Distributed Programming II

A.Y. 2018/19

Assignment n. 3 – 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. In this way, you should have the xsd and HTML documentation developed in part a) in the `[root]/xsd` and `[root]/doc` folders respectively.

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

1. Write a simplified Java implementation (using the JAX-RS framework) of the web service designed in part a). The implementation should include all the main functions specified in part a), but, in order to keep the implementation simple, if there are some features that are functionally not essential (e.g. paging, or some query parameters that have been introduced merely for performance or scalability reasons), they can be omitted in the implementation.

The web service has to be packaged into a war archive that can be deployed to Tomcat. This is done automatically by the ant script `[root]/build.xml`, which includes a target named `package-service` that builds and packages the service. The name of the service packaged in this way will be "RnsSystem", and its base URL will depend on the configuration of Tomcat (it will be http://localhost:8080/RnsSystem/rest on the Labinf machines and on the VM). A WADL description will be available at the standard Jersey-provided WADL location (on the Labinf machines, this is http://localhost:8080/RnsSystem/rest/application.wadl). The service must have no dependency on the actual location of Tomcat, nor on the Tomcat port.

The service does not have to provide data persistency (data will be stored in main memory) but has to manage concurrency, i.e. several clients can operate concurrently on the same service without restrictions.

When deployed, RnsSystem must be initialized by reading the data about places and relative connections from the RnsReader interface already used for Assignment 1 (defined in package `it.polito.dp2.RNS`), using the same abstract factory already used for Assignment 1, while information about vehicles will not be considered (i.e., initially there will be no tracked vehicle in the system). The information about places and their connections is never updated (i.e., the service always uses the information read at startup).

The service must exploit the NEO4J REST API (the one already used for Assignment 2) in order to store the graph of places and their connections, in the same way done for Assignment 2, and in order to compute the suggested path for a vehicle.

In order to compute the suggested path, the service finds the shortest paths between source and destination by means of NEO4J, in the same way done in Assignment 2. Then, if more than one path is found, it selects one. The criterion for selection can be chosen arbitrarily by the service (for simplicity, the service could just select one path randomly). Note that, for simplicity, the service is not requested to guarantee that the maximum number of vehicles in a place is respected. You are suggested to re-use the library you developed in Assignment 2 for the interaction with the NEO4J service (graph load and search for shortest paths).

RnsSystem has to read the actual base URL of the NEO4J service as the value of the system property `it.polito.dp2.RNS.lab3.Neo4JURL`. If the property is not set, the default value to be used is http://localhost:7474/db

The service must be developed entirely in package `it.polito.dp2.RNS.sol3.service`,

(and sub-packages) and the corresponding source code must be stored under `[root]/src/it/polito/dp2/RNS/sol3/service`. If you want to re-use code from the previous assignments, you can leave it in the `sol1` and `sol2` packages, but you have to copy the source code to this project (i.e. to `[root]/src/it/polito/dp2/RNS/sol*`).

Write an ant script that automates the building of your service, including the generation of any necessary artifacts. The script must have a target called `build-service` to build the service. All the class files must be saved under `[root]/build`. Customization files, if necessary, can be stored under `[root]/custom`, while XML schema files can be stored under `[root]/xsd`. Of course, the schema file developed in part a) can also be used (it is stored in `[root]/xsd`, as specified previously).
The ant script must be named `sol_build.xml` and must be saved directly in folder `[root]`.
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}`. Any other files that are necessary and that are not included in the other folders should be saved in `[root]/custom`.

