DATA 514 Homework 5: Database Application and Transaction Management

Objectives: To get experience with database application development and transaction management. To learn how to use SQL from within Java via JDBC.

Assignment tools: SQL Server through SQL Azure and starter code.

Additional files (you normally don't need these): SQL Server JDBC Jar files (the JDBC driver for older versions of Java)

Assigned date: Feb 19, 2019

Due date: March 4, 2019, at 11:59pm.

Warning: This can potentially be a long assignment. Do not wait until the last minute to start!

What to turn in:

Customer database schema in `setup.sql`, your completed version of the `Query.java`, and the test cases that you created with a descriptive name for each case.

You do not need to turn in your `dbconn.properties` file or any other starter files. We will be testing your implementations on the CSE lab (attu) machines. So make sure your submission works there before turning it in.

Assignment Details

Read this whole document before starting this project. There is a lot of valuable information here, including the Final Comments section at the bottom.

Congratulations, you are opening your own flight booking service!

You have access to a flights database. You will provide a service that lets users search through the database and book flights. (sounds familiar after HW3?)

In this homework, you have two main tasks. The first is to design a database of your customers and the flights they book, based on the schema that you have been working with in HW3. The second task is to complete a working prototype of your flight booking application that connects to the database then allows customers to use a command-line interface to search, book, and cancel flights. We have already provided code for a complete UI and partial back end; you will implement the rest of the back end. In real-life, you would develop a web-based interface instead of a command-line interface, but we use a command-line interface to simplify this homework.

For this homework you can use any of the classes from the Java 8 standard JDK as that is the supported platform on the CSE machines. If you like to use any external libraries beyond the JDK and those provided inside the `lib` folder, please check with the staff first.

Task 0: Running the starter code (0 points)

Your system will be a Java application. Download the starter code provided; you will see the following files:

- `FlightService.java`: the command-line interface to your flight-booking service. Calls into `Query.java` to run customer transactions. You do not need to change this file.
- `Query.java`: code to run customer transactions against your database, such as booking flights and reviewing reservations. You will need to change this file.
- `dbconn.properties`: a file containing settings to connect to your database. You need to edit it before running the starter code.
- `sqljdbc4.jar`: the JDBC to SQL Server driver. This tells Java how to connect to a SQL Azure database server, and needs to be in your `CLASSPATH` (see below).
- `hamcrest-core-1.3.jar`, `junit-4.12.jar`: these are libraries for running the test cases.
- `runTest.sh`: a simple bash script that compiles your code and runs test cases using the JUnit framework.
- `cases/*`: a number of sample test cases to test your implementation.

To run the starter code, you can simply run

```
javac -cp "lib/*" *.java
```

to compile your files in the starter-code directory.

Then, to launch the interactive interface, you can run it like this (with proper class path setting):

```
java -cp "lib/*":. FlightService
```

If you want to use Eclipse on your own, try the following:

1. Create a workspace with a project. We'll refer to this project as `hw5` from now on.
2. Copy the `.java` files into the project `src/` folder.
3. Copy the `.jar` and `.properties` files into the `hw5` folder. Make sure you add the necessary values to the `.properties` file.
4. Right click on the project, choose Build Path → Configure Build Path..., → Add Jars..., then expand `hw5` and select the `sqljdbc4.jar` file. Click Ok and then Ok again to close the menus.
5. Click the Run button.

You also need to access your Flights database on SQL Azure from HW3. Modify `dbconn.properties` with your servername, username, and password for SQL Azure. This allows your java program to connect to the database on SQL Azure. In practice, it is not safe to store passwords openly in text files, but it is ok for this assignment since we won't be collect it. We are assuming that all queries search and book flights for July 2015, which is in the dataset you used in HW3. It's ok if your database contains more flight data if you like.

