CSE 344 Homework 5: Database Application and Transaction Management

Objectives: To gain 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
- Maven (if using OSX, we recommend using Homebrew and installing with `brew install maven`, if on Windows you may find this installation guide helpful)
- Prepared Statements Java Doc
- Prepared Statements Example
- starter code files

Assigned date: November 5, 2020

Milestone 1 Due date:

~~Monday, November 16, 2020, at 11 pm PDT. **NO LATE DAYS may be used for milestone 1.**~~

Friday, November 20, 2020, at 11 pm PDT. **NO LATE DAYS may be used for milestone 1.**

Milestone 2 Due date:

~~Monday, November 23, 2020, at 11 pm PDT. **You may use late days for milestone 2.**~~

Saturday, November 28, 2020, at 11 pm PDT. **You may use late days for milestone 2.**

WARNING: This homework requires writing a non-trivial amount of Java code (our solution is about 800 lines, including the starter code) and test cases. It should take SIGNIFICANTLY more time than your previous 344 assignments. START EARLY!!!

Milestone 1 makes good progress towards the goal but is less than half of the work. We highly recommend getting milestone 1 done early and starting milestone 2 before the 20th. Don't put off milestone 2!

Assignment Details

Download starter code.

Read through this whole section before starting this project. There is a lot of valuable information here, and some implementation details depend on others.

Congratulations, you are opening your own flight booking service!

In this homework, you have two main tasks:

- Design a database of your customers and the flights they book
- Complete a working prototype of your flight booking application that connects to the database (in Azure) then allows customers to use a CLI (command line interface) to search, book, cancel, etc. flights

You will also be writing a few test cases and explaining your design in a short writeup. We have already provided code for a UI (FlightService.java) and partial backend (Query.java). For this homework, your task is to implement the rest of the backend. In real life, you would develop a web-based interface instead of a CLI, but we use a CLI to simplify this homework.

For this lab, you can use any of the classes from the Java 11 standard JDK.

Connect your application to your database

You will need to access your Flights database on SQL Azure from HW3. Alternatively, you may create a new database and use the HW3 specification for importing Flights data.

Configure your JDBC Connection

You need to configure the appropriate information to connect Query.java to your database.

In the top level directory, **create a file named dbconn.properties** with the following contents:

```
# Database connection settings

# TODO: Enter the server URL.
flightapp.server_url = SERVER_URL

# TODO: Enter your database name.
flightapp.database_name = DATABASE_NAME

# TODO: Enter the admin username of your server.
flightapp.username = USERNAME

# TODO: Add your admin password.
flightapp.password = PASSWORD
```

Check your Azure server and fill in the connection details:

- The server URL will be of the form [your_server_name].database.windows.net
- The database name can be found at the top of the page when you open your database in the Azure console
- The username and password are the same credentials you use to login to your database/server when you open the query editor in the Azure console
- If the connection isn't working for some reason, try using the fully qualified username: flightapp.username = USER_NAME@SERVER_NAME

Build the application

Make sure your application can run by entering the following commands in the directory of the starter code and pom.xml file. This first command will package the application files and any dependencies into a single .jar file:

```
$ mvn clean compile assembly:single
```

This second command will run the main method from FlightService.java, the interface logic for what you will implement in Query.java:

```
$ java -jar target/FlightApp-1.0-jar-with-dependencies.jar
```

If you want to run directly without creating the jar, you can run:

```
$ mvn compile exec:java
```

If you get our UI below, you are good to go for the rest of the lab!

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

Data Model

The flight service system consists of the following logical entities. These entities are *not necessarily database tables*. It is up to you to decide what entities to store persistently and

create a physical schema design that has the ability to run the operations below, which make use of these entities.

- **Flights / Carriers / Months / Weekdays**: modeled the same way as HW3.
  For this application, we have very limited functionality so you shouldn't need to modify the schema from HW3 nor add any new table to reason about the data.
- **Users**: A user has a username (varchar), password (varbinary), and balance (int) in their account. All usernames should be unique in the system. Each user can have any number of reservations. Usernames are case insensitive (this is the default for SQL Server). Since we are salting and hashing our passwords through the Java application, passwords are case sensitive. You can assume that all usernames and passwords have at most 20 characters.
- **Itineraries**: An itinerary is either a direct flight (consisting of one flight: origin --> destination) or a one-hop flight (consisting of two flights: origin --> stopover city, stopover city --> destination). Itineraries are returned by the search command.
- **Reservations**: A booking for an itinerary, which may consist of one (direct) or two (one-hop) flights. Each reservation can either be paid or unpaid, cancelled or not, and has a unique ID.

