Homework 5 | Database Application and Transaction Management

Updates made to the assignment after its release are highlighted in red.

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

Due dates:

- Milestone 0: Monday, Feb 7, 2022 11:59pm - no late days
- Milestone 1: Monday, Feb 14, 2022 11:59pm - no late days
- Milestone 2: Tuesday, Feb 22, 2022 11:59 pm - normal late days

Note: We do not allow late submissions for milestone 0 and milestone 1 to allow the staff enough turnaround timer to give feedback in a timely manner for you to use in the subsequent milestone.

Resources

For this assignment, you will need:

- 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 (*must be logged in using @cs account*)
- Prepared Statements Java Doc
- Prepared Statements Example (*must be logged in using @cs account*)
- Starter code (.zip format)
- Remote development over SSH

Introduction

Congratulations, you are opening a global airline management service!

In this homework, you have two main tasks:

- Design a database of airline companies, their customers.
- Complete a working prototype of your application that connects to the database (in Azure) and supports the functionality described below.

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.

✨✨🦄✨✨🦄✨✨🦄✨✨🦄✨✨🦄✨✨🦄✨✨🦄✨✨🦄✨✨🦄✨✨🦄✨✨🦄✨✨

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

Milestone 0 has a short deadline, to force you to start thinking early about designing the schema of your application.

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 M1 is due.

Milestone 2 requires more Java development. Don't put off milestone 2!

✨✨🦄✨✨🦄✨✨🦄✨✨🦄✨✨🦄✨✨🦄✨✨🦄✨✨🦄✨✨🦄✨✨🦄✨✨🦄✨✨

Setup

1. **Download the starter code**

2. **Connect your application to your database**
   You will need to access your Flights database from HW3. Alternatively, you may create a new database and use the HW3 specification for importing Flights data

3. **Configure your JDBC Connection**
   This allows Query.java to connect to your SQLServer on Azure.

   In the top level directory, **create a file named dbconn.properties** and copy-and-paste the following into it:

```
# 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
```

Next, populate the `.properties` file with your server's details:

- SERVER_URL will be of the form `[sqlserver_name].database.windows.net`. The server name can be found in the table of Azure resources when you first log in
- DATABASE_NAME is the SQLServer name, and is found in the table of Azure resources when you first log in
- 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@DATABASE_NAME

Your dbconn.properties file should look something like this:

```
# Database connection settings

flightapp.server_url = hctang.database.windows.net

flightapp.database_name = hctang-344-hw

flightapp.username = hctang

flightapp.password = obvsThisIsNotMyPassword
```

4. **Build the application**
   Package the application files and any dependencies into a single .jar file:

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

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 first creating a jar, you can run:

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

If either of those starts the UI below, you are good to go!

```
*** 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
```

Homework Requirements

Data Model

The airline management 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 to create a physical schema design that has the ability to run the operations below.

- **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 HW3 dataset.

- **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). However, 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, canceled 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.

Functional Specification

The flight service system is implemented in Query.java The methods you need to provide are indicated in the starter code, which you will fill out as you develop your implementation. There are two different functions of the system.

Customer Facing Function

This function interacts only with the customers. 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 private methods “getSalt” and “getHash” created to help you with salting and hashing:

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

private byte[] getHash(String password, byte[] 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();
     return hash;
    } catch (NoSuchAlgorithmException | InvalidKeySpecException
```

```
    ex) {
      throw new IllegalStateException();
    }
}
```

- **login** accepts a username and password; it checks that the user exists in the database and that the password matches. You can use the getHash private method to help you with this. Within a single session (that is, a single instance of your program), only one user should be logged in; 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; notably, you *should not track* a user's login status inside the database. If a second login attempt is made within the current session, please return "User already logged in".

  A good practice is for every test case to begin with a login request.

- **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. Assuming the user requests n itineraries to be returned, there are a number of possibilities:
  - *when direct=1*: return up to n direct itineraries
  - *when 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 along with the flight times.

  For itineraries that have one or more stops (an “indirect itinerary”), different carriers can be used for each leg. The first and second flight only must be on the same date (eg, if flight1 runs on July 3 and flight2 runs on July 4th, then you can't put these two flights in the same itinerary).

  Sort your results on total actual_time (ascending). If a tie occurs, break that tie by the fid value. For indirect itineraries, 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 and flights 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 itineraries 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
```

Notice that for indirect itineraries, the results are printed in the order of the flights taken (ie, starting with the flight leaving the origin and ending with the flight arriving at the destination).

The returned itineraries IDs should start from 0 and increase by 1 up to n as shown above. 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, regardless of whether they are flying a direct or indirect itinerary.

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.

