Distributed Programming II

A.Y. 2015/16

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 three 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 functionalities 1 and 2 specified in part a). 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.WF` package. The data generator must be created by the web service(s) implementations by using the abstract factory class `WorkflowMonitorFactory`, as it was done in Assignments 1 and 2. The web service that includes the operations for creating new processes must be published at the URL http://localhost:7070/wfcontrol, while the web service that includes the operations for reading all the available information must be published at the URL http://localhost:7071/wfinfo (if the service is the same, it must be published at both URLs). The WSDL document designed in part a) must be made available as usual at the URLs http://localhost:7070/wfcontrol?wsdl and http://localhost:7071/wfinfo?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 workflow.

The server main class must be named `WorkflowServer`, the server must be developed entirely in package `it.polito.dp2.WF.sol4.server`, and the source code for the server must be stored under `[root]/src/it/polito/dp2/WF/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 (by just changing the `it.polito.dp2.WF.WorkflowMonitorFactory` system property).

2. Implement a client for the web service that provides information about workflows and processes. The client must take the form of a library similar to the ones implemented in Assignments 1 and 2. The library must load information about workflows and processes from the web service at startup. The library must implement all the interfaces and abstract classes defined in package `it.polito.dp2.WF`. The classes of the library must be entirely in package `it.polito.dp2.WF.sol4.client1` and their sources must be stored in folder `[root]/src/it/polito/dp2/WF/sol4/client1`. The library must include a factory class named `it.polito.dp2.WF.sol4.client1.WorkflowMonitorFactory`, which extends the abstract factory `it.polito.dp2.WF.WorkflowMonitorFactory` and, through the method `newWorkflowMonitor()`, creates an instance of your concrete class implementing the `WorkflowMonitor` interface. The URL of the web service to be used for getting the information must be obtained by reading the `it.polito.dp2.WF.lab4.URL` system property. If this property is not set, the default URL http://localhost:7071/wfinfo, stored in your WSDL file, must be used.

In your `sol_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`.

3. Implement another client application for the developed web service(s) that can invoke the operation for creating a new process. The client must create a single new process and then it must terminate. The client must receive the URL of the service to be contacted as the first command-line argument, and the name of the workflow to be used for the process as the second command-line argument. The client has to assume the service available at the specified URL implements exactly the same WSDL designed in part a), the URL being the only possible difference. If the operation is completed successfully, the client must exit with exit code 0, while if an error occurs the client must exit with exit code 1 if the service was contacted, but an error occurred in the execution of the operation (e.g. because the workflow name was not known by the service), and with exit code 2 in all other cases (e.g. service unreachable).

The client application main class must be named `WFControlClient`, the client must be developed entirely in package `it.polito.dp2.WF.sol4.client2`, and the source code for the client must be stored under `[root]/src/it/polito/dp2/WF/sol4/client2`.

Update your `sol_build.xml` file, so that the target named `build-client` also builds your second client. 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 URL and workflow name by issuing the command (from the `[root]` directory)

`ant -DURL=<`*URL*`> -DWorkflowName=<`*workflowname*`> run-client2`

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 client behave as expected, in some scenarios: after calling a process creation operation (by means of your client2), the information about workflows and processes returned by your service is consistent with the performed operation.

The same tests that will run on the server can be executed on your computer by issuing the following command

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

Note that this command also runs the server. Before issuing it, make sure that your server is not running. If it is, you need to kill it before starting the tests. In order to run tests correctly it is necessary to have at least one workflow. For this reason, testcase 3 cannot be used.

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 also test your solution by running your server and then by using your client in different scenarios.

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.