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: **Tue**, Nov 8 @ 10pm - **no late days!**
- Milestone 1: Thu, Nov 17 @ 10pm - normal late days (highly recommend not using!)
- Milestone 2: Thu, Dec 1 @ 10pm - normal late days

We do not allow late submissions for milestone 0 to allow the staff enough time to return feedback for you to use in the subsequent milestone. We permit milestone 1 to be submitted late; however, in our experience (21au), only 21% of students who took late days for M1 avoided taking late days for M2. In other words, lateness tends to beget more lateness, so ensure you've budgeted appropriately.

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, try this installation guide (*must be logged in using @cs account*)
- Starter code (.zip format)
  - Documentation on our test case format

Helpful guides/documents:

- Prepared Statements:
  - Official Java Doc
  - HW5-specific Example (*must be logged in using @cs account*)
- Remote development over SSH with VSCode

Introduction

Congratulations, you are opening a global airline management service!

In this homework, you have two main tasks:

- Design a database of airline flights, their customers, and their reservations
- Prototype the management service; it should connect to a live database (in Azure) and implement the functionality below

- ○ The prototype uses a command-line interface (CLI), but in real life you would probably develop a web-based interface

We have provided code for the CLI (FlightService.java) and a partial backend (Query.java + PasswordUtils.java). For this homework, your task is to implement the rest of the backend. You can use any of the classes from the Java 11 standard JDK.

✨✨🦄✨✨🦄✨✨🦄✨✨🦄✨✨🦄✨✨🦄✨✨🦄✨✨🦄✨✨🦄✨✨

WARNING: This homework requires writing a lot of Java code and test cases; our solution is about 1000 lines (including the starter code). It will take SIGNIFICANTLY more time than your previous 344 assignments, so START EARLY!!!

- Milestone 0's goal is to design your application's schema; this gives us a chance to provide feedback and prevent you from going down the wrong path
- Milestone 1 is less than half of the work. We highly recommend finishing M1 before it is due, and using the extra time to work on M2.
- Milestone 2 is the bulk of the work.

✨✨🦄✨✨🦄✨✨🦄✨✨🦄✨✨🦄✨✨🦄✨✨🦄✨✨🦄✨✨🦄✨✨

Setup

1. **Download the starter code (see link above)**

2. **Connect your application to your database**
   You will need your Flights database from HW3, or set up a fresh database in Azure.

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 fill it with:

```
# Database connection settings

flightapp.server_url = SERVER_URL
flightapp.database_name = DATABASE_NAME
flightapp.username = USERNAME
flightapp.password = PASSWORD
flightapp.tablename_suffix = UWNetID
```

   You should use the following details from your SQLServer on Azure:

   - ○ `SERVER_URL` will be of the form `[sqlserver_name].database.windows.net`.
     - ■ This can be found in the table of Azure resources when you first log in

- ○ `DATABASE_NAME` is the SQLServer name
  - ■ Same location as `SERVER_URL`
- ○ 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, use the fully qualified username: `flightapp.username = USER_NAME@DATABASE_NAME`
- ○ We hope you can figure out `UWNetID` ;)

When done, 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 = obvsThisIsNotMyRealPasswordexe
flightapp.tablename_suffix = hctang
```

4. **Build the application**

Package the application files and its dependencies into a single .jar file, then run the main method from FlightService.java.
(You must run these commands in the hw5 directory where the pom.xml file is located. Otherwise, you will run into an error that says “...there is no POM in this directory”)

```
$ mvn clean compile assembly:single
$ 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 two commands 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
> 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.

- **Flights / Carriers / Months / Weekdays**: modeled the same way as HW3. For this homework you should consider them to be "read-only".

- **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 SQLServer. 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 direct or indirect.

  A **direct itinerary** or **direct flight** consists of a single flight, from the origin to the destination. In contrast, an **indirect itinerary** (alternatively known as a **two-hop itinerary**) consists of two flights, from the origin to a stopover city and then from the stopover city to the destination. This system does not support itineraries with more than one stopover city.

- **Reservations**: A booking for an itinerary, which may consist of one or two flights (ie, direct or indirect). Each reservation can either be paid or unpaid and has a unique ID.

Once you decide which logical entities to persist in a table, you will create them using `createTables.sql` which is discussed in more detail in the M1 section below.

Application Logic

The bulk of your application's logic is implemented in Query.java and PasswordUtils.java. Each command in the application menu has a corresponding method that you will implement.

- **create** takes in a new username, password, and initial account balance as input and creates a new user account with that initial balance. create should return an error someone attempts to create an account with a negative balance or if the username is already taken.

  Usernames are not case-sensitive; in other words, "User1", "USER1", and "user1" are all equivalent. You can assume usernames and passwords have at most 20 characters.

We will store the salted password hash, as well as the salt itself, to avoid storing passwords in plain text. See PasswordUtils.java for more information. *Note that we will store both the salted password hash and the salt itself in the same field in our table.*

- **login** accepts a username and password; it checks that the user exists in the database and that the provided password matches the stored one. You can use PasswordUtils.java to help with this.

