# Génie logiciel

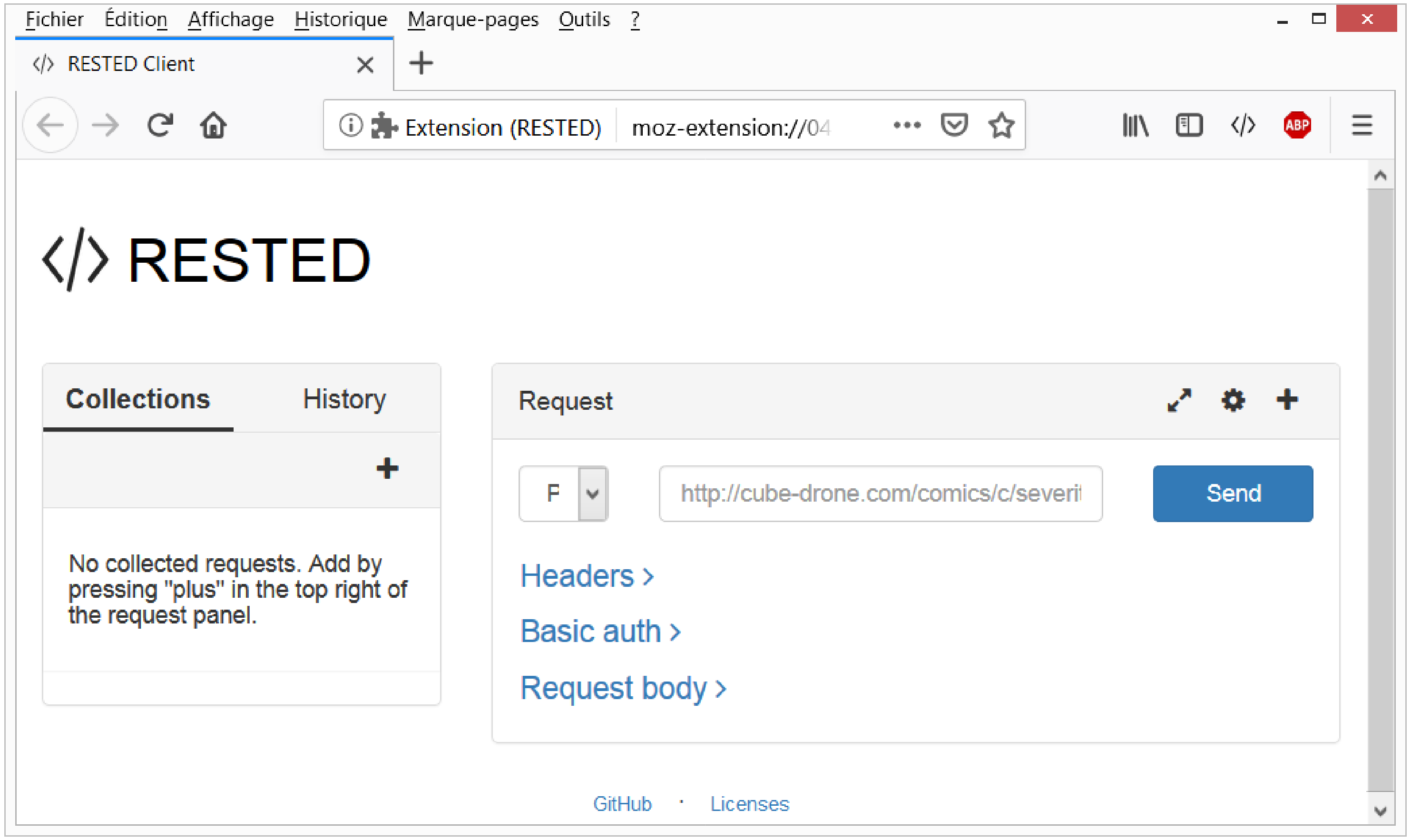
Philippe Dugerdil

21.11.2019

© Philippe Dugerdil - CUI - Université de Genève

# REST Client for Firefox (RESTED)

Plugin through which any HTTP query can be issued



No collected requests. Add by pressing "plus" in the top right of the request panel.

Located in the TOMCAT server. This is why we must specify the target system at the web project creation

# Hello world

```
import javax.ws.rs.*;
                                              Alternative declaration
@Path("/")                                    @Path("/helloworld")

public class Hello {

@GET
@Path("helloworld")                           NO @Path() declaration
public String getHelloTXT() {return "Hello World";}
}
```

Same URL for both : *domainName*/HelloWorld/helloworld

## REST APIs & annotations

© Philippe Dugerdil - CUI - Université de Genève

## Calling a REST service

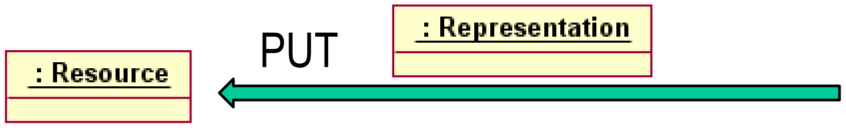
Each time one needs a new operation to work on a resource, one must combine one of the HTTP verbs with a specific link

