Flight Booking System

Gengiu Robert-Constantin

Group: 30421

1. Project Description

The project is a Flight Booking System designed to facilitate airline ticket reservations for both administrators and clients. The primary goal is to provide a user-friendly and efficient platform for managing flight information, booking tickets, and overseeing the overall flight operations. The system caters to both administrative staff responsible for managing flights and clients seeking to book flights seamlessly.

2.Architectureanddesign

The Flight Booking System follows a well-organized architecture, employing the Model-View-Controller (MVC) pattern to achieve a clear separation of concerns. This ensures a modular and maintainable structure, making it easier to extend or modify functionalities in the future.

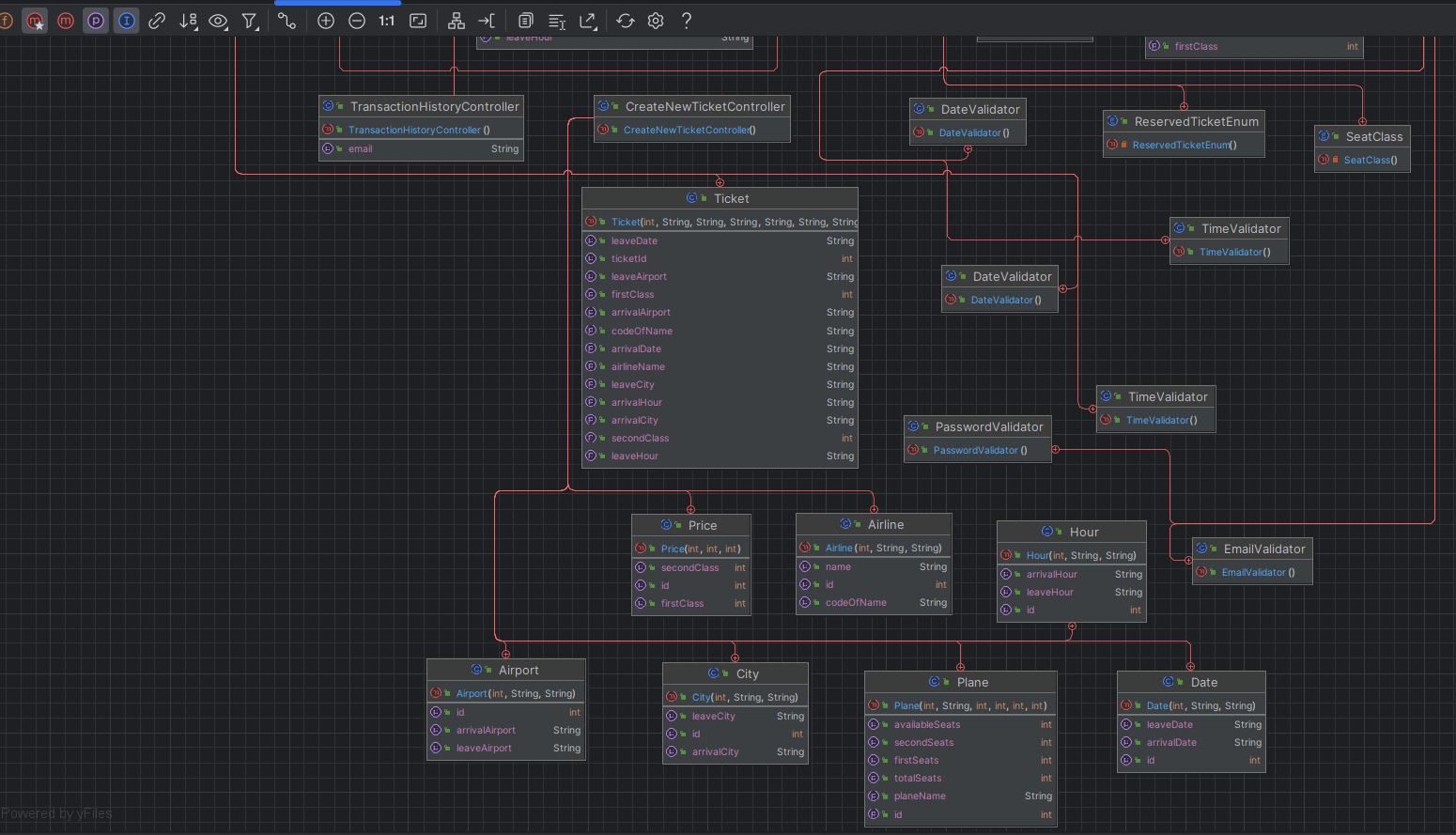
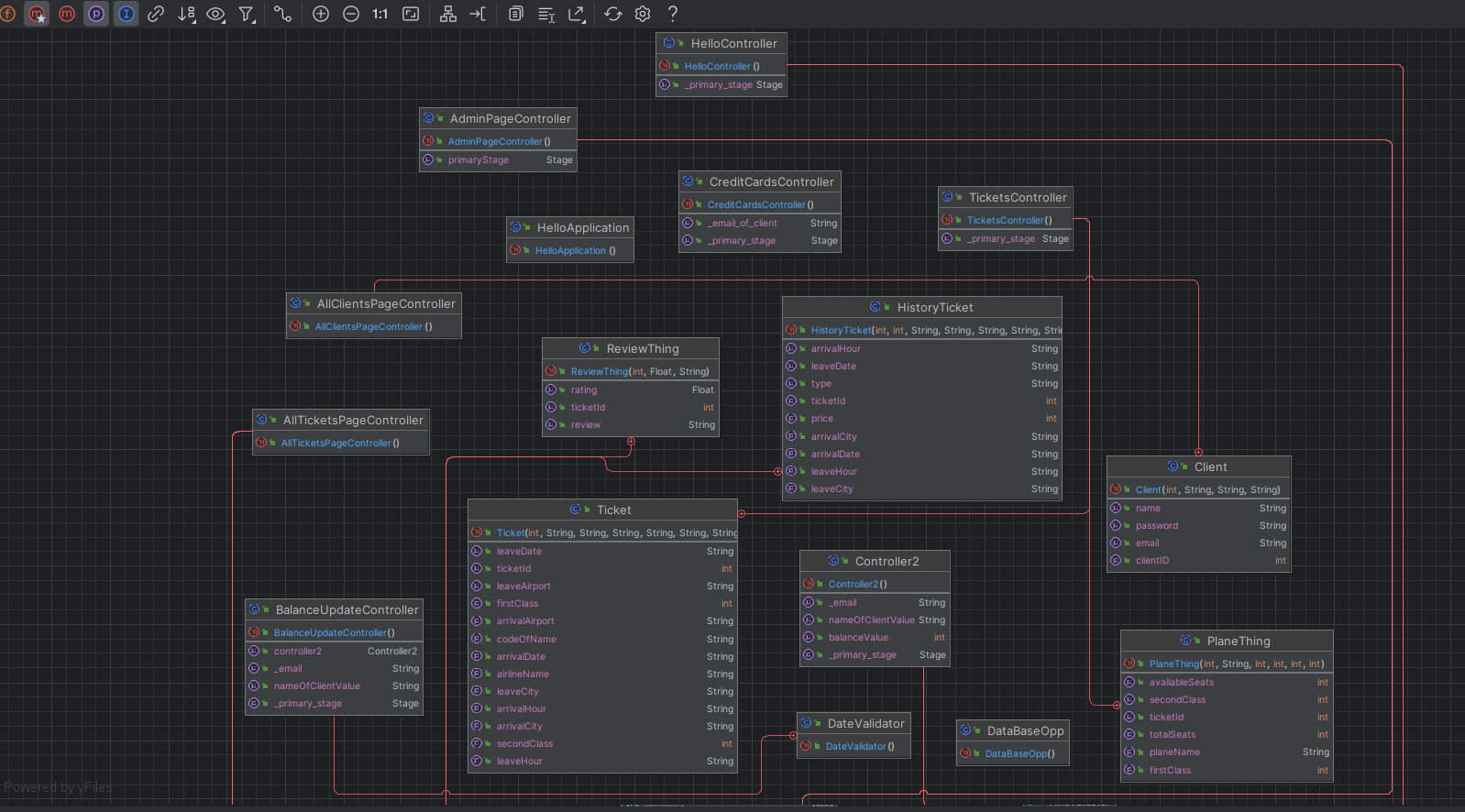
**MVC Architecture:**

* **Model:** Manages the data and business logic of the system. Includes classes for flights, airlines, users, bookings, and other entities. Communicates with the PostgreSQL database to retrieve and persist data.
* **View:** Represents the graphical user interface (GUI) using JavaFX FXML files. Each FXML file has an associated controller responsible for handling user interactions and updating the view.
* **Controller:** Acts as an intermediary between the model and the view. Handles user input, updates the model, and refreshes the view accordingly.

