

Middleware and Web Services

Lecture 4: Advanced Service Concepts and Technologies

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: Sun Oct 06 2019, 21:20:20
Humla v0.3

Overview

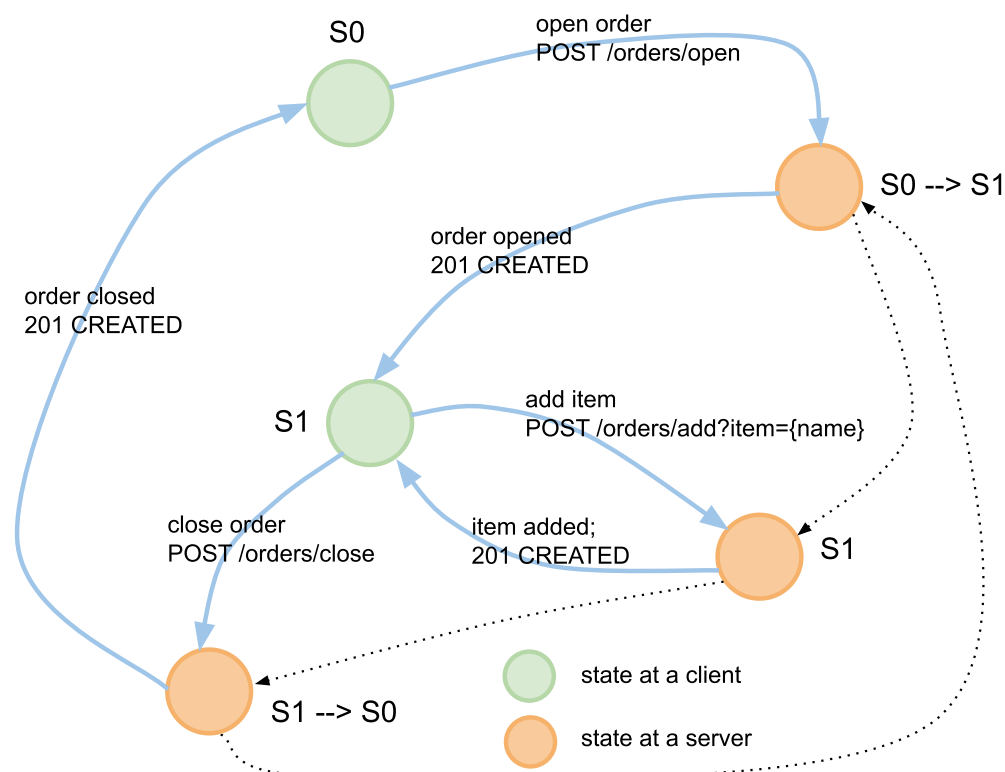
- HATEOAS
- Caching, Revalidation, Concurrency Control
- Richardson Maturity Model
- SOAP and WSDL

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*

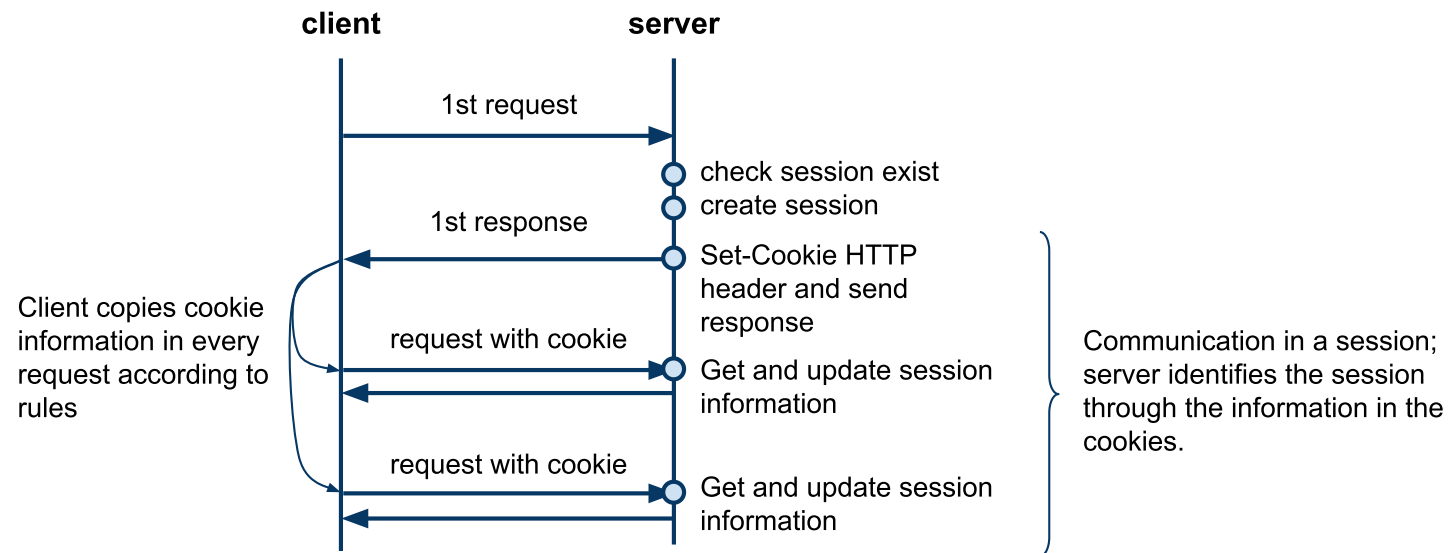


State Management

- HTTP is a stateless protocol – original design
 - *No information to relate multiple interactions at server-side*
 - Except **Authorization** header is copied in every request
 - IP addresses do not work, one public IP can be shared by multiple clients
- Solutions to check for a valid state at server-side
 - **Cookies** – obvious and the most common workaround
 - RFC 2109 – HTTP State Management Mechanism 
 - Allow clients and servers to talk in a context called **sessions**
 - **Hypertext** – original HTTP design principle
 - App states represented by resources (hypermedia), links define transitions between states
 - Adopted by the REST principle **statelessness**

Interaction with Cookies

- Request-response interaction with cookies
 - *Session is a logical channel maintained by the server*



- Stateful Server
 - *Server remembers the session information in a server memory*
 - *Server memory is a non-persistent storage, when server restarts the memory content is lost!*

Set-Cookie and Cookie Headers

- **Set-Cookie** response header

```
1 set-cookie = "Set-Cookie:" cookie ("," cookie)*
2   cookie   = NAME "=" VALUE (";" cookie-av)*
3   cookie-av = "Comment" "=" value
4             | "Domain" "=" value
5             | "Max-Age" "=" value
6             | "Path" "=" value
```

- **domain** – *a domain for which the cookie is applied*
- **Max-Age** – *number of seconds the cookie is valid*
- **Path** – *URL path for which the cookie is applied*

- **Cookie** request header. A client sends the cookie in a request if:

