T.C.
SİVAS UNIVERSITY OF SCIENCE AND TECHNOLOGY
FACULTY OF ENGINEERING AND NATURAL SCIENCES
COMPUTER ENGINEERING

MODULE PROJECT

AIRLINE RESERVATION SYSTEM

Prepared by

GROUP 7

Dila Kübra Diricanlı 230201039

Harun Can Eliaçık 230201043

Rümeysa Yücel 230201077

ADVISORS

ASSOC. PROF. FARHAN AADIL

DR. LECTURER REZAN BAKIR

TABLE OF CONTENTS

1. INTRODUCTION

This study analyzes the **Flight Search** module of an airline reservation system by examining both its **frontend visual components** and the **backend data structures** that support them. The system combines a user-friendly interface with efficient data access mechanisms, aiming to provide high performance and scalability. The design is evaluated in terms of **object-oriented programming (OOP) principles**, **data structure selection**, and **time complexity**.

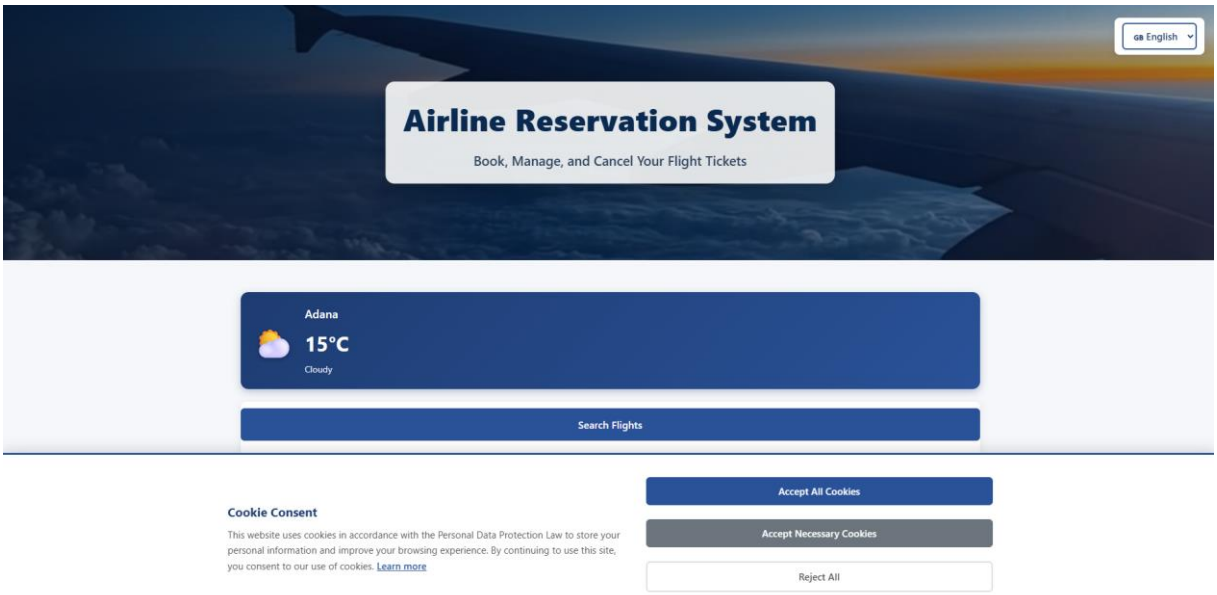

Figure 1.1: *Home page screen including cookie consent and language selection*

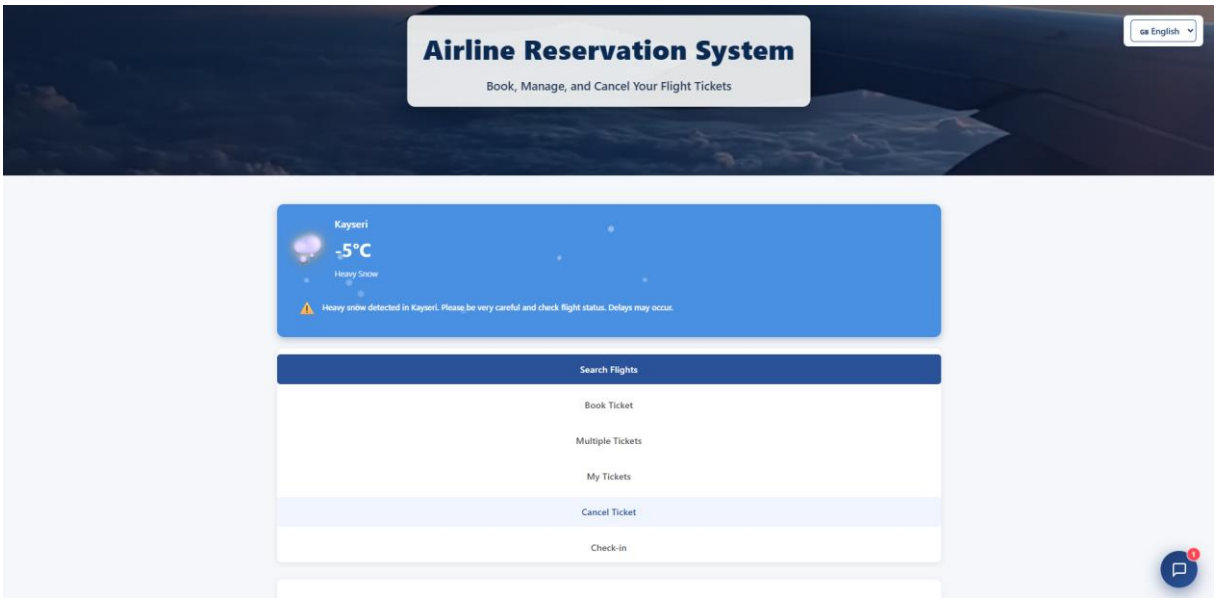

Figure 1.2: *Alternative view of the home page*

2. UML DIAGRAM AND FLOWCHART

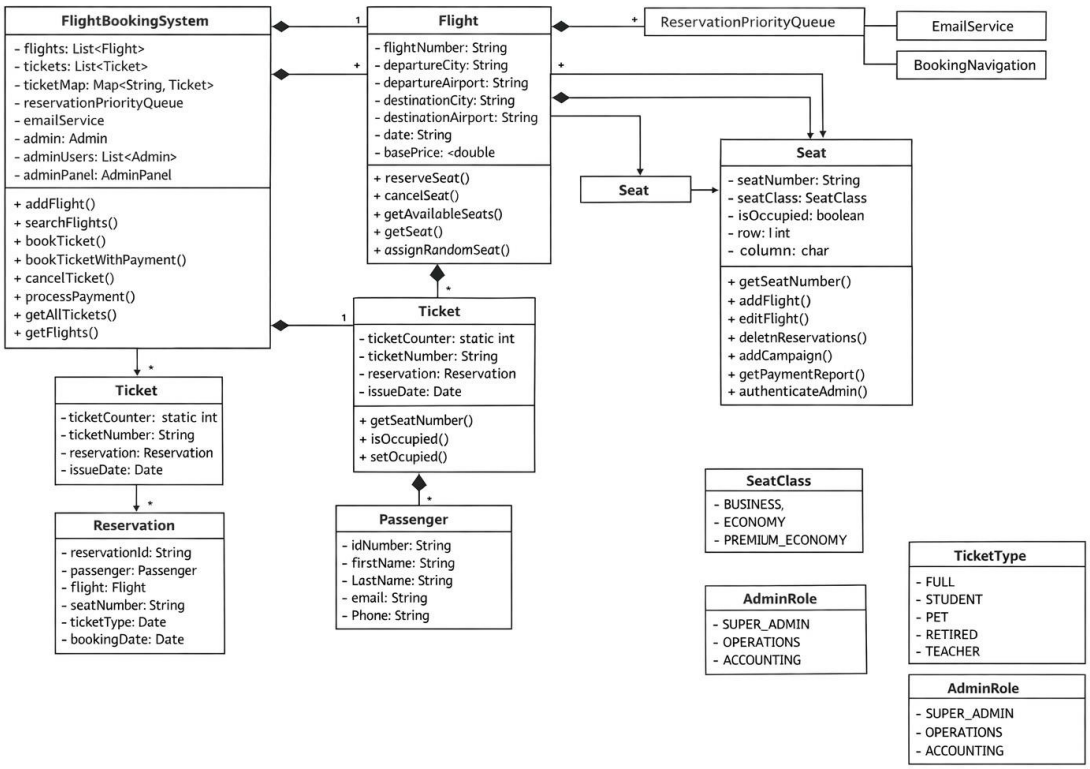

Figure 2.1: *UML class diagram illustrating the backend architecture of the Flight Booking System. The diagram presents the core domain classes such as FlightBookingSystem, Flight, Reservation, Ticket, Seat, Passenger, and Admin, along with their attributes, methods, and relationships. It highlights how flight management, seat reservation, ticket creation, and administrative control are structured using object-oriented design principles and appropriate data structures*

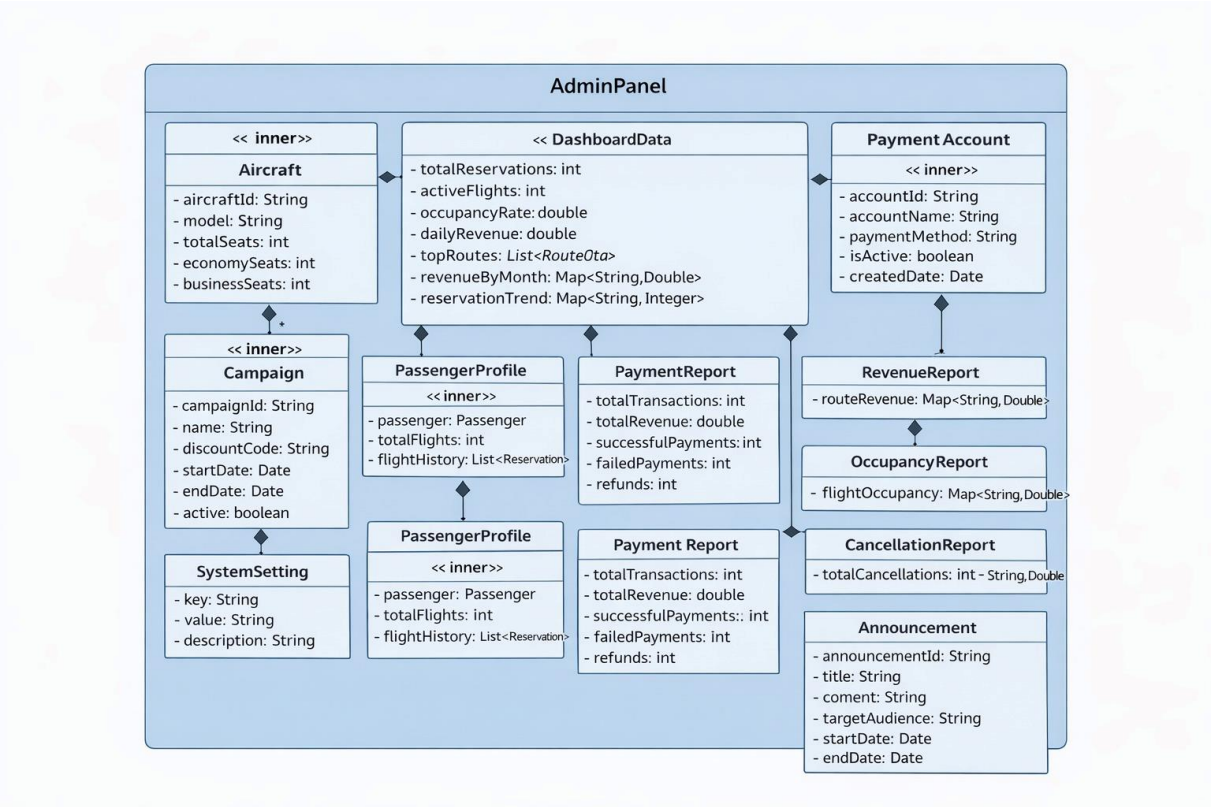

Figure 2.2: *UML class diagram representing the internal structure of the AdminPanel component. The diagram demonstrates how administrative functionalities are modularized through inner classes such as Aircraft, Campaign, SystemSetting, DashboardData, PaymentReport, and Notification. This structure emphasizes separation of concerns by isolating reporting, configuration, payment, and monitoring responsibilities within the administrative layer*

