Introduction

Congratulations, you are opening an airline management service!

In this project, 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 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 project, your task is to implement the rest of the backend. You can use any of the classes from the Java 21 standard JDK.

Changes to the specification after release are highlighted in red.

Project Setup

1. **Install Postgres (instructions)**

2. **Install Maven (build system)**
    a. If using OSX, we recommend using Homebrew and installing with `brew install maven`. If you don't have Homebrew: instructions @ https://brew.sh/
    b. On Windows, download the "Binary zip archive" .zip file from the maven download page linked above. Unzip it wherever you want maven to live permanently on your machine. Then add the …/bin folder to your PATH.

3. **Download the starter code** (.zip format)

4. **Set up your Flights database in Postgres**
    a. In psql, run: `CREATE DATABASE flights;`
    b. Then run `\c flights` to connect to the database you just created. In the future, when you invoke psql, you should repeat this \c command each time.
    c. Follow the comments in the provided createTables.sql file to design your create table statements. Run those create table statements. (Note the schema is slightly different from earlier homeworks.)
    d. Download the flights data. (Note this is slightly different from the data we used earlier this quarter.) Unzip it.
    e. Run the following \copy commands in psql

    ```
    \copy CARRIERS from 'carriers.csv' CSV
    \copy MONTHS from 'months.csv' CSV
    \copy WEEKDAYS from 'weekdays.csv' CSV
    \copy FLIGHTS from 'flights-small.csv' CSV
    ```

    You may need to adjust the filenames to point to where you unzipped the data.

5. **Configure your JDBC Connection**
   In the top level directory, edit the connection settings in `dbconn.properties`. Specifically, change `SERVER_URL`, `DATABASE_NAME`, `USERNAME`, `PASSWORD`, and `UWNetID`. This allows Query.java to connect to your Postgres instance.

```
# 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:

   - `SERVER_URL` will be "localhost" when working locally
   - `DATABASE_NAME` is the name of the database you created (flights)
   - The `USERNAME` and `PASSWORD` are the same credentials you use to login to your database/server when you installed Postgres

6. **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 project 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
```

   NOTE: `mvn compile exec:java` does not run your `createTables.sql` file. If your tables aren’t recognized the first time, or you make changes to them later, run any of the test commands first, even if it fails the tests. Alternatively, you can just manually paste and run your CREATE TABLE statements into Postgres the first time.

Project Requirements

Data Model

The airline management system consists of the following constructs. These constructs are *not necessarily* entity sets or *database tables*; it is up to you to decide what to store persistently.

- **Flights / Carriers / Months / Weekdays**: **For this project you should consider them to be "read-only".**

- **Users**: A user has a username (postgres type: text), password (postgres type: bytea for "byte array"), 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 Postgres~~ (no it's not). 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 (i.e. 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 (e.g. 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 QueryAbstract.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***.

What this means is we first *choose* direct itineraries, but we sort on flight time (whether or not the itinerary is direct). So if we want *k* itineraries, and we have *d* direct flights, we want *i* = *k* - *d* indirect flights (assuming k > d otherwise i = 0). You break ties for flight time with the fid.

For example, say we have the following itineraries (D for direct and I for indirect):

- D1: 50 min
- D2: 30 min

- I1: 40 min
- I2: 10 min
- I3: 100 min
- I4: 80 min

We would return the following itineraries in the order shown for these search requests:

User has requested only direct flights:

- 1 itinerary: D2
- 2 itineraries: D2 D1

User has requested indirect and direct flights:

- 1 itinerary: D2
- 2 itineraries: D2 D1
- 3 itineraries: I2 D2 D1
- 4 itineraries: I2 D2 I1 D1
- 5 itineraries: I2 D2 I1 D1 I4
- 6 itineraries: I2 D2 I1 D1 I4 I3

In other words, get the maximum number of direct flights, then fill any missing itineraries with indirect itineraries, combine the two lists, and finally sort on time.

- **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. **Notably, do not modify the `FLIGHTS` table**.

- **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.

- **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 1:

Prework

After reading through the requirements, you will translate the project requirements into a conceptual model (an E/R diagram) and then to a schema (physical tables).

First, you will create an E/R diagram. The diagram should adequately *represent* (not necessarily as their own entities) flights, carriers, months, weekdays, users, itineraries, and reservations. You may omit 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` 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`). To prevent interference, we require that any table names in your `createTables.sql` be **suffixed with your UWNetID** (eg, `MyTable_hctang`).

Java Customer Application

Your next task is to start writing the Java application that your customers will use. You will only need to modify `Query.java` and `PasswordUtils.java`; do not modify FlightService.java.

Note: Use PreparedStatements appropriately (refer to section and lecture if you are confused) when you execute queries that include user input. Dynamically-generated SQL statements permit SQL injection attacks; don't do this. **We will deduct points if we see that your code is susceptible to SQL injection.**

Step 1: Implement clearTables()

Implement the `clearTables()` method in Query.java to clear the contents of *any tables you have created* for this assignment (i.e., any tables in `createTables.sql`). Notably: **do not drop** any of your tables, and **do not modify** the contents of or drop the `Flights/Months/Weekdays/Carriers` tables.

`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 (i.e. with the `FLIGHTS` table populated and `createTables.sql` called). An incorrect implementation will cause difficult-to-debug test failures.

Step 2: Implement PasswordUtils.java

Implement salting and hashing by completing the three methods marked with `TODO` comments in `PasswordUtils.java`. You will need this working before implementing the functions in Query.java. You can test using `mvn test -Dtest=flightapp.PasswordUtilsTest` .

Step 3: Implement create, login, and search

Implement the **create**, **login** and **search** commands in Query.java (in that order). See QueryAbstract.java for javadoc. `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/l
ogin:cases/no_transaction/create"

# If on Windows, you will need to quote the entire argument
# to `mvn test` as follows:
$ mvn test "-Dtest.cases=folder_name_or_file_name_here"

# You'll need to change the quoting for all the `mvn test`
# commands above, too.
```

Step 4: Write Test Cases

Write at least 1 new **non-transactional** 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 (i.e. `<cmdname>_<description>.test.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 the following files to Gradescope:

- Your ER diagram
- `createTables.sql`
- Fully complete `PasswordUtils.java`
- Partially-complete `Query.java` with 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)

Milestone 2:

Step 1: Implement book, pay, and reservations

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

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

Step 2: Add transactions!

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 your application code, to ensure commands executing in parallel do not conflict. Think carefully as to which commands need transaction handling.

Step 3: 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 command except search and create (ie, you will submit a total of 4 parallel tests). By "parallel", we mean (1) concurrent users interfacing with your database, each in a separate application instance, and **(2) multiple possible outcomes/scenarios**. As before, you may find documentation on our test case format helpful.

M2 Submission

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

- `createTables.sql`
- Your fully-complete `Query.java` and `PasswordUtils.java`
- At least 10 test cases, `<command>_<description>.test.txt`
    - **Please be sure they are named correctly**
    - 6 must be serial tests, one for each command. You can resubmit tests previously submitted for M1.
    - 4 must be parallel tests, one for each command except search and create

Please check that your submission is visible and autograder score is what you expected.