You create these and any other Tables (and indexes) that are needed for this assignment in createTables.sql (which is discussed in more detail below).

Requirements

The following are the functional specifications for the flight service system, to be implemented in Query.java The methods you need to provide are indicated in the starter code, where they exist as stubs, skeletons that you will fill out as you develop your implementation. Refer to Query.java for the complete specification, including what conditions to handle and what error messages to return, etc.

- **create** takes in a new username, password, and initial account balance as input. It creates a new user account with the initial balance. Create() should return an error if it is passed an initial balance that is a negative dollar amount, or if the username already exists. When validating Usernames, please ensure that they are not case-sensitive. In other words, "UserId1", "USERID1", and "userid1" all map to the same User ID. You can assume that all usernames and passwords have at most 20 characters. We will store the salted password hash and the salt itself to avoid storing passwords in plain text. Use the following code snippet to as a template for computing the hash given a password string:

```
// Generate a random cryptographic salt
SecureRandom random = new SecureRandom();
byte[] salt = new byte[16];
random.nextBytes(salt);
```

```
// Specify the hash parameters
KeySpec spec = new PBEKeySpec(password.toCharArray(), salt, HASH_STRENGTH,
KEY_LENGTH);

// Generate the hash
SecretKeyFactory factory = null;
byte[] hash = null;
try {
  factory = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1");
  hash = factory.generateSecret(spec).getEncoded();
} catch (NoSuchAlgorithmException | InvalidKeySpecException ex) {
  throw new IllegalStateException();
}
```

- **login** takes in a username and password, and checks that the user exists in the database and that the password matches. To compute the hash, adapt the above code. Within a single session (that is, a single instance of your program), only one user should be logged in. A good practice is for every test case to begin with a login request. Make sure you log the User out when the program terminates. To keep things simple, you can track the login status of a User using a local variable in your program. You should not need to track a user's login status inside the database. If a second login attempt is made, please return "User already logged in".
- **search** takes as input an origin city (string), a destination city (string), a flag for only direct flights or not (0 or 1), the date (int), and the maximum number of itineraries to be returned (int). For the date, we only need the day of the month, since our dataset comes from July 2015. Return only flights that are not canceled, ignoring the capacity and number of seats available. If the user requests n itineraries to be returned, there are a number of possibilities:
    - direct=1: return up to n direct itineraries
    - direct=0: return up to n direct itineraries. If there are only k direct itineraries (where k < n), then return the k direct itineraries and up to (n-k) of the shortest indirect itineraries with the flight times. For one-hop flights, different carriers can be used for the flights. For the purpose of this assignment, an indirect itinerary means the first and second flight only must be on the same date (i.e., if flight 1 runs on the 3rd day of July, flight 2 runs on the 4th day of July, then you can't put these two flights in the same itinerary as they are not on the same day).

  Sort your results. In all cases, the returned results should be primarily sorted on total actual_time (ascending). If a tie occurs, break that tie by the fid value. Use the first then

the second fid for tie-breaking.
Below is an example of a single direct flight from Seattle to Boston. Actual itinerary numbers might differ, notice that only the day is printed out since we assume all flights happen in July 2015:

```
Itinerary 0: 1 flight(s), 297 minutes
ID: 60454 Day: 1 Carrier: AS Number: 24 Origin: Seattle WA Dest: Boston MA
Duration: 297 Capacity: 14 Price: 140
```

Below is an example of two indirect flights from Seattle to Boston:

```
Itinerary 0: 2 flight(s), 317 minutes
ID: 704749 Day: 10 Carrier: AS Number: 16 Origin: Seattle WA Dest: Orlando
FL Duration: 159 Capacity: 10 Price: 494
ID: 726309 Day: 10 Carrier: B6 Number: 152 Origin: Orlando FL Dest: Boston
MA Duration: 158 Capacity: 0 Price: 104
Itinerary 1: 2 flight(s), 317 minutes
ID: 704749 Day: 10 Carrier: AS Number: 16 Origin: Seattle WA Dest: Orlando
FL Duration: 159 Capacity: 10 Price: 494
ID: 726464 Day: 10 Carrier: B6 Number: 452 Origin: Orlando FL Dest: Boston
MA Duration: 158 Capacity: 7 Price: 760
```

Note that for one-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 the actual text.
The user need not be logged in to search for flights.
All flights in an indirect itinerary should be under 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 asuccessful 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 and cannot be booked.
A user cannot book a flight if the flight's maximum capacity would 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.
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 unpaid 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 currently logged-in 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 each reservation is 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. Another way is to define an id field as a Primary Key using the Identity type built into SQL Server, which allows SQL Server to generate unique values automatically for the Key field every time a new row is inserted into the Table. Your program won't know what the value of the Identity is until your INSERT statement has executed, at which time you can run a simple SELECT id FROM table WHERE ... statement to retrieve it.
  The user must be logged in to view reservations. The itineraries should be displayed using a similar format as that used to display the search results, and they should be shown in increasing order of reservation ID under that username. Cancelled reservations should not be displayed.
- **cancel(extra credit)** lets a user cancel an existing uncanceled 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.