**Class Organization:**

* **AdminController:** Manages interactions related to administrative tasks, such as adding or editing flights, managing airlines, and user accounts.
* **ClientController:** Handles user interactions for searching and booking flights, managing user accounts, and viewing booking history.
* **TicketController:** Responsible for the booking process, linking user selections to flight availability, and managing the booking model.
* **Validation Classes:** Custom classes for validating user inputs, ensuring data integrity, and enhancing the user experience.
* **Class Naming Conventions:** In the Flight Booking System project, the classes have been given suggestive names, ensuring clarity and reflecting their responsibilities. The naming conventions adhere to best practices, making the codebase more understandable

**Class Diagram:**



3. **Object-Oriented Programming Implementation:**

The Flight Booking System project has been developed with a strong adherence to Object-Oriented Programming (OOP) principles, including encapsulation, inheritance, and polymorphism. These principles are fundamental to creating a modular and extensible codebase. Below, I provide a detailed explanation of how OOP principles have been applied in the project:

1. **Encapsulation:**

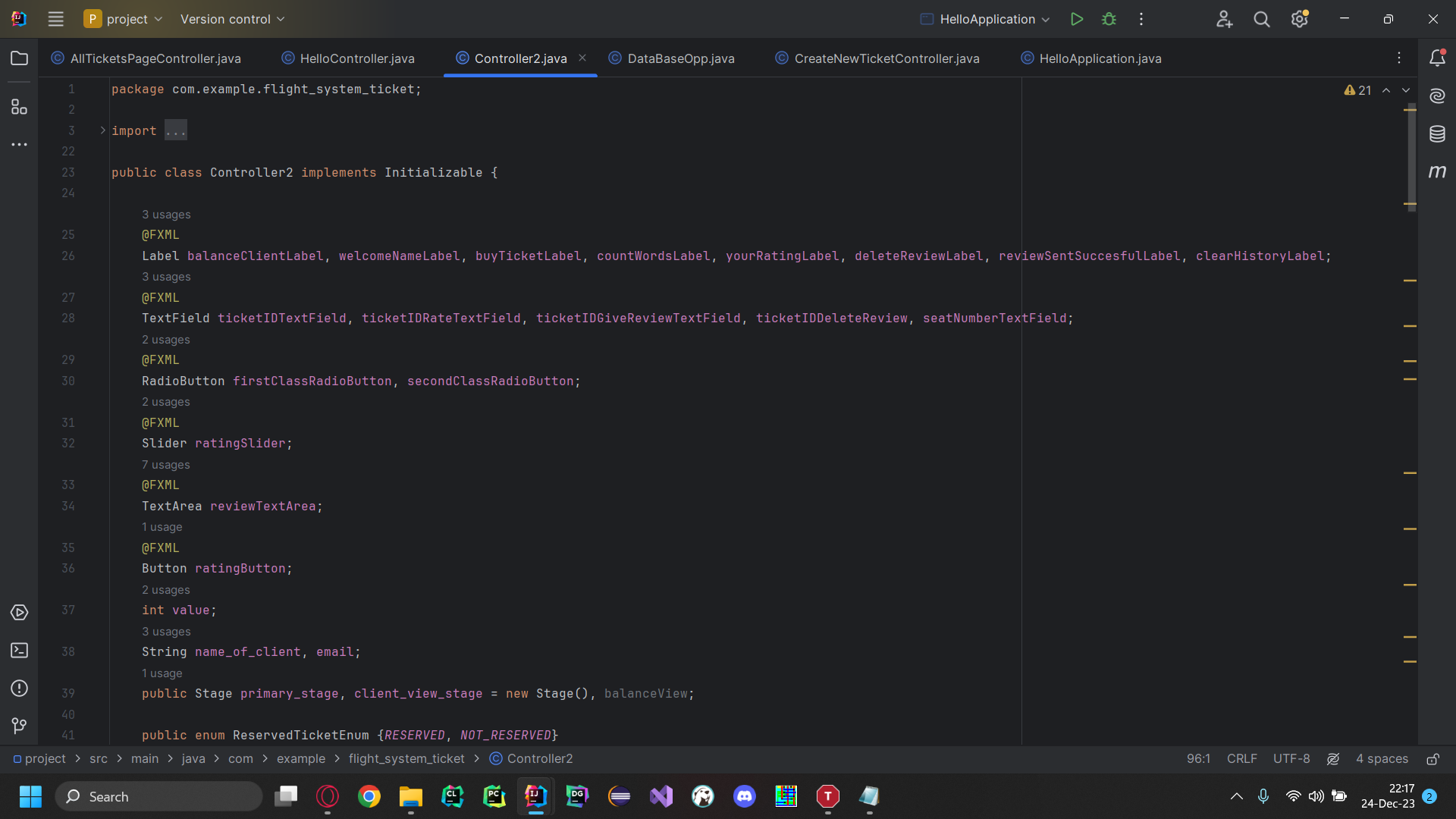
* Encapsulation is implemented by bundling the data (attributes) and methods (behavior) that operate on the data within a class. This shields the internal implementation details from the outside and allows controlled access to the class's functionality.