Now you should see the command-line prompt for your Flight service with a great UI:

```
*** Please enter one of the following commands ***
> create <username> <password> <initial amount>
> login <username> <password>
> search <origin city> <destination city> <direct> <date> <num
itineraries>
> book <itinerary id>
> pay <reservation id>
> reservations
> cancel <reservation id>
> quit
```

The search command works (sort of). Try typing:

```
search "Seattle WA" "Boston MA" 1 14 10
```

This asks the system to list the top 10 direct flights from Seattle to Boston on July 14th 2015 ordered by increasing flight time, which is the sum of the `actual_time` values for all flights in the itinerary. You should see a list of flights between the origin and destination city for your desired data. The starter code only lists direct flights.

Data Model

The flight service system consists of the following logical entities:

- **Flights**: modeled the same way as HW3. You can assume that the Flights table will be created and populated for you before your setup.sql runs. Feel free to ALTER Flights if you need to. More precisely, we provide the SQL declaration for Flights commented out in `setup.sql` that you can use as a reference. Notice that the Months, Weekdays, and Carriers are no longer needed for this assignment, so those ids are no longer foreign keys.
- **Users**: each user has a username (`varchar`), password (`varchar`), and balance in their account (`double`). All usernames should be unique in the system. Each user can have any number of reservations. There are no restriction on passwords and you can just store them in plain text.
- **Itineraries**: each itinerary consists of a number of flights (up to 2). These are returned by the search command.
- **Reservations**: each reservation consists of a number of flights (up to 2). Each reservation can either be paid or unpaid and has a unique ID.

It is up to you which of these entities you want to store persistently and what the physical design should be.

For this assignment we will assume the following table exists to store the flights from HW3:

```
FLIGHTS (fid int, year int, month_id int, day_of_month int,
         day_of_week_id int, carrier_id varchar(3), flight_num int,
         origin_city varchar(34), origin_state varchar(47),
         dest_city varchar(34), dest_state varchar(46),
         departure_delay double, taxi_out double, arrival_delay double,
```

```
            canceled int, actual_time double, distance double, capacity int,
            price double)
```

You will need to create any other tables / indexes you need for this assignment in `setup.sql` (see below).

Requirements

The following are the functional specifications for the flight service system, to be implemented in `Query.java` (see code for full specification as to what error message to return, etc):

- **create** takes in a new username, password, and initial account balance as input. It creates a new account with the initial balance. It should return error if negative, or if the username already exists. There are no restrictions on password. You can assume that we won't be testing with any username / password that are longer than 20 characters.

- **login** takes in a username and password, and checks that the user exists in the database and that the password matches.

- **search** takes as input an origin city, a destination city, a boolean value to indicate if the user wants to search only for direct flights or if one stop is ok, the date, and the number of itineraries to be returned. For one hop flights, different carriers can be used for the flights.

  As you recall from HW3, we only have flights for July 2015 and July 2005, so the date should simply be an integer indicating which day in July 2005 and July 2015 the user wants to book. Note that the starter code ignores the month and year entirely.

  The command should return a list of itineraries sorted by ascending total `actual_time`. **If two itineraries have the same duration, it should be sorted by ascending order of the flight IDs.** The number of itineraries parameter controls how many itineraries should be returned. If the user requests `n` itineraries to be returned, there are a number of possibilities:

  - `direct=1`: return up to `n` direct itineraries, or fewer if fewer than `n` direct itineraries exists.
  - `direct=0`: return up to `n` direct and indirect itineraries. If there are `k` direct itineraries (where `k < n`), then return the `k` direct itineraries first, and then return up to `(n-k)` indirect itineraries with the shortest flight time, or fewer if fewer than `(n-k)` indirect itineraries exists.

  For the purpose of this assignment, an indirect itinerary means first going through another city before reaching the destination. The first and second flight must be on the same date (i.e., Flight 1 that arrives at Seattle at 11:59pm on day 1 and flight 2 that departs from Seattle at 12:01am on day 2 will be considered as two separate itineraries).

  In all cases the returned results should be sorted on `actual_time`. Return only flights that are not canceled, ignoring the capacity and number of seats available.

