

# Web 2.0

## Lecture 3: REST Architecture 2

**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/w20>



Evropský sociální fond  
Praha & EU: Investujeme do vaší budoucnosti

Modified: Wed Apr 05 2017, 08:03:54  
Humla v0.3

# REST Core Principles

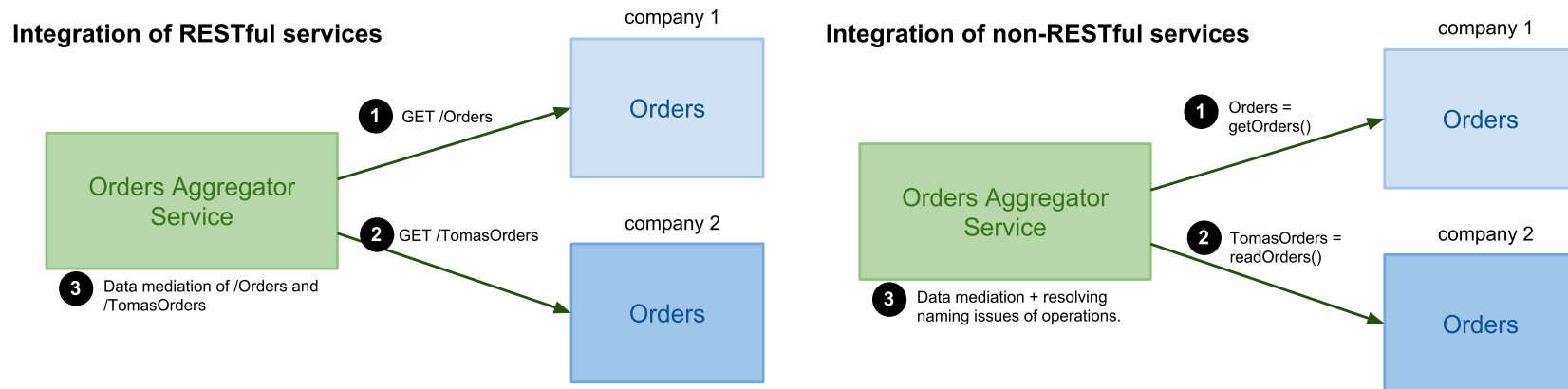
- REST architectural style defines constraints
  - *if you follow them, they help you to achieve a good design, interoperability and scalability.*
- Constraints
  - *Client/Server*
  - *Statelessness*
  - *Cacheability*
  - *Layered system*
  - ***Uniform interface***
- Guiding principles
  - *Identification of resources*
  - *Representations of resources and self-descriptive messages*
  - *Hypermedia as the engine of application state (HATEOAS)*

# Overview

- Uniform Interface
  - *Basic operations*
  - *Handling Errors*
  - *Advanced Design Issues*
- Selected Protocols
- Selected Extensions

# Uniform Interface

- Uniform interface = finite set of operations
  - Resource manipulation
    - *CRUD* – Create (*POST/PUT*), Read (*GET*), Update (*PUT/PATCH*), Delete (*DELETE*)
  - operations are not domain-specific
    - For example, **GET /orders** and not **getOrders()**
    - This reduces complexity when solving interoperability
- Integration issues examples



# Safe and Unsafe Operations

- Safe operations
  - *Do not change the resource state*
  - *Usually "read-only" or "lookup" operation*
  - *Clients can cache the results and refresh the cache freely*
- Unsafe operations
  - *May change the state of the resource*
  - *Transactions such as buy a ticket, post a message*
  - *Unsafe does not mean dangerous!*
- Unsafe interactions and transaction results
  - **POST** response may include transaction results
    - *you buy a ticket and submit a purchase data*
    - *you get transaction results*
    - *and you cannot bookmark this..., why?*
  - *Should be referable with a persistent URI*

