Laboratory 1 | Simple Application

CSE 344 - Introduction to Data Management

You will find the starter files linked on the website as “cse344-lab1.zip”.

Note on collaboration: For the labs, you are expected to work alone. However, you are allowed to exchange test case files.

While we provide a testing framework for most of your 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.

Because your solution is of your own design, please provide a short writeup and name it “lab1.pdf”. Any figures included should be neatly drawn and have legible text.

Submit your solution files to Canvas. Things to turn in:

- lab_code.zip
- lab1.pdf

0. Setup and General Specifications

10,000-foot view of the application

Congratulations! You are opening your own flight booking service!

In this lab, you have two main tasks:

1. Design a database of your customers and the flights they book.
2. Complete a working prototype of your flight booking application that connects to the database then allows customers to use a CLI to search, book, cancel, etc. flights.

You will also be writing a few test cases and explaining your implementation in a short writeup. We have already provided code for a UI and partial backend; you will 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 8 standard JDK.

Connect your application to your database

You will need to access your Flights database on SQL Azure from HW1. Alternatively, you may create a new database and use the HW1 specification for importing Flights data. Modify dbconn.properties for this lab with your server URL, database name, username, and password just as you did for HW1. Use a fake username and password for dbconn.properties or delete dbconn.properties before turning in your implementation.

Make sure your application can run by entering the following commands. 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/lab1-1.0-jar-with-dependencies.jar
```

If you get our super duper sexy 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

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 persist 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 HW1. For this application, we have very limited functionality so you shouldn't need to modify the schema from HW1 nor add any new tables to reason about the data.

- **Users**: A user has a username (varchar), password hash (varbinary), password salt (varbinary), and balance (int) for their account. All usernames should be unique in the system. Each user can have any number of reservations. There is no restriction on passwords when creating a new user. 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.
- **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 and has a unique ID.

Application Requirements

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 (string), password (string), and initial account balance (int) as input. It creates a new user account with the initial balance. It should return an error if negative, or if the username already exists. Usernames are checked case-insensitively. For simplicity, 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 (string) and password (string) 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. You can track this via a local variable in your program. If a second login attempt is made, please return “User already logged in”. Across multiple sessions (that is, if you run your program multiple times), the same user is allowed to be logged in. This means that you do not need to track a user’s "logged in status" inside the database.

- **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 k direct itineraries (where k < n), then return the k direct itineraries and then return 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 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 and cannot be booked. If the 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 an "ID" table that stores the next ID to use and update it each time when a new reservation is made successfully. 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 should not be displayed.

- **cancel** lets a user 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 a detailed specification/expected responses of the commands above. **Make sure your code produces outputs in the same formats as prescribed!**

1. Database Design (10 points)

Your first task is to design and add tables to your 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.

In the text file called createTables.sql, write the CREATE TABLE statements and any INSERT statements needed to implement the logical data model above. You may use anything provided in SQL Server.

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.

2. Java Application (70 points)

Your second task is to write the Java application that your customers will use, by completing the starter code. You only need to modify Query.java. Do not modify FlightService.java. For this lab, we are only concerned about the correctness of your methods. You do not need to implement any parallelization/transactions... yet. We expect that you use prepared statements where applicable. Please make your code reasonably easy to read.

To keep things neat we have provided you with the Flight inner class that acts as a container for your flight data. The toString method in the Flight class matches what is needed in methods like search. We have also provided a sample helper method checkFlightCapacity that uses a prepared statement. checkFlightCapacity outlines the way we think forming prepared statements should go for this assignment (creating a constant SQL string, preparing it in the prepareStatements method, and then finally using it).

Milestone 0: 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 createTables.sql called.

This method is for running the test harness where each test case is assumed to start with a clean database. clearTables should not take more than a minute. Make sure your database schema is designed with this in mind.

Milestone 1: Implement create, login, and search

Aim to complete this by April 23.

Milestone 2: Implement book, pay, reservations, and cancel

Aim to complete this by April 30.

3. Test Cases (10 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 in the lab1 folder:

```
mvn test
```

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
```

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 along with the original test cases. 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).

4. Writeup (10 points)

Please describe and/or draw your database design. This is so we can understand your implementation as close to what you were thinking. Justify your design choices in creating new tables. 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 (< ½ page).