REST-based Web Services (III)

Introduction to Service Design and Engineering 2013/2014.

Lab session #6

University of Trento

Cristhian Parra: Lab Lectures for IntroSDE-2013/2014
Some slides from LabSDE-2012/2013

Outline

- Exercise on making an HTTP java client and Pseudo REST APIs
- CRUD Restful API Example
- Assignment #2 Details

Exercise 1: HTTP Clients

- Take a look at the HTTP Restful Client example of lab6
- Create a client like the one of lab6, but in this case, it must communicate with a WEBDIS Pseudo-REST front-end to the Redis store

Exercise 1: WEBDIS Server details

The Pseudo-REST WEBIDS API (quiz: **why is Pseudo-REST and not entirely REST?**)

- BASEURL = http://test.lifeparticipation.org/webdis/ (please, be polite with this server)
- API Endpoints:
  1. **Store a {key,value}:** GET /SET/key/value
  2. **Retrieve a {key,value}** GET /GET/key
  3. **Store a {hashkey,{innerkey,value}}:** GET /HMSET/hashkey/innerkey1/value1/innerkey2/value2
  4. **Retrieve the value for {hashkey,innerkey}:** GET BASEURL/HGET/hashkey/key
  5. **Retrieve all the {innerkey,value} for hashkey:** GET /HGETALL/hashkey
- **Notes:**
  - 3 and 4 can recieve more than one *{key,value}*
  - The character "/" must be replaced by "%2f" in values and keys
  - The character "." must be replaced by "%2e" in values and keys
  - All these commands are actually supported

Exercise 1: WEBDIS Server Examples

- Some examples to test (just click on the links)
  - http://test.lifeparticipation.org/webdis/SET/1/pinco pallino
  - http://test.lifeparticipation.org/webdis/GET/1
  - http://test.lifeparticipation.org/webdis/HMSET/pinco pallino/weight/78/height/1%2e67
  - http://test.lifeparticipation.org/webdis/HGETALL/pinco pallino
  - http://test.lifeparticipation.org/webdis/HMGET/pinco pallino/height/weight
- **Notes:**
  - Don't break the server, be polite with your requests :-)
  - be mindful that you will all be querying the same key,value database, so you might want to use different "keys"
  - Solution to the exercise is here

Lab Examples

- Start by creating a Web Dynamic Project (as in the last session) with the name **CRUD RESTful**
- Add three packages to your project:
  - **introsde.crud.rest.dao**
  - **introsde.crud.rest.model**
  - **introsde.crud.rest.resources**

Example 1: Standalone HTTP Server (1)

- For testing purposes, it is reccomended to use an standalone HTTP server instead of tomcat.
- This is an example that you can use as baseline

Example 1: Standalone HTTP Server (2)

- Create this application on your **introsde.crud.rest.resources** package
- Run it as an standard Java Application

```
// the whole resource will be available at baseUrl/rest
@Path("/rest")
public class StandaloneServer
{
    public static void main(String[] args) throws IllegalArgumentException, IOException
    {
    String protocol = "http://"; // of course...
        String port = ":5900/"; // you can use any other as long as it is not in use
        String hostname = InetAddress.getLocalHost().getHostAddress();
        if (hostname.equals("127.0.0.1"))
        {
            hostname = "localhost";
        }
        String baseUrl = protocol + hostname + port;
        final HttpServer server = HttpServerFactory.create(baseUrl);
        server.start();
        System.out.println("Server starts on " + baseUrl + "\n [kill the process to exit]");
    }
}
```

Example 2: Simple CRUD Jersey API (1)

- Check the structure of the Example
  - **DAO:** stands for *data access objects* and is where our data providers are, for this example, we use a Singleton design pattern to implement a mock of a in memory Database.
  - **Models:** the classes that define our data model. Notice the use of JAXB annotations to allow Jersey to automatically find the way of marshalling and unmarshalling objects to xml
  - **Resources:** where our service endpoints are implemented. Notice that in PeopleResource we reference the PersonResource. People is a collection resource that aggregates PersonResources
- Add these classes to your project