  Example of a direct flight from Seattle to Boston (actual itinerary numbers might differ):

  ```
  Itinerary 0: 1 flight(s), 297.0 minutes
  ID: 60454 Date: 2005-7-1 Carrier: AS Number: 24 Origin: Seattle WA
  Dest: Boston MA Duration: 297.0 Capacity: 14 Price: 140.59
  ```

Example of indirect flights from Seattle to Boston:

```
Itinerary 0: 2 flight(s), 317.22 minutes
ID: 704749 Date: 2015-7-10 Carrier: AS Number: 16 Origin: Seattle WA
Dest: Orlando FL Duration: 159.22 Capacity: 10 Price: 494.23
ID: 726309 Date: 2015-7-10 Carrier: B6 Number: 152 Origin: Orlando FL
Dest: Boston MA Duration: 158.0 Capacity: 0 Price: 104.91
Itinerary 1: 2 flight(s), 317.22 minutes
ID: 704749 Date: 2015-7-10 Carrier: AS Number: 16 Origin: Seattle WA
Dest: Orlando FL Duration: 159.22 Capacity: 10 Price: 494.23
ID: 726464 Date: 2015-7-10 Carrier: B6 Number: 452 Origin: Orlando FL
Dest: Boston MA Duration: 158.0 Capacity: 7 Price: 760.91
```

Note that for multi-hop flights, the results are printed in the order of the itinerary, starting from the flight leaving the origin and ending with the flight arriving at the destination.

The returned itineraries should start from `0` and increase by `1` up to `n` as shown above. If no itineraries match the search query, the system should return an informative error message (see `Query.java` for actual text).

The user need not be logged in to search for flights.

All flights in an indirect itinerary should have the same itinerary ID. In other words, the user should only need to book once with the itinerary ID for direct or indirect trips.

- **book** lets a user book an itinerary by providing the itinerary number as returned by a previous search. The user must be logged in to book an itinerary, and must enter a valid itinerary id that was returned in the last search that was performed ***within the same login session***. Make sure you make the corresponding changes to the tables in case of a successful booking. Once the user logs out (by quitting the application), logs in (if they previously were not logged in), or performs another search within the same login session, then all previously returned itineraries are invalidated.

  If booking is successful, then assign a new reservation ID to the booked itinerary. Note that 1) each reservation can contain up to 2 flights (in the case of indirect flights), and 2) each reservation should have a unique ID that incrementally increases by 1 for each successful booking.

- **pay** allows a user to pay for an existing reservation. It first checks whether the user has enough money to pay for all the flights in the given reservation. If successful, it updates the reservation to be paid.

- **reservations** lists all reservations for the user. Each reservation must have ***a unique identifier (which is different for each itinerary) in the entire system***, starting from 1 and increasing by 1 after a reservation has been made.

  There are many ways to implement this. One possibility is to define a "ID" table that stores the next ID to use, and update it each time when a new reservation is made successfully. Make sure you check the case where two different users try to book two different itineraries at the same time!

The user must be logged in to view reservations. The itineraries should be displayed using similar format as that used to display the search results, and they should be shown in increasing order of reservation ID under that username.

- **cancel** lets a user to cancel an existing reservation. The user must be logged in to cancel reservations and must provide a valid reservation ID. Make sure you make the corresponding changes to the tables in case of a successful cancellation (e.g., if a reservation is already paid, then the customer should be refunded).
- **quit** leaves the interactive system and logs out the current user (if logged in).

Refer to the Javadoc in `Query.java` for full specification and the expected responses of the commands above.

Make sure your code produces outputs in the same formats as prescribed! (see test cases)

Task 1: Customer database design (6 points)

Your first task is to design and add tables to your flights database. You should decide on the physical layout given the logical data model described above. You can add other tables to your database as well.

What to turn in: a single text file called `setup.sql` with `CREATE TABLE` and any `INSERT` statements (and optionally any `CREATE INDEX` statements) needed to implement the logical data model above. We will test your implementation with the flights table populated with HW3 data using the schema above, and then running your `setup.sql`. So make sure your file is runnable on SQL Azure through SQL Server Management Studio.

Write a separate script file with `DROP TABLE` or `DELETE FROM` statements; it's useful to run it whenever you find a bug in your schema or data (don't turn in this file).

