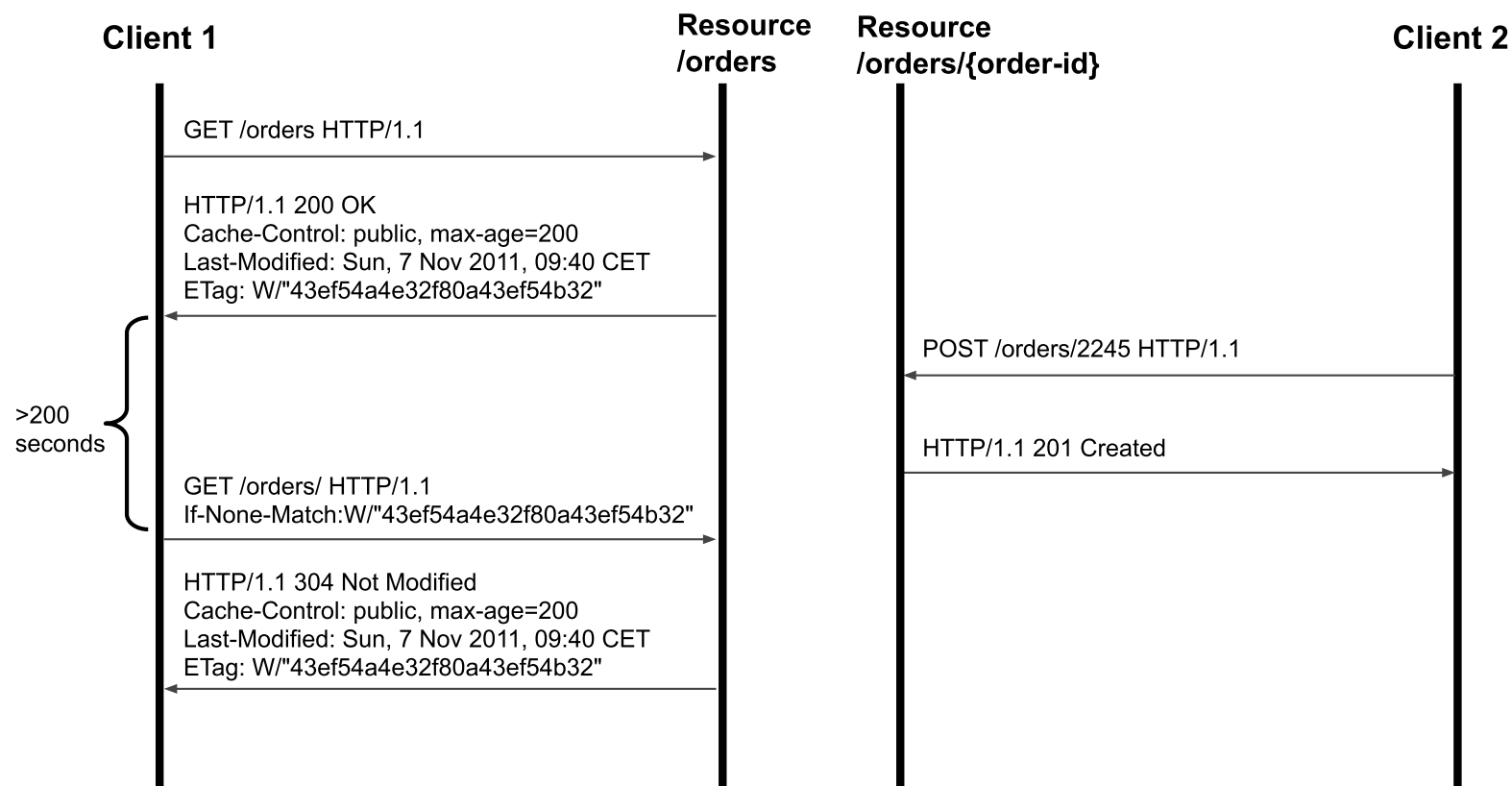
```
1  {
2    "orders" :
3      [
4        { "id"      : 2245,
5          "customer" : "Tomas",
6          "descr"    : "Stuff to build a house.",
7          "items"    : [...] },
8        { "id"      : 5546,
9          "customer" : "Peter",
10         "descr"    : "Things to build a pipeline.",
11         "items"    : [...] }
12      ]
13  }
```

- Weak ETag compute function example
 - *Any modification to an order's items is not significant for `/orders`:*

```
1  var crypto = require("crypto");
2
3  function computeWeakETag(orders) {
4    var content = "";
5    for (var i = 0; i < orders.length; i++)
6      content += orders[i].id + orders[i].customer + orders[i].descr;
7    return crypto.createHash('md5').update(content).digest("hex");
8  }
```

Weak ETag Revalidation

- Updating **/orders** resource
 - **POST** **/orders/{order-id}** *inserts a new item to an order*
 - *Any changes to orders' items will not change the Weak ETag*



Concurrency

- Two clients may update the same resource

1) a client GETs a resource GET /orders/5545

2) the client modifies the resource

3) the client updates the resource via PUT /orders/5545 HTTP/1.1

What happens if another client updates the resource between 1) and 3) ?

- Concurrency control

- Conditional PUT

- Update the resource only if it has not changed since a specified date or a specified ETag matches the resource content

- If-Unmodified-Since and If-Match headers

- Response to conditional PUT:

- 200 OK if the PUT was successful

- 412 Precondition Failed if the resource was updated in the meantime.

Concurrency Control Protocol



- Conditional PUT and ETags
 - *Conditional PUT must always use strong entity tags or date validation*

Overview

- HATEOAS
- Caching, Revalidation, Concurrency Control
- Richardson Maturity Model

Steps towards REST



See Richardson Maturity Model [details](#).

Levels

- LEVEL 0 – POX (Plain Old XML)
 - *HTTP as a tunneling mechanism*
 - *URL defines a service endpoint*
 - *No Web principles*
- LEVEL 1 – Resources
 - *Take advantages of resources and URIs*
- LEVEL 2 – HTTP Verbs
 - *Use HTTP methods and respect their semantics*
- LEVEL 3 – Hypermedia Controls
 - *HATEOAS*