

## **Object Oriented Programming [COMP205]**

---

### **Train Reservation System**

---

**VINAY GAUTAM**  
**2023 UG 000137**  
**B.TECH**  
**Prof Kiran D.C Sir**

### **1.Introduction**

The Train Reservation System (TRS) is a Java-based desktop application meticulously crafted to simulate the railway ticket reservation process, inspired by the vast and intricate railway network of India. This system provides a robust and intuitive platform for passengers to execute critical tasks, including account registration, ticket booking, profile management, PNR status checks, booking history reviews, ticket cancellations, and account balance monitoring. Leveraging Java's object-oriented programming paradigm, the TRS employs Swing for its graphical user interface (GUI) and MySQL for persistent data storage via JDBC, creating a reliable and educational prototype. The system serves a dual purpose: it functions as a simulation tool for railway reservation operations and as a valuable learning resource for developers seeking to master Java, Swing, and database

integration. Its scope encompasses essential functionalities such as user authentication, train search with advanced filtering, ticket booking with concession calculations, and comprehensive session logging, all accessible through a user-friendly GUI. The interface supports light and dark themes to enhance visual comfort and incorporates keyboard navigation to ensure accessibility for users with motor impairments. Designed as a standalone application, the TRS operates locally without requiring internet connectivity or external APIs, making it ideal for controlled environments. Key constraints include the use of Swing for the GUI, MySQL for data storage, and plain-text passwords for simplicity, which renders the system unsuitable for production use but ideal for educational and simulation purposes. The TRS assumes users have a Java Runtime Environment (JRE), MySQL server, and basic GUI familiarity, ensuring ease of deployment on standard hardware.

India's railway system, one of the largest and most complex in the world, connects millions of passengers across thousands of routes, offering diverse coach classes such as AC First Class, AC 2 Tier, Sleeper, and General, alongside concessions like 50% fare reductions for senior citizens or students. Passengers frequently encounter challenges, including navigating intricate train schedules, securing confirmed seats, and understanding dynamic fare structures influenced by quotas (e.g., General, Tatkal) and concessions. The TRS addresses these issues by providing a controlled, simulated environment where users can practice booking tickets, calculate fares with concessions, and manage cancellations without real-world consequences. By excluding integrations with external systems like payment gateways or the IRCTC API, the TRS maintains a focus on simulation, ensuring simplicity while demonstrating core railway operations. The system's design emphasizes extensibility, allowing for future enhancements such as multi-language support, web-based deployment, or integration with live railway data. For instance, developers could extend the GUI to support regional languages like Hindi or Tamil, or adapt the application for cloud-based access using frameworks like Spring. The TRS also includes a comprehensive session logging mechanism to track user actions, facilitating debugging and auditing within the

simulation. This focus on simulation makes the TRS an effective tool for both end-users learning railway processes and developers studying software design patterns.

The TRS has been developed with a clear structure, utilizing a modular architecture that separates concerns into distinct components: GUI management, business logic, and database operations. The GUI, built with Java Swing, employs layouts like BorderLayout and GridBagLayout to ensure consistent and visually appealing interfaces, with features like theme toggling and keyboard navigation enhancing usability. Business logic is encapsulated in classes such as 'User', 'Train', 'Booking', and 'Profile', each handling specific functionalities like authentication, train searches, or fare calculations. Database operations are centralized in the 'DBConnection' class, which uses JDBC to interact with a MySQL database named 'IRCTC\_System'. The database schema includes tables like 'Users', 'Trains', 'Bookings', 'AccountHolder', 'Profiles', and 'SessionLogs', designed with foreign keys to ensure data integrity. The system supports critical features like generating unique PNRs, simulating seat availability (Confirmed, Waiting List, RAC), and applying concessions, all while maintaining transaction consistency for operations like booking and cancellation. Security measures, though limited by the simulation context, include prepared statements to prevent SQL injection and UUID-based session IDs for tracking, with plain-text passwords noted as a limitation. The TRS is designed for single-user operation, running on commodity hardware (4GB RAM, 1GHz processor) with a localhost MySQL connection, ensuring accessibility for educational settings.

This requirements document represents a comprehensive effort to define the TRS's specifications, covering both functional and non-functional requirements to guide development, testing, and future enhancements. Functional requirements detail features like user registration, login, train search, booking, and cancellation, each

with specific inputs, outputs, and database interactions. Non-functional requirements address performance (e.g., 2-second GUI response time), usability (e.g., accessible navigation), and maintainability (e.g., modular code with Javadoc). The document also outlines external interfaces, including Swing-based GUI components, MySQL connectivity, and hardware requirements, ensuring clarity for developers. Extensive work has been done to specify the system's architecture, including class designs, database schemas, and setup instructions, with sample data (10 trains, 2 accounts) to facilitate testing. Constraints such as Swing-only GUI, MySQL storage, and single-user operation are clearly defined, alongside assumptions about user familiarity and software dependencies (JDK 8+, MySQL 5.7+). Future enhancements, such as multi-user support, password hashing, or web deployment, are proposed to extend the system's capabilities. This detailed specification ensures the TRS is a robust, extensible prototype that effectively simulates railway reservation processes while serving as an educational tool for developers.

## 2. General Code Requirements

### 2.1 Programming Language and Environment

- **Language:** Develop using Java (JDK 8 or higher), adhering to object-oriented principles (encapsulation, inheritance, polymorphism) for modularity and maintainability.
- **GUI Framework:** Use Java Swing to create a desktop-based graphical user interface, ensuring cross-platform compatibility.
- **Database:** Utilize MySQL (version 5.7 or higher) for persistent data storage, accessed through JDBC with MySQL Connector/J (version 8.0.27).
- **Development Environment:** Ensure code compiles and executes on Windows, macOS, or Linux systems with JRE installed.
- **Dependencies:** Include **mysql-connector-java-8.0.27.jar** in the classpath to enable database connectivity.

### 2.2 System Architecture

- **Modular Design:** Organize the codebase into distinct modules:
  - **GUI:** Managed by **TrainManagementSystem** for interface rendering and navigation.
  - **Business Logic:** Handled by classes like User, Train, Booking, and Profile for core functionalities.
  - **Database Operations:** Centralized in DBConnection for all database interactions.
- **Class Hierarchy:**
  - User: Manages user authentication, registration, password recovery, and logout.
  - Train: Handles train data storage and search operations.
  - Booking: Manages ticket booking, cancellation, PNR status queries, and booking history.
  - Profile: Stores and updates passenger profile information.
  - AccountHolder: Tracks user account balances for fare payments.
  - DBConnection: Provides database connectivity, query execution, and session logging.
- **Design Patterns:**
  - Use the Singleton pattern for DBConnection to ensure a single database connection instance.
  - Implement the MVC (Model-View-Controller) pattern to separate GUI (View), business logic (Model), and navigation (Controller).

## 2.3 Database Schema

- Database Name: IRCTC\_System.
- Tables:
  - Users: Stores user credentials and security details.
    - Columns: user\_id (PK, auto-increment, INT), name (VARCHAR(100)), mobile (VARCHAR(15)), email (VARCHAR(100), UNIQUE), password (VARCHAR(100)), security\_question (VARCHAR(200)), security\_answer (VARCHAR(200)).
  - Trains: Stores train schedules and route information.

- Columns: train\_id (PK, auto-increment, INT), train\_number (VARCHAR(10)), train\_name (VARCHAR(100)), source\_station (VARCHAR(100)), destination\_station (VARCHAR(100)), departure\_time (TIME), arrival\_time (TIME), duration (VARCHAR(10)), runs\_on (VARCHAR(50)).
  - Bookings: Stores ticket booking details.
    - Columns: booking\_id (PK, auto-increment, INT), user\_id (FK to Users, INT), train\_number (VARCHAR(10)), train\_name (VARCHAR(100)), from\_station (VARCHAR(100)), to\_station (VARCHAR(100)), journey\_date (DATE), class\_type (VARCHAR(50)), passenger\_name (VARCHAR(100)), age (INT), gender (VARCHAR(10)), berth\_preference (VARCHAR(20)), fare (DECIMAL(10,2)), pnr\_number (VARCHAR(10), UNIQUE), booking\_status (VARCHAR(20)), seat\_number (VARCHAR(10)).
  - AccountHolder: Manages user account balances.
    - Columns: account\_holder\_id (PK, auto-increment, INT), name (VARCHAR(100)), age (INT), account\_number (VARCHAR(20), UNIQUE), balance (DECIMAL(10,2)), interest (DECIMAL(5,2)), user\_id (FK to Users, INT).
  - Profiles: Stores passenger profile data.
    - Columns: profile\_id (PK, auto-increment, INT), user\_id (FK to Users, INT), age (INT), gender (VARCHAR(10)), preferred\_class (VARCHAR(50)), gov\_id (VARCHAR(50)), emergency\_contact (VARCHAR(15)), wheelchair\_access (BOOLEAN), medical\_condition (BOOLEAN).
  - SessionLogs: Logs user actions for auditing.
    - Columns: log\_id (PK, auto-increment, INT), user\_id (FK to Users, INT, NULLABLE), action (VARCHAR(200)), timestamp (DATETIME), session\_id (VARCHAR(36)).
- Constraints:
  - Enforce foreign key relationships (e.g., user\_id in Bookings, Profiles) with ON DELETE CASCADE.
  - Use prepared statements for all database queries to prevent SQL injection.

- Ensure unique constraints on email (Users), pnr\_number (Bookings), and account\_number (AccountHolder).

## 2.4 Installation and Setup

- Dependencies: Require JDK 8+, MySQL Server 5.7+, and MySQL Connector/J 8.0.27.
- Setup Instructions:
  - Install JDK and MySQL Server.
  - Configure MySQL with database IRCTC\_System and credentials (e.g., username: root, password: root).
  - Add mysql-connector-java-8.0.27.jar to the project classpath.
  - Compile code: `javac -cp ".;mysql-connector-java-8.0.27.jar" *.java`.
  - Run application: `java -cp ".;mysql-connector-java-8.0.27.jar" TrainManagementSystem`.
- Sample Data: Implement `DBConnection.initializeDatabase()` to create tables and insert sample data (e.g., 10 trains, 2 account holders) on first run.

## 3. Functional Requirements

### 3.1 Main Menu & Navigation (FR-MAIN)

- Objective: Provide a centralized GUI hub for accessing all passenger features with intuitive navigation and theme customization.
- Description: The main menu is a JFrame using BorderLayout, displaying options (Login, Register, Forgot Password, Theme Toggle) before login and features (Book Ticket, Booking History, Cancel Ticket, PNR Status, Manage Profile, Account Balance, Logout) after login. Navigation occurs via JButton clicks, with actions logged in SessionLogs. The GUI includes a header (JLabel for title, date, user info), footer (JLabel for customer care details), and dynamic content area (JPanel). Support light (white background, black text) and dark (gray background, white text) themes, toggled via a button. Ensure accessibility with keyboard navigation (Tab key).
- Inputs: Button clicks (e.g., Login, Book Ticket).
- Outputs: Display feature-specific panels; log navigation in SessionLogs; update theme.

- **Classes:**
  - TrainManagementSystem: Manages frame, panel transitions, and themes.
    - Attributes: User currentUser, JPanel currentPanel, boolean darkTheme, JLabel userInfoLabel.
    - Methods: applyTheme(), showLoginPanel(), showMainMenu(), updateHeader().
  - DBConnection: Logs actions.
    - Methods: logSession(String action, int userId, String sessionId).
- **Database Tables:**
  - SessionLogs: Stores navigation logs (log\_id, user\_id, action, timestamp, session\_id).
- **Constraints:**
  - Use Swing with BorderLayout for main frame and GridBagLayout for panels.
  - Log all navigation actions with timestamps and session IDs.
  - Support theme toggling with recursive component color updates.
  - Ensure keyboard accessibility for all buttons.
- **Priority:** High
- **Dependencies:** FR-PLOGIN, FR-REG, FR-FPASS, FR-DASH, NFR-UI

### 3.2 Passenger Registration (FR-REG)

- **Objective:** Allow new users to create accounts with secure data storage and linked account holder records.
- **Description:** Provide a registration form (JPanel, GridBagLayout) for users to input name, mobile number, email, password, security question, and security answer. Validate inputs (non-empty fields, unique email, valid mobile format). Store user data in Users, create an AccountHolder record with a default balance of ₹10,000 and random account number, generate a UUID session ID, and log the action in SessionLogs. Display success or error messages using JOptionPane.
- **Inputs:** Name (text), mobile number (text, 10–15 digits), email (text, valid format), password (text, min 6 characters), security question (dropdown), security answer (text).



- **Outputs:** Insert user and account holder records; generate session ID; log registration; show success/error dialog.
- **Classes:**
  - User: Handles registration logic.
    - Attributes: int userId, String name, mobile, email, password, securityQuestion, securityAnswer, sessionId.
    - Methods: register(String name, String mobile, String email, String password, String question, String answer), generateSessionId().
  - DBConnection: Manages database operations.
    - Methods: getConnection(), logSession(), initializeDatabase().
- **Database Tables:**
  - Users: Stores user data.
  - AccountHolder: Stores account details (default balance ₹10,000).
  - SessionLogs: Logs registration action.
- **Constraints:**
  - Validate non-empty fields and unique email via database check.
  - Store plain-text passwords (noted as insecure for production).
  - Generate unique session ID using UUID.
  - Create AccountHolder with random 10-digit account number.
  - Use prepared statements for secure inserts.
- **Priority:** High
- **Dependencies:** NFR-DATA, NFR-SEC

### 3.3 Passenger Login (FR-PLOGIN)

- **Objective:** Authenticate users to access personalized features and enforce profile completion.
- **Description:** Provide a login form (JPanel, GridBagLayout) with email and password fields. Verify credentials against Users, generate a UUID session ID, log the attempt in SessionLogs, and check Profiles for completeness. If the profile is incomplete, redirect to the profile panel; otherwise, display the dashboard. Show error dialogs for invalid credentials using JOptionPane.
- **Inputs:** Email (text), password (text).

- Outputs: Verify credentials; generate session ID; redirect to profile/dashboard; log attempt; show error dialog.
- Classes:
  - User: Manages authentication.
    - Attributes: int userId, String email, password, sessionId.
    - Methods: login(String email, String password), hasCompleteProfile().
  - DBConnection: Logs sessions.
    - Methods: logSession().
- Database Tables:
  - Users: Verifies credentials.
  - Profiles: Checks profile completion.
  - SessionLogs: Logs login attempts (success/failure).
- Constraints:
  - Use plain-text password verification.
  - Redirect to profile panel if no Profiles entry exists.
  - Generate UUID session ID for tracking.
  - Ensure keyboard accessibility for form fields and buttons.
- Priority: High
- Dependencies: FR-PROFILE, FR-DASH, NFR-SEC, NFR-DATA

### 3.4 Forgot Password (FR-FPASS)

- Objective: Enable secure account recovery using security questions.
- Description: Provide a recovery form (JPanel, GridBagLayout) where users input their email to retrieve their security question, provide an answer, and set a new password. Verify the answer (case-insensitive), update the password in Users, and log the attempt in SessionLogs. Display success or error messages via JOptionPane.
- Inputs: Email (text), security answer (text), new password (text), confirm password (text).
- Outputs: Retrieve security question; verify answer; update password; log attempt; show success/error dialog.
- Classes:
  - User: Handles recovery logic.

- Attributes: String email, securityQuestion, securityAnswer.
  - Methods:           getSecurityQuestion(String           email),  
                  verifySecurityAnswer(String   answer),   resetPassword(String  
                  newPassword).
- DBConnection: Logs recovery attempts.
  - Methods: logSession().
- Database Tables:
  - Users: Stores security question/answer and password.
  - SessionLogs: Logs recovery attempts.
- Constraints:
  - Verify security answer case-insensitively.
  - Store plain-text passwords.
  - Validate email existence and password match.
  - Log all recovery attempts with timestamps.
  - Ensure keyboard accessibility for form fields.
- Priority: Medium
- Dependencies: NFR-SEC, NFR-DATA

### 3.5 Passenger Dashboard (FR-DASH)

- Objective: Provide a central hub for accessing all passenger features post-login.
- Description: Display a dashboard (JPanel, GridBagLayout) with buttons for Search Trains, Book Ticket, Booking History, Cancel Ticket, PNR Status, Manage Profile, Account Balance, and Logout. Button clicks navigate to corresponding feature panels, with actions logged in SessionLogs. Ensure accessibility with keyboard navigation.
- Inputs: Button clicks (e.g., Search Trains, Logout).
- Outputs: Display feature panels; log navigation actions.
- Classes:
  - TrainManagementSystem: Manages dashboard UI.
    - Attributes: JPanel currentPanel, User currentUser.
    - Methods:    showMainMenu(),    showTrainSearchPanel(),  
                  showBookingPanel(), etc.
  - DBConnection: Logs actions.

- Methods: logSession().
- Database Tables:
  - SessionLogs: Logs dashboard interactions.
- Constraints:
  - Use GridBagLayout for button alignment.
  - Ensure consistent theme application (light/dark).
  - Log all button clicks with user ID and timestamp.
  - Support keyboard navigation for accessibility.
- Priority: High
- Dependencies: FR-SEARCH, FR-BOOK, FR-HIST-VIEW, FR-CANCEL, FR-PNR, FR-PROFILE, FR-BALANCE, FR-PLOGOUT, NFR-UI

### 3.6 Passenger Profile Management (FR-PROFILE)

- Objective: Allow users to manage personal and accessibility details for booking eligibility.
- Description: Provide a profile form (JPanel, GridBagLayout) for inputting age, gender, preferred class, government ID, emergency contact, wheelchair access (checkbox), and medical condition (checkbox). Validate inputs (age 0–120, non-empty fields), store data in Profiles, and log the action in SessionLogs. Load existing profiles for editing. Display success/error messages via JOptionPane.
- Inputs: Age (spinner, 0–120), gender (dropdown: Male, Female, Other), preferred class (dropdown: AC First, Sleeper, etc.), government ID (text), emergency contact (text, 10–15 digits), wheelchair access (boolean), medical condition (boolean).
- Outputs: Store/load profile data; log action; show success/error dialog.
- Classes:
  - Profile: Manages profile data.
    - Attributes: int userId, age, String gender, preferredClass, govId, emergencyContact, boolean wheelchairAccess, medicalCondition.
    - Methods: save(), getProfile().
  - DBConnection: Handles database operations.
    - Methods: logSession().

- Database Tables:
  - Profiles: Stores profile data.
  - SessionLogs: Logs profile updates.
- Constraints:
  - Validate age (0–120) and mandatory fields (gender, preferred class).
  - Support accessibility options (wheelchair, medical condition).
  - Use prepared statements for secure inserts/updates.
  - Ensure keyboard accessibility for form fields.
- Priority: High
- Dependencies: NFR-DATA, NFR-UI

### 3.7 Train Search & Filtering (FR-SEARCH)

- Objective: Enable users to find trains based on travel criteria with flexible filters.
- Description: Provide a search form (JPanel, GridBagLayout) for source, destination, journey date, class type, train type, and time range. Validate inputs (valid date format, existing stations), query Trains, check running days, and display results (train number, name, times, duration, availability) in a scrollable JTextArea with a Book button. Simulate availability (Available, Waiting List, RAC). Log searches in SessionLogs.
- Inputs: Source (dropdown), destination (dropdown), journey date (text, dd/MM/yyyy), class type (dropdown: AC First, Sleeper, etc.), train type (dropdown: Rajdhani, Shatabdi, etc.), time range (dropdown: 00:00–06:00, 06:00–12:00, etc.).
- Outputs: Display train list with availability; log search action.
- Classes:
  - Train: Handles train data and search logic.
    - Attributes: String trainNumber, trainName, sourceStation, destinationStation, departureTime, arrivalTime, duration, runsOn.
    - Methods: searchTrainsWithFilters(String source, String dest, String date, String classType, String trainType, String timeRange).
  - DBConnection: Manages queries.

■ Methods: logSession().

- Database Tables:
  - Trains: Stores train data.
  - SessionLogs: Logs search actions.
- Constraints:
  - Validate date format (dd/MM/yyyy) and running days against runs\_on.
  - Simulate availability randomly for each class.
  - Use prepared statements for secure queries.
  - Support partial filters (e.g., only source/destination).
  - Ensure scrollable results with JScrollPane.
- Priority: High
- Dependencies: NFR-DATA, NFR-UI

### 3.8 Seat Booking & Confirmation (FR-BOOK)

- Objective: Facilitate ticket booking with accurate fare calculation and seat assignment.
- Description: Provide a booking form (JPanel, GridBagLayout) for selecting a train, journey date, class, quota (General, Tatkal, etc.), and passenger details (name, age, gender, berth preference, concession: Senior Citizen, Student, None). Calculate fares based on class and concessions (e.g., 50% for Senior Citizen), deduct from AccountHolder balance, assign seats (or WL/RAC), generate a unique PNR (PNR + 6 digits), and store in Bookings. Use transactions for balance deduction and booking insertion. Log actions in SessionLogs and display PNR/fare via JOptionPane.
- Inputs: Train number (text, read-only), name (text), source/destination (text, read-only), journey date (text, read-only), class type (dropdown), quota (dropdown), passenger details (name, age [spinner, 0–120], gender [dropdown], berth preference [dropdown: Lower, Upper, etc.], concession [dropdown]).
- Outputs: Deduct fare; generate PNR and seat number; store booking; log action; show PNR/fare or error dialog.
- Classes:
  - Booking: Manages booking logic.

- Attributes: Map<String, Double> CLASS\_FARES (e.g., AC First: ₹2000, Sleeper: ₹500), Map<String, Double> CONCESSION\_RATES (e.g., Senior Citizen: 0.5), Passenger (nested class: name, age, gender, berthPreference, concessionType).
  - Methods: bookTickets(Train train, String date, String classType, String quota, List<Passenger> passengers), generatePNR(), convertDate(String date).
- DBConnection: Handles balance and logging.
  - Methods: deductBalance(int userId, double amount), logSession(), getAccountBalance(int userId).
- Database Tables:
  - Bookings: Stores booking details.
  - AccountHolder: Updates balance.
  - SessionLogs: Logs booking actions.
- Constraints:
  - Support concessions (e.g., 50% for Senior Citizen, 25% for Student).
  - Validate inputs (non-empty fields, sufficient balance).
  - Use transactions for atomic balance deduction and booking insertion.
  - Generate unique PNR (e.g., PNR123456).
  - Simulate seat assignment (random seat or WL/RAC status).
  - Support multiple passengers per booking.
- Priority: High
- Dependencies: FR-SEARCH, NFR-DATA, NFR-CONC, NFR-SEC

### 3.9 Booking History Recording (FR-HIST-REC)

- Objective: Automatically store booking details for reference and auditing.
- Description: After successful booking, insert booking data (PNR, train, passenger, fare, status) into Bookings within a transaction, logging the action in SessionLogs. This backend process is transparent to users and supports history viewing.
- Inputs: Booking data from bookTickets() (train, passengers, fare, PNR, etc.).
- Outputs: Insert booking record; log action.
- Classes:

- Booking: Handles recording.
  - Methods: bookTickets() (includes insertion logic).
- DBConnection: Manages inserts.
  - Methods: logSession().
- Database Tables:
  - Bookings: Stores booking details.
  - SessionLogs: Logs recording action.
- Constraints:
  - Use transactions to ensure atomic insertion.
  - Store all booking details (e.g., booking\_date as current timestamp).
  - Log action with booking ID and timestamp.
- Priority: High
- Dependencies: FR-BOOK, NFR-DATA, NFR-CONC

### 3.10 View Booking History (FR-HIST-VIEW)

- Objective: Display past bookings in a scrollable, readable format.
- Description: Provide a history panel (JPanel, GridBagLayout) to display booking details (PNR, train, stations, date, class, passenger, status, fare) sorted by booking date (descending). Query Bookings by user ID, display results in a scrollable JTextArea, and log the action in SessionLogs. Include a Back button to return to the dashboard.
- Inputs: User ID (from session).
- Outputs: Display booking history; log view action.
- Classes:
  - Booking: Fetches history.
    - Methods: getBookingHistory(int userId).
  - DBConnection: Manages queries.
    - Methods: logSession().
- Database Tables:
  - Bookings: Stores history data.
  - SessionLogs: Logs view actions.
- Constraints:
  - Sort results by booking date (descending).
  - Use scrollable JTextArea with JScrollPane.



