

COMP30220 Distributed Systems Practical

Lab 2: WS-based 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.

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 Web Services (WS) 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 provide 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 project for your solution, and for you to copy code from the original project as needed.

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 distributed objects, 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 (common interfaces, 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

Note: We did not use a core package in the ws-quote example project because there was no shared codebase. However, in this project, there will be some shared code: the AbstractQuotationService, and the ClientInfo and Quotation data classes.

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) Check that the Quotation and ClientInfo classes have default constructors:

```
public ClientInfo() {}
```

There does not need to be any code in the brackets – they just need to exist.

- d) Copy the pom.xml file from the "core" project of the RMI Calculator. Change the **groupId** to "quoco.ws" and the **artifactId** to "core".
- e) Compile & Install the "core" project

Task 2: Creating and Testing the Distributed 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 ws-quote server **pom.xml** file into the auldfellas folder. Modify the groupId to be “quoco.ws” and the artifactId to be “auldfellas”. Set the main class to be the java class: “service.core.Quoter”. Don’t forget to add the quoco.ws:core dependency.
- b) Create the “src/main/java” folder structure and copy the “service.auldfellas” package into it. Rename the class to “service.core.Quoter” (note – change the package AND the classname)

IMPORTANT: This change is essential because we need to standardise the namespace and service name for all our web services. Remember that, when we infer the web service interface (on the client-side), the interface we create depends on the namespace and service name. If they are different for each web service, then we will need a different interface to access each one... (new service = new interface = code must be recompiled and re-deployed = BAD).

There are some ways to do this through customisation of the annotations. This can have unexpected complications as the customisation must cover multiple areas. The approach I am recommending here is the simplest as it does not require any customisation.

- c) Annotate the class with @WebService and @SOAPBinding (use RPC and Literal – see ws-quote example).
- d) Annotate the generateQuotation() method with @WebMethod.
- e) Add the main() method below:

```
public static void main(String[] args) {  
    Endpoint.publish("http://0.0.0.0:9001/quote", new Quoter());  
}
```

Fix any missing imports and try to compile & run the “auldfellas” project – remember once it runs, you can check that it is working by loading the WSDL document. For this project it is:

<http://localhost:9001/quote?wsdl>

- f) The next step 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.

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”. Generate the “src/main/java” folder structure and create the class: service.core.QuoterService with the following implementation:

```
package service.core;  
  
@WebService  
public interface QuoterService {  
    @WebMethod Quotation generateQuotation(ClientInfo info);  
}
```

Now create a class called Client (default package). Copy the displayProfile(...), displayQuotation(...) and clients data from the client.Main class in the original project. Use the following main() method to test the service:

```
public static void main(String[] args) throws Exception {
    String host = "localhost";
    int port = 9001;

    // More Advanced flag-based configuration
    // [ copy this from the ws-quote example client ]

    URL wsdlUrl = new
        URL("http://" + host + ":" + port + "/quote?wsdl");

    QName serviceName = new QName("http://core.service/", "QuoterService");

    Service service = Service.create(wsdlUrl, serviceName);

    QName portName = new QName("http://core.service/", "QuoterPort");
    QuoterService quotationService =
        service.getPort(portName, QuoterService.class);

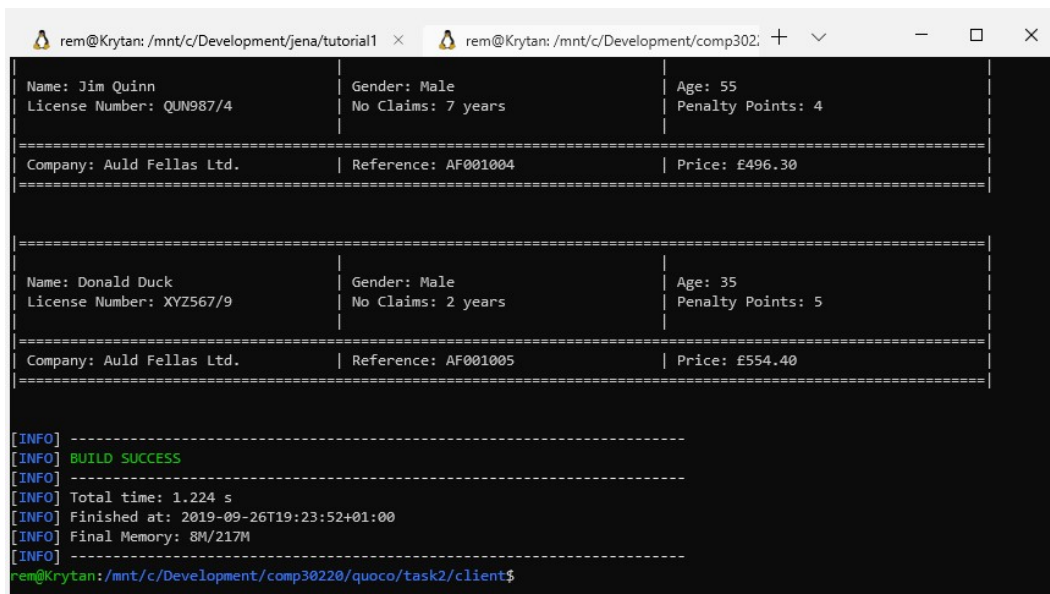
    for (ClientInfo info : clients) {
        displayProfile(info);

        Quotation quotation = quotationService.generateQuotation(info);
        displayQuotation(quotation);

        System.out.println("\n");
    }
}
```