# Idempotence

- Idempotent operation
  - *Invoking a method on the same resource always has the same effect*
  - *Operations GET, PUT, DELETE*
- Non-idempotent operation
  - *Invoking a method on the same resource may have different effects*
  - *Operation POST*
- Effect = a state change
  - *recall the effect definition in MDW*

# Overview

- Uniform Interface
  - *Basic operations*
  - *Handling Errors*
  - *Advanced Design Issues*
- Selected Protocols
- Selected Extensions

# GET

- Reading

- **GET** *retrieves a representation of a state of a resource*

- > GET /orders HTTP/1.1

- > Accept: application/xml

- < HTTP/1.1 200 OK

- < Content-Type: application/xml

- <

- < ...resource representation in xml...

- *It is read-only operation*

- *It is **safe***

- *It is **idempotent***

- **GET** *retrieves different states over time but the effect is always the same, cf. **resource state** hence it is idempotent.*

- *Invocation of **GET** involves content negotiation*



# PUT

- Updating or Inserting

- **PUT** *updates a representation of a state of a resource or inserts a new resource*

```
> PUT /orders/4456 HTTP/1.1  
> Content-Type: application/xml  
>  
> <order>...</order>
```

```
< HTTP/1.1 CODE
```


- *where CODE is:*

- **200 OK** or **204 No Content** *for updating: A resource with id 4456 exists, the client sends an updated resource*

- **201 Created** *for inserting: A resource **does not exist**, the client generates the id 4456 and sends a representation of it.*

- *It is **not safe** and it is **idempotent***

# POST

- Inserting
  - **POST** *inserts a new resource*
  - *A server generates a new resource ID, client only supplies a content and a resource URI where the new resource will be inserted.*
    - > POST /orders HTTP/1.1
    - > Content-Type: application/xml
    - >
    - > <**order**>...</**order**>
    - < HTTP/1.1 201 Created
    - < Location: /orders/4456
  - *It is **not safe** and it is **not idempotent***
  - *A client may "suggest" a resource's id using the **Slug** header*
    - *Defined in AtomPub protocol* 

# DELETE

- Deleting
  - **DELETE** *deletes a resource with specified URI*
    - > **DELETE** /orders/4456 HTTP/1.1
    - < HTTP/1.1 CODE
  - *where CODE is:*
    - **200 OK**: *the response body contains an entity describing a result of the operation.*
    - **204 No Content**: *there is no response body.*
  - *It is **not safe** and it is **idempotent***
    - *Multiple invocation of **DELETE** /orders/4456 has always the same effect – the resource /orders/4456 does not exist.*

## Other

- HEAD
  - same as **GET** but only retrieves *HTTP headers*
  - It is *safe* and *idempotent*
- OPTIONS
  - queries the resource for resource configuration
  - It is *safe* and *idempotent*

# Overview

- Uniform Interface
  - *Basic operations*
  - *Handling Errors*
  - *Advanced Design Issues*
- Selected Protocols
- Selected Extensions

# Types of Errors

- Client-side – status code **4xx**
  - **400 Bad Request**
    - *generic client-side error*
    - *invalid format, such as syntax or validation error*
  - **404 Not Found**
    - *server can't map URI to a resource*
  - **401 Unauthorized**
    - *wrong credentials (such as user/pass, or API key)*
    - *the response contains **WWW-Authenticate** indicating what kind of authentication the service accepts*
  - **405 Method Not Allowed**
    - *the resource does not support the HTTP method the client used*
    - *the response contains **Allow** header to indicate methods it supports*
  - **406 Not Acceptable**
    - *so many restrictions on acceptable content types (using **Accept-\***)*
    - *server cannot serialize the resource to requested content types*

## Types of Errors (Cont.)

- Server-side – status code **5xx**
  - **500 Internal Server Error**
    - *generic server-side error*
    - *usually not expressive, logs a message for system admins*
  - **503 Service Not Available**
    - *server is overloaded or is under maintenance*
    - *the response contains **Retry-After** header*

