Term Project Documentation

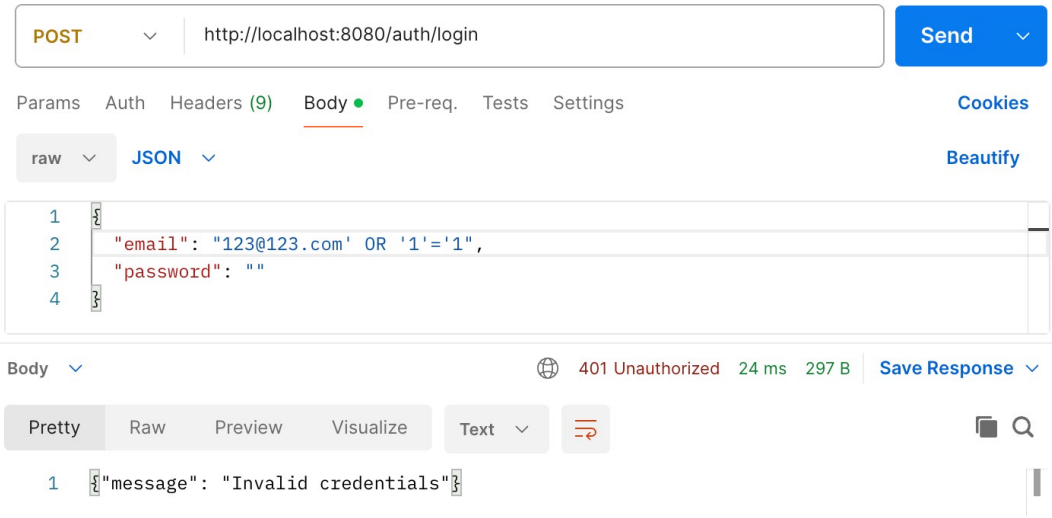
Project Overview

The Term Project implements a full-stack web applications of an Airline Ticketing Database. Our objective was to create an application that simplified the flight booking experience. User can create accounts in the passenger class, with a unique passenger ID and make reservations for as many people as they like. With an intuitive UI, it is very simple to navigate through it and in-case you enter invalid credentials or inputs, alerts are set up to let the user know exactly what the issue was. The user account will securely also store passenger details such as email, name, phone number, and a securely hashed password. As security was one of our top priorities, we handled User Authentication using a JWT token as well as accounted for SQL Injections.

Key Features

Secure Data Management:

- Passenger accounts store sensitive information such as passwords securely by storing the password that is hashed using BCrypt in the database. Additionally, the application handles possible SQL injections by returning a 401 Unauthorized Error.
- This is one example from all the SQL Injections that were attempted:



  Here, I tested whether the /auth/login endpoint would bypass the authentication when the password field is empty. And as you can see in the image above, it returned a message saying, “Invalid credentials” and outputted a *401 Unauthorized* HTML Error Code.

- This is a list of all the SQL Injections that were tested:

- {"email": "123@123' OR '1'='1", "password": "123"}
- {"email": "123@123'", "password": "123"}
- {"email": "123@123'--", "password": "123"}
- {"email": "123@123", "password": "123' OR '1'='1"}
- {"email": "123@123' UNION SELECT null, null, null, database(), user()--", "password": "123"}
- {"email": "123@123' AND IF(1=1, SLEEP(5), null)--", "password": "123"}
- {"email": "123@123' AND 1=CONVERT('test', SIGNED)--", "password": "123"}
- {"email": "123@123' OR '1'='1", "password": ""}

Efficient Booking System:

