

# Web 2.0

## Lecture 2: 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/w20>



Modified: Thu Mar 16 2017, 01:16:36  
Humla v0.3

## Overview

- Introduction to REST
- Uniform Resource Identifier
- Resource Representation

## 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

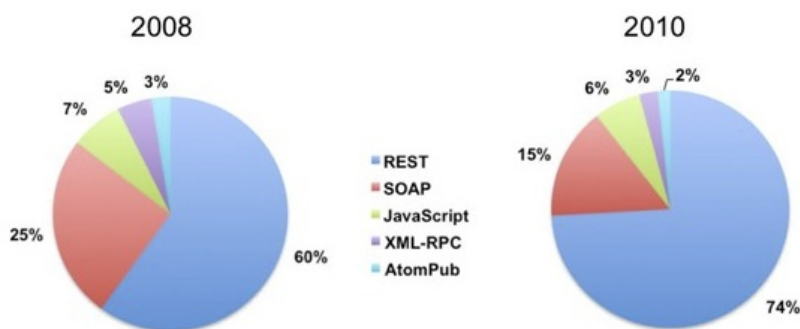
- Tim-Berners Lee
  - "creator", father of the Web
- Key Principles
  - Separation of Concerns
    - enables independent innovation
  - Standards-based
    - common agreement, big spread and adoption
  - Royalty-free technology
    - a lot of open source, no fees
- Architectural Basis
  - **Identification:** 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*

## Some Statistics

- ProgrammableWeb data
  - *Distribution of API protocols and styles*



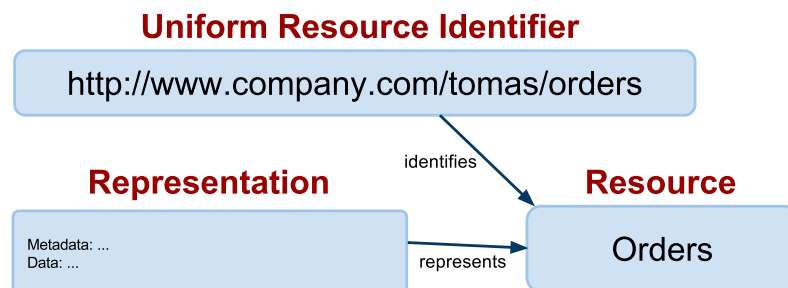
- *Based on directory of 2,000 Web APIs listed at ProgrammableWeb, May 2010.*
- *Source Open APIs: a State of the Market* [🔗](#)

## REST Core Principles

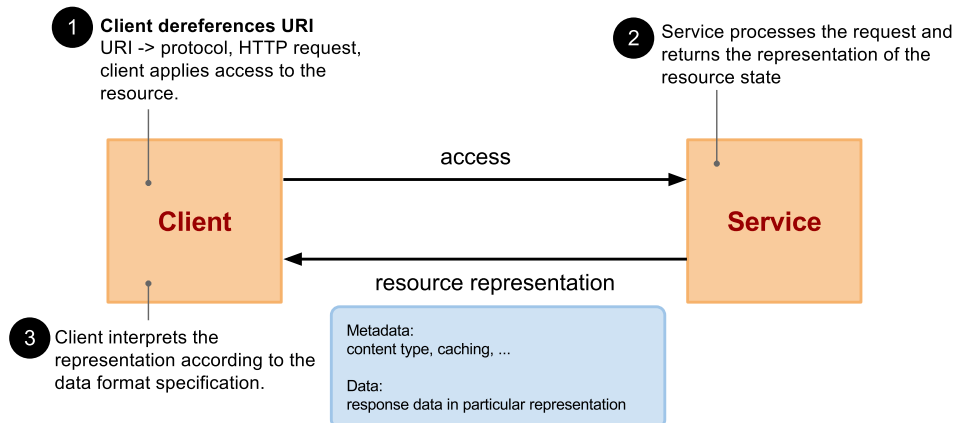
- REST architectural style defines constraints
  - *if you follow them, they help you to achieve a good design, interoperability and scalability.*
- Constraints
  - *Client/Server*
  - *Statelessness*
  - *Cacheability*
  - *Layered system*
  - *Uniform interface*
- Guiding principles
  - *Identification 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**
  - *Resources and Application Data*
  - *Good URI/URL Design*
- Resource Representation