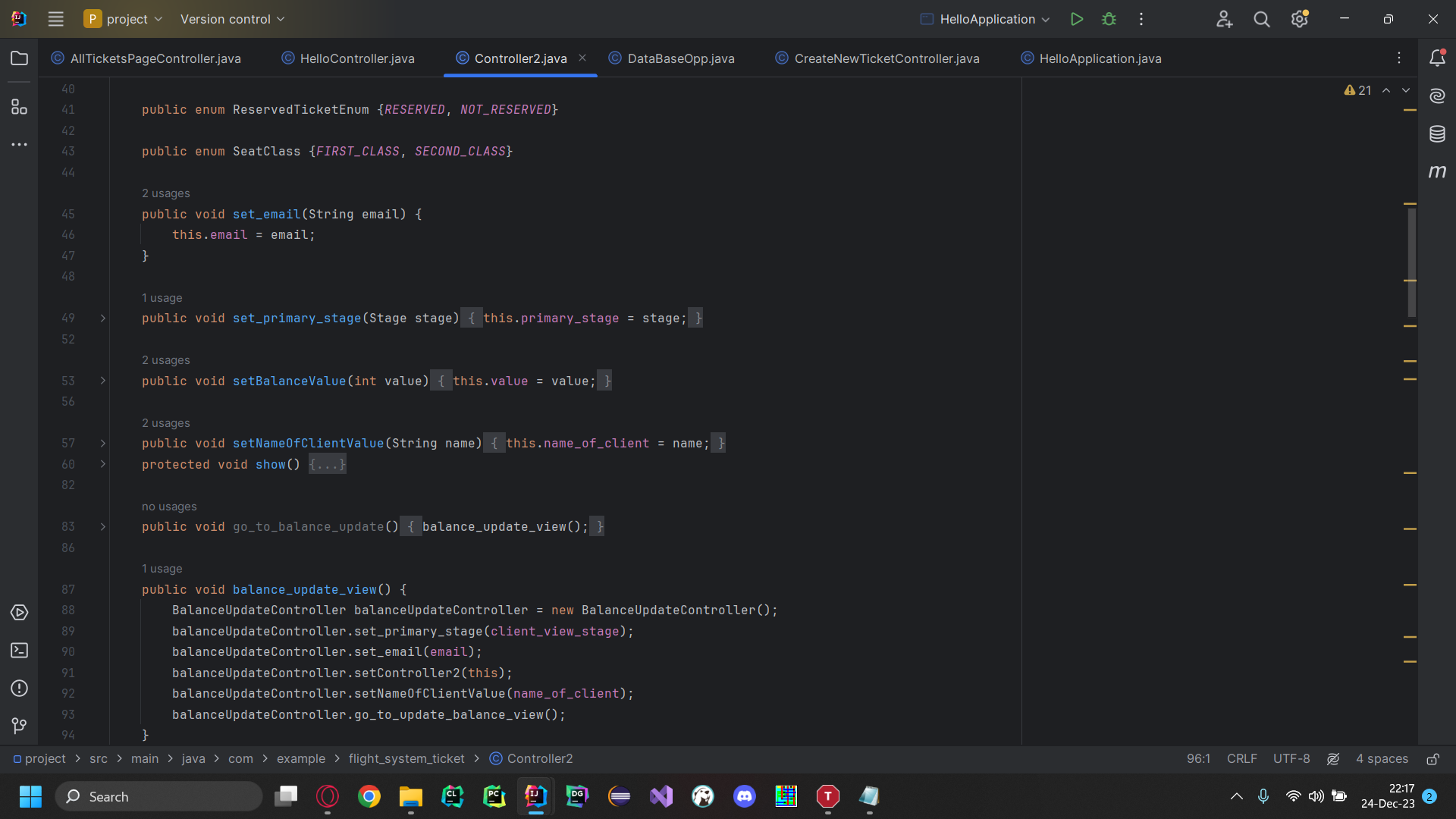
2. **Inheritance:**

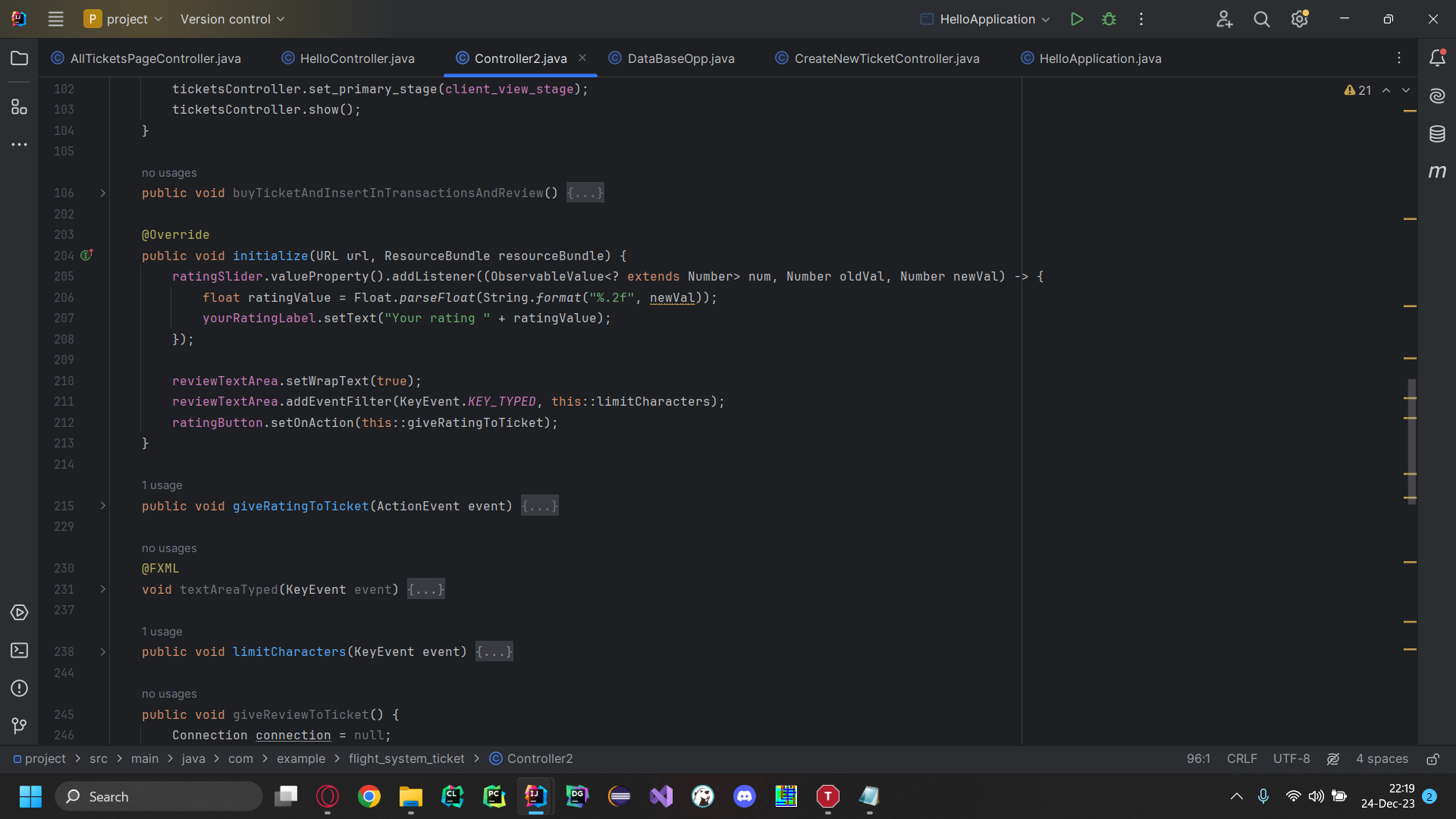
* Inheritance is utilized to create a hierarchy of classes, allowing a subclass to inherit attributes and behaviors from a superclass. This promotes code reuse and the establishment of an "is-a" relationship.

