COMP30220 Distributed Systems Practical

Lab 4: REST Services

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.

Download the original version of Quoco again (do not attempt to adapt your answer to the previous lab).

The broad objective of this practical is to adapt the code provided to use RESTful Web Services (REST) for interaction between each of the 3 quotation services and the broker and between the broker and the client. In the final version, each of these components should be deployable as a separate docker image and you should provider a docker-compose file that can be used to deploy the images.

To help you complete this challenge, I have broken the problem up into a set of tasks. My advice is to create a separate set of projects for your solution, and for you to copy code from the original project as needed.

As an additional step, please download the Postman application (https://www.getpostman.com/). You will need to create an account, but this is free to do.

Task 1: Setting up the Project Structure   **Grade: E**

As indicated above, we will break the original project into a set of projects: one of each of the REST services, and a core project that contains the code that is common across the overall application. Based on this, you should create a set of folders as follows:

- **core**: contains the common code (abstract base classes & any data classes)
- **auldfellas**: The Auldfella's Quotation Service
- **dodgydrivers**: The Dodgy Drivers Quotation Service
- **girlpower**: The Girl Power Quotation Service
- **broker**: The broker service
- **client**: The client service

The following steps will help you to transfer the existing code to the correct projects

a) For the project, create a "src/main/java" folder and copy the "service.core" package into it.

b) Delete the `BrokerService` & `QuotationService` interfaces. Remove "implements QuotationService" from the `AbstractQuotationService` class.

c) Modify the `Quotation` and `ClientInfo` classes to be Java Beans.

   Remember, a class is a Java Bean if it is a data class, if it has a default constructor (a constructor with no parameters), if its fields are private, and it has set/get methods for each field.

d) Copy the pom.xml file from the "core" project of the RMI Calculator. Change the **groupId** to "quoco.rest" and the **artifactId** to "core".

e) Compile & Install the "core" project

Task 2: Creating and Testing the Auldfellas Quotation Services   **Grade: D**

The second task involves creating a distributed version of the Quotation Services. I will start by explaining how to do it for one of the services – auldfellas – and you will need to do the same thing for the other services.

a) Copy the rest-greeting **pom.xml** file into the auldfellas folder. Modify the groupId to be "quoco.rest" and the artifactId to be "auldfellas". Add the "quoco.rest:core" dependency.

b) Create the "src/main/java" folder structure and copy the "service.auldfellas" package into it.

c) Remove the reference to the QuotationService interface. Annotate the class with @RestController.

d) Copy quotations field declaration and createQuotation(…) method below:

```
private Map<String, Quotation> quotations = new HashMap<>();

@RequestMapping(value="/quotations",method=RequestMethod.POST)
public Quotation createQuotation(@RequestBody ClientInfo info) {
    Quotation quotation = generateQuotation(info);
    quotations.put(quotation.getReference(), quotation);
    return quotation;
}
```

This code implements the URI associated with for a list of quotations. In contrast with other implementations, when we POST to this list resource, we much create a new resource that is accessible via another URI. This second URL takes the form /quotations/{reference} and support for implementing reading of can be modelled through the use of a GET operation as is implemented below:

```
@RequestMapping(value="/quotations/{reference}",method=RequestMethod.GET)
public Quotation getResource(@PathVariable("reference") String reference) {
    Quotation quotation = quotations.get(reference);
    if (quotation == null) throw new NoSuchQuotationException();
    return quotation;
}
```

NOTE: You will need to create a NoSuchQuotationException class as per the example in the notes.

Finally, we need to provide support for retrieving a list of the available quotations. This can be modelled as a GET operation applied to our /quotations resource and is implemented as follows:

```
@RequestMapping(value="/quotations",method=RequestMethod.GET)
public ArrayList<Quotation> listQuotations() {
    ArrayList<Quotation> list = new ArrayList<>();
    for(Quotation quotation: quotations.values()) list.add(quotation);
    return list;
}
```

e) Modify the generateQuotation() method implementation to adhere to the Java Bean implementation (fields are no longer public) – you will get compilation errors if you try to compile the codebase without doing this.