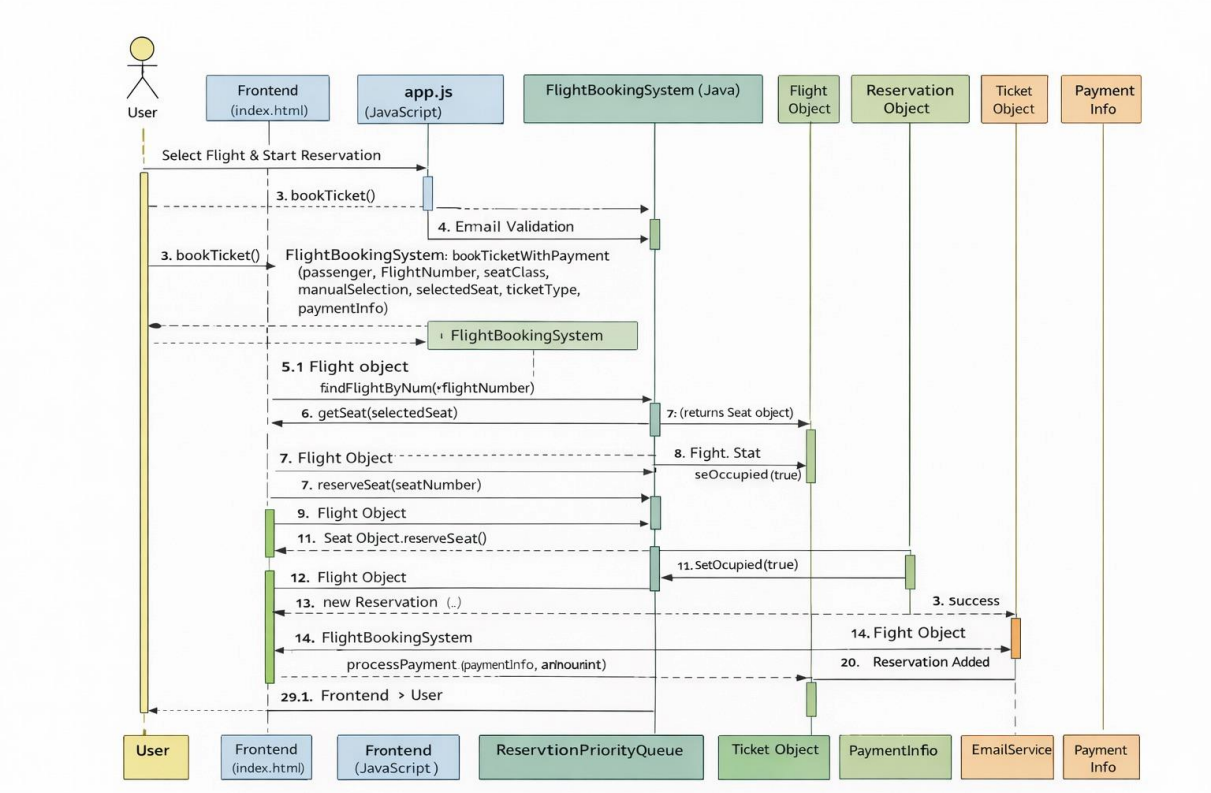

Figure 2.3: *UML sequence diagram illustrating the complete ticket booking workflow in the Flight*

Booking System. The diagram demonstrates the chronological interaction between the user, frontend components, backend business logic, domain objects, and external services during a reservation process

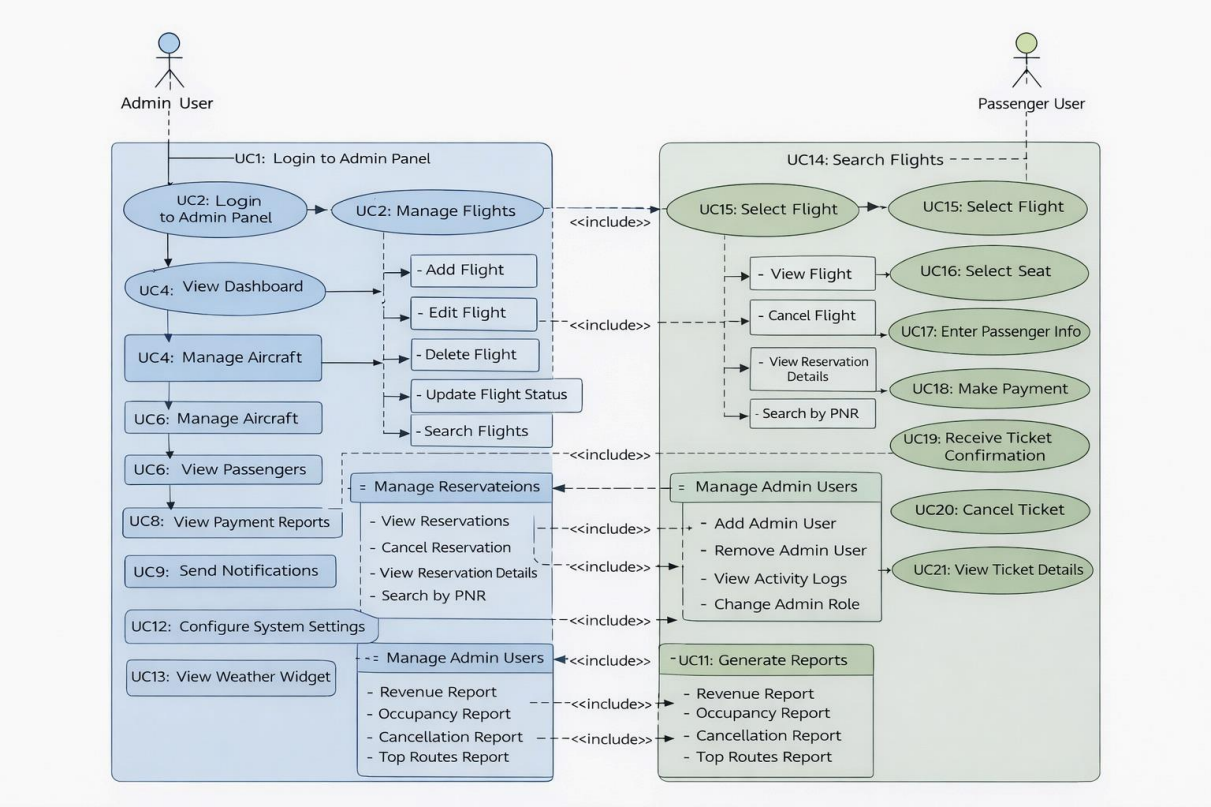

Figure 2.4: *UML use case diagram illustrating the functional scope of the Flight Booking System from both administrative and passenger perspectives. The diagram identifies system actors and their associated use cases, clearly separating administrative operations from passenger booking functionalities*

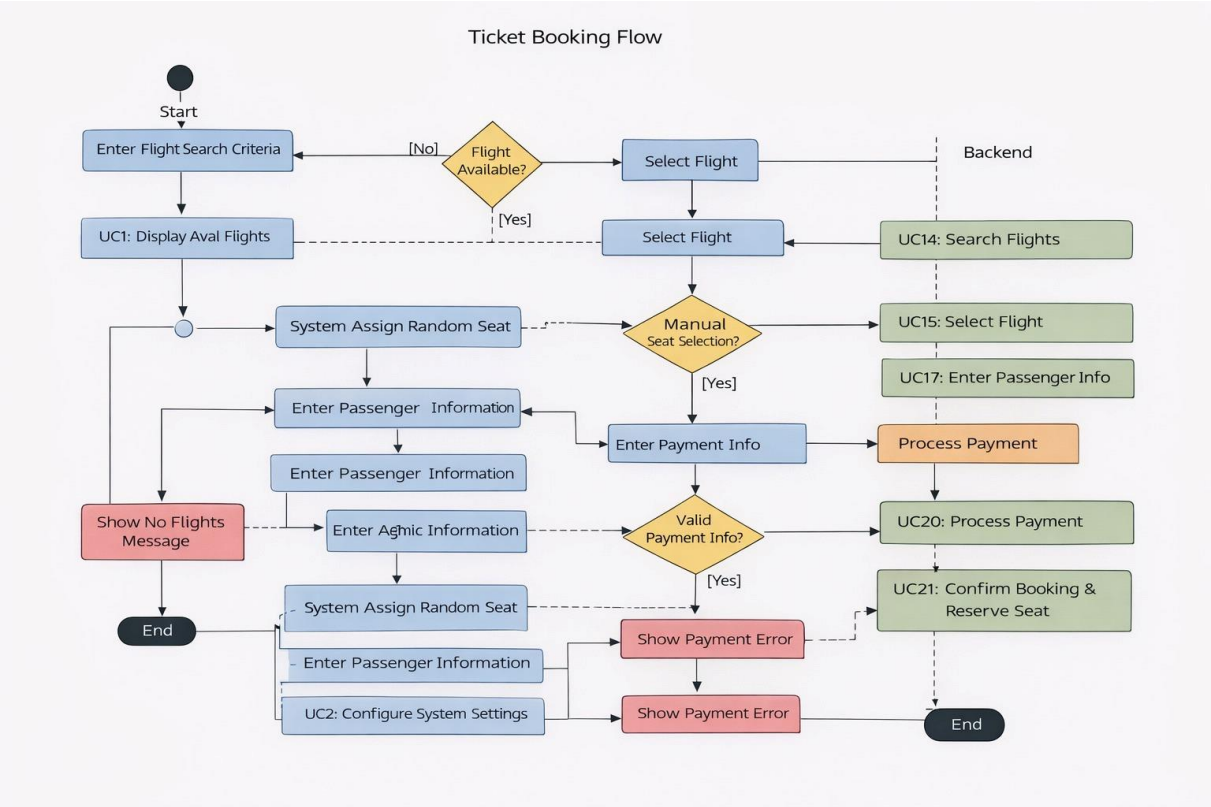

Figure 2.5: *UML use case diagram illustrating the functional scope of the Flight Booking System from both administrative and passenger perspectives. The diagram identifies system actors and their associated use cases, clearly separating administrative operations from passenger booking functionalities*