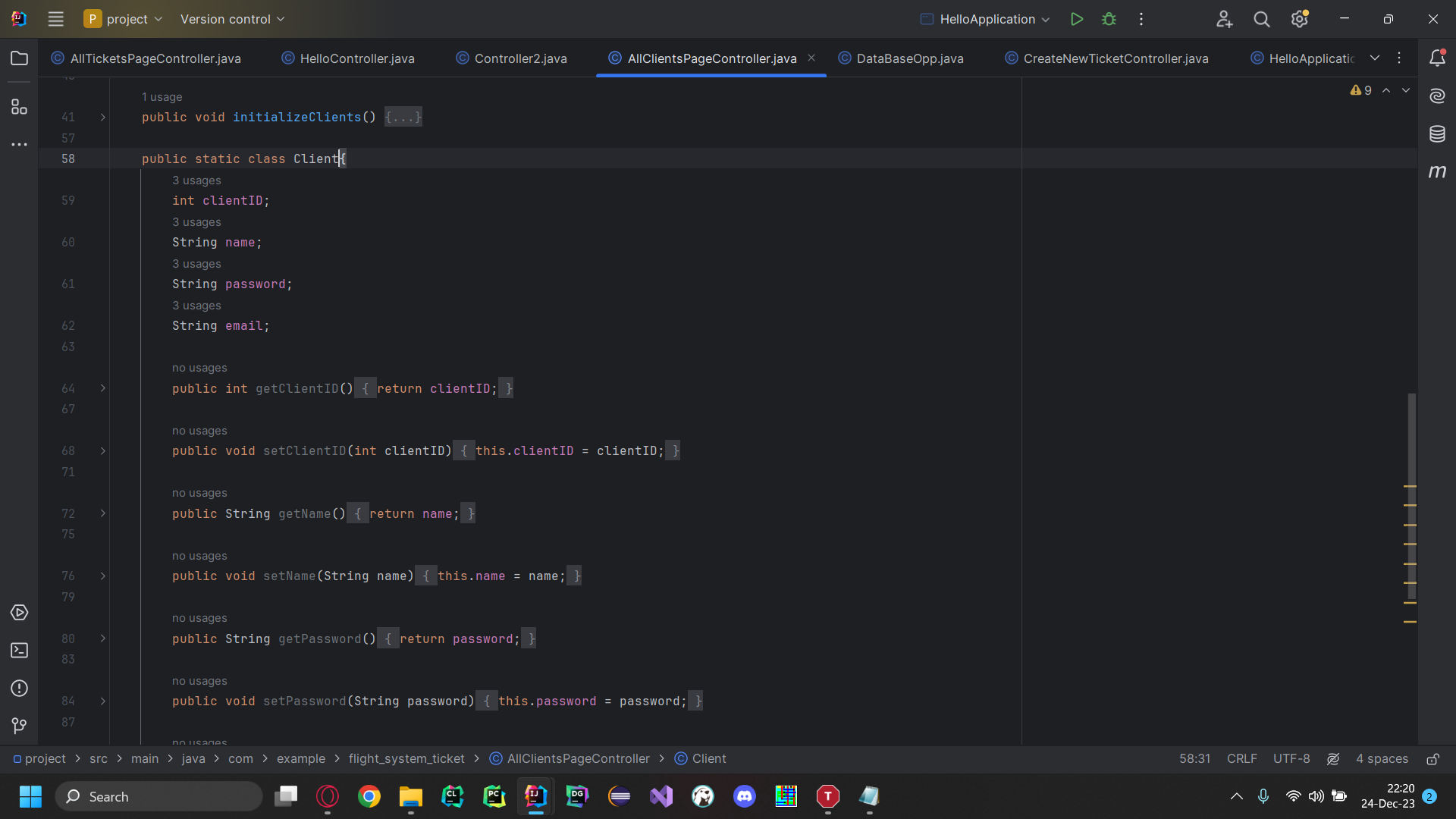
3. **Polymorphism:**

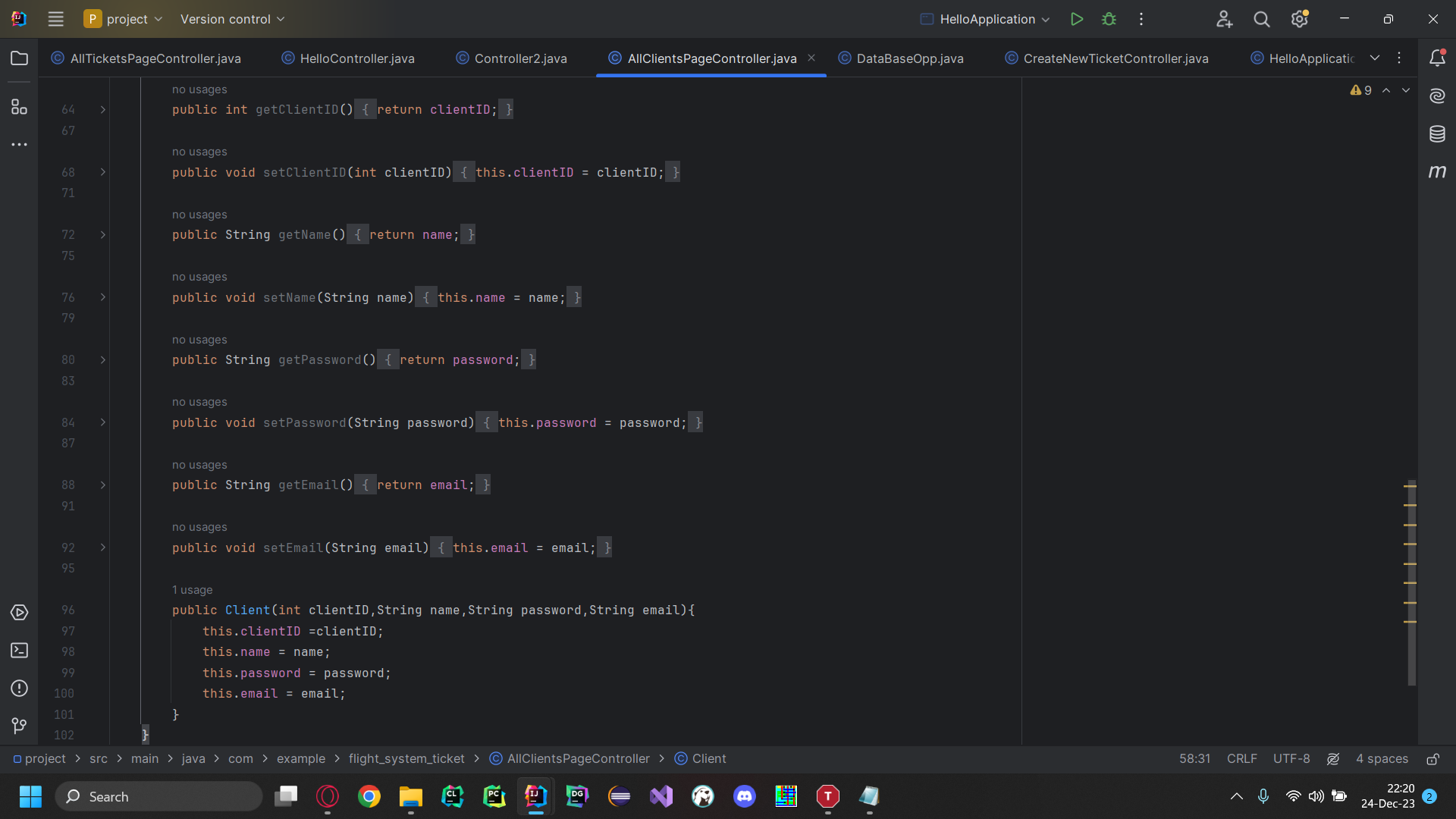
* Polymorphism allows objects to be treated as instances of their parent class, facilitating code flexibility. Method overriding and interfaces are common techniques for achieving polymorphism.
* Example









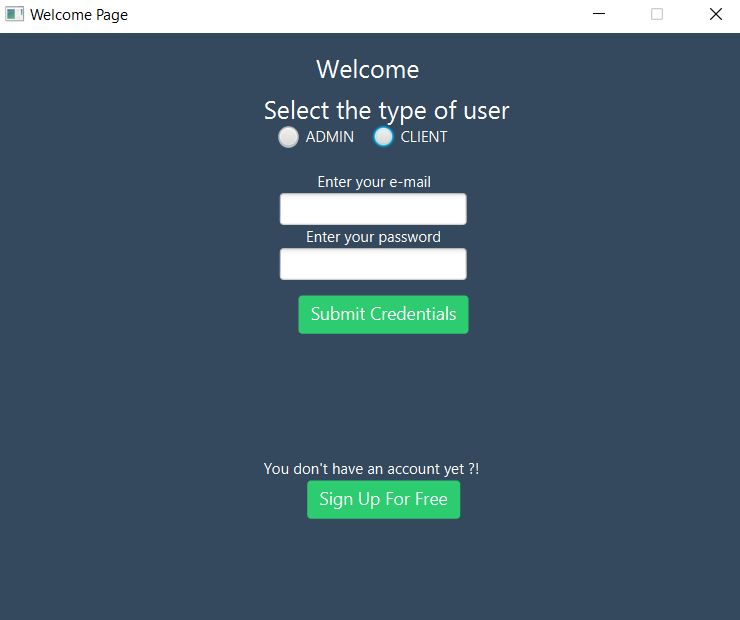


**4. User Interface:**

The Flight Booking System boasts a user-friendly interface developed using JavaFX. Below are descriptions and screenshots of various screens and components within the application:

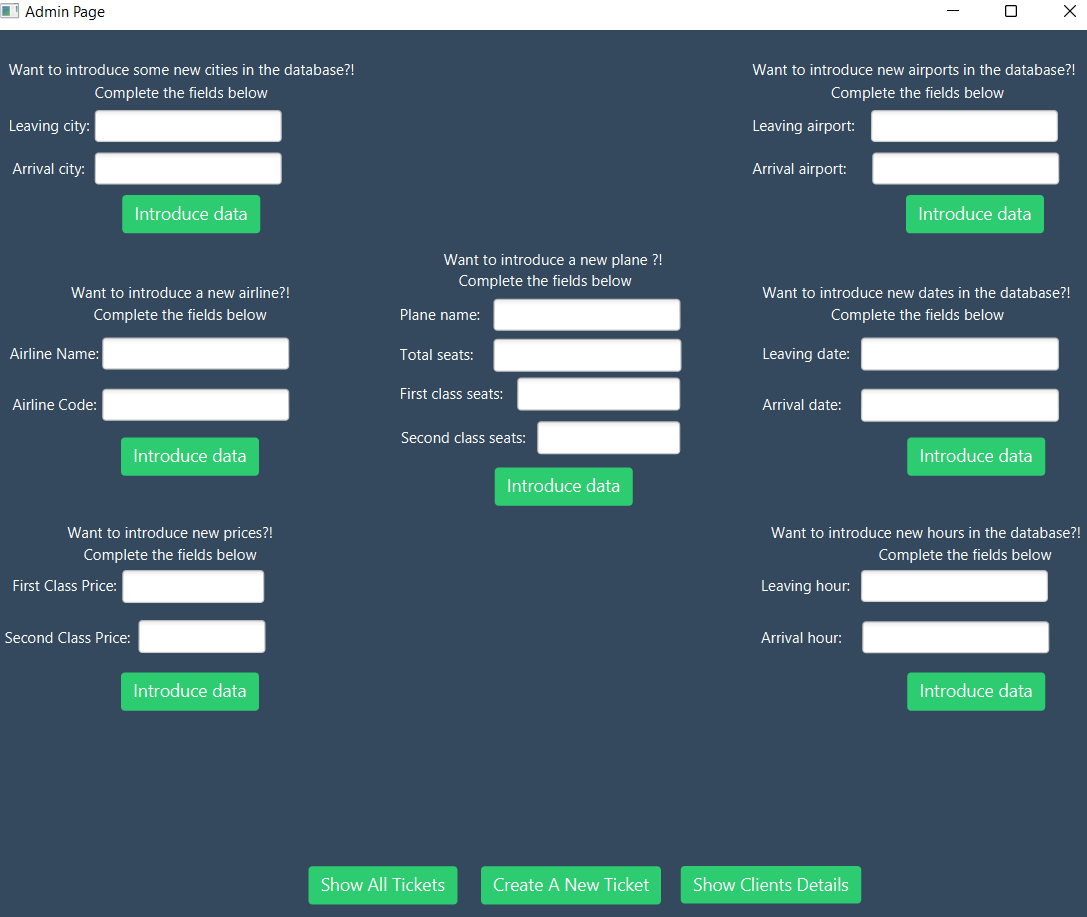
1. **Login Screen:**

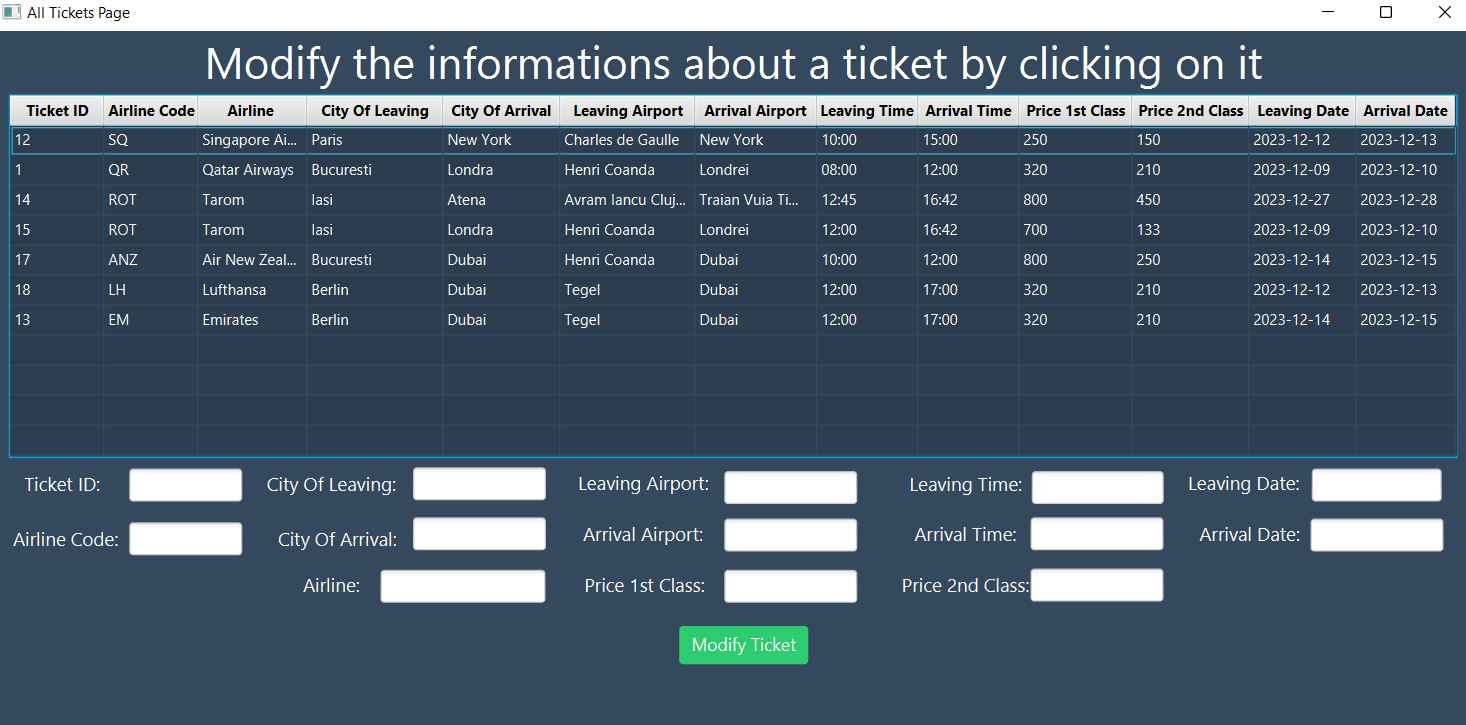
* *Description:*
  + The login screen prompts users to enter their credentials to access the system.
* *Components:*
  + Username and password fields, Login button,Sign up button.
* *Screenshot:*

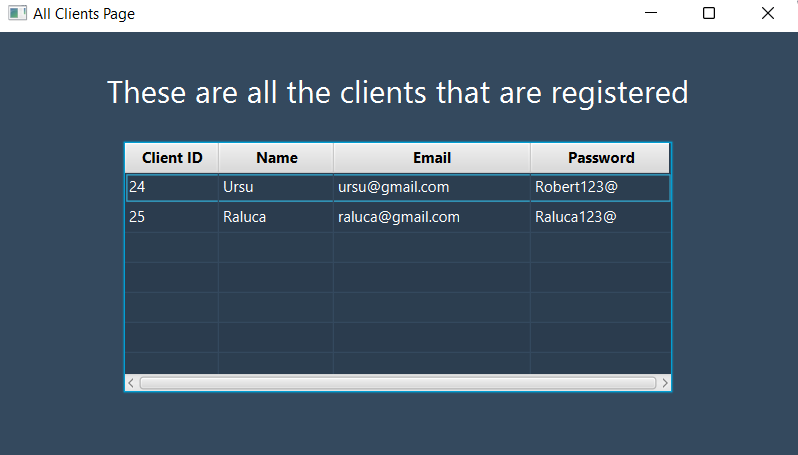


1. **Admin Dashboard:**
   * *Description:*
     + The admin dashboard provides access to administrative functionalities.
   * *Components:*
     + Navigation buttons for managing flights, airlines, and user accounts.
   * *Screenshot:*

* **Flight Management Screen:**
  + *Description:*
    - Allows admins to add, edit, or delete flight details.
  + *Components:*
    - Form fields for entering flight information.
    - Buttons for adding, editing, and deleting flights.
  + *Screenshot:*