- Operation “signature” : $\left\{\begin{array}{c}\text{GET}\\\text{PUT}\\\text{POST}\\\text{DELETE}\end{array}\right\}$ + URL

## When to use PUT ?

Must be used to change the state of an existing resource (i.e. update the resource)

- The **full** representation must be transferred from the client for any state change.
- The URL of the resource should **preexist**

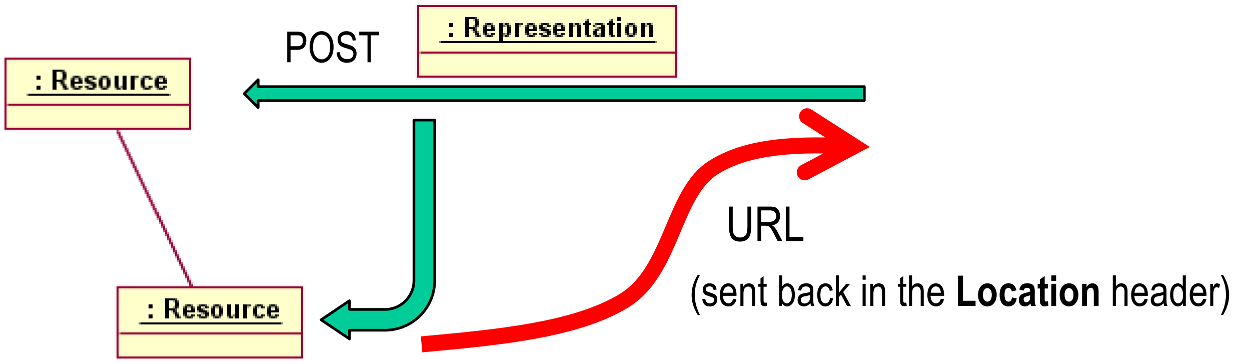


# When to use POST ?

To create new subordinate resources

(i.e. resource in relation to some “parent” resource)

- Then one should send a “POST” query to the parent.
  - The parent can be seen as a “Factory” resource
- The server will assign the resource a new URL and return it



## PUT vs POST

- If URL is generated by the server: POST

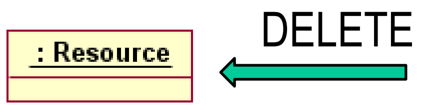
- If the state of an existing resource must be changed without URL creation: PUT

(In some rare cases PUT could be used to create a resource if the client knows the URL of the resource. But generally it does not)

# When to use DELETE ?

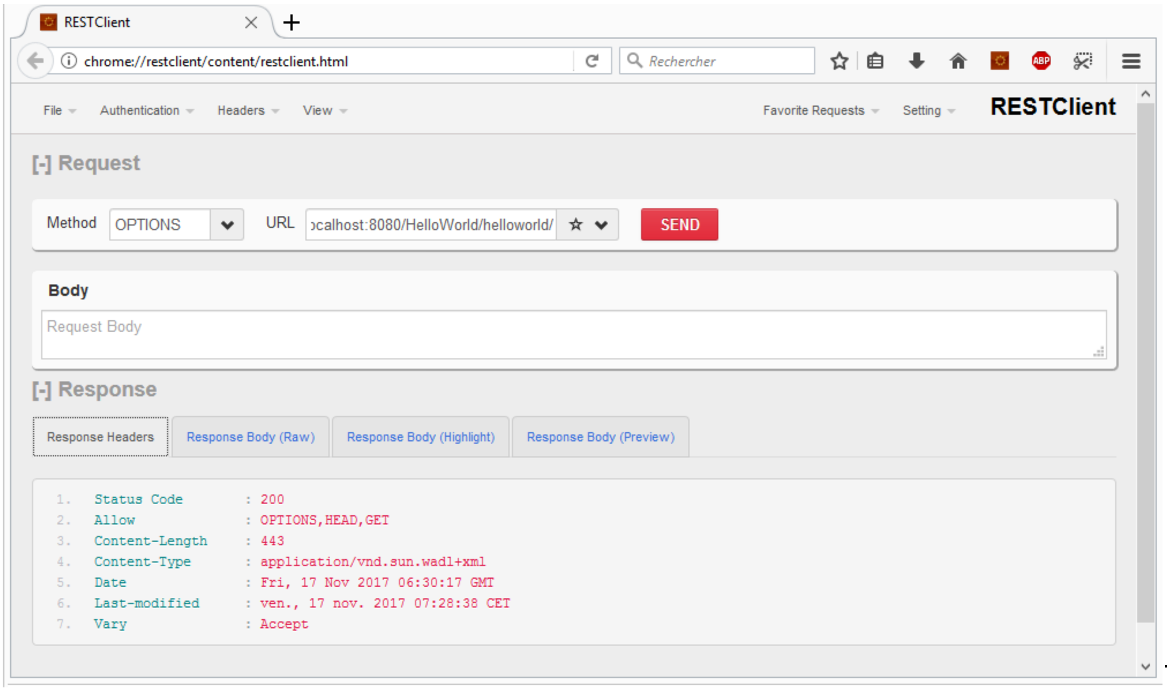
To delete the target resource, i.e. to remove it from the contents managed by the server.