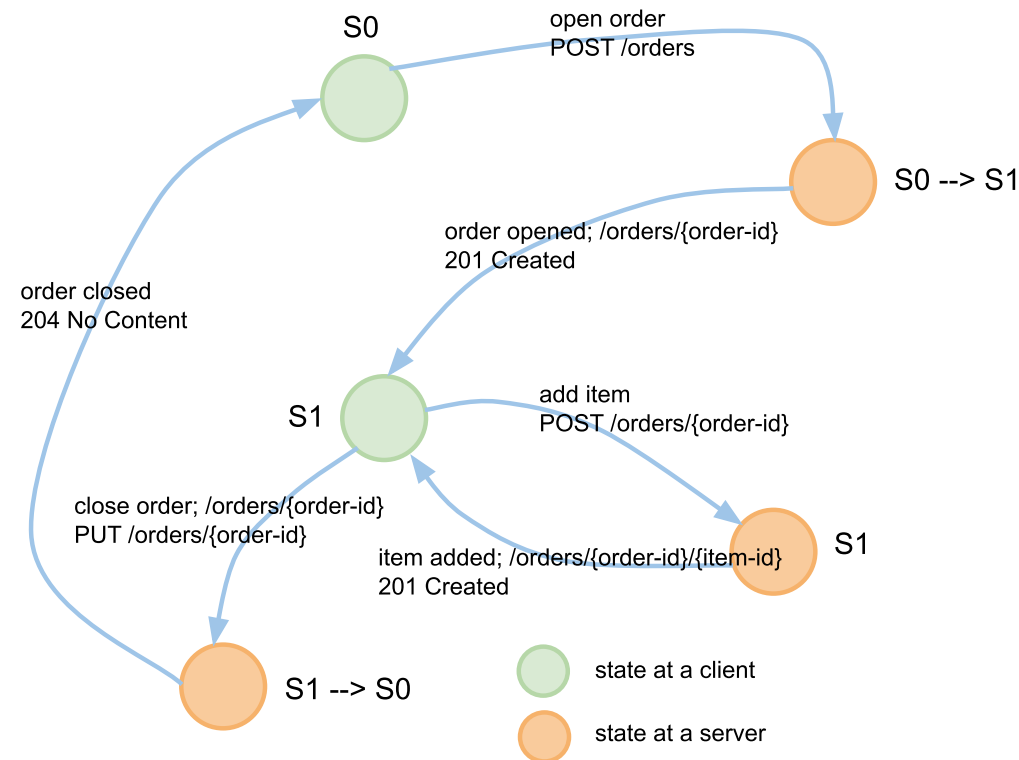
- **domain** *matches the origin server's fully-qualified host name*
- **path** *matches a prefix of the request-URI*
- **Max-Age** *has not expired*

```
1 cookie = "Cookie:" cookie-value (";" cookie-value)*
2   cookie-value = NAME "=" VALUE [";" path] [";" domain]
3   path         = "$Path" "=" value
4   domain       = "$Domain" "=" value
```

- **domain**, and **path** *are values from corresponding attributes of the Set-Cookie header*

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*

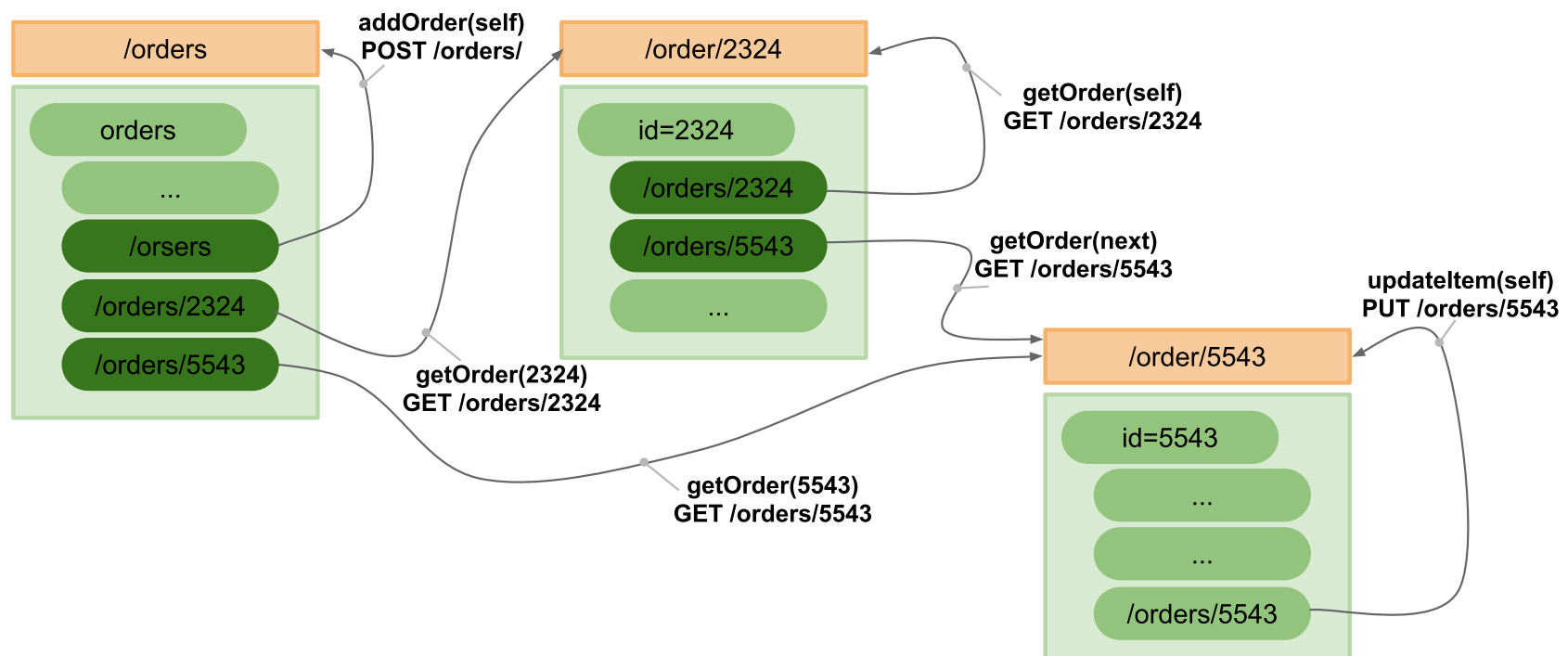


Persistent Storage and Session Memory

- 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* [!\[\]\(3da2b303d29c1ea489bbe26a3f5ac664_img.jpg\)](#)
 - *They have the same semantics as Atom Links*
 - *Example:*

