Tutorial 4 – Data Persistence in Java

The purpose of this tutorial is to introduce how to implement CRUD (Create, Read, Update, and Data) data persistence using Java and JDBC.

JDBC, stands for Java Database Connectivity and is similar to Microsoft's ODBC Object Database Connectivity. JDBC and ODBC both abstract aspects of connecting and interacting with databases so the programmer can write general database code and provide a database driver (in this case a JDBC driver) for a specific backend database. This allows the "underlying" database to be quickly changed with few or no code changes.

Check out the constructor for Model.java. Inside the constructor, the database driver is automatically injected into the code. If, for example, this code were to connect to a MySQL database, then the postgresql driver could be replaced with an appropriate declaration for the MySQL driver, and the MySQL JDBC driver could then be added as a maven project dependency.

1. Download and Configure the latest sample_maven_web_app

First, delete the your old sample_maven_web_app git repository by going to github.com using a web browser.

Next, navigate to the sample_maven_web_app repository and fork the latest version into your account to replace the deleted version. In the upper-right had portion of the screen click on the ***"Fork"*** Icon.

Fork:
https://github.com/wjlloyd/sample_maven_web_app

Now, from the command line, create a new directory, change into this directory (cd {new_dir}, and clone your forked code:

```
git clone git@github.com:{your-user-name}/sample_maven_web_app.git
```

From this directory, log into your heroku account:

```
heroku login
```

Then create a new heroku project for this source tree:

Note: you may want to purge old heroku projects since free accounts are limited to (4) projects.

```
heroku create
```

Next, add the postgresql add on to this project:

```
heroku addons:create heroku-postgresql:hobby-dev
```

Now you will want to create tables for this project.
The tables are described in the file: src/sql/create_database.sql.

Running psql on the command line requires installation of the postgresql-client package on Ubuntu: **`sudo apt-get install postgresql-client`**

Alternatively, create tables can be created using the PGAdmin GUI and loading the SQL file.

To create tables using the command line:

```
heroku pg:psql
```

Then:

```
\i src/sql/create_database.sql
```

Next, check that the tables have been created:

To see the list of tables:

```
\dt
```

And the structure of the tables:

```
\d+ public.users
\d+ public.messages
```

2. Build the project, and test locally

Now, using Netbeans, perform a “Clean and Build” of the project source.

From the command line, launch a local instance of the application using the script file called localrun.sh.

From the sample_maven_web_app project source directory:

```
./localrun.sh
```

This will launch the application locally using your local Apache Tomcat server. It will take several seconds to initialize. The application will be hosted at localhost:8080.

Once launched, you should be able to navigate to the following URL:

http://localhost:8080/tcss360/users

The script automatically generates the JDBC_DATABASE_URL environment variable setting based on information provided from heroku. This provides the connection string for the application.

Check out the localrun.sh script.

```
cat localrun.sh
```

If the project has been previously deployed to heroku, then heroku will have the value in the variable $JDBC_DATABASE_URL, so we retrieve it to set the local value.

If not, the database connection URL must be built. All of the required connection information is available from pg:credentials, but it is not in the proper order, so the information is parsed and a connection string is built. This is done in the script.

To use a locally hosted postgresql database server, or another server other than heroku, a different database connection URL could be provided in the localrun.sh script.

Next,test the CREATE, READ, UPDATE, and DELETE tasks under src/scripts. To run these tests against your locally hosted application, uncomment the line for “local web runner deploy”

Try each of the scripts to create, update, delete, and read data...

3. Deploy the project, and test the remote URL

Now, deploy the application to the cloud:

```
git push heroku master
```

Try visiting the web page for your heroku app.
Add “tcss360/users” to the URL provided from heroku.

Now, try out the CREATE, READ, UPDATE, and DELETE scripts under src/scripts for the cloud hosted version. The application should automatically connect to the same database.

To run these tests against your remotely hosted application, update the ENDPOINT variable setting in each of the scripts and point to your heroku hosted web app.

4. Implement Create Read Update Delete operations for

Inspect the code for create, read, update, and delete operations.

Create a new data class for messages in the same package as users.

Once you create the messages class, define the primitives:

```
private int messageId;
private int userId;
private String message;
private Date dateadded;
```

Note for the ObjectMapper that converts JSON into Java objects, if the field has a capital letter such as I in messageId, then the JSON object will be expected to specify it the same way. So JSON to Java conversion is case sensitive.

Once primitives are defined go to **S**ource | **I**nsert Code from the pull-down menu and select “Getter and Setter…” to automatically generate getter and setter methods.

Create a new service class for messages following the same design as the UserService class. You’ll need to create an empty constructor, and 4 methods, one for each crud operation. You’ll need to use language annotations as in UserService to indicate which parts of the class to provide to the service endpoints, and also define a path to the services:

```
@Path("Messages")
```

Using your messages services it should be possible to:

1. ***Create a new message***: The database should automatically generate a unique message id for each message. Messages should be linked to users via the userid. Messages should also have the message date/time. Use the postgresql function now() in SQL to automatically generate a time for new messages.

2. ***View all messages***: When calling the get service endpoint, HTML output should be generated to display all of the messages in a table. The table should display: the message ID, the user ID, the message text, the message creation time. As a **BONUS** try displaying the name of the user if you’d like. This will require a joining the two tables in the SQL query.

3. ***Update a message***: When updating a message the text of the message, time, and user id should be changeable. The message id is immutable. It should not be allowed to change as it is the primary key.

4. ***Delete a message***: Support deleting messages by the message ID.

Create new methods in the Model class to persist (create, read, update, and delete) messages. Note for now, you may simply add additional methods to the Model class, but ideally for a larger application a parent Model class could be defined which implements base methods, and children Model classes would be implemented to handle persistence for different types of data. (e.g. users, messages, etc.)
Here is a sample JSON for a message object: (create)

```
{
  "userId": 1,
  "message":"Hello Mr. Fred",
  "dateadded":"Mon 6 Feb 2017 12:00:00 PST"
}
```

To update or delete, it is necessary to provide a messageId.

5. BONUS EXERCISE: Change the User Class to use the Message Class

The initial implementation of the user class simply has a generic list of text messages. Once CRUD for messages is complete, try having the user class use the messages class. Try having the user class use a type specific list List<T> for messages where the type T is the Message class as it List<Message>.

This bonus exercise involves having the user class use composition to include messages instead of trying to implement messages within the user class. This supports the single responsibility design guideline.

A call to the user create service should also create messages nested within the user JSON object.

Here is the format of the user JSON object. Note, the userId is not provided because it must be created first, then passed in as an input to create the related messageIds:

```
{
  "name": "Fred Smith",
  "age": 33,
  "messages":
  [
    {
      "message":"Hello Compound Fred 1!"
    },
    {
      "message":"Hello Compound Fred 2!"
    }
  ]
}
```

One call to create User will create 1 user object, and 2 message objects.
Here is what the message objects look like in postgreSQL:

```
 messageid | userid |        message         |      dateadded
-----------+--------+------------------------+---------------------------
         9 |     12 | Hello Compound Fred 1! | 2017-02-05 21:30:06.503135
        10 |     12 | Hello Compound Fred 2! | 2017-02-05 21:30:06.585067
```

6. Receiving Credit For This Tutorial

To receive credit for this tutorial, have one person in your group submit:

1. The URL of your endpoint in Heroku with the completed crud services for messages. Be sure that database connectivity is working.

2. The URL of your github repository.