

# Web 2.0

## Lecture 3: Uniform Interface

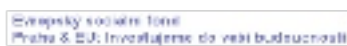
doc. Ing. Tomáš Vitvar, Ph.D.

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>



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

## REST Core Principles

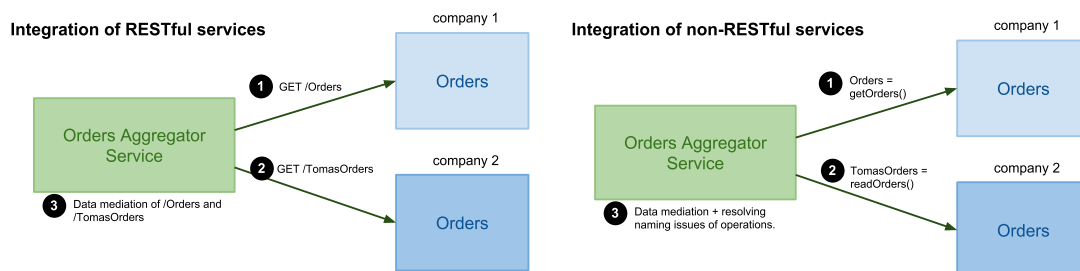
- REST architectural style defines constraints
  - *if you follow them, they help you to achieve a good design, interoperability and scalability.*
- Constraints
  - *Client/Server*
  - *Statelessness*
  - *Cacheability*
  - *Layered system*
  - ***Uniform interface***
- Guiding principles
  - *Identification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

## Overview

- **Uniform Interface**
  - *Basic operations*
  - *Handling Errors*
- Asynchronous Communication
- Implementing a RESTful Service
- Advanced Design Issues

## 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*
- Asynchronous Communication
- Implementing a RESTful Service
- Advanced Design Issues

## 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*
- Asynchronous Communication
- Implementing a RESTful Service
- Advanced Design Issues

## 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...

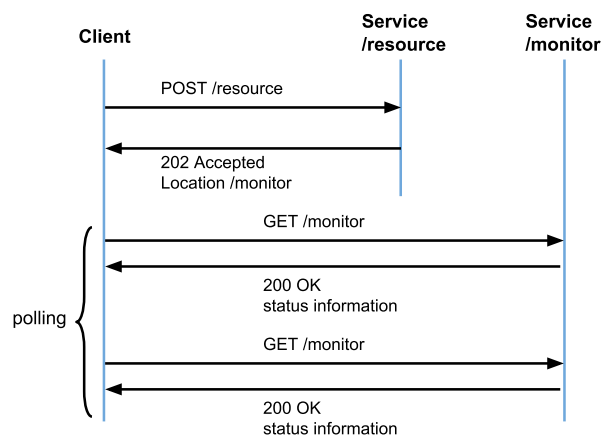


## Overview

- Uniform Interface
- **Asynchronous Communication**
- Implementing a RESTful Service
- Advanced Design Issues

## Asynchronous Communication

- Recall asynchronous communication from MDW
- Asynchronous communication in HTTP
  - *Server cannot establish a connection, always clients need to*  
→ *clients are browsers behind firewalls*



## Asynchronous and Polling/Pushing

- Submit request for processing
  - Always through HTTP request and **202 Accepted** response and **Location** header with a monitor resource
  - Methods: **PUT, POST, DELETE**
- Getting the status from the monitor resource
  - **polling** – a client periodically checks for changes via **GET**
    - Most natural solution, not a real-time solution
  - **pushing** – a server pushes changes back to the client
    - Part of real-time Web efforts
  - More details in *Lecture 8: Protocols for the Realtime Web*

## Overview

- Uniform Interface
- Asynchronous Communication
- **Implementing a RESTful Service**
  - *Basic Implementation*
- Advanced Design Issues