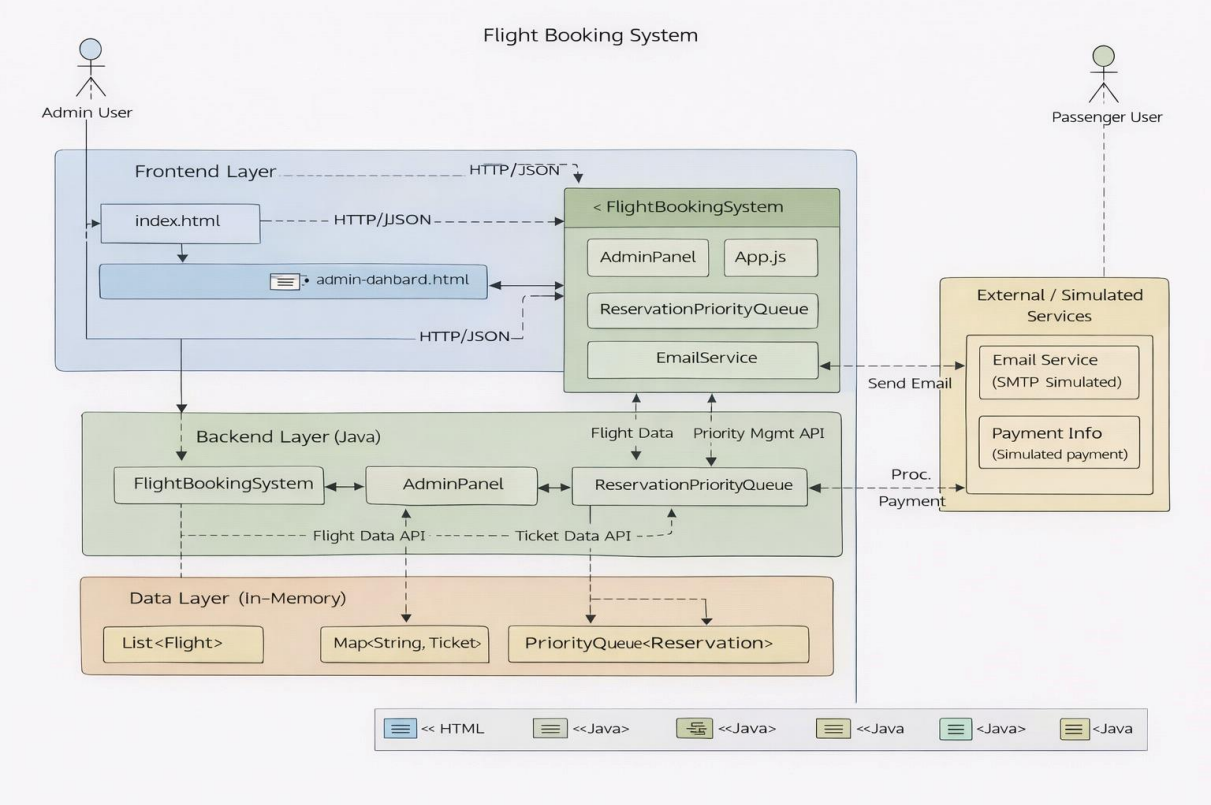

Figure 2.6: *UML component diagram illustrating the architectural structure of the Flight Booking System designed with an in-memory data management approach instead of a traditional database*

3. FLIGHT SEARCH USER INTERFACE (FRONTEND VISUAL FEATURES)

The Flight Search module consists of core visual components that allow users to select departure and destination cities and view available flights.

3.1. Visual Components

- **Departure City Dropdown**: Enables the user to select the departure city.
- **Destination City Dropdown**: Dynamically lists available destination cities based on the selected departure city.
- **Search Flights Button**: Initiates the flight search process.
- **Flight Cards (Flight Results List)**: Displays the available flights in a card-based layout.

This interface design minimizes user effort while maximizing information clarity and accessibility.

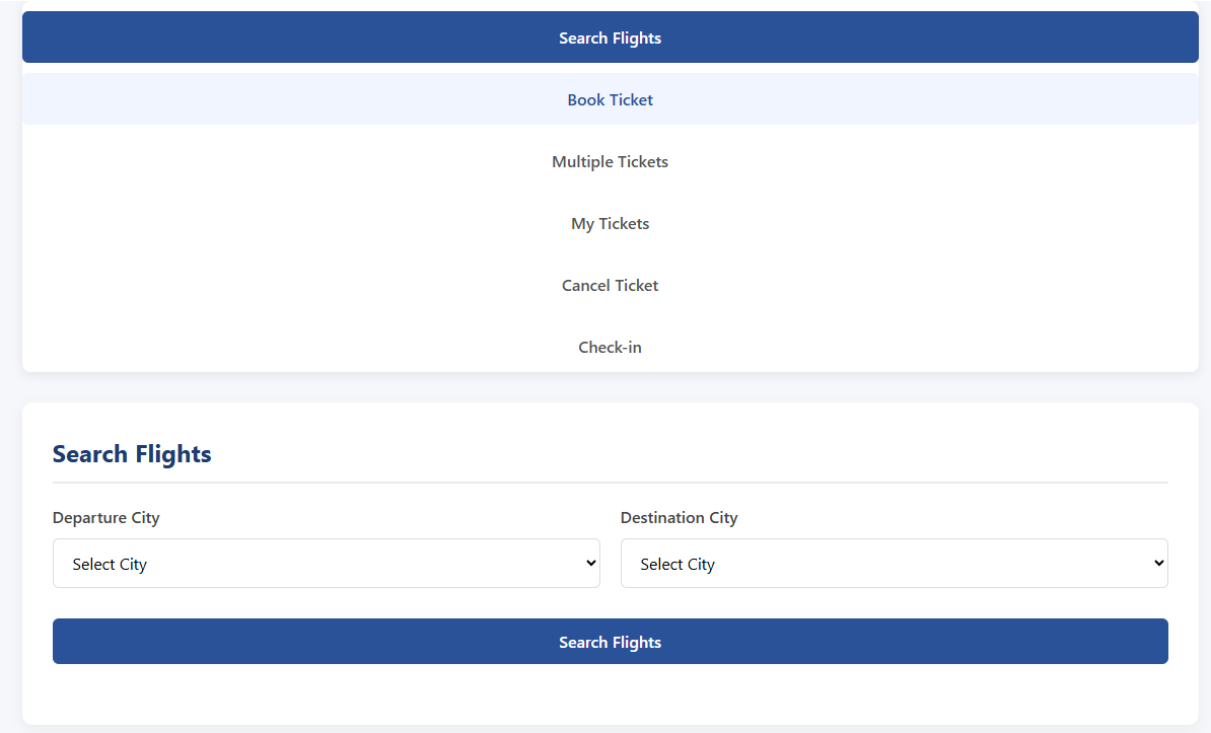

Figure 3.1.1: *Main dashboard for flight booking and search*

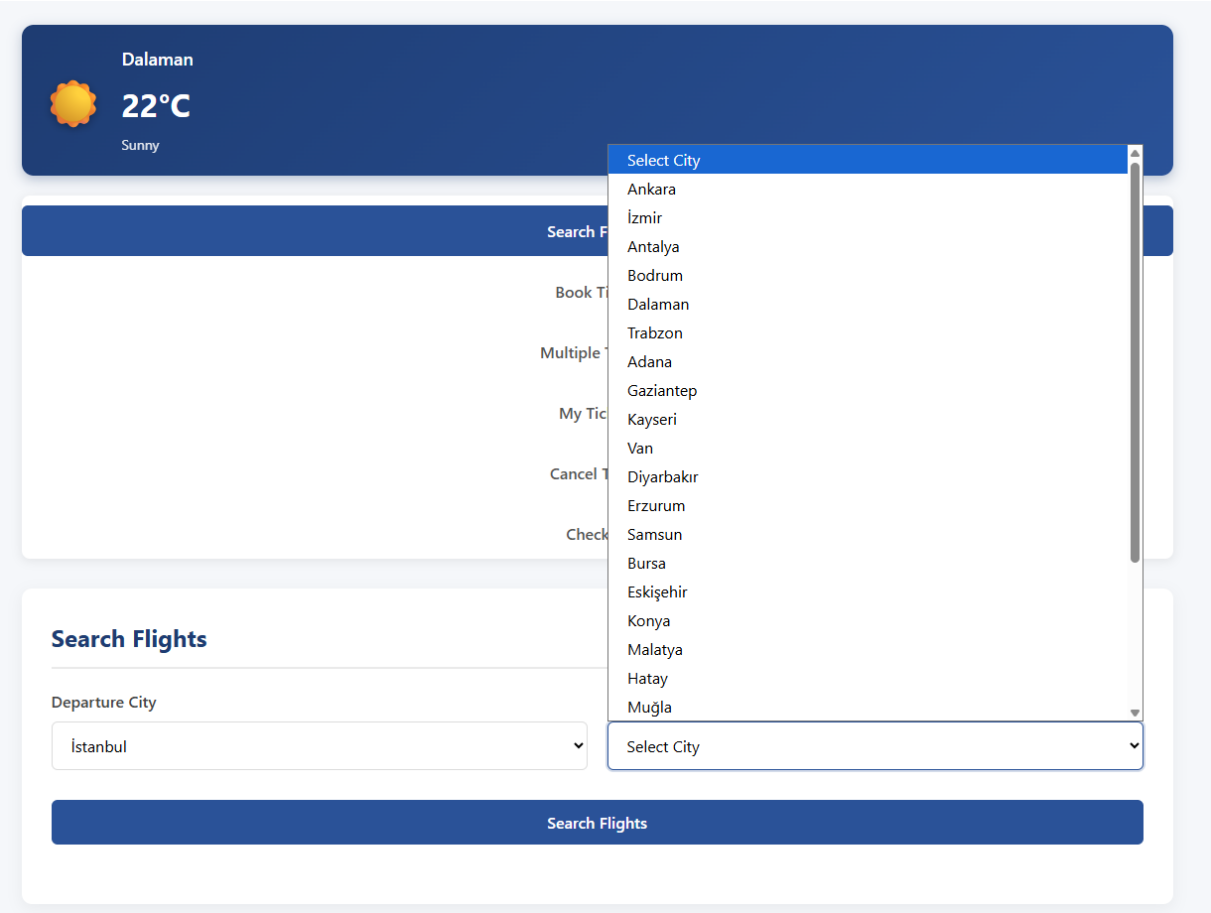

Figure 3.1.2: *Flight search screen featuring city dropdown menus*

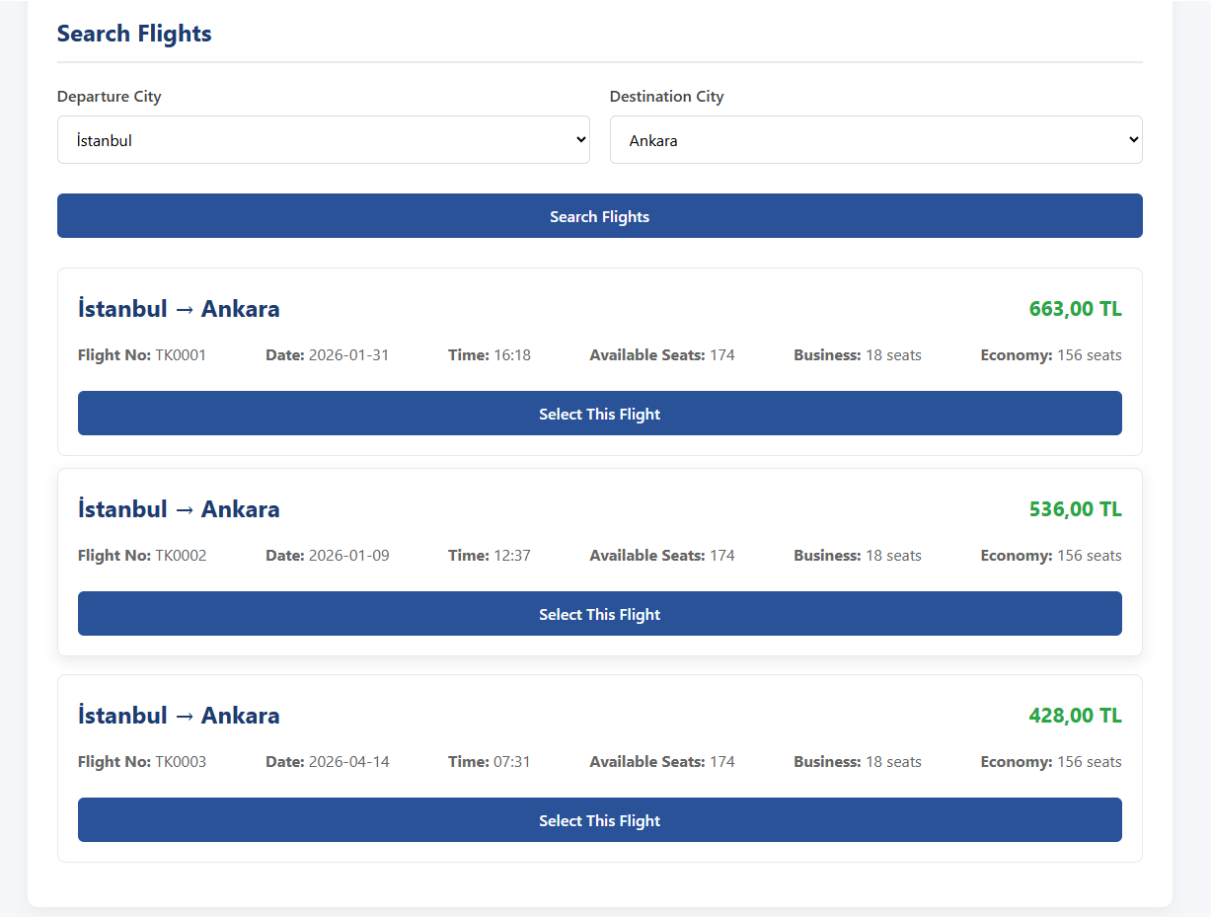

Figure 3.1.3: *Listing of available flights after city selection*

3.2. Backend Data Structure Design of Flight Search

3.2.1 AirportData Class

The management of flight routes is handled by a static utility class named **AirportData**. This class acts as a centralized data source that stores all available routes between cities.

```
private static final Map<String, List<String>> ROUTES = new HashMap<>();
```

In this structure, the **key** represents the departure city, while the **value** is a list of destination cities reachable from that departure point.