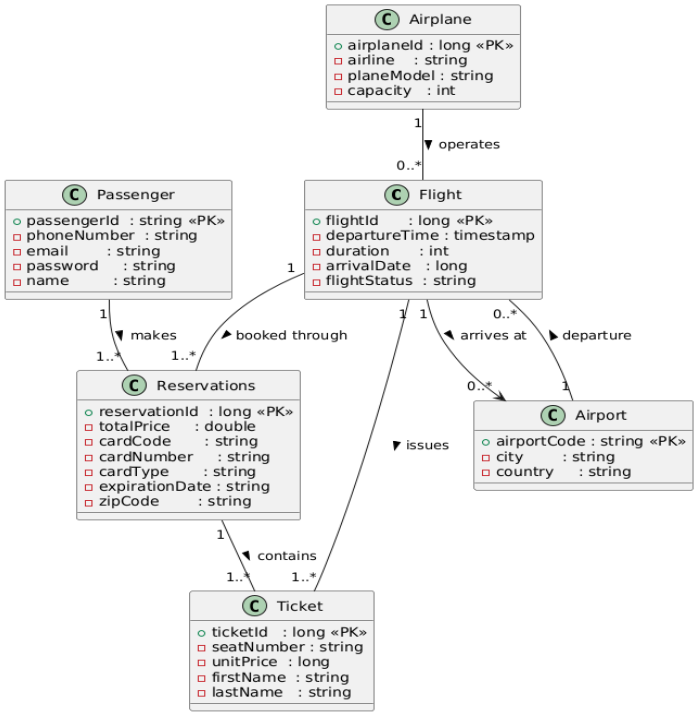
- The application allows customers to browse flights, make reservations, receive instant updates, make a payment and view all bookings.

Scalable Database Design:

- Robust database schema with structured data for flights, payments, reservations, and more. Currently, the schema only has 8 flights that we created as test cases, however the database is scalable to account for more flights.

Schema Comparison

UML Diagram



The diagram above, details the UML (Unified Modeling Language) Diagram for the database used for this application. As you can see above, we created 6 tables: (1) Flight, (2) Airplane, (3) Airport, (4) Ticket, (5) Reservations, and (6) Passenger.

To summarize the associations and municipalities of the database, each flight has a 1 airplane, and each airplane can have more than one flight. Flight essentially represents an airplane that goes from an origin to a destination. Therefore, each flight is associated with 2 airports, but an airport can have multiple flights going out or coming into it. If we look at passenger, each passenger can make 1 or many reservations, and a reservation can include more than 1 tickets. So, ticket and reservations have a Many to One relationship. This means that 1 ticket is only associated with 1 reservation, but a reservation can have multiple tickets. Finally, both reservation and ticket with be associated with 1 flight, but 1 flight can have 0 or many reservations and tickets.