- Ensure keyboard accessibility for navigation.
- Format output clearly (e.g., tabular layout in text).
- Priority: Medium
- Dependencies: FR-HIST-REC, NFR-DATA, NFR-UI

### 3.11 Ticket Cancellation (FR-CANCEL)

- Objective: Allow users to cancel confirmed bookings with automated refunds.
- Description: Provide a cancellation form (JPanel, GridBagLayout) for inputting a PNR. Verify the PNR and confirmed status in Bookings, update status to “Cancelled,” refund the fare to AccountHolder, and log the action in SessionLogs. Use transactions for status update and refund. Display success/error messages via JOptionPane.
- Inputs: PNR number (text).
- Outputs: Update booking status; refund fare; log action; show success/error dialog.
- Classes:
  - Booking: Manages cancellation.
    - Methods: cancelTicket(String pnr).
  - DBConnection: Handles refunds and logging.
    - Methods: refundBalance(int userId, double amount), logSession().
- Database Tables:
  - Bookings: Updates booking\_status.
  - AccountHolder: Updates balance.
  - SessionLogs: Logs cancellation actions.
- Constraints:
  - Cancel only confirmed bookings (status = “Confirmed”).
  - Use transactions for atomic status update and refund.
  - Refund full fare to balance.
  - Validate PNR existence and ownership.
- Priority: Medium
- Dependencies: FR-BOOK, FR-HIST-REC, NFR-DATA, NFR-CONC

### 3.12 PNR Status Query (FR-PNR)

- Objective: Enable users to check booking details and status by PNR.
- Description: Provide a PNR query form (JPanel, GridBagLayout) for inputting a PNR. Query Bookings by pnr\_number, retrieve details (PNR, train, stations, date, class, passenger, status, seat, fare), and display in a JTextArea with JScrollPane. Log the query in SessionLogs. Show error dialogs for invalid PNRs via JOptionPane.
- Inputs: PNR number (text).
- Outputs: Display booking details; log query; show error dialog for invalid PNR.
- Classes:
  - Booking: Fetches PNR data.
    - Methods: getPNRStatus(String pnr).
  - DBConnection: Manages queries.
    - Methods: logSession().
- Database Tables:
  - Bookings: Stores booking details.
  - SessionLogs: Logs query actions.
- Constraints:
  - Validate PNR existence and ownership.
  - Use readable JTextArea with JScrollPane.
  - Ensure keyboard accessibility for input and navigation.
- Priority: Medium
- Dependencies: FR-BOOK, FR-HIST-REC, NFR-DATA, NFR-UI

### 3.13 Account Balance Management (FR-BALANCE)

- Objective: Allow users to view their account balance for booking payments.
- Description: Provide a balance panel (JPanel, GridBagLayout) to display the current balance from AccountHolder in a read-only JTextField (formatted as ₹). Query the balance by user ID, log the action in SessionLogs, and show errors (e.g., query failure) via JOptionPane. Include a Back button to return to the dashboard.
- Inputs: User ID (from session).

- Outputs: Display balance; log view action; show error dialog if query fails.
- Classes:
  - DBConnection: Fetches balance.
    - Methods: getAccountBalance(int userId).
  - TrainManagementSystem: Displays balance.
    - Methods: showAccountBalance().
- Database Tables:
  - AccountHolder: Stores balance.
  - SessionLogs: Logs view actions.
- Constraints:
  - Display balance in INR format (e.g., ₹10,000.00).
  - Use read-only JTextField.
  - Ensure keyboard accessibility for navigation.
- Priority: Medium
- Dependencies: FR-BOOK, FR-CANCEL, NFR-DATA, NFR-UI

### 3.14 Passenger Logout (FR-PLOGOUT)

- Objective: Securely terminate user sessions.
- Description: Provide a Logout button on the dashboard. On click, clear the session ID, redirect to the login panel, and log the action in SessionLogs. Ensure no residual session data remains.
- Inputs: Logout button click.
- Outputs: Clear session ID; redirect to login panel; log action.
- Classes:
  - User: Manages logout.
    - Attributes: String sessionId.
    - Methods: logout().
  - DBConnection: Logs logout.
    - Methods: logSession().
- Database Tables:
  - SessionLogs: Logs logout actions.
- Constraints:
  - Invalidate session ID completely.
  - Redirect to login panel using showLoginPanel().

- Log action with user ID and timestamp.
- Ensure keyboard accessibility for Logout button.
- Priority: High
- Dependencies: NFR-SEC, NFR-DATA

### 3.15 Application Exit (FR-EXIT)

- Objective: Gracefully terminate the application, releasing resources.
- Description: Handle window close events (EXIT\_ON\_CLOSE) to release resources (e.g., database connections). If the user is logged in, log the termination in SessionLogs. Ensure no data loss during shutdown.
- Inputs: Window close action (e.g., clicking the close button).
- Outputs: Release resources; close database connections; log termination (if logged in).
- Classes:
  - TrainManagementSystem: Handles exit.
    - Attributes: None specific.
    - Methods: None specific (uses setDefaultCloseOperation(EXIT\_ON\_CLOSE)).
  - DBConnection: Closes connections.
    - Methods: getConnection() (implicit cleanup via driver).
- Database Tables:
  - SessionLogs: Logs termination (if logged in).
- Constraints:
  - Ensure clean shutdown without data loss.
  - Release all database connections.
  - Log termination only if user is logged in.
- Priority: Low
- Dependencies: NFR-DATA

## 4. Future Enhancements

- Implement multi-user support with concurrent session management.
- Add password hashing (e.g., BCrypt) and encryption for production use.
- Support multi-language GUI (e.g., Hindi, Tamil) using resource bundles.
- Enable web-based deployment with Java frameworks (e.g., Spring, Vaadin).

- Integrate real payment gateways and IRCTC APIs for live data.
- Support dynamic station updates and real-time seat availability

## 5.ER Diagram

### 5.1. Users

- **Primary Key:** user\_id
- **Attributes:** name, mobile, email (Unique), password, security\_question, security\_answer
- **Purpose:** Stores user credentials and basic security information.

### 5.2. AccountHolder

- **Primary Key:** account\_holder\_id
- **Foreign Key:** user\_id (linked to Users)
- **Attributes:** name, age, account\_number (Unique), balance, interest
- **Purpose:** Stores extended information about users who hold an account, including financial details.

### 5.3. Profiles

- **Primary Key:** profile\_id
- **Foreign Key:** user\_id (linked to Users)
- **Attributes:** age, gender, preferred\_class, gov\_id, emergency\_contact, wheelchair\_access, medical\_condition
- **Purpose:** Holds passenger-specific information which may be useful for travel or booking preferences.

### 5.4. SessionLogs

- **Primary Key:** log\_id

- **Foreign Key:** `user_id` (linked to **Users**)
- **Attributes:** `action`, `timestamp`, `session_id`
- **Purpose:** Logs user activities such as login/logout or booking actions, helpful for auditing or security.

## 5.5. Trains

- **Primary Key:** `train_id`
- **Attributes:** `train_number` (Unique), `train_name`, `source_station`, `destination_station`, `departure_time`, `arrival_time`, `duration`, `runs_on`
- **Purpose:** Contains schedule and route details for each train.

## 5.6. Bookings

- **Primary Key:** `booking_id`
- **Foreign Keys:** `user_id` (from **Users**), `train_number` (from **Trains**)
- **Attributes:**
  - Train Details: `train_name`, `from_station`, `to_station`, `journey_date`, `class_type`
  - Passenger Info: `passenger_name`, `age`, `gender`, `berth_preference`
  - Booking Info: `fare`, `pnr_number` (Unique), `booking_status`, `seat_number`, `booking_date`
- **Purpose:** Stores actual ticket bookings, linking users with train journeys

## Relationships Summary

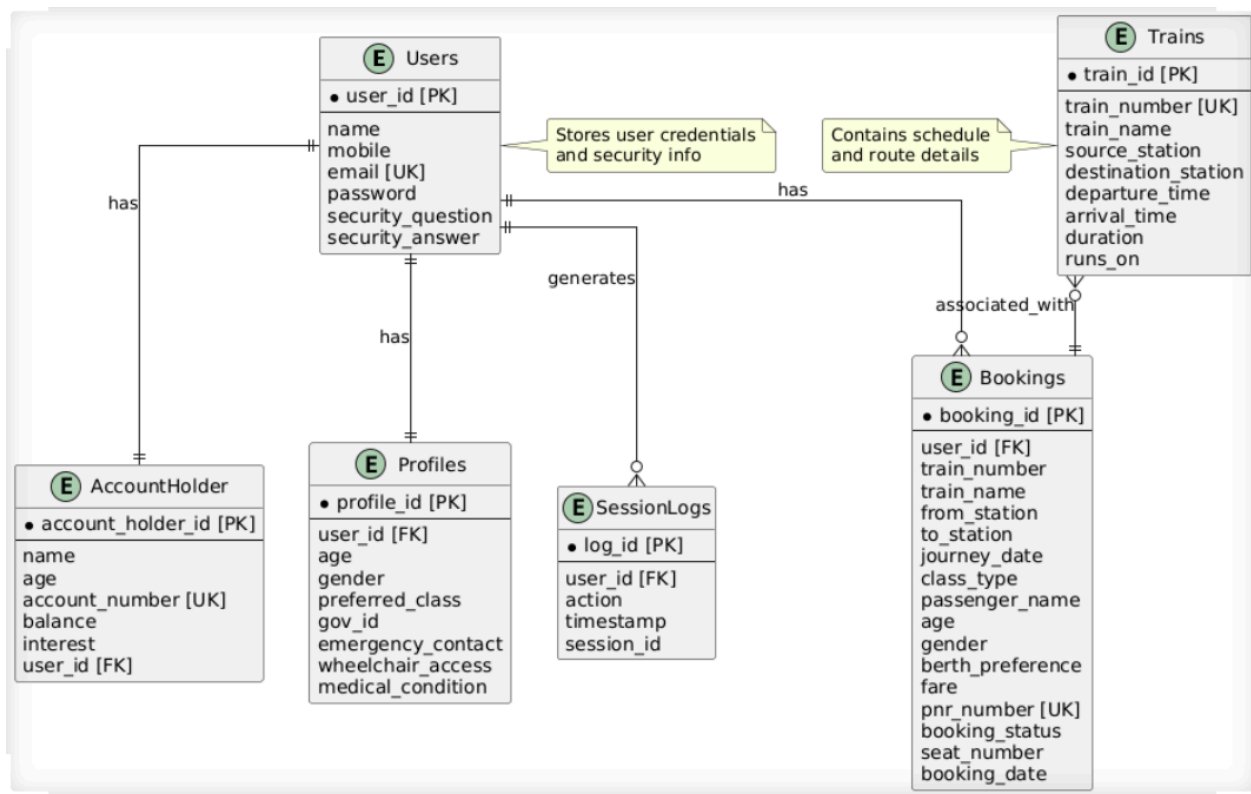
- **Users** → **Profiles**: One-to-many (a user can have multiple profiles).
- **Users** → **AccountHolder**: One-to-one or one-to-many (a user can have an account).
- **Users** → **SessionLogs**: One-to-many (a user can generate multiple sessions).

- **Users** → **Bookings**: One-to-many (a user can make multiple bookings).
- **Trains** → **Bookings**: One-to-many (a train can have multiple bookings).

## Use Case Overview

This database supports:

- User registration and login with secure credentials.
- Creation of travel profiles and account holders.
- Logging user sessions for security/auditing.
- Management of train schedules.
- Bookings linked to users and trains with detailed ticket and passenger data.



## 7. User Case Diagram

### 1. User (Primary Actor)

The **User** represents a typical end-user who utilizes the train management system to perform various tasks related to train travel. This actor interacts with most of the system functionalities. Here's a breakdown of what the User can do:

- **Register Account:** New users can create an account on the system.
- **Login:** Existing users must log in to access most features. This is a prerequisite for several other use cases (marked by «**extends**»).
- **Reset Password:** Users can reset their password if they forget it.
- **Search Trains:** Allows users to look for available trains between selected stations (requires login).
- **View Booking History:** Displays past bookings (requires login).
- **Cancel Ticket:** Enables cancellation of booked tickets (requires login).
- **Check PNR Status:** Users can check the status of their ticket (requires login).
- **Manage Profile:** Lets users update their personal details.
- **View Account Balance:** Users can see their wallet or account balance (requires login).
- **Book Tickets:** Core function allowing users to reserve seats on trains (requires login and includes payment processing).
  - **Process Payment:** This is handled by the **Payment System** (external actor), indicating integration with a third-party payment service.



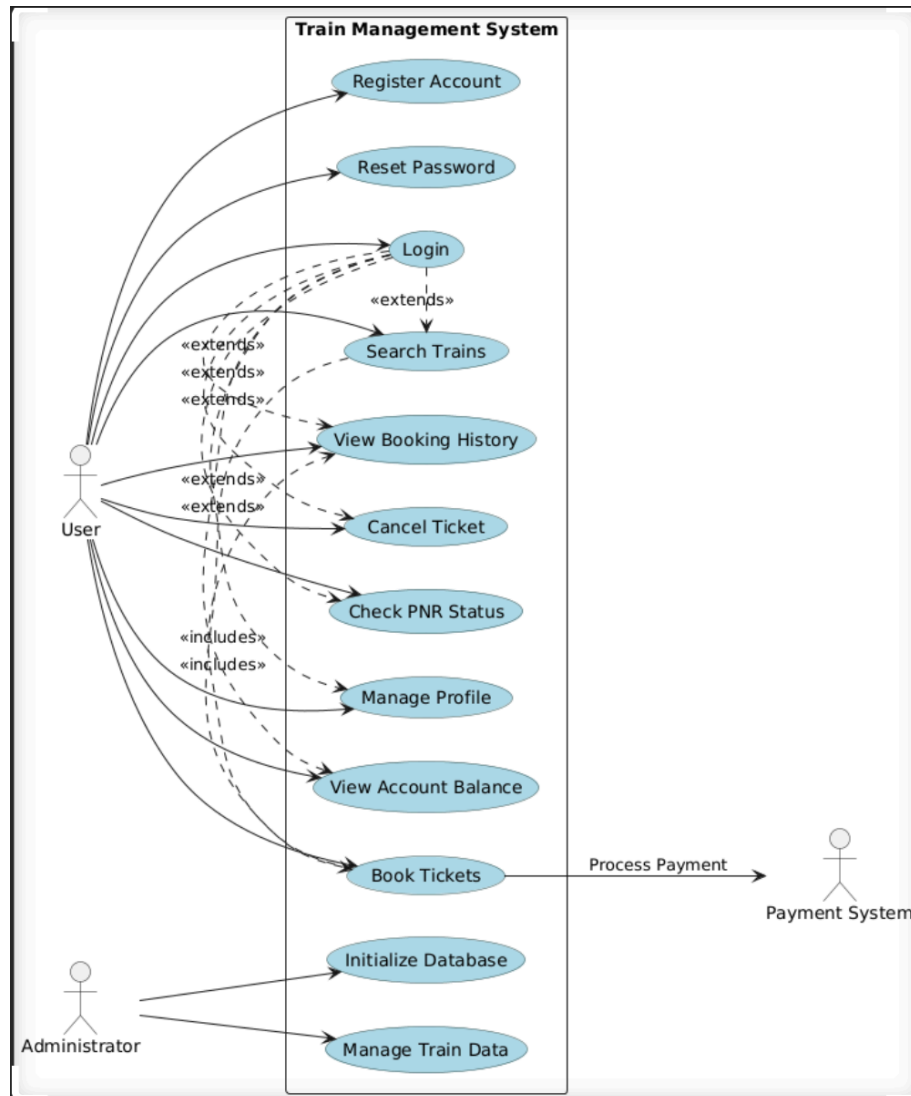
## 2. Administrator(Actor)

The **Administrator** is responsible for the backend management of the system. This actor has access to technical and managerial tasks, such as:

### Use case

- **Initialize Database:** Likely used during the initial system setup or major updates.
- **Manage Train Data:** Includes adding, updating, or removing train schedules, routes, and availability.

### Use Case Diagram



## 8. Class Diagram

### 1. Class Diagram Overview

The Train Management System architecture is structured into four main layers:

#### a. Database Layer (DBConnection)

- Responsibility:** Centralizes all database-related operations such as connections, table initialization, and financial transactions.

#### b. User & Profile Management (User, Profile)

- **Responsibility:** Manages user accounts, authentication, and profile details.

### **c. Train & Booking Management (Train, Booking, Passenger)**

- **Responsibility:** Handles all train-related data, availability, and ticket bookings.

### **d. GUI Layer (TrainManagementSystem)**

- **Responsibility:** Provides the graphical user interface for user interaction.

## **2. Class Descriptions**

### **2.1 Database Layer – DBConnection**

- **Purpose:** Acts as the gateway to the database for all system modules.
- **Key Methods:**
  - `getConnection()`: Establishes and returns a connection to the MySQL database.
  - `initializeDatabase()`: Creates necessary tables such as Users, Trains, and Bookings.
  - `logSession()`: Records user activity for auditing or analysis.
  - `deductBalance()` / `refundBalance()`: Manages financial transactions.
- **Relationships:**
  - Accessed by User, Train, and Booking classes for persistent data operations.

### **2.2 User & Profile Management**

## User Class

- **Purpose:** Manages user registration, login, and account actions.
- **Attributes:**
  - `userId`, `name`, `email`, `password`, `securityQuestion`, `securityAnswer`
- **Key Methods:**
  - `register()`: Registers a new user and stores their credentials.
  - `login()`: Validates user login credentials.
  - `resetPassword()`: Provides password recovery using security questions.
- **Relationships:**
  - **One-to-One** with `Profile` (composition).
  - **One-to-Many** with `Booking` (a user can have multiple bookings).

## Profile Class

- **Purpose:** Contains extended personal details of the user.
- **Attributes:**
  - `age`, `gender`, `preferredClass`, etc.
- **Key Methods:**
  - `save()`: Saves or updates the profile in the database.
  - `getProfile()`: Retrieves the user's profile information.

## 2.3 Train & Booking Management

### Train Class

- **Purpose:** Represents train data and handles train searches and availability.

- **Attributes:**
  - trainNumber, sourceStation, destinationStation, departureTime, etc.
- **Key Methods:**
  - searchTrainsWithFilters(): Retrieves trains based on source, destination, date, and class.
  - getAvailabilityForClass(): Checks seat availability by class.

## **Booking Class**

- **Purpose:** Manages ticket reservations, cancellations, and booking records.
- **Static Data:**
  - CLASS\_TYPES, QUOTA\_TYPES: Lists of available travel classes and booking quotas.
- **Key Methods:**
  - bookTickets(): Books tickets for users and saves booking data.
  - cancelTicket(): Cancels a booking and initiates a refund.
  - getPNRStatus(): Retrieves the status of a booking using PNR.

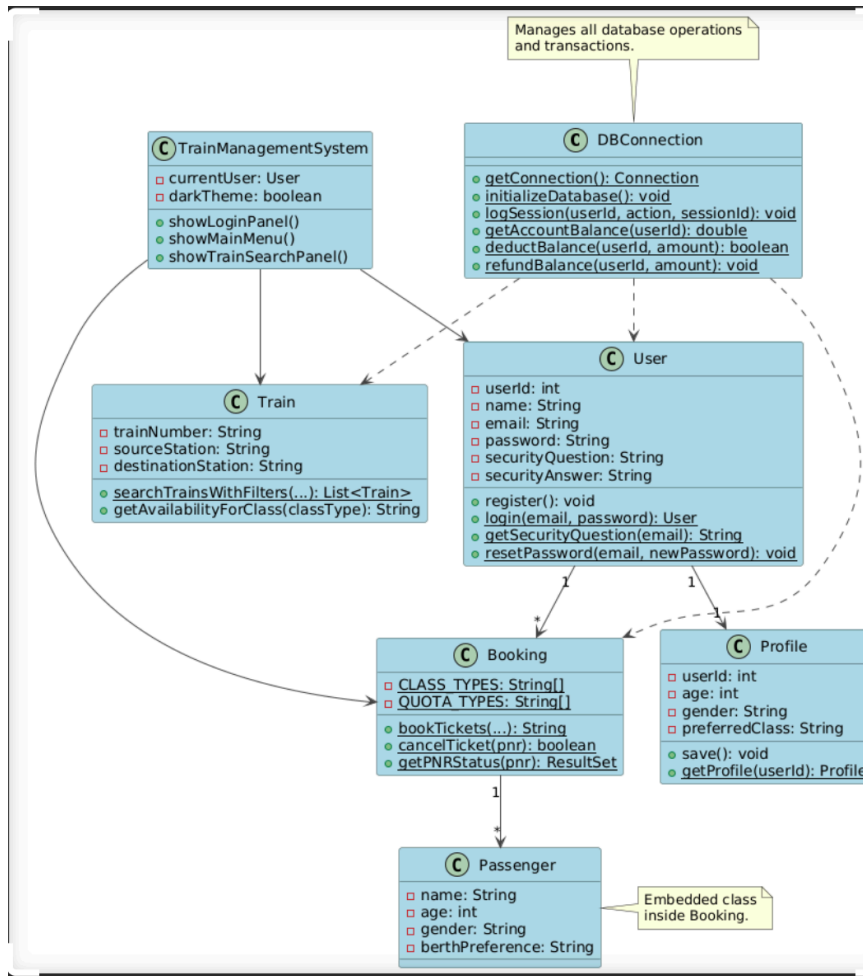
## **Passenger Class**

- **Purpose:** Captures details of individual passengers within a booking.
- **Attributes:**
  - name, age, gender, berthPreference
- **Note:** Typically embedded or associated with the Booking class.

## **2.4 GUI Layer – TrainManagementSystem**

- **Purpose:** Provides user-facing screens and orchestrates user interaction with the system.
- **Attributes:**
  - `currentUser`: Tracks the currently logged-in user.
  - `darkTheme`: Stores user preference for UI theme.
- **Key Methods:**
  - `showLoginPanel()`: Displays the login interface.
  - `showMainMenu()`: Loads the main navigation interface.
  - `showTrainSearchPanel()`: Renders the train search functionality.
- **Relationships:**
  - Communicates with `User`, `Train`, and `Booking` classes to support user workflows.

## 8.2.Diagram



## 9. Sequence Diagram

### 9.1. Functional Requirements

#### 9.1.1 User Authentication

- The system must identify the currently logged-in user before allowing any booking operations.

#### 9.1.2 Train Search Functionalit.

- Users must be able to search trains based on:
  - Source station
  - Destination station

- Travel date
  - Travel class (e.g., Sleeper, AC)
- The system should fetch and display a list of available trains matching the criteria.

### **9.1.3 Ticket Booking**

- Users should be able to select a train from the search results and proceed to book tickets.
- The system should collect passenger details (name, age, gender, berth preference).

### **9.1.4 Payment Processing**

- Before confirming the booking, the system must:
  - Deduct the appropriate fare from the user's account.
  - Check if the user has sufficient balance.
  - Proceed only if the payment is successful.
- If the balance is insufficient or any payment error occurs, the booking must be canceled and the user notified with an error message.

### **9.1.5 Booking Confirmation**

- Upon successful payment:
  - The booking details should be stored in the database.
  - A unique PNR number should be generated and shared with the user.
  - The system must display a confirmation message along with the PNR.

### **9.1.6 Session Logging**

- Every successful booking transaction must be logged for audit and tracking purposes.

## **9.2. Non-Functional Requirements**

### **2.1 Performance**



- Train search and booking confirmation must be processed within a few seconds to ensure a smooth user experience.

### 9.2.2 Security

- User account data and transaction processes must be secure and encrypted.
- Prevent unauthorized access to user details and booking records.

