REST - Part 2

Syed Gillani
Laboratoire Hubert Curien St-Etienne, France

***REST** is*

an architectural style

which specifies a set of architectural constraints to use HTTP for web services

[REST: HTTP Methods]

HTTP		SAFE	IDEM POTENT
POST	Create a sub resource	NO	NO
GET	Retrieve the *current state* of the resource	YES	YES
PUT	Initialize or update the state of a resource at the given URI	NO	YES
DELETE	Clear a resource, after the URI is no longer valid	NO	YES

[REST: HTTP Methods]

Safe →

GET	POST
PUT	DELETE

Safe-Operations
Read only operations: they do not change the server state. Call them 1, 10, or 1000 times. They will not change the server state (GET)

[REST: HTTP Methods]

Idempotent ⟶

GET	POST
PUT	DELETE

Idempotence Definition
En mathématiques et en informatique, le concept d'**idempotence** signifie essentiellement qu'une opération a le même effet qu'on l'applique une ou plusieurs fois, ou encore qu'en la réappliquant on ne modifiera pas le résultat. (wikipedia)

Idempotence Definition
It is the property of certain operations in mathematics and computer science, that can be applied multiple times without changing the result beyond the initial application (Wikipedia)

RESTfull Idempotence
An **operation** (or service call) to be **idempotent,** clients can make that **same call repeatedly** while producing the **same result**. In other words, making multiple identical requests has the same effect as making a single request.

[REST: HTTP Methods]

Name *n; List l*

Idempotent

```
For  i=0 to 9 do
{
l.add(0, bob);// puts bob as the first name in the list
}
```

Compares with:

Not Idempotent

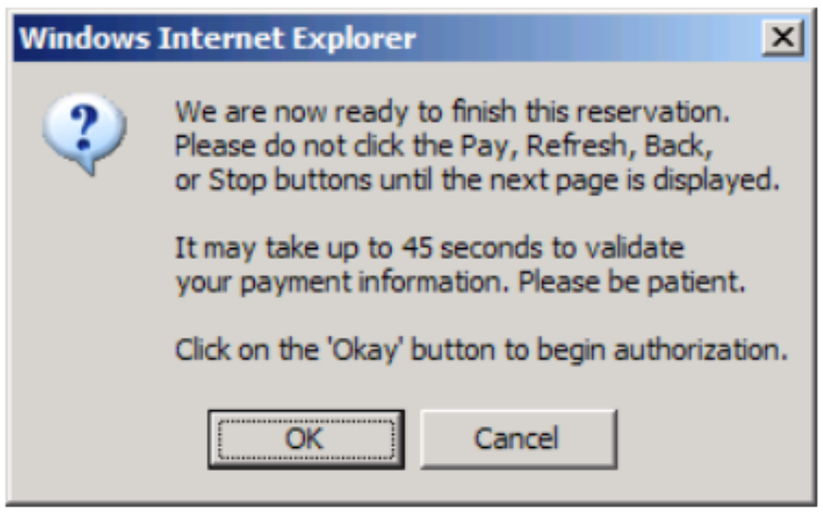
```
For  i=0 to 9 do
{
l.add(i, bob);// adds bob to the list
 }
```

[POST vs GET]

- **GET** is a read-only operation. It can be repeated without affecting the state of the resource (idempotent) and can be cached

***Note**: this does not mean that the same representation will be returned every time*

- **POST** is a read-write operation and may change the state of the resource and provoke side effects on the server

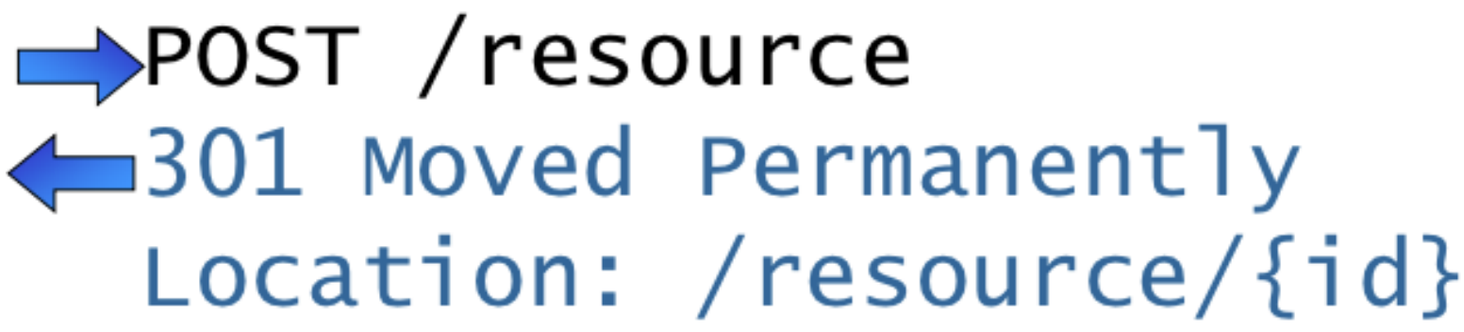


Web browsers warn you when refreshing a page generated with POST

[POST vs PUT]

- What is the right way of creating resources (initialising their state)?

- **Problem:** How to ensure resource {id} is unique?
- Resources can be created by multiple clines concurrently
- **Solution 1:** Let the client choose a unique id (e.g. GUID)



- **Solution 2:** Let the server compute the unique id
- **Problem:** Duplicate instances may be created if requests are repeated due to unreliable communication

[HATEOAS: **H**yper-media **A**s **T**he **E**ngine **O**f **A**pplication **S**tate]

[HATEOAS]

- In SOAP days, we had an XML document called WSDL
  - ✓what was the role of WSDL?

- With REST, do we have something like the WSDL?
  - ✓how to find, which operation is provided by a web service?

