COMP30220 Distributed Systems Practical

Lab 3: Web Services-based Distribution

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

You should use the basic version of LifeCo as a starting point for this lab. As a reminder, you can download the LifeCo source code from: https://gitlab.com/ucd-cs-rem/distributed-systems/lifeco-basic.

The broad objective of this practical is to adapt the LifeCo system to use Web Services 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. Discovery of services should be implemented using jmDNS a Java Multicast DNS implementation.

As with the previous lab, 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.

Each task should be completed in turn and **your submission for each task must be stored in a separate folder** called "taskX" where X is the task number. For example, your answer for Task 1 should be stored in a folder called "task1", while your answer for Task 2 should be stored in a folder called "task2". When moving onto the next task, simply duplicate and rename the previous task folder to represent the next task (i.e. after completing Task 1, copy the folder "task1" and rename the copy "task2". Modify the code in "task2" as required to complete the task.

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

As in the previous project the first task is to break the original codebase up into a set of projects: one of each of the web services we are going to create; a project for the client and a shared core project that contains the code that is common across the web services. In this lab, we will combine this with the second task, which involved setting up the core project.

a) You should create a folder "quoco-ws" and inside that create a folder called "task1" which will be the root of the project file structure. As before, this project should contain the following sub-folders:
   - **core**: contains the common code (distributed object interfaces, abstract base classes & any data classes)
   - **auldfellas**: The Auldfella's Quotation Service
   - **dodgygeezers**: The Dodgy Geezers Quotation Service
   - **girlsallowed**: The Girls Allowed Quotation Service
   - **broker**: The broker service
   - **client**: The client service

   You should create a multi-module pom.xml file in the root folder of the project that should reference each subfolder. A standard "jar" packaging pom.xml file should be added to each subfolder. Use the "calculator" project from class as an example, noting that the groupId in the screenshot below. The artifactId for each sub-project should be the same as the name of the folder containing the subproject (e.g. the "core" projects artifactId should be "core" and the "girlsallowed" project artifactId should be "girlsallowed".) The artifactId for the main project should be based on the task (e.g. for this task it should be "task1" while for the next task it should be "task2").

   HINT: You can probably copy this from your Lab 2 submission - it is the same with the exception that the groupId should be "quoco-ws"

b) Next, copy the "service.core" package from the basic project into "src/main/java" folder of the core folder. Remember that you should replicate the package structure. Peform the following modifications:

- Delete the `Constants` class.
- Modify the `QuotationService` and `BrokerService` interfaces by annotating the class with `@WebService` and the methods with `@WebMethod`. You should also delete "extends Service" for each interface.

  NOTE: We do this so that we can use the interface to implement the client side of the web service. The existence of a shared interface is not required in web services, but it is useful because it will allow us to do some unit testing in the service implementations. The downside is that it will make the configuration of each webservice a little more complicated because you will need to customise the WSDL description to match these interfaces. We will talk more about this in Task 2.
- Check that the `Quotation` and `ClientInfo` classes have default constructors or the form:

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

  There does not need to be any code in the brackets – they just need to exist. These constructors are required because the library we are going to use will automatically transform a java object into XML and vice versa. When converting XML to Java, the converter first creates an object and then assigns values to each of its fields. A missing default constructor will cause the object creation step to fail.
- Compile and install the "core" project

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

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

Unlike the last lab, we are going to change the name and package of the class that implements the Auldfellas quotation service. The reason for this is that the JAX-WS library uses the class name and package structure when it generates the WS endpoint. So, instead of using a common interface (as we did in the RMI lab), we are going to use a common package structure and classname across each of the quotation services.

NOTE: 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.

a) First, remember to duplicate the "task1" folder creating a new folder called "task2". All the changes discussed here should be made to "task2".

b) Start by copying the "service.auldfellas" package into the "src/main/java" folder of the auldfellas project.

   Make sure you update the package statement and the class name as appropriate.

   At this point, the project should compile if you type in:

   ```
   $ mvn compile -pl auldfellas
   ```

c) Annotate the class with `@WebService` and `@SOAPBinding` annotations (use the RPC and Literal configuration) and annotate the generateQuotation() method with `@WebMethod`.

d) Finally, create a Main class in the default package of the auldfellas project with the following code:

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

Make sure that the exec:java "mainClass"property refers to this class and then compile & run the "auldfellas" project by typing:

```
$ mvn compile exec:java -pl auldfellas
```

Remember once it runs, you can check that it is working by loading the WSDL document. For this project it is:

http://localhost:9001.quoation?wsdl

NOTE: In the publish() method we use 0.0.0.0 instead of localhost because this ensures that the socket is bound to all IP addresses available on your computer.