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

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

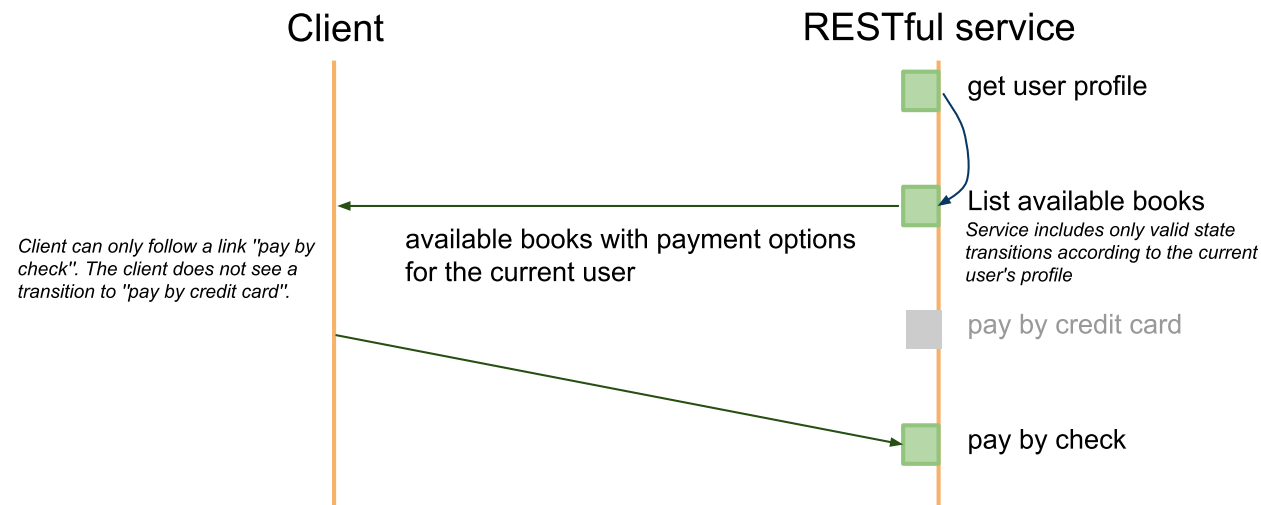
```
< Link: <http://company.com/orders/?page=2&size=10>; rel="next"
```

```
< 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*

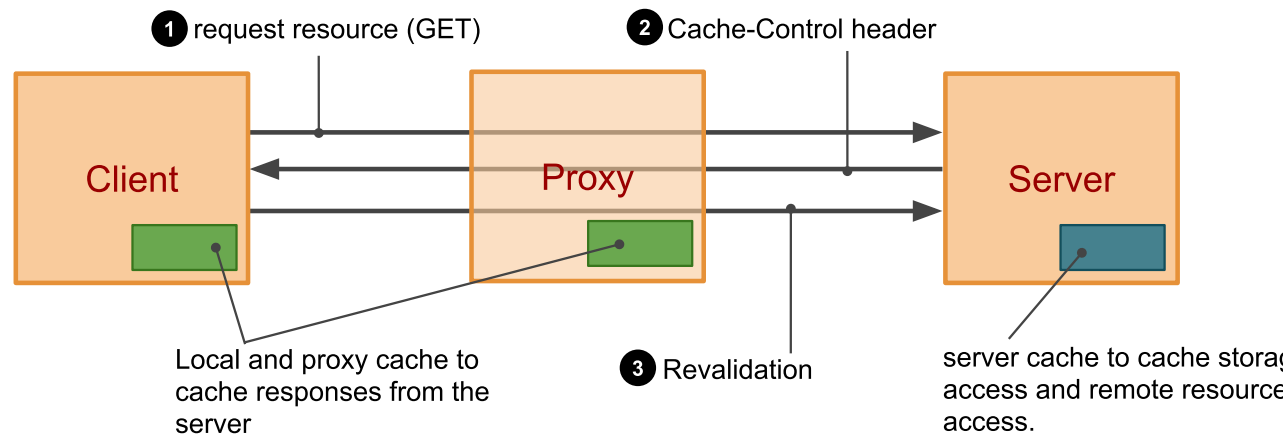
Overview

- HATEOAS
- Caching, Revalidation, Concurrency Control
- Richardson Maturity Model
- SOAP and WSDL

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

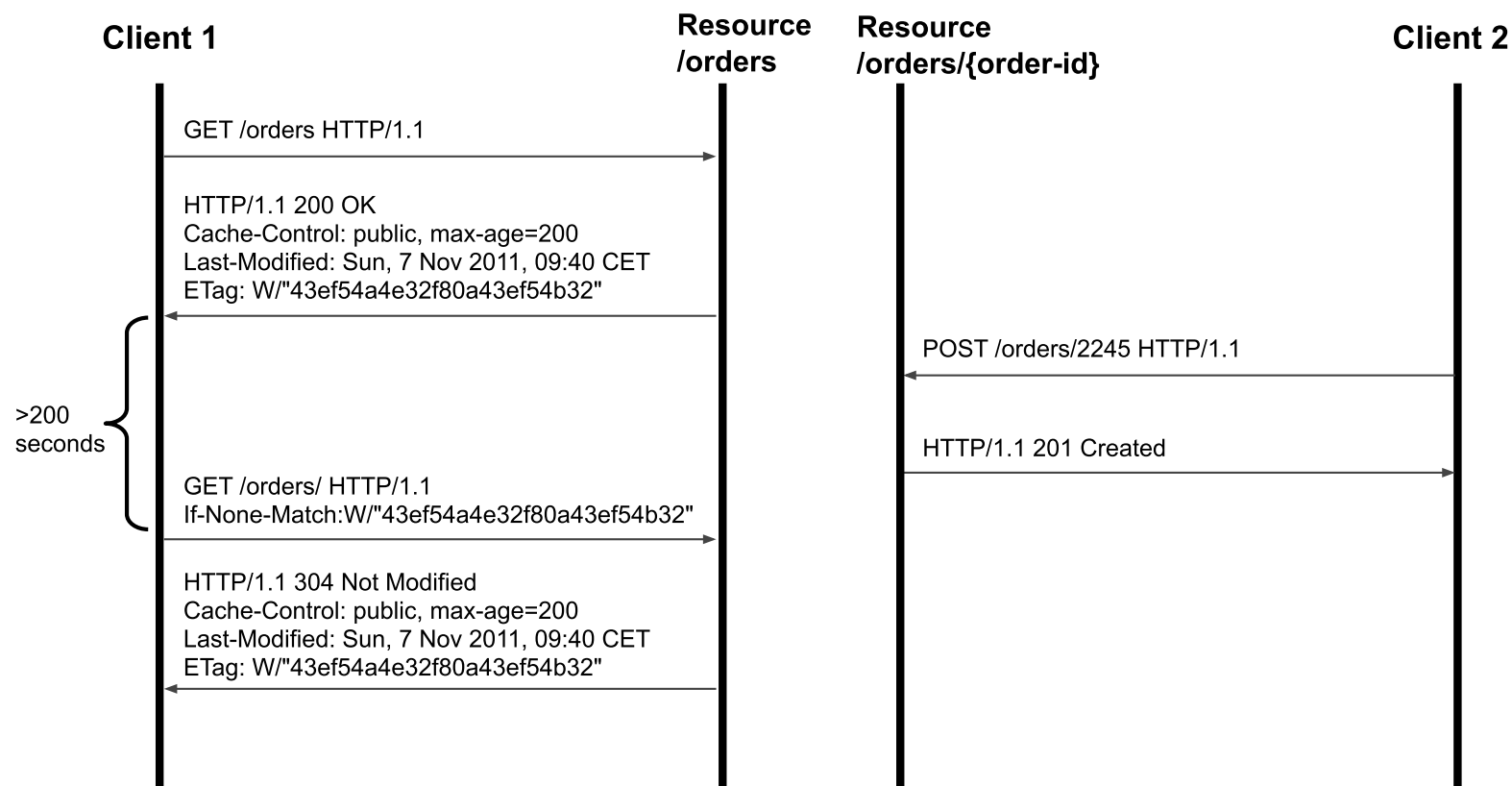
```
1  {  
2    "orders" :  
3      [  
4        { "id"      : 2245,  
5          "customer" : "Tomas",  
6          "descr"    : "Stuff to build a house.",  
7          "items"    : [...] },  
8        { "id"      : 5546,  
9          "customer" : "Peter",  
10         "descr"    : "Things to build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

