

COMP30220 Distributed Systems Practical

Lab 2: RMI-based Distribution

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.

Please read AND run the QuoCo scenario before attempting this practical. You can download the based QuoCo source code from: <https://gitlab.com/comp30220/quoco>.

The broad objective of this practical is to adapt the code provided to use RMI 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

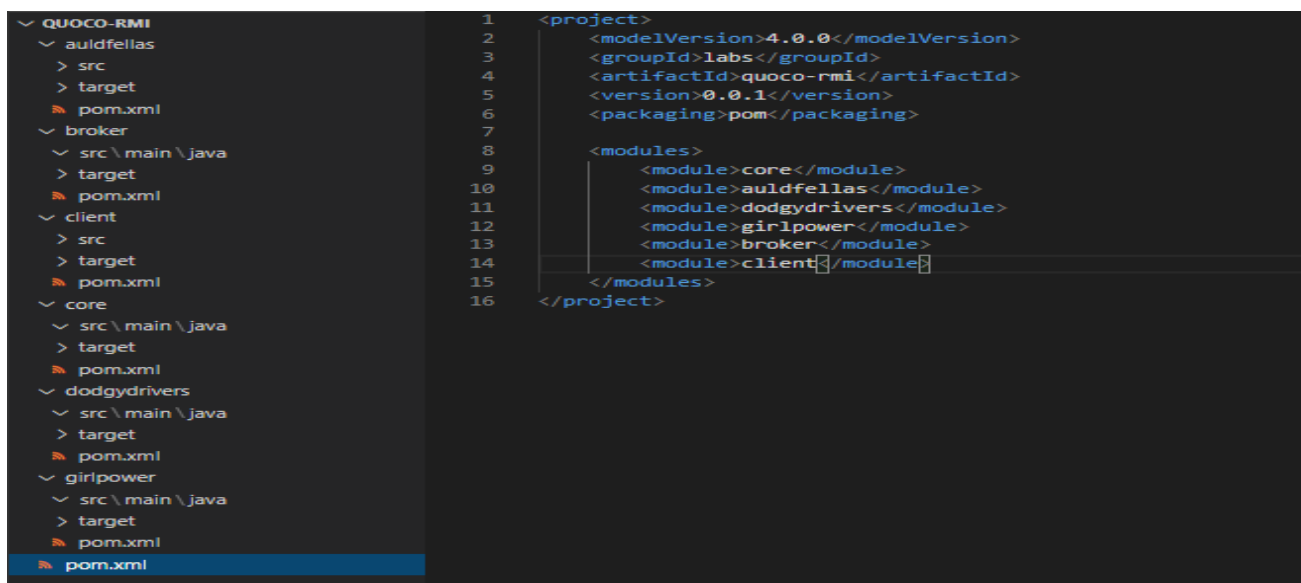
Grade: E

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. These projects should be created as modules within a single multi-module maven project. Please watch the video on how to set up such a project if you are not sure how to do it.

Based on this, you should create a folder called “quoco-rmi” for the main project and add a set of subfolders as follows:

- **core**: contains the common code (distributed object 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

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. Running “mvn compile” in the project root should succeed. ***If it does not compile, then the project is not correctly configured.***



Task 2: Creating the Core project

Grade: D

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

- a) Copy the “service.core” package into “src/main/java” folder of the “core” project.
- b) Modify the BrokerService & QuotationService interfaces to extend `java.rmi.Remote` and for the method signatures to throw `java.rmi.RemoteException` (this is the same as we did when we created the Calculator interface in class). Remove the reference to the `service.registry.Service` interface.
- c) Make the Quotation and ClientInfo data classes implement the `java.io.Serializable` interface.
- d) Compile the (multi-module) project. ***If it does not compile, you have done something wrong, so retrace the steps above.***