## 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

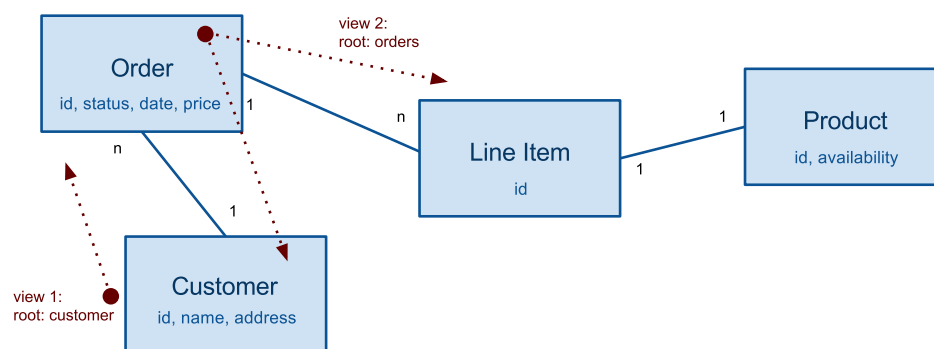
URI = scheme ":" [ "//" authority ] [ "/" path ] [ "?" query ] [ "#" frag ]
- Hierarchal sequence of components
  - **scheme**
    - refers to a spec that assigns IDs within that scheme
    - examples: **http**, **ftp**, **mailto**, **urn**
    - **scheme != protocol**
  - **authority**
    - registered 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

## Overview

- Introduction to REST
- Uniform Resource Identifier
  - *Resources and Application Data*
  - *Good URI/URL Design*
- Resource Representation

## 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

- Primary resource
  - Defined by URI path and query
  - could be complex, composed resources
- Sub-resource/secondary resource
  - Can be defined by a fragment
  - No explicit relationship between primary and sub-resource
    - For example, we cannot infer that the two resources are in **part-of**, or **sub-class-of** relationships.
  - Fragment semantics defined by a data format
- Usage of fragment
  - identification of elements in HTML
  - URI references in RDF
  - State of an application in a browser

## Fragment Semantics

- Fragment semantics for HTML
  - assume that **orders.html** are in **HTML** format.
    - 1 | `http://company.com/tomas/orders.html#3456`
  - ⇒ 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

## Resource ID vs. Resource URI

- Resource ID
  - Local ID, part of an entity in a data model
  - Unique within an application where the resource belongs
  - Usually generated on a server (cf. *PUT to update and insert*)
  - Exposed to the resource URI as a path element  
`/orders/{order-id}`
- Resource URI
  - Global identifier, valid on the whole Web
  - Corresponds to the view on the data model of the app
  - Include multiple "higher" resources' IDs
  - Example:  
`/customers/{customer-id}/orders/{order-id}/`
  - There can be more URIs identifying the same resource

## Overview

- Introduction to REST
- Uniform Resource Identifier
  - Resources and Application Data
  - *Good URI/URL Design*
- Resource Representation

## Capability URL

- What's capability URL
  - They are usually valid for a short period of time
  - They are not public, they are private to one person or a group of people
  - Ephemeral resources
- Examples
  - Password resets, Polls, Google calendar's private URLs, ...
  - Access control – key, session
- Design considerations
  - They should be https resources!
    - limits exposure, in logs or on the network
  - They should be revokable by the user/owner
  - They should not be persistent, they should expire
- Normal URLs
  - No URL collision, URI opacity, human readable, independence on a context, persistent URI

## URI Aliases and URI Collision

- URI Alias
  - More than one URI identifies a single resource
  - This happens, for example
    - Different views on the same data entity
      - view 1: `/customers/{customer-id}/orders`
      - view 2: `/orders/{customer-id}`
    - DNS load balancing:
      - domain name 1: `http://api.company.com/orders`
      - domain name 2: `http://api2.company.com/orders`
- URI Collision
  - Two resources have one URI
  - This should not happen, for example
    - A company uses an authority it does not own
      - company Amazon: `http://amazon.com/orders`
      - company Knihy.cz: `http://amazon.com/orders`
    - Exception: domain `example.org`

## Representation Reuse

- Compare this:
  - <http://company.com/tomas/orders/?date=111001>  
→ *all orders of Tomas till 1st October 2011*
  - <http://company.com/tomas/orders>  
→ *all orders of Tomas till today*  
→ *when retrieved on 1st October 2011, will be the same as the first resource*
  - *These are different resources*
    - *We say the two resources reused their representations*
    - *Representation reuse only happens under certain conditions*