- This is a physical delete not only a logical delete



## Useful other verbs: HEAD, OPTIONS

- HEAD: to retrieve metadata-only representation
  - Used for example to check if a resource exists
  - This avoid to download potentially huge contents
- OPTIONS: to do simple access control checks
  - Example: response contains the *Allow* header that tells the clients which verbs are supported



```
Status Code      : 200
Allow            : OPTIONS,HEAD,GET
Content-Length   : 443
Content-Type     : application/vnd.sun.wadl+xml
Date             : Fri, 17 Nov 2017 06:30:17 GMT
Last-modified    : ven., 17 nov. 2017 07:28:38 CET
Vary             : Accept
```

## Method parameters

### Method parameter are annotated too

In a method mapped to the (PUT, POST) verbs, a single method parameter only may have no annotation

- This is used to pass the payload of the query to the method
- All the other parameters **must be** annotated

## Payload in REST service's methods

- The payload to be outputted by a method is the object **returned** by the method
- The payload to be inputted in a method (PUT, POST) is passed to the single method parameter **without annotation**

## Example

Payload returned by the method in String format

```
@GET
String getString() {return "Hello world";}

@POST
void myPost(String inputString) {………………..}
```

Payload is injected in this parameter since it does not have an attached annotation

# @Produces, @Consumes

The type of content to be read/written by the Java method.
(Sets the value of the *Contents-Type* & *Accept* headers)

- @Produces("*MIME Contents-Type*")
  - This tell JAX RS the type of the contents **returned** as the message body of an operation. (HTTP header of the reply : **Contents-Type**)

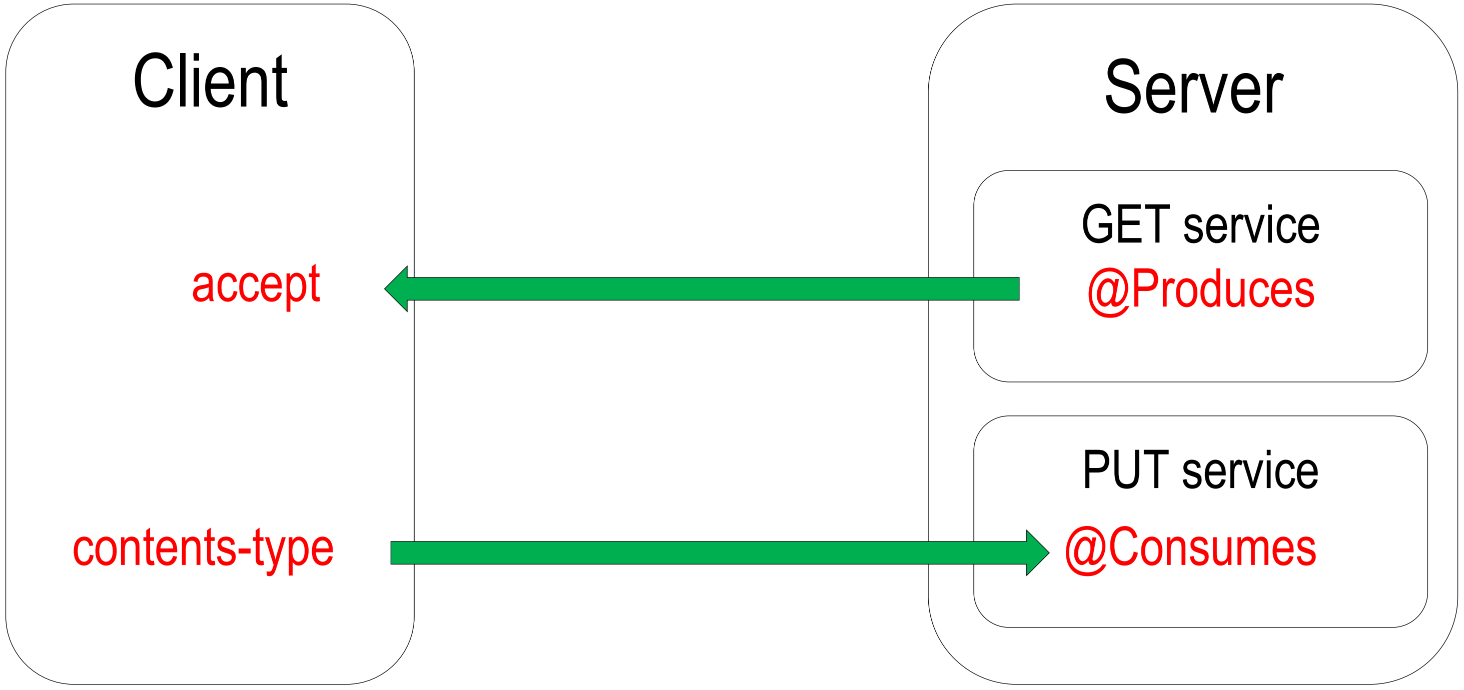
- @Consumes("*MIME Contents-Type*")
  - This tell JAX RS the type of the contents that is **expected** as input from the message body an operation. (matches the requests HTTP header : **Contents-Type**)
    - If some input does not comply with the type, the error 415 (contents not supported) is issued.

