Distributed Programming II

A.Y. 2019/20

Lab4

All the material needed for this Lab 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 suggested 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).

Purpose

The aim of this Lab is to experiment with consuming RESTful web services in Java, using the JAX-RS framework. The exercises give you the opportunity to apply the knowledge acquired about REST and about consuming RESTful web services in Java .

Exercise 1

1. Complete the development of a client for the RESTful web service of the Neo4J graph-oriented DB, which has been initiated in the classroom, using the JAX-RS framework.

   The service can be started locally on your machine by running the following ant command:

   ```
   ant start-neo4j
   ```

   A web application GUI for the service is available at `http://localhost:7474` while 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 a Java application with the following features: a) it reads the information about a set of bibliographic items (books or articles) from the data generator already used in Assignment 1, Lab 2; b) it loads the graph of these items (i.e. a graph having the items as nodes and the citations as directed edges) into a NEO4J DB by means of the *Neo4J REST API* as specified below; c) once finished loading, it answers queries about indirectly citing items, by properly getting this information from the *Neo4J REST API* as specified below.

   The items have to be loaded into the NEO4J DB as follows: a graph node has to be created for each **item**, with a property named **"title"**; a relationship has to be created for each **citation,** i.e., for each item I0 having a non-empty set of citing items I1, I2, …, a relationship has to be created for each citing item Ii, connecting the node that represents I0 to the node that represents Ii. The relationship **must be given type "CitedBy"**. 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 21.17 explains how to perform a graph traversal, which is useful for getting the indirectly citing items. This last operation consists of computing the items that cite directly or indirectly a given item, up to a given depth. It can be implemented by performing a traversal of the graph starting at the node that represents the given item, with a maximum depth equal to the given depth, following the outgoing "CitedBy" relationships and getting the visited nodes. 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, as explained in the README file (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.BIB.ass2.CitationFinder`, available in source form in the package of this assignment, **along with its documentation**. This interface enables the operation c) mentioned above, while the operations a) and b) must be performed by the constructor of the implementation class. More precisely, the library to be developed must include a factory class named `it.polito.dp2.BIB.sol2.CitationFinderFactory`, which extends the abstract factory `it.polito.dp2.BIB.ass2.CitationFinderFactory` and, through the method `newCitationFinder()`, creates an instance of your concrete class that implements the `it.polito.dp2.BIB.ass2.CitationFinder` interface. All the classes of your solution must belong to the package `it.polito.dp2.BIB.sol2` and their sources must be stored in the directory `[root]/src/it/polito/dp2/BIB/sol2`. However, if you want, you can use the classes developed in the classroom (their sources are available under `[root]/src`).

In your development you can use automatically generated classes or automatically generated schemas. If you use automatically generated classes, the schemas must be put in the `[root]/xsd` folder and the generated classes into `[root]/gen-src` (in sub-packages of `it.polito.dp2.BIB.sol2`). If you use automatically generated schemas, the generated schemas must be put into `[root]/gen-schema`. In order to perform generation as needed in your solution, you must complete the ant script `[root]/sol-build.xml`, which already includes an initially empty target named `generate-artifacts`. The main ant script `build.xml` will automatically call this target before compiling your solution and it will automatically compile the generated files, if any, and include them in the classpath when running the final acceptance tests. Any other custom files needed by your solution or ant script have to be stored under `[root]/custom`.

The actual base URL and port of the service to be contacted by the client must be customizable: the actual URL and port have to be read as the values of the following two system properties: `it.polito.dp2.BIB.ass2.URL` and `it.polito.dp2.BIB.ass2.PORT`.

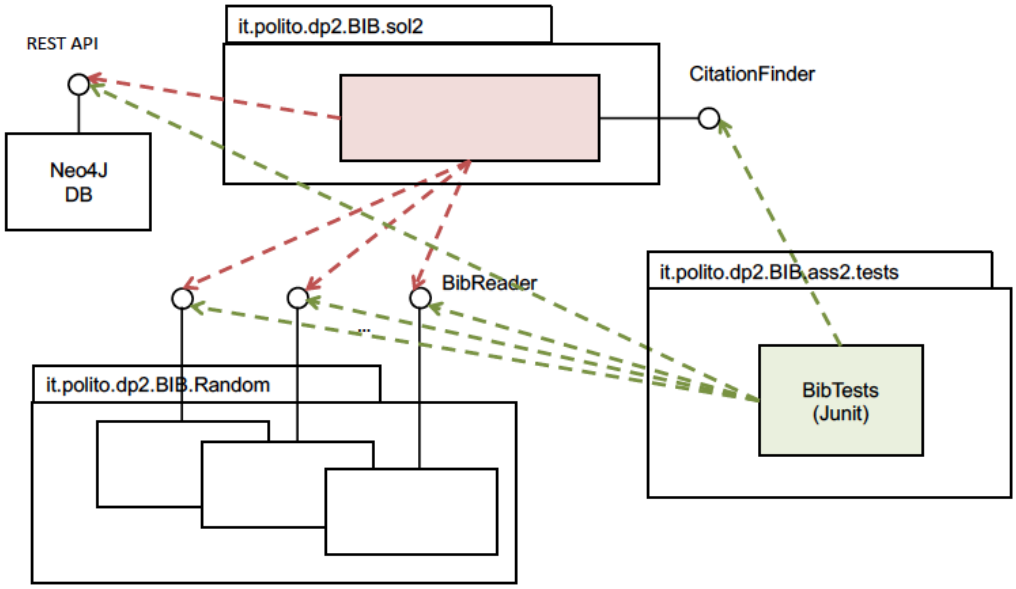
The client classes must be robust and portable, without dependencies on locales. However, these classes are meant for single-thread use only, i.e. the classes will be used by a single thread, which means there cannot be concurrent calls to the methods of these classes.

2. Test your implementation class by creating a main that instantiates the class and performs some find operations. Remember that, in case of errors related to the interaction with the service, you can use wireshark in order to understand the cause of the error. Moreover, you can use the NEO4J GUI in order to check the effects of the operations made on the DB.
3. Once you are confident that your implementation is correct, you can run the final acceptance tests, by running the command

   ```
   ant runFuncTest
   ```

   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 amount of data. During this time, no output is produced, until the tests are finished. If you want to clear the contents of the NEO4J DB you can call the `clear-and-restart-neo4j` ant target.

   The way the final tests are organized is shown in the following picture:



Exercise 2 (Optional)

1. Starting from the client for the *Google Books REST API* developed in the classroom, and included in the Lab material (package `it.polito.dp2.rest.gBooks.client`), create an extended version of it, with the following features:
   - The main class of your client must be `it.polito.dp2.BIB.sol2.BookClient_e` and any other class developed as part of the solution of this exercise must be put in the same package.
   - Your client must behave as the client developed in the classroom, but it must query also the *Crossref* repository, which is available through its public REST API (http://api.crossref.org). More precisely, the application must create `PrintableBook` objects for the first N valid results returned by *Google Books* and for the first N valid results returned by Crossref, where N is the first (and mandatory) argument on the command line (while the search keywords are given as the second and subsequent arguments). After having created the objects, the application must print them, by calling their print methods.
2. Test your client by running it with different keywords as input, and check that the printed results match the data obtained by accessing the same information manually by means of your browser.

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.

The **deadline for submission** of this assignment is December 8, 2019 23:59 CET.