### 2.3 Reliability

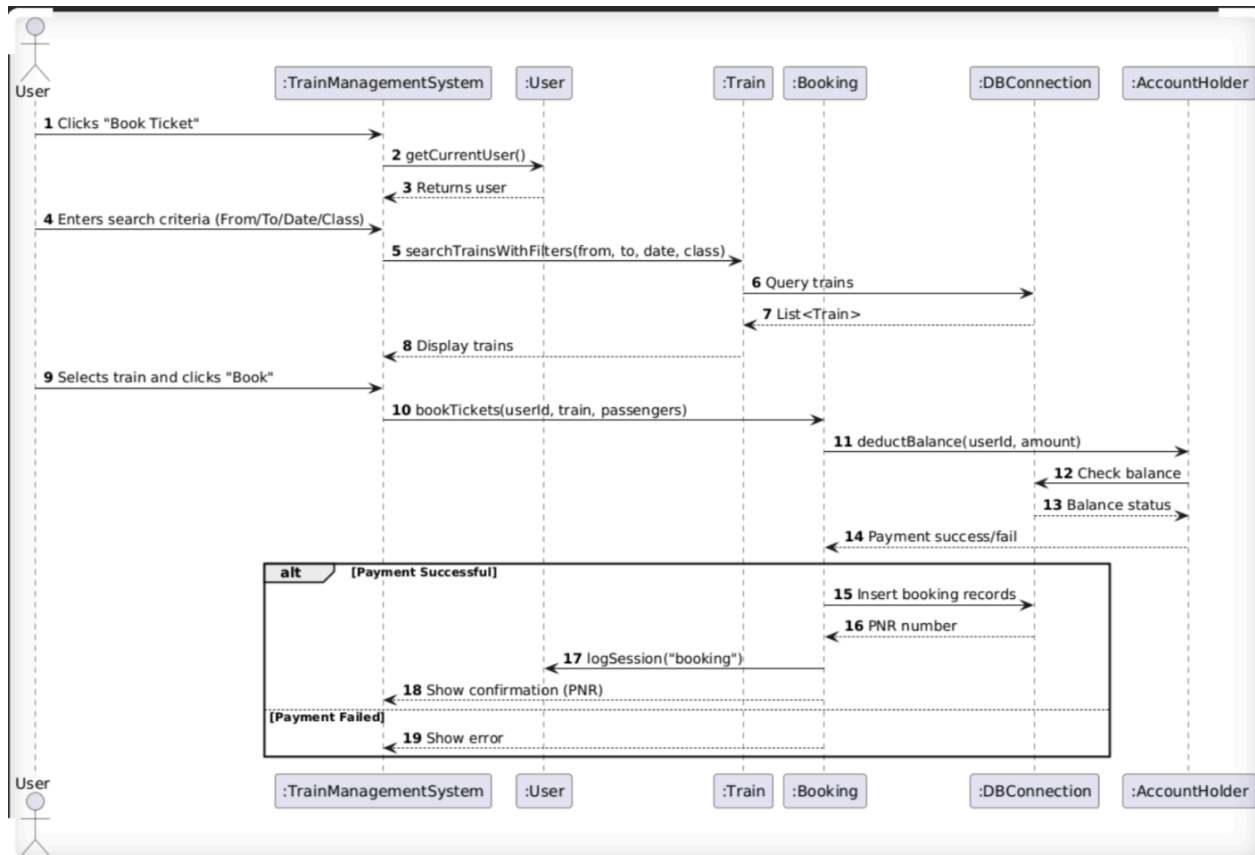
- The system should handle both success and failure scenarios gracefully without data corruption.
- Partial failures (like payment issues) should not create inconsistent booking records.

### 2.4 Usability

- The interface must clearly guide the user through each step of the booking process.
- Success and failure messages should be user-friendly and informative.

### 9.3. System Component Interactions

- **User Interface (TrainManagementSystem):** Orchestrates the interaction between the user and backend logic.
- **User Module:** Validates and provides user-related information.
- **Train Module:** Handles train filtering and availability queries.
- **Booking Module:** Manages ticket bookings, cancellations, and PNR generation.
- **DBConnection:** Executes all data operations, including transactions and session logs.
- **AccountHolder:** Verifies balance and manages fund deduction or refunds.



## 10.Code

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.sql.*;
import java.text.*;
import java.util.*;
import java.util.List;

```

```

class DBConnection {
    private static final String URL = "jdbc:mysql://localhost:3306/IRCTC_System";
    private static final String USER = "root";
    private static final String PASSWORD = "root"; // Replace with your MySQL
password

```

```

public static Connection getConnection() throws SQLException {
    return DriverManager.getConnection(URL, USER, PASSWORD);
}

```

```

public static void initializeDatabase() {
    try (Connection conn = getConnection()) {
        String createUsersTable = "CREATE TABLE IF NOT EXISTS Users (" +
            "user_id INT AUTO_INCREMENT PRIMARY KEY, " +
            "name VARCHAR(100) NOT NULL, " +
            "mobile VARCHAR(15) NOT NULL, " +
            "email VARCHAR(100) UNIQUE NOT NULL, " +
            "password VARCHAR(100) NOT NULL, " +
            "security_question VARCHAR(200) NOT NULL, " +
            "security_answer VARCHAR(100) NOT NULL)";

        String createTrainsTable = "CREATE TABLE IF NOT EXISTS Trains (" +
            "train_id INT AUTO_INCREMENT PRIMARY KEY, " +
            "train_number VARCHAR(10) UNIQUE NOT NULL, " +
            "train_name VARCHAR(100) NOT NULL, " +
            "source_station VARCHAR(100) NOT NULL, " +
            "destination_station VARCHAR(100) NOT NULL, " +
            "departure_time VARCHAR(10) NOT NULL, " +
            "arrival_time VARCHAR(10) NOT NULL, " +
            "duration VARCHAR(10) NOT NULL, " +
            "runs_on VARCHAR(50) NOT NULL)";

        String createBookingsTable = "CREATE TABLE IF NOT EXISTS
Bookings (" +
            "booking_id INT AUTO_INCREMENT PRIMARY KEY,
            " +
            "user_id INT NOT NULL, " +
            "train_number VARCHAR(10) NOT NULL, " +
            "train_name VARCHAR(100) NOT NULL, " +
            "from_station VARCHAR(100) NOT NULL, " +

```

```

        "to_station VARCHAR(100) NOT NULL, " +
        "journey_date DATE NOT NULL, " +
        "class_type VARCHAR(50) NOT NULL, " +
        "passenger_name VARCHAR(100) NOT NULL, " +
        "age INT NOT NULL, " +
        "gender VARCHAR(10) NOT NULL, " +
        "berth_preference VARCHAR(20), " +
        "fare DECIMAL(10,2) NOT NULL, " +
        "pnr_number VARCHAR(15) UNIQUE NOT NULL, " +
        "booking_status VARCHAR(20) DEFAULT 'Confirmed',
    " +
        "seat_number VARCHAR(10), " +
        "booking_date TIMESTAMP DEFAULT
CURRENT_TIMESTAMP, " +
        "FOREIGN KEY (user_id) REFERENCES
Users(user_id)";

```

```

String createAccountHolderTable = "CREATE TABLE IF NOT EXISTS
AccountHolder (" +
        "account_holder_id INT AUTO_INCREMENT
PRIMARY KEY, " +
        "name VARCHAR(100), " +
        "age INT, " +
        "account_number VARCHAR(20) UNIQUE NOT
NULL, " +
        "balance DECIMAL(15, 2), " +
        "interest DECIMAL(5, 2), " +
        "user_id INT, " +
        "FOREIGN KEY (user_id) REFERENCES
Users(user_id)";

```

```

String createProfilesTable = "CREATE TABLE IF NOT EXISTS Profiles
(" +
        "profile_id INT AUTO_INCREMENT PRIMARY KEY, "
+

```

```

        "user_id INT NOT NULL, " +
        "age INT, " +
        "gender VARCHAR(10), " +
        "preferred_class VARCHAR(50), " +
        "gov_id VARCHAR(20), " +
        "emergency_contact VARCHAR(15), " +
        "wheelchair_access BOOLEAN DEFAULT FALSE, " +
        "medical_condition BOOLEAN DEFAULT FALSE, " +
        "FOREIGN KEY (user_id) REFERENCES
Users(user_id))";

```

```

String createSessionLogsTable = "CREATE TABLE IF NOT EXISTS
SessionLogs (" +
        "log_id INT AUTO_INCREMENT PRIMARY KEY, " +
        "user_id INT, " +
        "action VARCHAR(100) NOT NULL, " +
        "timestamp TIMESTAMP DEFAULT
CURRENT_TIMESTAMP, " +
        "session_id VARCHAR(50))";

```

```

Statement stmt = conn.createStatement();
stmt.execute(createUsersTable);
stmt.execute(createTrainsTable);
stmt.execute(createBookingsTable);
stmt.execute(createAccountHolderTable);
stmt.execute(createProfilesTable);
stmt.execute(createSessionLogsTable);

```

```

String checkTrains = "SELECT COUNT(*) FROM Trains";
ResultSet rs = stmt.executeQuery(checkTrains);
rs.next();
if (rs.getInt(1) == 0) {
    insertSampleTrainData(conn);
}

```

```

String checkAccounts = "SELECT COUNT(*) FROM AccountHolder";
rs = stmt.executeQuery(checkAccounts);
rs.next();
if (rs.getInt(1) == 0) {
    insertSampleAccountHolderData(conn);
}

} catch (SQLException e) {
    e.printStackTrace();
    JOptionPane.showMessageDialog(null, "Database initialization failed: " +
e.getMessage(),
                                "Database Error", JOptionPane.ERROR_MESSAGE);
}
}

```

```

private static void insertSampleTrainData(Connection conn) throws
SQLException {
    String sql = "INSERT INTO Trains (train_number, train_name,
source_station, destination_station, " +
                "departure_time, arrival_time, duration, runs_on) VALUES (?, ?, ?, ?,
?, ?, ?, ?)";
    PreparedStatement stmt = conn.prepareStatement(sql);

    Object[][] trains = {
        {"12001", "Shatabdi Express", "New Delhi", "Bhopal", "08:10", "16:25",
"08:15", "Daily"},
        {"12309", "Rajdhani Express", "New Delhi", "Kolkata", "17:00", "10:55",
"17:55", "Daily"},
        {"12621", "Tamil Nadu Express", "New Delhi", "Chennai", "22:30",
"07:10", "32:40", "Daily"},
        {"12295", "Durgam Express", "Mumbai", "Kolkata", "17:15", "18:50",
"25:35", "Mon,Wed,Fri"},
        {"12055", "Jan Shatabdi", "Dehradun", "New Delhi", "15:20", "21:10",
"05:50", "Daily"},
    };
}

```

```

        {"12951", "Rajdhani Express", "Mumbai", "Delhi", "16:35", "08:15",
"15:40", "Daily"},
        {"12952", "Rajdhani Express", "Delhi", "Mumbai", "16:55", "08:30",
"15:35", "Daily"},
        {"12301", "Duronto Express", "Howrah", "Delhi", "22:05", "08:00",
"09:55", "Tue,Thu,Sat"},
        {"12302", "Duronto Express", "Delhi", "Howrah", "22:45", "08:40",
"09:55", "Mon,Wed,Fri"},
        {"12305", "Duronto Express", "Sealdah", "Delhi", "22:50", "09:40",
"10:50", "Daily"}
    };

```

```

    for (Object[] train : trains) {
        for (int i = 0; i < train.length; i++) {
            stmt.setString(i + 1, (String) train[i]);
        }
        stmt.addBatch();
    }
    stmt.executeBatch();
}

```

```

private static void insertSampleAccountHolderData(Connection conn) throws
SQLException {

```

```

    String sql = "INSERT INTO AccountHolder (name, age, account_number,
balance, interest, user_id) " +

```

```

        "VALUES (?, ?, ?, ?, ?, ?)";

```

```

    PreparedStatement stmt = conn.prepareStatement(sql);

```

```

    Object[][] accounts = {

```

```

        {"John Doe", 30, "ACC1001", 50000.00, 3.5, null},

```

```

        {"Jane Smith", 25, "ACC1002", 75000.00, 3.5, null}

```

```

    };

```

```

    for (Object[] account : accounts) {

```

```

        stmt.setString(1, (String) account[0]);

```

```

        stmt.setInt(2, (Integer) account[1]);
        stmt.setString(3, (String) account[2]);
        stmt.setDouble(4, (Double) account[3]);
        stmt.setDouble(5, (Double) account[4]);
        stmt.setObject(6, account[5], Types.INTEGER);
        stmt.addBatch();
    }
    stmt.executeBatch();
}

```

```

public static void logSession(int userId, String action, String sessionId) throws
SQLException {
    try (Connection conn = getConnection()) {
        String sql = "INSERT INTO SessionLogs (user_id, action, session_id)
VALUES (?, ?, ?)";
        PreparedStatement stmt = conn.prepareStatement(sql);
        stmt.setInt(1, userId);
        stmt.setString(2, action);
        stmt.setString(3, sessionId);
        stmt.executeUpdate();
    }
}

```

```

public static double getAccountBalance(int userId) throws SQLException {
    try (Connection conn = getConnection()) {
        String sql = "SELECT balance FROM AccountHolder WHERE user_id =
?";
        PreparedStatement stmt = conn.prepareStatement(sql);
        stmt.setInt(1, userId);
        ResultSet rs = stmt.executeQuery();
        if (rs.next()) {
            return rs.getDouble("balance");
        }
        return 0.0;
    }
}

```



```
}
```

```
    public static boolean deductBalance(int userId, double amount) throws
SQLException {
    try (Connection conn = getConnection()) {
        conn.setAutoCommit(false);
        try {
            String checkSql = "SELECT balance FROM AccountHolder WHERE
user_id = ?";
            PreparedStatement checkStmt = conn.prepareStatement(checkSql);
            checkStmt.setInt(1, userId);
            ResultSet rs = checkStmt.executeQuery();

            if (rs.next()) {
                double currentBalance = rs.getDouble("balance");
                if (currentBalance >= amount) {
                    String updateSql = "UPDATE AccountHolder SET balance =
balance - ? WHERE user_id = ?";
                                                                    PreparedStatement updateStmt =
conn.prepareStatement(updateSql);
                    updateStmt.setDouble(1, amount);
                    updateStmt.setInt(2, userId);
                    updateStmt.executeUpdate();
                    conn.commit();
                    return true;
                }
            }
            conn.rollback();
            return false;
        } catch (SQLException e) {
            conn.rollback();
            throw e;
        } finally {
            conn.setAutoCommit(true);
        }
    }
}
```

```

    }
}

    public static void refundBalance(int userId, double amount) throws
SQLException {
    try (Connection conn = getConnection()) {
        String sql = "UPDATE AccountHolder SET balance = balance + ? WHERE
user_id = ?";
        PreparedStatement stmt = conn.prepareStatement(sql);
        stmt.setDouble(1, amount);
        stmt.setInt(2, userId);
        stmt.executeUpdate();
    }
}
}

```

```

class User {
    private int userId;
    private String name;
    private String mobile;
    private String email;
    private String password;
    private String securityQuestion;
    private String securityAnswer;
    private String sessionId;

    public User(String name, String mobile, String email, String password,
        String securityQuestion, String securityAnswer) {
        this.name = name;
        this.mobile = mobile;
        this.email = email;
        this.password = password;
        this.securityQuestion = securityQuestion;
        this.securityAnswer = securityAnswer;
        this.sessionId = UUID.randomUUID().toString();
    }
}

```

```
}
```

```
private User(int userId, String name, String mobile, String email, String password,
```

```
String securityQuestion, String securityAnswer, String sessionId) {
```

```
this.userId = userId;
```

```
this.name = name;
```

```
this.mobile = mobile;
```

```
this.email = email;
```

```
this.password = password;
```

```
this.securityQuestion = securityQuestion;
```

```
this.securityAnswer = securityAnswer;
```

```
this.sessionId = sessionId;
```

```
}
```

```
public void register() throws SQLException {
```

```
try (Connection conn = DBConnection.getConnection()) {
```

```
String sql = "INSERT INTO Users (name, mobile, email, password, security_question, security_answer) " +
```

```
"VALUES (?, ?, ?, ?, ?, ?)";
```

```
PreparedStatement stmt = conn.prepareStatement(sql, Statement.RETURN_GENERATED_KEYS);
```

```
stmt.setString(1, name);
```

```
stmt.setString(2, mobile);
```

```
stmt.setString(3, email);
```

```
stmt.setString(4, password);
```

```
stmt.setString(5, securityQuestion);
```

```
stmt.setString(6, securityAnswer);
```

```
stmt.executeUpdate();
```

```
ResultSet rs = stmt.getGeneratedKeys();
```

```
if (rs.next()) {
```

```
this.userId = rs.getInt(1);
```

```
}
```

```
DBConnection.logSession(userId, "register", sessionId);
```

```
        String accountSql = "INSERT INTO AccountHolder (name, age,
account_number, balance, interest, user_id) " +
        "VALUES (?, ?, ?, ?, ?, ?)";
        PreparedStatement accountStmt = conn.prepareStatement(accountSql);
        accountStmt.setString(1, name);
        accountStmt.setInt(2, 0);
        accountStmt.setString(3, "ACC" + (1000 + new Random().nextInt(9000)));
        accountStmt.setDouble(4, 10000.0);
        accountStmt.setDouble(5, 3.5);
        accountStmt.setInt(6, userId);
        accountStmt.executeUpdate();
    }
}
```

```
public static User login(String email, String password) throws SQLException {
    try (Connection conn = DBConnection.getConnection()) {
        String sql = "SELECT * FROM Users WHERE email = ? AND password =
?";
        PreparedStatement stmt = conn.prepareStatement(sql);
        stmt.setString(1, email);
        stmt.setString(2, password);
        ResultSet rs = stmt.executeQuery();

        if (rs.next()) {
            String sessionId = UUID.randomUUID().toString();
            User user = new User(
                rs.getInt("user_id"),
                rs.getString("name"),
                rs.getString("mobile"),
                rs.getString("email"),
                rs.getString("password"),
                rs.getString("security_question"),
                rs.getString("security_answer"),
            );
        }
    }
}
```

```

        sessionId
    );
    DBConnection.logSession(user.userId, "login", sessionId);
    return user;
}
return null;
}
}

```

```

public boolean hasCompleteProfile() throws SQLException {
    try (Connection conn = DBConnection.getConnection()) {
        String sql = "SELECT COUNT(*) FROM Profiles WHERE user_id = ?";
        PreparedStatement stmt = conn.prepareStatement(sql);
        stmt.setInt(1, userId);
        ResultSet rs = stmt.executeQuery();
        rs.next();
        return rs.getInt(1) > 0;
    }
}

```

```

public void logout() {
    try {
        DBConnection.logSession(userId, "logout", sessionId);
    } catch (SQLException e) {
        e.printStackTrace();
    }
    sessionId = null;
}

```

```

public static String getSecurityQuestion(String email) throws SQLException {
    try (Connection conn = DBConnection.getConnection()) {
        String sql = "SELECT security_question FROM Users WHERE email = ?";
        PreparedStatement stmt = conn.prepareStatement(sql);
        stmt.setString(1, email);
        ResultSet rs = stmt.executeQuery();
    }
}

```

```

        if (rs.next()) {
            return rs.getString("security_question");
        }
        return null;
    }
}

```

```

    public static boolean verifySecurityAnswer(String email, String answer) throws
SQLException {
        try (Connection conn = DBConnection.getConnection()) {
            String sql = "SELECT security_answer FROM Users WHERE email = ?";
            PreparedStatement stmt = conn.prepareStatement(sql);
            stmt.setString(1, email);
            ResultSet rs = stmt.executeQuery();
            if (rs.next()) {
                return rs.getString("security_answer").equalsIgnoreCase(answer);
            }
            return false;
        }
    }
}

```

```

    public static void resetPassword(String email, String newPassword) throws
SQLException {
        try (Connection conn = DBConnection.getConnection()) {
            String sql = "UPDATE Users SET password = ? WHERE email = ?";
            PreparedStatement stmt = conn.prepareStatement(sql);
            stmt.setString(1, newPassword);
            stmt.setString(2, email);
            stmt.executeUpdate();
        }
    }
}

```

```

public int getUserId() { return userId; }
public String getName() { return name; }
public String getSessionId() { return sessionId; }

```

```
}
```

```
class Profile {
```

```
    private int userId;
```

```
    private int age;
```

```
    private String gender;
```

```
    private String preferredClass;
```

```
    private String govId;
```

```
    private String emergencyContact;
```

```
    private boolean wheelchairAccess;
```

```
    private boolean medicalCondition;
```

```
    public Profile(int userId, int age, String gender, String preferredClass, String govId,
```

```
                    String emergencyContact, boolean wheelchairAccess, boolean medicalCondition) {
```

```
        this.userId = userId;
```

```
        this.age = age;
```

```
        this.gender = gender;
```

```
        this.preferredClass = preferredClass;
```

```
        this.govId = govId;
```

```
        this.emergencyContact = emergencyContact;
```

```
        this.wheelchairAccess = wheelchairAccess;
```

```
        this.medicalCondition = medicalCondition;
```

```
    }
```

```
    public void save() throws SQLException {
```

```
        try (Connection conn = DBConnection.getConnection()) {
```

```
            String sql = "INSERT INTO Profiles (user_id, age, gender, preferred_class, gov_id, " +
```

```
                                "emergency_contact, wheelchair_access, medical_condition) VALUES (?, ?, ?, ?, ?, ?, ?, ?)";
```

```
            PreparedStatement stmt = conn.prepareStatement(sql);
```

```
            stmt.setInt(1, userId);
```

```
            stmt.setInt(2, age);
```

```

        stmt.setString(3, gender);
        stmt.setString(4, preferredClass);
        stmt.setString(5, govId);
        stmt.setString(6, emergencyContact);
        stmt.setBoolean(7, wheelchairAccess);
        stmt.setBoolean(8, medicalCondition);
        stmt.executeUpdate();
    }
}

```

```

public static Profile getProfile(int userId) throws SQLException {
    try (Connection conn = DBConnection.getConnection()) {
        String sql = "SELECT * FROM Profiles WHERE user_id = ?";
        PreparedStatement stmt = conn.prepareStatement(sql);
        stmt.setInt(1, userId);
        ResultSet rs = stmt.executeQuery();
        if (rs.next()) {
            return new Profile(
                userId,
                rs.getInt("age"),
                rs.getString("gender"),
                rs.getString("preferred_class"),
                rs.getString("gov_id"),
                rs.getString("emergency_contact"),
                rs.getBoolean("wheelchair_access"),
                rs.getBoolean("medical_condition")
            );
        }
        return null;
    }
}

```

```

public int getAge() { return age; }
public String getGender() { return gender; }
public String getPreferredClass() { return preferredClass; }

```



```

    public String getGovId() { return govId; }
    public String getEmergencyContact() { return emergencyContact; }
    public boolean isWheelchairAccess() { return wheelchairAccess; }
    public boolean isMedicalCondition() { return medicalCondition; }
}

```

```

class Train {
    private String trainNumber;
    private String trainName;
    private String sourceStation;
    private String destinationStation;
    private String departureTime;
    private String arrivalTime;
    private String duration;
    private String runsOn;

```

```

    public Train(String trainNumber, String trainName, String sourceStation, String
destinationStation,
                    String departureTime, String arrivalTime, String duration, String
runsOn) {
        this.trainNumber = trainNumber;
        this.trainName = trainName;
        this.sourceStation = sourceStation;
        this.destinationStation = destinationStation;
        this.departureTime = departureTime;
        this.arrivalTime = arrivalTime;
        this.duration = duration;
        this.runsOn = runsOn;
    }

```

```

    public static List<Train> searchTrainsWithFilters(String from, String to, String
journeyDate,
                                                    String classType, String trainType, String
timeRange) throws SQLException {
        List<Train> trains = new ArrayList<>();

```

```

try (Connection conn = DBConnection.getConnection()) {
    StringBuilder sql = new StringBuilder(
        "SELECT * FROM Trains WHERE source_station LIKE ? AND
destination_station LIKE ?"
    );

    if (!trainType.equals("All")) {
        sql.append(" AND train_name LIKE ?");
    }

    PreparedStatement stmt = conn.prepareStatement(sql.toString());
    stmt.setString(1, "%" + from + "%");
    stmt.setString(2, "%" + to + "%");

    int paramIndex = 3;
    if (!trainType.equals("All")) {
        stmt.setString(paramIndex++, "%" + trainType + "%");
    }

    ResultSet rs = stmt.executeQuery();
    while (rs.next()) {
        Train train = new Train(
            rs.getString("train_number"),
            rs.getString("train_name"),
            rs.getString("source_station"),
            rs.getString("destination_station"),
            rs.getString("departure_time"),
            rs.getString("arrival_time"),
            rs.getString("duration"),
            rs.getString("runs_on")
        );

        SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");
        try {
            java.util.Date date = sdf.parse(journeyDate);

```