## Useful constants for HTTP Headers

- Javax.ws.rs.core :
  - The interface *HttpHeaders* declares the header keywords constants (useful on the client side)
  - The *MediaType* class declares the media types keywords constants
- Examples
  - HttpHeaders.CONTENT_TYPE
  - HttpHeaders.ACCEPT

| | |
|---|---|
| – MediaType.APPLICATION_XML | "application/xml" |
| – MediaType.APPLICATION_ATOM_XML | "application/atom+xml" |
| – MediaType.APPLICATION_JSON | "application/json" |
| – MediaType.TEXT_PLAIN | "text/plain" |

# Contents declaration match
## (Contents negociation)



## Selecting the service’s method to execute

There are 3 parameters which selects the method to be executed when processing a query:

1. The HTTP verb of the query (@PUT,…)
2. The URL (@Path(…))
3. The type of contents (@Produces(…))

# Example

```
@Path("/")
public class SwitchPath {
        @GET
        @Path("hello")
        @Produces(MediaType.TEXT_PLAIN)
        public String hello1(){…}
        @GET
        @Path("hello")
        @Produces(MediaType.APPLICATION_XML )
        public String hello2(){…}
        @GET
        @Path("goodbye")
        @Produces(MediaType.APPLICATION_XML )
        public String hello3(){…}
        @PUT
        @Path("goodbye")
        @Consumes(MediaType.APPLICATION_XML )
        public void hello4(){…}
```

GET domain/app/hello
Accept : text/plain

GET domain/app/hello
Accept : application/xml

GET domain/app/goodbye
Accept : application/xml

PUT domain/app/goodbye
Contents-type: application/xml

## REST architectural principles

## 5 Principles

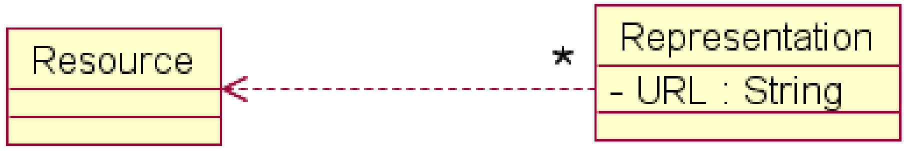
1. Addressability
2. Uniform interface
3. Resource oriented
4. Statelessness
5. Connectedness

- This is to assure protocol **visibility**. Visibility is a key feature of HTTP
- Visibility is “the ability of a component to monitor or mediate the interaction between two other components” [Fielding’s doctoral dissertation]

# 1. Adressability

All the client's relevant representations of a resource should get its own ID which is a URL

- An URL must identify a **single** resource **representation**.
- The same representation of a single resource may get multiples URLs



## 2. Uniform interface

HTTP Verbs only

- (we have seen this already)

## 3. Resource oriented

The complexity of the client-server interaction is within the **representation of resources** being passed back and forth. It is not in the set of actions verbs.

– The representation could be XML, JSON, TEXT,… (or any application-specific format, but it will not be interoperable across applications)

## 4. Statelessness I

Reminder: state = internal configuration of a system that specifies the response to the event it receives.

There are 2 kinds of states:

- **Resource state**: configuration of a resource on the server that is available to all clients. This determines what representation one could get from the resource.
- **Client state**: configuration of the client that determines what query it could issue next (following some user input for example). This is also called the ***application*** *state.*

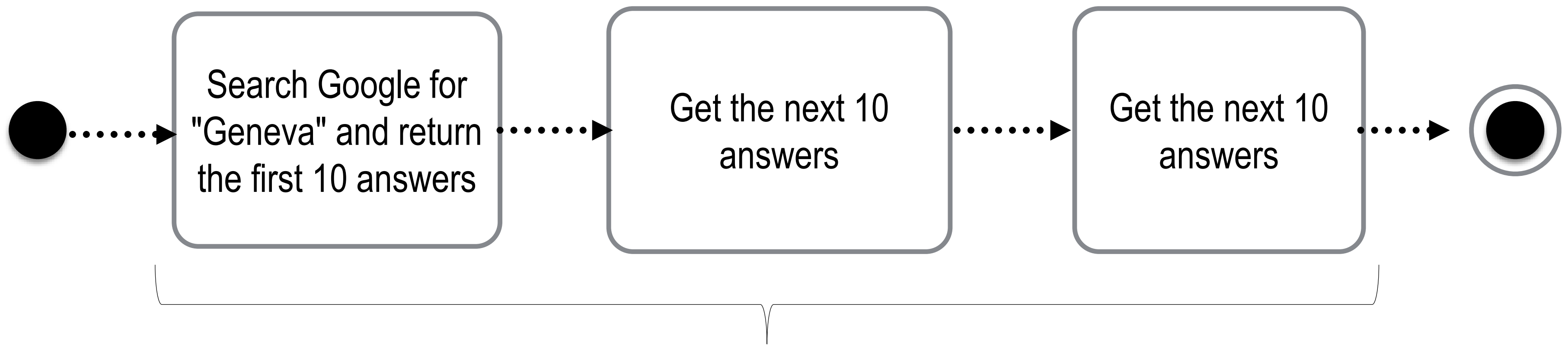
## Statelessness II

**All queries should be independent from each other from the point of view of the server**

- The server does not record the interaction with a particular client.
- Each client "navigates" among the resources and records the last (client) state he reached.