## Service Description

- Example service: Oder processing

<https://github.com/tomvit/w20/tree/master/examples/restful-service>

```
1  * the service provides three operations: 'open', 'add', 'close'
2  * operation 'open' opens the order
3    - input: none
4    - output: text informing that the order was opened
5
6  * operation 'add' adds an item to the order
7    - input: an item name, the syntax is [0-9A-Za-z\-\_]+
8    - output: text informing that the item was added to the order
9
10 * operation 'close' closes the order and returns all items in the order
11   - input: none
12   - output: list of all items previously added to the order
13
14 * the public process is: S0--open--S1, S1--add--S1, S1--close--S0, where
15   S0, S1 are states such that S0 = order is closed, and S1 = order is opened.
16
17 * protocol is HTTP, RESTful service
18   running at ec2.vitvar.com, tcp/9900
```

- Basic steps to define a RESTful service

1. *identify resources and URIs*
2. *specify resources' representations*
3. *define service operations (methods and status codes)*

## Overview

- Uniform Interface
- Asynchronous Communication
- Implementing a RESTful Service
  - *Basic Implementation*
- Advanced Design Issues

## Resources, URIs and Representations

- There are three resources
  - Resource `/orders` is a container of all orders
  - Resource `/orders/{order-id}` is an order with resource id `order-id`.
  - Resource `/orders/{order-id}/{item-id}` is an item that belongs to the order `order-id` and that has a resource id `item-id`.
- Structure
  - `/orders`
    - list of all orders
  - `/orders/{order-id}`
    - status, order id, list of all items in the order
  - `/orders/{order-id}/{item-id}`
    - item id, name, price
- Resource representations
  - We define representations in JSON

## Open Order

- To open an order
  - Insert a new order to `/orders` using **POST**
  - Set the new order's status to "open"

```
45 | if (method == "POST") { // open order
46 |     // create a new order object
47 |     var order = {
48 |         id : storage.getOrderSeqId(),
49 |         status : "open",
50 |         items : []
51 |     };
52 |
53 |     // add the order to the list of orders and return the result
54 |     storage.orders.push(order);
55 |     return {
56 |         status : "201", // created
57 |         headers : { Location: "http://" + host + "/orders/" + order.id }
58 |     };
59 | }
```

- `storage.getOrderSeqId()` returns the order ID
- `storage.orders` (line 37) is an array of all orders in a storage

## Add Item to Order

- To add an item to the order
  - Insert a new item to the order `/orders/{order-id}` using POST

```
74  if ((id = uri.match("^/orders/([0-9]+)$")) {
75      if (method == "POST") {
76          // get the order object
77          var order = storage.getOrder(id[1]);
78          if (order && order.status == "open") {
79              // get the item object from the request data and set it's id
80              var item = JSON.parse(data);
81              item.id = storage.getItemSeqId(order);
82
83              // store the item in the order and return the result
84              // location is the URI of the newly created item
85              order.items.push(item);
86              return {
87                  status : "201", // created
88                  headers : { Location: "http://" + host + "/orders/" +
89                             order.id + "/" + item.id }
90              };
91          } else
92              // not found or bad request (the order is not open)
93              return { status : (order ? "400" : "404") };
94      }
```

## Close Order

- To close an order
  - Update the status of the order `/orders/{order-id}` using PUT

```
97  // update the order status
98  if (method == "PUT") {
99      // get the order object
100     var order = storage.getOrder(id[1]);
101     if (order && order.status == "open") {
102         var o2 = JSON.parse(data);
103
104         // check for the valid status
105         if (o2.status && (s = o2.status.match("(close)"))) {
106             order.status = s[1];
107             return {
108                 status : "204", // no content
109             };
110         } else
111             // bad request
112             return { status : "400" };
113     } else
114         // not found or bad request (the order is not open)
115         return { status : (order ? "400" : "404") };
116 }
```

