Chapter 1

REST Service

1.1 Lab description

The purpose of this weeks lab exercise is to develop a REST web service (and also a client) for management of tasks in the Task manager. As part of the exercise,

1. Develop a web service which exposes operations for the management of the tasks from the taskmanager.xml in a RESTful way. To be more specific, you will use the following HTTP methods explicitly for the operations on the tasks.

   (a) HTTP GET: to get tasks as resources
   (b) HTTP POST: to create a new task
   (c) HTTP PUT: to update a task
   (d) HTTP DELETE: to delete a task

2. Develop a client application to test the functionality of task manager RESTful service.

In order to develop the above RESTful service and client, you can use The Java API for RESTful Web Services (JAX-RS). A very good tutorial on developing a REST service in java using JAX-RS specification can be found at REST with Java (JAX-RS) using Jersey - Tutorial. The tutorial also clearly explains about how to expose data entities as resource URIs and you can take inspiration from the article.

1.2 Solution

Our solution makes use of the Jersey API in order to implement a RESTful web service. The Jersey framework provides a series of tools allowing for handling

of Http-requests by mapping Http-methods to Java methods. This mapping is carried out for our part mainly by use of dedicated Java attributes.
The attributes we are using are:

- @Path: modifies class and method declarations to allow for the address of resources to determine which class or method is targeted.
- @GET, @POST, @PUT and @DELETE: modifies method declarations to signify which Http-methods are allowed to map to the given Java-method.
- @Produces: modifies method declarations and determines how the textual output from the method is interpreted when converted to a http reply message. The Java type MediaType encapsulates valid test formats, such as Html, Xml or Json.
- @Consumes: modifies method declarations and defines, corresponding to @Produces, which format of text is accepted in the method body of the Http-method when mapping to this Java-method.
- @FormParam: modifies arguments in declarations and allows for mapping from named parameters in the Http-method to named String parameters in the Java-method.

With the use of these attributes, a number of classes and methods can b defined to accept input from all relevant Http-method formats. Based on the form of the request the Jersey framework finds the Java-method suitable for handling the request. Ambiguities are reported at compile time.
As part of the Http specification it is only necessary to implement the GET method to qualify for a valid Http server. We have implemented GET, POST, PUT and DELETE as part of the exercise. All four methods are only implemented to produce html output. The logic behind the methods are rather simple: a List of Task objects are extracted from persistence, its content is returned in the GET method, while the remaining methods modify the lists contents and and writes it back to persistence.

Persistence of Tasks is carried out in a manner nearly identical to the previous exercise; a Task-class encapsulating the data content of the provided Xml-schema is made serializable by Xml-binding, and a wrapping class, TaskSerializer, serializes a collection of tasks with a similar binding. The purpose of the construct have been to correspond exactly to the provided schema without modification.

Several difficulties have been encountered during the exercise:

- General problems with implementing and handling the Jersey framework. Were resolved.

- Difficulties getting the Jersey-defined Java servlet to run on the Apache Tomcat server used for test runs. Were not resolved.
  We have had general problems with running code on the server throughout the entire course, and in this case we have not been able to resolve the problem. When attempting to run the code the server responds with "404 resource not found". As a consequence it has not been possible to run the code and test the client, but the core logic of manipulating the task collection is so simple that we are pretty confident in asserting that it is correct.

- Problems reading from persistence when running the program on the server. Were not resolved.
  It seemed these problems where caused by the execution being run from a different system position when run on the server. During development we had no problems reading from the .xml-document included in the eclipse-project when running the program as a stand-alone Java application. This leads us t believe that the relative file path needed to define access to the file were pointing to an invalid position when the default base path where defined from the server. We were not able to isolate the correct position to place the document at in order for the server to gain access to it, other than to use an absolute filepath, which would break the build when ported to another machine.

1.3 Test run

Due to the problems stated in section 1.2 we have not been able to produce a test run.

1.4 Reflection

This exercise exemplifies an implementation of a RESTful web service through Http, and an underlying handling of the central Http methods "GET", "POST", "PUT" and "DELETE".
The purpose of the REST architecture model is to enforce scalability of web based systems, and definition of uniform and generalized interfaces for communication between web components. The service implemented follows the principal rules of RESTful services:

- The server is separated from any client implementation, as communication happens only by Http.

- The server is stateless; it stores no session state and no information about which requests have been made. During the individual request the server only knows about the state of the resources it exposes before and after the request has been executed.

- The rules on limitations of side effects for the different methods have been followed:
    - The GET method should have no side effects; that is it should not change the representation of any resources exposed by the server. This is followed, as GET never calls TaskRestServer.writeObjects(), an therefore does not change the content of persistence.
    - GET, PUT and DELETE should not cause different results in the content of exposed data if they are called with the same content several times, as opposed to being called only once.
      The is followed by GET for the reasons stated above.
      PUT requests provide a task, and adds it to the collection only if it can find an existing task with the same id. It then deletes the existing task, ensuring that only the first in a sequence of identical calls can change the contents of the collection.
      DELETE removes a task from the collection with the id provided. As it is defined as an invariant of the collection that only one task with a given id can exist (both POST and PUT enforces this invariant), only the first call in a sequence will find any task to remove. Following calls will therefore have no effects.
- The data exposed by the server (tasks) is exposed, transported and modified through an Xml-representation; the implementation of tasks as a Java class is irrelevant to the client.
- Difference in URI identifies resources being addressed. The current solution only supports one resource, but the design would allow for additional URIs to be defined.

1.5 Conclusion

We have implemented a service that follows the ruled and guidelines for a RESTful web service.
The service uses Http for communication, and exposes a collection of tasks for retrieval and modification through the methods "GET", "POST", "PUT" and "DELETE". The tasks are exposed as an Xml-representation.
The implementation of the Jersey framework and the handling of the different requests seems to be working correctly.
We have not been able to run the program correctly on the server used for the assignment, an as such have not been able to fully test the solution. Reading from and writing to the persistent storage of tasks have not been resolved satisfactory.