3.3. Object-Oriented Programming Analysis

3.3.1. Applied OOP Principles

3.3.1.1. Encapsulation

- The ROUTES map is declared as **private static final**, preventing direct external modification.
- Access to the data is strictly controlled through public static methods.

- The final keyword ensures immutability of the route configuration during runtime, preserving data integrity.

3.3.1.2. Static Utility Class Design

- The AirportData class is designed to be used without instantiating objects.
- All methods are static and operate on shared application-level data.

3.3.1.3. Static Initialization Block

- Flight routes are initialized when the class is loaded into memory.
- This approach ensures that the data is immediately available without additional runtime overhead.

3.5. Rationale for Using HashMap

The **HashMap** data structure was chosen due to its efficiency and suitability for route-based queries.

3.5.1 Advantages

- **O(1) Average Access Time**: Destination cities can be retrieved instantly using the departure city as the key.
- **Key-Value Mapping**: Provides a natural representation of city-to-route relationships.
- **Fast Route Validation**: The routeExists() method checks route availability in constant time on average.

These characteristics are particularly critical for flight search systems that require frequent and fast lookups.

3.6. Airport Code Management

3.6.1. Airport Code Mapping

A dedicated static method is used to convert city names into their corresponding airport codes.

```
   public static String getAirportCode(String city) {
      Map<String, String> codes = new HashMap<>();
      codes.put("İstanbul", "IST/SAW");
      codes.put("Ankara", "ESB");
      codes.put("İzmir", "ADB");
      return codes.getOrDefault(city, city.substring(0, Math.min(3,
city.length())).toUpperCase());
```

3.6.2 OOP Evaluation of Flight Search

- **Static Method Usage**: Eliminates the need for object creation.
- **HashMap Mapping**: Ensures fast and flexible city-to-airport-code resolution.
- **Default Value Strategy**: Prevents runtime errors by generating a fallback airport code when none is predefined.

3.7. Time Complexity Analysis

Method	Time Complexity	Description
getDestinations()	O(1)	Direct HashMap lookup
routeExists()	O(1) (average)	HashMap access + list containment check
getAllCities()	O(n)	Iteration over all cities
TreeSet usage	O(n log n)	Ensures uniqueness and alphabetical ordering
getAirportCode()	O(1)	HashMap lookup

- This analysis demonstrates that the system maintains efficient performance even as the dataset grows.

4. TICKET RESERVATION (BOOK TICKET)

4.1. Visual Features

The Ticket Reservation module is implemented as a **three-step booking workflow**, ensuring a structured and user-friendly reservation process.

- **Step 1: Passenger Information**
  Collection of passenger details required for ticket issuance.
- **Step 2: Flight & Seat Selection**
  Selection of the desired flight and assignment of an available seat.
- **Step 3: Payment**
  Completion of the reservation through a payment transaction.

This step-based design reduces user errors and mirrors real-world airline booking systems.