- Therefore, there are 2 locations for the states:
  - Client states (client side, specific to the client).
  - Resource states (server side, available to all clients)

# Examples of client states



Application (client) states in the browser

Transitions
- www.google.com/search?q=geneva
- www.google.com/search?q=geneva & start=10

Client's state communicated to the server

## Statelessness, what does it mean?

- It means that there are **no user sessions** on the server.
  - The server must not record the state of the conversation with some user
  - But this does not mean that it should not record states in a database, provided that it is **resource** state.

- Tricky example: shopping cart
  - Application or resource state ?
    - Specific to client, since other customer do not know about it ?
    - But if the client crashes, it could retrieve it, so it is not on the client ?

Buying on internet is part of a business process the customer must go through to complete the transaction. The steps are required to buy the goods. Therefore the server must know about it : resource states (the steps of the process). But if the user only browses products without buying, this is client state.

## 5. Connectedness (HATEOAS) I

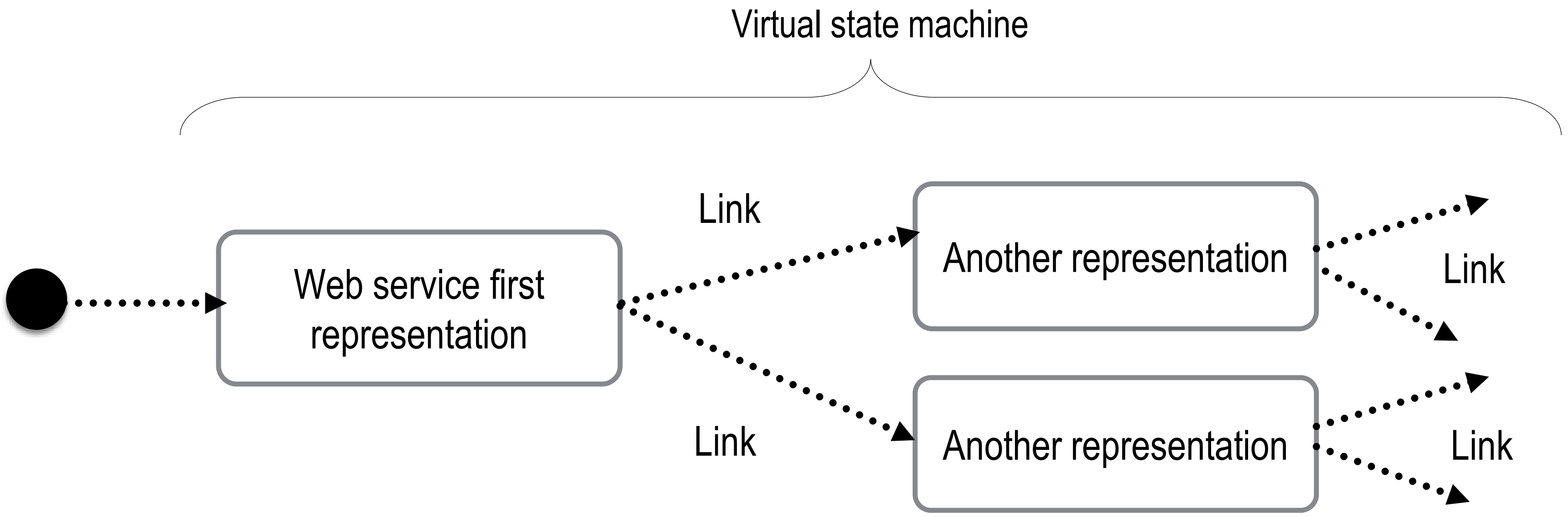
### Hypermedia As The Engine Of Application State

- The principle is that a client interacts with a network application entirely through hypermedia provided dynamically by application servers.
- A REST client needs no prior knowledge about how to interact with any particular application or server beyond a generic understanding of hypermedia.

[Wikipedia]

# HATEOAS II

- REST service response (representation) may contain hypermedia links to tell the client what the next relevant (client) state could be.
- Then, it is the server that shapes the possible state transition of the client.



## Safe and idempotent requests

- **Safe**: a request is *safe* if it does not change the state of the resource.
  - GET and HEAD must be safe.
- **Idempotent**: a request is *idempotent* if the effect of making one request is the same as making it several times. No relative change of resource state should be allowed.
  - PUT and DELETE must be idempotent.
  - So are GET and HEAD, of course

These properties are important in an unreliable network environment.