Example 2: Simple CRUD Jersey API (2)

DAO: for this simple example, our Data Provider is a single HashMap that associates string ids with person objects

```
public enum PersonDao {
  instance;
  private Map<String, Person> contentProvider = new HashMap<String, Person>();
  private PersonDao() {
    Person pallino = new Person();
    Person pallo = new Person("Pinco","Pallo");
    HealthProfile hp = new HealthProfile(68.0,1.72);
    Person john = new Person("John","Doe",hp);
    pallino.setId("1");
    pallo.setId("2");
    john.setId("3");
    contentProvider.put("1", pallino);
    contentProvider.put("2", pallo);
    contentProvider.put("3", john);
  }
  public Map<String, Person> getModel() {
    return contentProvider;
  }
}
```

Example 2: Simple CRUD Jersey API (3)

- **Models:** our model is composed by our typical Person/HealthProfile model (with the addition of basica JAXB Annotations)
- **Resources:** our resources allow reading the list of people, one person by id, the count of people in the database and the creation, update and delete of one person

Example 2: Simple CRUD Jersey API (4)

- Since People wraps the PersonResource, we need to hava a way of passing the *Request* information to the PersonResource.

```
// Allows to insert contextual objects into the class,
// e.g. ServletContext, Request, Response, UriInfo
@Context
UriInfo uriInfo;
@Context
Request request;
```

Example 2: Simple CRUD Jersey API (5)

Reading People collection:

```
// Return the list of people for applications
@GET
@Produces({ MediaType.APPLICATION_XML})
public List<Person> getPersonListXML() {
  List<Person> people = new ArrayList<Person>();
  people.addAll(PersonDao.instance.getModel().values());
  return people;
}
```

Returning the number of People in our database

```
@GET
@Path("count") // corresponds to /person/count
@Produces(MediaType.TEXT_PLAIN)
public String getCount() {
  System.out.println("Getting count...");
  int count = PersonDao.instance.getModel().size();
  return String.valueOf(count);
}
```

Example 2: Simple CRUD Jersey API (6)

Accesing single elements in the people collection (i.e., person resources)

```
// Return the list of people for applications
@GET
@Produces({ MediaType.APPLICATION_XML})
public List<Person> getPersonListXML() {
  List<Person> people = new ArrayList<Person>();
  people.addAll(PersonDao.instance.getModel().values());
  return people;
}
```

Calling PersonResource Endpoints

```
// Defines that the next path parameter after the base url is
// treated as a parameter and passed to the PersonResources
// Allows to type http://localhost:599/person/1
// 1 will be treaded as parameter todo and passed to PersonResource
@Path("{personId}")
public PersonResource getPerson(@PathParam("personId") String id) {
  return new PersonResource(uriInfo, request, id);
}
```

Example 2: Simple Create of a Person (6)

Creating the person (update and delete in the PersonResource)

```
@Produces(MediaType.APPLICATION_XML) // will be called when content-type header set to xml
@Consumes(MediaType.APPLICATION_XML)
public Person newPerson(Person person) throws IOException {
  System.out.println("Creating new person...");
  int count = PersonDao.instance.getModel().size();
  String newId = count+1+"";
  person.setId(newId);
  PersonDao.instance.getModel().put(newId, person);
  return person;
}
```

Exercise 2: Extending the simple CRUD API

- Add a history attribute to the "HealthProfile" where a new value will be attached to the list of a measure every time this is updatedß
- Add a service to get the history of a measure
- **Where should these services go?**

Assignment #2: Part 1

- Create a model that supports
  - **People** identified by an **id** and having at least *birthdate*, *first* and *lastname*
  - A Health/Lifestyle profile for each person, with measures such as **weight** and **height** (and others if you like)
  - The history of these measures by date

Assignment #2: Part 1

- Examples of how the model look like (you are not required to reproduce this, this is only as a minimum example)