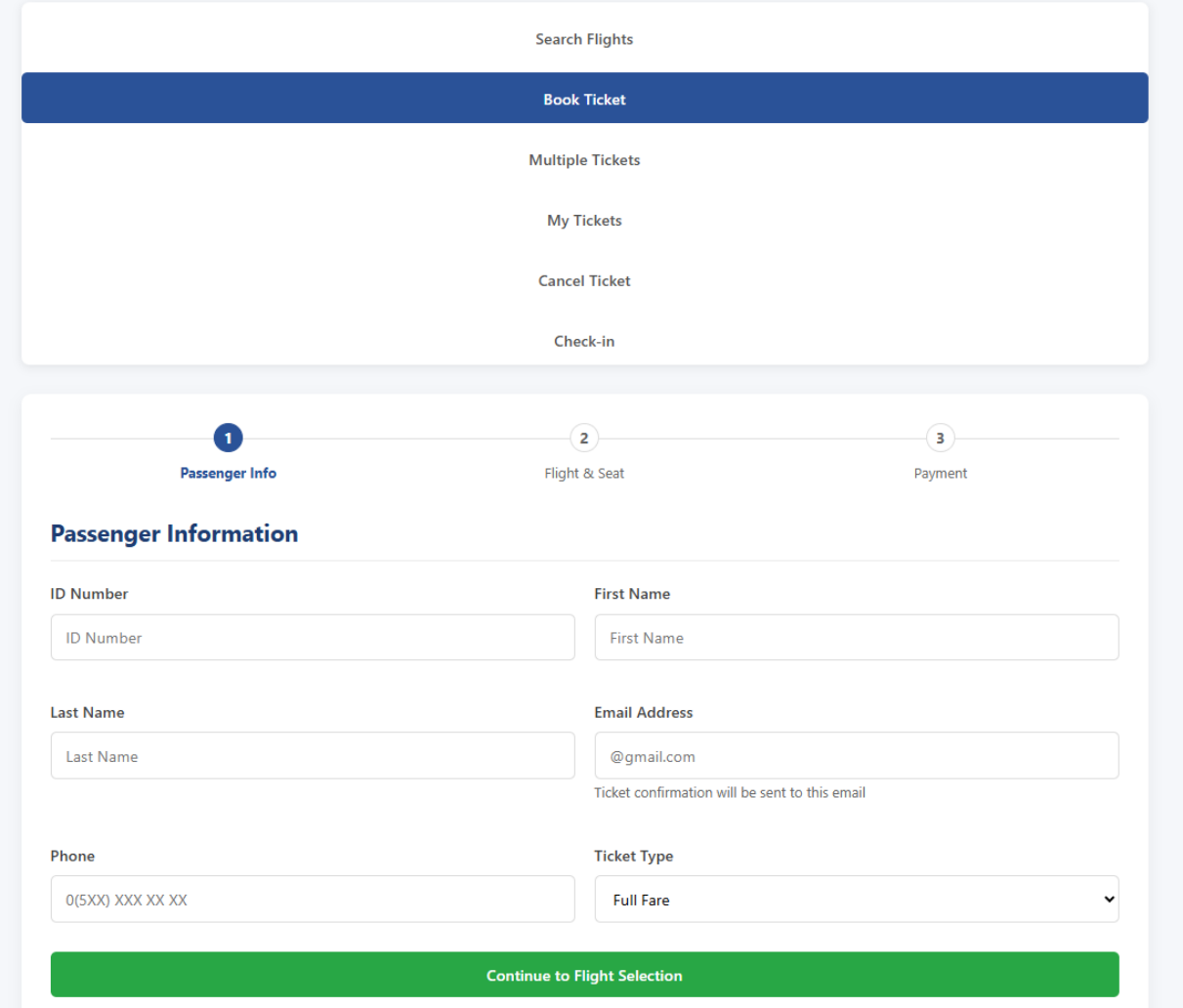

Figure 4.1.1: *Booking flow for flight ticket reservation*

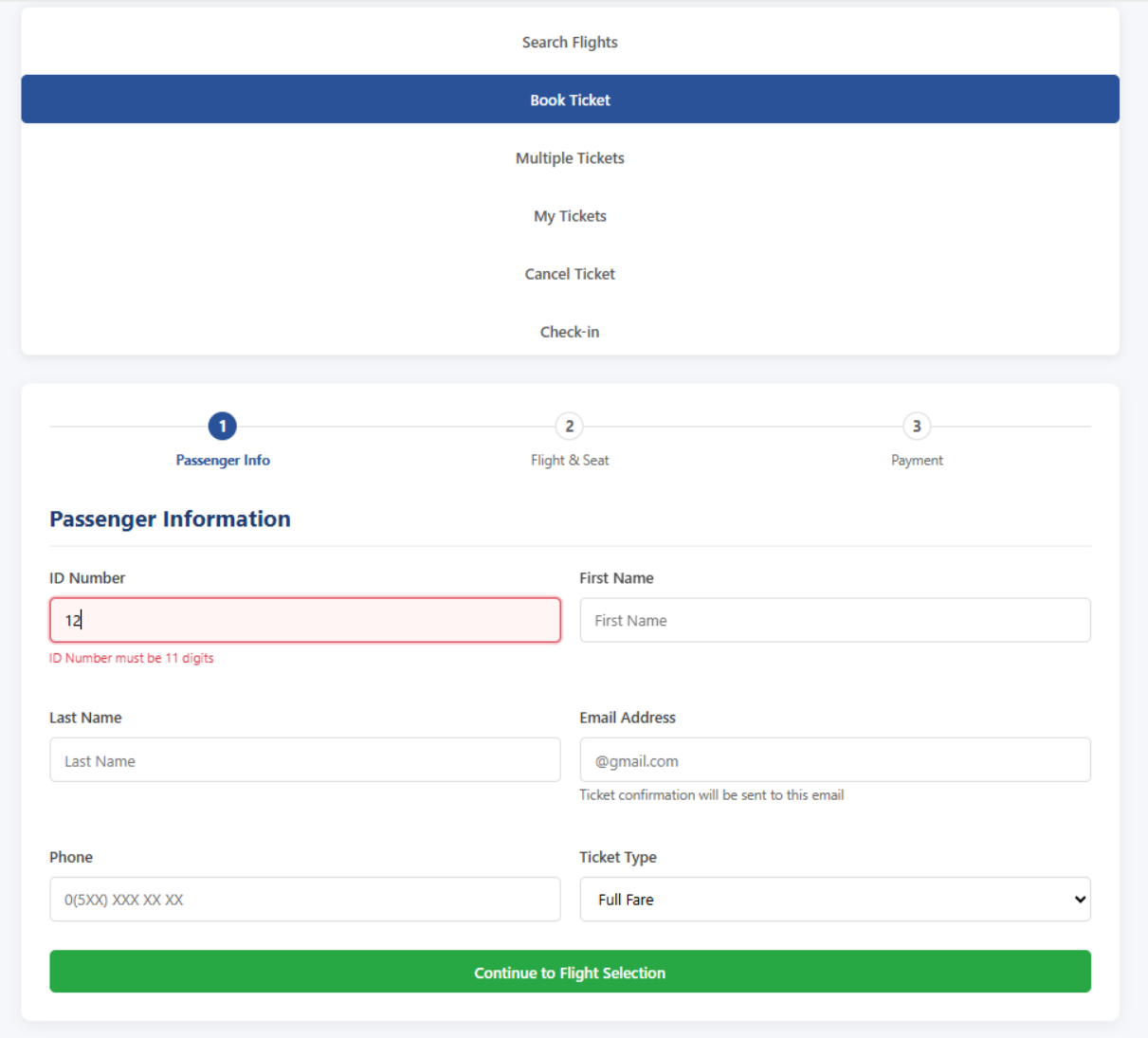

Figure 4.1.2: *Missing ID number notification*

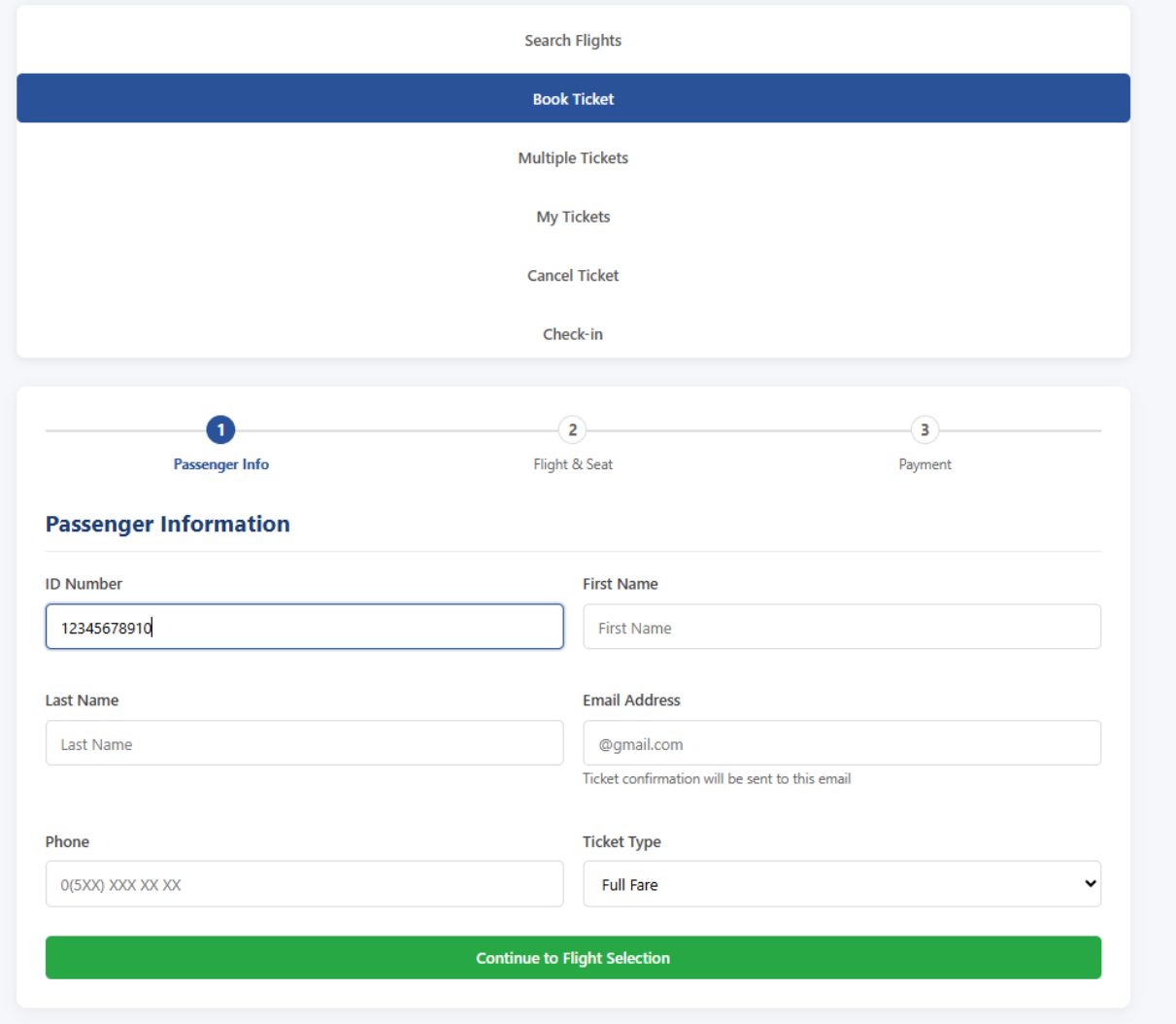

Figure 4.1.3: *Interface showing a valid ID number input*

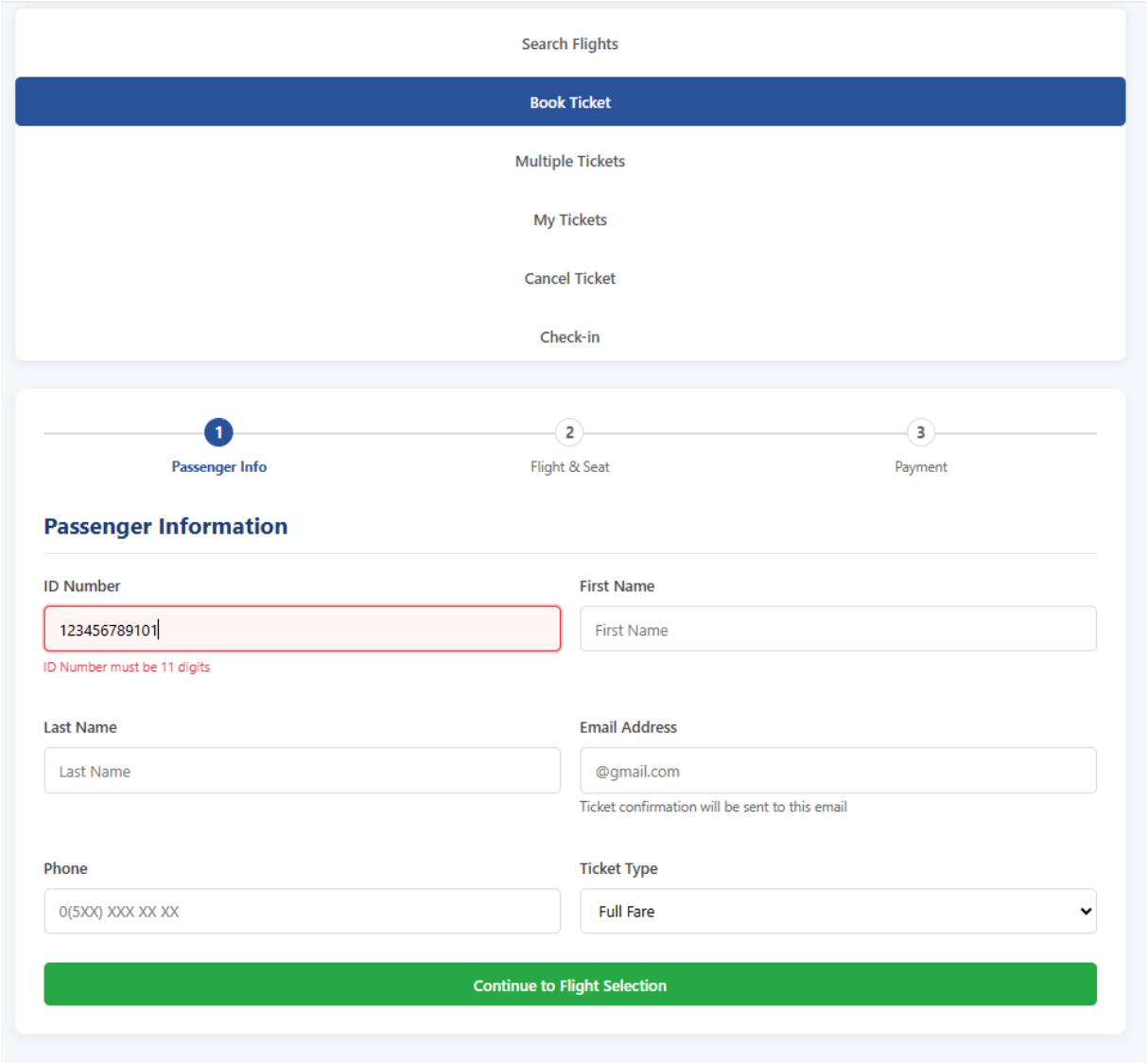

Figure 4.1.4: *Alert showing invalid ID number due to excessive characters*

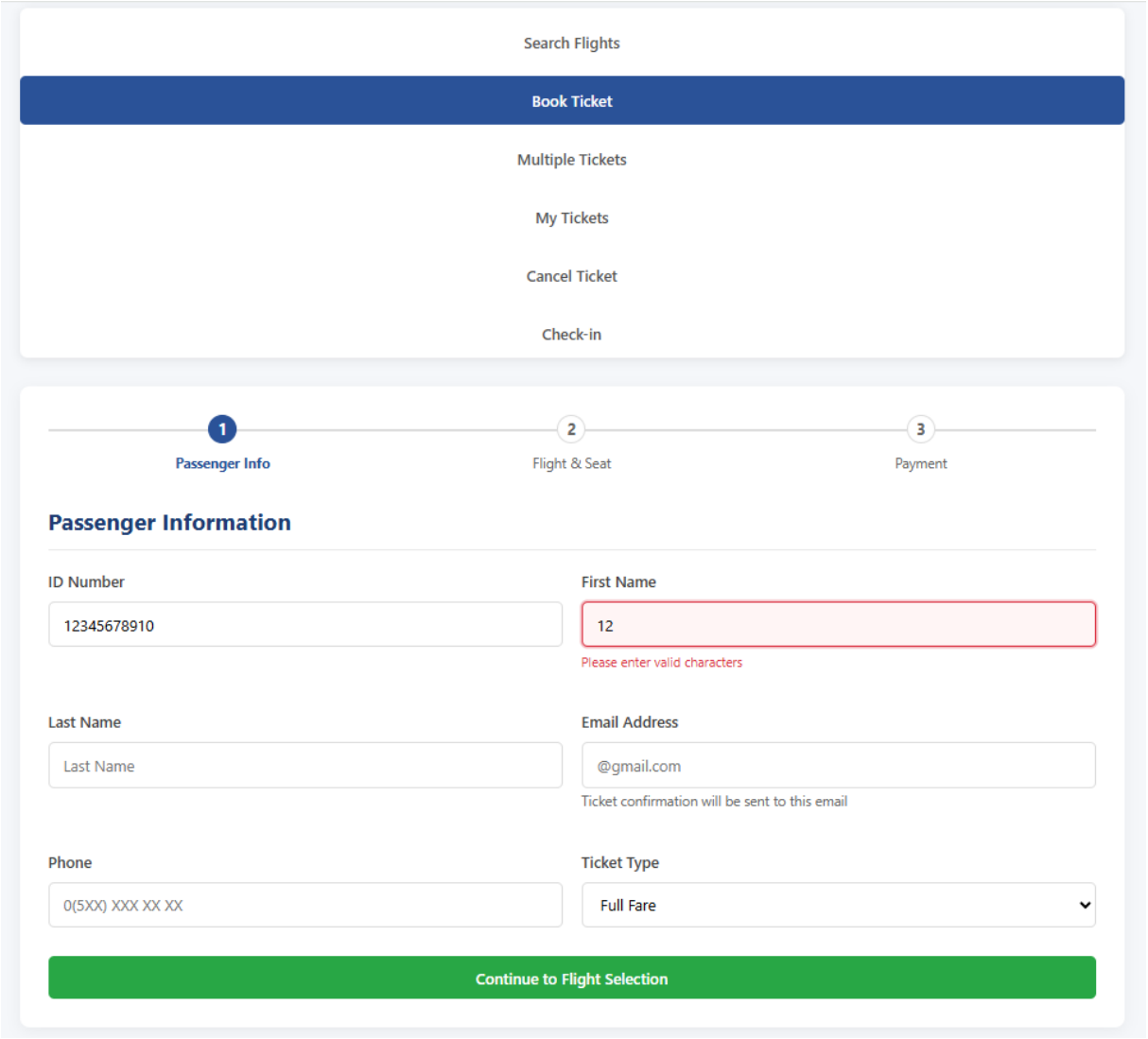

Figure 4.1.5: *Validation error for invalid characters entered in the name field*