An approach for searching for n indirect itineraries:

- Query1: Select n direct flights, sort by duration, fid
- If query1 returns k flights (and k < n), do query2: select (n-k) indirect flight, sort by total_duration, fid1, fid2.
- Using Java code, combine the result of query1 and query2 in java; sort the combination by duration, fid1, fid2. Note that the results from query 1 do not have fid2; you may have to use some ✨programming magic✨ :)
    - How to sort? Comparator/Comparable is the go-to option. You may need to create a class to sort itineraries (or not, it is your program after all).

- **book** lets a user reserve an itinerary using its itinerary number (as returned by a previous search). The user must be logged in to book an itinerary, and they must enter a valid itinerary id returned from the *most recent search* performed *within the same login session*. So it is important to think about how we can save the results from the latest search query to be used in this booking function. 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 the booking is successful, assign a new reservation ID to the booked itinerary. Recall that each reservation can contain up to 2 flights (in the case of indirect flights).

- **pay** allows a user to pay for an existing-but-unpaid reservation. It should first verify the user has enough money to pay for all the flights in the given reservation; if so, it updates the reservation to be paid.

- **reservations** lists the currently logged-in user's reservations. 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. Canceled reservations, if implemented, should not be displayed.

  As noted above, each reservation must have a numeric identifier *which is different for each itinerary in the entire system*. These identifiers should start from 1 and increase by 1 after each successful reservation. There are several ways to implement this:
  - Define a "ID" table that stores the next value to use, and update it each time a new reservation is made successfully.
  - Declare a column as having SQLServer's built-in Identity type, which tells SQLServer to automatically generate a unique value every time a new row is inserted. Since you do not specify a value for that column in your `INSERT` statement, your program won't know what its value is without running a simple `SELECT id FROM table WHERE ...` statement to retrieve it.

- **cancel (extra credit)** lets a user cancel an existing uncanceled reservation of this user. The user must be logged in to cancel reservations and must provide a valid reservation ID. Make sure you make *all* relevant changes to your tables (eg, if a reservation is already paid, 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 its output in the exact same format as described! (see test cases and Javadoc for what to expect). By this point, you should know what happens when the autograder doesn't see the output it's expecting.

❗🕴❗ ❗🕴❗ ❗🕴❗ ❗🕴❗ ❗🕴❗ ❗🕴❗ ❗🕴❗ ❗🕴❗ ❗🕴❗

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 feature, make sure to write more tests to cover this new feature. This allows you to know exactly where a newly-introduced bug can be found.

To run the test 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"
```

We have organized our testing code into ***test cases***. For every test case, it will either print pass or fail; for all failed cases, it will print out what the implementation returned, and you can compare it against the expected output. 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 incomplete. It is **up to you** to implement your solutions so that they completely adhere to the specification; "but it passed all the provided tests!" is no guarantee that your code will get full points.

Furthermore, you're required to write new test cases for each of the commands (except `quit`). Separate each test case in its own file and name it `<command name>_<some descriptive name for the test case>.txt`. 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, but you'll also want test cases for successful conditions as well. Be creative!

Transaction management

For the second milestone, you must use SQL transactions to guarantee ACID properties. We have set the isolation level for your Connection, but you will need to define begin-transaction and end-transaction statements and to insert them in appropriate places in Query.java. 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.

Furthermore, you must ensure that the following functional constraints are always satisfied, even if multiple instances of your application connect to the database at the same time:

> *C1:* Each flight has 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. For example, a customer can have one reservation on a given day that includes two flights, because the reservation is for an indirect itinerary.

Do not include 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?).

Recall that, by default, **each SQL statement executes in its own transaction**. As discussed in lecture, to group multiple statements into a transaction, we use the following SQL statements:

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

Executing transactions from Java has the same semantics: by default, each SQL statement will be executed as its own transaction. To group multiple statements into a single transaction in Java, you need to use `setAutoCommit()` and call either `commit()` or `rollback()`:

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

// Before each collection of SQL statements which form a single
// logical transaction.  This informs JDBC that you are starting
// a multi-statement transaction:
conn.setAutoCommit(false);

// ... execute your updates and queries ...

// Finally, decide what to do with your transaction and undo
// your transaction settings:
conn.commit();
// ~OR~
conn.rollback();

conn.setAutoCommit(true);  // future SQL stmts will execute as
                           // individual transactions
```

When auto-commit is set to true, each SQL statement executes in its own transaction; when auto-commit is set to false, you can execute multiple SQL 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 if an error occurs (eg, multiple customers try to book the same flight concurrently); ensure you handle it appropriately. For example, assume that a SQLException is thrown when a booking attempt failed:

- If a seat is still available due to a temporary failure such as deadlock, the booking should eventually go through (though you might need to retry).
- If no seat is available, the booking should be rolled back, etc.

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

Milestone 0: (Due Monday 2/7 11:59 PM - NO LATE SUBMISSIONS)

Database design

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

Fill the provided `createTables.sql` file with `CREATE TABLE`, `INSERT`, and optionally any `CREATE INDEX` statements needed to implement the logical data model. We will test your implementation with a `FLIGHTS` table populated with HW2 data and then running your `createTables.sql`, so ensure your file is runnable on SQL Azure through the Azure query editor web interface. You do not need to include statements to create tables already in your db (FLIGHTS, CARRIERS, WEEKDAYS, or MONTHS).