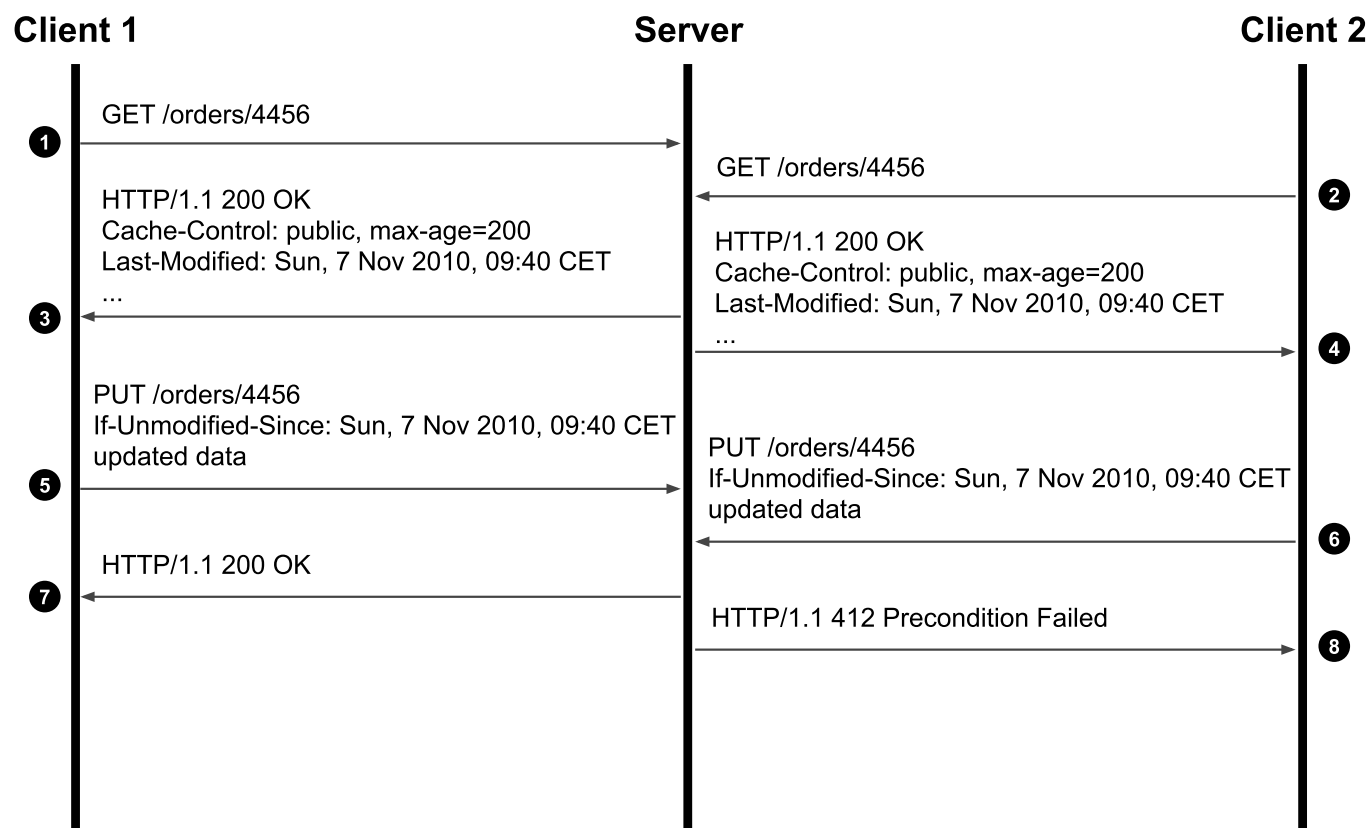
- If-Unmodified-Since and If-Match headers