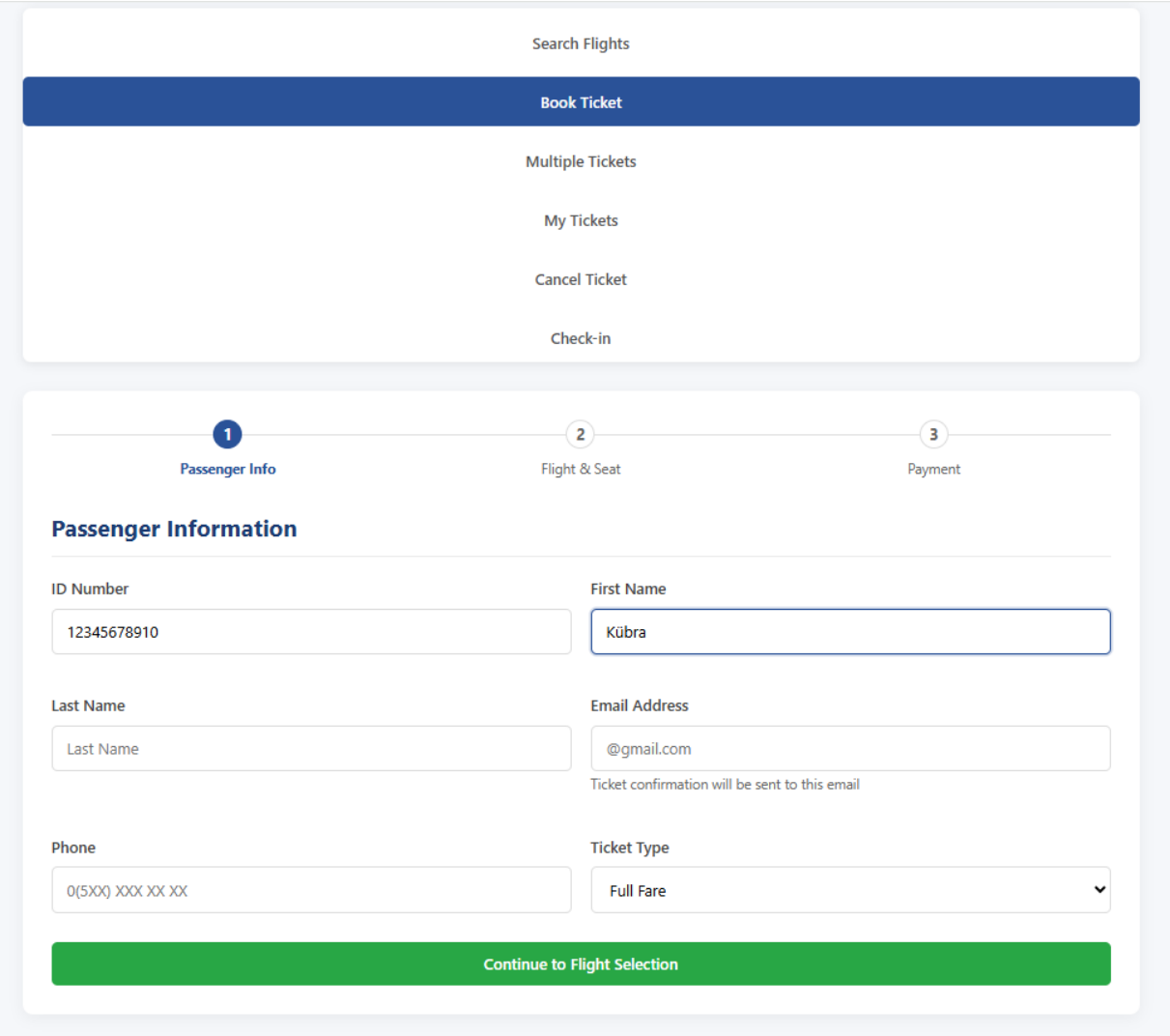

Figure 4.1.6: *Successful validation for valid character entry in the name field*

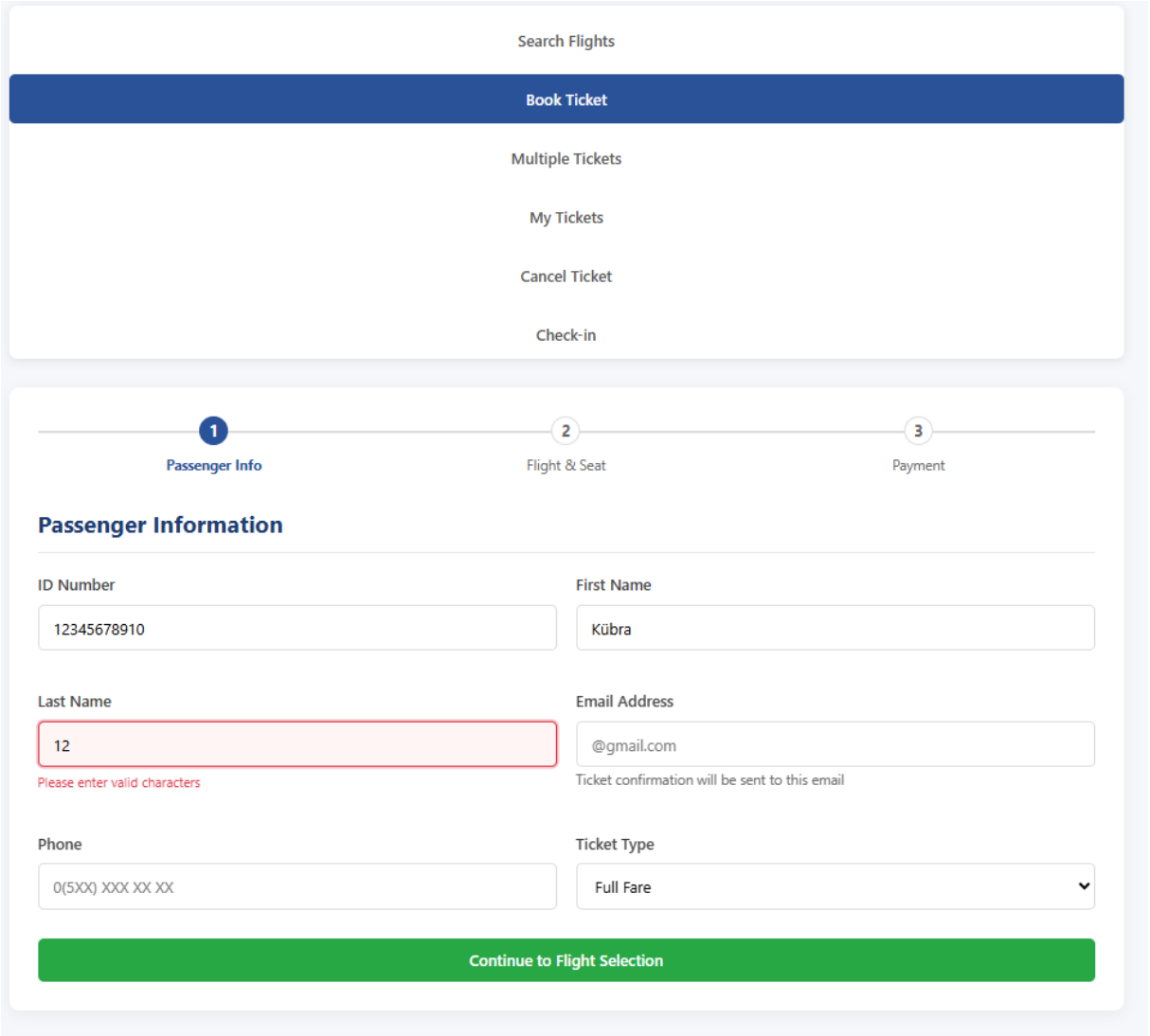

Figure 4.1.7: *Validation error for invalid characters entered in the surname field*

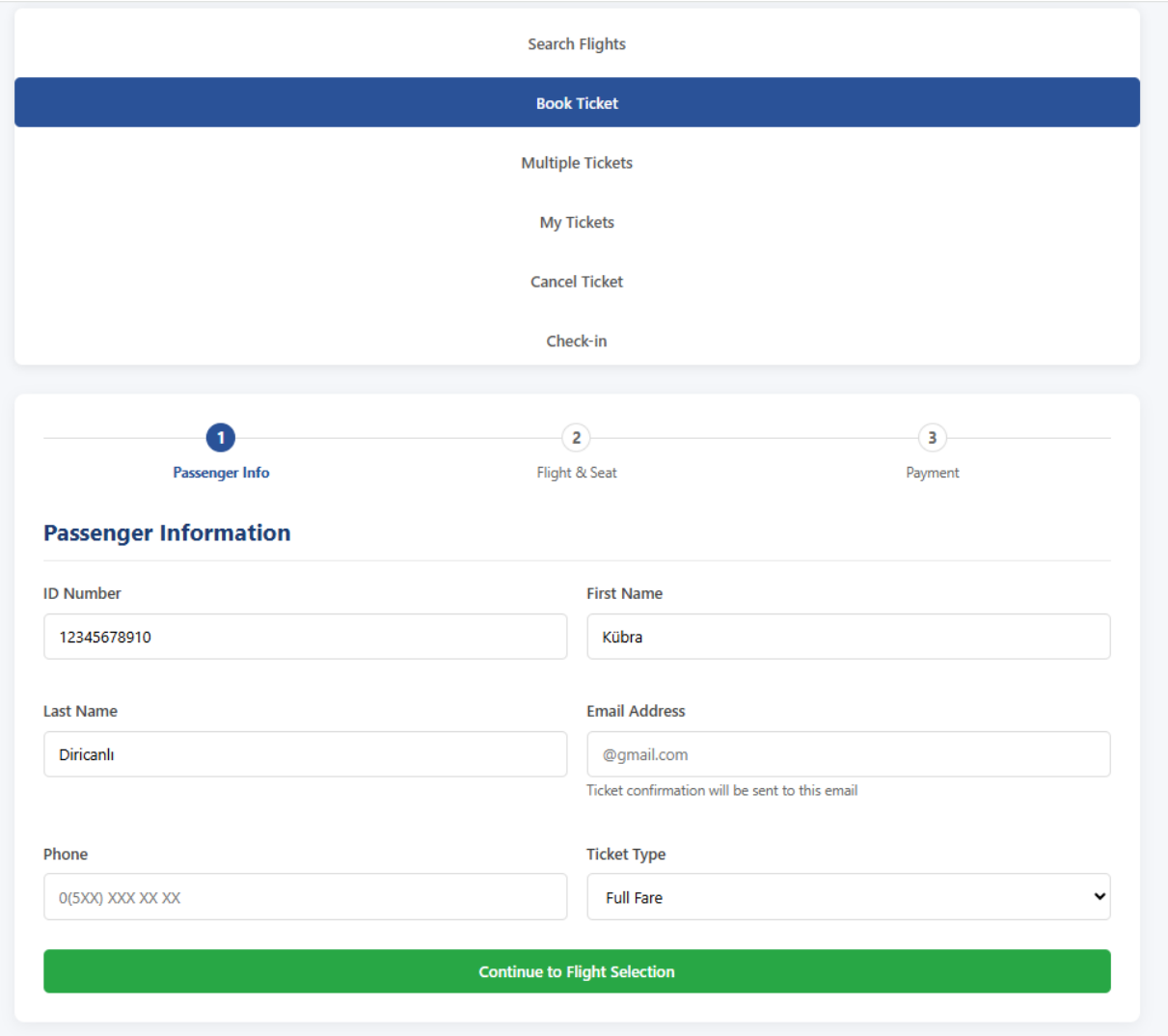

Figure 4.1.8: *Successful validation for valid character entry in the surname field*