UML Diagram to Relational Model

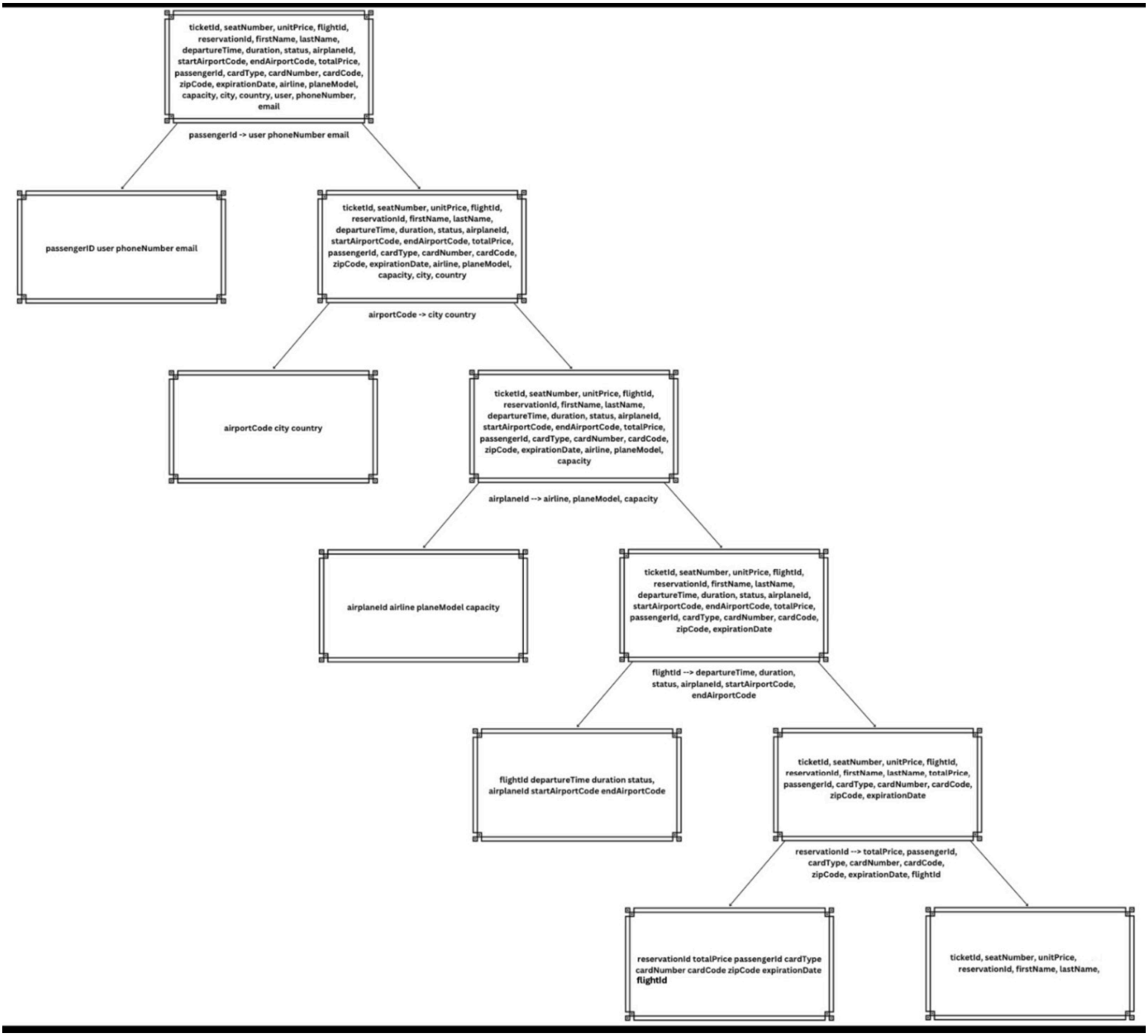
Table number: 6
Primary key: 6
Foreign key: 7
Tables:
R1: (Airplane): airplaneId, airline, planeModel, capacity
R2: (Airport): airportCode, city, country
R3: (Flight): flightId, departureTime, duration, flightStatus, *airplaneid*, *startAirportCode*, *endAirportCode*
R4: (Passenger): passengerId, phoneNumber, email, password, name
R5: (Ticket): ticketId, seatNumber, unitPrice, *flightId*, *reservationId*, firstName, lastName
R6: (Reservation): reservationId, totalPrice, cardCode, cardNumber, cardType, expirationDate, zipCode, *passengerId, flightId*

Since we have 6 tables in our UML diagram, each with 1 Primary Key, we ended up with 6 PKs. As for Foreign Keys, the Flight table has the FKs startAirportCode and endAirport code which are PKs in the Airport table. Since the Flight table has 2 FKs and the other tables have 1 FK, that is how we ended up with a total of 7 FKs. The Relational Model shows the table names in ( ), the primary keys underlined and the foreign keys italicized.

Additionally, we found that the ticket table was the Global Key because using it, you can reach all the attributes in the database.

Boyce Codd Normal Form – BCNF

airplaneId --> airline, planeModel, capacity
airportCode --> city, country
flightId --> departureTime, duration, flightStatus, airplaneid, startAirportCode, endAirportCode
passengerId --> phoneNumber, email, password, name
ticketId --> seatNumber, unitPrice, flightId, reservationId, firstName, lastName
reservationId --> totalPrice, cardCode, cardNumber, cardType, expirationDate, zipCode, passengerId, flightId



In the BCNF, we started with all the attributes in the Relational Model and deduced each branch based on the Functional Dependencies of the RM. So at each split, the attributes of the Functional dependency is on the left branch, and the attributes without the right side of the functional dependency is on the right branch after split. So, first I deduced it by passengerId, then airportCode, airplaneId, and flightId. When I got to reserverationId, it posed an issue because the ticket table also references the flightId. So, in order to not lose any information, I had to push flightId to the end, which meant that the model above was no longer in BCNF form. Therefore, we decided against using this model as our final SQL Schema Design.