Task 2: Java customer application (80 points)

Your second task is to write the Java application that your customers will use, by completing the starter code. You need to modify only `Query.java`. You do not need to modify `FlightService.java` as we will test your homework by running a grader script. If you do make modifications, make sure we can still test your `Query.java` on the original `FlightService.java`.

What to turn in: `Query.java`

When your application starts, it should show the above menu of options to the user. Your task is to implement the functionality behind these options.

Be sure to use SQL transactions when appropriate. The same user can log in multiple times from different terminals, concurrently. Different users can also use the same application at the same time from different terminals, and your application should not return inconsistent results (e.g., allowing a user to book an already full flight).

Task 2A: Stop SQL Injection

Consider the search function. Type this at the prompt:

```
search "Seattle WA" "Boston MA' and actual_time > 300 and
dest_city='Boston MA" 1 14 10
```

Whoa! You get only flights longer than 300 min. Now type this:

```
search "Seattle WA" "Boston MA'; create table Foo(a int); SELECT year,
month_id,day_of_month,carrier_id,flight_num,origin_city,actual_time from
Flights where origin_city = 'Seattle WA" 1 14 10
```

Check that this statement actually did successfully create a new table (hint: do not try this with `DELETE FROM Flights`). Imagine if it did other things instead like drop tables or look up the list of customers and their passwords. This is called SQL injection: hackers like to do it on Website interfaces to databases. Implement your own search function instead, and comment out the first line in `transaction_search` that calls `transaction_search_unsafe` in `Query.java` to call the safe version instead that you will implement by using `PreparedStatements`.

What to turn in: You will turn in your safe implementation as part of `Query.java`.

Task 2B: Transaction management

You must use SQL transactions to guarantee ACID properties: you must set isolation level for your `Connection`, define begin- and end-transaction statements, and insert them in appropriate places in `Query.java`. In particular, you must ensure that the following constraints are always satisfied, even if multiple instances of your application talk to the database at the same time.

C1. Each flight should have a maximum capacity that must not be exceeded. Each flight's capacity is stored in the Flights table as in HW3, and you should have records as to how many seats remain on each flight based on the reservations.

C2. A customer may have at most one reservation on any given day, but they can be on more than 1 flight on the same day.

You must use transactions correctly such that race conditions introduced by concurrent execution cannot lead to an inconsistent state of the database. For example, multiple customers may try to book the same flight at the same time. Your properly designed transactions should prevent that.

Design transactions correctly. Avoid including user interaction inside a SQL transaction: that is, don't begin a transaction then wait for the user to decide what she wants to do (why?). The rule of thumb is that transactions need to be as short as possible, but not shorter.

When one uses a DBMS, recall that by default *each statement executes in its own transaction*. As discussed in lecture, to group multiple statements into a transaction, we use

```
BEGIN TRANSACTION
....
COMMIT or ROLLBACK
```

This is the same when executing transactions from Java, by default each SQL statement will be executed as its own transaction. To group multiple statements into one transaction in java, you can do one of three things:

Approach 1:

We provide you with three helper methods. So before your first statement in the transaction, simply execute:

```
beginTransaction();
```

When you are done with the transaction, then call:

```
commitTransaction();
```

OR

```
rollbackTransaction();
```

Approach 2:

Execute the SQL code for `START TRANSACTION` and friends directly, using the SQL code we have provided in the starter code (also check out SQL Azure's transactions documentation):

```
// When you start the database up
Connection conn = [...]
conn.setAutoCommit(true); // This is the default setting, actually
conn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);

// In each operation that is to be a multi-statement SQL transaction:
conn.setAutoCommit(false);
// You MUST do this in order to tell JDBC that you are starting a
// multi-statement transaction

beginTransactionStatement.executeUpdate();

// ... execute updates and queries.

commitTransactionStatement.executeUpdate();
// OR
rollbackTransactionStatement.executeUpdate();

conn.setAutoCommit(true);
// To make sure that future statements execute as their own transactions.
```

Approach 3:

```
// When you start the database up
Connection conn = [...]
conn.setAutoCommit(true); // This is the default setting, actually
conn.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);

// In each operation that is to be a multi-statement SQL transaction:

conn.setAutoCommit(false);
// ... execute updates and queries.
conn.commit();
// OR
conn.rollback();
conn.setAutoCommit(true);
```

When auto-commit is set to true, each statement executes in its own transaction. With auto-commit set to false, you can execute many statements within a single transaction. By default, on any new connection to a DB auto-commit is set to true.