- With restful services, no Formal WSDL
- **Providers offer a documentation**

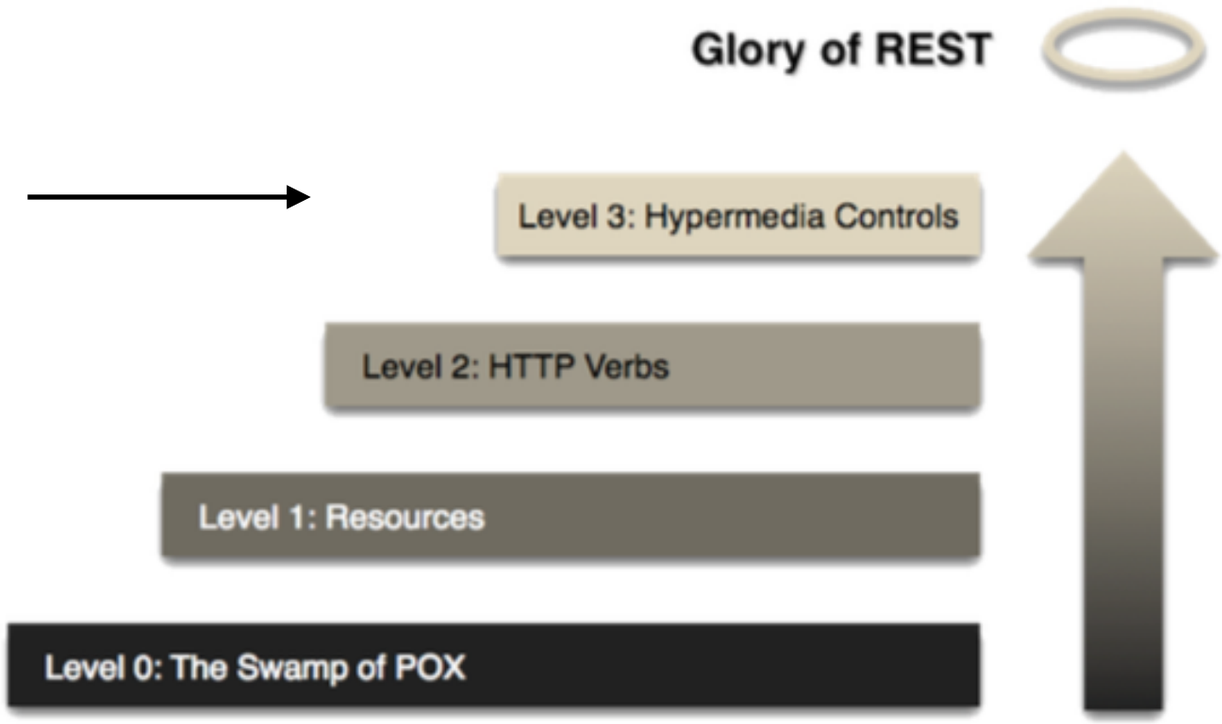
- The best RESTful API offer very simple documentation Why? Because you do not need documentations to navigate a web site

https://dev.twitter.com/rest/public

We need something more, since we would like to automate the resource discover

[Richardson's Maturity Model]

HATEOAS ⟶



[Level 0: The Swamp of Pox]

(Nothing to do with REST)

- One URI, one HTTP method (XML-RPC/SOAP)
- Giant ‘black box’, this is what eBay used to use

```
POST http://svcs.ebay.com/services/search/FindingService/v1

<findItemsByKeywordsRequest xmlns="http://www.ebay.com/marketplace/
search/v1/services">
  <affiliate>
    <networkId>9</networkId>
    <trackingId>1234567890</trackingId>
    <customId>k-man</customId>
  </affiliate>
  <sortOrder>EndTime</sortOrder>
  <paginationInput>
    <entriesPerPage>2</entriesPerPage>
  </paginationInput>
  <keywords>camalots</keywords>
</findItemsByKeywordsRequest>
```

[Level 1: Resource]

(Moving towards REST)

- Each resource has a **unique URI**

- No general end-point (e.g. http://twitterbook/binding/users-api)
  - ✓Instead, there is a URI for each user
  - ✓http://twitterbook/user/1
  - ✓http://twitterbook/user/2

- Single HTTP method/**verb** (usually POST or GET)

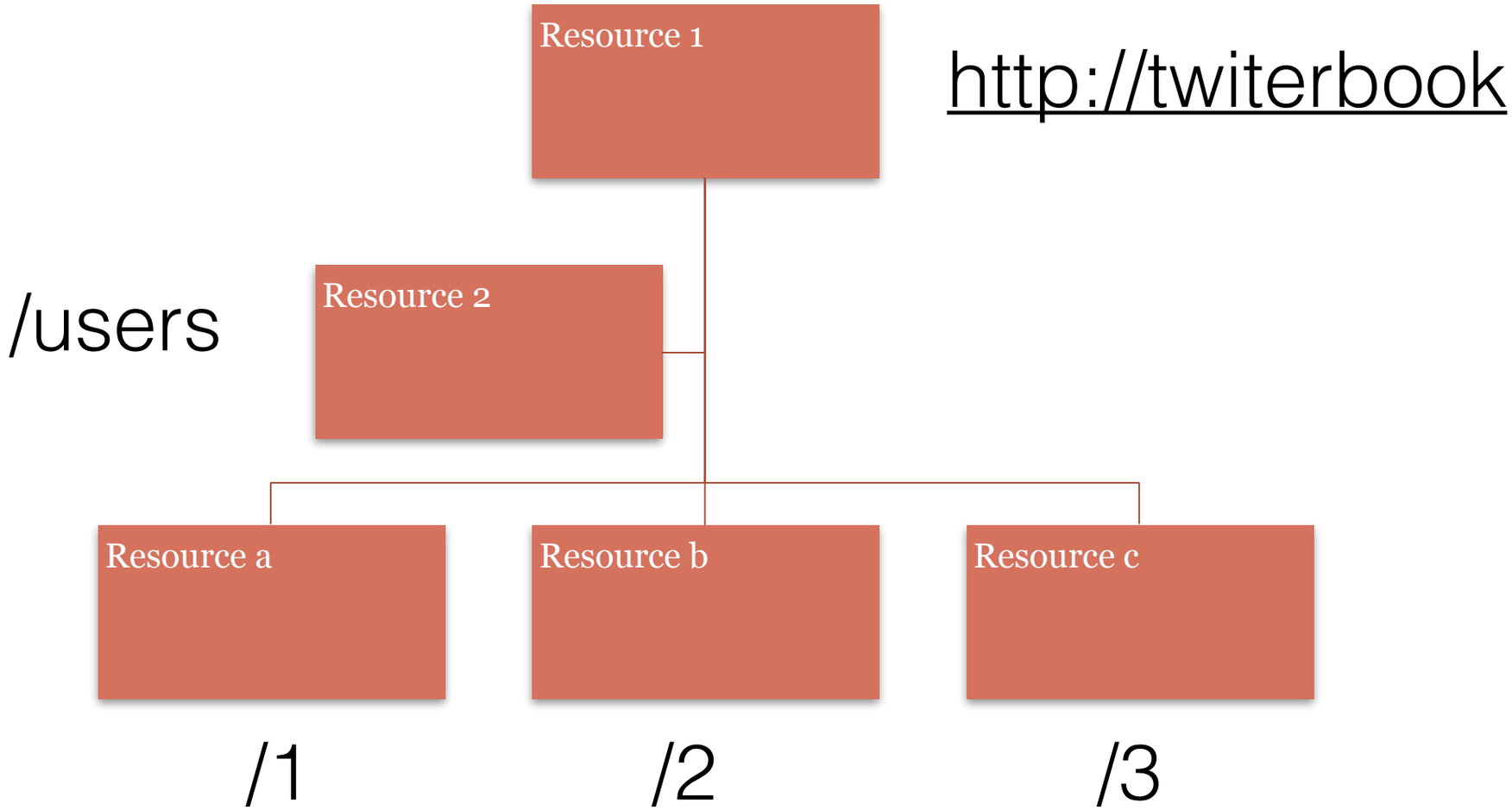
- Usually clients send the name of the method it wants to run

- Early versions of Flicker, Amazon, etc.

```
POST /slots/1234 HTTP/1.1
[various other headers]

<appointmentRequest>
  <patient id = "jsmith"/>
</appointmentRequest>
```