- 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

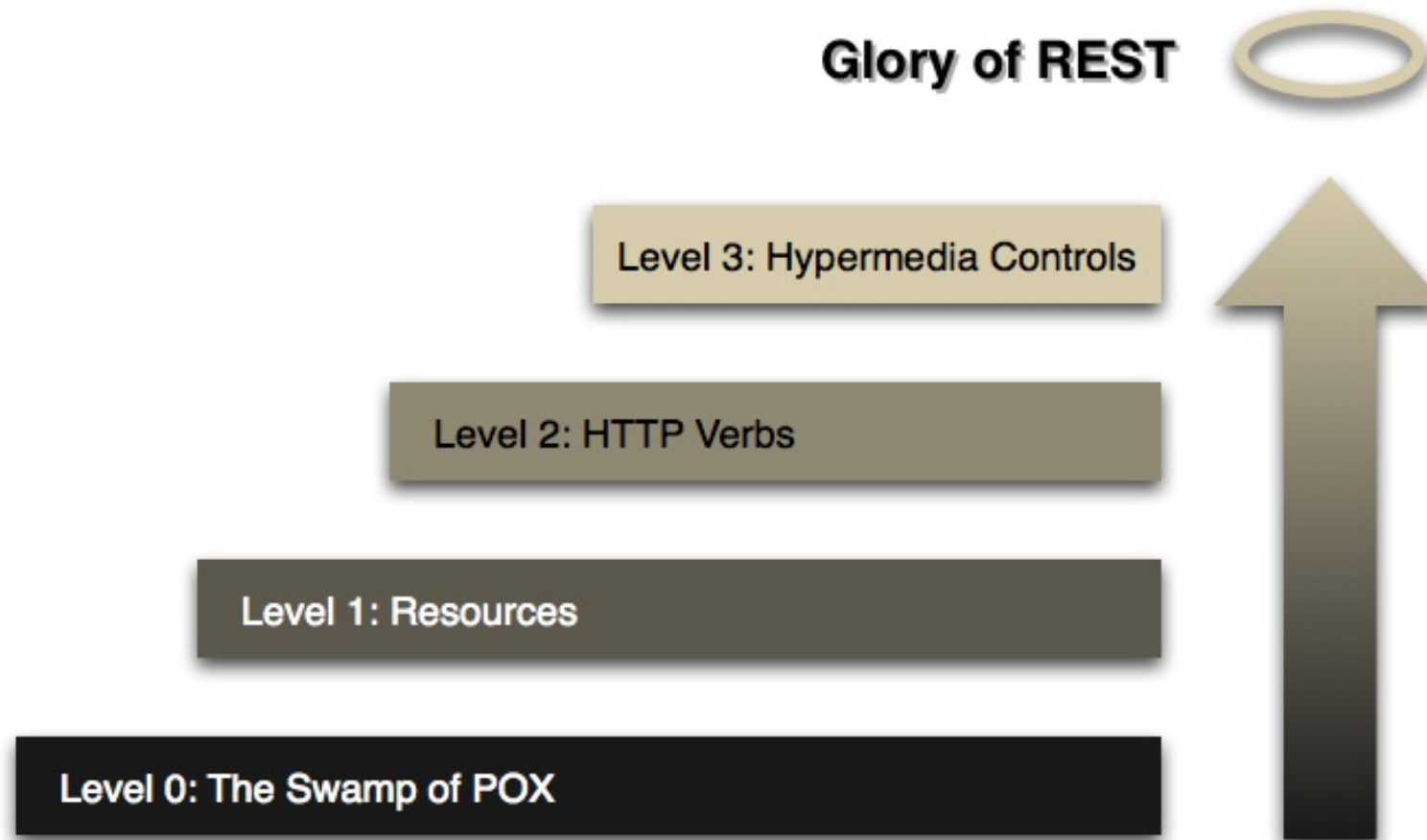


- Conditional PUT and ETags
 - *Conditional PUT must always use strong entity tags or date validation*

Overview

- HATEOAS
- Caching, Revalidation, Concurrency Control
- Richardson Maturity Model
- SOAP and WSDL

Steps towards REST



See Richardson Maturity Model [details](#).

Levels

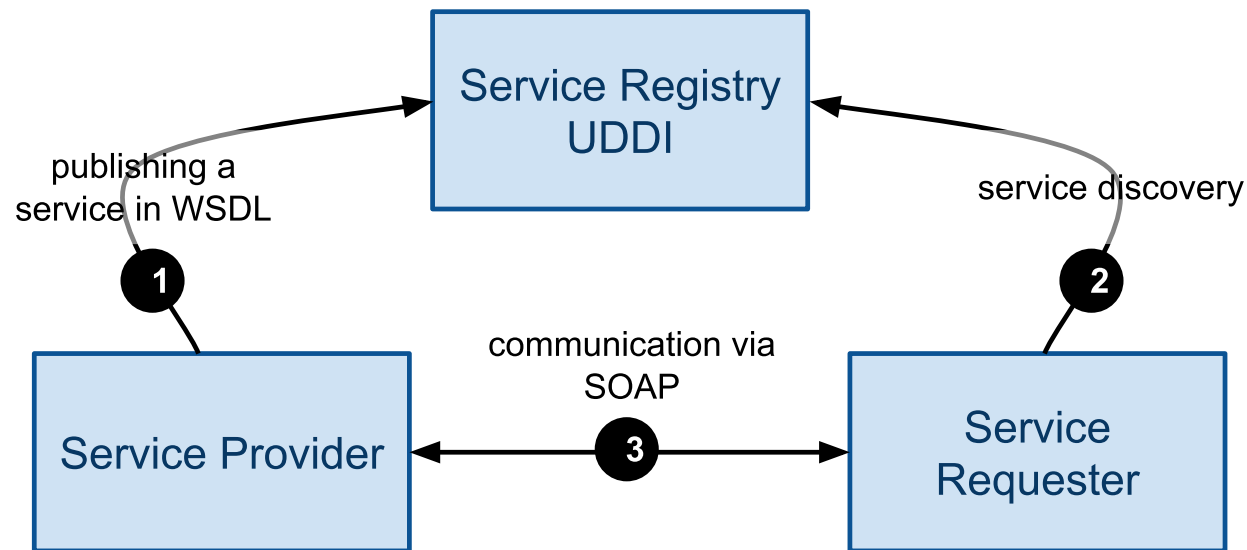
- LEVEL 0 – POX (Plain Old XML)
 - *HTTP as a tunneling mechanism*
 - *URL defines a service endpoint*
 - *No Web principles*
- LEVEL 1 – Resources
 - *Take advantages of reosources and URIs*
- LEVEL 2 – HTTP Verbs
 - *Use HTTP methods and respect their semantics*
- LEVEL 3 – Hypermedia Conrols
 - *HATEOS*

Overview

- HATEOAS
- Caching, Revalidation, Concurrency Control
- Richardson Maturity Model
- SOAP and WSDL
 - *Introduction to SOAP*
 - *WSDL*
 - *WS-Addressing*

Web Service Architecture

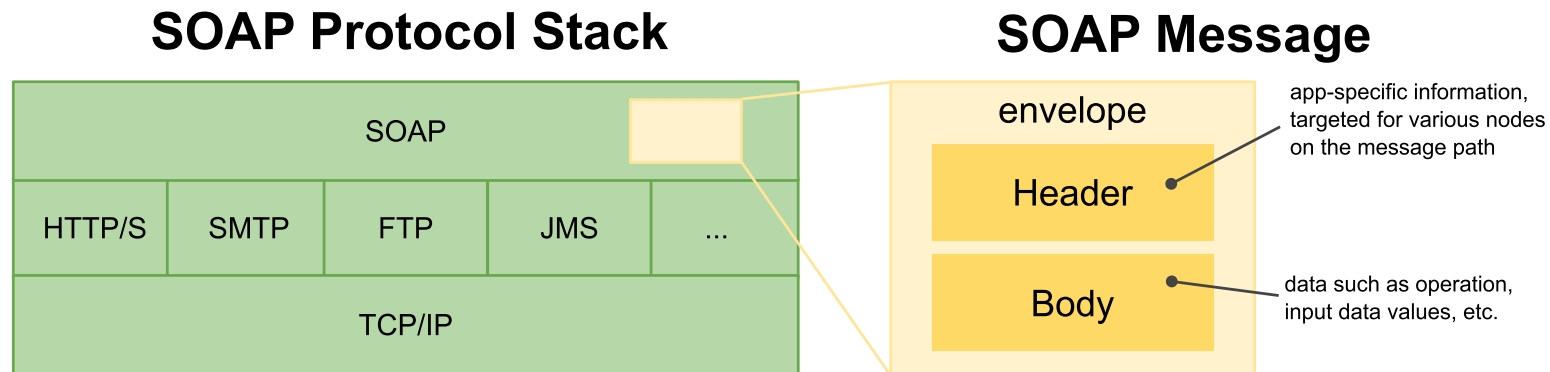
- WSDL, SOAP and UDDI



- *Realization of SOA*
- *Message-Oriented view*
 - *SOAP messaging (header, body)*
 - *types of messages – input, output, fault*