```

        String[] days = train.runsOn.split(",");
        String journeyDay = new
SimpleDateFormat("EEE").format(date).toLowerCase();
        boolean runsOnDay = false;
        for (String day : days) {
            if (day.trim().toLowerCase().startsWith(journeyDay)) {
                runsOnDay = true;
                break;
            }
        }
        if (runsOnDay) {
            if (timeRange != null && !timeRange.equals("All")) {
                String[] times = timeRange.split("-");
                int startHour = Integer.parseInt(times[0].split(":")[0]);
                int endHour = Integer.parseInt(times[1].split(":")[0]);
                int departureHour =
Integer.parseInt(train.departureTime.split(":")[0]);
                if (departureHour >= startHour && departureHour <= endHour)
            {
                trains.add(train);
            }
            } else {
                trains.add(train);
            }
        }
    } catch (ParseException e) {
        throw new SQLException("Invalid date format: " + journeyDate, e);
    }
}
}
return trains;
}

```

```

public boolean isClassAvailable(String classType) {
    return getAvailabilityForClass(classType).equals("Available");
}

```

```
}
```

```
public String getAvailabilityForClass(String classType) {
```

```
    Random random = new Random();
```

```
    int availability = random.nextInt(100);
```

```
    if (availability > 80) {
```

```
        return "WL " + (random.nextInt(50) + 1);
```

```
    } else if (availability > 60) {
```

```
        return "RAC " + (random.nextInt(20) + 1);
```

```
    } else {
```

```
        return "Available";
```

```
    }
```

```
}
```

```
public String getTrainNumber() { return trainNumber; }
```

```
public String getTrainName() { return trainName; }
```

```
public String getSourceStation() { return sourceStation; }
```

```
public String getDestinationStation() { return destinationStation; }
```

```
public String getDepartureTime() { return departureTime; }
```

```
public String getArrivalTime() { return arrivalTime; }
```

```
public String getDuration() { return duration; }
```

```
public String getRunsOn() { return runsOn; }
```

```
}
```

```
class Booking {
```

```
    private static final String[] CLASS_TYPES = {
```

```
        "Anubhuti Class (EA)", "AC First Class (1A)", "Vistadome AC (EV)",
```

```
        "Exec. Chair Car (EC)", "AC 2 Tier (2A)", "AC 3 Tier (3A)",
```

```
        "AC 3 Economy (3E)", "Sleeper (SL)", "Second Sitting (2S)", "General (GN)"
```

```
    };
```

```
    private static final String[] QUOTA_TYPES = {"General", "Tatkal", "Premium Tatkal", "Ladies", "Senior Citizen"};
```

```
private static final Map<String, Double> CLASS_FARES = new HashMap<>();
    private static final Map<String, Double> CONCESSION_RATES = new
HashMap<>();
```

```
static {
    CLASS_FARES.put("Anubhuti Class (EA)", 4500.0);
    CLASS_FARES.put("AC First Class (1A)", 3500.0);
    CLASS_FARES.put("Vistadome AC (EV)", 3200.0);
    CLASS_FARES.put("Exec. Chair Car (EC)", 2800.0);
    CLASS_FARES.put("AC 2 Tier (2A)", 2500.0);
    CLASS_FARES.put("AC 3 Tier (3A)", 1800.0);
    CLASS_FARES.put("AC 3 Economy (3E)", 1500.0);
    CLASS_FARES.put("Sleeper (SL)", 800.0);
    CLASS_FARES.put("Second Sitting (2S)", 400.0);
    CLASS_FARES.put("General (GN)", 300.0);

    CONCESSION_RATES.put("Senior Citizen", 0.5);
    CONCESSION_RATES.put("Student", 0.25);
    CONCESSION_RATES.put("Person with Disability", 0.4);
    CONCESSION_RATES.put("Child (5-12 years)", 0.5);
    CONCESSION_RATES.put("Armed Forces", 0.3);
}
```

```
public static Map<String, Double> getClassFares() {
    return CLASS_FARES;
}
```

```
public static Map<String, Double> getConcessionRates() {
    return CONCESSION_RATES;
}
```

```
    public static String bookTickets(int userId, String trainNumber, String
trainName,
        String fromStation, String toStation, String journeyDate,
```

```

        String classType, String quota, List<Passenger> passengers,
        String paymentMethod) throws SQLException {
    Connection conn = DBConnection.getConnection();

    String pnr = generatePNR();
    double totalFare = 0.0;
    Random random = new Random();

    double baseFare = getClassFares().getOrDefault(classType, 0.0);
    if (quota.equals("Tatkal")) baseFare *= 1.3;
    if (quota.equals("Premium Tatkal")) baseFare *= 1.5;

    for (Passenger passenger : passengers) {
        double fare = baseFare;
        if (passenger.getConcessionType() != null &&
!passenger.getConcessionType().isEmpty() &&
        getConcessionRates().containsKey(passenger.getConcessionType())) {
            double discount = fare *
getConcessionRates().get(passenger.getConcessionType());
            fare -= discount;
        }
        totalFare += fare;
    }

    boolean paymentSuccess = DBConnection.deductBalance(userId, totalFare);
    if (!paymentSuccess) {
        throw new SQLException("Insufficient balance to complete the booking.");
    }

    try {
        String sql = "INSERT INTO Bookings (user_id, train_number, train_name,
from_station, " +
            "to_station, journey_date, class_type, passenger_name, age, gender,
" +

```

```

        "berth_preference, fare, pnr_number, booking_status, seat_number)
" +
        "VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)";

```

```

PreparedStatement stmt = conn.prepareStatement(sql);
for (Passenger passenger : passengers) {
    double fare = baseFare;
    if (passenger.getConcessionType() != null &&
!passenger.getConcessionType().isEmpty()) {
        double discount = fare *
getConcessionRates().get(passenger.getConcessionType());
        fare -= discount;
    }
}

```

```

String bookingStatus = "Confirmed";
String seatNumber = String.format("%03d", random.nextInt(1000));
Train tempTrain = new Train(trainNumber, trainName, fromStation,
toStation, "", "", "", "");
String availability = tempTrain.getAvailabilityForClass(classType);
if (availability.startsWith("WL")) {
    bookingStatus = "Waiting List";
    seatNumber = availability;
} else if (availability.startsWith("RAC")) {
    bookingStatus = "RAC";
    seatNumber = availability;
}

```

```

stmt.setInt(1, userId);
stmt.setString(2, trainNumber);
stmt.setString(3, trainName);
stmt.setString(4, fromStation);
stmt.setString(5, toStation);
stmt.setString(6, convertDate(journeyDate));
stmt.setString(7, classType);
stmt.setString(8, passenger.getName());

```

```

        stmt.setInt(9, passenger.getAge());
        stmt.setString(10, passenger.getGender());
        stmt.setString(11, passenger.getBerthPreference());
        stmt.setDouble(12, fare);
        stmt.setString(13, pnr);
        stmt.setString(14, bookingStatus);
        stmt.setString(15, seatNumber);

        stmt.addBatch();
    }
    stmt.executeBatch();

    DBConnection.logSession(userId, "book_tickets via " + paymentMethod,
pnr);
    return pnr;
} catch (SQLException e) {
    DBConnection.refundBalance(userId, totalFare);
    throw e;
}
}

private static String generatePNR() {
    Random random = new Random();
    return "PNR" + (100000 + random.nextInt(900000));
}

private static String convertDate(String journeyDate) throws SQLException {
    try {
        SimpleDateFormat inputFormat = new
SimpleDateFormat("dd/MM/yyyy");
        SimpleDateFormat outputFormat = new
SimpleDateFormat("yyyy-MM-dd");
        return outputFormat.format(inputFormat.parse(journeyDate));
    } catch (ParseException e) {
        throw new SQLException("Invalid date format: " + journeyDate, e);
    }
}

```



```
}  
}
```

```
public static ResultSet getBookingHistory(int userId) throws SQLException {  
    Connection conn = DBConnection.getConnection();  
    String sql = "SELECT * FROM Bookings WHERE user_id = ? ORDER BY  
booking_date DESC";  
    PreparedStatement stmt = conn.prepareStatement(sql);  
    stmt.setInt(1, userId);  
    return stmt.executeQuery();  
}
```

```
public static boolean cancelTicket(String pnrNumber) throws SQLException {  
    Connection conn = DBConnection.getConnection();  
    conn.setAutoCommit(false);  
    try {  
        String getBookingSql = "SELECT user_id, fare FROM Bookings WHERE  
pnr_number = ? AND booking_status = 'Confirmed'";  
        PreparedStatement getBookingStmt =  
conn.prepareStatement(getBookingSql);  
        getBookingStmt.setString(1, pnrNumber);  
        ResultSet rs = getBookingStmt.executeQuery();  
  
        if (!rs.next()) {  
            conn.rollback();  
            return false;  
        }  
  
        int userId = rs.getInt("user_id");  
        double totalFare = 0.0;  
  
        do {  
            totalFare += rs.getDouble("fare");  
        } while (rs.next());  
    }  
}
```

```

        String updateSql = "UPDATE Bookings SET booking_status = 'Cancelled'
WHERE pnr_number = ?";
        PreparedStatement updateStmt = conn.prepareStatement(updateSql);
        updateStmt.setString(1, pnrNumber);
        int rowsAffected = updateStmt.executeUpdate();

        if (rowsAffected > 0) {
            DBConnection.refundBalance(userId, totalFare);
            DBConnection.logSession(userId, "cancel_ticket", pnrNumber);
            conn.commit();
            return true;
        }

        conn.rollback();
        return false;
    } catch (SQLException e) {
        conn.rollback();
        throw e;
    } finally {
        conn.setAutoCommit(true);
    }
}

public static ResultSet getPNRStatus(String pnrNumber) throws SQLException
{
    Connection conn = DBConnection.getConnection();
    String sql = "SELECT * FROM Bookings WHERE pnr_number = ?";
    PreparedStatement stmt = conn.prepareStatement(sql);
    stmt.setString(1, pnrNumber);
    return stmt.executeQuery();
}

public static String[] getClassTypes() {
    return CLASS_TYPES;
}

```

```

public static String[] getQuotaTypes() {
    return QUOTA_TYPES;
}

public static String[] getConcessionTypes() {
    return CONCESSION_RATES.keySet().toArray(new String[0]);
}

public static class Passenger {
    private String name;
    private int age;
    private String gender;
    private String berthPreference;
    private String concessionType;

    public Passenger(String name, int age, String gender, String berthPreference,
String concessionType) {
        this.name = name;
        this.age = age;
        this.gender = gender;
        this.berthPreference = berthPreference;
        this.concessionType = concessionType;
    }

    public String getName() { return name; }
    public int getAge() { return age; }
    public String getGender() { return gender; }
    public String getBerthPreference() { return berthPreference; }
    public String getConcessionType() { return concessionType; }
}

}

public class TrainManagementSystem extends JFrame {
    private User currentUser;

```

```

private JPanel currentPanel;
private JTextArea outputArea;
private boolean darkTheme = false;
private JLabel userInfoLabel;
private List<String> recentSearches = new ArrayList<>();

public TrainManagementSystem() {
    super("IRCTC Train Booking System");
    setSize(1200, 800);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setLayout(new BorderLayout());

    DBConnection.initializeDatabase();

    JPanel headerPanel = createHeaderPanel();
    add(headerPanel, BorderLayout.NORTH);

    currentPanel = new JPanel();
    add(currentPanel, BorderLayout.CENTER);

    JPanel footerPanel = createFooterPanel();
    add(footerPanel, BorderLayout.SOUTH);

    showLoginPanel();

    setLocationRelativeTo(null);
    setVisible(true);
}

private JPanel createHeaderPanel() {
    JPanel panel = new JPanel(new BorderLayout());
    panel.setBackground(new Color(0, 100, 0));
    panel.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));
}

```

```
JLabel titleLabel = new JLabel("IRCTC - Indian Railway Catering and  
Tourism Corporation", SwingConstants.CENTER);  
titleLabel.setFont(new Font("Arial", Font.BOLD, 24));  
titleLabel.setForeground(Color.WHITE);  
panel.add(titleLabel, BorderLayout.CENTER);
```

```
JPanel rightPanel = new JPanel(new FlowLayout(FlowLayout.RIGHT));  
rightPanel.setBackground(new Color(0, 100, 0));
```

```
userInfoLabel = new JLabel("Welcome, Guest");  
userInfoLabel.setForeground(Color.WHITE);  
userInfoLabel.setFont(new Font("Arial", Font.PLAIN, 14));  
rightPanel.add(userInfoLabel);
```

```
JLabel dateLabel = new JLabel(new SimpleDateFormat("dd-MMM-yyyy  
[HH:mm:ss]").format(new java.util.Date()));  
dateLabel.setForeground(Color.WHITE);  
dateLabel.setFont(new Font("Arial", Font.PLAIN, 14));  
dateLabel.setBorder(BorderFactory.createEmptyBorder(0, 10, 0, 10));  
rightPanel.add(dateLabel);
```

```
panel.add(rightPanel, BorderLayout.EAST);
```

```
return panel;  
}
```

```
private JPanel createFooterPanel() {  
    JPanel panel = new JPanel();  
    panel.setBackground(new Color(0, 100, 0));  
    panel.setBorder(BorderFactory.createEmptyBorder(5, 10, 5, 10));
```

```
JLabel footerLabel = new JLabel("Customer Care Numbers: 1464 /  
80904687989 / 08035734999 | " +  
    "BEWARE OF FRAUDSTERS: Always download official  
IRCTC Rail Connect App from the Google Play Store or Apple App Store only.");
```

```

        footerLabel.setForeground(Color.WHITE);
        footerLabel.setFont(new Font("Arial", Font.PLAIN, 10));
        panel.add(footerLabel);

        return panel;
    }

    private void applyTheme() {
        if (darkTheme) {
            currentPanel.setBackground(new Color(60, 63, 65));
            currentPanel.setForeground(Color.WHITE);
            if (outputArea != null) {
                outputArea.setBackground(new Color(60, 63, 65));
                outputArea.setForeground(Color.WHITE);
            }
        } else {
            currentPanel.setBackground(new Color(238, 238, 238));
            currentPanel.setForeground(Color.BLACK);
            if (outputArea != null) {
                outputArea.setBackground(Color.WHITE);
                outputArea.setForeground(Color.BLACK);
            }
        }
    }

    private void showLoginPanel() {
        currentPanel.removeAll();
        currentPanel.setLayout(new GridBagLayout());
        GridBagConstraints gbc = new GridBagConstraints();
        gbc.insets = new Insets(10, 10, 10, 10);
        gbc.fill = GridBagConstraints.HORIZONTAL;

        JLabel titleLabel = new JLabel("Login to IRCTC", SwingConstants.CENTER);
        titleLabel.setFont(new Font("Arial", Font.BOLD, 24));
        gbc.gridx = 0;

```

```
gbc.gridy = 0;  
gbc.gridwidth = 2;  
currentPanel.add(titleLabel, gbc);
```

```
JLabel emailLabel = new JLabel("Email:");  
gbc.gridx = 0;  
gbc.gridy = 1;  
gbc.gridwidth = 1;  
currentPanel.add(emailLabel, gbc);
```

```
JTextField emailField = new JTextField(20);  
gbc.gridx = 1;  
gbc.gridy = 1;  
currentPanel.add(emailField, gbc);
```

```
JLabel passwordLabel = new JLabel("Password:");  
gbc.gridx = 0;  
gbc.gridy = 2;  
currentPanel.add(passwordLabel, gbc);
```

```
JPasswordField passwordField = new JPasswordField(20);  
gbc.gridx = 1;  
gbc.gridy = 2;  
currentPanel.add(passwordField, gbc);
```

```
JButton loginButton = new JButton("Login");  
gbc.gridx = 0;  
gbc.gridy = 3;  
gbc.gridwidth = 2;  
currentPanel.add(loginButton, gbc);
```

```
JButton registerButton = new JButton("Register New User");  
gbc.gridx = 0;  
gbc.gridy = 4;  
currentPanel.add(registerButton, gbc);
```

```
JButton forgotPasswordButton = new JButton("Forgot Password?");
gbc.gridx = 0;
gbc.gridy = 5;
currentPanel.add(forgotPasswordButton, gbc);
```

```
JButton themeToggleButton = new JButton("Toggle Dark Theme");
gbc.gridx = 0;
gbc.gridy = 6;
currentPanel.add(themeToggleButton, gbc);
```

```
loginButton.addActionListener(e -> {
    try {
        String email = emailField.getText();
        String password = new String(passwordField.getPassword());
        currentUser = User.login(email, password);
        if (currentUser != null) {
            userInfoLabel.setText("Welcome, " + currentUser.getName());
            showMainMenu();
        } else {
            JOptionPane.showMessageDialog(this, "Invalid email or password",
"Login Failed", JOptionPane.ERROR_MESSAGE);
        }
    } catch (SQLException ex) {
        JOptionPane.showMessageDialog(this, "Database error: " +
ex.getMessage(), "Error", JOptionPane.ERROR_MESSAGE);
    }
});
```

```
registerButton.addActionListener(e -> showRegistrationPanel());
forgotPasswordButton.addActionListener(e -> showForgotPasswordPanel());
themeToggleButton.addActionListener(e -> {
    darkTheme = !darkTheme;
    applyTheme();
});
```



```

    applyTheme();
    currentPanel.revalidate();
    currentPanel.repaint();
}

private void showRegistrationPanel() {
    currentPanel.removeAll();
    currentPanel.setLayout(new GridBagLayout());
    GridBagConstraints gbc = new GridBagConstraints();
    gbc.insets = new Insets(10, 10, 10, 10);
    gbc.fill = GridBagConstraints.HORIZONTAL;

    JLabel titleLabel = new JLabel("New User Registration",
SwingConstants.CENTER);
    titleLabel.setFont(new Font("Arial", Font.BOLD, 24));
    gbc.gridx = 0;
    gbc.gridy = 0;
    gbc.gridwidth = 2;
    currentPanel.add(titleLabel, gbc);

    String[] securityQuestions = {
        "What was your first pet's name?",
        "What was the name of your first school?",
        "What is your mother's maiden name?",
        "What city were you born in?",
        "What was your favorite food as a child?"
    };

    JLabel nameLabel = new JLabel("Full Name:");
    gbc.gridx = 0;
    gbc.gridy = 1;
    gbc.gridwidth = 1;
    currentPanel.add(nameLabel, gbc);

```

```
JTextField nameField = new JTextField(20);  
gbc.gridx = 1;  
gbc.gridy = 1;  
currentPanel.add(nameField, gbc);
```

```
JLabel mobileLabel = new JLabel("Mobile Number:");  
gbc.gridx = 0;  
gbc.gridy = 2;  
currentPanel.add(mobileLabel, gbc);
```

```
JTextField mobileField = new JTextField(20);  
gbc.gridx = 1;  
gbc.gridy = 2;  
currentPanel.add(mobileField, gbc);
```

```
JLabel emailLabel = new JLabel("Email:");  
gbc.gridx = 0;  
gbc.gridy = 3;  
currentPanel.add(emailLabel, gbc);
```

```
JTextField emailField = new JTextField(20);  
gbc.gridx = 1;  
gbc.gridy = 3;  
currentPanel.add(emailField, gbc);
```

```
JLabel passwordLabel = new JLabel("Password:");  
gbc.gridx = 0;  
gbc.gridy = 4;  
currentPanel.add(passwordLabel, gbc);
```

```
JPasswordField passwordField = new JPasswordField(20);  
gbc.gridx = 1;  
gbc.gridy = 4;  
currentPanel.add(passwordField, gbc);
```

```
JLabel confirmPasswordLabel = new JLabel("Confirm Password:");
gbc.gridx = 0;
gbc.gridy = 5;
currentPanel.add(confirmPasswordLabel, gbc);
```

```
JPasswordField confirmPasswordField = new JPasswordField(20);
gbc.gridx = 1;
gbc.gridy = 5;
currentPanel.add(confirmPasswordField, gbc);
```

```
JLabel securityQuestionLabel = new JLabel("Security Question:");
gbc.gridx = 0;
gbc.gridy = 6;
currentPanel.add(securityQuestionLabel, gbc);
```

```
                                JComboBox<String>    securityQuestionCombo    =    new
JComboBox<>(securityQuestions);
gbc.gridx = 1;
gbc.gridy = 6;
currentPanel.add(securityQuestionCombo, gbc);
```

```
JLabel securityAnswerLabel = new JLabel("Security Answer:");
gbc.gridx = 0;
gbc.gridy = 7;
currentPanel.add(securityAnswerLabel, gbc);
```

```
JTextField securityAnswerField = new JTextField(20);
gbc.gridx = 1;
gbc.gridy = 7;
currentPanel.add(securityAnswerField, gbc);
```

```
JButton registerButton = new JButton("Register");
gbc.gridx = 0;
gbc.gridy = 8;
gbc.gridwidth = 2;
```

```

currentPanel.add(registerButton, gbc);

JButton backButton = new JButton("Back to Login");
gbc.gridx = 0;
gbc.gridy = 9;
currentPanel.add(backButton, gbc);

registerButton.addActionListener(e -> {
    String name = nameField.getText();
    String mobile = mobileField.getText();
    String email = emailField.getText();
    String password = new String(passwordField.getPassword());
    String confirmPassword = new String(confirmPasswordField.getPassword());
    String securityQuestion = (String) securityQuestionCombo.getSelectedItem();
    String securityAnswer = securityAnswerField.getText();

    if (!password.equals(confirmPassword)) {
        JOptionPane.showMessageDialog(this, "Passwords do not match", "Error",
JOptionPane.ERROR_MESSAGE);
        return;
    }

    if (name.isEmpty() || mobile.isEmpty() || email.isEmpty() ||
password.isEmpty() || securityAnswer.isEmpty()) {
        JOptionPane.showMessageDialog(this, "All fields are required", "Error",
JOptionPane.ERROR_MESSAGE);
        return;
    }

    try {
        User newUser = new User(name, mobile, email, password,
securityQuestion, securityAnswer);
        newUser.register();
        JOptionPane.showMessageDialog(this, "Registration successful! Please
login.", "Success", JOptionPane.INFORMATION_MESSAGE);
    }

```

```
        showLoginPanel();
    } catch (SQLException ex) {
        JOptionPane.showMessageDialog(this, "Registration failed: " +
ex.getMessage(), "Error", JOptionPane.ERROR_MESSAGE);
    }
});
```

```
backButton.addActionListener(e -> showLoginPanel());
```

```
    applyTheme();
    currentPanel.revalidate();
    currentPanel.repaint();
}
```

```
private void showForgotPasswordPanel() {
    currentPanel.removeAll();
    currentPanel.setLayout(new GridBagLayout());
    GridBagConstraints gbc = new GridBagConstraints();
    gbc.insets = new Insets(10, 10, 10, 10);
    gbc.fill = GridBagConstraints.HORIZONTAL;

    JLabel titleLabel = new JLabel("Password Recovery",
SwingConstants.CENTER);
    titleLabel.setFont(new Font("Arial", Font.BOLD, 24));
    gbc.gridx = 0;
    gbc.gridy = 0;
    gbc.gridwidth = 2;
    currentPanel.add(titleLabel, gbc);

    JLabel emailLabel = new JLabel("Registered Email:");
    gbc.gridx = 0;
    gbc.gridy = 1;
    gbc.gridwidth = 1;
    currentPanel.add(emailLabel, gbc);
}
```

```
JTextField emailField = new JTextField(20);  
gbc.gridx = 1;  
gbc.gridy = 1;  
currentPanel.add(emailField, gbc);
```

```
JButton nextButton = new JButton("Next");  
gbc.gridx = 0;  
gbc.gridy = 2;  
gbc.gridwidth = 2;  
currentPanel.add(nextButton, gbc);
```

```
JButton backButton = new JButton("Back to Login");  
gbc.gridx = 0;  
gbc.gridy = 3;  
currentPanel.add(backButton, gbc);
```

```
nextButton.addActionListener(e -> {  
    String email = emailField.getText();  
    try {  
        String question = User.getSecurityQuestion(email);  
        if (question == null) {  
            JOptionPane.showMessageDialog(this, "Email not found", "Error",  
JOptionPane.ERROR_MESSAGE);  
            return;  
        }  
    }  
}
```

```
currentPanel.removeAll();  
currentPanel.setLayout(new GridBagLayout());
```

```
JLabel questionLabel = new JLabel("Security Question: " + question);  
gbc.gridx = 0;  
gbc.gridy = 0;  
gbc.gridwidth = 2;  
currentPanel.add(questionLabel, gbc);
```

```
JLabel answerLabel = new JLabel("Answer:");  
gbc.gridx = 0;  
gbc.gridy = 1;  
gbc.gridwidth = 1;  
currentPanel.add(answerLabel, gbc);
```