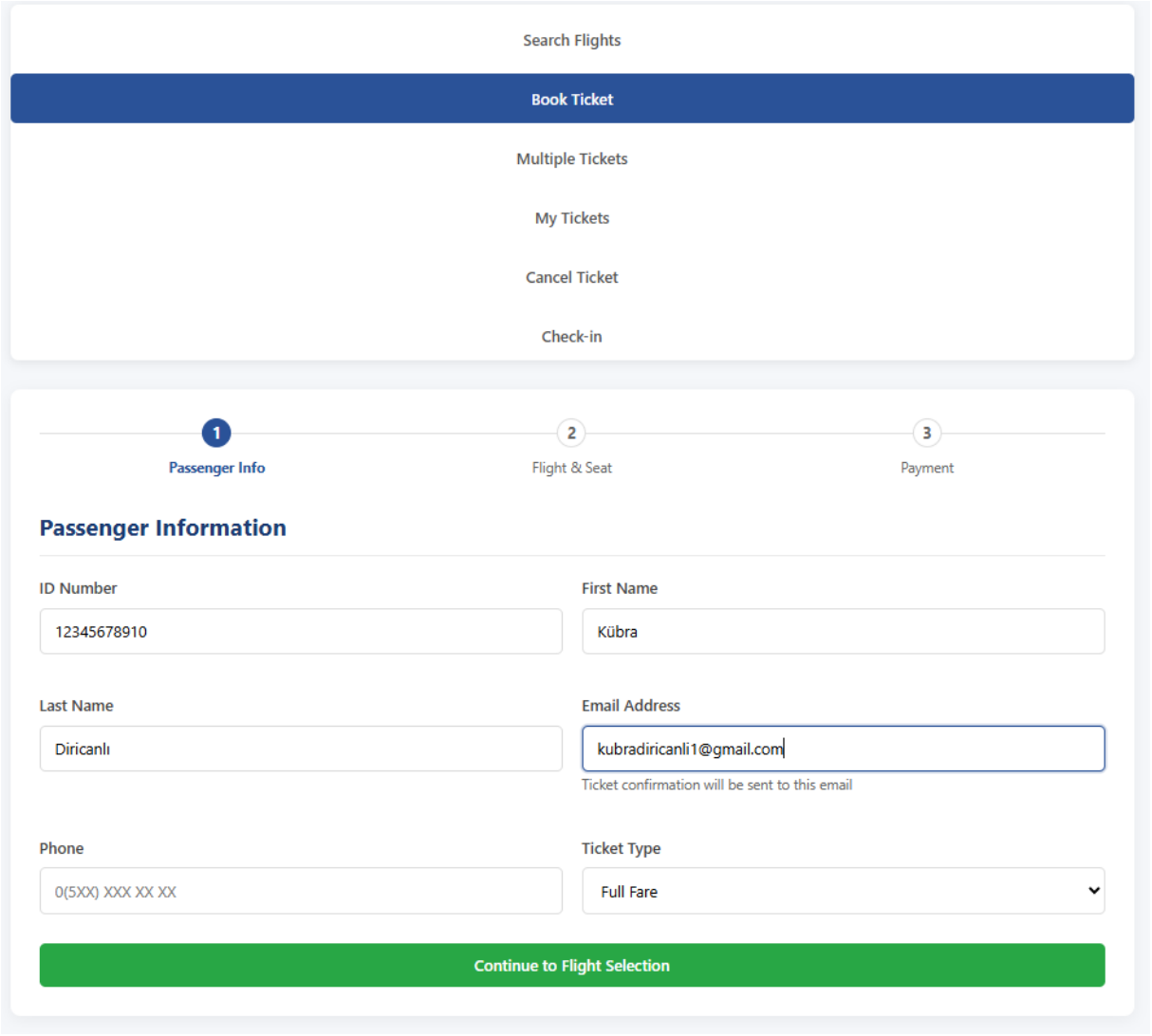

Figure 4.1.9: *Successful validation of a valid email address entry*

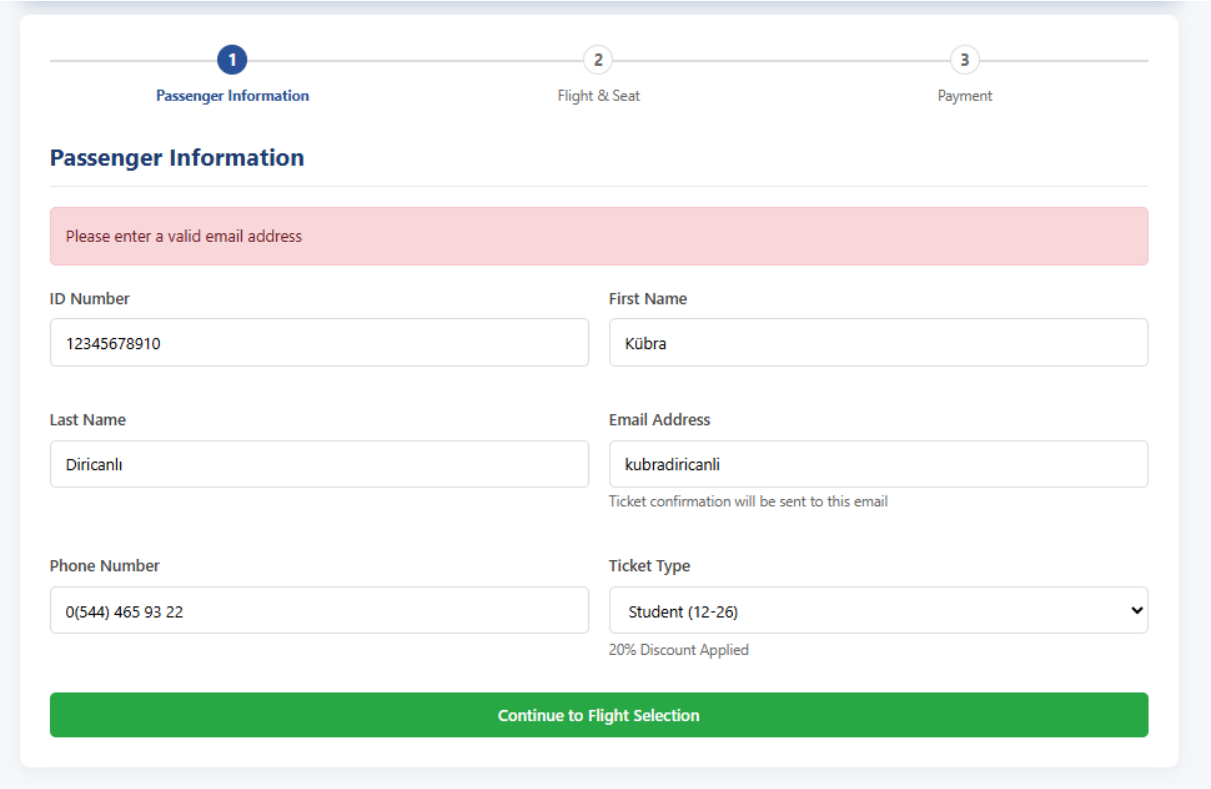

Figure 4.1.10: *Validation error for an invalid email address format*

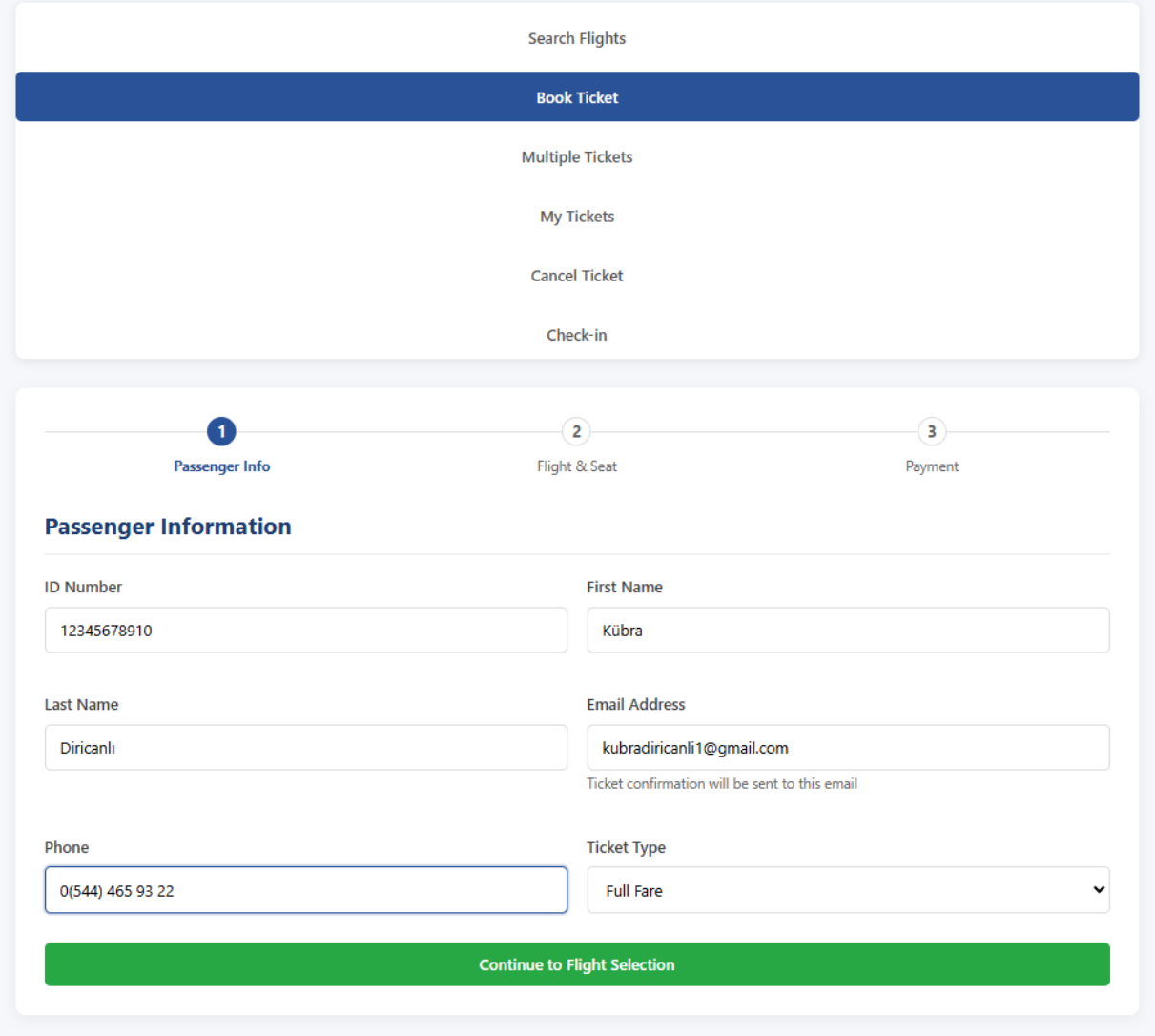

Figure 4.1.11: *Successful validation of a phone number with the correct number of digits*