Task 3: Creating and Testing the Distributed 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) Copy the auldfellas package into the “src/main/java” folder of the “auldfellas” module.
- b) Create a unit test to check that this class can be compiled and deployed using RMI. To do this, create a “src/test/java” folder in the “auldfellas” project. Next create a file called `AuldfellasUnitTest.java` in the default package of the test codebase and copy the code below into it:

```
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.rmi.server.UnicastRemoteObject;

import service.core.Constants;
import service.core.QuotationService;
import service.auldfellas.AFQService;

import org.junit.*;
import static org.junit.Assert.assertNotNull;

public class AuldfellasUnitTest {
    private static Registry registry;

    @BeforeClass
    public static void setup() {
        QuotationService afqService = new AFQService();
        try {
            registry = LocateRegistry.createRegistry(1099);

            QuotationService quotationService = (QuotationService)
                UnicastRemoteObject.exportObject(afqService, 0);

            registry.bind(Constants.AULD_FELLAS_SERVICE, quotationService);
        } catch (Exception e) {
            System.out.println("Trouble: " + e);
        }
    }
}
```

```

@Test
public void connectionTest() throws Exception {
    QuotationService service = (QuotationService)
        registry.lookup(Constants.AULD_FELLAS_SERVICE);
    assertNotNull(service);
}
}

```

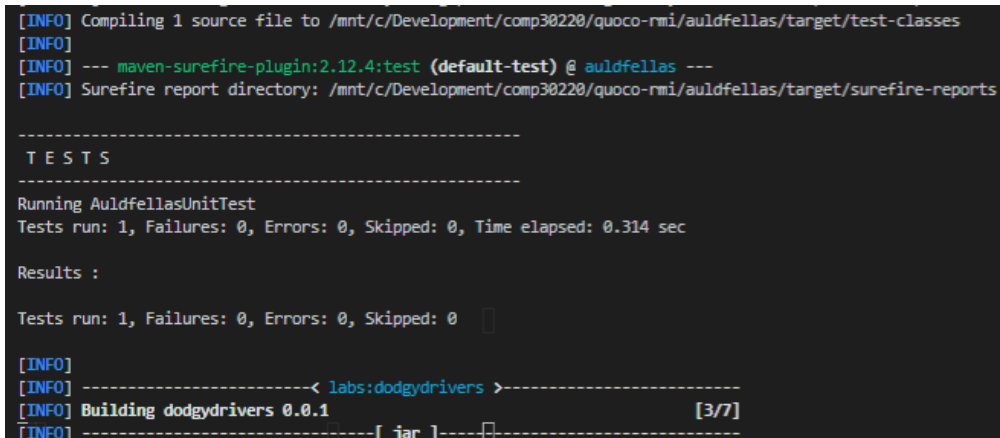
Remember to add the following dependency to the “auldfellas” pom.xml file:

```

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

```

Once completed, simply type “mvn test” in the root folder. You should get a lot of output, specifically look for the section below:



```

[INFO] Compiling 1 source file to /mnt/c/Development/comp30220/quoco-rmi/auldfellas/target/test-classes
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ auldfellas ---
[INFO] Surefire report directory: /mnt/c/Development/comp30220/quoco-rmi/auldfellas/target/surefire-reports

-----
T E S T S
-----

Running AuldfellasUnitTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.314 sec

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO]
[INFO] -----< labs:dodgydrivers >-----
[INFO] Building dodgydrivers 0.0.1 [3/7]
[INFO] -----[ jar ]-----

```

If the test fails, then the Maven build will stop and report the failure.