## URI Opacity

- URI does not describe a resource data format
  - *In general it does not describe any resource metadata*
  - *Thus we cannot determine a format through URIs*
    - *There is no relation between URI and HTTP*
    - *HTTP media types does not affect URI path component*
- Example
  - <http://company.com/orders.html>  
→ *there is no guarantee that the resource is in **text/html** format*
- However, it sometimes comes handy
  - *Easy to retrieve a data format by tweaking URL (browser)*
  - *For example, Google API uses query parameter **alt***
  - *No need to fiddle with headers and using tools such as **curl***

## Human Readable URI

- URIs are both for machines and users
  - *Users should be able to memorize them*
  - *URIs should contain pronounceable words, good number of path components, clear query parameters, etc.*
- Example
  - *A human readable:*  
`http://company.com/tomas/orders/`
  - *Not really human readable:*  
`http://company.com/?c=gjddjsj224&a=58584&jbd=5553a`
- URIs generated by a machine – capability URLs
  - *URLs that are not meant to be "remembered"*

## Independence on a Context

- URIs are independent on a user context
  - *It should be possible to share URIs among users*
    - *For example, you send an URI over an IM system*
    - *Others should be able to retrieve the same resource as you (if they have rights)*
- BUT:
  - *URL may include an access or a session information – capability URL*
- Example
  - *Capability URL:* `http://company.com/orders/?session=5582&user=bob`
    - *This cannot be reused by other user than Bob*
  - *No context:* `http://company.com/orders/`
    - *a user needs to be logged in to access the resource*
    - *HTTP authorization header identifies the user*

## Resource Versions

- Resources evolve over time
- Need to deal with various versions
  - *need to support old clients on old versions*
  - *allow new clients to use new versions*
- Versioning at URI level
  - *one path element to identify a version*  
`http://company.com/v1/tomas/orders`
  - *should be part of the path component not a query*
  - *API version*  
→ *version applies to a set of resources*
- Versioning at resource meta-data level
  - *cf. [Version control via content negotiation](#)*

## Persistent URI

- Good URLs should not change
  - *They should be indefinitely assigned to a resource*
  - *even if the resource does not exist anymore*
- HTTP and URI persistence
  - *new URI associated with the resource*
  - *HTTP redirection through **3xx** response codes*  
→ *See response codes*
- Capability URLs are not usually persistent

## Overview

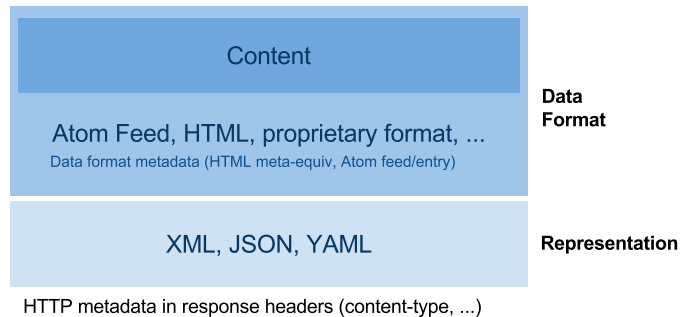
- Introduction to REST
- Uniform Resource Identifier
- Resource Representation
  - *Representation, Data Format and Metadata*
  - *Resource State*
  - *Content Negotiation*

## 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
    - AtomPub protocol 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

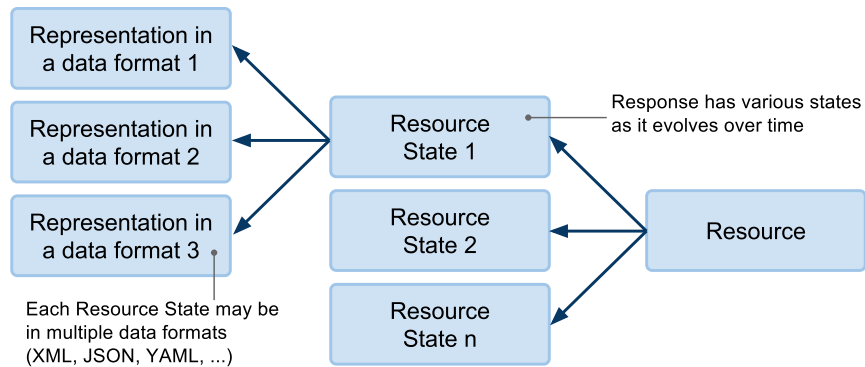
- Common Standard Media Types
  - `text/plain`  
→ *natural text in no formal structures*
  - `text/html`  
→ *natural text embedded in HTML format*
  - `application/xml`, `application/json`  
→ *XML-based/JSON-based, application specific format*
  - `application/wsdl+xml`  
→ *+xml suffix to indicate 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`