Task 2C: Implement `clearTables`

Implement this method in `Query.java` to clear the contents of any tables you have created for this assignment (e.g., reservations). However, do not drop any of them and do not delete the contents or drop the `flights` table .

After calling this method the database should be in the same state as the beginning, i.e., with the flights table populated and `setup.sql` called. This method is for running the test harness where each test case is assumed to start with a clean database. You will see how this works after running the test harness.

Task 3: Write test cases (14 points)

To test that your transactions work correctly, we have provided a test harness using the JUnit framework. Our test harness will compile your code and run the test cases in the folder you provided. To run the harness, execute:

```
runTests.sh <folder to your java source files> <output folder that you
want the compiled class files to be stored> <folder that contains the test
cases>
```

For instance, if you are currently in the folder where you downloaded the starter code, then run

```
runTests.sh . tmp cases
```

This first compiles `*.java` in the input folder you specified (i.e., `.`), deletes and recreates the output folder `./tmp` (make sure you have nothing important in there as it will be erased!!), and then run the test cases in the `cases` directory.

An example run should look like this (assuming only one test case present):

```
$ ./runTests.sh . tmp cases

compiling from  .
added manifest
adding: FlightService.class(in = 4609) (out= 2478)(deflated 46%)
...
...
JUnit version 4.12
running cases from: cases

running setup
.running test: <folder name>/book_2UsersSameFlight.txt
passed

Time: 8.503

OK (1 test)
```

Run this on the starter code and you will see what the failed test cases print out. For every test case it will either print pass or fail, and for all failed cases it will dump out what the implementation returned, and you can compare it with the expected output in the corresponding case file.

Each test case file is of the following format:

```
[command 1]
[command 2]
...
*
[expected output line 1]
[expected output line 2]
...
*
# everything following '#' is a comment on the same line
```

The `*` separates between commands and the expected output. To test with multiple concurrent users, simply add more `[command...] * [expected output...]` pairs to the file. Each user is expected to start concurrently in the beginning. If there are multiple output possibilities due to transactional behavior, then separate each group of expected output with `|`. See `book_2UsersSameFlight.txt` for an example.

Your task is to write at least 1 test case for each of the 7 commands (you don't need to test `quit`). Separate each test case in its own file and name it `<command name>_<some descriptive name for the test case>.txt` and turn them in. It's fine to turn in test cases for erroneous conditions (e.g., booking on a full flight, logging in with a non-existent username), but at least some of the test cases should do something useful.

Grading and final comments

- For grading purposes, you can assume that we will only issue one of the commands above and without spelling errors.
- For Task 2, we will grade your implementation by running the test harness using a number of staff test cases, and assigning points based on how many test cases your implementation passes. Just like Make sure you adhere to the expected output format as specified in `Query.java`. We won't be able to give you points even if all you miss was a `\n`!
- We will be grading your implementations on the CSE linux machines. So make sure that your implementation compiles and runs there. Any implementation that doesn't even compile will receive very few, if any, points.
- The starter code is designed to give you a gentle introduction to embedding SQL into Java. Start by running the starter code, examine it and make sure you understand the part that works. You will need to create new tables and slowly add code to the application.
- The completed project has multiple simple SQL queries embedded in the Java code. Some queries are parameterized: a parameter is a constant that is known only at runtime, and therefore appears as a `?` in the SQL code in Java: you already have examples in the starter code.
- We won't use recovery in this homework. Instead you will rely on the script file that you need to prepare for Task 1 to delete records. This file should contain all the `CREATE TABLE` and `INSERT` statements that are needed to start your project.
- Make sure you do all error handling in Query.java since we will be testing your code using `FlightService.java` that is included in the starter code.

Submission Instructions

Add your Java code and `setup.sql` that you created