# Use of Status Codes

- Service should respect semantics of status codes!

```
> GET /orders HTTP/1.1  
> Accept: application/json
```

```
< HTTP/1.1 200 OK  
< Content-Type: application/json  
<  
< { "error" :  
<   { "error_text" :  
<     "you do not have rights to access this resource " }  
< }
```

- *Client must understand the semantics of the response.*
- *This breaks loose coupling and reusability service principles*
- *The response should be:*

```
< HTTP/1.1 401 Unauthorized  
< ...  
  
< ...optional text describing the error...
```



# Overview

- Uniform Interface
  - *Basic operations*
  - *Handling Errors*
  - *Advanced Design Issues*
- Selected Protocols
- Selected Extensions

# Respect HTTP Semantics

- Do not overload semantics of HTTP methods
  - For example, **GET** is read-only method and idempotent
  - REST Anti-pattern:  
**GET /orders/?add=new\_order**
    - This is not REST!
    - This breaks both safe and idempotent principles
- Consequences
  - Result of **GET** can be cached by proxy servers
  - They can revalidate their caches freely
  - You can end up with new entries in your storage without you knowing!
- The same is true for other methods

# Change Order Status

- **status** property of `/orders/{order-id}` resource
  - *reflects a state of the process*
  - *No need to use a stateful service, state is communicated through the order representation*
- How do you implement a canceling an order?
  - *You can delete it using **DELETE***
  - *But you may want to cancel it in order to:*
    - *maintain a list of canceled orders*
    - *have a possibility to "roll-back" canceled orders*

## DELETE to cancel

- A bad solution to cancel the order
  - *to cancel with DELETE*  
`DELETE /orders/3454/?cancel=true`
  - *you overload the meaning of DELETE*
  - *you violate the uniform interface principle*
- Always ask a question:
  - *Is the operation a state of the resource?*
  - *if yes, the operation should be:*
    - *modeled within the data format*
    - *or as a separated resource (sub-resource)*
- No verbs in **path** and **query** components!
  - `/cancelOrder`, `/orders/{order-id}/?action=delete`, etc.
  - *Verbs in URIs indicate that a resource is actually an operation!*

# PUT to cancel

- A RESTful solution to cancel an order

1. *first, have an order's status*

- *as part of the Order representation format*

- *we extend "open" and "close" with "cancel"*

2. *Use PUT to cancel an order*

```
1 > PUT /orders/{order-id}
2 > Content-Type: application/json
3 >
4 > { "status" : "cancel" }
5
6 < HTTP/1.1 204 No Content
```

- Clean-up all cancelled orders

- *you can have a resource "all valid orders": /orders/valid (~ all orders that are not canceled)*