// person/healthprofile

```
{
      "firstname"     : "Chuck",
      "lastname"      : "Norris",
      "birthdate"     : "1945-01-01"
      "healthProfile" : {
                "weight"  : 78.9,
                "height"  : 172
      }
}
```

// history of one measure (e.g., weight)

```
[
    {
        "mid" : 992,
        "value" : 78.9,
        "created" : "2007-12-09"
    },
    {
        "mid" : 999,
        "value" : 75,
        "created" : "2009-12-09"
    },
    {
```

Assignment #2: Part 1

// person/health profile

```
<person>
    <firstname>Chuck</name>
        <lastname>Norris</lastname>
        <birthdate>1945-01-01</birthdate>
        <healthProfile>
            <weight>78.9</weight>
            <height>172</height>
        </healthProfile>
</person>
```

// history of one measure (e.g., weight)

```
<measure-history>
    <measure>
        <mid>992</mid>
        <value>78.9</value>
        <created>2007-12-09</created>
    </measure>
    <measure>
        <mid>999</mid>
        <value>75</value>
        <created>2009-12-09</created>
    </measure>
    <measure>
        <mid>1002</mid>
        <value>72</value>
        <created>2011-12-09</created>
    </measure>
</measure-history>
```

Assignment #2: Part 2

- With that model, expose the following services through a RESTful API as follows: * CRUD operations for person (GET,PUT,DELETE) on /person/{id} and POST on /person
  - GET /person should list all the names in your database
  - GET /person/{id} should give all the personal information plus current measures of person {id}
  - GET /person/{id}/{measure} should return the list of values (the history) of {measure} for person {id}
  - GET /person/{id}/{measure}/{mid} should return the value of {measure} identified by {mid} for person {id}
  - POST /person/{id}/{measure} should save a new value for the {measure} of person {id}
  - GET /measures should return the list of measures your model supports in plain text and separated by commas as follows: weight,height,steps

Assignment #2: Part 2

- **Extra points:**
  - Having a real database in sqlite
  - PUT /person/{id}/{measure}/{mid} should update the value for the {measure} of person {id}
  - GET /person/{id}/{measure}?before={beforeDate}&after={afterDate} should return the history of {measure} for person {id} in the specified range of date
  - GET /person?measure={measure}&max={max}&min={min} retrieves people whose {measure} value is in the [{min},{max}] range (if only one fo the query params is provided, use only that)

Assignment #2: Part 2

- Implement a client that call all these services and print the returned information (you can print them as you wish, in a web page or in the console)
- Notes:
  - Use only the **Standalone server**
  - On the date of the **VIVA** evaluation, we will test your implementation live using the client of one of your fellow students (pairs of *server port* and *client port* will be send out before the VIVA, together with server deployment details)
  - Either XML or JSON support is (only one is required)
  - The client must request both and print the one that works or a message saying "NOT IMPLEMENTED" if it does not work
  - Some of these services are going to be part of your final projet, so try to do them well.
  - While for the GET services you will be required to call a service of one of your fellow colleagues, the POST/PUT services can be tested against your server (to avoid problems of not having standar model)

Assignment Rules

- Before submission make a zip file that includes only
  - All Java source files
  - please, do not include .class or IDE generated project files
- Rename the Zip file to: your full name + assignment_no. for example: cristhian_parra_2.zip
- Submission link: www.dropitto.me/introsde2013
- Password will be given and class and sent to the group
- **Soft Deadline:** 3/december, midnight.
- **Hard Deadline:** 17/december (with the third assignment)
  - On this date, we will test the services matching clients and servers

Assignment Evaluation

- The assignment will be evaluated in terms of:
  - Requirements satisfaction
  - Execution & Deployment
  - Code design/independence/competence
  - Submitted in time ?
  - Report (or documentation)
  - Code originality (if you choose to do it in pairs)
- Extra points are used as "recovery" you didn't finish the requirements or didn't submit in time