COMP30220 Distributed Systems Practical

Lab 4: Message Oriented Middleware

Work individually. Submit your code on csmoodle.ucd.ie by the deadline given on Brightspace. 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 JMS 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. The only exception is the client which should be executable as a maven project.

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.

NOTE: You should have a look at the example projects I have released on brightspace

Task 1: Setting up the Project Structure — Grade: D

As indicated above, we will break the original project into a set of projects: one of each of the JMS clients you will develop, and a core project that contains the code that is common across the overall application. This should all be implemented as a multi-module maven project in a folder called "quoco-jms". This project should include the following modules:

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

b) Delete all references to the `ServiceRegistry` implementation.

c) Modify the `Quotation` and `ClientInfo` classes to implement the `java.io.Serializable` interface.

d) Create a basic `pom.xml` file for each of the modules and the main project. The groupId for each pom should be "lab4".

e) Compile & Install the "core" project.

f) Run the ActiveMQ Docker Image (you can keep this running until task 5)

```
$ docker run --name='activemq' -it -p 8161:8161 -p 61616:61616 rmohr/activemq:latest
```

g) Log into the console (http://localhost:8161/admin) username/password = admin/admin

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

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) Modifly the auldfellas **pom.xml** file to support JMS development. This includes adding the activemq dependency; the exec-maven-plugin and the maven-assembly-plugin.

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

c) Before we get into the implementation of the code to link the quotation service with the messaging infrastructure, it is important to take a moment to reflect on the impact of JMS on the design of the system. Interaction with the quotation services will take the form of asynchronous message passing. This means that requests are no longer automatically linked to responses – the client no longer blocks, waiting for the response from the server. Instead, we must manually link requests with responses. To do this, we need to create a set of custom "message classes" that model the interactions. Here, we introduce 2 message types: a `QuotationRequestMessage` and a `QuotationResponseMessage`. The first of these will be a data class that contains a reference to the `ClientInfo` object together with a unique (application) number that will be returned in the second response message (along with the resulting `Quotation` object). A sketch of these classes is given below. All message classes should be included in the "core" project (which should be recompiled) as they will be used across multiple projects.

```
package service.message;

public class QuotationRequestMessage implements Serializable {
    public long id;
    public ClientInfo info;

    public QuotationRequestMessage(long id, ClientInfo info) {
        this.id = id;
        this.info = info;
    }
}

package service.message;

public class QuotationResponseMessage implements Serializable {
    public long id;
    public Quotation quotation;

    public QuotationResponseMessage(long id, Quotation quotation) {
        this.id = id;
        this.quotation = quotation;
    }
}
```

d) Now that we have our messages sorted, the next step is to link the `AFQService` to the messaging infrastructure. How we do this will be slightly different to the lecture notes because we need to cater for the future containerisation of the code. All of the code below will be written within a single `main()` method that should be declared in a class called `service.Receiver`. This class should also include a static field that contains a reference to an instance of the `AFQService` (it is referred to as "service" in the code below). As has become the norm, you should write some code that allows a hostname to be passed as a parameter to the `main()` method -the default value should be "localhost" (if you are using Docker Toolkit – perhaps you could default this to the IP address of the docker virtual machine). This will be used to specify the host name of the machine that is running the ActiveMQ broker instance that we will connect to. Because of this, we need to explicitly specify the URL of this broker (in the class code, we used a DEFAULT URL that was linked to the localhost). This URL takes the form: `failover://tcp://<host>:61616` (NOTE: port 61616 was one of the two ports we mapped when we ran the ActiveMQ instance as a docker container). The common code to link to the broker is given below:

```
ConnectionFactory factory =
          new ActiveMQConnectionFactory("failover://tcp://"+host+":61616");
Connection connection = factory.createConnection();
connection.setClientID("auldfellas");
Session session = connection.createSession(false, Session.CLIENT_ACKNOWLEDGE);
```