Note that the default values for host and port will work here, but you should copy the code from the ws-quote example that allows the host and port to be set through the program arguments.

Compile and run this code – you should see a set of clients with a single quote from “Auld Fellas Ltd.” (like below)



The terminal window shows the output of the Java application. It displays two client profiles and a quotation from 'Auld Fellas Ltd.' for each. The first client is Jim Quinn, and the second is Donald Duck. The quotation for Jim Quinn is £496.30, and for Donald Duck it is £554.40. Below the quotations, there is a build success message and some performance statistics.

```
rem@Krytan: /mnt/c/Development/jena/tutorial1 x rem@Krytan: /mnt/c/Development/comp302: + v - □ ×

Name: Jim Quinn      | Gender: Male      | Age: 55
License Number: QUN987/4 | No Claims: 7 years | Penalty Points: 4
=====
Company: Auld Fellas Ltd. | Reference: AF001004 | Price: £496.30
=====

Name: Donald Duck   | Gender: Male      | Age: 35
License Number: XYZ567/9 | No Claims: 2 years | Penalty Points: 5
=====
Company: Auld Fellas Ltd. | Reference: AF001005 | Price: £554.40
=====

[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 1.224 s
[INFO] Finished at: 2019-09-26T19:23:52+01:00
[INFO] Final Memory: 8M/217M
[INFO] -----
rem@Krytan: /mnt/c/Development/comp30220/quoco/task2/client$
```

- g) Repeat the above to create and test the “girlpower” and “dodgydrivers” projects.

Note: the non-docker version will require you to use a different port for each service (e.g. 9001 for AF, 9002 for GP and 9003 for DD).

Task 3: Implementing the Broker and Client

Grade: C

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

- a) Convert the LocalBrokerService as a web service (I recommend using the classname: `service.core.Broker`) and modify the implementation to access the web services. If you are smart, you can reduce the changes to an array of URL strings that can be set through the program arguments (e.g. `java -cp /broker-1.0.jar http://localhost:9001/quote,http://localhost:9002/quote, ...`) You can use the `split(",")` method to break this list into an array of strings that are the constituent URLs. I also recommend publishing the broker on the url: `"http://0.0.0.0:9000/broker"`)

NOTES:

- You may need to modify the signature of the `getQuotations()` method as Jax-WS does not like interfaces (so use a concrete class like `LinkedList` instead of `List`).
 - You will need to change the SOAP Style type to `DOCUMENT` (instead of `RPC`) as Jax-WS fails to interpret the contents of the `LinkedList` correctly (you get back a list of quotations with null/default values – try it to see).
- b) 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.
- c) Compile and run both projects 😊

Task 4: Multicast DNS-based Service Discovery

Grade: B

The penultimate task is to replace the manual configuration of web services with a jmDNS based service discovery mechanism. This should be implemented only for the links between the broker and the quotation services (the client should still be configured manually). Receiving a service advert should result in the list of available quotation services being updated (try to make sure replicas are not added). You can use service types – this was `"_http_.tcp.local."` in the lecture example – to identify what type of service is being advertised (e.g. `"_quote_.tcp.local."` for quotation services).

NOTE: You may have issues with this task depending on the configuration of the underlying network (at home, my wifi did not allow jmDNS to work). However, I was able to use jmDNS once I containerised the app (so, you may need to combine tasks 4 & 5)

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. As a final step, create a client 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 through lack of commenting and indentation, bad naming conventions or sloppy code.