f) Create an Application class in the auldfellas package that looks like the code below:

```
package service.auldfellas;

@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

g) Fix any missing imports and try to compile & run the "auldfellas" project – remember to run the project use "spring-boot:run".

To test your service, run Postman and create a POST request with url: "http://localhost:8080/quotation". Copy the following JSON into the body of the request and click the "Send" button. You should see a response like the one below (notice that Spring Boot automatically converts the Quotation objects into JSON):
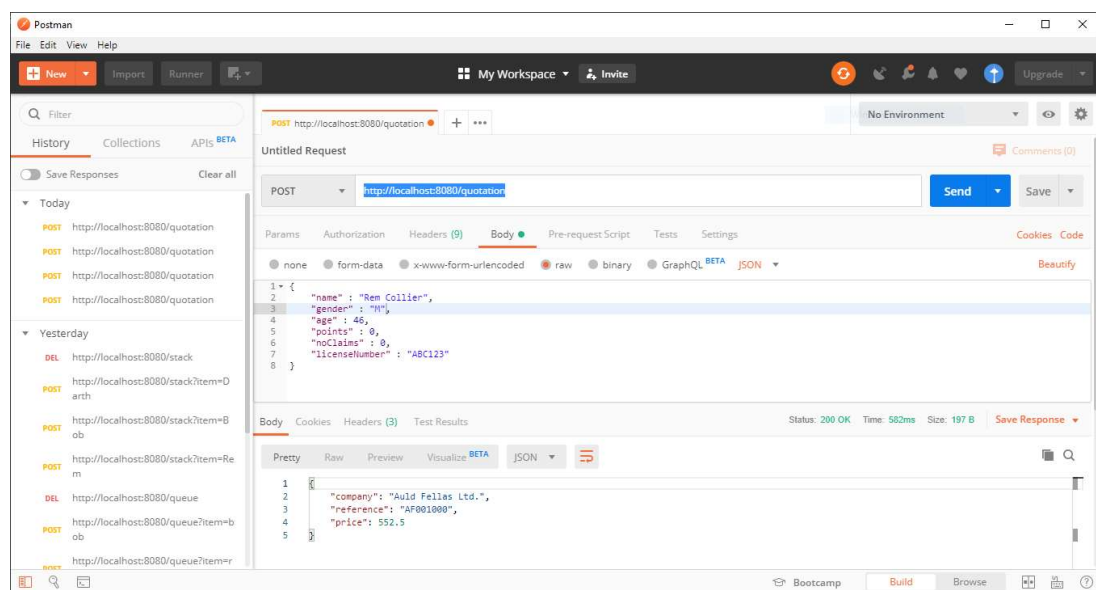
h) The final step of this task is to write some code to test this service. To do this, we will do some work on the "client" project. This will be a temporary version of the client code that we will use to test the quotation service we have just created.