[Level 1: Resource]



[Level 2: HTTP Verbs]

(Generally its REST service)

- Many URIs, using all the HTTP methods/verbs (GET, PUT, DELETE, POST)
- HTTP becomes the application protocol
- Correct use of response codes
  - ✓200 for ok
  - ✓500 for internal error
- Expose state, not behaviour
- **CRUD** services can be useful, e.g. Amazon S3

```
GET /doctors/mjones/slots?date=20100104&status=open HTTP/1.1
Host: royalhope.nhs.uk

HTTP/1.1 200 OK
<openSlotList>
  <slot id = "1234" start = "1400" end = "1450"/>
  <slot id = "5678" start = "1600" end = "1650"/>
</openSlotList>
```

[Level 2: Hypermedia Controls (HATEOAS)]

(True RESTful Service)

- Resources are self-describing
- Expose state and behaviour
- Return useful links

```
GET /doctors/mjones/slots?date=20100104&status=open HTTP/1.1
Host: royalhope.nhs.uk
```

```
<appointment>
  <slot id = "1234" doctor = "mjones" start = "1400"
        end = "1450"/>
  <patient id = "jsmith"/>
  <link rel = "/linkrels/appointment/addTest"
        uri = "/slots/1234/appointment/tests"/>
  <link rel = "/linkrels/appointment/updateContactInfo"
        uri = "/patients/jsmith/contactInfo"/>
</appointment>
```

[HATEOAS (Hyperlinks) (Continued)]

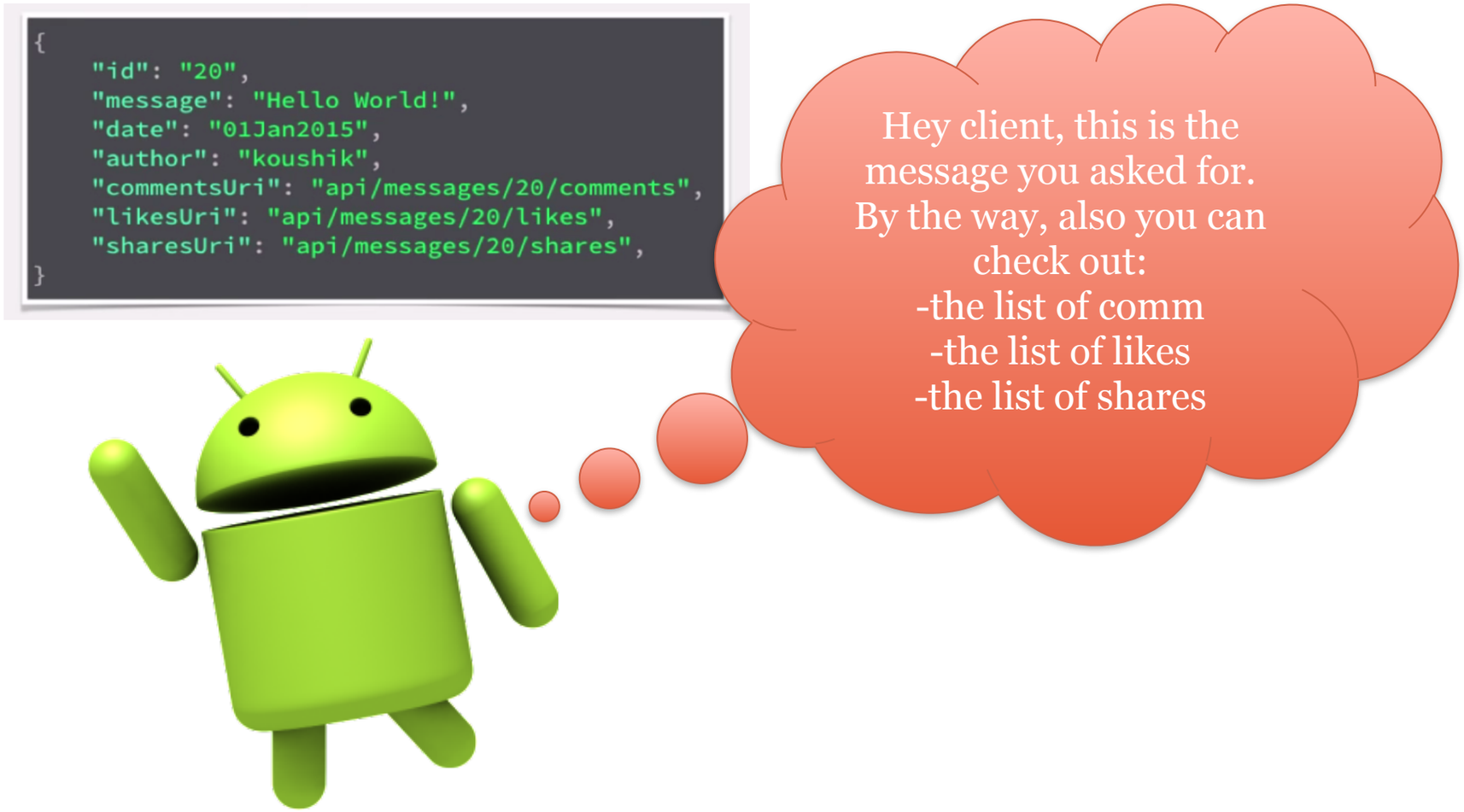
- Describe the current state of the request (in the response)
- Provide links with the response to **transit** to the next state
- HATEOAS makes the surfing of REST possible

> *In each response message, include the links for the next message. Its up to the client to use the link for further requests or nor*