  Within a single session (that is, a single instance of your program), only one user should be logged in. To keep things simple, track the login status of a User using a local variable in your program; you *should not track* a user's login status inside the database.

- **search** takes as input an origin city (string), a destination city (string), a flag indicating whether the results should only consist of direct flights (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. For indirect itineraries, 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).

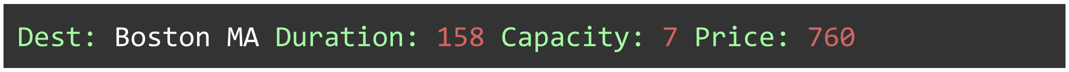
  Sort your results on total actual_time (ascending). If a tie occurs, break that tie by choosing the smaller fid value; for indirect itineraries, use the first then second fid for tie-breaking.

  Below is an example of a single direct itinerary from Seattle to Boston:

```
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 an indirect itinerary 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
```

The returned itinerary IDs should start from 0 and increase by exactly 1, 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 using a single 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, but these search results cannot be booked (see **book** for more details).

Lastly, your code should always prefer *returning* direct itineraries*, even if the direct itinerary is slower than an indirect one*. Assume a pair of cities has exactly two itineraries between them: *D* (a direct itinerary that takes 100m) and *I* (an indirect itinerary whose two flights take a total of 90m).

- If the user only requests direct itineraries, return *{D}*
- If the user requests 2+ itineraries, some which may be indirect, return *{I, D}* in that order
  - Note that *D* doesn't have a fid2; you may have to use some ✨programming magic✨ to sort these mixed results :)
- If the user requests 1 itinerary, some of which may be indirect, return *{D}* even though {I} would be faster.

- **book** lets a user reserve an itinerary using its itinerary number, as returned by the 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*. 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, and you should have records as to how many seats have already been reserved. If the booking is successful, assign a new reservation ID to the booked itinerary.

- **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 status.

- **reservations** lists the currently logged-in user's reservations. The reservations should be displayed using a format similar to the search results, and they should be shown in increasing order of reservation ID.

  As noted above, each reservation must have a numeric identifier *which is different for each reservation in the entire system*. 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.
  - Calculate the next reservation ID by counting the number of existing reservations or calculating the current maximum ID.
  - *Not recommended due to limitations in our testing framework:* ~~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 subsequent `SELECT id FROM table WHERE ...` statement to retrieve it.~~

- **quit** leaves the interactive system.

❗🕴❗ ❗🕴❗ ❗🕴❗ ❗🕴❗ ❗🕴❗ ❗🕴❗ ❗🕴❗ ❗🕴❗

CAUTION*:* Ensure your code produces its output in the exact same format as described! The autograder expects its output in our format.

❗🕴❗ ❗🕴❗ ❗🕴❗ ❗🕴❗ ❗🕴❗ ❗🕴❗ ❗🕴❗ ❗🕴❗

Testing

The test format is described in more detail in this companion document. You will be required to submit your own test cases for both M1 and M2.

Although we've provided some test cases, 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. 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 (M2 only)

For the second milestone, you must use SQL transactions to guarantee ACID properties; specifically, 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. Hint: 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
....
-- eg, UPDATE or DELETE FROM statements
....
COMMIT or ROLLBACK
```

Executing transactions from Java has the same semantics: by default, each SQL statement will be executed as its own transaction. This is configured using the auto-commit value: 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.

To group multiple statements into a single transaction in Java, you need to disable `setAutoCommit()` (which implicitly starts a transaction), execute your statements, and finish the transaction by either calling `commit()` or `rollback()`:

```
// ---------------------------------------------------------
// Surrounding each collection of SQL statements which
// constitute a single logical transaction:

// Disable the one-statement-per-transaction behavior:
conn.setAutoCommit(false);

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

// Decide what to do with your statements:
conn.commit();
// ~OR~
conn.rollback();