SOAP Protocol

- SOAP defines a messaging framework



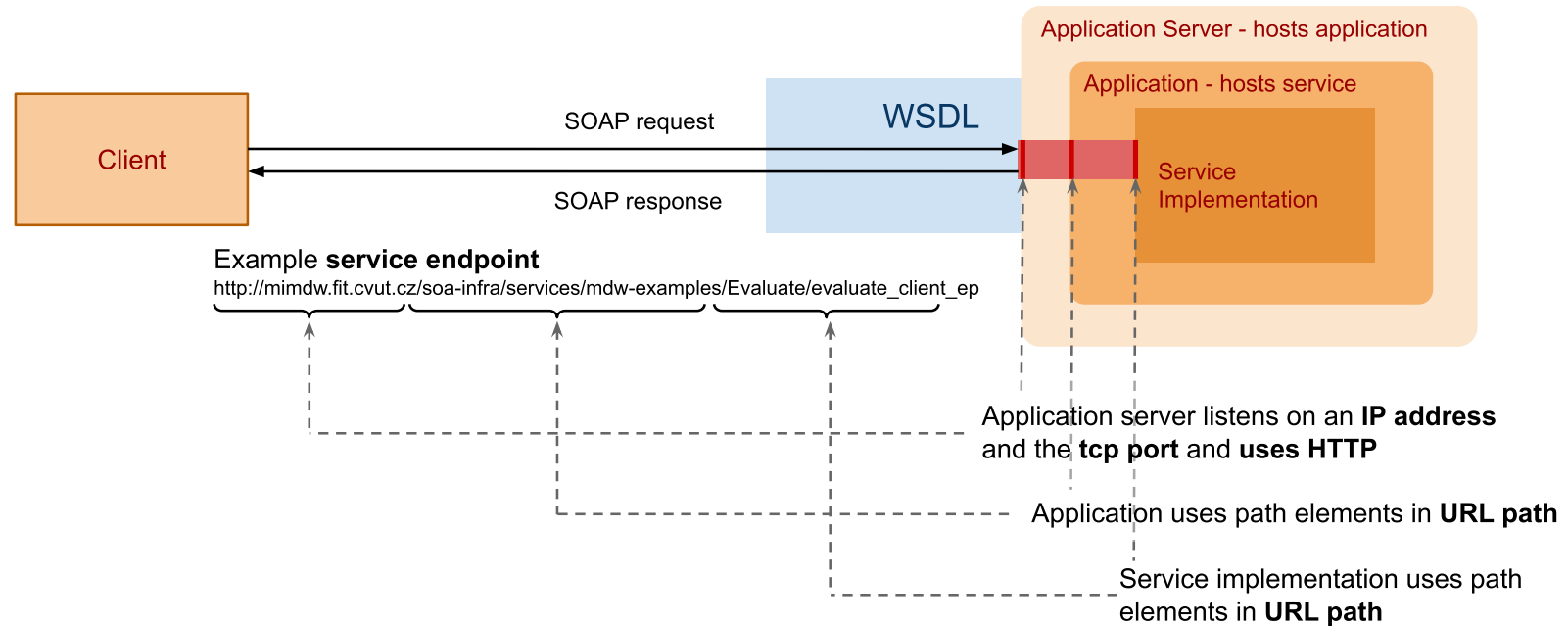
- *XML-based protocol*
- *a layer over transport protocols*
 - *binding to HTTP, SMTP, JMS, ...*
- *involves multiple nodes (message path)*
 - *sender, receiver, intermediary*

SOAP Message

- Envelope
 - *A container of a message*
- Header
 - *Metadata – describe a message, organized in header blocks*
 - *routing information*
 - *security measures implemented in the message*
 - *reliability rules related to delivery of the message*
 - *context and transaction management*
 - *correlation information (request and response message relation)*
 - *WS extensions (WS-*) utilize the message header*
- Body (payload)
 - *Actual contents of the message, XML formatted*
 - *Contains also faults for exception handling*
- Attachment
 - *Data that cannot be serialized into XML such as binary data*

Endpoint

- SOAP service endpoint definition



- *Endpoint – a network address used for communication*
- *Communication – request-response, SOAP messages over a communication (application) protocol*
- *Synchronous communication – only service defines endpoint*
- *Asynchronous communication – service and client define endpoints*

Service Invocation Example (1)

- Example service implementation
 - *A service that evaluates an expression*
 - *Uses SOAP over HTTP*
 - *We can use standard HTTP tools to invoke the service*

- SOAP request message

`evaluate-input.xml`

```
1 <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
2   <soap:Body>
3     <ns1:evaluateRequest
4       xmlns:ns1="http://xmlns.oracle.com/mdw_examples/Evaluate/evaluate"
5       <ns1:x>12</ns1:x>
6       <ns1:y>18</ns1:y>
7     </ns1:evaluateRequest>
8   </soap:Body>
9 </soap:Envelope>
```

- Invoking the service using `curl`

```
1 curl -s -X POST --header "Content-Type: text/xml; charset=UTF-8" \
2 --header "SOAPAction: \"evaluate\"" --data @evaluate-input.xml \
3 http://mimdw.fit.cvut.cz/soa-infra/services/mdw-examples/Evaluate/evaluate_client_
```

Service Invocation Example (2)

- Invocation result

```
1  * About to connect() to mimdw.fit.cvut.cz port 80 (#0)
2  *   Trying 147.32.233.55... connected
3  * Connected to sb.vitvar.com (147.32.233.55) port 80 (#0)
4  > POST /soa-infra/services/mdw-examples/Evaluate/evaluate_client_ep HTTP/1.1
5  > User-Agent: curl/7.19.7 (x86_64-redhat-linux-gnu) libcurl/7.19.7 NSS/3.14.0.0 z1
6  > Host: mimdw.fit.cvut.cz
7  > Accept: */*
8  > Content-Type: text/xml; charset=UTF-8
9  > SOAPAction: "evaluate"
10 > Content-Length: 302
11 >
12 } [data not shown]
13 < HTTP/1.1 200 OK
14 < Date: Sun, 17 Nov 2013 11:24:59 GMT
15 < Server: Oracle-Application-Server-11g
16 < Content-Length: 569
17 < X-ORACLE-DMS-ECID: 004upqiWhdD0zkwVlybQ8A0005uX0004Y^
18 < SOAPAction: ""
19 < X-Powered-By: Servlet/2.5 JSP/2.1
20 < Content-Type: text/xml; charset=UTF-8
21 < Content-Language: en
```

Service Invocation Example (3)

- SOAP response message

```
1  <?xml version="1.0"?>
2  <env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
3      xmlns:wsa="http://www.w3.org/2005/08/addressing">
4      <env:Header>
5          <wsa:MessageID>urn:E42018C04F7A11E3BFD5D1953058407C</wsa:MessageID>
6      </env:Header>
7      <env:Body>
8          <evaluateResponse
9              xmlns="http://xmlns.oracle.com/mdw_examples/Evaluate/evaluate">
10             <result>30</result>
11          </evaluateResponse>
12      </env:Body>
13  </env:Envelope>
```