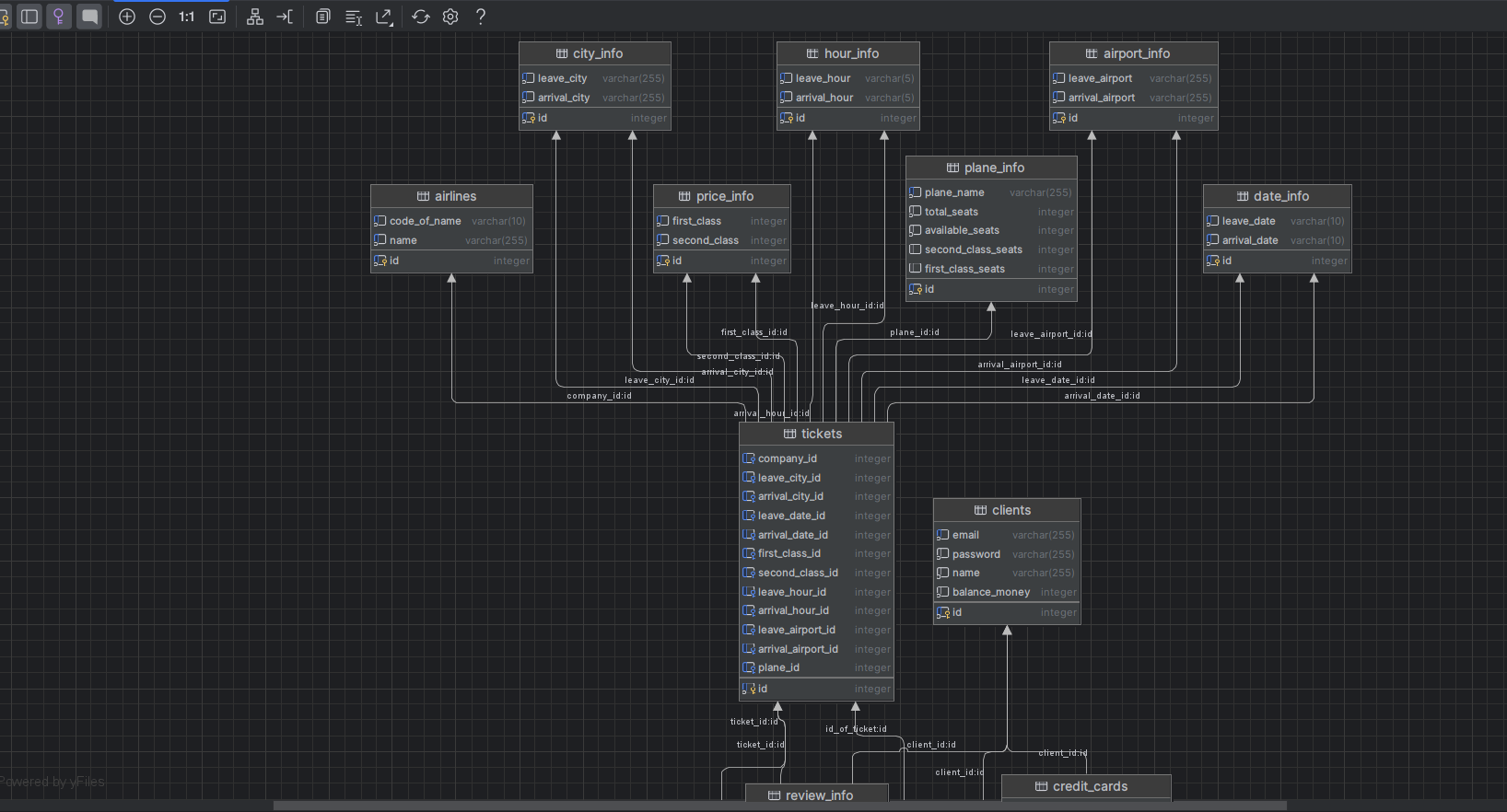


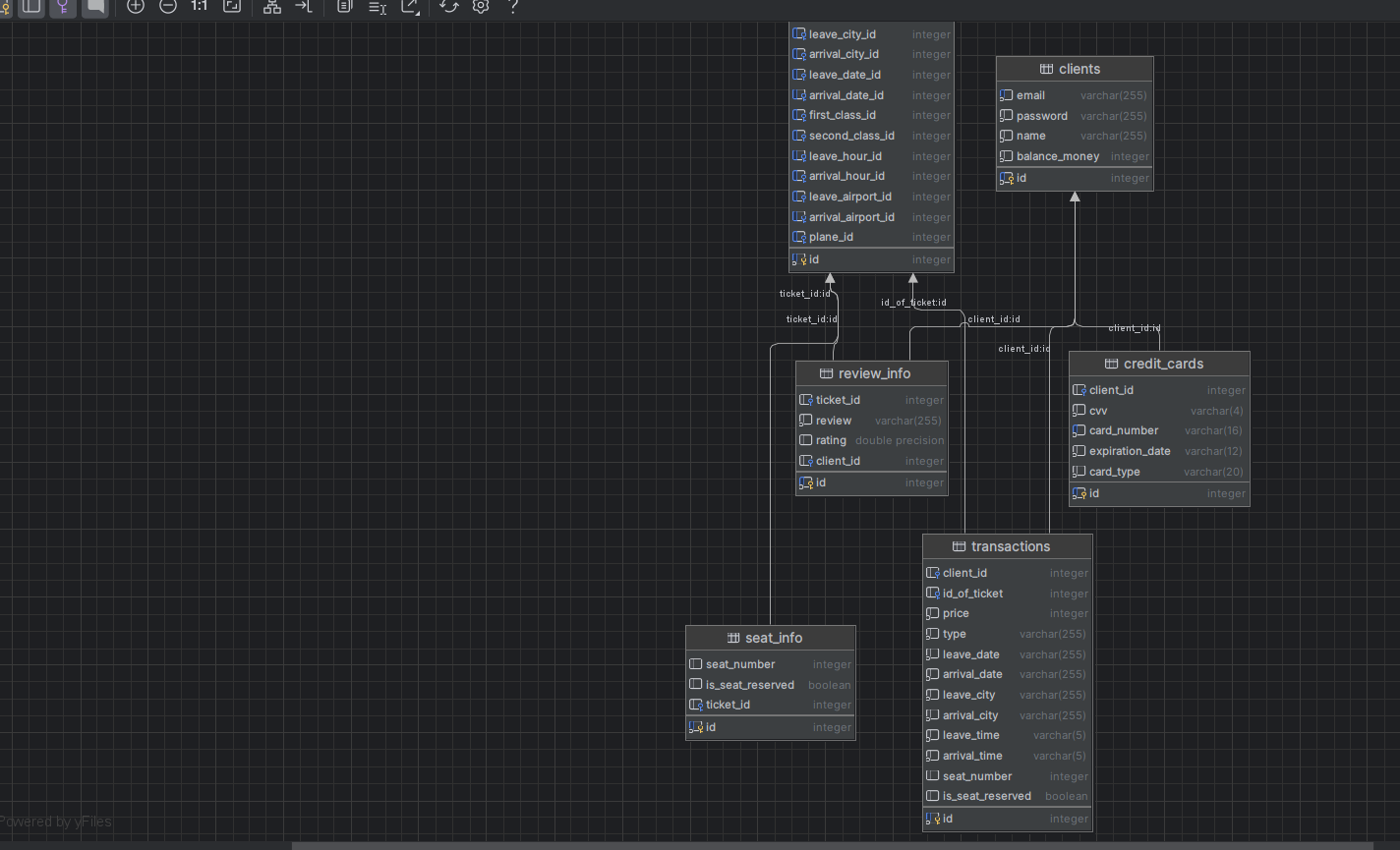


5.  **Database Overview and Integration:**

The PostgreSQL database for the Flight Booking System is designed with a comprehensive schema to efficiently store and manage data related to various aspects of the system. Let's integrate the provided information about tables into the overview:

* **Tables:**
  + **transactions:**
    - Fields: transaction\_id (PK), user\_id (FK), amount, transaction\_date.
  + **ticket\_details:**
    - Fields: ticket\_id (PK), flight\_id (FK), user\_id (FK), seat\_id (FK), class, price\_id (FK), booking\_date, status.
  + **cities:**
    - Fields: city\_id (PK), city\_name.
  + **airlines:**
    - Fields: airline\_id (PK), airline\_code, airline\_name.
  + **airports:**
    - Fields: airport\_id (PK), airport\_name.
  + **planes:**
    - Fields: plane\_id (PK), plane\_name, total\_seats, available\_seats, second\_class\_seats, first\_class\_seats.
  + **prices:**
    - Fields: price\_id (PK), first\_class\_price, second\_class\_price.
  + **seats:**
    - Fields: seat\_id (PK), plane\_id (FK), class, seat\_number, available.
  + **tickets:**
    - Fields: ticket\_id (PK), flight\_id (FK), user\_id (FK), seat\_id (FK), class, price\_id (FK), booking\_date, status.
  + **ticket\_reviews:**
    - Fields: review\_id (PK), ticket\_id (FK), user\_id (FK), rating, review\_text, review\_date.
  + **credit\_cards:**
    - Fields: card\_id (PK), user\_id (FK), card\_number, expiration\_date, cvv.
* **Relationships:**
  + **transactions → users:**
    - One-to-Many relationship (One user can have multiple transactions).
  + **ticket\_details → flights, users, seats, prices:**
    - Many-to-One relationships (Many ticket details can be associated with one flight, one user, one seat, and one price).
  + **cities, airlines, airports, planes → (referenced in) flights:**
    - Many-to-One relationships (Many flights can be associated with one city, one airline, one airport, and one plane).
  + **prices → ticket\_details:**
    - One-to-Many relationship (One price can be associated with multiple ticket details).
  + **seats, planes → (referenced in) ticket\_details:**
    - Many-to-One relationships (Many seats and planes can be associated with one ticket detail).
  + **reviews → tickets, users:**
    - Many-to-One relationships (Many reviews can be associated with one ticket and one user).
  + **credit\_cards → users:**
    - One-to-Many relationship (One user can have multiple credit cards).
* **Data Types:**
  + **Integer:** Used for primary and foreign keys.
  + **Varchar:** Used for storing variable-length character data.
  + **Date:** Used for storing date values.
  + **Numeric/Decimal:** Used for storing monetary values.
  + **Boolean:** Used for representing true/false values.
* **CRUD Operations:**
  + **Create (INSERT):**
    - New transactions, ticket details, cities, airlines, airports, planes, prices, seats, tickets, reviews, and credit cards can be added to their respective tables.
  + **Read (SELECT):**
    - Queries can be executed to retrieve information about transactions, ticket details, cities, airlines, airports, planes, prices, seats, tickets, reviews, and credit cards based on specific criteria.
  + **Update (UPDATE):**
    - Existing records in transactions, ticket details, cities, airlines, airports, planes, prices, seats, tickets, reviews, and credit cards can be modified to reflect changes.
  + **Delete (DELETE):**
    - Records from transactions, ticket details, cities, airlines, airports, planes, prices, seats, tickets, reviews, and credit cards can be deleted based on specific conditions.





6. **Database Connection in Java Application:**

The Flight Booking System application connects to the PostgreSQL database using JDBC (Java Database Connectivity). Below is an explanation of how the application establishes a connection, including details on connection configuration, error handling, and connection security.



**Explanation:**

In the updated code snippet, the database name is set to "tickets\_system," the username is set to "postgres," and the password is set to "12345678" when they are parsed to the function.

**Loading JDBC Driver:**

* + The **Class.forName("org.postgresql.Driver");** statement loads the PostgreSQL JDBC driver. This step is necessary before establishing a connection.

**Establishing Connection:**

* + The **DriverManager.getConnection** method is used to create a connection to the PostgreSQL database. The connection URL includes the database name, host, and port.