## Other Operations

- To get, delete an order and get, delete and update an item
  - Delete an order `/orders/{order-id}` using **DELETE**
  - Get an order's item `/orders/{order-id}/{item-id}` using **GET**
  - Update an order's item `/orders/{order-id}/{item-id}` using **PUT**
  - Delete an order's item `/orders/{order-id}/{item-id}` using **DELETE**
- Other methods are not allowed
  - Send **405 Not Allowed** status with **Allow** header to indicate which methods are allowed on a resource

```
1  if (method != "GET" && method != "PUT" && method != "POST" && method != "DELETE")
2      return {
3          status: "405", // method not allowed
4          headers : { "Allow" : "GET, PUT, POST, DELETE" }
5      };
```



### Task

- Implement the remaining methods listed above

## Testing

- Test the service using a bash script **test.sh**

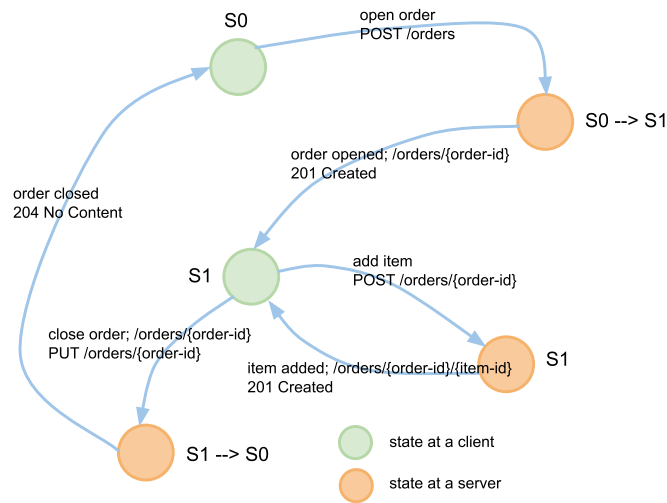
```
3  # set your server address here
4  SERVER_URL="http://127.0.0.1:8881"
5
6  # add an order and get the new location; assume everything went ok
7  order_uri=$(curl -v -X POST $SERVER_URL/orders 2>&1 | \
8      awk '/Location/ {print $3}')
9
10 # remove whitespace
11 order_uri=${order_uri//[:space:]}
12
13 echo "New order with URI $order_uri has been created"
14
15 # add items to the order
16 curl -X POST -d "{ \"name\" : \"socks\", \"price\" : 5 }" $order_uri
17 curl -X POST -d "{ \"name\" : \"t-shirt\", \"price\" : 20 }" $order_uri
18 curl -X POST -d "{ \"name\" : \"jumper\", \"price\" : 45 }" $order_uri
19
20 # close the order
21 curl -X PUT -d "{ \"status\" : \"close\" }" $order_uri
22
23 # list the items in the order and the order's status
24 curl $order_uri
```



### Task

- Run service and test it using the **test.sh** script.

## RESTful Public Process



- Note
  - *client, service communicate through metadata and representations*
  - *There is no need for a stateful server*

## Overview

- Uniform Interface
- Asynchronous Communication
- Implementing a RESTful Service
- **Advanced Design Issues**

## 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

# Evaluation

- How "good" is our Order Book service?
  - Analysis of the service by service characteristics (see MDW for details) and HTTP principles.

Principle	+/-	Comment
Loose Coupling	+	Uses standard response codes.
	+	Uses representation of resources and HTTP Location header to implement the public process.
	–	Does not use hypermedia; client needs to construct links for some resources.
	+	Properly models resource URIs and resource IDs; they have hierarchical nature; does not use verbs.
	+	Respects semantics of HTTP methods and extensively uses them.
Reusability	+	Unforeseen clients will likely use the service as the application state is communicated through HTTP.
	–	Only offers one representation format (JSON).
Contracting and Discoverability	–	Does not describe content type nor public process such as by using Internet Media Types.
Composability	+	Does not obstruct composition.
Abstraction	+	Service description can be implemented by various implementation technologies.
Encapsulation	+	Distinguishes interface from implementation, processing logic is not exposed to clients through the interface.