INFO-H-511: Web Services
TP 2 - Developping REST services

Lecturer: Stijn Vansummeren
Teaching Assistant: Francois Picalausa
http://cs.ulb.ac.be/public/teaching/infoh511

2011–2012

Part I: Handling HTTP requests

In this exercise, we will handle HTTP requests in order to implement an Hello World REST Service. This service consists of a single Hello World resource the is capable of serving some text data with the GET method and of updating that data with the PUT method. For this purpose, we build on the well known Jetty web server to accept and parse HTTP requests.

The skeleton code provided on the labs website instantiates a Jetty server, and calls the `handle` method of `SimpleResourceServer` to handle HTTP requests. Furthermore, the `getHello` and `changeHello` methods constitute the model supporting the Hello World resource.

Exercise 2.1

We ask you to implement the following and to test your service accordingly using the command-line tool `curl`.

- Fill in the `handle` method to answer the client with "Hello World" in plain text, for every request.
- Update the code in such a way that only the GET requests directed to `/hello` are processed. Other HTTP methods should be met with the "Method Not Allowed" status code[1] and other URIs should be "Not Found".
- Using the `acceptMatcher` object that is provided as part of the skeleton code to answer queries according to their desired media-type (defined in the "Accept" HTTP header). You should only serve media-types supported by the `getHello` method; if no media-type can be found, answer with the appropriate HTTP status code.
- Implement the PUT method such that subsequent GET return the message last PUT. Pay attention to the Content-Type header, and to the status code being returned.

Part II: Designing REST Web-Services

In the first part of this lab, we have explored ways of manually handling HTTP queries. To develop full-fledged REST services, higher level APIs are recommended. For instance, Java includes a specification (JAX-RS[2]) that aims to ease the development of REST services. In the following exercises, we will implement a contact manager service on top of the Jersey implementation of this specification. This service will rely on Apache Adbera[3] to provide its data as Atom feeds, following the AtomPub recommendation[4].

[1] `http://tools.ietf.org/html/rfc2068#section-6.1.1`
[2] `http://jsr311.java.net/`, `http://docs.oracle.com/javaee/6/tutorial/doc/gilik.html`
[3] `http://abdera.apache.org/`
[4] RFC 5023, `http://tools.ietf.org/html/rfc5023`

The skeleton code provided on the labs web page instantiates a Grizzly server, with support for JAX-RS. This skeleton also implements a model for the contacts list, which allows adding, removing, editing, and listing contacts, as well as managing groups of contacts.

Exercise 1.2

Identify the resources that will be provided by your web service. For each of these resources, indicate (1) which URIs will be available, (2) the HTTP methods that each URI will support, and (3) the semantics of these methods from the service point of view. In particular, the semantics should include the types of Atom document that will be served or consumed.

To help you in this task, consider the following scenarios.

- A new contact is created for "John Doe", with the email address `johndoe@company.org`.
- The last name of "John Smith" is changed to "Smithe"
- The "friends" group is rechristened the "party all night long" group.
- "Joe Bloggs" is removed from the contact manager.
- The "friends" group is rechristened the "party all night long" group.
- "Jane Doe" is removed from the "acquaintances" group and added to the "friends" group.

Exercise 1.3

In this exercise, we create a basic service that does not manage contact groups. We ask you to implement the following, and to test your service using the command-line tool `curl`[5].

- Fill in the `ServicesDocument` class with the Atom collections that you have identified.
- Create classes corresponding to the URIs you have defined, and implement the corresponding GET methods. Note that helper functions are provided in the `be.ac.ulb.code.wit.resources.serializers` package.
- Implement the remaining methods.

Exercise 1.4

Add groups to the implementation of the service of Exercise 1.2.

[5] More formal unit testing is possible using e.g. the Jersey HTTP client