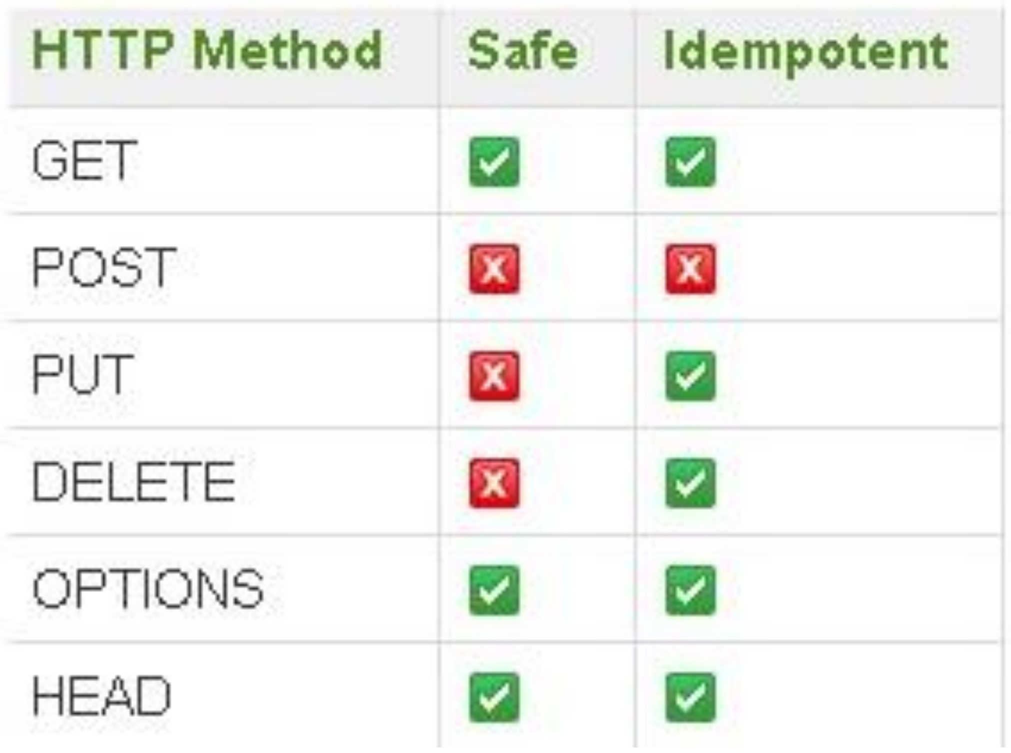
- POST and PATCH are neither safe nor idempotent.

## Example: unsafe GET

- The web is full of applications that misuse GET
  - For example a GET that changes the state of the resource.
  - This is not RESTful and should be avoided
  - When browsing the pages, search engines issue GET queries !

- The key issue is **standardization**: one should design the services like the others on the web do.
  - Comply with safe and idempotent services.
  - Consistently use GET, PUT, DELETE and POST in accordance with RFC7231

## Summary



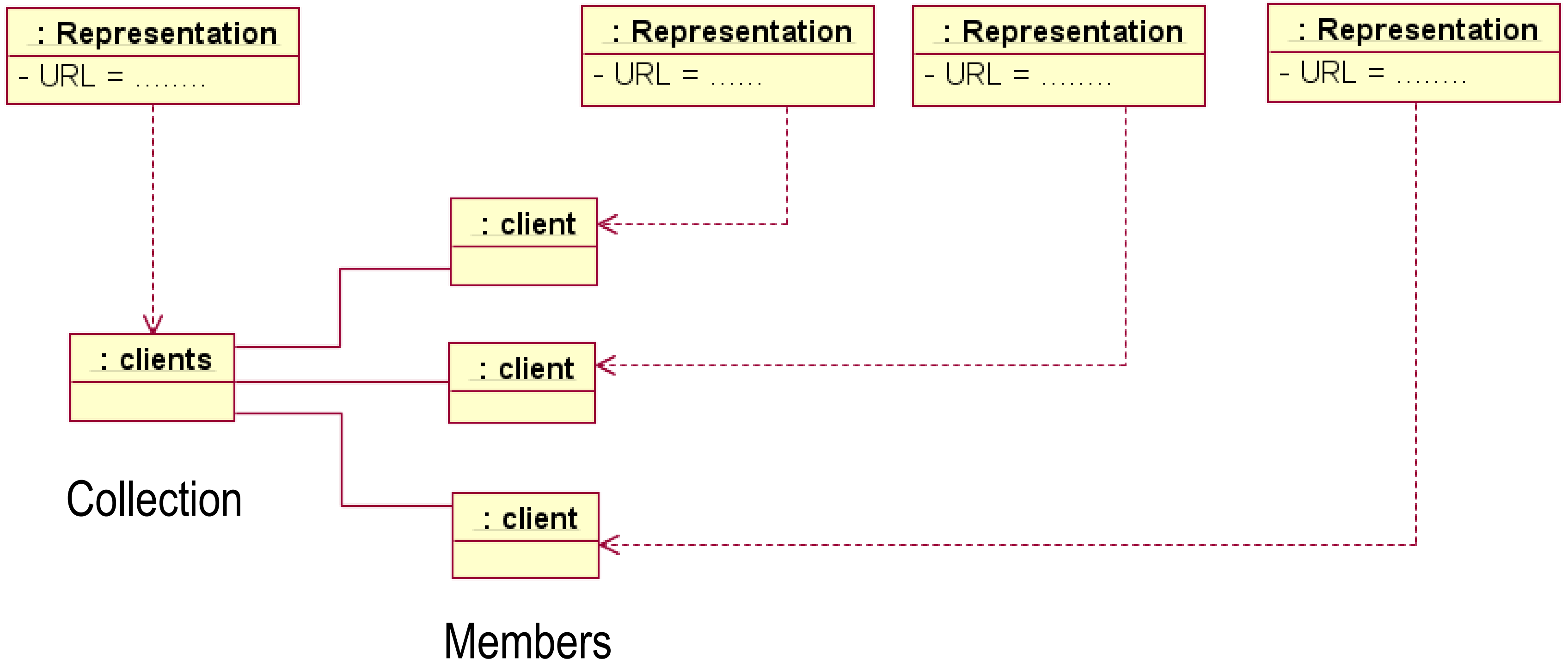
| HTTP Method | Safe | Idempotent |
|---|---|---|
| GET | ✓ | ✓ |
| POST | ✗ | ✗ |
| PUT | ✗ | ✓ |
| DELETE | ✗ | ✓ |
| OPTIONS | ✓ | ✓ |
| HEAD | ✓ | ✓ |

