

# Middleware Architectures 1

## Lecture 5: Representational State Transfer

**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>

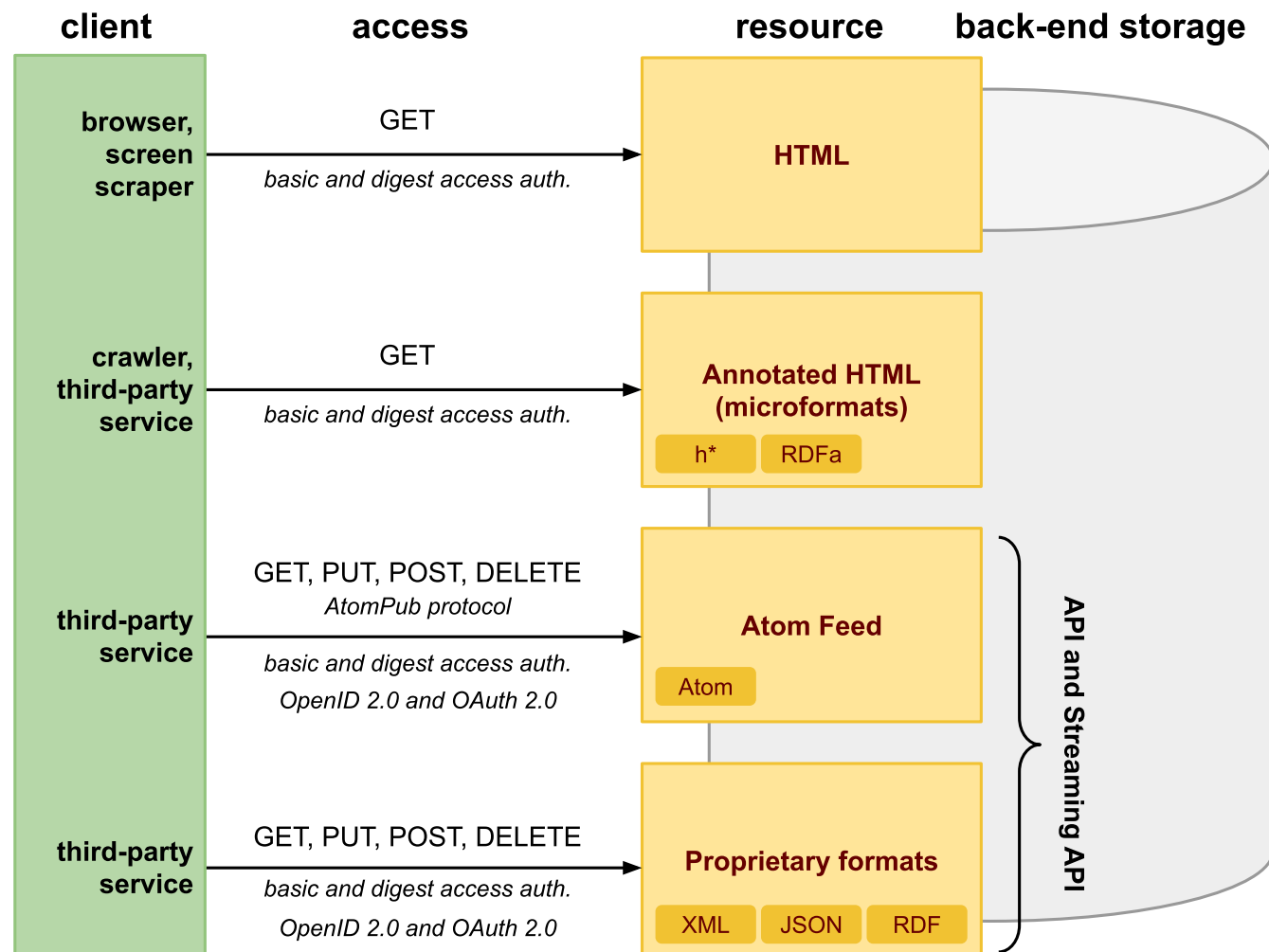


Modified: Sun Nov 22 2020, 21:28:55  
Humla v0.3

# Overview

- Introduction to REST
- Uniform Resource Identifier
- Resource Representation
- Uniform Interface

# Data on the Web



# REST

- REST
  - *Representational State Transfer*
- Architecture Style
  - Roy Fielding – co-author of HTTP
  - He coined REST in his PhD thesis [🔗](#).
    - The thesis abstracts from HTTP technical details
    - HTTP is one of the REST implementation → **RESTful**
    - REST is a leading programming model for Web APIs
- REST (RESTful) proper design
  - people break principles often
  - See REST Anti-Patterns [🔗](#) for some details.
- REST and Web Service Architecture
  - REST is a realization of WSA resource-oriented model