The `package-service` target available in `build.xml` calls the `build-service` target of your `sol_build.xml` and then packages the service into the archive `[root]/war/RnsSystem.war`. The descriptor of the service used for the package is under `[root]/WebContent`. The contents of this folder can be customized by you. The deployment of the package to the Tomcat server can be done by calling the `deployWS` target available in the `build.xml` script. Note that this target re-builds the war. Before running this target, you need to start Tomcat. This has to be done by calling the `start-tomcat` ant target on `build.xml` (which also sets the expected system properties; **do not run Tomcat in another way otherwise the system properties may not be set correctly**).
Complete the documentation you already developed for part a), by adding another HTML document that illustrates the most relevant implementation choices you made (including any assumptions you made, and which parts of the design, if any, were omitted in your implementation). Limit the size of this document to 1 page. Store this document in the `[root]/doc/impl` folder and use the name `index.html` for its entry point. Add the contents of all your `[root]/doc` folder to the `[root]/WebContent` folder, so that it can be browsed from Tomcat when the service is deployed.
Before proceeding with the next parts, you are advised to test your service by means of the postman client (or other HTTP client) and debug it.

2. Implement an administration client for your web service (designed in part a and implemented in part b.1), which can provide read access to information about places, their connections, and vehicles in the system. The client must take the form of a library similar to the one implemented in Assignment 1, but with a different main interface, which extends the one used for Assignment 1. The library must load information about places and connections from the web service at startup (this information will be assumed as fixed, during the whole execution time of the client). The library must implement the interface is `it.polito.dp2.RNS.lab3.AdmClient` (available in source form in the zip archive of this assignment, with documentation included in the source files), and the interfaces in package `it.polito.dp2.RNS` referenced by it. Note that, as documented, the `getVehicles` and `getVehicle` methods inherited from RnsReader should return an empty set or null). The classes of the library must be entirely in package `it.polito.dp2.RNS.sol3.admClient` and their sources have to be stored in `[root]/src/it/polito/dp2/RNS/sol3/admClient`. The library must include a factory class named `it.polito.dp2.RNS.sol3.admClient.AdmClientFactory`, which extends the abstract factory `it.polito.dp2.RNS.lab3.AdmClientFactory` and, through

the method `newAdmClient()`, creates an instance of your concrete class implementing the `it.polito.dp2.RNS.lab3.AdmClient` interface. The actual base URL of the web service to be used by the client must be obtained by reading the `it.polito.dp2.RNS.lab3.URL` system property. If this property is not set, the default URL http://localhost:8080/RnsSystem/rest must be used. Note that your client cannot even assume that port 8080 is 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 artifacts if necessary. All the class files must be saved under `[root]/build`. You can assume that Tomcat is running with your RnsSystem already deployed when the `build-client` target is called (of course, the target may fail if this is not the case). In your script, you can assume that Tomcat is available on the localhost, with the port number specified by the ant property PORT (inherited automatically by your script). Do not use hard-coded values for the port. Customization files, if necessary, can be stored under `[root]/custom`.

3. Implement a vehicle client for the web service designed in part a and implemented in part b.1. This client must take the form of a Java library that implements the interface `it.polito.dp2.RNS.lab3.vehClient`, available in source form in the package of this assignment (its documentation is included in the source file).

The library to be developed must include a factory class named `it.polito.dp2.RNS.sol3.vehClient.VehClientFactory`, which extends the abstract factory `it.polito.dp2.RNS.lab3.VehClientFactory` and, through the method `newVehClient()`, creates an instance of your concrete class that implements the `it.polito.dp2.RNS.lab3.VehClient` interface.

The client should be developed entirely in package `it.polito.dp2.RNS.sol3.vehClient`, and the source code for the client must be stored under `[root]/src/it/polito/dp2/RNS/sol3/vehClient`. The actual base URL of the web service to be used by the client must be obtained in the same way done for sub-part 2.

Update your `sol_build.xml` file, so that the target named `build-client` also compiles your client, including the generation of artifacts if necessary.

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 schema is valid (you can check this requirement by means of the Eclipse XML schema validator);
- the implemented web service and clients behave as expected, in some scenarios: vehicle clients can perform the various operations allowed by the interface and the information read by the administration client is consistent with the operations performed by the vehicles.

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

```
ant -Dseed=<seed> run-tests
```

Note that this command also deploys your service. However, before running this command you have to start Tomcat and Neo4J on your local machine, as already explained for Assignment 2.

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 (e.g. you can also test your solution by running your service and then test it by using your clients in different scenarios, and by means of other general-purpose clients).

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
```

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.