```
JTextField answerField = new JTextField(20);  
gbc.gridx = 1;  
gbc.gridy = 1;  
currentPanel.add(answerField, gbc);
```

```
JLabel newPasswordLabel = new JLabel("New Password:");  
gbc.gridx = 0;  
gbc.gridy = 2;  
currentPanel.add(newPasswordLabel, gbc);
```

```
JPasswordField newPasswordField = new JPasswordField(20);  
gbc.gridx = 1;  
gbc.gridy = 2;  
currentPanel.add(newPasswordField, gbc);
```

```
JLabel confirmPasswordLabel = new JLabel("Confirm Password:");  
gbc.gridx = 0;  
gbc.gridy = 3;  
currentPanel.add(confirmPasswordLabel, gbc);
```

```
JPasswordField confirmPasswordField = new JPasswordField(20);  
gbc.gridx = 1;  
gbc.gridy = 3;  
currentPanel.add(confirmPasswordField, gbc);
```

```
JButton resetButton = new JButton("Reset Password");  
gbc.gridx = 0;  
gbc.gridy = 4;  
gbc.gridwidth = 2;
```

```

currentPanel.add(resetButton, gbc);

JButton cancelButton = new JButton("Cancel");
gbc.gridx = 0;
gbc.gridy = 5;
currentPanel.add(cancelButton, gbc);

resetButton.addActionListener(ev -> {
    String answer = answerField.getText();
    String newPassword = new String(newPasswordField.getPassword());
    String confirmPassword = new
String(confirmPasswordField.getPassword());

    if (!newPassword.equals(confirmPassword)) {
        JOptionPane.showMessageDialog(this, "Passwords do not match",
"Error", JOptionPane.ERROR_MESSAGE);
        return;
    }

    try {
        if (User.verifySecurityAnswer(email, answer)) {
            User.resetPassword(email, newPassword);
            JOptionPane.showMessageDialog(this, "Password reset
successful!", "Success", JOptionPane.INFORMATION_MESSAGE);
            showLoginPanel();
        } else {
            JOptionPane.showMessageDialog(this, "Incorrect security answer",
"Error", JOptionPane.ERROR_MESSAGE);
        }
    } catch (SQLException ex) {
        JOptionPane.showMessageDialog(this, "Error: " + ex.getMessage(),
"Error", JOptionPane.ERROR_MESSAGE);
    }
});

```



```

        cancelButton.addActionListener(ev -> showLoginPanel());

    } catch (SQLException ex) {
        JOptionPane.showMessageDialog(this, "Error: " + ex.getMessage(),
"Error", JOptionPane.ERROR_MESSAGE);
    }

    applyTheme();
    currentPanel.revalidate();
    currentPanel.repaint();
});

backButton.addActionListener(e -> showLoginPanel());

applyTheme();
currentPanel.revalidate();
currentPanel.repaint();
}

private void showMainMenu() {
    currentPanel.removeAll();
    currentPanel.setLayout(new GridBagLayout());
    GridBagConstraints gbc = new GridBagConstraints();
    gbc.insets = new Insets(10, 10, 10, 10);
    gbc.fill = GridBagConstraints.HORIZONTAL;

    JLabel welcomeLabel = new JLabel("Welcome, " + currentUser.getName(),
SwingConstants.CENTER);
    welcomeLabel.setFont(new Font("Arial", Font.BOLD, 24));
    gbc.gridx = 0;
    gbc.gridy = 0;
    gbc.gridwidth = 2;
    currentPanel.add(welcomeLabel, gbc);

    JButton bookTicketButton = new JButton("Book Ticket");

```

```
gbc.gridx = 0;  
gbc.gridy = 1;  
gbc.gridwidth = 1;  
currentPanel.add(bookTicketButton, gbc);
```

```
JButton bookingHistoryButton = new JButton("Booking History");  
gbc.gridx = 1;  
gbc.gridy = 1;  
currentPanel.add(bookingHistoryButton, gbc);
```

```
JButton cancelTicketButton = new JButton("Cancel Ticket");  
gbc.gridx = 0;  
gbc.gridy = 2;  
currentPanel.add(cancelTicketButton, gbc);
```

```
JButton pnrStatusButton = new JButton("PNR Status");  
gbc.gridx = 1;  
gbc.gridy = 2;  
currentPanel.add(pnrStatusButton, gbc);
```

```
JButton manageProfileButton = new JButton("Manage Profile");  
gbc.gridx = 0;  
gbc.gridy = 3;  
currentPanel.add(manageProfileButton, gbc);
```

```
JButton accountBalanceButton = new JButton("Account Balance");  
gbc.gridx = 1;  
gbc.gridy = 3;  
currentPanel.add(accountBalanceButton, gbc);
```

```
JButton logoutButton = new JButton("Logout");  
gbc.gridx = 0;  
gbc.gridy = 4;  
gbc.gridwidth = 2;  
currentPanel.add(logoutButton, gbc);
```

```

bookTicketButton.addActionListener(e -> showTrainSearchPanel());
bookingHistoryButton.addActionListener(e -> showBookingHistory());
cancelTicketButton.addActionListener(e -> showCancelTicketPanel());
pnrStatusButton.addActionListener(e -> showPNRStatusPanel());
manageProfileButton.addActionListener(e -> showProfilePanel());
accountBalanceButton.addActionListener(e -> showAccountBalance());
logoutButton.addActionListener(e -> {
    currentUser.logout();
    currentUser = null;
    userInfoLabel.setText("Welcome, Guest");
    showLoginPanel();
});

applyTheme();
currentPanel.revalidate();
currentPanel.repaint();
}

private void showTrainSearchPanel() {
    currentPanel.removeAll();
    currentPanel.setLayout(new GridBagLayout());
    GridBagConstraints gbc = new GridBagConstraints();
    gbc.insets = new Insets(10, 10, 10, 10);
    gbc.fill = GridBagConstraints.HORIZONTAL;

    JLabel titleLabel = new JLabel("Search Trains", SwingConstants.CENTER);
    titleLabel.setFont(new Font("Arial", Font.BOLD, 24));
    gbc.gridx = 0;
    gbc.gridy = 0;
    gbc.gridwidth = 4;
    currentPanel.add(titleLabel, gbc);

    JLabel fromLabel = new JLabel("From:");
    gbc.gridx = 0;

```

```
gbc.gridy = 1;  
gbc.gridwidth = 1;  
currentPanel.add(fromLabel, gbc);
```

```
JTextField fromField = new JTextField(15);  
gbc.gridx = 1;  
gbc.gridy = 1;  
currentPanel.add(fromField, gbc);
```

```
JLabel toLabel = new JLabel("To:");  
gbc.gridx = 2;  
gbc.gridy = 1;  
currentPanel.add(toLabel, gbc);
```

```
JTextField toField = new JTextField(15);  
gbc.gridx = 3;  
gbc.gridy = 1;  
currentPanel.add(toField, gbc);
```

```
JLabel dateLabel = new JLabel("Journey Date (dd/mm/yyyy):");  
gbc.gridx = 0;  
gbc.gridy = 2;  
currentPanel.add(dateLabel, gbc);
```

```
JTextField dateField = new JTextField(15);  
gbc.gridx = 1;  
gbc.gridy = 2;  
currentPanel.add(dateField, gbc);
```

```
JLabel classLabel = new JLabel("Class:");  
gbc.gridx = 2;  
gbc.gridy = 2;  
currentPanel.add(classLabel, gbc);
```

```

        JComboBox<String> classCombo = new
JComboBox<>(Booking.getClassTypes());
gbc.gridx = 3;
gbc.gridy = 2;
currentPanel.add(classCombo, gbc);

JLabel trainTypeLabel = new JLabel("Train Type:");
gbc.gridx = 0;
gbc.gridy = 3;
currentPanel.add(trainTypeLabel, gbc);

String[] trainTypes = {"All", "Express", "Rajdhani", "Shatabdi", "Duronto", "Jan
Shatabdi"};
JComboBox<String> trainTypeCombo = new JComboBox<>(trainTypes);
gbc.gridx = 1;
gbc.gridy = 3;
currentPanel.add(trainTypeCombo, gbc);

JLabel timeRangeLabel = new JLabel("Departure Time:");
gbc.gridx = 2;
gbc.gridy = 3;
currentPanel.add(timeRangeLabel, gbc);

String[] timeRanges = {"All", "00:00-06:00", "06:00-12:00", "12:00-18:00",
"18:00-24:00"};
JComboBox<String> timeRangeCombo = new JComboBox<>(timeRanges);
gbc.gridx = 3;
gbc.gridy = 3;
currentPanel.add(timeRangeCombo, gbc);

JButton searchButton = new JButton("Search Trains");
gbc.gridx = 0;
gbc.gridy = 4;
gbc.gridwidth = 4;
currentPanel.add(searchButton, gbc);

```

```

JButton backButton = new JButton("Back to Main Menu");
gbc.gridx = 0;
gbc.gridy = 5;
currentPanel.add(backButton, gbc);

outputArea = new JTextArea(15, 60);
outputArea.setEditable(false);
JScrollPane scrollPane = new JScrollPane(outputArea);
gbc.gridx = 0;
gbc.gridy = 6;
gbc.gridwidth = 4;
gbc.fill = GridBagConstraints.BOTH;
currentPanel.add(scrollPane, gbc);

searchButton.addActionListener(e -> {
    String from = fromField.getText();
    String to = toField.getText();
    String date = dateField.getText();
    String classType = (String) classCombo.getSelectedItem();
    String trainType = (String) trainTypeCombo.getSelectedItem();
    String timeRange = (String) timeRangeCombo.getSelectedItem();

    try {
        List<Train> trains = Train.searchTrainsWithFilters(from, to, date,
classType, trainType, timeRange);
        displayTrainResults(trains, classType);
    } catch (SQLException ex) {
        JOptionPane.showMessageDialog(this, "Error searching trains: " +
ex.getMessage(), "Error", JOptionPane.ERROR_MESSAGE);
    }
});

backButton.addActionListener(e -> showMainMenu());

```

```

        applyTheme();
        currentPanel.revalidate();
        currentPanel.repaint();
    }

    private void displayTrainResults(List<Train> trains, String classType) {
        outputArea.setText("");
        if (trains.isEmpty()) {
            outputArea.append("No trains found for the given criteria.\n");
            return;
        }

        outputArea.append(String.format("%-10s %-30s %-20s %-20s %-10s %-10s
%-10s %-15s\n",
            "Train No", "Train Name", "Departure", "Arrival", "Duration", "From",
            "To", "Availability"));

        outputArea.append("-----
-----\n");

        for (Train train : trains) {
            String availability = train.getAvailabilityForClass(classType);
            outputArea.append(String.format("%-10s %-30s %-20s %-20s %-10s %-10s
%-10s %-15s\n",
                train.getTrainNumber(),
                train.getTrainName(),
                train.getSourceStation() + " (" + train.getDepartureTime() + ")",
                train.getDestinationStation() + " (" + train.getArrivalTime() + ")",
                train.getDuration(),
                train.getSourceStation(),
                train.getDestinationStation(),
                availability));
        }

        if (trains.size() > 0) {

```

```

JButton bookButton = new JButton("Book Selected Train");
GridBagConstraints gbc = new GridBagConstraints();
gbc.gridx = 0;
gbc.gridy = 7;
gbc.gridwidth = 4;
gbc.insets = new Insets(10, 10, 10, 10);
currentPanel.add(bookButton, gbc);

bookButton.addActionListener(e -> {
    String selectedTrain = JOptionPane.showInputDialog(this, "Enter Train
Number to book:");
    if (selectedTrain != null && !selectedTrain.isEmpty()) {
        for (Train train : trains) {
            if (train.getTrainNumber().equals(selectedTrain)) {
                showPassengerDetailsPanel(train);
                break;
            }
        }
    }
});

currentPanel.revalidate();
currentPanel.repaint();
}
}

private void showPassengerDetailsPanel(Train train) {
    currentPanel.removeAll();
    currentPanel.setLayout(new GridBagLayout());
    GridBagConstraints gbc = new GridBagConstraints();
    gbc.insets = new Insets(10, 10, 10, 10);
    gbc.fill = GridBagConstraints.HORIZONTAL;

    JLabel titleLabel = new JLabel("Passenger Details for " + train.getTrainName(),
SwingConstants.CENTER);

```



```
titleLabel.setFont(new Font("Arial", Font.BOLD, 24));
gbc.gridx = 0;
gbc.gridy = 0;
gbc.gridwidth = 4;
currentPanel.add(titleLabel, gbc);
```

```
        JLabel journeyLabel = new JLabel(train.getSourceStation() + " to " +
train.getDestinationStation());
gbc.gridx = 0;
gbc.gridy = 1;
gbc.gridwidth = 4;
currentPanel.add(journeyLabel, gbc);
```

```
JLabel dateLabel = new JLabel("Journey Date (dd/mm/yyyy):");
gbc.gridx = 0;
gbc.gridy = 2;
gbc.gridwidth = 1;
currentPanel.add(dateLabel, gbc);
```

```
JTextField dateField = new JTextField(15);
gbc.gridx = 1;
gbc.gridy = 2;
currentPanel.add(dateField, gbc);
```

```
JLabel classLabel = new JLabel("Class:");
gbc.gridx = 2;
gbc.gridy = 2;
currentPanel.add(classLabel, gbc);
```

```
                                JComboBox<String> classCombo = new
JComboBox<>(Booking.getClassTypes());
gbc.gridx = 3;
gbc.gridy = 2;
currentPanel.add(classCombo, gbc);
```

```
JLabel quotaLabel = new JLabel("Quota:");
gbc.gridx = 0;
gbc.gridy = 3;
currentPanel.add(quotaLabel, gbc);
```

```
JComboBox<String> quotaCombo = new
JComboBox<>(Booking.getQuotaTypes());
gbc.gridx = 1;
gbc.gridy = 3;
currentPanel.add(quotaCombo, gbc);
```

```
JLabel passengersLabel = new JLabel("Number of Passengers:");
gbc.gridx = 2;
gbc.gridy = 3;
currentPanel.add(passengersLabel, gbc);
```

```
SpinnerModel spinnerModel = new SpinnerNumberModel(1, 1, 6, 1);
JSpinner passengersSpinner = new JSpinner(spinnerModel);
gbc.gridx = 3;
gbc.gridy = 3;
currentPanel.add(passengersSpinner, gbc);
```

```
JPanel passengersPanel = new JPanel();
passengersPanel.setLayout(new BorderLayout(passengersPanel,
BoxLayout.Y_AXIS));
JScrollPane scrollPane = new JScrollPane(passengersPanel);
gbc.gridx = 0;
gbc.gridy = 4;
gbc.gridwidth = 4;
gbc.fill = GridBagConstraints.BOTH;
currentPanel.add(scrollPane, gbc);
```

```
JButton addPassengersButton = new JButton("Add Passenger Details");
gbc.gridx = 0;
gbc.gridy = 5;
```

```
gbc.gridwidth = 4;  
gbc.fill = GridBagConstraints.HORIZONTAL;  
currentPanel.add(addPassengersButton, gbc);
```

```
JButton bookButton = new JButton("Proceed to Payment");  
gbc.gridx = 0;  
gbc.gridy = 6;  
currentPanel.add(bookButton, gbc);
```

```
JButton backButton = new JButton("Back to Search");  
gbc.gridx = 1;  
gbc.gridy = 6;  
currentPanel.add(backButton, gbc);
```

```
List<JPanel> passengerForms = new ArrayList<>();
```

```
addPassengersButton.addActionListener(e -> {  
    int numPassengers = (Integer) passengersSpinner.getValue();  
    passengersPanel.removeAll();  
    passengerForms.clear();
```

```
    for (int i = 0; i < numPassengers; i++) {  
        JPanel formPanel = new JPanel(new GridLayout(0, 2, 5, 5));  
        formPanel.setBorder(BorderFactory.createTitledBorder("Passenger " + (i +  
1)));
```

```
        formPanel.add(new JLabel("Name:"));  
        JTextField nameField = new JTextField();  
        formPanel.add(nameField);
```

```
        formPanel.add(new JLabel("Age:"));  
        JSpinner ageSpinner = new JSpinner(new SpinnerNumberModel(18, 1,  
120, 1));  
        formPanel.add(ageSpinner);
```

```

        formPanel.add(new JLabel("Gender:"));
        JComboBox<String> genderCombo = new JComboBox<>(new
String[]{"Male", "Female", "Other"});
        formPanel.add(genderCombo);

        formPanel.add(new JLabel("Berth Preference:"));
        JComboBox<String> berthCombo = new JComboBox<>(new String[]{"No
Preference", "Lower", "Middle", "Upper", "Side Lower", "Side Upper"});
        formPanel.add(berthCombo);

        formPanel.add(new JLabel("Concession:"));
        JComboBox<String> concessionCombo = new
JComboBox<>(Booking.getConcessionTypes());
        concessionCombo.insertItemAt("None", 0);
        concessionCombo.setSelectedIndex(0);
        formPanel.add(concessionCombo);

        passengersPanel.add(formPanel);
        passengerForms.add(formPanel);
    }

    passengersPanel.revalidate();
    passengersPanel.repaint();
});

bookButton.addActionListener(e -> {
    String journeyDate = dateField.getText();
    String classType = (String) classCombo.getSelectedItem();
    String quota = (String) quotaCombo.getSelectedItem();

    if (journeyDate.isEmpty()) {
        JOptionPane.showMessageDialog(this, "Please enter journey date",
>Error", JOptionPane.ERROR_MESSAGE);
        return;
    }
}

```

```

List<Booking.Passenger> passengers = new ArrayList<>();
for (JPanel form : passengerForms) {
    Component[] components = form.getComponents();
    String name = ((JTextField) components[1]).getText();
    int age = (Integer) ((JSpinner) components[3]).getValue();
    String gender = (String) ((JComboBox<?>)
components[5]).getSelectedItem();
    String berthPreference = (String) ((JComboBox<?>)
components[7]).getSelectedItem();
    String concessionType = (String) ((JComboBox<?>)
components[9]).getSelectedItem();

    if (name.isEmpty()) {
        JOptionPane.showMessageDialog(this, "Please enter name for all
passengers", "Error", JOptionPane.ERROR_MESSAGE);
        return;
    }

    if ("None".equals(concessionType)) {
        concessionType = null;
    }

    passengers.add(new Booking.Passenger(name, age, gender,
berthPreference, concessionType));
}

if (passengers.isEmpty()) {
    JOptionPane.showMessageDialog(this, "Please add passenger details first",
"Error", JOptionPane.ERROR_MESSAGE);
    return;
}

try {
    String pnr = Booking.bookTickets(

```

```

        currentUser.getUserId(),
        train.getTrainNumber(),
        train.getTrainName(),
        train.getSourceStation(),
        train.getDestinationStation(),
        journeyDate,
        classType,
        quota,
        passengers,
        "Account Balance"
    );

    JOptionPane.showMessageDialog(this,
        "Booking successful!\nPNR: " + pnr + "\nTotal Fare: " +
        calculateTotalFare(passengers, classType, quota),
        "Success", JOptionPane.INFORMATION_MESSAGE);
    showMainMenu();
} catch (SQLException ex) {
    JOptionPane.showMessageDialog(this, "Booking failed: " +
        ex.getMessage(), "Error", JOptionPane.ERROR_MESSAGE);
}
});

backButton.addActionListener(e -> showTrainSearchPanel());

applyTheme();
currentPanel.revalidate();
currentPanel.repaint();
}

private double calculateTotalFare(List<Booking.Passenger> passengers, String
classType, String quota) {
    double total = 0.0;
    double baseFare = Booking.getClassFares().getOrDefault(classType, 0.0);

```

```

if (quota.equals("Tatkal")) baseFare *= 1.3;
if (quota.equals("Premium Tatkal")) baseFare *= 1.5;

for (Booking.Passenger passenger : passengers) {
    double fare = baseFare;
    if (passenger.getConcessionType() != null &&
!passenger.getConcessionType().isEmpty()) {
        double discount = fare *
Booking.getConcessionRates().get(passenger.getConcessionType());
        fare -= discount;
    }
    total += fare;
}

return total;
}

private void showBookingHistory() {
    currentPanel.removeAll();
    currentPanel.setLayout(new BorderLayout());

    JLabel titleLabel = new JLabel("Your Booking History",
SwingConstants.CENTER);
    titleLabel.setFont(new Font("Arial", Font.BOLD, 24));
    currentPanel.add(titleLabel, BorderLayout.NORTH);

    outputArea = new JTextArea();
    outputArea.setEditable(false);
    JScrollPane scrollPane = new JScrollPane(outputArea);
    currentPanel.add(scrollPane, BorderLayout.CENTER);

    JButton backButton = new JButton("Back to Main Menu");
    currentPanel.add(backButton, BorderLayout.SOUTH);

    try {

```

```

        ResultSet rs = Booking.getBookingHistory(currentUser.getUserId());
        outputArea.setText(String.format("%-10s %-15s %-30s %-15s %-15s %-12s
%-10s %-10s %-10s\n",
            "PNR", "Train No", "Train Name", "From", "To", "Journey Date",
            "Class", "Status", "Fare"));

        outputArea.append("-----
-----\n");

        while (rs.next()) {
            outputArea.append(String.format("%-10s %-15s %-30s %-15s %-15s
%-12s %-10s %-10s ₹%.2f\n",
                rs.getString("pnr_number"),
                rs.getString("train_number"),
                rs.getString("train_name"),
                rs.getString("from_station"),
                rs.getString("to_station"),
                rs.getDate("journey_date"),
                rs.getString("class_type"),
                rs.getString("booking_status"),
                rs.getDouble("fare")));
        }
    } catch (SQLException ex) {
        outputArea.setText("Error retrieving booking history: " + ex.getMessage());
    }

    backButton.addActionListener(e -> showMainMenu());

    applyTheme();
    currentPanel.revalidate();
    currentPanel.repaint();
}

private void showCancelTicketPanel() {
    currentPanel.removeAll();

```



```
currentPanel.setLayout(new GridBagLayout());
GridBagConstraints gbc = new GridBagConstraints();
gbc.insets = new Insets(10, 10, 10, 10);
gbc.fill = GridBagConstraints.HORIZONTAL;
```

```
JLabel titleLabel = new JLabel("Cancel Ticket", SwingConstants.CENTER);
titleLabel.setFont(new Font("Arial", Font.BOLD, 24));
gbc.gridx = 0;
gbc.gridy = 0;
gbc.gridwidth = 2;
currentPanel.add(titleLabel, gbc);
```

```
JLabel pnrLabel = new JLabel("Enter PNR Number:");
gbc.gridx = 0;
gbc.gridy = 1;
gbc.gridwidth = 1;
currentPanel.add(pnrLabel, gbc);
```

```
JTextField pnrField = new JTextField(15);
gbc.gridx = 1;
gbc.gridy = 1;
currentPanel.add(pnrField, gbc);
```

```
JButton checkButton = new JButton("Check PNR");
gbc.gridx = 0;
gbc.gridy = 2;
currentPanel.add(checkButton, gbc);
```

```
JButton cancelButton = new JButton("Cancel Ticket");
gbc.gridx = 1;
gbc.gridy = 2;
currentPanel.add(cancelButton, gbc);
```

```
JButton backButton = new JButton("Back to Main Menu");
gbc.gridx = 0;
```

```
gbc.gridy = 3;  
gbc.gridwidth = 2;  
currentPanel.add(backButton, gbc);
```

```
outputArea = new JTextArea(5, 30);  
outputArea.setEditable(false);  
JScrollPane scrollPane = new JScrollPane(outputArea);  
gbc.gridx = 0;  
gbc.gridy = 4;  
gbc.gridwidth = 2;  
gbc.fill = GridBagConstraints.BOTH;  
currentPanel.add(scrollPane, gbc);
```

```
checkButton.addActionListener(e -> {  
    String pnr = pnrField.getText().trim();  
    if (pnr.isEmpty()) {  
        JOptionPane.showMessageDialog(this, "Please enter a PNR number",  
"Error", JOptionPane.ERROR_MESSAGE);  
        return;  
    }  
    try {  
        ResultSet rs = Booking.getPNRStatus(pnr);  
        StringBuilder status = new StringBuilder();  
        boolean found = false;  
        while (rs.next()) {  
            found = true;  
            status.append("PNR: ").append(rs.getString("pnr_number"))  
                .append("\nTrain: ").append(rs.getString("train_name"))  
                .append(" (").append(rs.getString("train_number")).append(")")  
                .append("\nFrom: ").append(rs.getString("from_station"))  
                .append("\nTo: ").append(rs.getString("to_station"))  
                .append("\nDate: ").append(rs.getDate("journey_date"))  
                .append("\nClass: ").append(rs.getString("class_type"))  
                .append("\nPassenger: ").append(rs.getString("passenger_name"))  
                .append("\nStatus: ").append(rs.getString("booking_status"))
```