## Architecting REST queries & URLs

The *Addressability* principle asks for some discipline in URL design.

- The resource to which the operation is applied must be explicitly addressed in the URL (never in the payload)
- HTTP Tunneling must be avoided

# Example: client's resources



## REST URL design : GET

To get a representation of a **collection** of resources, send a GET command with the specific URL of the collection.

- There must therefore be a resource representing the collection ("one-off" kind of resources)

```
GET /myHotelApplication/clients
Host: www.myserver.com
. . .
```

Return code: 200 (OK) or 404 (NOT FOUND) if URL not found

## REST URL design : GET

To get a representation of a **single resource**, issue a GET command with the specific URL to the resource, that is managed by the collection:

```
GET /myHotelApplication/clients/123
Host: www.myserver.com
. . .
```

Return code: 200 (OK) or 404 (NOT FOUND) if URL not found

## REST URL design : PUT

To update a resource, issue a PUT command with the specific URL to the resource, and payload= **full** representation :

```
PUT /myHotelApplication/clients/123
Host: www.myserver.com
Content-type: application/XML

<client>
   <name>philippe</name>
   <address>Geneva</address/>
   <number>3256</number>
   <phone>0041013123457</phone>
</client>
```

This must NOT create the client 123. It should only update an existing client. The client's contents will be replaced by this contents

Return code: 200 (OK), 204 (NO CONTENTS) if nothing is returned, 404 (NOT FOUND) if URL not found or 403 (FORBIDDEN) if request not accepted by the service (payload = reason why it is not acceptable).

## REST URL design : POST

To create a resource, issue a POST command with the URI to the container resource. Payload= **full** representation :

```
POST /myHotelApplication/clients
Host: www.myserver.com
Content-type: application/XML
Accept: application/XML

<client>
   <name>paul</name>
   <address>Lausanne</address/>
   <number>124</number>
   <phone>004101312345 7</phone>
</client>
```

URL of the new client is NOT provided. So the query is directed to the container (i.e. the collection)

New URL:
Attached to the **Location** header of the response

Return code: 201 (CREATED), 404 (NOT FOUND) if URL not found or 403 (FORBIDDEN) if request not accepted by the service (payload = reason why it is not acceptable).

## REST URL design : DELETE

To delete a resource, issue a DELETE command with the specific URI to the resource:

```
DELETE /myHotelApplication/clients/123
Host: www.myserver.com
```

Return code:  200  (OK), 204 (NO CONTENTS) if nothing is returned, 404 (NOT FOUND) if URL not found or 403 (FORBIDDEN) if request not accepted by the service (payload = reason why it is not acceptable).

## URL design, basic technique

- Always use the *container/id* pair in URLs
  - The name of the container must be plural

- Examples
  - students/123
  - students/123/addresses
  - students/123/addresses/ad2/city

students represents the root container (one_of resource)

