Distributed Programming II

A.Y. 2014/15

Assignment n. 4 – part b)

All the material needed for this assignment is included in the *.zip* archive where you have found this file. Please extract the archive to the same working directory where you have already extracted the material for part a) and where you will work.

This second part of the assignment consists of two sub-parts:

1. Write a Java server application that implements and publishes (using the JAX-WS `Endpoint` class) a simplified version of the web service(s) designed in part a). The simplified version of the web service(s) to be developed must implement only part of the functionalities 2. and 3. specified in part a). More precisely, for what concerns board management, only the operations that start boarding and that register single passengers as boarded have to be implemented. Instead, for what concerns the other operations, only the operations for reading all the available information about flights, flight instances, and passengers have to be implemented. The non-implemented operations that are in the same interfaces as the ones you have to implement can be left empty (returning null arguments, so that the compiler does not complain).

The web service(s) must be initialized with the data retrieved from a data generator that implements the Java interfaces that are in the `it.polito.dp2.FDS` package. The data generator must be created by the web service(s) implementations by using the abstract factory class `FlightMonitorFactory`, as it was done in Assignments 1 and 2. The web service that includes the implemented operations for board management must be published at the URL http://localhost:7070/fdscontrol, and this web service must also publish the WSDL document `FDSControl.wsdl` at the URL http://localhost:7070/fdscontrol?wsdl. Instead, the web service that includes the implemented operations for reading all the available information must be published at the URL http://localhost:7071/fdsinfo, and this web service must also publish the WSDL document `FDSInfo.wsdl` at the URL http://localhost:7071/fdsinfo?wsdl. The server does not have to manage persistency but has to manage concurrency, i.e. more clients can operate concurrently on the same service(s), and even on the same flight instance.

The server main class must be named `FDSControlServer,` the server must be developed entirely in package `it.polito.dp2.FDS.sol4.server`, and the source code for the server must be stored under `[root]/src/it/polito/dp2/FDS/sol4/server` .

Write an ant script that automates the building of your server, including the generation of the necessary artifacts. The script must have a target called `build-server` to build the server. All the class files must be saved under `[root]/build`. Customization files, if necessary, can be stored under `[root]/custom`.
The ant script must be called `sol_build.xml` and must be saved directly in folder `[root].`
Once you have created the ant script as specified, you can run your server with a specific seed and testcase for the data generator by issuing the command (from the `[root]` directory)

```
    ant –Dseed=<seed> –Dtestcase=<testcase> run-server
```

If this command fails it is likely that you have not strictly followed the specifications given above.

Important: your ant script must define `basedir="."` and all paths used by the script must be under `${basedir}` and must be referenced in a relative way starting from `${basedir}`. If other files are necessary (e.g. for customization), they should be saved in the directory `[root]/custom`.

For the purpose of your testing, the random data generator or one of the libraries developed in Assignments 1 and 2 can be used interchangeably.

2. Implement a client for the developed service(s) that can execute the implemented boarding management operations. More precisely, the client must use the information provided in an input XML file for simulating the boarding phase for a flight instance, and finally query the service for the current list of boarded passengers for the same flight instance, and write the list to an output XML file. The name of the input and output XML files are passed as command line arguments (input file name first). The format and meaning of the information of the input XML file is specified by the schema `[root]/xsd/fdsBoarding.xsd` (see the annotations for the meaning). A set of commented XML sample files (with names starting with `fdsBoarding`) is available under `[root]/samples`. The client must exit with exit code 0 if all the boarding management operations have been successfully completed. In this case, before exiting, the client must write the collected final list of passengers to an XML file with the format specified by the `[root]/xsd/boardList.xsd` schema. In case the sequence of operations cannot be completed successfully, the client must exit with exit code 1 if the error was due to wrong data in the input XML file (e.g. trying to board a passenger who did not check in or who was not registered for the flight instance), and with exit code 2 in the other cases (e.g. if the service could not be contacted or some other problem in the server occurred). The URL of the service for board management to be used by the client is specified by the `endpoint` attribute in the root element of the input XML file (you can assume the WSDL is the same as the one you submitted; if not, the client will exit with exit code 2). The URL of the service for reading the information about flights, flight instances, and passengers is fixed (http://localhost:7071/fdsinfo).

The client application main class must be named `FDSControlClient`, the client must be developed entirely in package `it.polito.dp2.FDS.sol4.client`, and the source code for the client must be stored under `[root]/src/it/polito/dp2/FDS/sol4/client`.

In your `build.xml` file, add a new target named `build-client` that automates the building of your client, including the generation of the necessary artifacts. All the class files must be saved under `[root]/build`. You can assume that the server is running when the `build-client` target is called (of course, the target may fail if this is not the case). Customization files, if necessary, can be stored under `[root]/custom`.

Once you have created the ant script as specified, you can run your client with specific input and output files by issuing the command (from the `[root]` directory)

```
ant –Dinput=<inputfilename> –Doutput=<outputfilename> run-client
```

Of course, you should run the server before running the client. If this command fails it is likely that you have not strictly followed the specifications given above.

Correctness verification

Before submitting your solution, you are expected to verify its correctness and adherence to all the specifications given here. In order to be acceptable for examination, your assignment must pass at least all the automatic mandatory tests. Note that these tests check just part of the functional specifications! In particular, they only check that:

- the submitted WSDL files are valid (you can check this requirement by means of the Eclipse WSDL validator);
- the implemented web service(s) and clients behave as expected, in some scenarios: the boarding operations are correctly performed by the client and by the server as specified in the input XML files (i.e. the list of passengers finally returned is coherent with the

information given in the input XML file). This can be verified by feeding the XML sample files to the application and checking that the produced XML files match the ones specified in the input files (see the comment at the beginning of each sample file). Matching of output XML files can be checked by running

```
ant –Dfile1=<filename1> –Dfile2=<filename2> run-check
```

Other checks and evaluations on the code will be done at exam time (i.e. passing all tests does not guarantee the maximum of marks). Hence, you are advised to test your program with care.

You can test your solution by running your server and then by using your client with other different XML input files (you can use the FDSInfo application in order to see the data generated for a given seed and testcase).

Submission format

A single *.zip* file must be submitted, including all the files that have been produced. The *.zip* file to be submitted must be produced by issuing the following command (from the `[root]` directory):

```
$ ant make-final-zip
```

In order to make sure the content of the zip file is as expected by the automatic submission system, do not create the *.zip* file in other ways.