## Overview

- Introduction to REST
- Uniform Resource Identifier
- Resource Representation
  - *Representation, Data Format and Metadata*
  - *Resource State*
  - *Content Negotiation*

## 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  $t_1$ : client A retrieves a resource `/orders` (GET)

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

- Time  $t_2$ : client B adds a new order (POST)

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

- Time  $t_3$ : 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  $t_1$  and  $t_3$ .

## Overview

- Introduction to REST
- Uniform Resource Identifier
- Resource Representation
  - *Representation, Data Format and Metadata*
  - *Resource State*
  - *Content Negotiation*

## Content Negotiation

- Advantages
  - *Different clients may want to use different formats*
    - *Web browser: JSON*
    - *Java client: XML*
    - *Ruby client: YAML*
  - *Clients want internationalized data*
    - *translated information in various languages*
  - *applications evolve*
    - *need for version support*
- HTTP Content Negotiation
  - *a protocol, also called conneg*
  - *format, encoding, language*

## Representation Negotiation

- Client requests specific media types it supports
  - client sets **Accept** header in the request
    - the value is a comma delimited set of content types
- Specific requests
  - to ask for xml or json representations:
    - > GET /orders HTTP/1.1
    - > Accept: application/xml, application/json
  - server chooses one of the types (by applying preference ordering) and serializes the resource in that type
  - when the server cannot find any type, it sends **406 Not Acceptable** response code
- Generic requests
  - client may specify wildcards to ask for any type or subtype
    - > GET /orders HTTP/1.1
    - > Accept: text/\*, text/html; level=1

## Preference Ordering – Implicit Rule

- Implicit rule
  - More specific media type takes preference over less specific ones.  
*Example:*
    - > GET /orders HTTP/1.1
    - > Accept: text/\*, text/html;level=1, \*/\*, application/xml
  - server interprets the client preference as:
    1. **text/html;level=1** – most specific
    2. **application/xml** – no parameters
    3. **text/\*** – more concrete than match-all
    4. **\*/\*** – less specific

## Preference Ordering – Explicit Rules

- Explicit rules

- using **q** parameter (qualifier), numeric value from 0.0 to 1.0 (1.0 indicates the most preferred type)

- > GET /orders HTTP/1.1

- > Accept: text/\*;q=0.9, \*/\*;q=0.1, application/json, application/xml;q=0.

- server interprets the client preference as:

- 1. **application/json** – implicit qualifier 1.0, most specific

- 2. **text/\*** – the second next highest qualifier 0.9

- 3. **application/xml** – more specific but lower pref. value 0.5

- 4. **\*/\*** – anything otherwise

## Language and Encoding Negotiation

- Language negotiation

- Client uses **Accept-Language** header; the value is a comma separated list of language (ISO 639) and country codes (ISO 3166)

- > GET /orders HTTP/1.1

- > Accept-Language: en-us, cs, fr

- < Content-Language: en-us

- Supports preference qualifiers too

- Encoding negotiation

- Client uses **Accept-Encoding** for message compression  
the value is a comma separated list of acceptable compressions

- > GET /orders HTTP/1.1

- > Accept-Encoding: gzip, deflate

- < Content-Encoding: gzip

- Supports preference qualifiers too

- When a client or a server compress a message body the **Content-Encoding** must always be specified!

## Resource Version Negotiation

- Applications and their resources evolve
  - *A service need to support old clients*
  - *The service's URI and methods do not need to change – the content it provides may be in different versions*
  - *cf. resource versions in Lecture 2.*
- Encode the version information
  - > GET /orders HTTP/1.1
  - > Accept: application/xml; version=2.0
  - < HTTP/1.1 200 OK
  - < Content-Type: application/xml; version=2.0

## Respecting Standards?

- Negotiation by URI patterns
  - *quite common, for example:*
    - `http://company.com/orders/?alt=json` (*Google APIs*)
  - *or in the URI path component:*
    - `http://company.com/orders.xml`
    - `http://company.com/orders.xml.en-us`
    - `http://company.com/orders.json`
  - *But be aware of the URI Opacity!*