# REST and Web Architecture

- Tim-Berners Lee
  - *"creator", father of the Web*
- Key Principles
  - *Separation of Concerns*
    - *enables independent innovation*
  - *Standards-based*
    - *common agreement, big spread and adoption*
  - *Royalty-free technology*
    - *a lot of open source, no fees*
- Architectural Basis
  - **Identification:** *universal linking of resources using URI*
  - **Interaction:** *protocols to retrieve resources – HTTP*
  - **Formats:** *resource representation (data and metadata)*

# HTTP Advantages

- Familiarity
  - *HTTP protocol is well-known and widely used*
- Interoperability
  - *All environments have HTTP client libraries*
    - *technical interoperability is thus no problem*
    - *no need to deal with vendor-specific interoperability issues*
  - *You can focus on the core of the integration problem*
    - *application (domain, content) interoperability*
- Scalability
  - *you can use highly scalable Web infrastructure*
    - *caching servers, proxy servers, etc.*
  - *HTTP features such as HTTP GET idempotence and safe allow you to use caching*

# 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)*

# Resource

- A resource can be anything such as
  - *A real object: car, dog, Web page, printed document*
  - *An abstract thing such as address, name, etc. → RDF*
- A resource in REST
  - *A resource corresponds to one or more entities of a data model*
  - *A representation of a resource can be conveyed in a message electronically (information resource)*
  - *A resource has an identifier and a representation and a client can apply an access to it*





# Access to a Resource



- Terminology
  - *Client = User Agent*
  - **Dereferencing URI** – a process of obtaining a protocol from the URI and creating a request.
  - **Access** – a process of sending a request and obtaining a response as a result; access usually realized through HTTP.

# Overview

- Introduction to REST
- **Uniform Resource Identifier**
- Resource Representation
- Uniform Interface

# URI, URL, URN

- URI – Uniform Resource Identifier
  - *URI only identifies a resource*
    - *it does not imply the resource physically exists*
  - *URI could be URL (locator) or URN (name)*
- URL – Uniform Resource Locator
  - *in addition allows to locate the resource*
    - *that is — its network location*
  - *every URL is URI but an URI does not need to be URL*
- URN – Uniform Resource Name
  - *refers to URI under "urn" scheme (RFC 2141 [🔗](#))*
  - *require to be globally unique and persistent*
    - *even if the resource cease to exist/becomes unavailable*

# URI

- Definition

URI = scheme ":" [ "//" authority ] [ "/" path ] [ "?" query ] [ "#" frag ]

- Hierarchal sequence of components

- **scheme**

- *refers to a spec that assigns IDs within that scheme*

- *examples: http, ftp, mailto, urn*

- **scheme != protocol**

- **authority**

- *registered name (domain name) or server address*

- *optional port and user*

- **path and query**

- *identify resource within the scheme and authority scope*

- *path – hierarchal form*

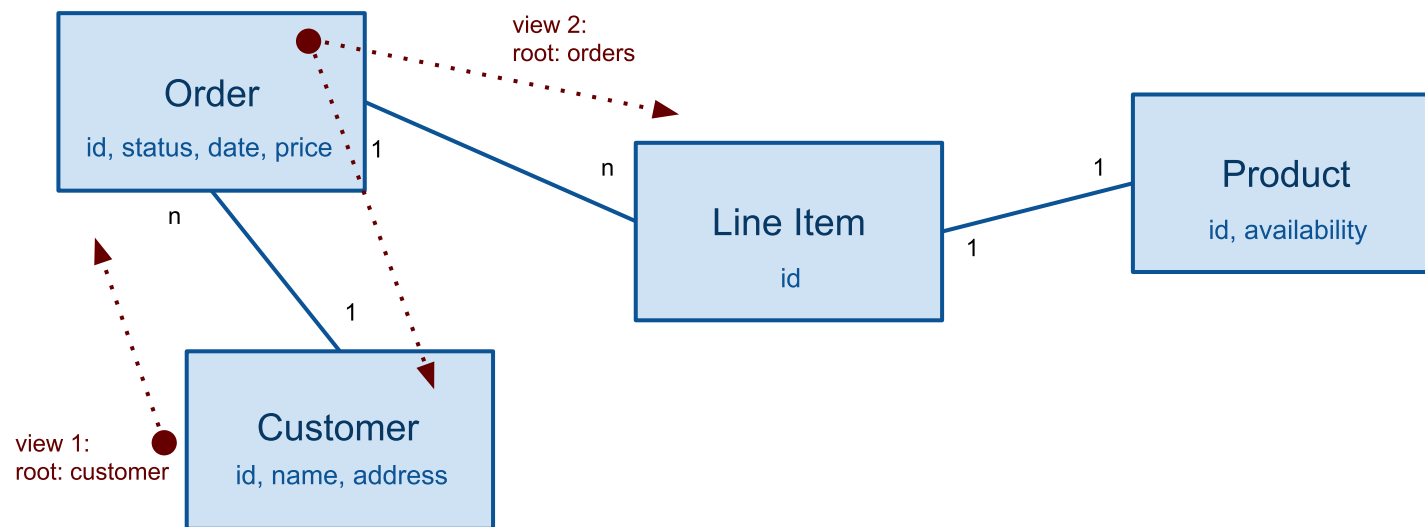
- *query – non-hierarchal form (parameters key=value)*