- **GET /orders/valid** will return all non-canceled orders

- **POST /orders/valid** will purge all cancelled orders

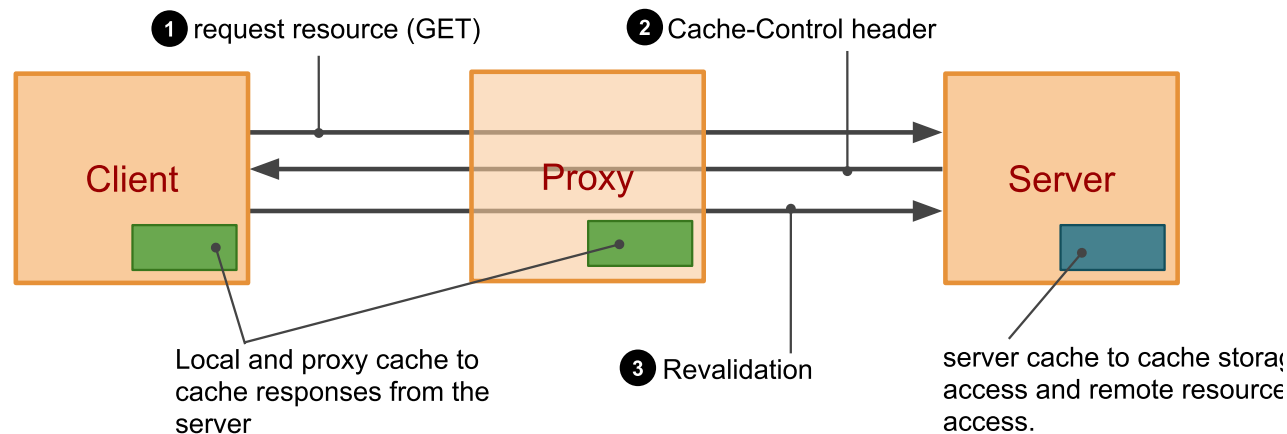
# Overview

- Uniform Interface
- Selected Protocols
  - *Caching and Revalidation*
  - *Concurrency Control*
- Selected Extensions

# 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** – *can cache but should not store persistently. When a client restarts, content is lost*
  - **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, no-store, max-age=200
< Last-Modified: Sun, 7 Nov 2011, 09:40 CET
<
< ...data...
```

— *only client can cache, must not be stored on the disk, 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, no-store, 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, no-store, 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, no-store, 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

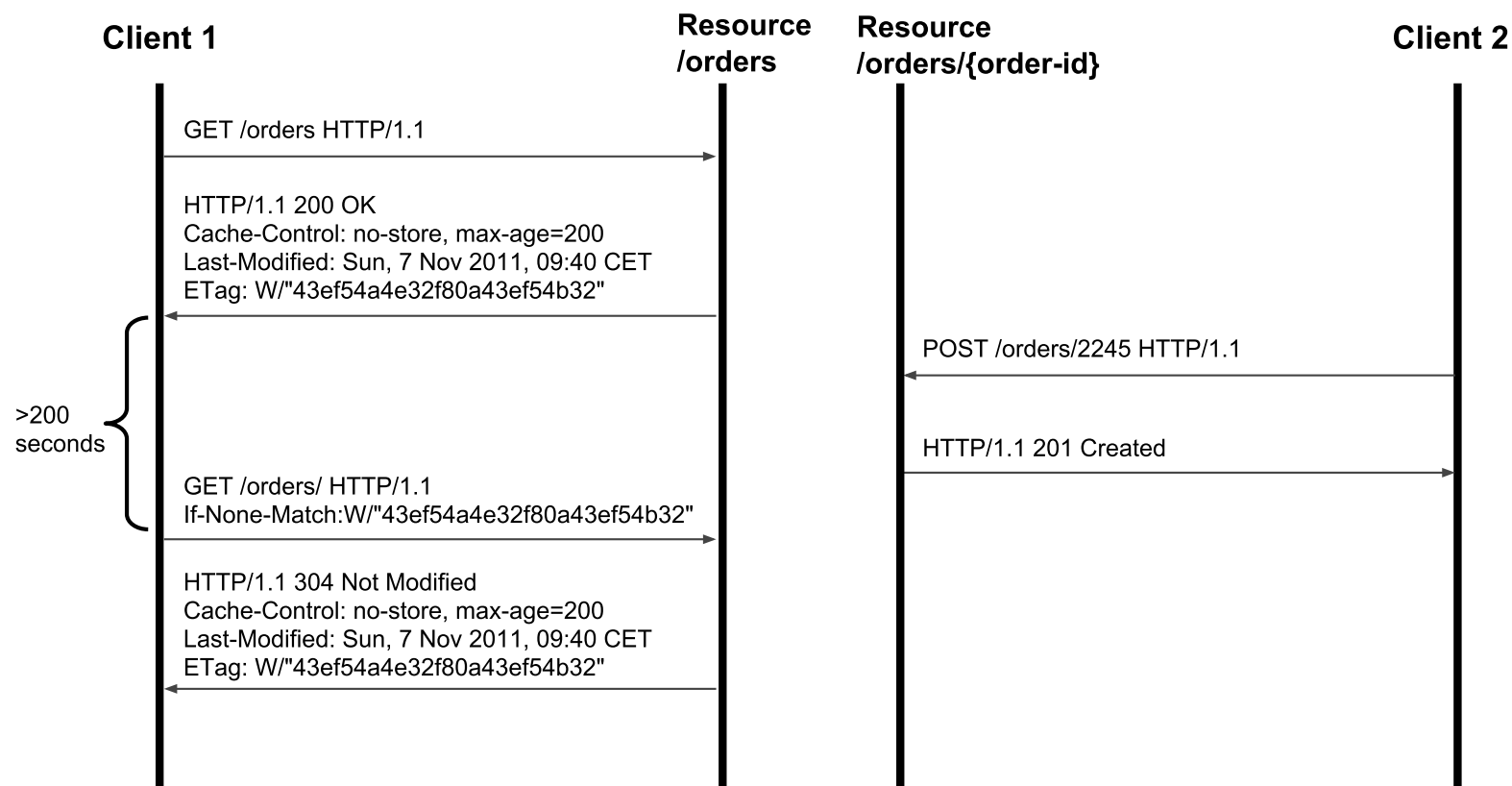
```
1  {  
2    "orders" :  
3      [  
4        { "id"      : 2245,  
5          "customer" : "Tomas",  
6          "descr"    : "Stuff to build a house.",  
7          "items"    : [...] },  
8        { "id"      : 5546,  
9          "customer" : "Peter",  
10         "descr"    : "Things to build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