NOTE: You should spend some time looking at this WSDL. Note the introduction of the types element, and the associated schema that is imported through it: http://localhost:9001/quotation?xsd=1. If you look at this schema in a web browser, you will see how the `ClientInfo` and `Quotation` classes have been translated to XML.

NOTE: You should also try the various combinations of the style and use parameters for the SOAPBinding annotation. Again, this will help you to understand the meaning of each configuration.

e) While the code runs, if you look carefully at the WSDL document you might be able to identify a potential problem: our goal is to provide a uniform interface for accessing all of the quotation services. Specifically, we need three parts of the interface to be consist across each service: targetNamespace, the service name and the portType name. The first two of these can be found in the definitions tag of the WSDL document and the third is defined by the name attribute in the portType tag. What we really want here is to match the WSDL document to the `QuotationService` interface we modified in task 1.

Luckily JAX-WS provides a way for us to customise these when creating the WSDL document. We simply modify the `@WebService` annotation as follows:

```
@WebService(name="QuotationService",
      targetNamespace="http://core.service/",
      serviceName="QuotationService")
```

Make this change and have a look at how the WSDL document changes.

f) Now we are ready to test the service. To do this we will use Junit as we did in lab 1. Lets' start by adding the junit dependency.

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.12</version>
</dependency>
```

Now we can create the following unit test:

```
public class QuotationServiceUnitTest {
  @BeforeClass
  public static void setup() {
    Endpoint.publish("http://0.0.0.0:9001/quotation", new AFQService());
  }

  @Test
  public void connectionTest() throws Exception {
    URL wsdlUrl = new URL("http://localhost:9001/quotation?wsdl");
    QName serviceName =
        new QName("http://core.service/", "QuotationService");
    Service service = Service.create(wsdlUrl, serviceName);
    QName portName =
        new QName("http://core.service/", "QuotationServicePort");
    QuotationService quotationService =
        service.getPort(portName, QuotationService.class);
    Quotation quotation = quotationService
        .generateQuotation(new ClientInfo(
            "Niki Collier", ClientInfo.FEMALE, 49,
            1.5494, 80, false, false));
    assertNotNull(quotation);
  }
}
```

Notice that the `@BeforeClass` method simply replicates the main method we wrote in (d). The test method is basically the standard client code for connecting to a web service. Here, we use the information we provided when customising the WSDL document to specify the correct qualified service name and qualified port name. You should now be able to test that your codebase runs by simply typing:

```
$ mvn test -pl auldfellas
```

You could add some additional assertions to confirm that the quotation you have received is valid (e.g. the company or reference number).

g) Replicate the above for GirlsAllowed and DodgyGeezers. I suggest using port 9002 for DodgyGeezers and 9003 for GirlsAllowed.

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) Create a folder "task3".

b) Convert the LocalBrokerService into a web service using the same technique as we did in task 2. The key challenge is to replace the registry.lookup() method call with something that will get the URLs of the web

services we want to access. A first step might be to create a list of strings that contains the URLs of the quotation services. You might start with this as a way of testing that it works and then once you are happy that your code works, you could attempt a better solution, which is to pass the list of URLs via the command line (this will make containerising the system easier). We will use a fancier method later in task 5.

NOTES:

- I recommend publishing the broker on port 9000 under context: /broker
- You may need to modify the signature of the getQuotations() method as Jax-WS does not like interfaces for input/output (so use a concrete class like LinkedList instead of List).
- You will also need to change the SOAP Style type to DOCUMENT (instead or 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).

Finally, create a unit test to confirm that you can connect up to the broker and that it returns the right kind of information (for example, it should return 0 quotations because none of the quotation services would be running).

c) Using the test code you have created in part (b), it should be relatively easy to modify the client codebase to connect to the broker and print out the quotations.

Task 4: Containerisation **Grade: B**

This task involves containerisation of the LifeCo-WS system. In contrast with lab 1, the goal here is to dockerise all the services, but not the client. The broker service should be the only service that is publicly accessible (i.e. its port is mapped to the host machine) the broker services should not be port mapped.

Task 5: Multicast DNS-based Service Discovery **Grade: A**

The final 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 testing jmDNS depending on the configuration of the underlying network (at home, my wifi did not allow jmDNS to work). However, jmDNS does work once the services are containerised.