```

        .append("\nFare: ₹").append(rs.getDouble("fare"))
        .append("\n\n");
    }
    if (!found) {
        outputArea.setText("No booking found for PNR: " + pnr);
    } else {
        outputArea.setText(status.toString());
    }
} catch (SQLException ex) {
    outputArea.setText("Error retrieving PNR status: " + ex.getMessage());
}
});

cancelButton.addActionListener(e -> {
    String pnr = pnrField.getText().trim();
    if (pnr.isEmpty()) {
        JOptionPane.showMessageDialog(this, "Please enter a PNR number",
"Error", JOptionPane.ERROR_MESSAGE);
        return;
    }
    try {
        boolean success = Booking.cancelTicket(pnr);
        if (success) {
            outputArea.setText("Ticket with PNR " + pnr + " cancelled successfully.
Amount refunded.");
            JOptionPane.showMessageDialog(this, "Ticket cancelled successfully!",
"Success", JOptionPane.INFORMATION_MESSAGE);
        } else {
            outputArea.setText("No valid ticket found for PNR: " + pnr);
            JOptionPane.showMessageDialog(this, "Cancellation failed: No valid
ticket found", "Error", JOptionPane.ERROR_MESSAGE);
        }
    } catch (SQLException ex) {
        outputArea.setText("Error cancelling ticket: " + ex.getMessage());
    }
});

```

```
        JOptionPane.showMessageDialog(this, "Error cancelling ticket: " +
ex.getMessage(), "Error", JOptionPane.ERROR_MESSAGE);
    }
});
```

```
backButton.addActionListener(e -> showMainMenu());
```

```
applyTheme();
currentPanel.revalidate();
currentPanel.repaint();
}
```

```
private void showPNRStatusPanel() {
    currentPanel.removeAll();
    currentPanel.setLayout(new GridBagLayout());
    GridBagConstraints gbc = new GridBagConstraints();
    gbc.insets = new Insets(10, 10, 10, 10);
    gbc.fill = GridBagConstraints.HORIZONTAL;
```

```
    JLabel titleLabel = new JLabel("PNR Status", SwingConstants.CENTER);
    titleLabel.setFont(new Font("Arial", Font.BOLD, 24));
    gbc.gridx = 0;
    gbc.gridy = 0;
    gbc.gridwidth = 2;
    currentPanel.add(titleLabel, gbc);
```

```
    JLabel pnrLabel = new JLabel("Enter PNR Number:");
    gbc.gridx = 0;
    gbc.gridy = 1;
    gbc.gridwidth = 1;
    currentPanel.add(pnrLabel, gbc);
```

```
    JTextField pnrField = new JTextField(15);
    gbc.gridx = 1;
    gbc.gridy = 1;
```

```
currentPanel.add(pnrField, gbc);
```

```
JButton checkButton = new JButton("Check Status");
```

```
gbc.gridx = 0;
```

```
gbc.gridy = 2;
```

```
gbc.gridwidth = 2;
```

```
currentPanel.add(checkButton, gbc);
```

```
JButton backButton = new JButton("Back to Main Menu");
```

```
gbc.gridx = 0;
```

```
gbc.gridy = 3;
```

```
currentPanel.add(backButton, gbc);
```

```
outputArea = new JTextArea(10, 30);
```

```
outputArea.setEditable(false);
```

```
JScrollPane scrollPane = new JScrollPane(outputArea);
```

```
gbc.gridx = 0;
```

```
gbc.gridy = 4;
```

```
gbc.gridwidth = 2;
```

```
gbc.fill = GridBagConstraints.BOTH;
```

```
currentPanel.add(scrollPane, gbc);
```

```
checkButton.addActionListener(e -> {
```

```
    String pnr = pnrField.getText().trim();
```

```
    if (pnr.isEmpty()) {
```

```
        JOptionPane.showMessageDialog(this, "Please enter a PNR number",  
"Error", JOptionPane.ERROR_MESSAGE);
```

```
        return;
```

```
    }
```

```
    try {
```

```
        ResultSet rs = Booking.getPNRStatus(pnr);
```

```
        StringBuilder status = new StringBuilder();
```

```
        boolean found = false;
```

```
        while (rs.next()) {
```

```
            found = true;
```

```

        status.append("PNR: ").append(rs.getString("pnr_number"))
            .append("\nTrain: ").append(rs.getString("train_name"))
            .append(" (").append(rs.getString("train_number")).append(")")
            .append("\nFrom: ").append(rs.getString("from_station"))
            .append("\nTo: ").append(rs.getString("to_station"))
            .append("\nDate: ").append(rs.getDate("journey_date"))
            .append("\nClass: ").append(rs.getString("class_type"))
            .append("\nPassenger: ").append(rs.getString("passenger_name"))
            .append("\nStatus: ").append(rs.getString("booking_status"))
            .append("\nSeat: ").append(rs.getString("seat_number"))
            .append("\nFare: ₹").append(rs.getDouble("fare"))
            .append("\n\n");
    }
    if (!found) {
        outputArea.setText("No booking found for PNR: " + pnr);
    } else {
        outputArea.setText(status.toString());
    }
} catch (SQLException ex) {
    outputArea.setText("Error retrieving PNR status: " + ex.getMessage());
}
});

```

```

backButton.addActionListener(e -> showMainMenu());

```

```

    applyTheme();
    currentPanel.revalidate();
    currentPanel.repaint();
}

```

```

private void showProfilePanel() {
    currentPanel.removeAll();
    currentPanel.setLayout(new GridBagLayout());
    GridBagConstraints gbc = new GridBagConstraints();
    gbc.insets = new Insets(10, 10, 10, 10);
}

```

```
gbc.fill = GridBagConstraints.HORIZONTAL;
```

```
JLabel titleLabel = new JLabel("Manage Profile", SwingConstants.CENTER);  
titleLabel.setFont(new Font("Arial", Font.BOLD, 24));  
gbc.gridx = 0;  
gbc.gridy = 0;  
gbc.gridwidth = 2;  
currentPanel.add(titleLabel, gbc);
```

```
JLabel ageLabel = new JLabel("Age:");  
gbc.gridx = 0;  
gbc.gridy = 1;  
gbc.gridwidth = 1;  
currentPanel.add(ageLabel, gbc);
```

```
JTextField ageField = new JTextField(15);  
gbc.gridx = 1;  
gbc.gridy = 1;  
currentPanel.add(ageField, gbc);
```

```
JLabel genderLabel = new JLabel("Gender:");  
gbc.gridx = 0;  
gbc.gridy = 2;  
currentPanel.add(genderLabel, gbc);
```

```
JComboBox<String> genderCombo = new JComboBox<>(new  
String[]{"Male", "Female", "Other"});  
gbc.gridx = 1;  
gbc.gridy = 2;  
currentPanel.add(genderCombo, gbc);
```

```
JLabel classLabel = new JLabel("Preferred Class:");  
gbc.gridx = 0;  
gbc.gridy = 3;  
currentPanel.add(classLabel, gbc);
```

```

        JComboBox<String> classCombo = new
JComboBox<>(Booking.getClassTypes());
        gbc.gridx = 1;
        gbc.gridy = 3;
        currentPanel.add(classCombo, gbc);

        JLabel govIdLabel = new JLabel("Government ID:");
        gbc.gridx = 0;
        gbc.gridy = 4;
        currentPanel.add(govIdLabel, gbc);

        JTextField govIdField = new JTextField(15);
        gbc.gridx = 1;
        gbc.gridy = 4;
        currentPanel.add(govIdField, gbc);

        JLabel emergencyLabel = new JLabel("Emergency Contact:");
        gbc.gridx = 0;
        gbc.gridy = 5;
        currentPanel.add(emergencyLabel, gbc);

        JTextField emergencyField = new JTextField(15);
        gbc.gridx = 1;
        gbc.gridy = 5;
        currentPanel.add(emergencyField, gbc);

        JLabel wheelchairLabel = new JLabel("Wheelchair Access:");
        gbc.gridx = 0;
        gbc.gridy = 6;
        currentPanel.add(wheelchairLabel, gbc);

        JCheckBox wheelchairCheck = new JCheckBox();
        gbc.gridx = 1;
        gbc.gridy = 6;

```



```
currentPanel.add(wheelchairCheck, gbc);
```

```
JLabel medicalLabel = new JLabel("Medical Condition:");
```

```
gbc.gridx = 0;
```

```
gbc.gridy = 7;
```

```
currentPanel.add(medicalLabel, gbc);
```

```
JCheckBox medicalCheck = new JCheckBox();
```

```
gbc.gridx = 1;
```

```
gbc.gridy = 7;
```

```
currentPanel.add(medicalCheck, gbc);
```

```
JButton saveButton = new JButton("Save Profile");
```

```
gbc.gridx = 0;
```

```
gbc.gridy = 8;
```

```
gbc.gridwidth = 2;
```

```
currentPanel.add(saveButton, gbc);
```

```
JButton backButton = new JButton("Back to Main Menu");
```

```
gbc.gridx = 0;
```

```
gbc.gridy = 9;
```

```
currentPanel.add(backButton, gbc);
```

```
// Load existing profile if available
```

```
try {
```

```
    Profile profile = Profile.getProfile(currentUser.getUserId());
```

```
    if (profile != null) {
```

```
        ageField.setText(String.valueOf(profile.getAge()));
```

```
        genderCombo.setSelectedItem(profile.getGender());
```

```
        classCombo.setSelectedItem(profile.getPreferredClass());
```

```
        govIdField.setText(profile.getGovId());
```

```
        emergencyField.setText(profile.getEmergencyContact());
```

```
        wheelchairCheck.setSelected(profile.isWheelchairAccess());
```

```
        medicalCheck.setSelected(profile.isMedicalCondition());
```

```
    }
```

```

    } catch (SQLException ex) {
        JOptionPane.showMessageDialog(this, "Error loading profile: " +
ex.getMessage(), "Error", JOptionPane.ERROR_MESSAGE);
    }

saveButton.addActionListener(e -> {
    try {
        int age = Integer.parseInt(ageField.getText().trim());
        String gender = (String) genderCombo.getSelectedItem();
        String preferredClass = (String) classCombo.getSelectedItem();
        String govId = govIdField.getText().trim();
        String emergencyContact = emergencyField.getText().trim();
        boolean wheelchairAccess = wheelchairCheck.isSelected();
        boolean medicalCondition = medicalCheck.isSelected();

        if (govId.isEmpty() || emergencyContact.isEmpty()) {
            JOptionPane.showMessageDialog(this, "Please fill all required fields",
"Error", JOptionPane.ERROR_MESSAGE);
            return;
        }

        Profile profile = new Profile(
            currentUser.getUserId(),
            age,
            gender,
            preferredClass,
            govId,
            emergencyContact,
            wheelchairAccess,
            medicalCondition
        );
        profile.save();
        JOptionPane.showMessageDialog(this, "Profile saved successfully!",
"Success", JOptionPane.INFORMATION_MESSAGE);
        showMainMenu();
    }
});

```

```
        } catch (NumberFormatException ex) {
            JOptionPane.showMessageDialog(this, "Invalid age format", "Error",
JOptionPane.ERROR_MESSAGE);
        } catch (SQLException ex) {
            JOptionPane.showMessageDialog(this, "Error saving profile: " +
ex.getMessage(), "Error", JOptionPane.ERROR_MESSAGE);
        }
    });
```

```
backButton.addActionListener(e -> showMainMenu());
```

```
    applyTheme();
    currentPanel.revalidate();
    currentPanel.repaint();
}
```

```
private void showAccountBalance() {
    currentPanel.removeAll();
    currentPanel.setLayout(new GridBagLayout());
    GridBagConstraints gbc = new GridBagConstraints();
    gbc.insets = new Insets(10, 10, 10, 10);
    gbc.fill = GridBagConstraints.HORIZONTAL;

    JLabel titleLabel = new JLabel("Account Balance", SwingConstants.CENTER);
    titleLabel.setFont(new Font("Arial", Font.BOLD, 24));
    gbc.gridx = 0;
    gbc.gridy = 0;
    gbc.gridwidth = 2;
    currentPanel.add(titleLabel, gbc);

    JLabel balanceLabel = new JLabel("Current Balance: ");
    gbc.gridx = 0;
    gbc.gridy = 1;
    gbc.gridwidth = 1;
    currentPanel.add(balanceLabel, gbc);
}
```

```

    JTextField balanceField = new JTextField(15);
    balanceField.setEditable(false);
    gbc.gridx = 1;
    gbc.gridy = 1;
    currentPanel.add(balanceField, gbc);

    JButton backButton = new JButton("Back to Main Menu");
    gbc.gridx = 0;
    gbc.gridy = 2;
    gbc.gridwidth = 2;
    currentPanel.add(backButton, gbc);

    try {
        double balance =
        DBConnection.getAccountBalance(currentUser.getUserId());
        balanceField.setText(String.format("₹%.2f", balance));
    } catch (SQLException ex) {
        balanceField.setText("Error retrieving balance");
        JOptionPane.showMessageDialog(this, "Error retrieving balance: " +
ex.getMessage(), "Error", JOptionPane.ERROR_MESSAGE);
    }

    backButton.addActionListener(e -> showMainMenu());

    applyTheme();
    currentPanel.revalidate();
    currentPanel.repaint();
}

public static void main(String[] args) {
    SwingUtilities.invokeLater(() -> new TrainManagementSystem());
}
}

```

## 9.2 Explanation code

### Key Concepts Used in the Code

#### 9.1. Abstraction

Abstraction is the concept of hiding complex implementation details and exposing only the necessary features to the user. It simplifies interaction with the system by providing a clear interface.

- **In the Code:**
  - **Database Operations (DBConnection Class):**
    - The DBConnection class abstracts the database interaction details. Methods like getConnection(), deductBalance(), and logSession() hide the underlying SQL queries and JDBC mechanics.

For example, deductBalance(int userId, double amount) abstracts the process of checking the balance, deducting the amount, and committing/rolling back the transaction:

```
java
Copy
public static boolean deductBalance(int userId, double
amount) throws SQLException {
    try (Connection conn = getConnection()) {
        conn.setAutoCommit(false);
        String checkSql = "SELECT balance FROM
AccountHolder WHERE user_id = ?";
        // ... balance deduction logic ...
        conn.commit();
        return true;
    }
```

■ }

The user of this method doesn't need to know about the SQL queries or transaction management; they only need to provide the `userId` and `amount`.

- **GUI Panels (TrainManagementSystem Class):**

The `TrainManagementSystem` class abstracts the UI creation process by providing methods like `showLoginPanel()`, `showTrainSearchPanel()`, and `showPassengerDetailsPanel()`. These methods encapsulate the details of creating Swing components (`JLabel`, `TextField`, etc.) and their layouts, presenting a simplified interface to switch between panels.

java

Copy

```
private void showLoginPanel() {  
    currentPanel.removeAll();  
    currentPanel.setLayout(new GridBagLayout());  
    // ... add Swing components ...
```

■ }

- **Business Logic (Booking Class):**

The `Booking` class abstracts the ticket booking process in the `bookTickets()` method. It handles fare calculation, payment, and database storage, but the caller only needs to provide inputs like `userId`, `trainNumber`, and `passengers`:

java

Copy

```
public static String bookTickets(int userId, String  
trainNumber, String trainName,  
                                String fromStation,  
String toStation, String journeyDate,  
                                String classType,  
String quota, List<Passenger> passengers,
```

```

                                String paymentMethod)
throws SQLException {
    // Fare calculation, payment, and booking logic
    abstracted

}

```

## 9.2. Encapsulation

Encapsulation is the bundling of data (fields) and methods that operate on that data within a single unit (class), while restricting direct access to the data using access modifiers (e.g., private).

- **In the Code:**
  - **Class Fields:**

Most classes (User, Train, Profile, Booking.Passenger) encapsulate their fields by declaring them as private and providing public getter methods:

```

java
Copy
class User {
    private int userId;
    private String name;
    public int getUserId() { return userId; }
    public String getName() { return name; }

}

```

This ensures that the `userId` and `name` cannot be modified directly from outside the class, protecting the integrity of the data.

- **Database Credentials (DBConnection Class):**

The database URL, username, and password are encapsulated as private static final constants in the `DBConnection` class:

```
private static final String URL =  
"jdbc:mysql://localhost:3306/IRCTC_System";  
private static final String USER = "root";
```

```
    private static final String PASSWORD =  
        "root";
```

These fields are not exposed, ensuring that database credentials are not accidentally modified.

- **Passenger Details (Booking.Passenger Class):**

The Passenger class encapsulates passenger data and provides getters but no setters, making the fields immutable after creation:

java

Copy

```
public static class Passenger {  
    private String name;  
    private int age;  
    public String getName() { return name; }  
    public int getAge() { return age; }  
  
}
```

### 9.3. Inheritance

Inheritance allows a class to inherit properties and methods from another class, promoting code reuse.

- **In the Code:**

- **Swing Components (TrainManagementSystem Class):**

The TrainManagementSystem class extends JFrame, inheriting its properties and methods for creating a windowed application:

java



Copy

```
public class TrainManagementSystem extends JFrame {  
    // Inherits JFrame methods like setSize(),  
    setDefaultCloseOperation()  
  
}
```

- This allows TrainManagementSystem to use JFrame's functionality (e.g., setting the window size, layout, and close operation) without reimplementing it.
- **No Explicit Class Inheritance:**
  - Beyond JFrame, the code does not use inheritance extensively for custom classes. Instead, it relies on composition (e.g., a TrainManagementSystem object contains a User object).

## 9.4. Polymorphism

Polymorphism allows objects of different classes to be treated as objects of a common superclass (via inheritance) or to implement the same method in different ways (via method overriding or interfaces).

- **In the Code:**
  - **Method Overloading** (Compile-Time Polymorphism):

The code uses method overloading to provide multiple versions of a method with different parameters. For example, the User class has two constructors:

```
java  
Copy  
public User(String name, String mobile, String email,  
String password,  
String securityQuestion, String  
securityAnswer) {  
    // Constructor for creating a new user  
}
```

```
private User(int userId, String name, String mobile,
String email, String password,
                String securityQuestion, String
securityAnswer, String sessionId) {
    // Constructor for loading a user from the database

    ■ }

```

These constructors allow the User class to be instantiated in different contexts (e.g., during registration vs. login).

- **Method Overriding** (Runtime Polymorphism):

The code doesn't explicitly override methods from superclasses (beyond Swing's inherited methods), but Swing components internally use polymorphism. For example, ActionListener's actionPerformed method is overridden in various button listeners:

```
java
Copy
loginButton.addActionListener(e -> {
    // Custom implementation of actionPerformed

    });

```

- **Dynamic Binding:**

- The List<Train> returned by Train.searchTrainsWithFilters() is an example of polymorphism, as List is an interface, and the actual object is an ArrayList. The caller interacts with the List interface without needing to know the concrete implementation:

```
java
Copy
List<Train> trains = new ArrayList<>();

```

## 9.5. Composition

Composition is a "has-a" relationship where one class contains an instance of another class to reuse its functionality.

- **In the Code:**
  - **TrainManagementSystem and User:**
    - The TrainManagementSystem class contains a currentUser field of type User:  
java  
Copy  

```
private User currentUser;
```

This allows the TrainManagementSystem to manage the logged-in user's details and operations (e.g., booking tickets, viewing history).
  - **Booking and Passenger:**
    - The Booking class uses a List<Passenger> to manage multiple passengers in a single booking:  
java  
Copy  

```
List<Booking.Passenger> passengers = new ArrayList<>();
```

The Passenger class is a nested class within Booking, demonstrating a strong composition relationship.
  - **Swing Components:**
    - The TrainManagementSystem class composes various Swing components (JPanel, JLabel, JButton, etc.) to build the GUI. For example, the currentPanel field is a JPanel that dynamically holds other components:  
java  
Copy  

```
private JPanel currentPanel;
```

## 9.6. Single Responsibility Principle (SRP)

The SRP states that a class should have only one reason to change, meaning it should have a single responsibility.

- **In the Code:**

- Each class has a well-defined responsibility:
  - DBConnection: Manages database connections and operations.
  - User: Handles user-related operations (registration, login, logout).
  - Train: Manages train data and search functionality.
  - Booking: Handles ticket booking, cancellation, and history.
  - Profile: Manages user profile data.
  - TrainManagementSystem: Manages the GUI and user interactions.
- For example, the Booking class focuses solely on booking-related logic, such as fare calculation and database storage, without handling UI or database connection details.

## 9.7. Modularity

Modularity refers to dividing a system into smaller, independent modules that can be developed and tested separately.

- **In the Code:**

- The code is modular, with separate classes for different concerns (DBConnection, User, Train, etc.).
- The TrainManagementSystem class uses modular UI methods (showLoginPanel(), showTrainSearchPanel(), etc.) to manage different screens, making it easy to add or modify panels without affecting the rest of the system.

## 9.8. Exception Handling

Exception handling ensures that the program can handle errors gracefully without crashing.

- **In the Code:**

The code uses try-catch blocks to handle SQLException and ParseException:

java

Copy

```
try {  
    List<Train> trains =  
Train.searchTrainsWithFilters(from, to, date,  
classType, trainType, timeRange);  
} catch (SQLException ex) {  
    JOptionPane.showMessageDialog(this, "Error  
searching trains: " + ex.getMessage(), "Error",  
JOptionPane.ERROR_MESSAGE);  
    }  
}
```

Database transactions use commit and rollback to ensure data consistency:

java

Copy

```
conn.setAutoCommit(false);  
try {  
    // ... database operation ...  
    conn.commit();  
} catch (SQLException e) {  
    conn.rollback();  
    throw e;  
    }  
}
```

## 9.9. Event-Driven Programming

Event-driven programming is a design pattern where the control flow is determined by events, such as user actions (e.g., button clicks).

- **In the Code:**

The GUI in TrainManagementSystem uses Swing's event-driven model. Buttons and other components have ActionListeners that respond to user actions:

```
java
Copy
loginButton.addActionListener(e -> {
    String email = emailField.getText();
    String password = new
String(passwordField.getPassword());
    currentUser = User.login(email, password);
    // ... handle login ...