Third Normal Form - 3NF Synthesis

airplaneId --> airline, planeModel, capacity
airportCode --> city, country
flightId --> departureTime, duration, flightStatus, airplaneid, startAirportCode, endAirportCode
passengerId --> phoneNumber, email, password, name
ticketId --> seatNumber, unitPrice, flightId, reservationId, firstName, lastName
reservationId --> totalPrice, cardCode, cardNumber, cardType, expirationDate, zipCode, passengerId, flightId

Step 1 : Minimal Coverage (RHS are singleton sets)

- airplaneId --> airline
- airplaneId --> planeModel
- airplaneId --> capacity
- airportCode -> city
- airportCode -> country
- flightId -> departureTime
- flightId -> duration
- flightId -> flightStatus
- flightId -> airplaneid
- flightId -> startAirportCode
- flightId -> endAirportCode
- passengerId -> phoneNumber
- passengerId -> email
- passengerId -> password
- passengerId -> name
- ticketId -> seatNumber
- ticketId -> unitPrice
- ticketId -> flightId
- ticketId -> reservationId
- ticketId -> firstName
- ticketId -> lastName
- reservationId -> totalPrice
- reservationId -> cardCode
- reservationId -> cardNumber
- reservationId -> cardType
- reservationId -> expirationDate
- reservationId -> zipCode
- reservationId -> passengerId
- reservationId -> flightId

Step 2 : Minimal Coverage (LHS has no extraneous attributes)

- airplaneId --> airline
- airplaneId --> planeModel
- airplaneId --> capacity
- airportCode -> city
- airportCode -> country
- flightId -> departureTime
- flightId -> duration
- flightId -> flightStatus
- flightId -> airplaneid
- flightId -> startAirportCode
- flightId -> endAirportCode
- passengerId -> phoneNumber
- passengerId -> email
- passengerId -> password
- passengerId -> name
- ticketId -> seatNumber
- ticketId -> unitPrice
- ticketId -> flightId
- ticketId -> reservationId
- ticketId -> firstName
- ticketId -> lastName
- reservationId -> totalPrice
- reservationId -> cardCode
- reservationId -> cardNumber
- reservationId -> cardType
- reservationId -> expirationDate
- reservationId -> zipCode
- reservationId -> passengerId
- reservationId -> flightId

Step 3 : Minimal Coverage (No redundant FD's)

- airplaneId --> airline
- airplaneId --> planeModel
- airplaneId --> capacity
- airportCode -> city
- airportCode -> country
- flightId -> departureTime
- flightId -> duration
- flightId -> flightStatus
- flightId -> airplaneid
- flightId -> startAirportCode
- flightId -> endAirportCode
- passengerId -> phoneNumber
- passengerId -> email
- passengerId -> password
- passengerId -> name
- ticketId -> seatNumber
- ticketId -> unitPrice
- ticketId -> flightId
- ticketId -> reservationId
- ticketId -> firstName
- ticketId -> lastName
- reservationId -> totalPrice
- reservationId -> cardCode
- reservationId -> cardNumber
- reservationId -> cardType
- reservationId -> expirationDate
- reservationId -> zipCode
- reservationId -> passengerId
- reservationId -> flightId