Client Implementation

- WSDL – Web Service Description Language
 - *definitions for the client to know how to communicate with the service*
 - *which operations it can use*
 - *data formats for input (request), output (response) and fault messages*
 - *how to serialize the data as payloads of a communication protocol (binding)*
 - *where the service is physically present on the network*
- Clients' environments
 - *Clients implemented in a language such as Java*
 - *Tools to generate service API for the client, e.g. WSDL2Java*
 - *Can be written manually too, e.g. our example in bash*
 - *Clients reside on the middleware, e.g. on an Enterprise Service Bus*
 - *They provide added values in end-to-end communication, proxy services, SOAP intermediaries*

Overview

- HATEOAS
- Caching, Revalidation, Concurrency Control
- Richardson Maturity Model
- SOAP and WSDL
 - *Introduction to SOAP*
 - *WSDL*
 - *WS-Addressing*

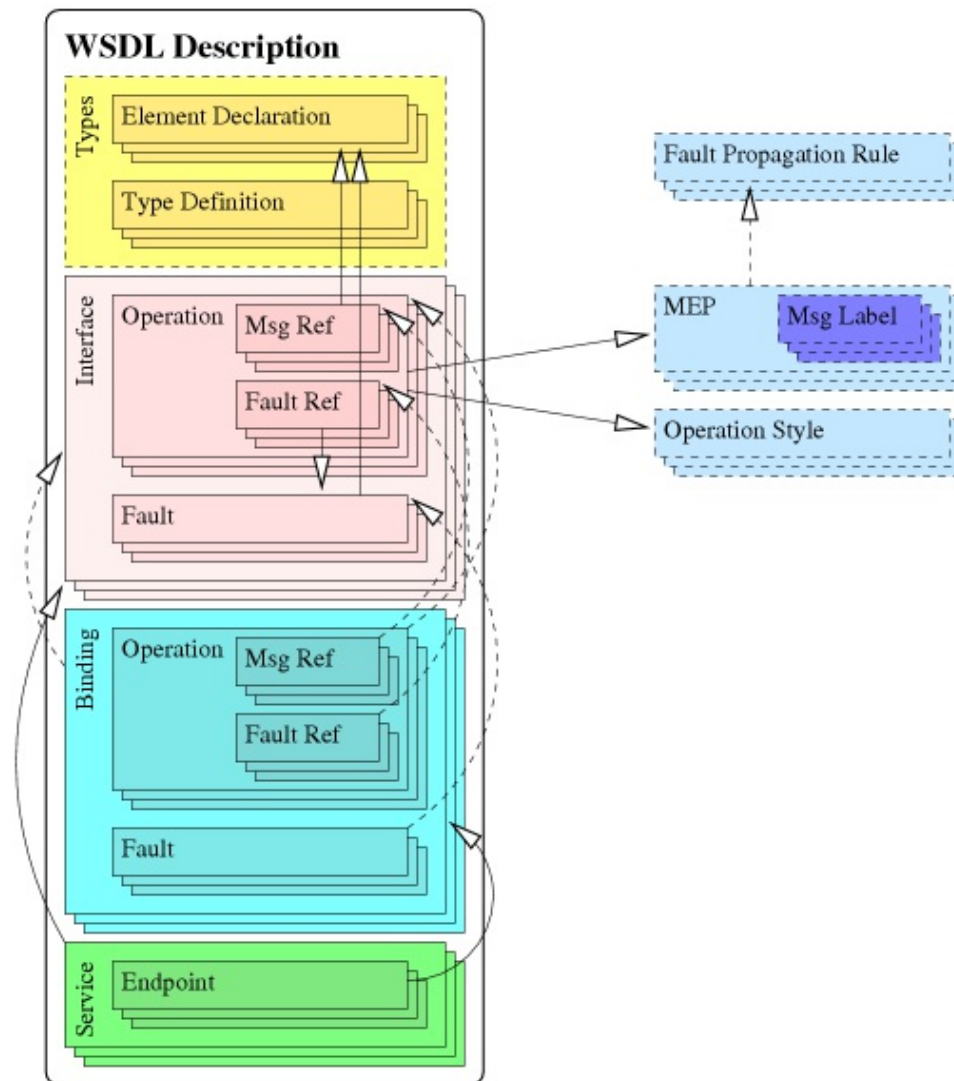
Specifications

- WSDL = Web Service Description Language
 - *A standard that allows to describe Web services explicitly (main aspects)*
 - *A contract between a requester and a provider*
- Specifications
 - *WSDL 1.1 – still widely used*
 - *Web Service Description Language 1.1* [!\[\]\(cd3e54d951a9fb854f48e4697cf550f9_img.jpg\)](#)
 - *WSDL 2.0 – An attempt to address several issues with WSDL 1.1*
 - *SOAP vs. REST, naming, expressivity*
 - *WSDL 2.0 Primer (part 0)* [!\[\]\(cc729e263f29c0a76fbdc4cfe67fceb0_img.jpg\)](#)
 - *WSDL 2.0 Core Language (part 1)* [!\[\]\(90d36d418f8f7ab67431ba2525e00a5e_img.jpg\)](#)

WSDL Overview and WSDL 1.1 Syntax

- Components of WSDL
 - Information model (**types**)
 - Element types, message declarations (XML Schema)
 - Set of operations (**portType**)
 - A set of operations is "interface" in the WSDL terminology
 - operation name, input, output, fault
 - Binding (**binding**)
 - How messages are transferred over the network using a concrete transport protocol
 - Transport protocols: HTTP, SMTP, FTP, JMS, ...
 - Endpoint (**service**)
 - Where the service is physically present on the network
- Types of WSDL documents
 - **Abstract WSDL** – only information model and a set of operations
 - **Concrete WSDL** – everything, a concrete service available in the environment

WSDL Components and Dependencies



Overview

- HATEOAS
- Caching, Revalidation, Concurrency Control
- Richardson Maturity Model
- SOAP and WSDL
 - *Introduction to SOAP*
 - *WSDL*
 - *WS-Addressing*