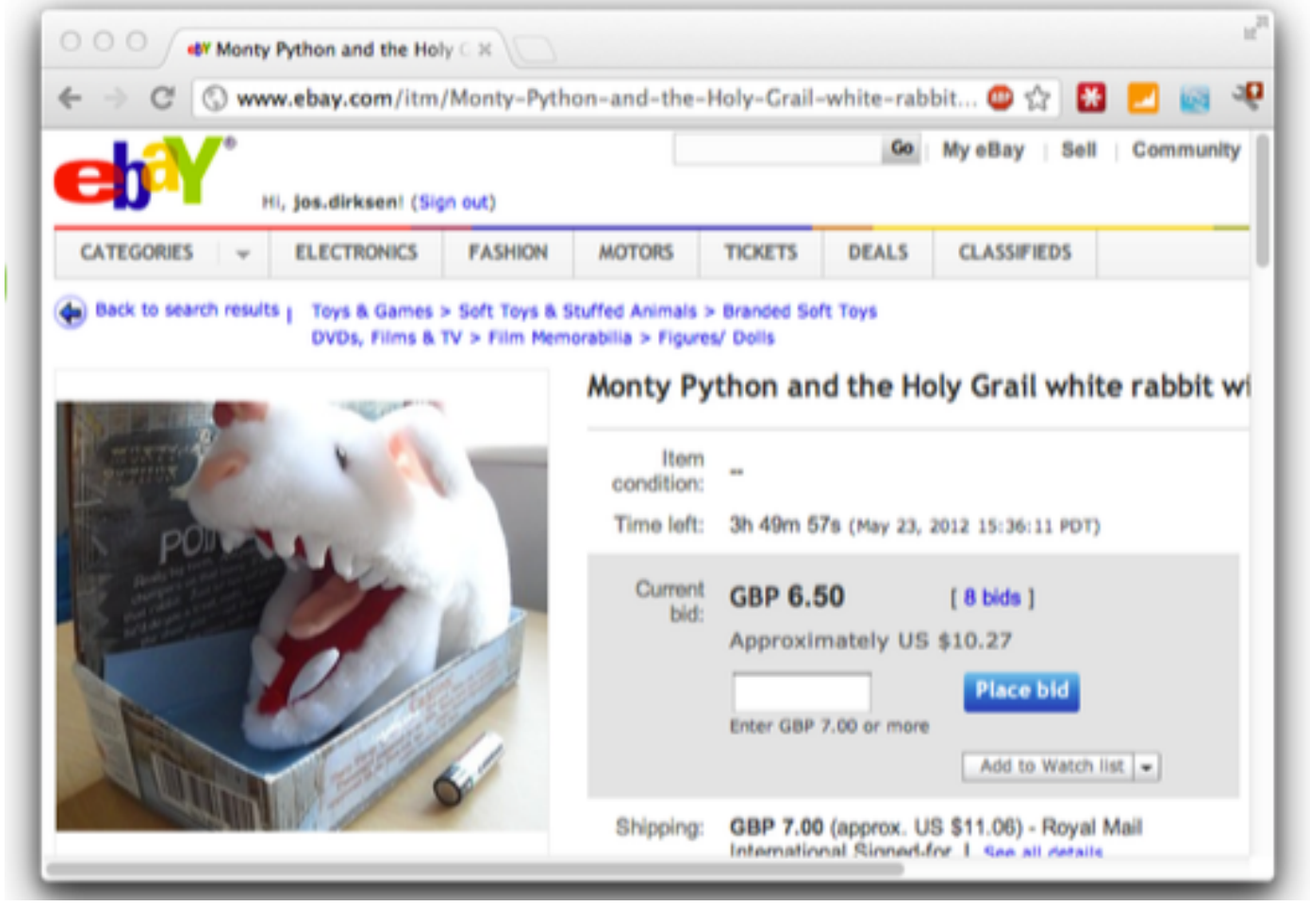
```
GET .../followers/ids.json?cursor=-1&screen_name=josdirksen

{
 “previous_cursor”: 0,
 “id”: {
   “name”: “John Smit”,
   “id”: “12345678”
   “links” : [
   { “rel”: “User info”,
     “href”: “https://.../user/12345678”},
   { “rel”: “Follow user”,
     “href”: “https://.../friendship/12345678”}
   ]
  } // and add other links: tweet to, send direct message,
  ..// block, report for spam, add or remove from list
}
```

[HATEOAS (Hyperlinks) (Continued)]



[Case Study: eBay]

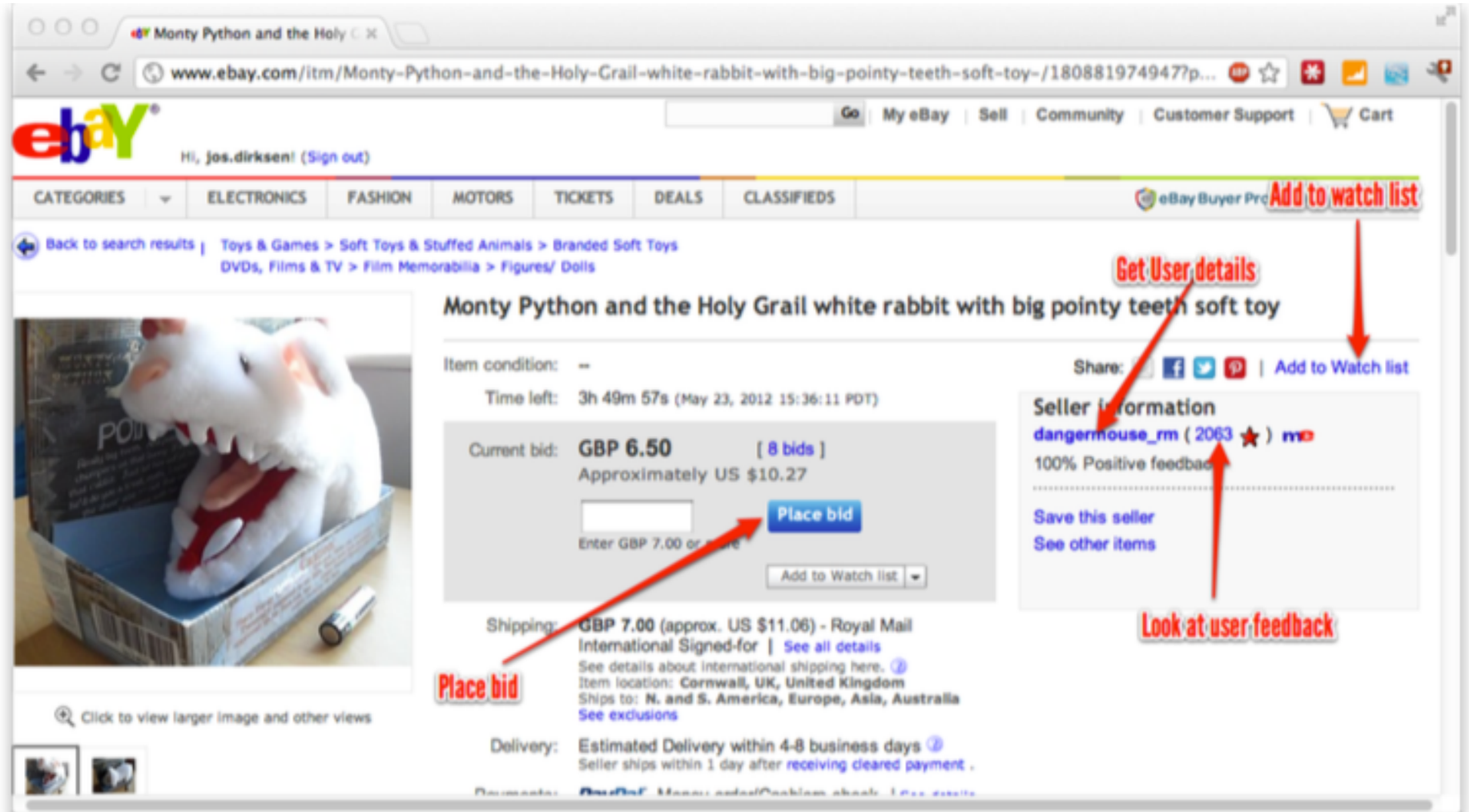


Common scenario: bidding on item

1. Add item to watch list: keep track of the item.
2. Get user details: find out more about the buyer.
3. Get user feedback: is seller trustworthy?
4. Make a bid: place a bid for the item.