CAUTION: Make sure your code produces outputs in the same formats as prescribed! (see test cases and Javadoc for what to expect)

Testing:

To test that your application works correctly, we have provided an automated testing harness using the JUnit framework. Our test harness will compile your code and run all the test cases in the provided cases/ folder. Automated testing is extremely helpful and, when used properly, should speed up your development. As you develop a new capability, develop an automated

test to verify that the capability works. Each time you add a new capability, make sure you haven't broken anything that previously worked by running the existing tests against the augmented solution.

To run the harness, execute in the project directory:

```
$ mvn test
```

If you want to run a single test file or run files from a different directory (recursively), you can run the following command:

```
$ mvn test -Dtest.cases="folder_name_or_file_name_here"
```

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
```

While we've provided test cases for most of the methods, the testing we provide is partial (although significant). It is **up to you** to implement your solutions so that they completely follow the provided specification.

For this homework, you're required to write test cases for each of the commands (you don't need to test quit). These custom test cases must test different cases than the tests we give you. Be creative! 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 a good practice to develop test cases for all erroneous conditions (e.g., booking on a full flight, logging in with a non-existent username) that your code is built to handle. The test cases you develop are one of your important project deliverables.

Milestone 1:

Database design

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

You should fill the provided createTables.sql file 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 HW2 data using the schema above, and then running your createTables.sql. So make sure your file is runnable on SQL Azure through the Azure query editor web interface.

NOTE: You may want to 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. You don't need to turn in anything for this.

Java customer application

Your second task is to start writing the Java application that your customers will use. To make your life easier, we've broken down this process into 5 different steps across the two milestones (see details below). You only need to modify Query.java. Do not modify FlightService.java.

Please use unqualified table names in all of your SQL queries (e.g. say SELECT * FROM Flights rather than SELECT * FROM [dbo].[Flights]) Otherwise, the grading scripts won't be able to run using your code.

We expect that you use Prepared Statements when you execute queries that include user input.

Since we will be looking at your code, it is important to make your code easy to read. Use descriptive variable names, for instance. Take a look at the Flight class we provide, a class that serves as a container for your flight data, as an example to follow. In methods like search, for example, you will see that you need to add a method similar to the 'toString' method that we provided in the Flight class. Use our toString as a style guide.

We have also provided a sample helper method checkFlightCapacity that uses a prepared statement.checkFlightCapacity is also intended as an example that outlines the way prepared statements should be used in this assignment (creating a constant SQL string, preparing it using the prepareStatements method, and then, ultimately, executing it).

Step 1: Implement clearTables

Implement the clearTables 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 modify the contents or drop the Flightstable. **Any attempt to modify the Flights table will result in a harsh penalty in your score of this homework.**

After calling this method the database should be in the same state as the beginning, i.e., with the flights table populated and createTables.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.

clearTables should not take more than a minute. Make sure your database schema is designed with this in mind.

Step 2: Implement create, login, and search

Implement the create, login and search commands in Query.java. Using mvn test, you should now pass our provided test cases that only involve these three commands.

After implementation of these, you should pass the following test:

```
mvn test -Dtest.cases="cases/no_transaction/search"
mvn test -Dtest.cases="cases/no_transaction/login"
mvn test -Dtest.cases="cases/no_transaction/create"
```

Or you can run as a single command:

```
mvn test
-Dtest.cases="cases/no_transaction/search:cases/no_transaction/login:cases/
no_transaction/create"
```

Step 3: Write some test cases

Write at least 1 test case for each of the three commands you just implemented. Follow the same format as our provided test cases. Include your written test files in the provided cases/mycases/ folder together with our provided test files.

Using `mvn test -Dtest.cases="cases/mycases"`, you should now also pass your newly created test cases.

What to turn in:

- Customer database schema in createTables.sql
- Your Query.java with the create, login and search commands implemented

- At least 3 custom test cases (one for each command) in the cases/mycases/ folder, with a descriptive name for each case

Turn in these 5 (or more) files to Gradescpe. The easiest way to do this is by copying all the necessary files to a single folder and selecting them all when submitting to Gradescope.

Grading:

This milestone is worth 50 points (25% of the total homework grade), based on two points:

- Whether your create, login, and search methods pass the provided test cases and,
- Your custom test cases both pass and test edge cases outside the original test suite.

Milestone 2:

Step 4: Implement book, pay, reservations, cancel (extra credit) , and add transactions!

Implement the book, pay , reservations and cancel commands in Query.java.

While implementing & trying out these commands, you'll notice that there are problems when multiple users try to use your service concurrently. To resolve this challenge, you will need to implement transactions that ensure concurrent commands do not conflict.

Think carefully as to *which* commands need transaction handling. Do the create, login and search commands need transaction handling? Why or why not?