Step 2 : Merge the FD with the same LHS

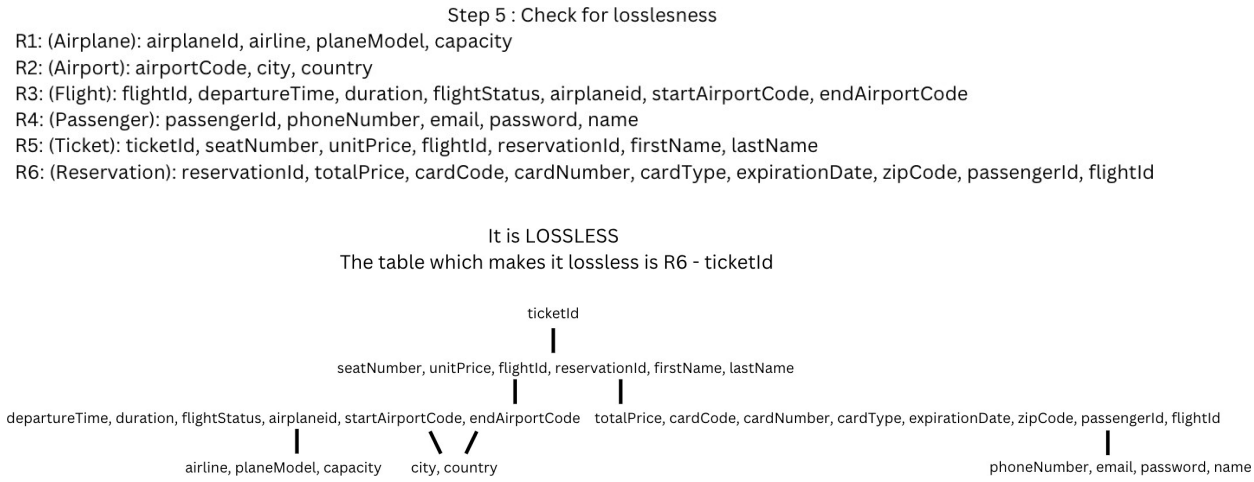
airplaneId --> airline, planeModel, capacity
airportCode --> city, country
flightId --> departureTime, duration, flightStatus, airplaneid, startAirportCode, endAirportCode
passengerId --> phoneNumber, email, password, name
ticketId --> seatNumber, unitPrice, flightId, reservationId, firstName, lastName
reservationId --> totalPrice, cardCode, cardNumber, cardType, expirationDate, zipCode, passengerId, flightId

Step 3 : Form a table for each FD

R1: (Airplane): airplaneId, airline, planeModel, capacity
R2: (Airport): airportCode, city, country
R3: (Flight): flightId, departureTime, duration, flightStatus, airplaneid, startAirportCode, endAirportCode
R4: (Passenger): passengerId, phoneNumber, email, password, name
R5: (Ticket): ticketId, seatNumber, unitPrice, flightId, reservationId, firstName, lastName
R6: (Reservation): reservationId, totalPrice, cardCode, cardNumber, cardType, expirationDate, zipCode, passengerId, flightId

Step 4 : Remove a subset table

R1: (Airplane): airplaneId, airline, planeModel, capacity
R2: (Airport): airportCode, city, country
R3: (Flight): flightId, departureTime, duration, flightStatus, airplaneid, startAirportCode, endAirportCode
R4: (Passenger): passengerId, phoneNumber, email, password, name
R5: (Ticket): ticketId, seatNumber, unitPrice, flightId, reservationId, firstName, lastName
R6: (Reservation): reservationId, totalPrice, cardCode, cardNumber, cardType, expirationDate, zipCode, passengerId, flightId



So, as you can see in the image above, we normalized the Relational Model using Third Normal Form (3NF Synthesis). These were the steps we followed:

1. Minimal Coverage
    a. Handle Singleton Sets
    b. No Extraneous Attributes
    c. No Redundant FDs
2. Merge FDs with the same Left-Hand Side (LHS)
3. Form a Table for each FD
4. Remove Subset Tables
5. Check for Losslessness

After performing all these steps, we came to the conclusion that the 3NF Synthesis was lossless and the table that made it lossless was the Ticket table (the one with ticketId global key). Since, the 3NF gave us the best design structure due to its lossless nature, we decided to use 3NF as our basis for the final SQL Schema.

UML	3NF	BCNF
Visual representation of database structure and relationships	Reduces redundancy	Is a stricter normalization technique
Does not address functional dependencies	Removes transitive dependencies	Removes all redundancies
Not a database normalization method	Is a practical implementation of normalization	Normalization can result in a more complex schema

After analyzing the three schemas resulting from the UML, 3NF and BCNF, we decided to use 3NF as our final schema because it resulted in a lossless structure, removed redundancy and was a more practical implementation.

Software Architecture and Components

- **Frontend:** React – HTML, CSS, JavaScript
- **Backend:** Java, SpringBoot, Node.js
- **Database:** MySQL

What's next?

So, for this project, we created our own tables for the database with example data values; however, to further improve on this project, we could integrate an API that retrieves real-time flight data (like FlightAPI) into our database. Additionally, we could also incorporate price comparisons for different airlines as well as the different seat classes. We could also allow users to save multiple payment options in their account, track price drops for preferred flights, use search history to recommend flights, as well as integrating payment gateways for a seamless transaction experience. We look forward to further developing this project!