eBay API should guide the users/clients

[eBay: API should help the Client]



[Case Study: eBay]

Get all the information of this Item

```
GET .../item/180881974947
{
 “name” : “Monty Python and the Holy Grail white rabbit big pointy teeth”,
 “id” : “180881974947”,
 “start-price” : “6.50”,
 “currency” : “GBP”,
 ...
 “links” : [
   { “type: “application/vnd.ebay.item”,
     “rel”: “Add item to watchlist”,
     “href”: “https://.../user/12345678/watchlist/180881974947”},
   {
     // and a whole lot of other operations
   ]
}
```

- The response should guide the client for future/available functions
- We get the link to add item to watch list

[Case Study: eBay]

Get all the information of this Item

```
GET .../item/180881974947
{
 “name” : “Monty Python and the Holy Grail white rabbit big pointy teeth”,
 “id” : “180881974947”,
 “start-price” : “6.50”,
 “currency” : “GBP”,
 // whole lot of other general item data
 “bidder” : {
    “name” : “dangermouse_rm”,
    “link” : {
      “type” : “application/vnd.ebay.user”,
      “rel : “Get user details”,
      “href : “https://.../user/314512346523”
    }
 }
}
```

- Link to get more details about the user

[Case Study: eBay]

Get all the information of this Item

```
GET .../item/180881974947
{
 “name” : “Monty Python and the Holy Grail white rabbit big pointy teeth”,
 “id” : “180881974947”,
 “start-price” : “6.50”,
 “currency” : “GBP”,
 ...
 “links” : [
   { “type: “application/vnd.ebay.bid”,
     “rel”: “Place bid”,
     “href”: “https://.../user/12345678/bid/180881974947”},
   {
     // and a whole lot of other operations
   ]
}
```

- Link to place a bid

[Richardson's Maturity Model: Case Study-1]

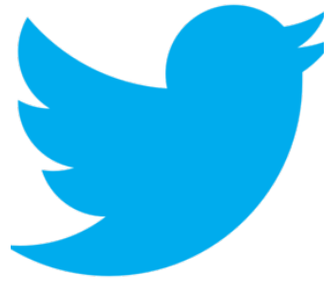
- Mandrill Email Delivery API
- It claims to be most RESTful, however:

  ✓resources has no unique URI

  ✓all HTTP verbs/methods are not used (it only uses POST)

  ✓no links provided by the responses

[Richardson's Maturity Model: Case Study-2]

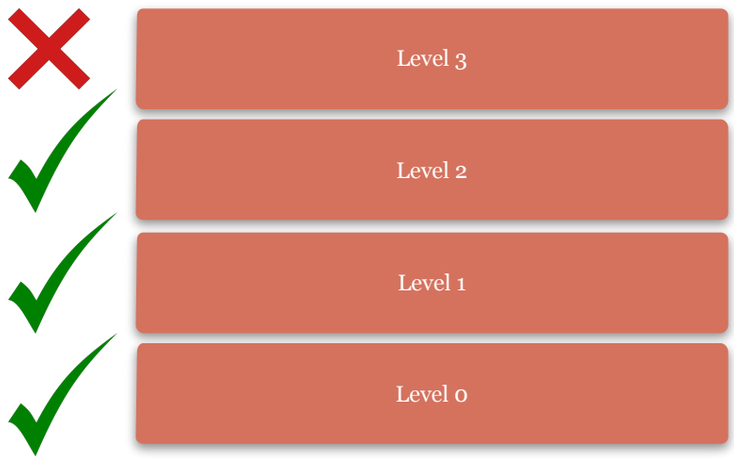


- Twitter API (to get tweets, send tweets, etc.)
- One of the first API to adopt the RESTful principles:

✓end-points are resources (unique URIs) (Level 1 OK)

✓most of the HTTP verbs/methods are used (Debatable but ok, Level 2 OK)

✓responses do not includes links (Level 3 not OK)



[Recall REST Principles]

1. **Resource Identification**

✓You need to know what are the resources we model

2. **Addressability:**

✓URI

3. **Statelessness:**

✓DO NOT track the client state

4. **Resource Representations**

✓Different possible representations

5. **Links & Connectedness**

✓Resources are connected

6. **Uniform interface:**

✓The HTTP methods

[Resource-Oriented Architecture (ROA)]

- Based on these principles we define

Resource-Oriented Architecture

- Composed of Resources
- Their names (URI address)
- Representations (JSON/XML etc.)
- Links between them
- Other properties:
  - ✓ Addressability
  - ✓ Statelessness
  - ✓ Connectedness
  - ✓ Uniform Interface

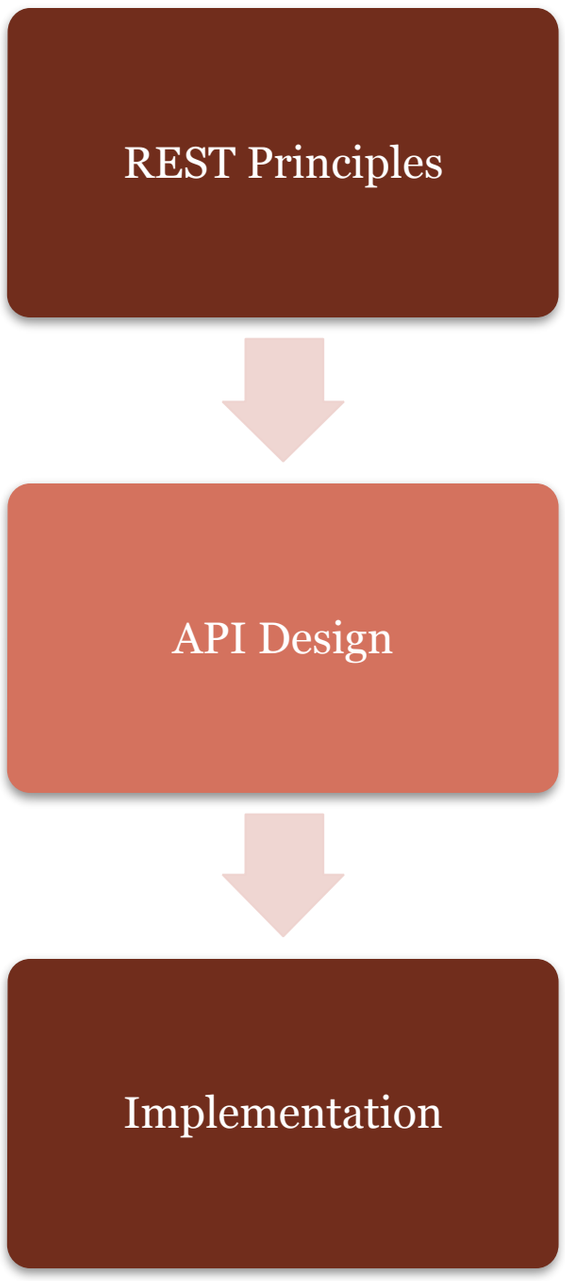
How to create/design a concert Resource-oriented service?

