Enterprise Programmering 2

Lesson 04: REST and 3xx

Dr. Andrea Arcuri

Goals

- Understand the basics of RESTful APIs
- Understand 3xx HTTP redirections

RESTful APIs

RESTful APIs

- **Re**presentational **S**tate **T**ransfer (REST)
- Most common type of web services
- Access to set of resources using HTTP
- REST is *not a protocol*, but just architectural guidelines on how to define HTTP endpoints
  - Example: should not delete a resource when answering a GET, but no one will stop you from implementing an API that does that
- Introduced in a PhD thesis in 2000

REST Constraints

1. Uniform Interface
2. Stateless
3. Cacheable
4. Client-Server
5. Layered System
6. Code on demand (optional)

1: Uniform Interface

- Resource-based, identified by a URI
- The actual resource could be anything
  - e.g., rows in a SQL database, or image files on disk
- Client sees a *representation* of the resource, and the same resource can be given in different formats
  - eg, XML, JSON and TXT
- Hypermedia as the Engine of Application State (HATEOAS)
  - Resources connected by links… hardly anyone uses it… we will go back to this point later in the course

2: Stateless

- Resources could be stored in databases or files
- But the web service itself should be stateless
- This means that all info to process a request should come with the request itself
  - eg, as HTTP headers
- Consequence examples:
  - can restart process of web service at any time
  - horizontal scalability: can have 2 more instances of same service, does not matter which one is answering and in which order

3: Cacheable

- Cacheable: avoid making a request if previous retrieved data is still valid
- Very important for scalability
- Resources should define if they are cacheable or not, and how

4: Client–Server

- Clear cut between clients and servers
- Client only sees the URIs and the representation (eg JSON), but no internal details of server
  - eg does not even know if resource is stored in a database or on file
- Server does not know how data used on clients
- Consequence: clients and servers can be developed/updated independently, as long as URIs/representation are the same

5: Layered System

- For clients, should not matter if there is any intermediary on the way to the server
- Typical example: *reversed proxy*
  - eg, used for load balancing and access policy enforcement

6: Code on Demand (optional)

- Servers can temporarily extend or customize the functionality of a client by transferring executable code
  - eg, transfer JavaScript code
- Among the constraints that define REST, this is optional

The Term “REST”

- Most APIs out there are called REST by their developers…
- … but “technically”, they aren’t
- For example, nearly *no one* uses HATEOAS
- So, nowadays, REST loosely means: “*A web API where resources are hierarchically structured with URIs, and operations follow the semantics of the HTTP verbs/methods*”

Example for a Product Catalog

- Full URLs, eg **www.foo.com/products**
- GET **/products**
  - (return all available products)
- GET **/products?k=v**
  - (return all available products filtered by some custom parameters)
- POST **/products**
  - (create a new product)
- GET **/products/{id}**
  - (return the product with the given id)
- GET **/products/{id}/price**
  - (return the price of a specific product with a given id)
- DELETE **/products/{id}**
  - (delete the product with the given id)

Resource Hierarchy

- Consider the resource: **/users/3457/items/42/description**
- **/users**: resource representing a set of users
- **/3457**: a specific user with that given id among the set of users */users*
- **/items**: a set of items belonging to the user 3457
- **/42***:* a specific item with id 42 that the user 3457 owns
- **/description**: among the different properties/fields of item 42, just consider its *description*

Cont.

- *GET **/users/3457/items/42/description***
- It means: retrieve the description of item with id 42, which belongs to the user with id 3456
- But what about *GET **/items/42/description*** ???
- “Technically”, they would be 2 *different* resources, because there are two different URIs
- But in practice, they are the same

Backend Representation

- ***/users/3457/items/42/description***
- Could be two different tables in a SQL database, eg *Users* and *Items*
- Or could be a single JSON file on disk…
- or the REST API just collects such data from two other different web services…
- or whatever you fancy…
- Point is, for the client this does not matter at all!

Available URIs

- 1st) *GET* ***/users/3457/itemIds***
- 2nd) *GET* ***/items/42/description***
- It means: first retrieve the ids of all items belonging to user 3457. Then, to get description for a specific one of them with id 42, make a second GET
- But in the 2nd GET, what if we rather used ***/users/3457/items/42/description*** ???

Cont.

- 1st) *GET* ***/users/3457/itemIds***
- 2nd) *GET* ***/items/42/description***
- 3rd) *GET* ***/users/3457/items/42/description***
- Whether the 2nd or the 3rd (or both) endpoint is needed depends on how clients will typically interact with the API
  - do they need to access to items regardless of their user owners?
- Point is: you need to *implement* a handler for each endpoint

Path Elements

- ***/users/3457/items/42/description***
- How does a client know that ***/users*** and ***/items*** are collections/sets but not ***/description*** ?
- “Technically”, each of those tokens are path elements, with no specific semantics
- Client has to read the documentation of the API
- However, to make things simpler, it is a convention to use *plural* names for set resources

Resource Filtering

- Assume you want to retrieve all users that are in Norway
- 1st) *GET **/users/inNorway***
- Problem is, what if you still want to retrieve single users by id?
- 2nd) *GET **/users/{id}***
  - Where {} just represents a variable matching any single path element input
- Ambiguity: here ***/users/inNorway*** would be matched by both endpoints
  - ie, *inNorway* could be treated as a user id

Cont.