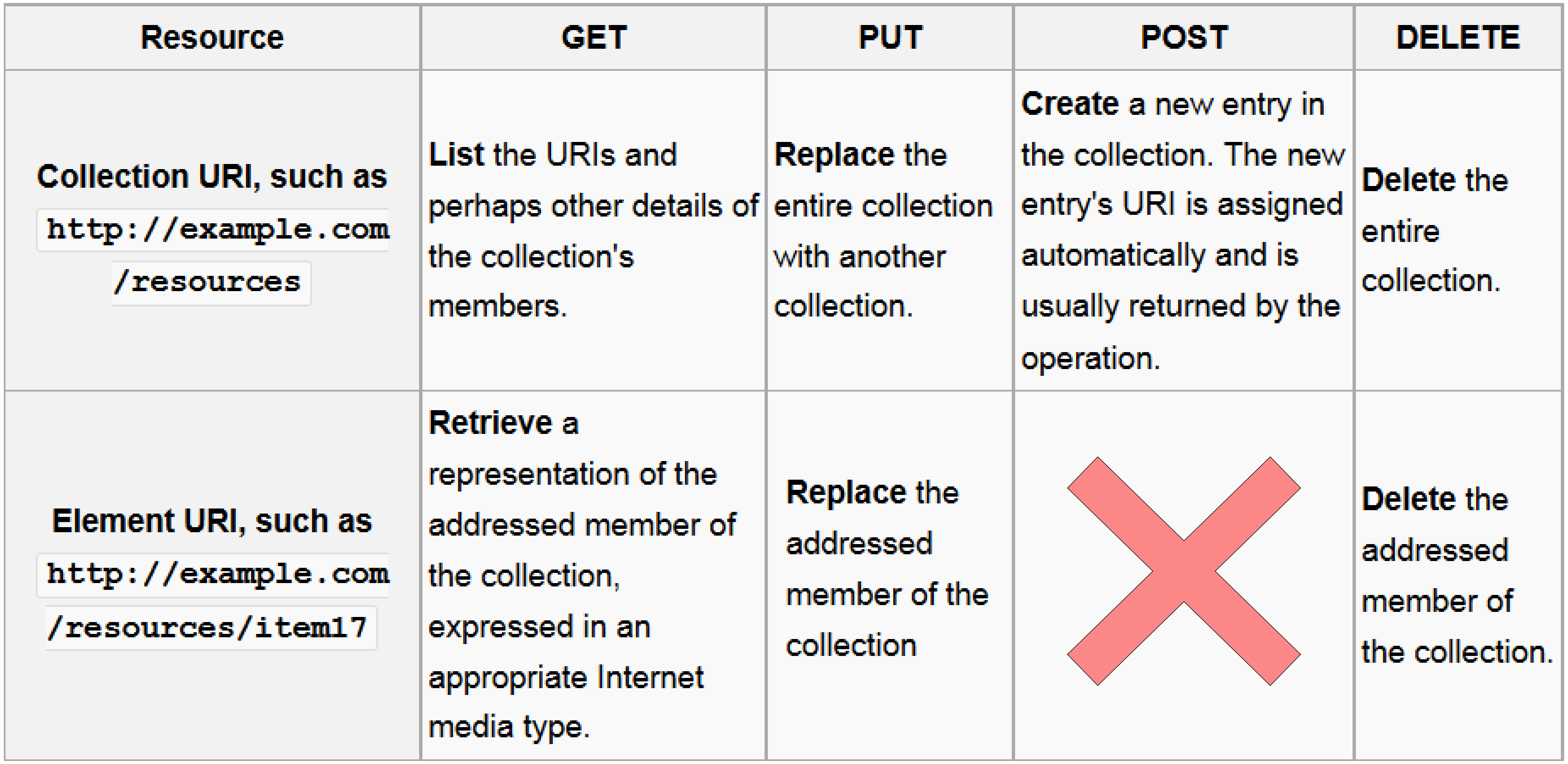
# Advice

## Keep URL consistency among queries

- If a GET query is `domain/app/containers/{id}`
- Then a PUT / DELETE query MUST use the same

- A POST query must use the same but the {id} :
  - `domain/app/containers`
  - Since the server will assign the id

# Summary RESTful APIs



| Resource | GET | PUT | POST | DELETE |
|---|---|---|---|---|
| **Collection URI, such as** `http://example.com/resources` | **List** the URIs and perhaps other details of the collection's members. | **Replace** the entire collection with another collection. | **Create** a new entry in the collection. The new entry's URI is assigned automatically and is usually returned by the operation. | **Delete** the entire collection. |
| **Element URI, such as** `http://example.com/resources/item17` | **Retrieve** a representation of the addressed member of the collection, expressed in an appropriate Internet media type. | **Replace** the addressed member of the collection | ❌ | **Delete** the addressed member of the collection. |

[adapted from Wikipedia]

© Philippe Dugerdil - CUI - Université de Genève

## HTTP-tunelling RPC: NOT RESTful

Some people use HTTP operations to simulate RPC :

- Single HTTP verb (POST, GET, PUT)
- Single entry point to the server (single URI)
- Payload with operation name + parameters

```
GET /myHotelApplication/customers
Host: www.myserver.com
Content-type: application/XML

<operation
   <opname= “update”/>
   <parameter name=“number” value=“customer21”/>
   <parameter name=“name” value=“philippe”/>
</operation>
```

## UnRESTFul example revisited

```
PUT /myHotelApplication/customers/customer21
Host: www.myserver.com
Content-type: application/XML

<customer>
   <name>philippe</name>
   <address>Geneva</address/>
   <job>IT developper</job>
   <phone>004101312345</phone>
</customer>
```

Full representation

NB: the full representation must be given in a PUT request, even if a single attribute must be changed.

## Example

- Design the resource architecture & the URL to read/write:
  - The list of the students of a faculty
  - The personal info of one student
  - The list of exams taken by some student
  - The mark got in a specific exam (i.e. the detailed information about it)
  - The course associated to this exam (i.e. the detailed information about it)
  - The personal info of the professor who teaches this course
  - The list of all the addresses of the professor
  - The detailed information about one address in particular

## Examples of API design

Have a look at Spotify

https://developer.spotify.com/web-api/endpoint-reference/