In creating this client, we will use the RestTemplate class. This is a Spring API that simplified the task of creating a client that can interact with a RESTful Service. The code below is a simple example of how to use this API.

Copy the pom.xml file you created in step (a) of this task into the client folder. Modify the **artifactId** to be "client" and the main class to be "client.Client". Generate the "src/main/java" folder structure and create a Client class. Copy the displayProfile(), displayQuotation(), and the clients data field from the old client.Main class into this new class. Copy the code below:
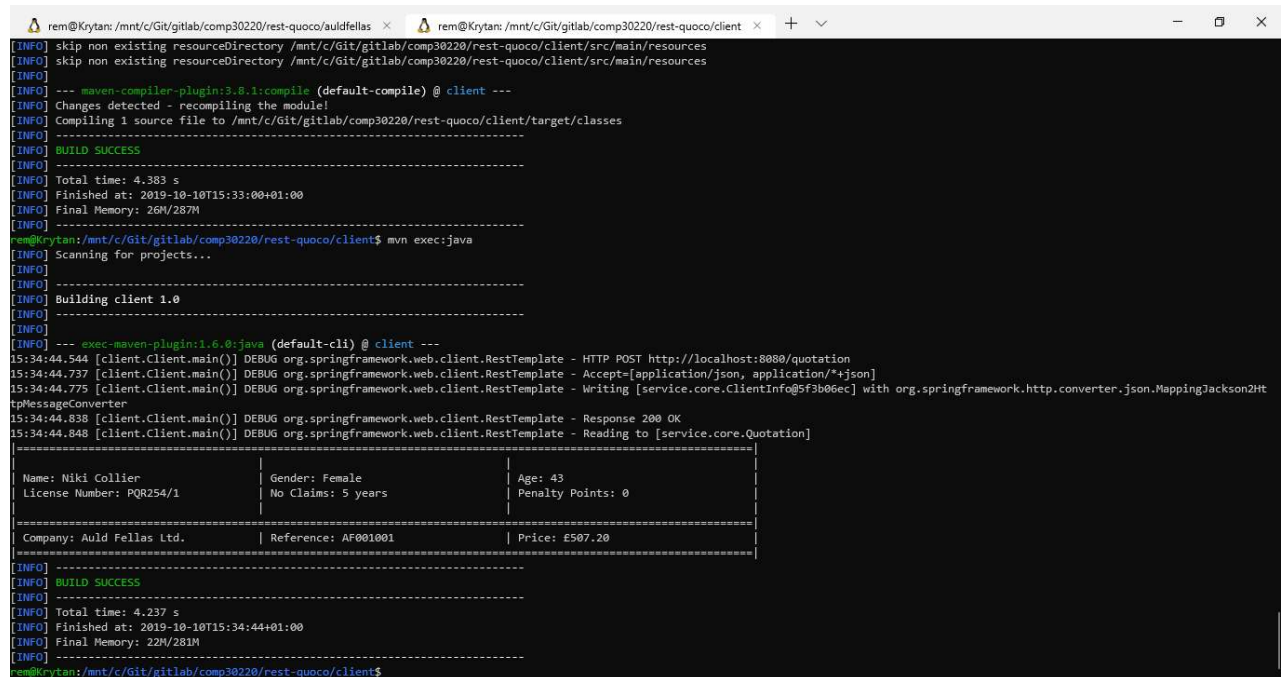
```
public static void main(String[] args) {
    RestTemplate restTemplate = new RestTemplate();
    HttpEntity<ClientInfo> request = new HttpEntity<>(clients[0]);
    Quotation quotation =
        restTemplate.postForObject("http://localhost:8080/quotations",
            request, Quotation.class);
    displayProfile(clients[0]);
    displayQuotation(quotation);
}
```

Also, add the following imports to use the `RestTemplate` class.