- c) Add another unit test that checks whether the generateQuotation() method works (and returns a Quotation object).
- d) Now that we have tested that our “auldfellas” quotation service can be deployed and executed using RMI, we need to create a main method to run it. You will notice the method (which is given below) looks a lot like the setup() method we used in the unit test. Create a class in the default package called `Server.java`. Copy the code below into it.

```

import java.rmi.registry.*;
import java.rmi.server.UnicastRemoteObject;

import service.auldfellas.AFQService;
import service.core.ClientInfo;
import service.core.Quotation;
import service.core.QuotationService;
import service.core.Constants;

public class Server {
    public static void main(String[] args) {

        QuotationService afqService = new AFQService();
        try {
            // Connect to the RMI Registry - creating the registry will be the

```

```

        // responsibility of the broker.
        Registry registry = LocateRegistry.createRegistry(1099);

        // Create the Remote Object
        QuotationService quotationService = (QuotationService)
            UnicastRemoteObject.exportObject(afqService, 0);

        // Register the object with the RMI Registry
        registry.bind(Constants.AULD_FELLAS_SERVICE, quotationService);

        System.out.println("STOPPING SERVER SHUTDOWN");
        while (true) {Thread.sleep(1000); }
    } catch (Exception e) {
        System.out.println("Trouble: " + e);
    }
}
}

```

Try to understand what this class is doing – it is basically the same as the CalculatorServer class, but that it is creating the Auldfellas Quotation Service distributed object.

- e) Finally, lets try to run this project. When using a multi-module project, you can target a subset of modules by using the `-pl` flag (followed by a comma-delimited list of module names). For example, we can perform the `exec:java` goal on the auldfellas project by using the following command:

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

However, running this will cause an error:

```

[ERROR] Failed to execute goal on project auldfellas: Could not resolve dependencies for project labs:auldfellas:jar:0.0.1:
Failure to find labs:core:jar:0.0.1 in https://repo.maven.apache.org/maven2 was cached in the local repository, resolution
will not be reattempted until the update interval of central has elapsed or updates are forced -> [Help 1]
[ERROR]
[ERROR] To see the full stack trace of the errors, re-run Maven with the -e switch.
[ERROR] Re-run Maven using the -X switch to enable full debug logging.
[ERROR]
[ERROR] For more information about the errors and possible solutions, please read the following articles:
[ERROR] [Help 1] http://cwiki.apache.org/confluence/display/MAVEN/DependencyResolutionException
ren@Krytan:/mnt/c/Development/comp30220/quoco-rmi$

```

This happens because there is a dependency between the core and auldfellas modules. When you compile the project, the “Reactor” system takes care of this problem, but when you use the custom `exec:java` goal, it does not.

To run the auldfellas module, you need to first install the core module in the local maven repository. You do this by using the `mvn install` command in the project root. Once you have run this install command, you can then run the auldfellas service using the earlier maven command.

- f) Once you are convinced that service works, you will need to make a last change to the Server class so that it is possible to point the service at an existing registry rather than having to create one every time— in the final system, the broker will create the registry, and the quotation services will register with it.

Replace:

```
Registry registry = LocateRegistry.createRegistry(1099);
```

With:

```
Registry registry = null;
if (args.length == 0) {
    registry = LocateRegistry.createRegistry(1099);
} else {
    registry = LocateRegistry.getRegistry(host, 1099);
}
```

To run auldfellas with its own registry, you still use:

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

To run auldfellas with an existing registry, you can use:

```
$ mvn exec:java -pl auldfellas -Dexec.args="localhost"
```

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

NOTE: To run multiple modules at the same time, you will need multiple terminal windows. When you run one of the quotation services, it blocks until you kill it (using CTRL-C). That is why you need a different terminal for each service. A good test to do when you have two services working independently is to run one service so that it creates a registry and the other service so that it uses the created registry (you cannot create two registries because they use the same port).

Task 4: Implementing the Broker and Client

Grade: B

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

- a) Expose the LocalBrokerService as a distributed object and modify the local broker service to use the RMI Registry to find the quotation services. This should all be implemented in the “broker” project. Write unit tests to check that the code works (it should return an empty list of quotations if no services are registered).
- 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 5: Containerisation

Grade: A

The final task is to convert the multi module project you have developed through the first 4 tasks into a set of docker images that can be run from a docker compose file. Use the RMI calculator example as a basis for learning how to do this.

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.