The next issue is what we are linking the AFQService to. AS per the master/slave pattern described in the lecture notes, our quotation services will ultimately play the roles of slaves to the broker (which will be the master). As such, the quotation services will listen for incoming applications on a shared topic (a pub/sub channel) and will respond directly to the broker on an associated queue (P2P channel). We will call these two channels APPLICATIONS and QUOTATIONS respectively. The quotation service will be a consumer of messages from the former and a producer of messages for the latter. The code for setting up these channels is given next:

```
Queue queue = session.createQueue("QUOTATIONS");
Topic topic = session.createTopic("APPLICATIONS");
MessageConsumer consumer = session.createConsumer(topic);
MessageProducer producer = session.createProducer(queue);
```

The final step is to create the main algorithm that handles each incoming `QuotationRequestMessage` and returns an associated `QuotationResponseMessage`:

```
connection.start();

while (true) {
       // Get the next message from the APPLICATION topic
       Message message = consumer.receive();

       // Check it is the right type of message
       if (message instanceof ObjectMessage) {
              // It’s an Object Message
              Object content = ((ObjectMessage) message).getObject();
              if (content instanceof QuotationRequestMessage) {
                     // It’s a Quotation Request Message
                     QuotationRequestMessage request = (QuotationRequestMessage) content;

                     // Generate a quotation and send a quotation response message…
                     Quotation quotation = service.generateQuotation(request.info);
                     Message response = session.createObjectMessage(
                                   new QuotationResponseMessage(request.id, quotation));
                     producer.send(response);
              }
       } else {
              System.out.println("Unknown message type: " +
                                          message.getClass().getCanonicalName());
       }
}
```

NOTE: As is mentioned above - all of this code should be written in a main() method that is part of the `service.Receiver` class. Some exception handling code will be required and there will also need to be a static field with name “service” that references an instance of the AFQService.

NOTE: Remember to update the pom.xml to point to this class before you recompile and run the project.

e) The final step is to implement some code to test that the service runs correctly. To do this, we will create a temporary version of the client project that matches the behaviour specified above. To do this, create a “client” project – copy and adapt the “auldfellas” `pom.xml` file. Create the `src/main/java` folder and copy in the `client.Main` code. Remove all references to the `ServiceRegistry` (and all direct references to quotation/broker services).

We will modify the `main()` method to implement the JMS connectivity. It looks a lot like the code we wrote for the quotation service but is inverted (we act as a producer for the APPLICATION topic and as a consumer for the QUOTATION queue)…

```
ConnectionFactory factory =
          new ActiveMQConnectionFactory("failover://tcp://"+host+":61616");
Connection connection = factory.createConnection();
connection.setClientID("client");
Session session = connection.createSession(false, Session.CLIENT_ACKNOWLEDGE);
```

Note: Again, we must include the code to pass in the host name for the ActiveMQ broker.

```
Queue queue = session.createQueue("QUOTATIONS");
Topic topic = session.createTopic("APPLICATIONS");
MessageProducer producer = session.createProducer(topic);
MessageConsumer consumer = session.createConsumer(queue);
```

Now comes the trick(ier) part. We need to implement the following behaviour: create a `QuotationRequestMessage` object with a unique number assigned; Associate the `ClientInfo` with that number (we will use a `Map` to do this); publish the request; await a QuotationResponseMessage and print out the `ClientInfo` + the quotation when we get one. To create the unique number, use a static field called `SEED_ID` that is incremented every time it is used (i.e. `SEED_ID++`). I would start by creating a single test `QuotationRequestMessage`:

```
QuotationRequestMessage quotationRequest =
              new QuotationRequestMessage(SEED_ID++, clients[0]);
Message request = session.createObjectMessage(quotationRequest);
cache.put(quotationRequest.id, quotationRequest.info);
producer.send(request);
```