[Design of Resource-Oriented Architecture]

- How should a real data set be split into resources?
- How should the resources be laid out (addresibility)
- What should go into the actual HTTP requests and responses?
- This is called REST API Design

- Resources are usually called **nouns** in RESTful world (e.g. http//twitter/user/1)
- HTTP methods are usually called **verbs** (e.g. GET http//twitter/user/1)

[REST API Design]



Principles proposed by Fielding, and the maturity model

How to define resources and their links

A JAVA based system: JAX-RS (we will see in your TDs)

[Design of a Social Network]

- What kind of resources are expected in a social network
- What kind of actions they can perform (links between them)

- Look for nouns (resources)
- How some verbs can be transformed into nouns

[Design of a Social Network]

- Break a system down into its moving parts: its nouns.
  - ✓ Social Network system will have:
    - A Person
    - A Comment
    - A Like
    - A Photo
- A noun is equivalent to a class. It has Data & a behaviour with other nouns:
  - ✓ Bob posts a photo
  - ✓ Sarah likes a photo
- A noun will become a resource

[REST VS SOAP]

REST

- ✓ Break the the system into nouns
- ✓ A noun is a resource
- ✓ A resource has an URI
- ✓ Actions are: the uniform interface(GET, PUT, POST)

SOAP

- ✓ Break the system into verbs or actions

[REST VS SOAP]

- A publisher web service:
  - In this example, clients can read stories, publish drafts, modify an existing draft write comments and reviews.

- REST style:
  - ✓Resources:
    - ✓Draft, Story, Client, Comment etc.
  - ✓Available actions:
    - ✓GET, POST, DELETE , UPDATE

- SOAP Style
  - Services:
    - ✓PublishStory()
    - ✓UpdateDraft()
    - ✓ReadStory()

[Design of an Online Chess Game]

1- Every **Resource** must be **identified**

[Design of an Online Chess Game]

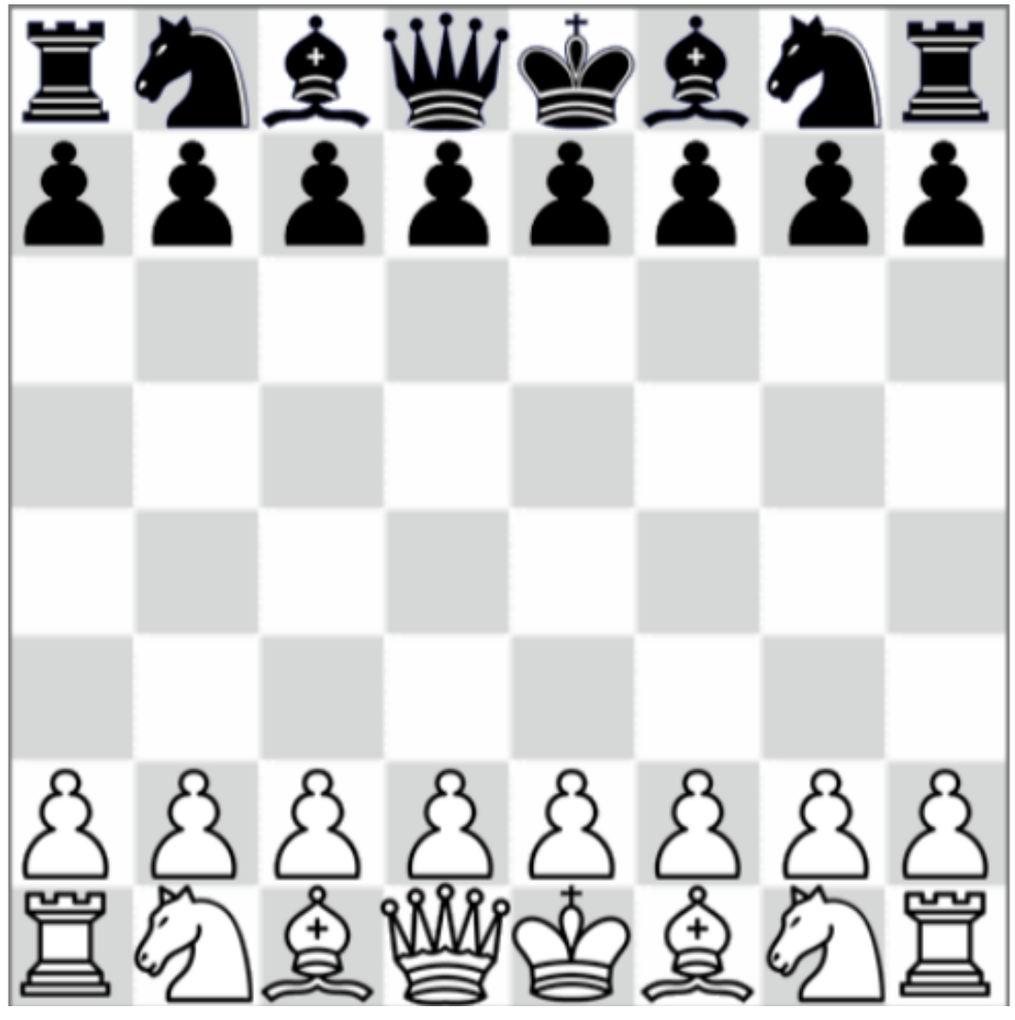
2- Every **Concept** in your domain be a **Resource**

	\BlackBishopOnBlack		\BlackRookOnBlack		\WhiteKingOnBlack
	\BlackBishopOnWhite		\BlackRookOnWhite		\WhiteKingOnWhite
	\BlackEmptySquare		\symbishop		\WhiteKnightOnBlack
	\BlackKingOnBlack		\symking		\WhiteKnightOnWhite
	\BlackKingOnWhite		\symknight		\WhitePawnOnBlack
	\BlackKnightOnBlack		\sympawn		\WhitePawnOnWhite
	\BlackKnightOnWhite		\symqueen		\WhiteQueenOnBlack
	\BlackPawnOnBlack		\symrook		\WhiteQueenOnWhite
	\BlackPawnOnWhite		\WhiteBishopOnBlack		\WhiteRookOnBlack
	\BlackQueenOnBlack		\WhiteBishopOnWhite		\WhiteRookOnWhite

[Design of an Online Chess Game]

2- Design the **URIs** of **Resources** and **Actions**

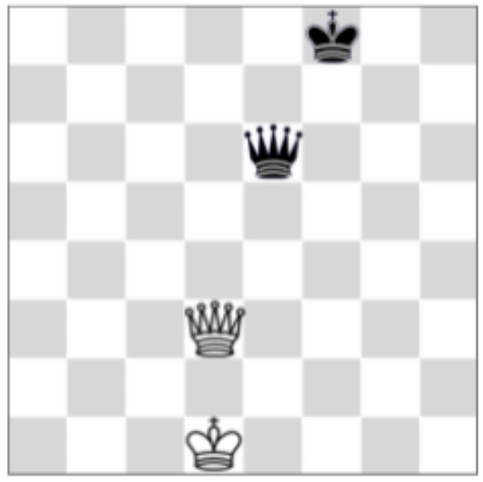
/position/initial



[Design of an Online Chess Game]