TIP: You may find it useful to write a separate .sql script file with `DROP TABLE` or `DELETE FROM` statements; this may come in handy if you find a bug in your schema or data. Please do not submit this.

M0 Submission

What to turn in:

- A .pdf file containing your hand-drawn or web-generated (draw.io) E/R diagram.
- createTables.sql file containing your tables

What we will be grading on (20 points):

- A reasonable schema and createTables file.

Before M1

The staff is targeting to give you feedback on your design no later than Wednesday 2/9 at 11:59 pm. This should give you ample time to rework your design with feedback, but this should not be a blocker to start implementing the skeleton and thinking about M1. A lot of the functionality in M1 can be done without a finalized schema and you may notice you switch your schema between every milestone as you experiment

Milestone 1: (DUE Monday 2/14 11:59 PM - NO LATE SUBMISSIONS)

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 both milestones. You only need to modify Query.java; do not modify FlightService.java.

We require that your application:

- **Use unqualified table names** in all of your SQL queries (e.g. `SELECT * FROM Flights` instead of `SELECT * FROM [dbo].[Flights]`). Doing otherwise will allow the grading scripts from being able to run using your code.
- Use Prepared Statements (refer to section and lecture if you are confused) when you execute queries that include user input.
    - We have provided a helper method `checkFlightCapacity()` which uses a prepared statement and demonstrates the way prepared statements should be used (ie, creating a constant SQL string, preparing it using the `prepareStatements()` method, and executing it).
- Write code that we can understand. For example, use descriptive variable names, well-factored methods, and follow a consistent style (eg, the `toString()` method that we provided in the Flight class).

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). After calling this method, the database should be in its initial state, ie, with the `FLIGHTS` table populated and `createTables.sql` called. This method is used for running the test harness, where each test case assumes it has a clean database.

Notably: do **NOT** drop any of your tables and do **NOT** modify the contents or drop the `FLIGHTS` table. Any attempt to modify the Flights table will result in a huge loss in points.

`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 the provided test cases which only involve these three commands. Specifically, you should pass:

```
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 all three cases using this a single command:

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

Note: Look for a helper method for hashing to assist you with create and login

Step 3: Write test cases

Write at least 1 new test case for each of the three commands you just implemented. Follow the same format as our provided test cases, and include your test files in the provided `cases/mycases/ folder` alongside our provided tests.

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

M1 Submission

For this milestone, you should submit these 5 (or more) files to Gradescope:

- Query.java with implemented **create**, **login** and **search** commands
    - Recall that we will not implement transaction handling until M2!
- At least 3 new test cases (one for each command) in the `cases/mycases/` folder, with a descriptive name for each case
- createTables.sql (your schema)

This milestone (30 points) is graded on:

- Whether you have completed the create, login and search commands
- Whether you wrote sufficient additional test cases

Milestone 2:

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

Implement the **book**, **pay**, and **reservations** commands in Query.java. For extra credit, you may also attempt the **cancel** command.

While implementing and trying out these commands, you'll notice that there are problems when multiple users try to use your service concurrently. To resolve this, you will need to implement transactions to 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?

We've created a few test cases which might illustrate some of the issues:

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

As before, you can run entire an entire directory's worth of tests:

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

In contrast to M1, these commands will require the addition of transaction handling. Once you have completed these commands *with correct transactions*, your program should pass all the test cases when you execute `mvn test`.

Step 5: Write More (transaction) Test Cases

Write at least 1 test case for each of the 3 commands you just implemented (possibly 4 commands, if you implemented extra credit). Follow the same format as our provided test cases.

Next, 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 commands and their expected output. To test with multiple concurrent users, 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 will start concurrently. If there are multiple possible outputs due to transactional behavior, separate each group of expected output with `|`. See `book_2UsersSameFlight.txt` for an example.

As before, put your written test files in the `cases/mycases/` folder. You should now pass ALL the test cases in the `cases/` folder when running `mvn test -Dtest.cases="cases"` - this command recursively runs our provided test cases as well as your own.

✨🥂✨ *Congratulations!* ✨🥂✨ You have completed the entire flight booking application and are ready to launch your new business :)

M2 Submission

For this milestone (100 points), you should submit these files to Gradescope:

- createTables.sql (your schema)
- Your fully-complete Query.java
    - You will submit the same Query.java file regardless of whether you attempted the extra credit
- At least 12 new test cases in the `cases/mycases/` folder
    - 6 must be serial tests, one for each command
    - 6 must be parallel tests, one for each command
    - If you attempted the extra credit, add 2 more test cases for the **cancel** method (ie, 1 serial + 1 parallel)

We will be testing your implementations using the home VM.