**Connection Status:**

* + The application checks whether the connection is successful. If successful, a message is printed to the console. Otherwise, an appropriate message is displayed.

**Exception Handling:**

* + The **try-catch** block handles exceptions that may occur during the connection process. If an error occurs, an error message is printed to the console.

**Connection Configuration:**

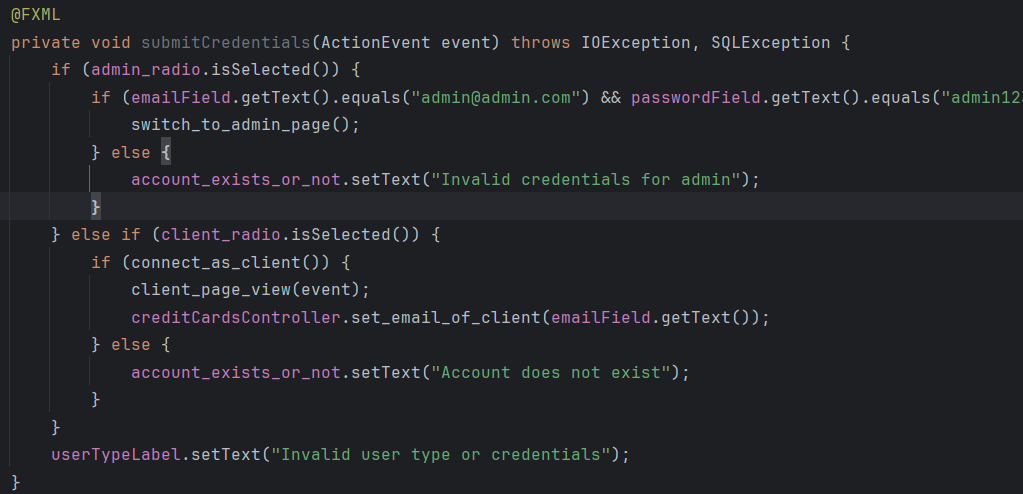
The **connect\_to\_db** method takes parameters such as **db\_name**, **user**, and **password** to dynamically configure the connection based on the provided database name, username, and password.

**Error Handling:**

Exceptions during the connection process, such as a Exception or SQLException, are caught and handled. Error messages are printed to the console for debugging purposes.

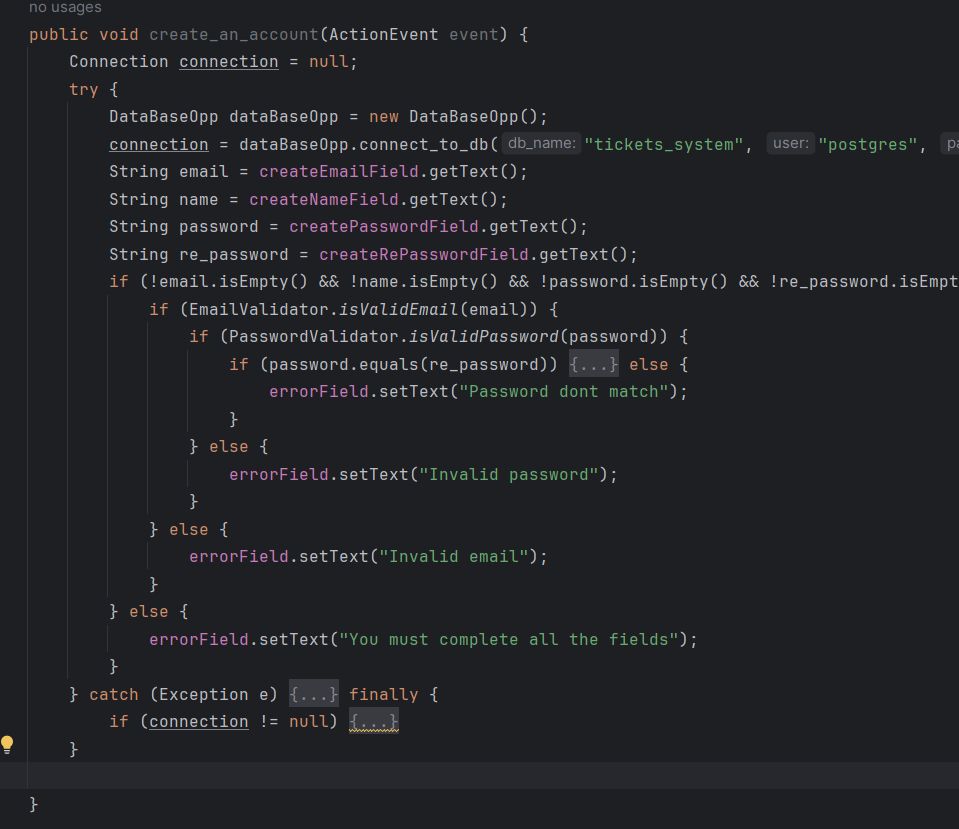
7. **Authentication:**

* **Description:** Users can log in as administrators or clients, gaining access to role-specific functionalities.
* **Code Example:**

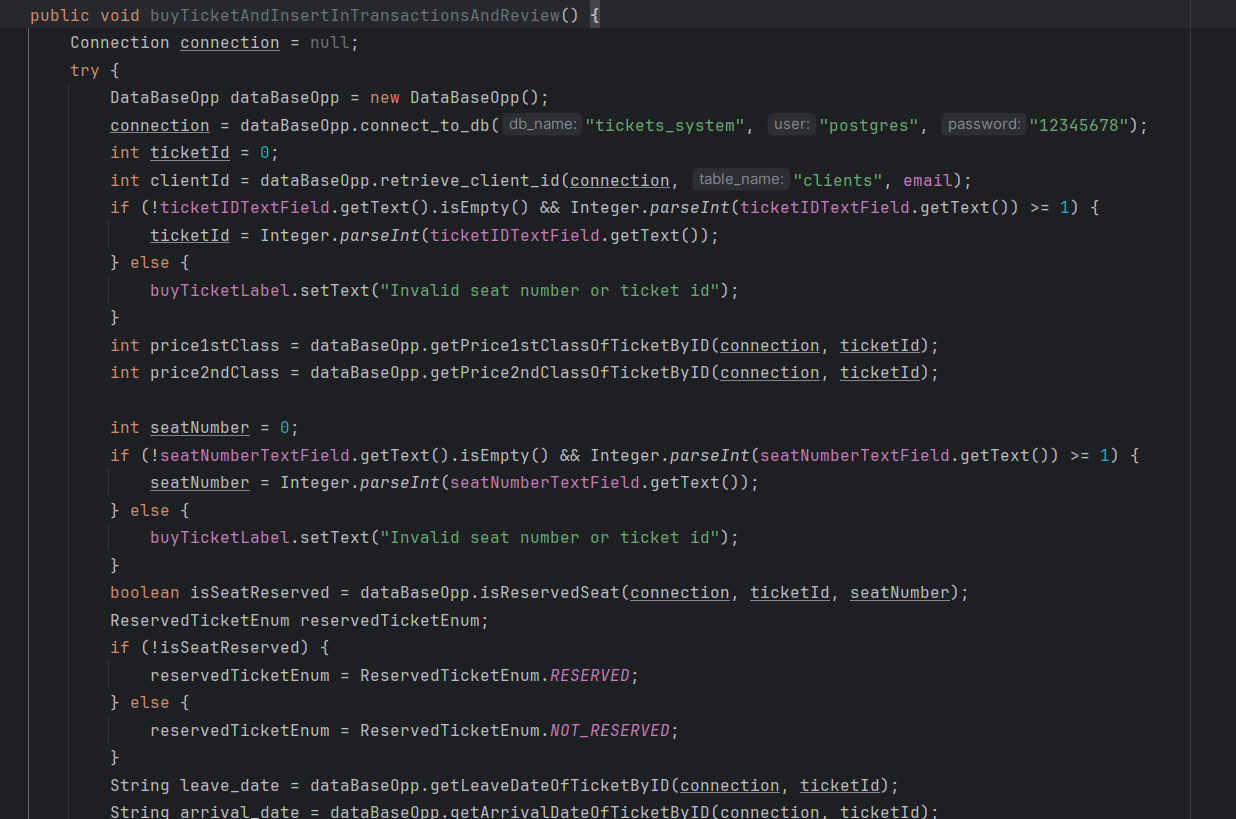
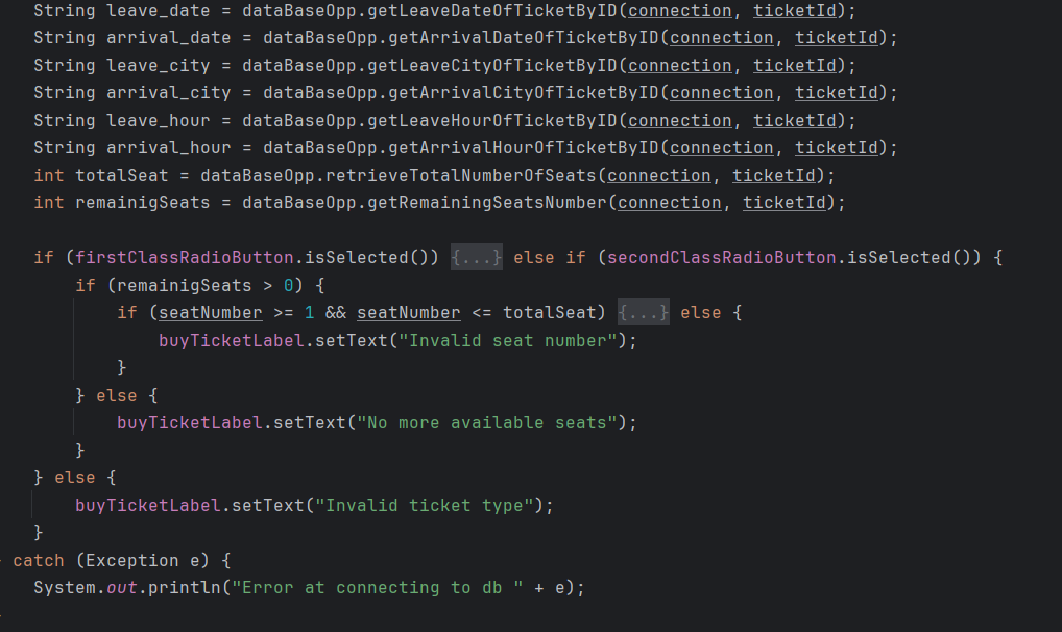