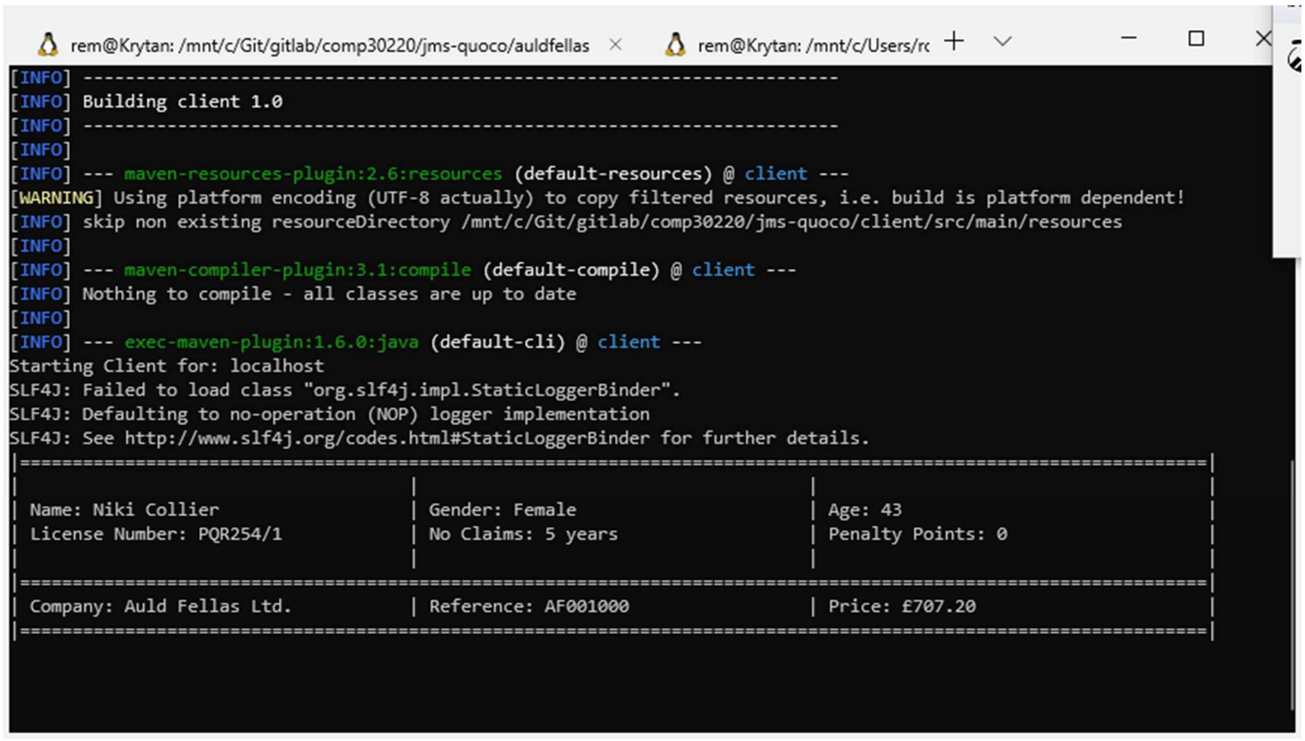
Note: cache is a static field that is an instance of `Map<Long, ClientInfo>`.

Now, we need to block for the response. We do this using `consumer.receive()`. Once we receive a message, we need to check that it is the correct form, match it to the ClientInfo in the cache and display the results…

```
Message message = consumer.receive();
if (message instanceof ObjectMessage) {
       Object content = ((ObjectMessage) message).getObject();
       if (content instanceof QuotationResponseMessage) {
              QuotationResponseMessage response  = (QuotationResponseMessage) content;
              ClientInfo info = cache.get(response.id);
              displayProfile(info);
              displayQuotation(response.quotation);
              System.out.println("\n");
       }
       message.acknowledge();
} else {
       System.out.println("Unknown message type: " +
                                   message.getClass().getCanonicalName());
}
```