    ○ });
```

- This allows the application to respond dynamically to user inputs, such as logging in, searching for trains, or booking tickets.

## 9.10. Data Persistence

Data persistence refers to storing data in a non-volatile storage medium (e.g., a database) so that it can be retrieved later.

- **In the Code:**

- The application uses a MySQL database (IRCTC\_System) to persist data:
  - Users table for user details.
  - Trains table for train data.
  - Bookings table for booking records.
  - AccountHolder table for user balances.
  - Profiles table for user profiles.
  - SessionLogs table for logging user actions.
- The DBConnection class handles all database interactions, ensuring that data is saved and retrieved reliably.

## 9.11. Static Members

Static members belong to the class rather than an instance, making them shared across all instances.

- **In the Code:**

The Booking class uses static fields and methods for shared data, such as class types and fares:

java

Copy

```
private static final String[] CLASS_TYPES = {"Anubhuti  
Class (EA)", "AC First Class (1A)", ...};  
private static final Map<String, Double> CLASS_FARES =  
new HashMap<>();  
static {  
    CLASS_FARES.put("Anubhuti Class (EA)", 4500.0);  
    // ... initialize fares ...  
    }
```

The DBConnection class uses static methods for database operations, as they don't depend on instance state:

java

Copy

```
public static Connection getConnection() throws  
SQLException {  
    return DriverManager.getConnection(URL, USER,  
PASSWORD);  
    }
```

## 9.12. MVC Pattern (Implicit)

The Model-View-Controller (MVC) pattern separates the application into three interconnected components:

- **Model:** Represents the data and business logic.
- **View:** Handles the user interface.
- **Controller:** Manages user input and updates the model and view.
- **In the Code:**
  - **Model:** Classes like User, Train, Booking, and Profile represent the data and logic.
  - **View:** The TrainManagementSystem class (Swing GUI) displays the interface.

**Controller:** The TrainManagementSystem class also acts as the controller, handling user inputs (via ActionListeners) and updating the model and view:

java

Copy

```
searchButton.addActionListener(e -> {  
    String from = fromField.getText();  
    String to = toField.getText();  
    // ... update model (search trains) and view  
    (display results) ...  
    });
```

### 9.13. Factory-Like Behavior

The factory pattern involves creating objects without specifying the exact class of the object to be created.

- **In the Code:**

The User.login() method acts like a factory method, creating and returning a User object based on database data:

java

Copy



```

public static User login(String email, String password)
throws SQLException {
    // ... query database ...
    if (rs.next()) {
        return new User(
            rs.getInt("user_id"),
            rs.getString("name"),
            // ... other fields ...
        );
    }
    return null;
}

```

## 9.14. Immutability

Immutability ensures that an object's state cannot be changed after it is created.

- **In the Code:**

The `Booking.Passenger` class is immutable, as it provides only getters and no setters:

java

Copy

```

public static class Passenger {
    private String name;
    private int age;

    public Passenger(String name, int age, String
gender, String berthPreference, String concessionType)
{
    this.name = name;
    this.age = age;
}
}

```

```

        // ... initialize fields ...
    }
    public String getName() { return name; } // No
setter

    ○ }

```

- This ensures that passenger details cannot be modified after the object is created, maintaining data consistency during the booking process.

### 9.15. Randomization

Randomization is used to simulate dynamic behavior, such as seat availability or PNR generation.

- **In the Code:**

The `Train.getAvailabilityForClass()` method uses `Random` to simulate seat availability:

```

java
Copy
public String getAvailabilityForClass(String classType)
{
    Random random = new Random();
    int availability = random.nextInt(100);
    if (availability > 80) {
        return "WL " + (random.nextInt(50) + 1);
    } else if (availability > 60) {
        return "RAC " + (random.nextInt(20) + 1);
    } else {
        return "Available";
    }
}

```

- }

The `Booking.generatePNR()` method generates a random PNR number:

java

Copy

```
private static String generatePNR() {  
    Random random = new Random();  
    return "PNR" + (100000 + random.nextInt(900000));
```

- }

## Additional Design Principles

### 1. DRY (Don't Repeat Yourself)

The code avoids repetition by reusing methods and components:

The `applyTheme()` method centralizes theme application logic:

java

Copy

```
private void applyTheme() {  
    if (darkTheme) {  
        currentPanel.setBackground(new Color(60, 63,  
65));  
        // ... apply dark theme ...  
    } else {  
        // ... apply light theme ...  
    }  
}
```

- }

This method is called whenever the panel changes or the theme toggles, avoiding repeated code.

## 2. Separation of Concerns

The code separates concerns by dividing responsibilities:

- Database operations are handled by DBConnection.
- Business logic (e.g., booking, train search) is handled by Booking and Train.
- UI logic is managed by TrainManagementSystem.

## 3. Error Reporting

The application uses JOptionPane to report errors to the user in a user-friendly way:

java

Copy

```
JOptionPane.showMessageDialog(this, "Invalid email or  
password", "Login Failed", JOptionPane.ERROR_MESSAGE);
```

## Code Structure and Workflow

### Class Responsibilities

- **DBConnection**: Database connectivity and operations.
- **User**: User management (registration, login, logout).
- **Profile**: User profile management.
- **Train**: Train data and search functionality.
- **Booking**: Ticket booking, cancellation, and history.
- **TrainManagementSystem**: GUI and user interaction.

### Workflow

1. **Startup**: The TrainManagementSystem constructor initializes the GUI and database (DBConnection.initializeDatabase()).
2. **User Interaction**: Users log in, search for trains, book tickets, and manage their profiles through the GUI.

3. **Business Logic:** The Booking and Train classes handle the core logic, interacting with the database via DBConnection.
4. **Data Persistence:** All data is stored in the MySQL database, ensuring persistence across sessions.

### Potential Improvements

1. **Security:** Hash passwords using a library like BCrypt instead of storing them as plain text.
2. **Real Availability System:** Replace the random availability simulation with a proper seat allocation system.
3. **Input Validation:** Add more robust validation for user inputs (e.g., email format, date format).
4. **Dependency Injection:** Use dependency injection to decouple classes (e.g., pass DBConnection as a dependency).
5. **Unit Testing:** Add unit tests to verify the behavior of classes like Booking and Train.

### 10.SQL Code

-- Create the IRCTC\_System database

```
CREATE DATABASE IF NOT EXISTS IRCTC_System;
```

```
USE IRCTC_System;
```

-- Create Users table

```
CREATE TABLE IF NOT EXISTS Users (  
    user_id INT AUTO_INCREMENT PRIMARY KEY,  
    name VARCHAR(100) NOT NULL,  
    mobile VARCHAR(15) NOT NULL,
```

```
email VARCHAR(100) UNIQUE NOT NULL,  
password VARCHAR(100) NOT NULL,  
security_question VARCHAR(200) NOT NULL,  
security_answer VARCHAR(100) NOT NULL  
);
```

-- Create Trains table

```
CREATE TABLE IF NOT EXISTS Trains (  
    train_id INT AUTO_INCREMENT PRIMARY KEY,  
    train_number VARCHAR(10) UNIQUE NOT NULL,  
    train_name VARCHAR(100) NOT NULL,  
    source_station VARCHAR(100) NOT NULL,  
    destination_station VARCHAR(100) NOT NULL,  
    departure_time VARCHAR(10) NOT NULL,  
    arrival_time VARCHAR(10) NOT NULL,  
    duration VARCHAR(10) NOT NULL,  
    runs_on VARCHAR(50) NOT NULL  
);
```

-- Create Bookings table

```
CREATE TABLE IF NOT EXISTS Bookings (
```

```
booking_id INT AUTO_INCREMENT PRIMARY KEY,  
user_id INT NOT NULL,  
train_number VARCHAR(10) NOT NULL,  
train_name VARCHAR(100) NOT NULL,  
from_station VARCHAR(100) NOT NULL,  
to_station VARCHAR(100) NOT NULL,  
journey_date DATE NOT NULL,  
class_type VARCHAR(50) NOT NULL,  
passenger_name VARCHAR(100) NOT NULL,  
age INT NOT NULL,  
gender VARCHAR(10) NOT NULL,  
berth_preference VARCHAR(20),  
fare DECIMAL(10,2) NOT NULL,  
pnr_number VARCHAR(15) UNIQUE NOT NULL,  
booking_status VARCHAR(20) DEFAULT 'Confirmed',  
seat_number VARCHAR(10),  
booking_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,  
FOREIGN KEY (user_id) REFERENCES Users(user_id)  
);  
  
-- Check if seat_number column exists in Bookings table and add it if it doesn't
```

```

SET @column_exists = (SELECT COUNT(*)
                        FROM INFORMATION_SCHEMA.COLUMNS
                        WHERE TABLE_NAME = 'Bookings'
                        AND COLUMN_NAME = 'seat_number'
                        AND TABLE_SCHEMA = 'IRCTC_System');

SET @sql = IF(@column_exists = 0,
              'ALTER TABLE Bookings ADD COLUMN seat_number VARCHAR(10)
              AFTER booking_status',
              'SELECT 1');

PREPARE stmt FROM @sql;

EXECUTE stmt;

DEALLOCATE PREPARE stmt;

-- Create AccountHolder table

CREATE TABLE IF NOT EXISTS AccountHolder (
    account_holder_id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100),
    age INT,
    account_number VARCHAR(20) UNIQUE NOT NULL,
    balance DECIMAL(15, 2),
    interest DECIMAL(5, 2),
    user_id INT,

```



```
FOREIGN KEY (user_id) REFERENCES Users(user_id)
);
```

-- Create Profiles table

```
CREATE TABLE IF NOT EXISTS Profiles (
    profile_id INT AUTO_INCREMENT PRIMARY KEY,
    user_id INT NOT NULL,
    age INT,
    gender VARCHAR(10),
    preferred_class VARCHAR(50),
    gov_id VARCHAR(20),
    emergency_contact VARCHAR(15),
    wheelchair_access BOOLEAN DEFAULT FALSE,
    medical_condition BOOLEAN DEFAULT FALSE,
    FOREIGN KEY (user_id) REFERENCES Users(user_id)
);
```

-- Create SessionLogs table

```
CREATE TABLE IF NOT EXISTS SessionLogs (
    log_id INT AUTO_INCREMENT PRIMARY KEY,
    user_id INT,
```

```
action VARCHAR(100) NOT NULL,  
timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP,  
session_id VARCHAR(50)  
);
```

-- Insert sample train data

```
INSERT IGNORE INTO Trains (train_number, train_name, source_station,  
destination_station, departure_time, arrival_time, duration, runs_on) VALUES  
(12001, 'Shatabdi Express', 'New Delhi', 'Bhopal', '08:10', '16:25', '08:15', 'Daily'),  
(12309, 'Rajdhani Express', 'New Delhi', 'Kolkata', '17:00', '10:55', '17:55', 'Daily'),  
(12621, 'Tamil Nadu Express', 'New Delhi', 'Chennai', '22:30', '07:10', '32:40',  
'Daily'),  
(12295, 'Duronto Express', 'Mumbai', 'Kolkata', '17:15', '18:50', '25:35',  
'Mon,Wed,Fri'),  
(12055, 'Jan Shatabdi', 'Dehradun', 'New Delhi', '15:20', '21:10', '05:50', 'Daily'),  
(12951, 'Rajdhani Express', 'Mumbai', 'Delhi', '16:35', '08:15', '15:40', 'Daily'),  
(12952, 'Rajdhani Express', 'Delhi', 'Mumbai', '16:55', '08:30', '15:35', 'Daily'),  
(12301, 'Duronto Express', 'Howrah', 'Delhi', '22:05', '08:00', '09:55',  
'Tue,Thu,Sat'),  
(12302, 'Duronto Express', 'Delhi', 'Howrah', '22:45', '08:40', '09:55',  
'Mon,Wed,Fri'),  
(12305, 'Duronto Express', 'Sealdah', 'Delhi', '22:50', '09:40', '10:50', 'Daily');
```

-- Insert a sample user

```
INSERT IGNORE INTO Users (name, mobile, email, password, security_question, security_answer) VALUES
```

```
('Admin User', '9876543210', 'admin@irctc.com', 'password123', 'What was your first pet's name?', 'Fluffy');
```

-- Set user\_id variable for sample data

```
SET @user_id = (SELECT user_id FROM Users WHERE email = 'admin@irctc.com' LIMIT 1);
```

-- Insert sample AccountHolder data

```
INSERT IGNORE INTO AccountHolder (name, age, account_number, balance, interest, user_id) VALUES
```

```
('John Doe', 30, 'ACC1001', 50000.00, 3.5, NULL),
```

```
('Jane Smith', 25, 'ACC1002', 75000.00, 3.5, NULL);
```

-- Insert a sample booking for testing

```
INSERT IGNORE INTO Bookings (user_id, train_number, train_name, from_station, to_station, journey_date, class_type, passenger_name, age, gender, berth_preference, fare, pnr_number, booking_status, seat_number)
```

```
VALUES (@user_id, '12001', 'Shatabdi Express', 'New Delhi', 'Bhopal', '2025-05-05', 'AC 2 Tier (2A)', 'John Doe', 30, 'Male', 'Lower', 2500.00, 'PNR123456', 'Confirmed', '001');
```

-- Sample queries for testing and verification

SELECT \* FROM Users;

SELECT \* FROM Trains;

SELECT \* FROM Bookings;

SELECT \* FROM AccountHolder;

SELECT \* FROM Profiles;

SELECT \* FROM SessionLogs;

-- Find bookings for the sample user

SELECT \* FROM Bookings WHERE user\_id = @user\_id;

-- Find account details for the sample user

SELECT \* FROM AccountHolder WHERE user\_id = @user\_id;

-- Find bookings by PNR number

SELECT \* FROM Bookings WHERE pnr\_number = 'PNR123456';

-- Find trains between two stations

SELECT \* FROM Trains

WHERE source\_station LIKE '%NEW DELHI%'

AND destination\_station LIKE '%BHOPAL%';

-- Update a booking status

```
UPDATE Bookings SET booking_status = 'Cancelled' WHERE pnr_number = 'PNR123456';
```

-- Delete a booking

```
DELETE FROM Bookings WHERE pnr_number = 'PNR123456';
```

-- Delete a user (cascade deletes bookings, account holder, profiles, and session logs)

```
DELETE FROM Bookings WHERE user_id = @user_id;
```

```
DELETE FROM AccountHolder WHERE user_id = @user_id;
```

```
DELETE FROM Profiles WHERE user_id = @user_id;
```

```
DELETE FROM SessionLogs WHERE user_id = @user_id;
```

```
DELETE FROM Users WHERE user_id = @user_id;
```

-- Set the root password

```
ALTER USER 'root'@'localhost' IDENTIFIED BY 'root';
```

## **14.2.Explanation of SQL code**

### **Explanation of the SQL Code for the IRCTC\_System Database**

The SQL script is designed to initialize and populate the IRCTC\_System database, which supports a train booking system developed in Java (TrainManagementSystem.java). The script creates the database, defines its schema (tables and relationships), inserts sample data, and provides test queries to verify the setup. It also includes maintenance operations such as updates and deletions.

## 10.1. Database Creation and Selection

sql

Copy

```
CREATE DATABASE IF NOT EXISTS IRCTC_System;  
  
USE IRCTC_System;
```

- **Purpose:** Creates a database named IRCTC\_System if it does not already exist and sets it as the active database for subsequent operations.
  - **Details:**
    - CREATE DATABASE IF NOT EXISTS: Ensures the database is created only if it doesn't already exist, preventing errors if the database is already present.
    - USE IRCTC\_System: Selects IRCTC\_System as the default database for all following SQL statements.
- 

## 10.2. Table Creation

The script creates six tables to store data for users, trains, bookings, account holders, profiles, and session logs. Each table is designed with appropriate columns, data types, constraints, and relationships.

### a. Users Table

sql

Copy

```
CREATE TABLE IF NOT EXISTS Users (  
    user_id INT AUTO_INCREMENT PRIMARY KEY,  
    name VARCHAR(100) NOT NULL,  
    mobile VARCHAR(15) NOT NULL,  
    email VARCHAR(100) UNIQUE NOT NULL,  
    password VARCHAR(100) NOT NULL,  
    security_question VARCHAR(200) NOT NULL,  
    security_answer VARCHAR(100) NOT NULL  
);
```

- **Purpose:** Stores user information for authentication and account management.
- **Columns:**
  - user\_id: Auto-incrementing primary key to uniquely identify each user.
  - name, mobile, email, password, security\_question, security\_answer: Store user details, with NOT NULL constraints ensuring these fields are mandatory.
  - email has a UNIQUE constraint to prevent duplicate email addresses.
- **Use Case:** Used for user registration, login, and password recovery in the application.

## b. Trains Table

sql

Copy

```
CREATE TABLE IF NOT EXISTS Trains (  
    train_id INT AUTO_INCREMENT PRIMARY KEY,  
    train_number VARCHAR(10) UNIQUE NOT NULL,  
    train_name VARCHAR(100) NOT NULL,  
    source_station VARCHAR(100) NOT NULL,  
    destination_station VARCHAR(100) NOT NULL,  
    departure_time VARCHAR(10) NOT NULL,  
    arrival_time VARCHAR(10) NOT NULL,  
    duration VARCHAR(10) NOT NULL,  
    runs_on VARCHAR(50) NOT NULL  
);
```

- **Purpose:** Stores train details for the booking system.
- **Columns:**
  - train\_id: Auto-incrementing primary key.
  - train\_number: Unique identifier for each train (e.g., "12001").
  - train\_name, source\_station, destination\_station, departure\_time, arrival\_time, duration, runs\_on: Store train metadata, all mandatory.
- **Use Case:** Used to search for trains based on user criteria (e.g., source, destination).

### c. Bookings Table

sql

Copy

```
CREATE TABLE IF NOT EXISTS Bookings (  

```



```
booking_id INT AUTO_INCREMENT PRIMARY KEY,  
user_id INT NOT NULL,  
train_number VARCHAR(10) NOT NULL,  
train_name VARCHAR(100) NOT NULL,  
from_station VARCHAR(100) NOT NULL,  
to_station VARCHAR(100) NOT NULL,  
journey_date DATE NOT NULL,  
class_type VARCHAR(50) NOT NULL,  
passenger_name VARCHAR(100) NOT NULL,  
age INT NOT NULL,  
gender VARCHAR(10) NOT NULL,  
berth_preference VARCHAR(20),  
fare DECIMAL(10,2) NOT NULL,  
pnr_number VARCHAR(15) UNIQUE NOT NULL,  
booking_status VARCHAR(20) DEFAULT 'Confirmed',  
seat_number VARCHAR(10),  
booking_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,  
FOREIGN KEY (user_id) REFERENCES Users(user_id)  
);
```

- **Purpose:** Stores booking details for train tickets.
- **Columns:**
  - booking\_id: Auto-incrementing primary key.
  - user\_id: Foreign key referencing the Users table, linking the booking to a user.
  - train\_number, train\_name, from\_station, to\_station, journey\_date, class\_type: Booking metadata.
  - passenger\_name, age, gender, berth\_preference: Passenger details.
  - fare: Ticket cost (decimal with 2 places).
  - pnr\_number: Unique identifier for the booking.
  - booking\_status: Status of the booking (default is "Confirmed").
  - seat\_number: Assigned seat number (optional).
  - booking\_date: Timestamp of when the booking was made.
- **Constraints:**
  - FOREIGN KEY (user\_id) REFERENCES Users(user\_id): Ensures referential integrity between Bookings and Users.
- **Use Case:** Used to save, retrieve, and manage ticket bookings.

#### d. Dynamic Column Addition for Bookings Table

sql

Copy

```
SET @column_exists = (SELECT COUNT(*)
                        FROM INFORMATION_SCHEMA.COLUMNS
                        WHERE TABLE_NAME = 'Bookings'
                        AND COLUMN_NAME = 'seat_number'
                        AND TABLE_SCHEMA =
'IRCTC_System');

SET @sql = IF(@column_exists = 0,
```

```
'ALTER TABLE Bookings ADD COLUMN  
seat_number VARCHAR(10) AFTER booking_status',
```

```
'SELECT 1');
```

```
PREPARE stmt FROM @sql;
```

```
EXECUTE stmt;
```

```
DEALLOCATE PREPARE stmt;
```

- **Purpose:** Dynamically adds the seat\_number column to the Bookings table if it doesn't already exist.
- **Details:**
  - Uses INFORMATION\_SCHEMA.COLUMNS to check if the seat\_number column exists.
  - If it doesn't exist (@column\_exists = 0), the script adds the column using ALTER TABLE.
  - Uses prepared statements (PREPARE, EXECUTE, DEALLOCATE PREPARE) for dynamic SQL execution.
- **Use Case:** Ensures backward compatibility if the database schema evolves.

#### e. AccountHolder Table

sql

Copy

```
CREATE TABLE IF NOT EXISTS AccountHolder (  
    account_holder_id INT AUTO_INCREMENT PRIMARY KEY,  
    name VARCHAR(100),  
    age INT,  
    account_number VARCHAR(20) UNIQUE NOT NULL,
```

```

    balance DECIMAL(15, 2),
    interest DECIMAL(5, 2),
    user_id INT,
    FOREIGN KEY (user_id) REFERENCES Users(user_id)
);

```

- **Purpose:** Stores account details for users, including their balance for payments.
- **Columns:**
  - account\_holder\_id: Auto-incrementing primary key.
  - name, age, account\_number, balance, interest, user\_id: Account details.
  - account\_number is unique to prevent duplicates.
- **Constraints:**
  - FOREIGN KEY (user\_id) REFERENCES Users(user\_id): Links the account to a user (optional, as user\_id can be NULL for sample data).
- **Use Case:** Used for managing user balances and processing payments/refunds.

#### f. Profiles Table

sql

Copy

```

CREATE TABLE IF NOT EXISTS Profiles (
    profile_id INT AUTO_INCREMENT PRIMARY KEY,
    user_id INT NOT NULL,
    age INT,

```

```

gender VARCHAR(10),

preferred_class VARCHAR(50),

gov_id VARCHAR(20),

emergency_contact VARCHAR(15),

wheelchair_access BOOLEAN DEFAULT FALSE,

medical_condition BOOLEAN DEFAULT FALSE,

FOREIGN KEY (user_id) REFERENCES Users(user_id)

);

```

- **Purpose:** Stores user profile information for personalized settings.
- **Columns:**
  - profile\_id: Auto-incrementing primary key.
  - user\_id: Foreign key linking to the Users table.
  - age, gender, preferred\_class, gov\_id, emergency\_contact, wheelchair\_access, medical\_condition: Profile details.
- **Use Case:** Used to save and retrieve user preferences and accessibility needs.

#### **g. SessionLogs Table**

sql

Copy

```

CREATE TABLE IF NOT EXISTS SessionLogs (

    log_id INT AUTO_INCREMENT PRIMARY KEY,

    user_id INT,

    action VARCHAR(100) NOT NULL,

```

```
timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP,  
  
session_id VARCHAR(50)  
  
);
```

- **Purpose:** Logs user actions for auditing and debugging.
  - **Columns:**
    - log\_id: Auto-incrementing primary key.
    - user\_id: References the user who performed the action (optional, as it can be NULL).
    - action: Describes the action (e.g., "login", "book\_tickets").
    - timestamp: Records when the action occurred.
    - session\_id: Unique identifier for the user session.
  - **Use Case:** Tracks user activities for security and troubleshooting.
- 

### 3. Sample Data Insertion

The script inserts sample data to populate the tables for testing purposes.

#### a. Insert Sample Trains

sql

Copy

```
INSERT IGNORE INTO Trains (train_number, train_name,  
source_station, destination_station, departure_time,  
arrival_time, duration, runs_on) VALUES  
  
('12001', 'Shatabdi Express', 'New Delhi', 'Bhopal',  
'08:10', '16:25', '08:15', 'Daily'),  
  
('12309', 'Rajdhani Express', 'New Delhi', 'Kolkata',  
'17:00', '10:55', '17:55', 'Daily'),
```

```
-- ... additional train data ...
```

```
('12305', 'Duronto Express', 'Sealdah', 'Delhi',  
'22:50', '09:40', '10:50', 'Daily');
```

- **Purpose:** Adds 10 sample trains to the Trains table.
- **Details:**
  - INSERT IGNORE: Inserts records only if they don't already exist (based on the UNIQUE constraint on train\_number).
  - Includes various train types (e.g., Shatabdi, Rajdhani, Duronto) with realistic routes and schedules.
- **Use Case:** Provides data for train search functionality in the application.

## b. Insert Sample User

sql

Copy

```
INSERT IGNORE INTO Users (name, mobile, email,  
password, security_question, security_answer) VALUES
```

```
('Admin User', '9876543210', 'admin@irctc.com',  
'password123', 'What was your first pet's name?',  
'Fluffy');
```

```
SET @user_id = (SELECT user_id FROM Users WHERE email =  
'admin@irctc.com' LIMIT 1);
```

- **Purpose:** Adds a sample user and retrieves their user\_id for use in subsequent inserts.
- **Details:**
  - Inserts a user with email admin@irctc.com and password password123.

- Stores the user\_id in a variable (@user\_id) for linking with other tables.
- **Use Case:** Provides a test user for booking and account operations.

### c. Insert Sample AccountHolder Data

sql

Copy

```
INSERT IGNORE INTO AccountHolder (name, age,
account_number, balance, interest, user_id) VALUES
('John Doe', 30, 'ACC1001', 50000.00, 3.5, NULL),
('Jane Smith', 25, 'ACC1002', 75000.00, 3.5, NULL);
```

- **Purpose:** Adds sample account holders with initial balances.
- **Details:**
  - user\_id is NULL for these sample accounts, indicating they are not yet linked to a user.
  - Balances are set to 50,000 and 75,000, respectively.
- **Use Case:** Used for testing payment and refund operations.

### d. Insert Sample Booking

sql

Copy

```
INSERT IGNORE INTO Bookings (user_id, train_number,
train_name, from_station, to_station, journey_date,
class_type, passenger_name, age, gender,
berth_preference, fare, pnr_number, booking_status,
seat_number)
```