- 1st) *GET* ***/users/inNorway***
- 2nd) *GET* ***/users/byId/{id}***
- Here there would be no ambiguity, but…
- … what would be the semantics of the intermediate resource ***/users/byId*** ???
- Paths in the URIs should represent resources, and not actions on them

Cont.

- 1st) *GET **/users?country=norway***
- 2nd) *GET **/users/{id}***
- When we want to apply a filter to get a subset of a collection, then we use *query parameters*
  - recall URIs: start with “?”, followed by pairs <key>=<value>
- Extra benefit: we can later add extra filter options (e.g., *ageMin=18*), without altering the routing of requests to the endpoint ***/users***

Resource Creation

- *POST **/users***
  - POST operation on a collection
  - Payload used to create new element added to the collection
  - Response will have *Location* HTTP header telling where to find the newly created resource, eg *Location:/users/42*
- *PUT **/users/42***
  - PUT operation directly on the URI of the new resource
  - Need to specify id

Cont.

- Which one to use? POST or PUT?
- When id is chosen by server (eg linked to an id from SQL database), you need POST
- If you use PUT, client must choose the id, and it must be *unique*
  - otherwise, you would just overwrite an existing resource

PUT vs POST

- *1st) GET /users/42* => Response 404
- *2nd) PUT /users/42*
- This would make no sense, because:
  1. Not going to do hundreds of GETs until find one with 404 Not Found
  2. Two HTTP requests in sequence are not necessarily atomic, eg, before PUT is executed, someone else could have create the resource, and you would just then overwrite it

Cont.

- 1st) *POST **/users*** => Location: */users/42*
- 2nd) *PUT **/users/42/address***
- Assume you create a new user with a POST operation, but without an *address*
- You could then want to create the *address* resource directly by using a PUT
  - point is that the resource does not have an id in itself, but rather the id is in a path element ancestor
- However, most of the time you would not expose each single field of an object as its own URI endpoint, but rather do a PATCH
  - eg, *PATCH /users/42*

Resource Representation

- 1st) *GET /users/42*
- 2nd) *GET /users/42**.json***
- 3rd) *GET /users/42**.xml***
- For what you know, the REST service could store users in a SQL database or a CSV file
- What you get is a *representation* of a resource, which can be in different formats, based on client's needs
- But what's the problem here?

Cont.

- 1st) *GET /users/42*
- 2nd) *GET /users/42**.json***
- 3rd) *GET /users/42**.xml***
- Because the URIs are different, they are technically 3 *different* resources
  - whether they map to the same entity on the backend is another story…
- A URI has no concept of type: adding a "*.json*" extension does NOT change the semantics

