# Distributed Programming II

A.Y. 2018/19

## *Assignment n. 2*

All the material needed for this assignment is included in the *.zip* archive where you have found this file, except the jars of the JAX-RS libraries and related dependencies that are provided separately. Please extract the archive to an empty directory (that will be called `[root]`) where you will work.

The additional jar files necessary for the assignment are available in the course web site. In the Labinf machines and in the VM they are already installed under `/opt/dp2/shared/lib`. When setting the build path of your project, add all these jar files, in addition to the ones under `[root]/lib`. Also, you are adviced to attach (right click on library, Properties, Java Source Attachment) the source jars found under `[root]/lib-src` (`junit-4.12`, `javax.ws.rs-api` and `hamcrest-core` libraries).

The assignment consists of developing a client for the RESTful web service of the *Neo4J* graph-oriented DB, using the JAX-RS framework.

A documentation of the Neo4J REST API is included in the NEO4J manual https://neo4j.com/docs/pdf/neo4j-manual-2.3.12.pdf (also included in this package), chapter 21.

The client to be developed must be able to: a) read the information about a set of places and about their connections, from the random generator already used in Assignment 1; b) load the graph of these places with their connections into NEO4J by means of the *Neo4J REST API* as specified below; c) answer queries about shortest paths between places, by properly getting this information from *the REST API* as specified below.

The places have to be loaded into NEO4J as follows: a graph node has to be created for each **place**, with a property named **"id"** with the value that is the **place id**; a relationship has to be created for each **connection,** connecting the nodes of the corresponding places, **with type "ConnectedTo"**. For this exercise, don't use the transactional Cypher HTTP endpoint, but use the endpoint with the service root described in section 21.4 of the above-mentioned manual (differently from the other endpoint, this second endpoint is a real RESTful web service, and it does not require that you learn the Cypher language). Sections 21.8 and 21.9 of the documentation explain how to create and read nodes and relationships while section 21.18 explains how to use the shortest path algorithm available in NEO4J. The client has to access the service without authentication. In order to make this possible, it is necessary to disable authentication in the NEO4J configuration (in the VM and in the Labinf machines, authentication is already disabled).

Only the above mentioned data have to be stored in NEO4J. Any additional information, (e.g. the addresses of the graph nodes that are necessary to perform further operations on the graph) has to be stored in the client.

The client to be developed must take the form of a Java library that implements the interface `it.polito.dp2.RNS.lab2.PathFinder`, available in source form in the package of this assignment, **along with its documentation**. This interface enables the operations a), b) and c) mentioned above. More precisely, the library to be developed must include a factory class named `it.polito.dp2.RNS.sol2.PathFinderFactory`, which extends the abstract factory `it.polito.dp2.RNS.lab2.PathFinderFactory` and, through the method `newPathFinder()`, creates an instance of your concrete class that implements the `it.polito.dp2.RNS.lab2.PathFinder` interface. All the classes of your solution must belong to the package `it.polito.dp2.RNS.sol2` and their sources must be stored in the directory `[root]/src/it/polito/dp2/RNS/sol2`.

In your development you can use automatically generated classes or an automatically generated schema. In this case, you have to create an ant script named `sol-build.xml`, with a target named `generate-artifacts`, and place it in `[root]`. When invoked, the `generate-artifacts` target must generate the source code of the classes in the folder `[root]/gen-src` or the schema in the folder `[root]/gen-schema`. An empty version of this script is already present in the Assignment zip archive. The main ant script `build.xml` will automatically call this script before compiling your solution and will automatically compile the generated files, if any, and include them in the classpath when running the final tests. Note that the generated classes must belong to sub-packages of the solution package. Custom files needed by your solution or ant script (e.g. a schema) have to be stored under `[root]/custom`.

The actual base 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.RNS.lab2.URL` system property.

The client classes must be robust and portable, 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 these classes. When developing the client, consider that the operation that finds paths can be time-consuming.

## 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 client behaviour is consistent with the data received from the data generator and that the nodes and relationships are created on the server according to the specifications.

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.

The tests can be run by the ant script included in the *.zip* file, which also compiles your solution. Before running the tests you must have started the NEO4J server by running the following ant command:

```
ant start-neo4j
```

Then, you can run the tests using the `runFuncTest` target, which also accepts the `-Dseed` and `-Dtestcase` options for controlling the random generation of data. The results of the junit tests can be displayed graphically by double clicking on the `testout.xml` file in Eclipse. Note that the execution of these tests may take some time when successful, depending on the seed, and during the junit tests no output is produced, until the tests are finished. It is recommended that, before trying the automatic tests provided in the package, you create a main that lets you test the single calls to your client library and lets you debug your library.

If you want to clear the contents of the NEO4J DB you can call the `clear-and-restart-neo4j` target.

## 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-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.