// Undo the changes to your transaction settings; future SQL
// statements will execute as individual transactions
conn.setAutoCommit(true);
```

`executeQuery()` and `executeUpdate()` throw SQLExceptions if an error occurs; determine if the error is transient (eg, deadlock) or permanent (eg, bad SQL syntax) and retry if appropriate.

The total amount of code to add transaction handling is quite small, but getting everything to work harmoniously may take some time.

Milestone 0: Database design

First, you will translate the logical data model into an ER diagram and then to physical tables.

Your E/R diagram should include flights, carriers, months, weekdays, users, reservations, and itineraries. You may elide some of flights' attributes to keep your diagram small, if you wish.

Next, fill in the provided `createTables.sql` file with the necessary `CREATE TABLE` (and optionally any `CREATE INDEX`) statements to implement your E/R diagram. *Recall that E/R diagrams may contain entities or relationships which become columns rather than a table.* Do not include statements for the tables already in your database (ie, `Flights`, `Carriers`, `Weekdays`, or `Months`). We will test your `createTables.sql` by running it in parallel with other student submissions on our Azure SQLServer. To prevent interference, we require that your table names be **suffixed with your UWNetID** (eg, `MyTable_hctang`).

M0 Submission

For this milestone, you should submit these 2 files to Gradescope:

- A .pdf file containing an E/R diagram
- createTables.sql

This milestone's goal is to provide feedback, so its point value is a small fraction of HW5's total points. Once you receive our comments, you do not need to resubmit a "fixed" E/R diagram.

Proceed Directly After M0

The staff is targeting to give you feedback on your design within 3d of your submission, with the goal of providing ample time to incorporate our feedback. However, do not let our feedback prevent you from thinking about and starting M1. A lot of M1's functionality can be implemented without a finalized schema; in fact, you may even notice yourself making schema changes *after* our feedback as you refine your understanding.

Milestone 1:

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 several steps spread across two milestones. You will only need to modify Query.java and PasswordUtils.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]`).
- Use Prepared Statements (refer to section and lecture if you are confused) when you execute queries that include user input.
    - Dynamically-generate SQL statements permit SQL injection attacks; don't do this
- Write code that we can understand.
    - For example, use descriptive variable names, well-factored methods, and follow a consistent style

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`). Notably: **do not drop** any of your tables, and **do not modify** the contents or drop the `Flights` table.

`clearTables()` should not require more than a minute to run. This method is used for running the test harness, where each test case assumes it has a clean database (ie, with the `FLIGHTS` table populated and `createTables.sql` called). An incorrect implementation will cause difficult-to-debug test failures.

Step 2: Implement create, login, and search

Implement the **create**, **login** and **search** commands in Query.java. `mvn test` should now pass the (non-transactional) test cases which only involve these three commands:

```
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 single command:
mvn test
-Dtest.cases="cases/no_transaction/search:cases/no_transaction/lo
gin:cases/no_transaction/create"

# As mentioned before, you may need to change the quotation
# marks if running this command on Windows.
```

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; you should copy our naming convention (ie,

~~[no_]transaction/~~<cmdname>_<description>.txt) as well as the test case format. Failure to match our naming convention will cause your tests to be miscategorized.

You may find the documentation on our test case format helpful.

M1 Submission

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

- Partially-complete Query.java with **create**, **login** and **search** commands
  - Recall that we will not implement transaction handling until M2!
- A fully complete PasswordUtils.java
- At least 3 new test cases (one for each command)
- createTables.sql
- hw51-writeup.txt containing:
  - *The answers to our 3 standard writeup questions ("What is one thing that you learned; one thing that surprised you; a question you still have after doing this assignment")*
  - **[optional]** how many hours you spent on M0+M1, plus the UWNetIDs of any collaborators

This milestone's goal is to ensure you are making progress, so its point value is a fraction of HW5's total points. We will merely verify compilation and execute a handful of tests; if you want a full check of your application, including our hidden tests, proceed to milestone 2.

Milestone 2:

Step 4: Implement book, pay, and reservations. Add transactions!

Implement the **book**, **pay**, and **reservations** commands in Query.java. At this point you have a fully-functional app! To verify, you can run entire an entire directory's worth of tests:

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

Unfortunately, you quickly notice problems when multiple users use your service concurrently:

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

You will need to add transactions, to ensure commands executing in parallel do not conflict. Think carefully as to which commands need transaction handling. Do the create, login and search commands need transaction handling? What about book, pay, and reservations? Why or why not?

Step 5: Write More (Transaction) Test Cases

Write at least 1 test case for each of the 3 new commands you just implemented. Follow the same format as our provided test cases.

Next, write at least 1 *parallel* test case for each of the ~~7~~ 6 commands. By *parallel*, we mean concurrent users interfacing with your database, with each user in a separate application instance. As before, you may find the documentation on our test case format helpful.

Step 6: Reflect on Your Voyage of Learning

Lastly, please reflect on your personal experience with this project. Specifically:

1. *The 3 standard reflection questions, but specific to your experience with M2:*
    - *What is one thing that you learned while doing M2?*
    - *What is one thing that surprised you while doing M2?*
    - *What is one question that you still have after doing M2?*
2. *Were you able to make any progress on your "one question I still have" from M1? If so, please describe how it happened (eg, did you stumble upon the answer, did it come up in lecture, did you read about it yourself)*
3. *This homework demonstrated how application logic can perform simple transformations on query results (eg, merging direct and indirect itineraries) or on user input (eg, salting and hashing passwords). What other logic would you like to see and/or implement in Java, that cannot be done in SQL or by the user?*
4. ***[optional]** how many hours you spent on M2, how many hours you perceived to be valuable, and how many students (if any!) you collaborated with*

Save your answer to these reflection questions in a file called hw52-writeup.txt in the submission directory.

M2 Submission

For this milestone, you should submit these files to Gradescope:

- createTables.sql
- Your fully-complete Query.java and PasswordUtils.java
- At least 12 test cases ~~in the `cases/mycases/` folder~~, named as ~~`[no_]transaction/`~~`<command>_<description>.txt`
    - 6 must be serial tests, one for each command
    - 6 must be parallel tests, one for each command
- hw52-writeup.txt

The bulk of HW5's total points will be allocated to this final milestone.

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