```
VALUES (@user_id, '12001', 'Shatabdi Express', 'New
Delhi', 'Bhopal', '2025-05-05', 'AC 2 Tier (2A)', 'John
Doe', 30, 'Male', 'Lower', 2500.00, 'PNR123456',
'Confirmed', '001');
```

- **Purpose:** Adds a sample booking for the test user.
  - **Details:**
    - Links the booking to the sample user via @user\_id.
    - Books a ticket on the "Shatabdi Express" for May 5, 2025.
    - Includes passenger details, fare, and a PNR number.
  - **Use Case:** Used for testing booking history, PNR status, and cancellation features.
- 

## 10.4. Test and Verification Queries

The script includes several SELECT queries to verify the data.

### a. Retrieve All Data

sql

Copy

```
SELECT * FROM Users;
```

```
SELECT * FROM Trains;
```

```
SELECT * FROM Bookings;
```

```
SELECT * FROM AccountHolder;
```

```
SELECT * FROM Profiles;
```

```
SELECT * FROM SessionLogs;
```

- **Purpose:** Retrieves all records from each table to verify the data insertion.
- **Use Case:** Ensures that the tables are populated correctly.

#### **b. Find Bookings for the Sample User**

sql

Copy

```
SELECT * FROM Bookings WHERE user_id = @user_id;
```

- **Purpose:** Retrieves bookings for the sample user.
- **Use Case:** Verifies that the sample booking was created correctly.

#### **c. Find Account Details for the Sample User**

sql

Copy

```
SELECT * FROM AccountHolder WHERE user_id = @user_id;
```

- **Purpose:** Retrieves account details for the sample user.
- **Details:** Since the sample accounts have user\_id as NULL, this query will return no results, but it demonstrates how to link accounts to users.

#### **d. Find Bookings by PNR Number**

sql

Copy

```
SELECT * FROM Bookings WHERE pnr_number = 'PNR123456';
```

- **Purpose:** Retrieves booking details for a specific PNR number.
- **Use Case:** Tests the PNR status feature.

#### **e. Find Trains Between Two Stations**

sql

Copy

```
SELECT * FROM Trains  
  
WHERE source_station LIKE '%NEW DELHI%'  
  
AND destination_station LIKE '%BHOPAL%';
```

- **Purpose:** Searches for trains between "New Delhi" and "Bhopal".
- **Details:** Uses LIKE for partial matching, mimicking the train search functionality in the application.
- **Use Case:** Verifies the train search feature.

## 10.5. Maintenance Operations

The script includes operations to update and delete data, demonstrating how to manage the database.

### a. Update a Booking Status

sql

Copy

```
UPDATE Bookings SET booking_status = 'Cancelled' WHERE  
pnr_number = 'PNR123456';
```

- **Purpose:** Changes the status of a booking to "Cancelled".
- **Use Case:** Simulates the ticket cancellation feature, which would also trigger a refund in the application.

### b. Delete a Booking

sql

Copy

```
DELETE FROM Bookings WHERE pnr_number = 'PNR123456';
```

- **Purpose:** Deletes a specific booking by PNR number.
- **Use Case:** Cleans up test data or simulates a full booking deletion.

### c. Delete a User and Related Data

sql

Copy

```
DELETE FROM Bookings WHERE user_id = @user_id;
```

```
DELETE FROM AccountHolder WHERE user_id = @user_id;
```

```
DELETE FROM Profiles WHERE user_id = @user_id;
```

```
DELETE FROM SessionLogs WHERE user_id = @user_id;
```

```
DELETE FROM Users WHERE user_id = @user_id;
```

- **Purpose:** Deletes a user and all related data to maintain referential integrity.
- **Details:**
  - Deletes records from dependent tables (Bookings, AccountHolder, Profiles, SessionLogs) before deleting the user from the Users table.
  - This cascading deletion ensures that no orphaned records remain.
- **Use Case:** Simulates user account deletion in the application.

## 10.6. Set the Root Password

sql

Copy

```
ALTER USER 'root'@'localhost' IDENTIFIED BY 'root';
```

- **Purpose:** Sets the password for the MySQL root user to "root".
- **Details:**

- This matches the credentials used in the Java code (DBConnection class).
- Ensures that the application can connect to the database using these credentials.
- **Use Case:** Configures the database for use with the Java application.

## Summary of the SQL Script

### Purpose

The SQL script sets up the IRCTC\_System database for a train booking system. It defines the schema, populates it with sample data, and provides queries for testing and maintenance. The database supports key functionalities of the TrainManagementSystem.java application, including user management, train search, ticket booking, and payment processing.

### Structure

- **Database:** IRCTC\_System.
- **Tables:**
  - Users: User accounts.
  - Trains: Train schedules.
  - Bookings: Ticket bookings.
  - AccountHolder: User balances.
  - Profiles: User preferences.
  - SessionLogs: Action logs.
- **Relationships:**
  - Foreign keys (user\_id) in Bookings, AccountHolder, Profiles, and SessionLogs link to the Users table.
- **Sample Data:** Includes trains, a user, accounts, and a booking for testing.
- **Queries:** Test queries verify the setup, while update/delete queries demonstrate maintenance.

### Key Features

1. **Schema Design:** Uses appropriate data types, constraints (e.g., NOT NULL, UNIQUE, foreign keys), and defaults (e.g., CURRENT\_TIMESTAMP).

2. **Data Integrity:** Enforces relationships with foreign keys and cascading deletions.
3. **Dynamic SQL:** Adds columns dynamically if missing (e.g., seat\_number in Bookings).
4. **Sample Data:** Provides realistic data for testing application features.
5. **Maintenance:** Includes operations for updating and deleting records.

## Use in the Application

The IRCTC\_System database is integral to the TrainManagementSystem.java application:

- The DBConnection class connects to this database using JDBC.
- The User, Train, Booking, and Profile classes perform CRUD operations on these tables.
- The application's features (e.g., train search, booking, cancellation) rely on this schema for data persistence.

## Potential Improvements

1. **Security:** Hash passwords in the Users table instead of storing them as plain text.
2. **Indexes:** Add indexes on frequently queried columns (e.g., train\_number, pnr\_number) to improve performance.
3. **Triggers:** Use triggers to automate logging or balance updates.
4. **Validation:** Add check constraints (e.g., age  $\geq$  0 in Bookings).

## 11..GUI

## Design

IRCTC Train Booking System

IRCTC - Indian Railway Catering and Tourism Corporation

Welcome, Guest 04-May-2025 [23:30:12]

### Login to IRCTC

Email:

Password:

Login

Register New User

Forgot Password?

Toggle Dark Theme

Customer Care Numbers: 1464 / 80904687989 / 08035734999 | BEWARE OF FRAUDSTERS: Always download official IRCTC Rail Connect App from the Google Play Store or Apple App Store only.

## Search Trains

From:  To:   
Journey Date (dd/mm/yyyy):  Class: **Anubhuti Class (EA)** ▼  
Train Type: **All** ▼ Departure Time: **All** ▼

Search Trains

Back to Main Menu

Customer Care Numbers: 1464 / 80904687989 / 08035734999 | BEWARE OF FRAUDSTERS: Always download official IRCTC Rail Connect App from the Google Play Store or Apple App Store only.

## Your Booking History

PNR	Train No	Train Name	From	To	Journey Date	Class	Status	Fare
-----	----------	------------	------	----	--------------	-------	--------	------

Back to Main Menu

Customer Care Numbers: 1464 / 80904687989 / 08035734999 | BEWARE OF FRAUDSTERS: Always download official IRCTC Rail Connect App from the Google Play Store or Apple App Store only.



### Cancel Ticket

Enter PNR Number:

Check PNR

Cancel Ticket

Back to Main Menu

Customer Care Numbers: 1464 / 80904687989 / 08035734999 | BEWARE OF FRAUDSTERS: Always download official IRCTC Rail Connect App from the Google Play Store or Apple App Store only.

### PNR Status

Enter PNR Number:

Check Status

Back to Main Menu

Customer Care Numbers: 1464 / 80904687989 / 08035734999 | BEWARE OF FRAUDSTERS: Always download official IRCTC Rail Connect App from the Google Play Store or Apple App Store only.

### Manage Profile

Age:

Gender:

Preferred Class:

Government ID:

Emergency Contact:

Wheelchair Access: ☐

Medical Condition: ☐

[Save Profile](#)[Back to Main Menu](#)

Customer Care Numbers: 1464 / 80904687989 / 08035734999 | BEWARE OF FRAUDSTERS: Always download official IRCTC Rail Connect App from the Google Play Store or Apple App Store only.

### Account Balance

Current Balance:

[Back to Main Menu](#)

Customer Care Numbers: 1464 / 80904687989 / 08035734999 | BEWARE OF FRAUDSTERS: Always download official IRCTC Rail Connect App from the Google Play Store or Apple App Store only.

IRCTC Train Booking System

IRCTC - Indian Railway Catering and Tourism Corporation

Welcome, Vinay Gautam 06-May-2025 [11:59:59]

### Search Trains

From: Mumbai

To: Kolkata

Journey Date (dd/mm/yyyy): 05/05/2025

Class: Anubhuti Class (EA)

Train Type: All

Departure Time: All

Search Trains

Back to Main Menu

Train No	Train Name	Departure	Arrival	Duration	From	To	Availability
12295	Duronto Express	Mumbai (17:15)	Kolkata (18:50)	25:35	Mumbai	Kolkata	RAC 9

Book Selected Train

Customer Care Numbers: 1464 / 80904687989 / 08035734999 | BEWARE OF FRAUDSTERS: Always download official IRCTC Rail Connect App from the Google Play Store or Apple App Store only.

IRCTC Train Booking System

IRCTC - Indian Railway Catering and Tourism Corporation

Welcome, Vinay Gautam 06-May-2025 [11:59:59]

### Search Trains

From: Mumbai

To: Kolkata

Journey Date (dd/mm/yyyy): 05/05/2025

Class: Anubhuti Class (EA)

Train Type: All

Departure Time: All

Search Trains

Input

Enter Train Number to book:  
12295

OK Cancel

Train No	Train Name	Departure	Arrival	Duration	From	To	Availability
12295	Duronto Express	Mumbai (17:15)	Kolkata (18:50)	25:35	Mumbai	Kolkata	RAC 9

Book Selected Train

Customer Care Numbers: 1464 / 80904687989 / 08035734999 | BEWARE OF FRAUDSTERS: Always download official IRCTC Rail Connect App from the Google Play Store or Apple App Store only.

## Passenger Details for Durgam Express

Mumbai to Kolkata

Journey Date (dd/mm/yyyy): 05/05/2025

Class:

Anubhuti Class (EA)

Quota:

General

Number of Passengers:

3

Passenger 1	
Name:	vinay Gautam
Age:	18
Gender:	Male
Berth Preference:	Middle
Concession:	Armed Forces
Passenger 2	
Name:	Suraj
Age:	18
Gender:	Male
Berth Preference:	No Preference
Concession:	None
Passenger 3	
Name:	
Age:	18
Gender:	Male
Berth Preference:	No Preference
Concession:	None

Add Passenger Details

Proceed to Payment

## Passenger Details for Durgam Express

Mumbai to Kolkata

Journey Date (dd/mm/yyyy): 05/05/2025

Class:

Anubhuti Class (EA)

Quota:


General

Number of Passengers:

3

Passenger 1	
Name:	vinay Gautam
Age:	18
Gender:	Male
Berth Preference:	Middle
Concession:	
Passenger 2	
Name:	
Age:	18
Gender:	Male
Berth Preference:	Upper
Concession:	Student
Passenger 3	
Name:	Neeraj
Age:	18
Gender:	Male
Berth Preference:	Middle
Concession:	Student

Error

 Booking failed: Insufficient balance to complete the booking.

OK

Add Passenger Details

Proceed to Payment

## **12 Packages are use in the code**

### **Explicitly Imported Packages**

These packages are directly imported at the top of the code:

#### **1. javax.swing**

- Used for creating the graphical user interface (GUI) components such as JFrame, JPanel, JLabel, JButton, JTextField, JPasswordField, JComboBox, JSpinner, JTextArea, JScrollPane, JCheckBox, and JOptionPane.
- Example: JFrame is used to create the main window of the application (TrainManagementSystem extends JFrame).

#### **2. java.awt**

- Provides classes for GUI components and layout management, such as BorderLayout, GridBagLayout, GridBagConstraints, FlowLayout, GridLayout, BoxLayout, Color, Font, and Insets.
- Example: BorderLayout is used to arrange components in the main frame (setLayout(new BorderLayout())).

#### **3. java.awt.event**

- Contains classes for handling user events, such as ActionListener and ActionEvent.
- Example: ActionListener is used to handle button clicks (e.g., loginButton.addActionListener).

#### **4. java.sql**

- Provides classes for interacting with the database, such as Connection, DriverManager, PreparedStatement, Statement, ResultSet, SQLException, and Types.
- Example: DriverManager.getConnection is used in DBConnection.getConnection() to establish a database connection.

#### **5. java.text**

- Includes classes for formatting and parsing dates, such as SimpleDateFormat and ParseException.

- Example: SimpleDateFormat is used to parse and format dates in Train.searchTrainsWithFilters and Booking.convertDate.

## 6. **java.util**

- Contains utility classes like List, ArrayList, Map, HashMap, Random, UUID, and Date.
- Example: ArrayList is used to store a list of trains in Train.searchTrainsWithFilters.

## Implicitly Used Packages

These packages are not explicitly imported but are used because the code references classes or methods from them. Java automatically imports some packages (like java.lang), and others are inferred based on the dependencies of the explicitly imported classes.

### 1. **java.lang**

- Automatically imported in all Java programs.
- Provides fundamental classes like String, Integer, Double, Boolean, Object, System, and Exception.
- Example: String is used throughout the code (e.g., String email = emailField.getText()).

### 2. **java.util.function** *(optional, depending on implementation)*

- If any functional programming constructs (like Consumer, Predicate, etc.) were used with collections, this package would be involved. However, the code doesn't use these directly.
- Example: Not explicitly used, but could be relevant if the code were extended to use streams or lambdas with collections like List<Train>.

### 3. **javax.swing.border**

- Used for border-related classes like BorderFactory and TitledBorder.
- Example: BorderFactory.createEmptyBorder is used in createHeaderPanel to add padding to the header panel.

### 4. **javax.swing.event** *(optional, depending on implementation)*

- Potentially used for handling additional Swing events (e.g., ChangeListener for JSpinner), though the code primarily relies on ActionListener from java.awt.event.
  - Example: JSpinner in showPassengerDetailsPanel might implicitly use this package for its internal event handling.
5. **java.io** (*optional, for exception handling*)
- While not directly used, some database operations might throw java.io.IOException indirectly (e.g., through SQLException chaining), though this isn't explicitly handled in the code.
  - Example: Not directly used, but could be relevant for future extensions like file I/O.

## Complete List of Packages Used

Combining both explicitly and implicitly used packages, here is the full list:

- **javax.swing**
- **java.awt**
- **java.awt.event**
- **java.sql**
- **java.text**
- **java.util**
- **java.lang** (implicit)
- **javax.swing.border** (implicit)
- **javax.swing.event** (implicit, for some Swing components)
- **java.io** (implicit, for potential exception handling)

## Explanation of Package Usage

- **GUI and Event Handling:** javax.swing, java.awt, java.awt.event, and javax.swing.border are used to build and manage the graphical interface, handle user interactions, and style components.

- **Database Operations:** java.sql is used for all database interactions, connecting to MySQL, executing queries, and managing transactions.
- **Date and Time Handling:** java.text is used for parsing and formatting dates, critical for handling journey dates in the application.
- **Utility Classes:** java.util provides essential data structures (List, Map), random number generation (Random), and unique identifiers (UUID).
- **Core Java Features:** java.lang provides basic types and utilities that are fundamental to the program.

## 13. Checklist

### Checklist Evaluation

#### 12.1. Class (Private, Public, Protected)

- **YES**
  - **Explanation:** The code uses different access modifiers for classes and their members:
    - **Public:** The TrainManagementSystem class is public, as are classes like User, Profile, Train, and Booking.
    - **Private:** Fields in classes like User (private int userId), Train (private String trainNumber), and Booking.Passenger (private String name) are private.
    - **Protected:** Not explicitly used in the code, but the presence of public and private satisfies the requirement for access modifiers.

#### 12.2. Sub Class

- **YES**
  - **Explanation:** The TrainManagementSystem class is a subclass of JFrame (via inheritance: extends JFrame), making it a subclass:
 

```
java
Copy
public class TrainManagementSystem extends
```



## JFrame

### 12.3. Inner Class

- YES

**Explanation:** The Booking class contains a nested (inner) class called Passenger:

java

Copy

```
public static class Passenger {
```

```
    private String name;
```

```
    private int age;
```

```
    // ...
```

```
    }
```

- This inner class is used to represent passenger details within the context of a booking.

### 12.4. Method with Object Reference

- YES

**Explanation:** Several methods operate on object references. For example, in the User class, the register() method operates on the current User object:

java

Copy

```
public void register() throws SQLException {
```

```
    try (Connection conn =  
        DBConnection.getConnection()) {
```

```
String sql = "INSERT INTO Users (name, mobile, email, password, security_question, security_answer) VALUES (?, ?, ?, ?, ?, ?)";
```

```
// Uses 'this.name', 'this.email', etc.
```

```
}
```

```
○ }
```

- Here, this (implicitly) refers to the User object being registered.

## 12.5. Methods without Object Reference

- YES

**Explanation:** The code includes static methods that don't require an object reference. For example, in the DBConnection class:

```
java
```

```
Copy
```

```
public static Connection getConnection() throws SQLException {
```

```
    return DriverManager.getConnection(URL, USER, PASSWORD);
```

```
○ }
```

- This method can be called without creating an instance of DBConnection (e.g., DBConnection.getConnection()).

## 12.6. Constructors

- YES

**Explanation:** Multiple classes have constructors. For example, the User class has a constructor:

java

Copy

```
public User(String name, String mobile, String email,
String password,
String securityQuestion, String
securityAnswer) {

    this.name = name;

    this.mobile = mobile;

    // ...

    }
```

## 12.7. Overloaded Constructors

- YES

**Explanation:** The User class has overloaded constructors (multiple constructors with different parameter lists):

java

Copy

```
public User(String name, String mobile, String email,
String password,
String securityQuestion, String
securityAnswer) {

    // Constructor for new user

}
```

```

private User(int userId, String name, String mobile,
String email, String password,

                String securityQuestion, String
securityAnswer, String sessionId) {

    // Constructor for loading existing user

    ○ }

```

## 12.8. Collections: 1, 2, 3

- YES

- **Explanation:** The code uses several collection types:

**1. List:** List<Train> in Train.searchTrainsWithFilters() and List<Passenger> in Booking.bookTickets():

java

Copy

```
List<Train> trains = new ArrayList<>();
```

- List<Booking.Passenger> passengers = new ArrayList<>();

- **2. Map:** Map<String, Double> in Booking for CLASS\_FARES and CONCESSION\_RATES:

java

Copy

```
private static final Map<String, Double>
CLASS_FARES = new HashMap<>();
```

- **3. ArrayList:** Used as the concrete implementation of List (e.g., new ArrayList<>()).

- The use of List, Map, and ArrayList satisfies this requirement.

## 12.9. Overloaded Methods

- YES

- **Explanation:** While the code doesn't heavily use method overloading, there are examples where methods could be considered overloaded in a broader sense. For instance, the `DBConnection.logSession()` method could be seen as a candidate for overloading if additional versions were added with different parameters. However, the constructors (e.g., in `User`) already satisfy the concept of overloading, as discussed earlier.

## 12.10. Abstraction through Abstract Class

- NO

- **Explanation:** The code does not use abstract classes for abstraction. While abstraction is achieved through encapsulation and method design (e.g., hiding database details in `DBConnection`), there are no abstract classes defined.

## 12.11. Abstraction through Interface

- NO

- **Explanation:** The code does not use interfaces for abstraction. It relies on concrete classes and encapsulation to achieve abstraction, but no interface keyword is used.

## 12.12. Inheritance: Single

- YES

- **Explanation:** The `TrainManagementSystem` class demonstrates single inheritance by extending `JFrame`:

```
java
```

```
Copy
```

```
public class TrainManagementSystem extends  
JFrame
```

### 12.13. Inheritance Multilevel

- NO

- **Explanation:** There is no multilevel inheritance in the code. Multilevel inheritance would require a class hierarchy like A -> B -> C, but the code only has single-level inheritance (TrainManagementSystem extends JFrame).

### 12.14. Inheritance Hierarchical

- NO

- **Explanation:** Hierarchical inheritance involves one superclass with multiple subclasses (e.g., A -> B and A -> C). The code does not implement this; there's only one subclass (TrainManagementSystem extending JFrame).

### 12.15. Exceptions

- YES

**Explanation:** The code handles exceptions, particularly SQLException and ParseException:

```
java
Copy
try {

                                List<Train>      trains      =
Train.searchTrainsWithFilters(from,      to,      date,
classType, trainType, timeRange);

} catch (SQLException ex) {

                                JOptionPane.showMessageDialog(this, "Error
searching trains: " + ex.getMessage(), "Error",
JOptionPane.ERROR_MESSAGE);
```

- }

### 12.16. User Defined Exceptions

- NO

- **Explanation:** The code does not define custom exceptions (e.g., no class CustomException extends Exception). It uses built-in exceptions like SQLException.

### 12.17. Multiple Inheritance

- NO

- **Explanation:** Java does not support multiple inheritance for classes (only through interfaces). The code does not attempt multiple inheritance, as it uses single inheritance (TrainManagementSystem extends JFrame).

### 12.18. Design

#### a. Use Case Diagram: Use Cases, Relation, Actors, Pre Condition and Post Conditions, Consistency with Method in Java Code

- NO

- **Explanation:** The code submission does not include a use case diagram or documentation specifying use cases, actors, relationships, pre/post conditions, or consistency with the Java code. A use case diagram would typically describe actors (e.g., User, Admin) and use cases (e.g., Book Ticket, Cancel Ticket), but this is not provided.

#### b. Class Diagram: Class, Attributes, Cardinality, Correct Notations, Consistency with Method in Java Code

- NO

- **Explanation:** The code does not include a class diagram. A class diagram would show classes (e.g., User, Train), their attributes (e.g., userId, trainNumber), relationships (e.g., 1-to-many between User and Bookings), and methods, but this is not provided.

**c. Sequence Diagram: Lifeline, Message Flow, Blocks (loop, condition...), Consistency with Method in Java Code**

- **NO**

- **Explanation:** The code does not include a sequence diagram. A sequence diagram would illustrate lifelines (e.g., TrainManagementSystem, DBConnection), message flows (e.g., method calls like bookTickets()), and control structures (e.g., loops for multiple passengers), but this is not provided.

**Final Checklist with Results**

Item	Status
Class (Private, Public, Protected)	YES
Sub Class	YES
Inner Class	YES
Method with Object Reference	YES
Methods without Object Reference	YES
Constructors	YES



Overloaded Constructors	YES
-------------------------	-----

Collections: 1, 2, 3	YES
----------------------	-----

Overloaded Methods	YES
--------------------	-----

Abstraction through Abstract Class	NO
------------------------------------	----

Abstraction through Interface	NO
-------------------------------	----

Inheritance: Single	YES
---------------------	-----

Inheritance Multilevel	NO
------------------------	----

Inheritance Hierarchical	NO
--------------------------	----

Exceptions	YES
------------	-----

User Defined Exceptions	NO
-------------------------	----

Multiple Inheritance	NO
----------------------	----

Use Case Diagram NO

Class Diagram NO

Sequence Diagram NO

## Summary of Findings

- **Strengths:**

- The code effectively uses core OOP concepts like classes with access modifiers, inheritance (single), constructors, and collections.
- It includes inner classes, methods with/without object references, and exception handling.
- The application is functional and demonstrates a complete train booking system.

- **Gaps:**

- The code lacks abstraction through abstract classes or interfaces, which could improve extensibility.
- There is no multilevel or hierarchical inheritance, limiting the inheritance structure.
- User-defined exceptions are not implemented, which could enhance error handling.
- Design documentation (use case, class, and sequence diagrams) is missing, which is critical for understanding the system's architecture and behavior.

- **Recommendations:**

- Add interfaces or abstract classes to improve abstraction (e.g., an Authenticable interface for User).
- Include design diagrams to document the system's structure and behavior.

- Implement user-defined exceptions for specific error scenarios (e.g., `InsufficientBalanceException`).
- Explore multilevel or hierarchical inheritance if additional class relationships are needed.

This evaluation highlights the code's strengths in implementing a functional system while identifying areas for improvement in abstraction, design documentation, and advanced OOP features.