Note: The above code is a bit of a fudge, in that we really need to decouple sending quotation requests from receiving quotation responses. To do this, we will probably need to introduce multi-threading at some point (luckily, the above code is good enough to test our basic service implementation)…

f) Fix any errors in both projects. Remember to run the activemq docker image. Run auldfellas and then run the client. You should get something like the output below:


rem@Krytan: /mnt/c/Git/gitlab/comp30220/jms-quoco/auldfellas
rem@Krytan: /mnt/c/Users/rc

```
[INFO] ------------------------------------------------------------------------
[INFO] Building client 1.0
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ client ---
[WARNING] Using platform encoding (UTF-8 actually) to copy filtered resources, i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory /mnt/c/Git/gitlab/comp30220/jms-quoco/client/src/main/resources
[INFO]
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ client ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- exec-maven-plugin:1.6.0:java (default-cli) @ client ---
Starting Client for: localhost
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
|=================================================================================================|
|                                 |                                |                               |
| Name: Niki Collier              | Gender: Female                 | Age: 43                       |
| License Number: PQR254/1        | No Claims: 5 years             | Penalty Points: 0             |
|                                 |                                |                               |
|=================================================================================================|
| Company: Auld Fellas Ltd.       | Reference: AF001000            | Price: £707.20                |
|=================================================================================================|
```

Task 3: Implementing the Other services **Grade: B**

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

NOTE: When you copy the code for these services – remember that the clientId used for each JMS connection must be unique (so "auldfellas" will have to be changed as appropriate).

Task 4: Implementing the Broker and Client **Grade: A**

Now we have working quotation services, the next task is to create and test the broker. The broker code in particular is quite complex due to the asynchronous message passing model used. As a result, I don't require that you reuse any code for it – instead create a new class called `service.Broker` that implements the same behaviour as `LocalBrokerService`.

a) Start with the Broker – large parts of this are the same as the test client developed in 2(e). Specifically, this program will handle incoming `QuotationRequestMessages` from the final client and will return a (new) `ClientApplicationMessage` that contains the original `ClientInfo` object together with a list of `Quotation` objects related to the original request (remember the `ClientApplication` class from Lab 4). Implementing this introduces some problems: (1) we no longer know how many quotation services we are interacting with; (2) the order in which we send quotation requests and receive quotation responses is non-deterministic (so if we processed multiple requests in parallel, the response messages could be interleaved – this means that we could receive a response to request 1, followed by a response to request 3, followed by a response to request 1, followed by a response to request 2, …). We already have the code to handle the non-determinism, but we need to decouple the request and response code we used in tasks 2 & 3. Specifically, you will need to run the consumer code in a separate thread to the producer code. Further, you will need 2 consumers – one to handle incoming `QuotationRequestMessage` object from the client (you will need to create a REQUESTS queue to handling this channel), and one to handle the responses from the quotation services. In fact, the code that consumes the messages from the REQUESTS channel should create a new thread to process each request. In response to problem (1) - this thread should publish the request on the APPLICATIONS topic and then sleep for a fixed time period (a timeout). After the timeout completes, the thread should grab any quotations received, package them in a `ClientApplicationMessage` object and send them back to the client (which should listen on another (new) RESPONSES channel.

HINT: You should use the `ClientApplicationMessage` class to formulate the response to the client (i.e. cache instances of this class matched against the broker generated request id rather than `ClientInfo`).

b) The client should be a variant of the temporary client implementation. Again, I give the solution in outline rather than in detail. Here, the client submits each `ClientInfo` object to the REQUESTS queue. A separate thread should monitor the REPONSES queue for responses. A Map should be used to cache `ClientInfo` objects and to match them to a unique request id. Incoming `ClientApplicationMessage` responses should be matched to requests and the associated client/quotation details output.

NOTE: responses are non-deterministic, so the ordering of the output is not guaranteed.

c) Compile and run all projects 😊

Note: Creation of a docker-compose.yml file that launches the ActiveMQ message broker, the 3 quotation services, and the broker service is left as a challenge and those that complete it will receive an A+.

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.