# Overview

- Uniform Interface
- Selected Protocols
  - *Caching and Revalidation*
  - *Concurrency Control*
- Selected Extensions

# 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

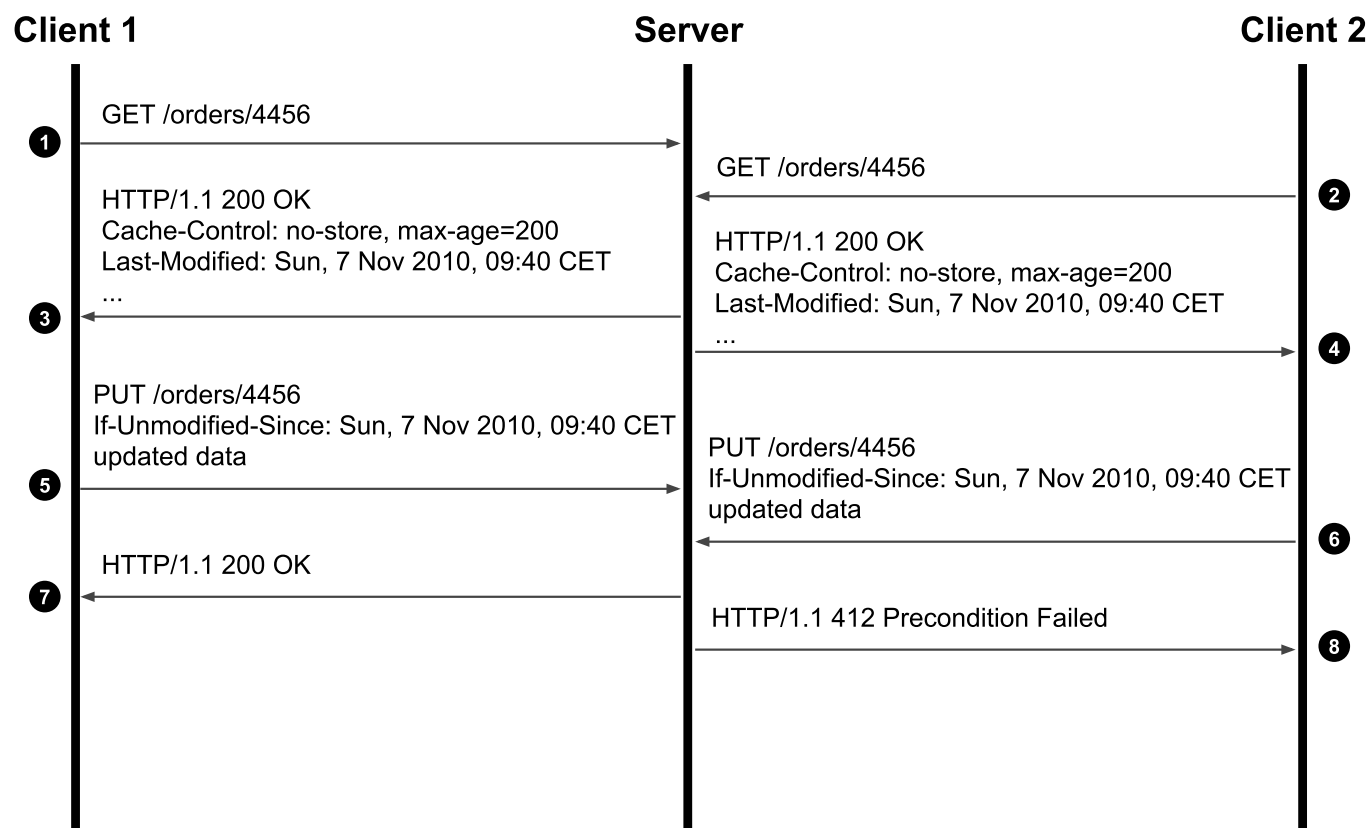
- Response to conditional PUT:

- 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

- Uniform Interface
- Selected Protocols
- Selected Extensions

# GData Protocol: Partial Update

- **PATCH** HTTP Method
  - *IETF specification, see PATCH Method for HTTP* [🔗](#)
  - *Add, modify or delete selected elements of an entry*
- Examples
  - *To delete a description element and add a new title element*
  - **gd:fields** uses partial response syntax

```
1 PATCH /myFeed/1/1/  
2 Content-Type: application/xml  
3  
4 <entry xmlns='http://www.w3.org/2005/Atom'  
5     xmlns:gd='http://schemas.google.com/g/2005'  
6     gd:fields='description'>  
7     <title>New title</title>  
8 </entry>
```

- Rules
  - *Fields not already present are added*
  - *Non-repeating fields already present are updated*
  - *Repeating fields already present are appended*

# GData Protocol: Entity Tags

- Resource Versioning
  - *Conditional GET and PUT (concurrency control)*
    - See *Lecture 4 – scalability*
  - *Etags on atom and entry elements*

- Example

```
1  GData-Version: 2.0
2  ETag: W/"C0QBRXcycSp7ImA9WxRVFuk."
3  ...
4  <?xml version='1.0' encoding='utf-8'?>
5  <feed xmlns='http://www.w3.org/2005/Atom'
6      xmlns:gd='http://schemas.google.com/g/2005'
7      gd:etag='W/"C0QBRXcycSp7ImA9WxRVFuk."'>
8      ...
9      <entry gd:etag=' "CUUEQX47eCp7ImA9WxRVEkQ." '>
10         ...
11     </entry>
12 </feed>
```

- *It is possible to do a conditional GET/PUT on the entry by using the ETag "CUUEQX47eCp7ImA9WxRVEkQ."*

# GData Protocol: HTTP Methods Overriding

- Firewall restrictions
  - *Some firewall configurations do not allow to send HTTP request other than GET and POST*

- HTTP methods overriding through **POST**

X-HTTP-Method-Override: PUT  
X-HTTP-Method-Override: DELETE  
X-HTTP-Method-Override: PATCH

- Example

```
1 | POST /myfeed/1/1/  
2 | X-HTTP-Method-Override: PATCH  
3 | Content-Type: application/xml  
4 | ...
```