- **fragment**

- *reference to a secondary resource within the primary resource*

# Resources over Entities

- Application's data model
  - *Entities and properties that the app uses for its data*



- URI identifies a resource within the app's data model
  - **path** – a "view" on the data model
    - data model is a graph
    - URI identifies a resource using a path in a tree with some root

# Examples of Views

- View 1
    - *all customers*: `/customers`
    - *a particular customer*: `/customers/{customer-id}`
    - *All orders of a customer*: `/customers/{customer-id}/orders`
    - *A particular order*: `/customers/{customer-id}/orders/{order-id}`
  - View 2
    - *all orders*: `/orders`
    - *All orders of a customer*: `/orders/{customer-id}`
    - *A particular order*: `/orders/{customer-id}/{order-id}`
- ⇒ Design issues
- Good design practices
    - *No need for 1:1 relationship between resources and data entities*
      - *A resource may aggregate data from two or more entities*
      - *Thus only expose resources if it makes sense for the service*
    - *Try to limit URI aliases, make it simple and clear*

# Path vs. Query

- Path
  - *Hierarchical component, a view on the data*
  - *The main identification of the resource*
- Query
  - *Can define selection, projection or other processing instructions*
  - *Selection*
    - *filters entries of a resource by values of properties*  
`/customers/?status=valid`
  - *Projection*
    - *filters properties of resource entries*  
`/customers/?properties=id,name`
  - *Processing instructions examples*
    - *data format of the resource* → *cf. URI opacity*  
`/customers/?format=JSON`
    - *Access keys such as API keys*  
`/customers/?key=3ae56-56ef76-34540aeb`

# Fragment Semantics

- Fragment semantics for HTML

- *assume that `orders.html` are in HTML format.*

- 1 | `http://company.com/tomas/orders.html#3456`

- $\Rightarrow$  *there is a HTML element with `id=3456`*

- But:

- *Consider `orders` resource in `application/xml`*

- 1 | `<orders>`
    - 2 |     `<order id="3456">...</order>`
    - 3 |     `...`
    - 4 | `</orders>`

- *Can't say that `http://company.com/tomas/orders.xml#3456` identifies an order element within the `orders` resource.*

- *`application/xml` content type does not define fragment semantics*



# Major characteristics

- Capability URL
  - *Short lived URL generated for a specific purpose*
  - *For example, an user e-mail verification*
- URI Alias
  - *Two different URIs identifying the same resource*
- URI Collision
  - *One URI identifying two different resources (misuse of an URI authority)*
- URI Opacity
  - *Content type encoded as part of an URI*
  - `http://www.example.org/customers.xml`
- Resource versions encoded in an URI
  - *Two URIs identifying the same resource of different versions*
  - `http://www.example.org/v1/customers.xml`
- Persistent URL
  - *URL is valid even when the resource is obsolete*
  - *For example, a redirection should be in place*

# Overview

- Introduction to REST
- Uniform Resource Identifier
- **Resource Representation**
- Uniform Interface

# Representation and Data Format

- Representation
  - *Various languages, one resource can have multiple representations*
    - *XML, HTML, JSON, YAML, RDF, ...*
    - *should conform to Internet Media Types*
- Data format
  - *Format of resource data*
  - *Binary format*
    - *specific data structures*
    - *pointers, numeric values, compressed, etc.*
  - *Textual format*
    - *in a defined encoding as a sequence of characters*
    - *HTML, XML-based formats are textual*

# Metadata

- Metadata ~ self-description
  - *Data about the resource*
  - *e.g., data format, representation, date the resource was created, ...*
    1. *Defined by HTTP response headers*
    2. *Can be part of the data format*
      - *Atom Syndication Format such as author, updated, ...*
      - *HTML http-equiv meta tags*
- Resource anatomy