**Create New Account:**

* **Description:** Users can create new accounts to access application functionalities.
* **Code Example:**

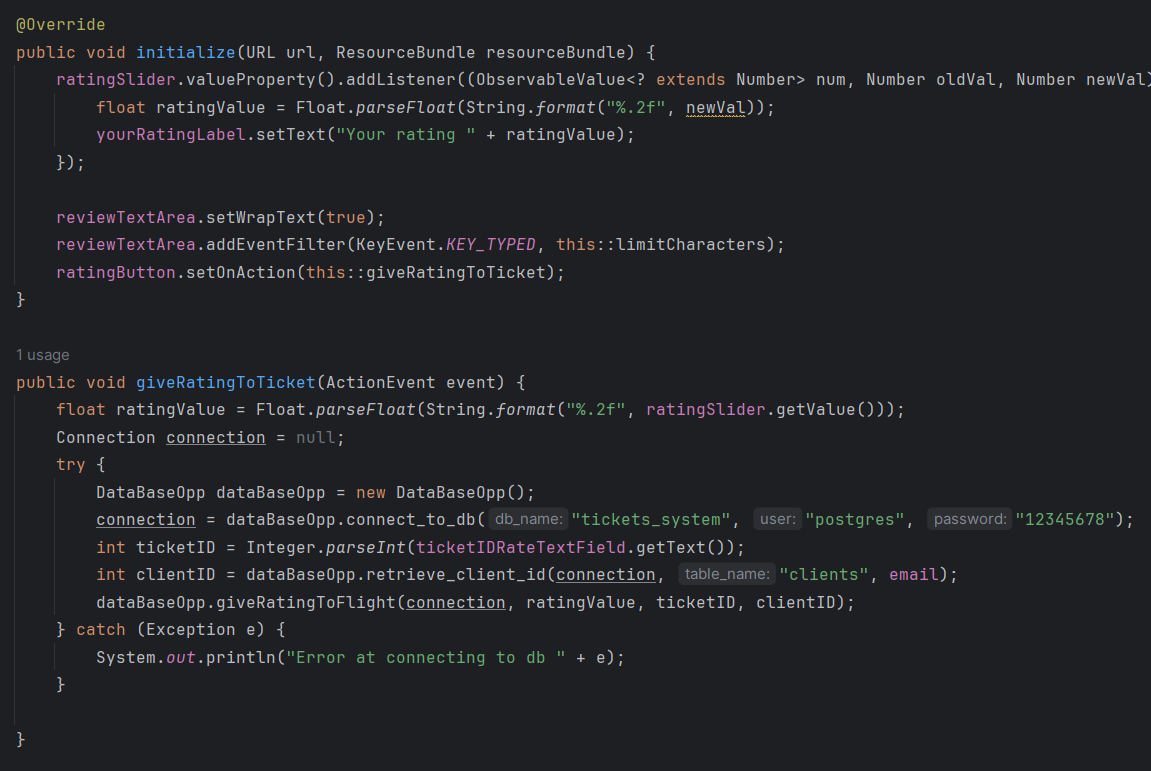


**Purchase Ticket (Client):**

* **Description:** Clients can buy tickets for available flights.
* **Code Example:**
* 
* 

**Add Review (Client):**

* **Description:** Clients can leave reviews and ratings for purchased tickets.
* **Code Example:**



**Ticket Administration (Admin):**

* **Description:** Administrators have access to ticket administration functionalities, including modification and creation.
* **Code Example:**



And many more functionalities, these are just some examples

**8. How to use the app:**

**When the app opens, you must choose a user type from ADMIN or CLIENT and introduce the credentials. The credentials for the admin part are:** [**email:admin@admin.com**](mailto:email->admin@admin.com) **, password:admin123. After you log in as an admin, you can introduce data in the database related to tickets, if they are already in the database, you are goanna see an error message below. “Show Client Details” button is goanna pop up a window where you can see all the clients that are registered, “Create A New Ticket” button is goanna pop up a window where you can see all the ticket related information and you can complete the fields to create a new ticket, if the ticket already exist, you will see an error message below. “Show All Tickets” is goanna pop up a window where you can see all the ticket in the database, if you click on a ticket, you can modify its data. The credentials for the client part are in the database, after you login you are goanna see a page where you can buy tickets by id, give them a rating for dragging the slider by id, you can write down a review by id, you can see your transaction history and your reviews, you can see all the tickets available and you can clear your transaction history. If your balance is too low, you need to update your balance. After you switch to “Update My Balance”, there you can register and delete a credit card, you cand see all your credit cards that you possess, and you can update your balance by completing the field and press “Update Balance”.**

**PS: you have a folder with all the fxml files in case that they are goanna be deleted somehow. I know you needed to have a repository for each table, but when you showed this at the lab, my project was already finished so it was too late to change, I have a repository with all the database interogations.**

**Testing Strategy:**

The testing strategy employed for ensuring code quality and functionality involves a combination of unit testing, integration testing, and manual testing through the graphical user interface (GUI). The testing process is iterative, with continuous testing at each stage of development.

**Unit Testing:**

* **Description:** Unit tests are conducted to verify the correctness of individual components and methods in isolation.
* **Results:** Unit tests were successful in identifying and rectifying several logical errors. Mock objects were utilized to simulate database interactions and ensure that methods perform as expected.

**Integration Testing:**

* **Description:** Integration tests assess the collaboration and interaction between different modules, ensuring seamless functionality.
* **Results:** Integration testing revealed issues with the transition between stages, uncovering null pointer exceptions and inconsistencies in data flow. Corrections were applied to enhance the integration between components.

**GUI Manual Testing:**

* **Description:** Manual testing is performed through the GUI to evaluate the end-user experience, ensuring proper navigation and functionality.
* **Results:** GUI testing uncovered errors related to stage management and database interactions. Null pointer exceptions were traced back to improper handling of user inputs and stage transitions. Corrections were made to enhance user experience and eliminate runtime errors.

**Challenges and Solutions:**

* **Null Pointer Exceptions:**
  + **Issue:** Null pointer exceptions were a common challenge, occurring when accessing uninitialized objects or variables.
  + **Solution:** Null checks were implemented in critical areas, and the initialization sequence was carefully reviewed to ensure variables are properly set.
* **Stage Transition Errors:**
  + **Issue:** Errors related to the transition between GUI stages were identified during GUI testing.
  + **Solution:** The management of stage transitions was revised, and appropriate event handlers were added to ensure smooth navigation between stages.
* **Database Interaction Errors:**
  + **Issue:** Errors related to database interactions resulted in inconsistent data retrieval and updates.
  + **Solution:** Database queries and update statements were carefully reviewed, and error handling mechanisms were enhanced to provide informative feedback in case of database-related issues.