Distributed Programming II

A.Y. 2015/16

Assignment n. 3

All the material needed for this assignment is included in the *.zip* archive where you have found this file. Please extract the archive to an empty directory (that will be called `[root]`) where you will work.

The assignment consists of developing a client, for a web service named *WorkflowInfoService*, using JAX-WS "dynamic proxy" programming. The service provides read access to the workflows available in the system. The service provider can be started on your own local machine by running the command

```
$ ant –Dseed=XXX –Dtestcase=Y run-server
```

The seed *XXX* and the testcase *Y* are used as parameters for the random data generator used by the service (the same random data generator used for Assignments 1 and 2 is used by the service; the generated data can be viewed by means of the WFInfo application). Once the server has been started, the WSDL can be obtained as usual at the URL where the service is available, i.e. `http://localhost:8181/WorkflowInfoService?wsdl`

The semantics of the operations made available by the service is intuitive. One operation (*getWorkflowNames*) provides the list of names of the available workflows, while the other operation (*getWorkflows*) provides the details of a selected set of workflows. No information about processes is provided by the service. As the provided information may occasionally change, the service also provides the last modification time of the data made available.

The client to be developed has to download all the information about the workflows provided by the service and must take the form of a library, similar to the ones developed for Assignments 1 and 2. The library must implement all the interfaces and abstract classes defined in package `it.polito.dp2.WF`, returning the data downloaded from the service (as no information about processes is provided by the service, the client must always return empty lists of processes). The library must include a factory class named `WorkflowMonitorFactory`, which extends the abstract factory `it.polito.dp2.WF.WorkflowMonitorFactory` and, through the method `newWorkflowMonitor()`, creates an instance of your concrete class that implements the `WorkflowMonitor` interface. The class that implements the `WorkflowMonitor` interface must also implement the `Refresh` interface, which is available in source form in package `it.polito.dp2.WF.lab3`. This interface includes the `refresh()` method, which must align the local information about workflows in the client with the information currently provided by the service.

The classes of the solution must be written entirely in package `it.polito.dp2.WF.sol3`, with all their sources stored in folder `[root]/src/it/polito/dp2/WF/sol3/`.

The actual URL used by the client class to contact the service must be customizable: the actual URL has to be read as the value of the `it.polito.dp2.WF.sol3.URL` system property.

The artifacts have to be generated from the WSDL using the ant script provided with this assignment. The generation of artifacts can be started by running the command

```
$ ant compile-wsdl
```

As you can see by inspecting the ant script, the `compile-wsdl` target gets the WSDL from the server, using the same service location specified in the system property mentioned above. For this reason, **the server must be running when this target is invoked**. If you need custom bindings,

you have to write them in the file `[root]/custom/binding.xml`.

The client classes must be robust and interoperable, without dependencies on locales. However, these classes are meant for single-thread use only, i.e. the classes will be used by a single thread, which means there cannot be concurrent calls to the methods of the classes.

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 data returned by the client are the same that are provided by the service.

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).

The *.zip* file of this assignment includes a set of tests like the ones that will run on the server after submission. Note that the server exposes another web server that can be used to alter the set of workflows provided by the service and that you can use for your additional tests. The WSDL of this web service is available at `http://localhost:8181/WorkflowSetService?wsdl`

The tests can be run by the ant script included in the *.zip* file, which also compiles your solution. Of course, before running the tests you must have started your server. Then, you can run the tests using the `runFuncTest` target. All the automatic tests use the random data generator provided with this assignment. For this reason, it is essential that **the server and the runFuncTest target are executed by setting the same testcase** (the seed can be different, but it is important to set it in order to avoid problems with non-admissible seed values that sometimes may be generated)

This is an example of how to properly run the two commands

```
ant –Dtestcase=2 –Dseed=12345 run-server
ant –Dtestcase=2 –Dseed=12345 runFuncTest
```

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
```

Do not create the *.zip* file in other ways, in order to avoid the contents of the zip file are not conformant to what is expected by the automatic submission system.
Note that the *.zip* file **must not** include the files generated automatically.