/board/4815162342

/position/wqd3,wkd1,bqe6,bkf8



move = d3-d8

/position/wqd8,wkd1,bqe6,bkf8

[Design of an Online Chess Game]

/player/50298

3- For each **Resource** expose a set of **Uniform interface**

GET (*Read*)

POST (*Create*)

PUT (*Update*)

DELETE (*Delete*)

[Design of an Online Chess Game]

```
GET /player/50298 HTTP/1.1
```

Retrieve informations

on identified Resource

Idempotent, should not change the Resource State

[Design of an Online Chess Game]

```
POST /position/wqd3,wkd1,bke8 HTTP/1.1
\n
move = d3-d8
```

```
HTTP/1.1 302 Found
Location: /position/wqd8,wkd1,bke8
```

POST to let Resources process informations

[Design of an Online Chess Game]

```
POST /game HTTP/1.1
\n
white = /player/50298 &
black = /player/39650
```

```
HTTP/1.1 201 Created
Location: /game/42
```

POST to Create new Resources

[Design of an Online Chess Game]

```
PUT /board/4815162342 HTTP/1.1
\n
position = /position/wqd3,wkd2,bke8
```

```
HTTP/1.1 205 Reset Content
```

PUT to Update Resource Informations

[Design of an Online Chess Game]

```
DELETE /board/4815162342 HTTP/1.1
```

```
HTTP/1.1 204 No Content
```

DELETE to

logically Remove Resources

4- For each **Resource** design a **representation form** for a client

```
GET /position/wqd3,wkd2,bke8 HTTP/1.1
Accept: text/plain
```

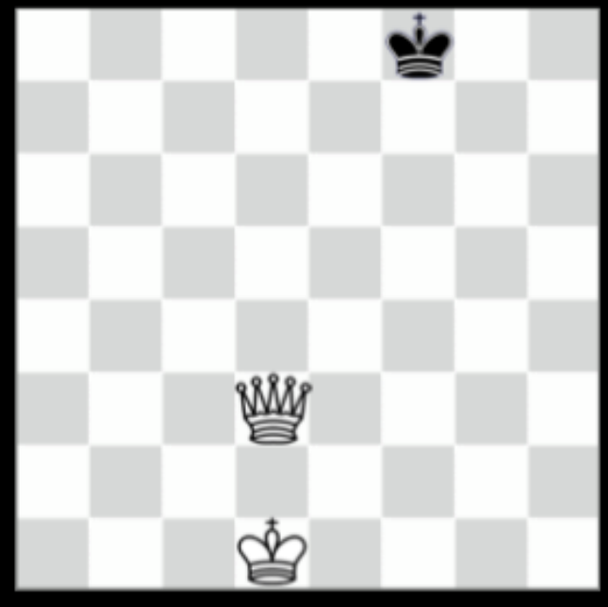
```
HTTP/1.1 200 OK
Content-Type: text/plain;format=fen
\n
3k4/8/3q4/8/8/8/8/5K2
```

Client/server content negotiation

[Design of an Online Chess Game]

GET /position/wqd3,wkd2,bke8 HTTP/1.1
Accept: image/png;q=0.8, text/plain;q=0.2

HTTP/1.1 200 OK
Content-Type: image/png
\n
PNG...



Client/server content negotiation

[Back to Core REST]

REST is not

a Protocol

an Architecture

a Software

a Standard

a fancy name for Web Services

a Buzzword

[Back to Core REST]

REST is

a Software Architecture Style

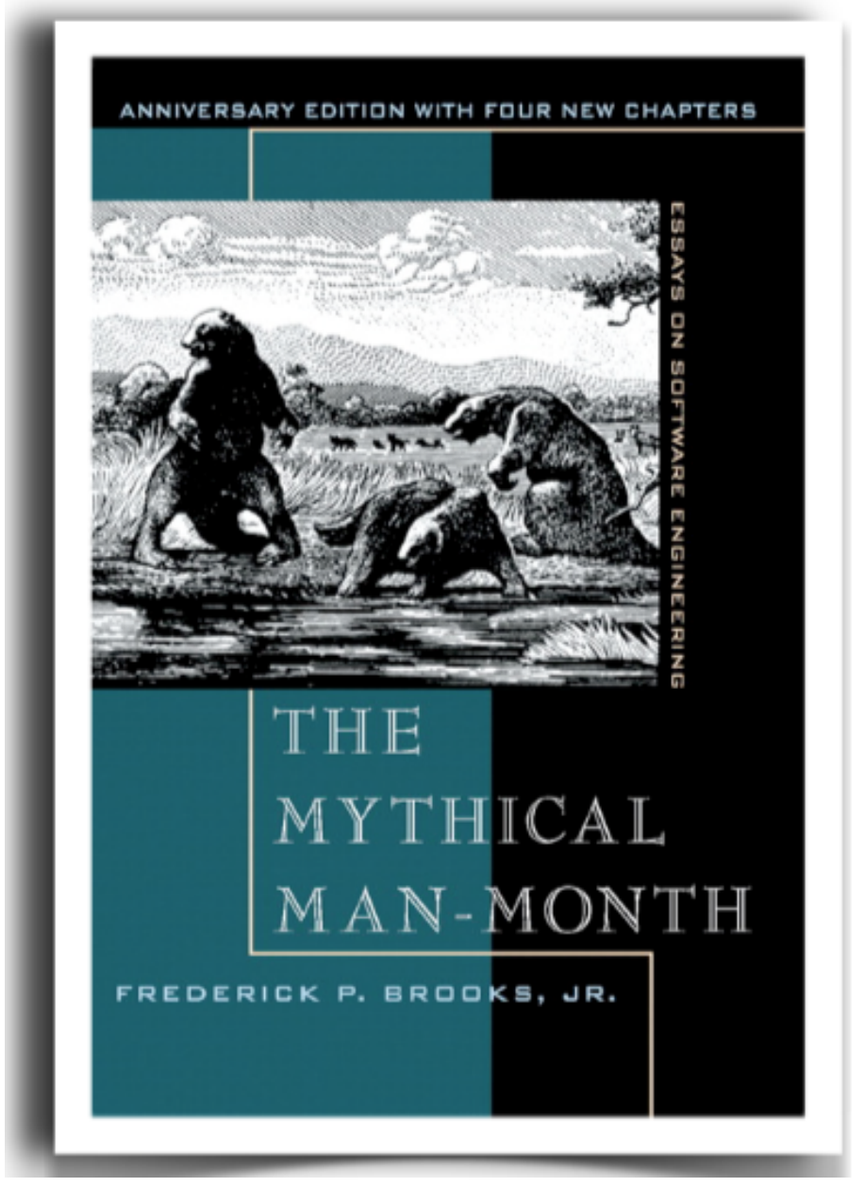
a set of Constraints on Component Interaction

that, when obeyed, cause the resulting

Architecture to have certain properties

[Back to Core REST]

no Silver Bullets



REST vs
SOA vs
SOAP vs
RPC vs
WS-* vs
...

