COMP30220 Distributed Systems Practical

Lab 4: RESTful Distribution

Work individually. Submit your code on csmoodle.ucd.ie by the deadline given on moodle. Please submit a single ZIP file (name is <student-num>.zip) containing your source code and any text files required for the solution.

This practical again uses the QuoCo scenario. You should start by redownloading the original source code. Do not attempt to adapt one of your previous solutions. The broad objective of this practical is to adapt the code provided to use RESTful Web Services. As before, I have chosen to do this is a piece-meal way. However, in contrast with the previous approaches, in this lab, we will start with the broker / quotation service interface and then look at the client / broker interface. This alternate approach is adopted largely because the quotation services are the services that create resources (the quotations), while the broker service does not create services – it returns a list of existing quotations to the client.

For this worksheet, I am not going to require the implementation of a registration/discovery service. Such services do exist, for example: Apache Zookeeper and Netflix's Eureka components include discovery services (amongst many other services). These services perform largely the same role as the UDDI servers – they allow attribute based registration of REST service endpoints, and discovery of those endpoints by clients looking for services.

For the more adventurous amongst you, I can recommend exploring the Netflix Eureka service which can be found at the following url: https://github.com/Netflix/eureka/wiki.

Task 1: Redesigning the Quotation Services **Grade: C**

The first task is to re-engineer the quotation services, which are responsible for generating quotation resources. For each service, we will create a resource URL of the form:

http://<host>:<port>/quotations (A)

This resource will accept POST requests that return a 201 response and the location of the created quotation resource.

Quotation resources will be accessible via the following URL:

http://<host>:<port>/quotation/<reference> (B)

The impact of this is that we will need to modify the quotation services to keep any quotations that are generated. To save time, we will only consider a GET operation for the quotation resources. This operation returns a 200 response code together with the quotation entity.

The steps below outline what needs to be done **FOR EACH of the quotation services**. The text below explains how to do this for **the Auld Fella's service**:

a) The first step is to copy the necessary jar files into your project. The files can be found in the example restlet code and are gson.jar and org.restlet.jar.

b) Next, create a RESTlet application that will deploy restlets to handle the two URLs described above. I suggest you follow the same design as the student example in the notes. Lets call this class: `AFApplication`. It should be created in the `service.auldfellas` package.

NOTE: Use the StudentApplication class as a basis for the design of this class. Specifically, notice how the student records were stored in a map (key=student id, value = student record). You will need to modify this to store quotation objects.

c) The next step is to implement the **/quotations** resource. This resource supports only POST operations, and returns a 201 error plus the URL of the quotation resource (template B above) that it creates. The content of the POST operation should be a ClientInfo object. This should be implemented using an anonymous subclass of the Restlet class (as per the **/students** example). When a valid POST request is received, the restlet should invoke the generateQuotation method on the underlying service to generate a quotation object. This object should be stored in the quotations map. To create and return the correct URL for the quotation entity, you should use the following code:

```
response.setLocationRef(request.getHostRef()+"/quotation/"+quotation.reference);
```

d) The fourth step is to implement the **/quotation/<reference>** resource. As per the specification above, this resource supports only the GET operation, which returns a 200 response code together with a json representation of the corresponding quotation entity, if it exists, and 404 error not found response otherwise.

e) Next, create a main method in the AFApplication class that creates a HTTP Server component and deploys an instance of AFApplication. The endpoint port should be 9000 (9001 for Dodgy Drivers, and 9002 for Girl Power).

f) Finally, create a class called TestClient and use the code below to test that the service works. This class will not be part of the final system. It is just used for testing purposes.

```
package service.auldfellas;

import java.io.IOException;

import org.restlet.resource.ClientResource;
import org.restlet.resource.ResourceException;
import org.restlet.util.NamedValue;
import org.restlet.util.Series;

import com.google.gson.Gson;

import client.Main;

public class TestClient {
    @SuppressWarnings("unchecked")
    public static void main(String[] args)
                                throws ResourceException, IOException {
        Gson gson = new Gson();

        ClientResource client = new
                    ClientResource("http://localhost:9000/quotations");

        client.post(gson.toJson(Main.clients[0]));

        String location =
              ((Series<NamedValue<String>>)
              client.
                  getResponseAttributes().
                  get("org.restlet.http.headers")).
                      getFirstValue("Location");

        System.out.println("URL: " + location);

        new ClientResource(location).get().write(System.out);
    }
}
```

The above code starts by sending a POST request to the quotation service. It assumes that this returns a 200 OK response and extracts the location from the header. Finally, it performs a GET operation on the returned location URL and displays the response on the console. The expectation is that the output would be:

```
Starting the internal HTTP client
Resource: http://localhost:9000/quotation/AF001000
Starting the internal HTTP client
{"company":"Auld Fellas Ltd.","reference":"AF001000","price":733.6}
```

Task 2: Re-implementing the Broker **Grade: B**

The second task is to redesign the broker to work with the three quotation services we developed in task 1. Start by hard coding the URLs of the quotation services (as you did for the SOAP lab). The test client of task 1(f) contains most of the code needed to implement the client side of the interaction with the quotation services.

Create a class called Server that contains a main method that invokes the main methods of each of the three quotation services. Run this program (so all three quotation services start).

Modify the Main class, removing the code that creates and registers the quotation services (for the ServiceRegistry), and run the code.

For readability purposes, modify the main method of the Main class to be:

```
public static void main(String[] args) {
	BrokerService brokerService = ServiceRegistry.lookup(BROKER_SERVICE,
                                                             BrokerService.class);

	// Create the broker and run the test data
	for (ClientInfo info : clients) {
		List<Quotation> quotations = brokerService.getQuotations(info);
		displayProfile(info);

		// Retrieve quotations from the broker and display them...
		for(Quotation quotation : quotations) {
			displayQuotation(quotation);
		}

		// Print a couple of lines between each client
		System.out.println("\n");
	}
}
```

NOTE: You can pass a Java List object to the Gson.toJson(..) method, and it will convert that list into a JSON array. Converting it back to the list object requires a little more work than for simple objects. First, you must specify a List Type and then pass that to the Gson.fromJson(…) method:

```
Type type = new TypeToken<List<Quotation>>(){}.getType();
List<Quotation> quotation = gson.fromJson(json, type);
```

Task 3: A RESTful Broker Service **Grade: A**

The final task is to convert the broker service into a RESTful service. This service will be implemented as another application. The broker service will be exposed as a resource accessible via URL:

http://<host>:<port>/broker

This resource responds to a POST request, but (unlike the quotation service) does not create another resource. Instead, it aggregates an existing set of resources (the quotations collected by the broker) and returns them in json.

Specifically, the task here is to wrap the LocalBrokerService in a BrokerApplication that, upon receipt of a valid POST request, executes the getQuotations() method, and then returns the list of quotations as a response. It does this with an associated 200 response code.

According to the HTTP specifications, this type of response is only allowed in cases where the POST operation does not generate a new resource (as in this case). The broker should be deployed on port 9003.

NOTE: Because the BrokerApplication acts as both a client and a server, you need to enable the HTTP protocol for clients as well as servers. This is done by the following line of code, which should be added to the main method of the BrokerApplication (before the start method is invoked):

```
component.getClients().add(Protocol.HTTP);
```

Additional Marks

+ grades (e.g. A+) can be attained through consideration of boundary cases, good exception handling, nice features that enhance the quality of your solution.

- grades (e.g. A-) can be attained though lack of commenting and indentation, bad naming conventions or sloppy code.