# Content-Type Metadata

- Access
  - *to be retrieved (GET)*
  - *to be inserted or updated (PUT, POST)*
  - *to be deleted (DELETE)*
- Request
  - HTTP header **Accept**, part of content negotiation protocol
- Response
  - HTTP header **Content-Type: type/subtype; parameters**
  - Specifies an Internet Media Type [🔗](#) of the resource representation.
    - IANA (Internet Assigned Numbers Authority) manages a registry of media types [🔗](#) and character encodings
    - subtypes of **text** type have an optional charset parameter  
**text/html; charset=iso-8859-1**
  - A resource may provide more than one representations
    - promotes services' loose coupling

# Major Media Types

- Common Standard Media Types
  - `text/plain`  
→ *natural text in no formal structures*
  - `text/html`  
→ *natural text embedded in HTML format*
  - `application/xml`, `application/json`  
→ *XML-based/JSON-based, application specific format*
  - `application/wsdl+xml`  
→ *+xml suffix to indicate a specific format*
- Non-standard media types
  - *Types or subtypes that begin with `x-` are not in IANA*  
`application/x-latex`
  - *subtypes that begin with `vnd.` are vendor-specific*  
`application/vnd.ms-excel`

# Resource State

- State
  - *Resource representation is in fact a **representation of a resource state***
  - *Resource may be in different states over time*



- In REST resource states represent application states

# Resource State Example

- Time **t1**: client A retrieves a resource **/orders** (GET)

```
1 <orders>
2   <order id="54467"/>
3   <order id="65432"/>
4 </orders>
```

- Time **t2**: client B adds a new order (POST)

```
1 <order>
2   ...
3 </order>
```

- Time **t3**: client A retrieves a resource **/orders** (GET)

```
1 <orders>
2   <order id="54467"/>
3   <order id="65432"/>
4   <order id="74567"/>
5 </orders>
```

- The resource **/orders** has different states in **t1** and **t3**.

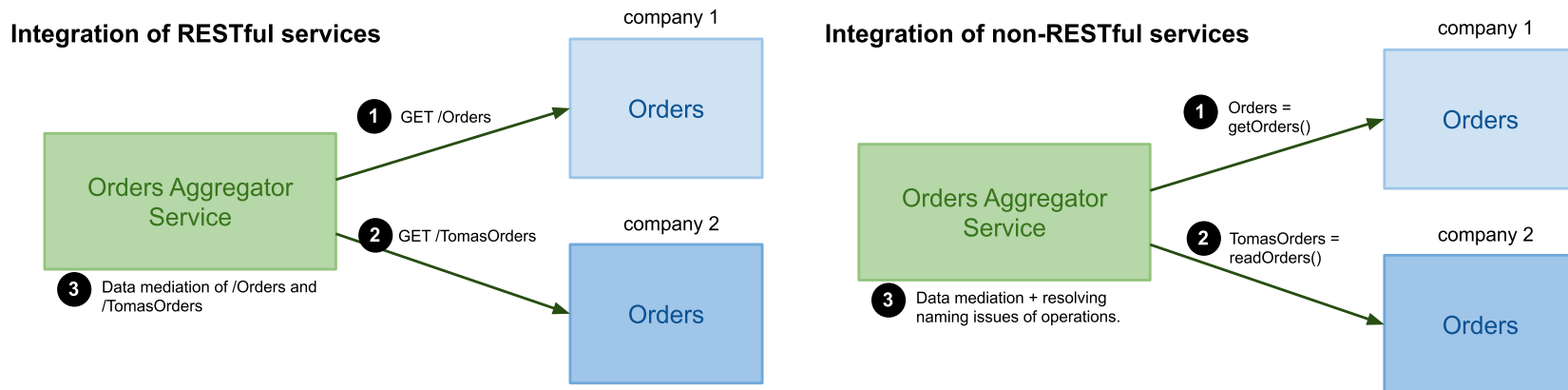


# Overview

- Introduction to REST
- Uniform Resource Identifier
- Resource Representation
- Uniform Interface

# 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*

# 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 or inserts a representation of a state of a resource*
- *Updating the resource is a **complete replacement of the 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***


# PATCH

- **PATCH** to partial update a resource
  - *IETF specification, see PATCH Method for HTTP* [🔗](#)
- Use in GData Protocol
  - *To add, modify or delete selected elements of an Atom feed entry*
  - *Example to delete a description element and add a new title element*
  - **gd:fields** uses the 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*

# 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*

# 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...
```

# 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