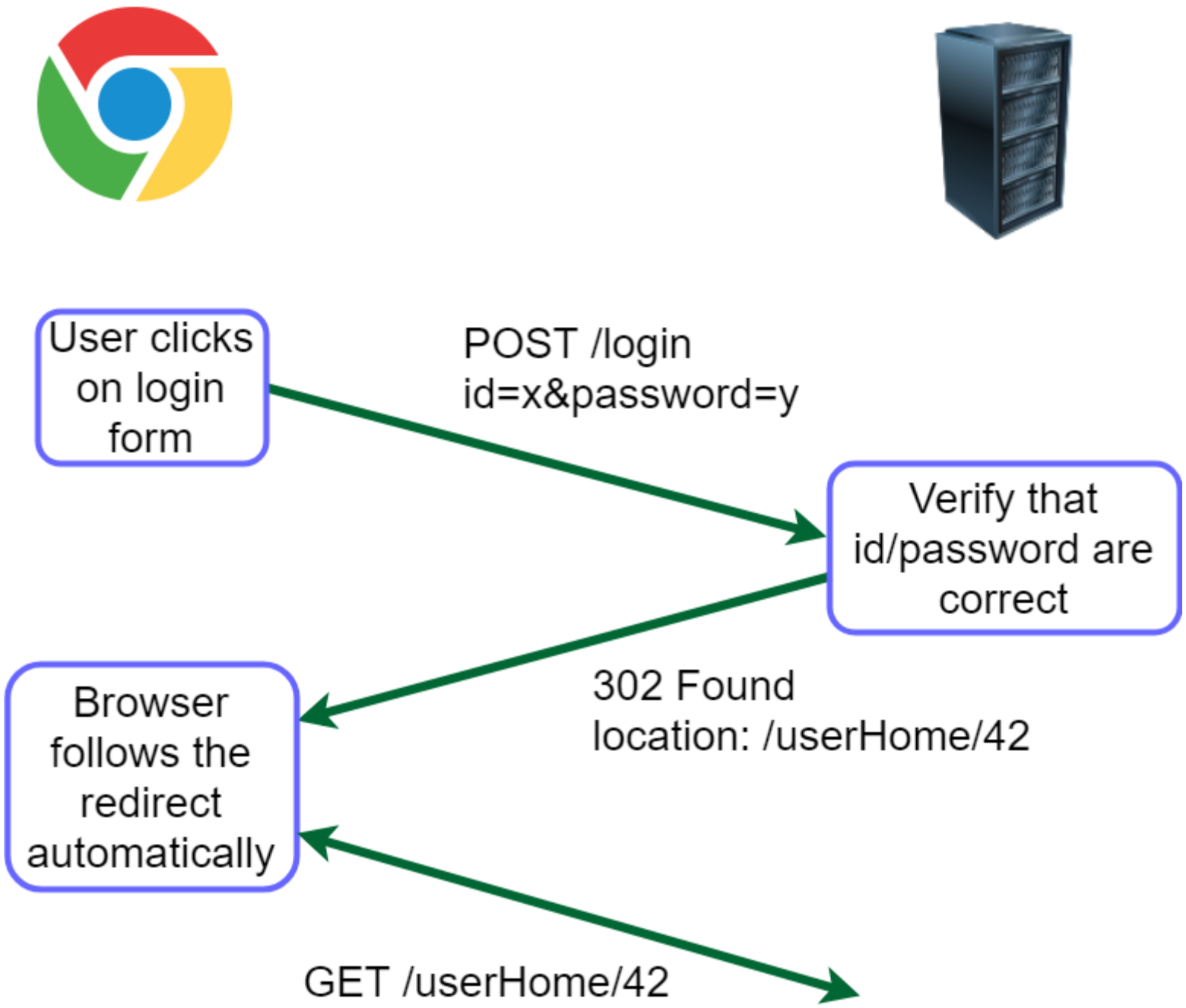
Cont.

- *GET **/users/42***
- You should avoid type extensions on your resources
  - although you might see many APIs doing it…
- Choosing among different types should be based on HTTP headers like *Accept*
  - eg, "*Accept: application/json*"
- If a client asks for a specific representation (eg XML), that does not mean that the server would support it
- If Accept missing, or generic */*, server would just use the default representation (e.g., JSON)

3xx Redirection

3xx Status Code

- They represent *redirection*
- You ask for a resource at URI X, but then the server tells you should rather go to URI Y
  - "where" to go will be specified in the *Location* header
- … or operation on X is completed, and result is visible at Y
- Example in a browser: how to tell the client to automatically go to homepage after a successful login on the login page?


User clicks on login form
POST /login
id=x&password=y
Verify that id/password are correct
302 Found
location: /userHome/42
Browser follows the redirect automatically
GET /userHome/42

Messy Standard

- The HTTP standard is a mess when it comes to 3xx status codes
  - ie, lot of ambiguities and undefined behavior
  - eg, see http://insanecoding.blogspot.no/2014/02/http-308-incompetence-expected.html
- You should use redirection when needed, but keep it minds that different clients might have different, strange behaviors
- The main issue is on how HTTP methods could be changed
  - eg, in previous example, a POST was redirected into a GET

Permanent Redirection

- You ask for X, but server tells you that now it is *permanently* moved to Y
- A client, if it follows redirects automatically, will do a new request to Y
- From now on, every time you ask for X, the client would rather call for Y directly, and never use X again
  - as the redirection is *permanent,* there is no point in asking for X, you can just go directly for Y

Temporary Redirection

- You ask for X, but server tells you that now it is *temporarily* moved to Y
- A client, if it follows redirects automatically, will do a new request to Y
- Every time you ask for X, the client will still ask for X, and ignore the previously obtained Ys
  - as the redirection is *temporary,* each time you ask for X you could get a different Y’

3xx Codes for RESTful APIs

- 301: Permanent redirection, but use it *only* for GET
  - unless you like random surprises, like clients transforming a PUT into a GET
- 304: For cache control
  - eg no need to retrieve resource, as the one in cache is still valid
- 307: Temporary redirection
- 308: Permanent redirection, for methods other than GET
  - note: many client libraries will not follow such redirect automatically, so do not rely too much on it

Git Repository Modules

- *NOTE: most of the explanations will be directly in the code as comments, and not here in the slides*
- **advanced/rest/redirect**
- **advanced/rest/news-rest-v2**
- Study relevant sections in *RESTful Service Best Practices*
- Study RFC-7538
- Study relevant sections in RFC-7230 and RFC-7231