```
import org.springframework.web.client.RestTemplate;
import org.springframework.http.HttpEntity;
```

Note that I have hardcoded the URL of the service here. This is sufficient for testing purposes.

Compile and run this code – you should see a single client with a single quote from "Auld Fellas Ltd."

Task 3: Implementing the Other services

Repeat the steps of task 2 to create and test the "girlpower" and "dodgydrivers" projects.

Note: a challenge here is to run multiple versions of spring boot concurrently (by default, it runs on port 8080). To make this change, you need to create an application.properties file (it is a text file that is created in the src/main/resources folder). To configure Spring Boot to run on port 8081, the application.properties file should contain the following line:

```
server.port=8081
```

Task 4: Implementing the Broker and Client

Now we have working quotation services, the next task is to create and test the broker.

a) Create a service.broker.Application class that implements the same main method as Task 2(f).

b) Convert the LocalBrokerService into a REST service – use the URI (/applications) to model the list of applications. Like the Quotation Services, this URI should support POST operations. The expected behaviour associated with this operation is to call the getQuotations() method which connects to the Quotation Services using the RestTemplate developed in Task 2(h). The List of Quotations returned by this method should be associated with the ClientInfo and a unique application-number and stored in a Map (where the key is the application number).

NOTE: the concept of an application is new to the system. An application is a combination of some ClientInfo, a List of Quotations, and a unique identifier (the application number). The identifier can be an integer value

that is unique to each application (rank in an ArrayList is not a valid application number because rank changes). You should create a new class `service.core.ClientAppliation` *to model this (it will need to be added to the core project and will need to be a Java Bean). Instances of this class should be stored in a Map in the same way that Quotations were stored in Task 2(d). As with the web services implementation, you should prefer concrete classes (e.g. ArrayList) over interfaces (e.g. List). Modify the code as necessary.*

NOTE: The implementation of the getQuotations() method should be modified in a similar way to what we did in the web services lab (an array of URIs). Ideally, this list should be passed as a parameter to the method that creates the Application (this is more tricky to do here because there is no direct link between the main() method and the controllers).

Run this application and test it using Postman – the output should look like the screenshot below:
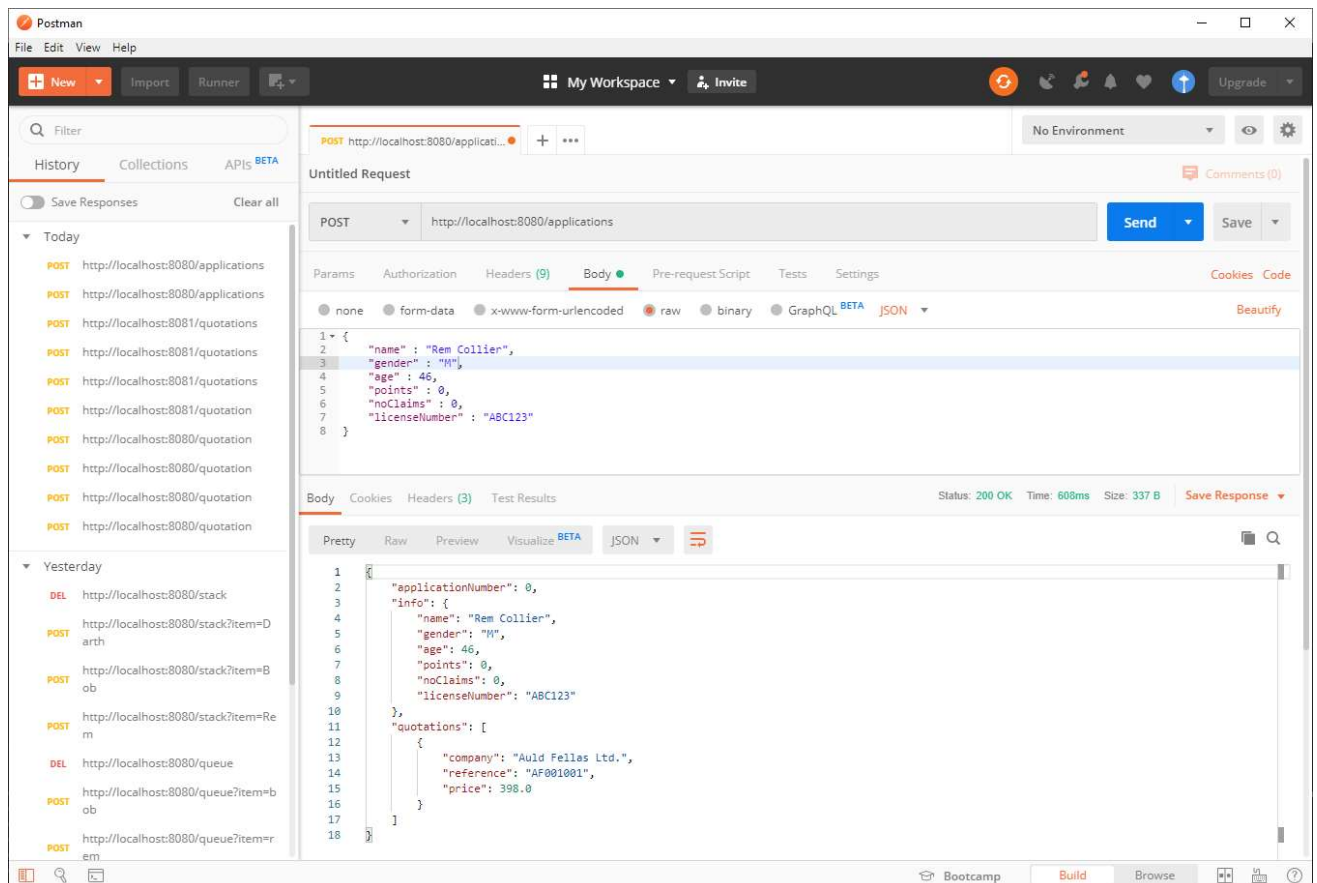
c) Create a method to handle GET requests submitted to the URI (/applications/{application-number}) which is used to refer to an individual application.

d) Create a method to handle GET requests submitted to the URI (/applications) which returns a list of ClientApplication objects.

e) Modify the test client to lookup the broker and modify the main() method to loop through and print out all the quotations returned by the broker service.

f) Compile and run both projects 😊

Task 5: Containerisation   **Grade: A**

The final task is to convert the output of task 4 into a set of docker images and an associated docker compose file. You should map the broker port so that it can be accessed by external programs and then run the client using maven.

There is no need to make the client into a docker image.

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.