```
mvn test -Dtest.cases="cases/no_transaction/search"
mvn test -Dtest.cases="cases/no_transaction/pay"
mvn test -Dtest.cases="cases/no_transaction/cancel"
```

Or you can run all non transaction test:

```
mvn test -Dtest.cases="cases/no_transaction/"
```

Transaction management

You must use SQL transactions to guarantee ACID properties: we have set the isolation level for your Connection, and you need to define begin-transaction 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. (i.e., a customer can have one reservation on a given day that includes two flights, because the reservation is for a one-hop itinerary).

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 need to use setAutoCommit and commit or rollback:

```
// 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

// ... execute updates and queries.

conn.commit();
// OR
conn.rollback();
```

```
conn.setAutoCommit(true);
// You MUST do this to make sure that future statements execute as their
own transactions.
```

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, any new connection to a DB auto-commit is set to true.

Your executeQuery calls will throw a SQLException when an error occurs (e.g., multiple customers try to book the same flight concurrently). Make sure you handle the SQLException appropriately. For instance, if a seat is still available but the execution failed due a temporary issue such as deadlock, the booking should eventually go through (even though you might need to retry due to SQLExceptions being thrown). If no seat is available, the booking should be rolled back, etc.

The total amount of code to add transaction handling is in fact small, but getting everything to work harmoniously may take some time. Debugging transactions can be a pain, but print statements are your friend!

At this point, you program should pass all the test cases you have provided when you execute mvn test

Step 5: Write more (transaction) test cases

Write at least 1 test case for each of the four commands you just implemented. Follow the same format as our provided test cases.

In addition, write at least 1 *parallel* test case for each of the 7 commands. By *parallel*, we mean concurrent users interfacing with your database, with each user in a seperate application instance.

Remember that each test case file is in 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, for instance:

```
[command 1 for user1]
[command 2 for user1]
...
*
[expected output line 1 for user1]
[expected output line 2 for user1]
...
*
[command 1 for user2]
[command 2 for user2]
...
*
[expected output line 1 for user2]
[expected output line 2 for user2]
 ...
*
```

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.

Put your written test files in the cases/mycases/ folder.

Using `mvn test -Dtest.cases="cases"`, you should now also pass ALL the test cases in the cases folder - it will recursively run the provided test cases as well as your own.

Congratulations! You now finish the entire flight booking application and are ready to launch your flight booking business :)

Write down your design

Please describe and draw your database design as an ER diagram. This is so we can understand your implementation as close to what you were thinking. Explain your design choices in creating new tables. The diagram should include both tables you made and the original 4 flights tables as these together make up your database. Also, describe your thought process in deciding what needs to be persisted on the database and what can be implemented in-memory (not persisted on the database). Please be concise in your writeup (< 1 page).

Save this file as **writeup.pdf**

What to turn in:

- Customer database schema in createTables.sql
- Your fully-completed version of the Query.java
- At least 14 custom test cases (one normal & one parallel for each command) in the cases/mycases folder, with a descriptive name for each case
- A writeup.pdf

Grading:

- Milestone 1 check in (50 points)
- Customer database design (20 points)
- Java customer application (100 points)
- Written test cases (20 points)
- Writeup (10 points) = 200 points total
- 25 points for implementing cancel (Extra credit)

We will be testing your implementations using the home VM.

Submission Instructions

Once you are completed with each milestone, you will need to submit the relevant files to Gradescope. The best way to do this is to copy all relevant files to a single folder so that you can select all of them at once when submitting to Gradescope. Once submitted, always double check that you can see your files submitted correctly.

The files needed for Milestone 1 are:

- createTables.sql
- Query.java
- 3 or more custom test cases

The files needed for Milestone 2 are:

- createTables.sql (the same file or updated if you changed your database design)
- Query.java (fully implemented)
    - Whether or not you implemented ‘cancel’, you will still submit the same Query.java file
- 12 or more custom test cases (14 or more if extra credit is done)
    - 6 must be serial tests
    - 6 must be parallel tests
    - Extra credit:
        - 1 serial test for the ‘cancel’ method
        - 1 parallel test for the ‘cancel’ method
- writeup.pdf