User Phone Record Database

Stores and retrieves information about users and their phone network use

Usage Instructions

Requirements

- JRE System Library 1.8
- JUnit 4
- MySQL Server version 5.7.17
- Eclipse

Before running the project, you must ensure that there is an appropriate database set up on your MySQL server. To do this, create a new database in mysql with an appropriate name. I have chosen the name "knowroaming_eidelman". If you use a different name, please modify the first twl lines of "generate_db.sql" to reflect your name choice.

To import the database, run the following command:

```
mysql -u USERNAME -p  < generate_db.sql
```

Where "USERNAME" is your mysql username.

Finally, update the settings.txt file to reflect your MySQL settings. The first line should be your MySQL username. The second line should be your MySQL password. The third line should be the database name (knowroaming_eidelman by default).

Running the Project

This project is packaged as an Eclipse project. In Eclipse, go to

```
file menu -> import -> Existing Projects into Workspace
```

And select the root directory of this git repo as the root directory of the project. You may have to set the build path to include your own JRE (1.8 or higher). Project compliance should be 1.7 or higher.

I have also included a prebuilt runnable JAR file in the *jar/* directory. This can be run with the following command while in the root directory:

```
java -jar jar/KnowRoaming.jar
```

following would also work:

```
java -jar KnowRoaming.jar /PATH_TO/settings.txt
```

To rebuild the KnowRoaming.jar file, run the following command in the root directory:

```
ant jar
```

KnowRoaming App Commands

There are three commands in the KnowRoaming APP

NEWUSER Command

```
NEWUSER name email phoneNumber
```

This command creates a new user with the given name, email and phone number in the database. If successful, this command will spit out the unique 10 character ID of the user. Otherwise it will give an error message. The "name" category is allowed to have single quotes around it, in which case we can give names that have spaces in them.

Example:

```
NEWUSER 'Jonathan Eidelman' jonathan.eidelman@email.com 123-4567
```

USAGE Command

```
USAGE userId dd-MM-yyyy dataType
```

This command enters a new usage record into the database. The new usage record is associated with a particular user identified by userId, has the time stamp given in the date and can have any allowable dataType. By default, the database is preset with the following datatypes:

- SMS
- ALL
- DATA
- VOICE

SHOW Command

This command will spit out all usage records associated with the user identified by user_id between the first date given and the last date given.

Design Decisions

I have implemented a simple framework to encapsulate interaction with the MySQL database. Each type of record in the Database (user_record, usage_record) has a class associated with it that implements the SQLRecord class. These records are able to commit themselves to the database, update records in the DB as well as update themselves to match a corresponding record in the DB.

I have also decided to encapsulate the communication between SQLRecord instances and the MySQL server into a new class called SQLCommunicator. This shields SQLRecords from having to deal with the complexities of SQL statements. SQLCommunicator initiates connection upon construction, and after that exposes methods for querying and updating the database.

In the instructions, we were asked to assign each user a unique ID that is ten characters long. I interpretted this to mean that these ID's should be a random set of characters (rather than simply having MySQL assign an integer ID and adding 0's to it to make it a unique ID). Although the likelihood of generating the same key twice is extremely low, there is the possibility we would do this.

I have made the assumption that the requirement that a User ID be unique is a hard requirement, and that we are not allowed to fail to commit a user just because we generated an ID that already exists (even though this situation is highly unlikely). Therefore, I have implemented a loop that repeatedly generates ID's and attempts to commit the UserRecord. If the DB raises an exception, we go back and generate a new ID. Since generating the same ID twice is extremely unlikely, this loop will only ever execute once in the overwhelming majority of cases.

Due to the simplicity of the assignment, I have designed a very rudimentary parser for command line inputs. In a more realistic version of this tool, we could use one of the many available parser generators for Java to create something more robust.

Testing

I have written JUnit tests to ensure the behaviour of this tool meets its specification. These tests are located in Tests.java. They require that the database be set up, and they should be run using the Eclipse JUnit plugin.