Overview

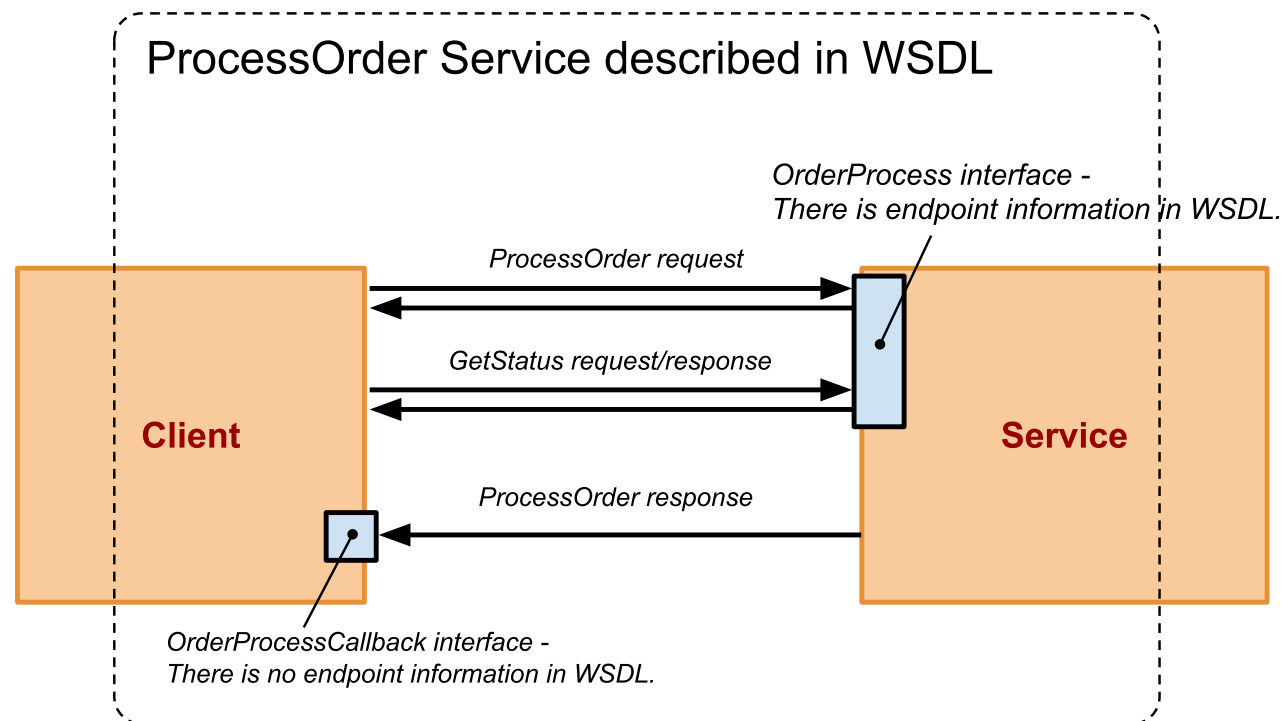
- WS-Addressing
 - *W3C Recommendation, May 2006* [🔗](#)
 - *A transport-independent mechanisms for web services to communicate addressing information*
 - *WSDL describes WS-Addressing as a policy attached to a WSDL binding*

```
1 <binding name="OrderProcessBinding" type="op:OrderProcess">
2   <soap:binding transport="http://schemas.xmlsoap.org/soap/http"/>
3   <PolicyReference xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
4     URI="#wsaddr_policy" wsdl:required="false"/>
```

- Two main purposes
 1. *Asynchronous communication*
 - *Client sends an endpoint where the server should send a response asynchronously*
 2. *Relating interactions to a conversation*
 - *Client and service communicate conversation ID*

Order Processing Example

- Asynchronous communication via callback, steps:
 - *Client submits an order request*
 - *Service starts processing of the order (CRM, OMS, back-office)*
 - *Client can retrieve the order status*
 - *Service responds asynchronously with an order response message*



Interface Example (1)

- Order process complex conversation
 1. The client invokes **processOrder**.
 2. The service responds back **synchronously** with order status.
 3. The client gets the status of order processing by invoking synchronous **getStatus** operation (this can be invoked several times).
 4. The service responds back **asynchronously** by invoking **processOrderResponse** – callback on client's interface
- Interface implemented by the order process service
 - **getStatus** operation must be executed in the same **conversation** as **processOrder** operation

```
1 <portType name="OrderProcess">
2   <operation name="processOrder">
3     <input message="op:OrderProcessRequestMessage"/>
4     <output message="op:OrderStatusResponseMessage"/>
5   </operation>
6   <operation name="getStatus">
7     <input message="op:OrderStatusRequestMessage"/>
8     <output message="op:OrderStatusResponseMessage"/>
```

Interface Example (2)

- Interface implemented by the client

```
1 <portType name="OrderProcessCallback">
2   <operation name="processOrderResponse">
3     <input message="op:OrderProcessResponseMessage"/>
4     <fault message="op:OrderProcessFaultMessage"/>
5   </operation>
6 </portType>
```

ProcessOrder Request Message

- Client sends process order request – **processOrder**
 - it sends addressing information where the client listens for the callback
 - it sends conversation ID (message ID) to start the conversation on the server

```
1 > POST /soa-infra/services/mdw-examples/ProcessOrder/orderprocess_client_ep HTTP/1.1
2 > Host: mimdw.fit.cvut.cz
3 > Content-Type: text/xml;charset=UTF-8
4 > SOAPAction: "processOrder"
5 > Content-Length: 810
6
7 <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
8   xmlns:ord="http://mimdw.fit.cvut.cz/mdw-examples/cdm/order">
9   <soap:Header xmlns:wsa='http://www.w3.org/2005/08/addressing'>
10     <wsa:ReplyTo>
11       <wsa:Address>http://192.168.94.110:2233/path/to/service</wsa:Address>
12     </wsa:ReplyTo>
13     <wsa:MessageID>urn:AXYYBA00531111E3BFACA780A7E5AF64</wsa:MessageID>
14   </soap:Header>
15   <soap:Body>
16     <ord:Order>
17       <ord:CustomerId>1</ord:CustomerId>
18       <ord:LineItems>
19         <ord:item>
20           <ord:label>Apple MacBook Pro</ord:label>
21           <ord:action>ADD</ord:action>
22         </ord:item>
23       </ord:LineItems>
24     </ord:Order>
25   </soap:Body>
26 </soap:Envelope>
```


GetStatus Request Message

- Client sends get status request – **getStatus**
 - after it invokes **processOrder** with conversation ID (message ID)
 - it uses the same conversation ID for get status request too
 - the request will be processed by the running service instance

```
1 > POST /soa-infra/services/mdw-examples/ProcessOrder/orderprocess_client_ep HTTP/1.1
2 > Host: mimdw.fit.cvut.cz
3 > Content-Type: text/xml; charset=UTF-8
4 > SOAPAction: "getStatus"
5 > Content-Length: 472
6
7 <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
8   <soap:Header xmlns:wsa="http://www.w3.org/2005/08/addressing">
9     <wsa:RelatesTo>urn:AXYYBA00531111E3BFACA780A7E5AF64</wsa:RelatesTo>
10   </soap:Header>
11   <soap:Body>
12     <ns1:StatusRequest
13       xmlns:ns1="http://mimdw.fit.cvut.cz/mdw_examples/ProcessOrder/OrderProcess
14     <ns1:process-id>18a9baec2d5ac0a2:64d155de:1425c4185f1:-7ff2</ns1:process-i
15     </ns1:StatusRequest>
16   </soap:Body>
17 </soap:Envelope>
```