[REST Why?]

"... the motivation for developing REST was to create an architectural model for how the Web should work, such that it could serve as the guiding framework for the Web protocol standards"

[Creating/Using REST with JAVA (JAX-RS)]

[JAX-RS]

- JAVA API for building RESTful services
- Based on **POJOs** with **annotations**
- The developer only focusses on URIs, HTTP methods and media types (XML, JSON, etc.)

POJOs (Plain Old JAVA Objects): an object that compiles under JDK can

Annotations: used sed by the compiler to detect errors or suppress warnings; and to generate code, XML files, and so forth

[JAX-RS]

```java
public class CrunchifyObject {
    public String name;
    public String address;
    public List<Employee> employees;

    // Class Employee
    public class Employee {
        public String firstName;
        public String lastName;
        public int phoneNumber;
```

POJO

CrunchifyObject has 3 fields

Employee Object also has 3 fields

Annotations

JUNIT ANNOTATIONS

@Test

@After

@RunWith

@Parameters

@Before

@Afterclass

@BeforeClass

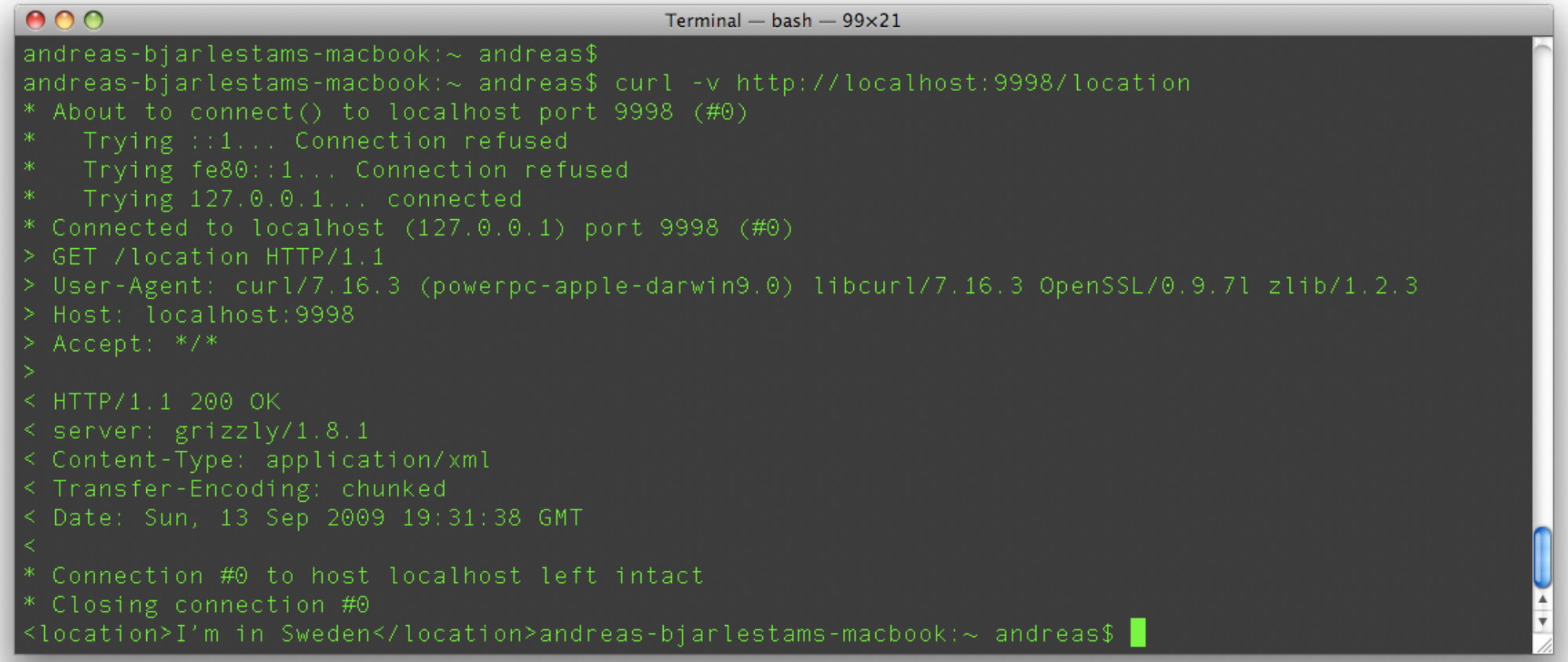
[Lets Code]

[JAX-RS]

http://service.com**/location**

```
@Path("location")
public class LocationResouce {
    @GET
    @Produces("application/xml")
    public String getLocation() {
        return "<location>I’m in Sweden</location>";
    }
}
```

CURL: to get HEAD of the HTTP method

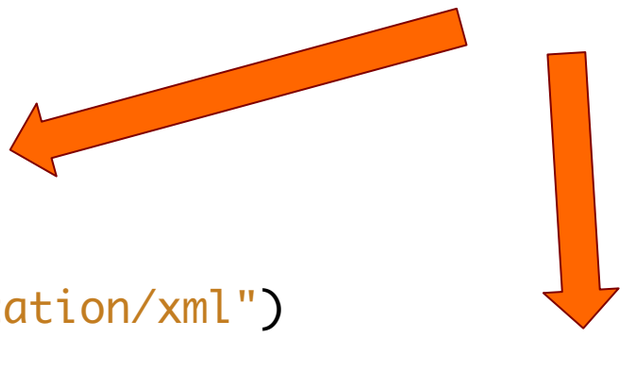


[JAX-RS]

http://service.com**/location/user**

```
@Path("location")
public class LocationResouce {

    @GET
    @Path("{user}")
    @Produces("application/xml")
    public String getLocation(@PathParam("name") String user) {
        return "<location>” + user + " is in Sweden</location>";
    }
}
```



Path and Query Parameters

http://service.com**/location/user?name=5**

[JAX-RS]

```
package com.example;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import com.thoughtworks.xstream.XStream;

@Path("/customers")
public class Customers{

        @GET
        public String getAllCustomers() {
                return "list of customers";
        }

        @GET
        @Path("/{id}")
        public String getCustomer(@PathParam("id") int id) {
                XStream xstream = new XStream();
                Customer customer = new Customer(id);
                return xstream.toXml(customer);
        }

        @GET
        @Path("/{id}/address")
        public String getAddress(@PathParam("id") int id) {
                Customer customer = new Customer(id);
                return customer.getAddress();
        }
}
```

Client Request

```
GET / HTTP/1.1
Host:
http://localhost:8081/TestRestfulService/rs/
customers
```

Client Request

```
GET / HTTP/1.1
Host:
http://localhost:8081/TestRestfulService/rs/
customers/1234
```

Client Request

```
GET / HTTP/1.1
Host:
http://localhost:8081/TestRestfulService/rs/
customers/1234/address
```

[Fin]

Acknowledgment: Michael P. Papazoglou, Web Services, 1st Edition, © Pearson Education Limited 2008