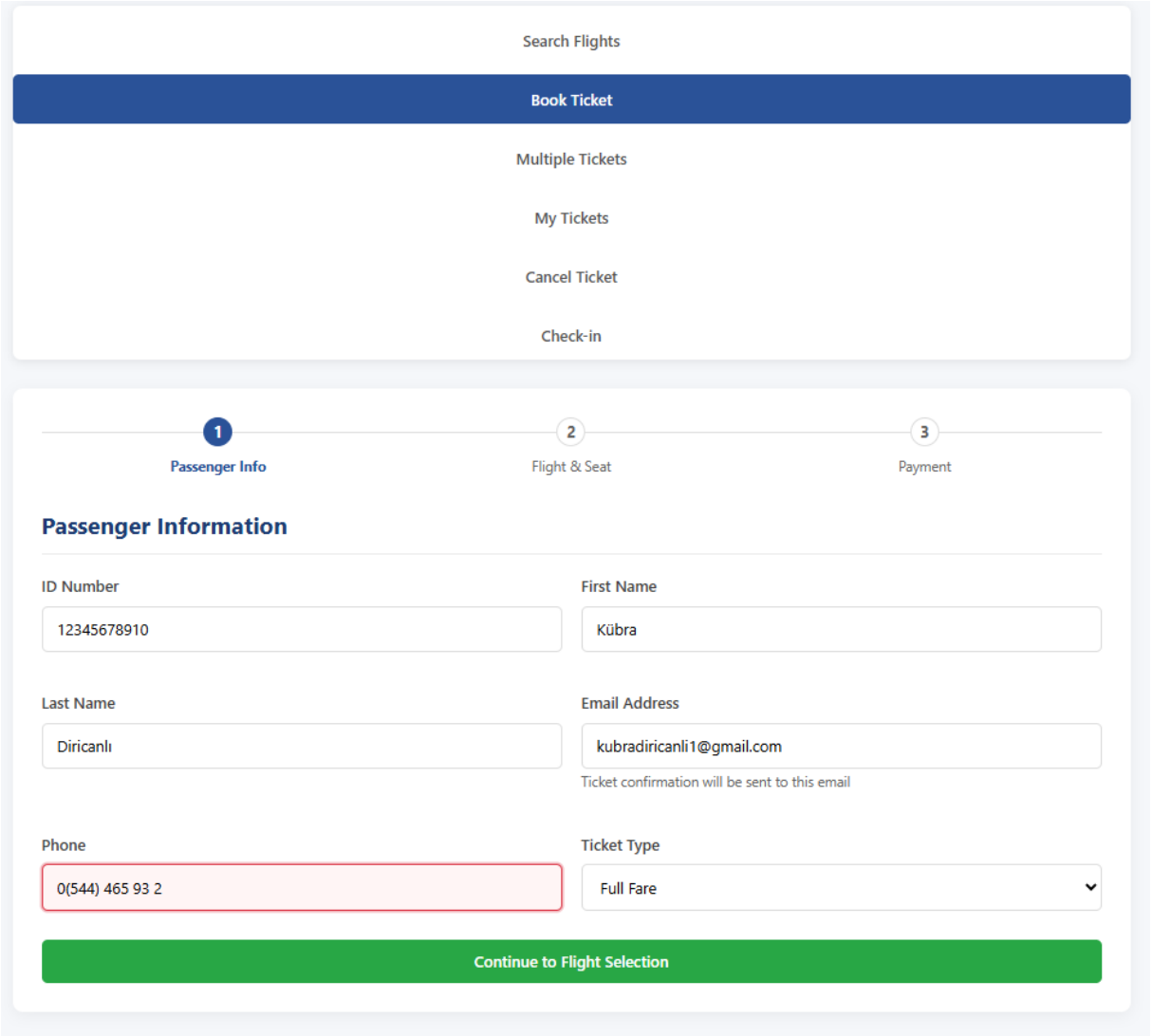

Figure 4.1.12: *Validation error for an incomplete phone number entry*

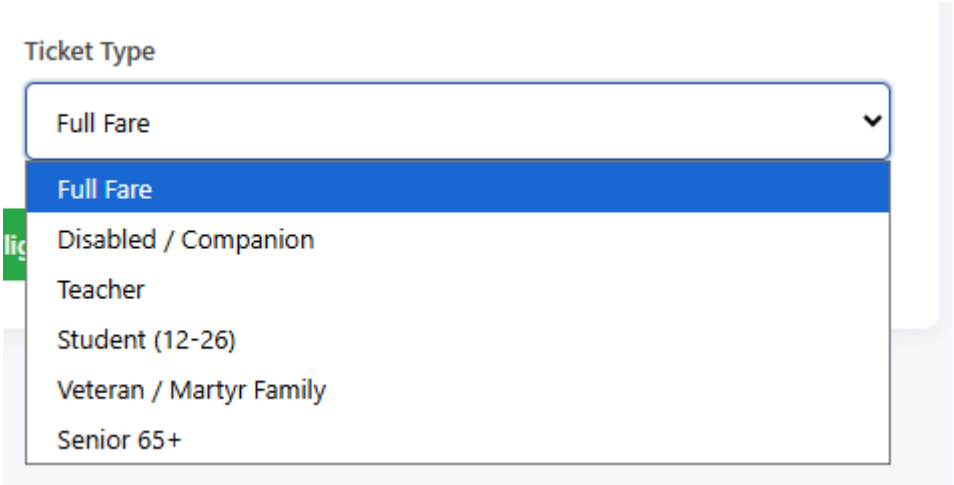

Figure 4.1.13: *Ticket type selection screen*

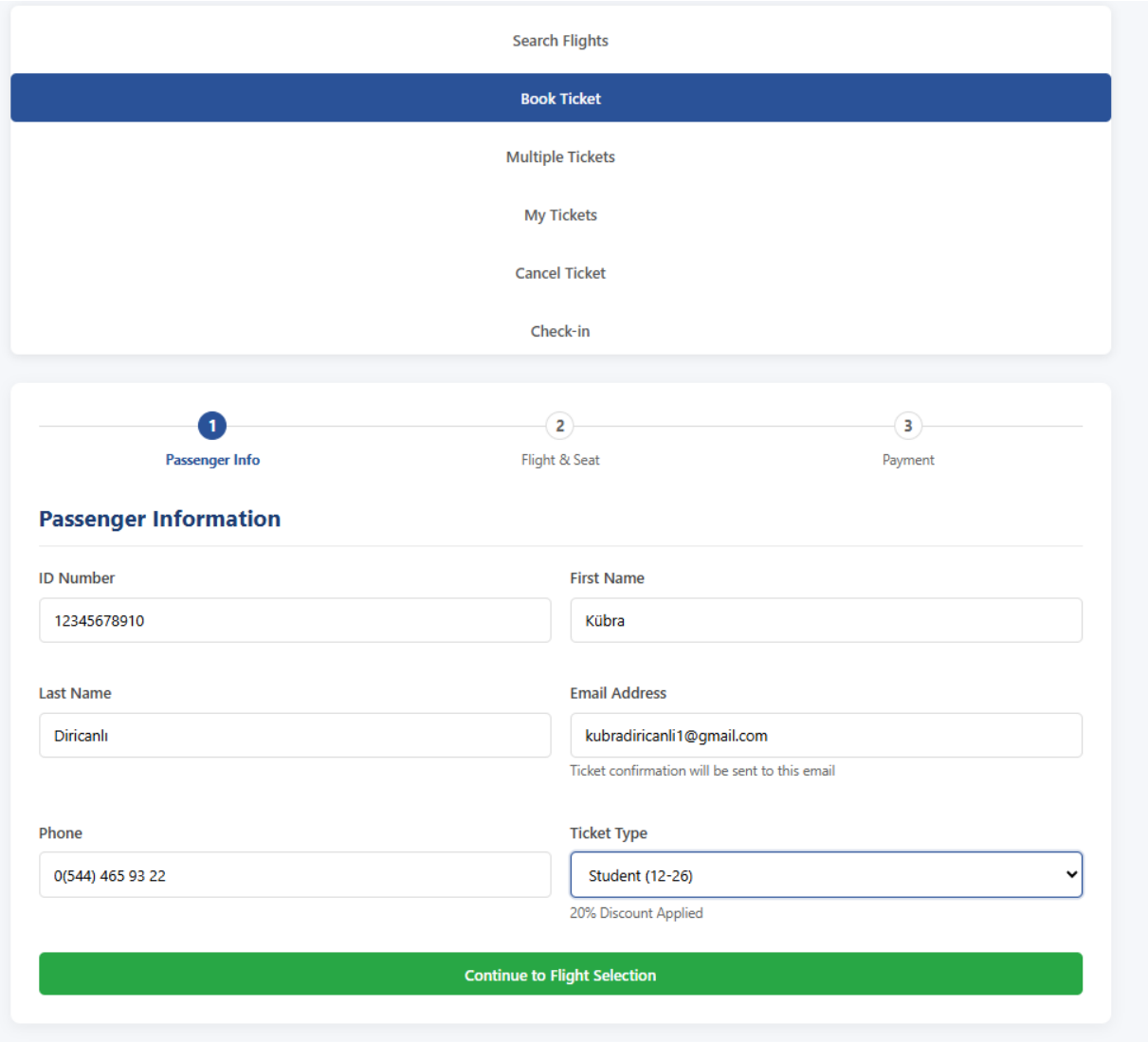

Figure 4.1.14: *Ticket type selection screen displaying the applied discount based on the chosen fare*

4.2. Backend Data Structure of Ticket Reservation

The backend of the reservation process is managed by the FlightBookingSystem class, which stores available flights using a dynamic collection.

```
private List<Flight> flights;
this.flights = new ArrayList<>();
```

The flights list maintains all Flight objects that can be searched, selected, and booked by users.

4.2.3. Object-Oriented Design

The Ticket Reservation module applies multiple object-oriented design principles:

- **Class**: FlightBookingSystem
  Acts as the central controller for flight-related operations.
- **Encapsulation**:
  The flights field is declared as private, ensuring controlled access through class methods.
- **Composition**:
  The system is composed of multiple Flight objects, representing a *has-a* relationship rather than inheritance.
- **Behavioral Methods**:
  - searchFlights() utilizes the **Java Stream API** to filter flights based on user-selected criteria.
  - addFlight() dynamically inserts new flight entries into the system.

This design improves modularity, maintainability, and code readability.

4.3. Rationale for Using ArrayList

The ArrayList data structure was selected due to its compatibility with reservation system requirements:

- **Sequential Processing**:
  Flights are processed in order, which is suitable for listing and filtering operations.
- **Dynamic Size**:
  Flights can be added or removed without fixed-size limitations.
- **Stream API Compatibility**:
  Enables functional-style operations such as filter(), map(), and collect().
- **Index-Based Access**:
  Provides O(1) time complexity for direct access by index, useful during selection operations.

4.4. Time Complexity Analysis

Operation	Time Complexity	Explanation
searchFlights()	O(n)	Iterates through all flights using stream filtering
addFlight()	O(1)	Appends a flight to the end of the ArrayList
findFlightByNumber()	O(n)	Linear search using Stream API filter

This complexity analysis indicates that the module prioritizes flexibility and clarity over constant-time lookup, which is acceptable for moderate-scale flight datasets.

5. BACK BUTTON (NAVIGATION HISTORY)

5.1. Visual Features

The Back Button functionality enables controlled navigation between booking steps, allowing users to return to previous stages of the reservation process.

- **Step 2**: Back button navigates to **Step 1 (Passenger Information)**
- **Step 3**: Back button navigates to **Step 2 (Flight & Seat Selection)**
- **Navigation History**: Maintains a history of visited steps to support backward navigation

This mechanism improves usability by allowing error correction without restarting the booking process.

Figure 5.1: *View of the functional back button available across multiple screens*

5.2. Backend Data Structure of Back Button

Navigation between booking steps is managed by the BookingNavigation class using a stack-based data structure.

```
private Stack<BookingStep> navigationHistory;
this.navigationHistory = new Stack<>();

public void goToStep(BookingStep step) {
    if (currentStep != step) {
        navigationHistory.push(step);  // O(1) Stack push
        currentStep = step;
    }
}

public BookingStep goBack() {
    if (navigationHistory.size() > 1) {
        navigationHistory.pop();       // O(1) Stack pop
        currentStep = navigationHistory.peek(); // O(1) Stack peek
        return currentStep;
```

```
    }
    return currentStep;
}
```

The stack maintains a sequential history of booking steps, ensuring that navigation follows a **Last-In-First-Out (LIFO)** order.

5.2.1. Rationale for Using Stack

The Stack data structure is a natural choice for implementing navigation history.

Advantages of Stack Usage:

1. **LIFO Principle**
   The most recently visited step is the first to be removed, which perfectly matches back button behavior.
2. **Constant Time Operations**
   Stack operations such as push(), pop(), and peek() execute in **O(1)** time.
3. **Navigation History Management**
   Enables precise tracking of user navigation flow.
4. **Simplicity and Clarity**
   Avoids complex algorithms while maintaining predictable behavior.
5. **Direct Mapping to UI Logic**
   The stack-based approach closely aligns with real-world navigation patterns in multi-step forms.

5.2.2. Time Complexity Analysis

Operation	Time Complexity	Description
goToStep()	O(1)	Pushes a step onto the stack
goBack()	O(1)	Pops and peeks the stack
getCurrentStep()	O(1)	Returns the current step
canGoBack()	O(1)	Checks stack size
reset()	O(n)	Clears the stack

This structure ensures efficient and predictable navigation behavior regardless of the number of steps.

5.3. Object-Oriented Design

The navigation system follows established object-oriented design principles:

- **Class**: BookingNavigation
  Responsible for managing navigation state within the booking workflow.
- **Encapsulation**:
  The navigationHistory stack is declared as private, preventing external modification and ensuring controlled access through class methods.
- **Enum Usage**:
  BookingStep defines all valid booking steps:
  - STEP1_PASSENGER_INFO
  - STEP2_FLIGHT_SEAT_SELECTION
  - STEP3_PAYMENT
- **Composition**:
  The BookingNavigation class is used within FlightBookingSystem, forming a *has-a* relationship.
- **Behavioral Methods**:
  - goToStep(BookingStep) – Pushes a new step onto the stack
  - goBack() – Pops the current step and returns the previous one
  - getCurrentStep() – Retrieves the active step
  - canGoBack() – Checks whether backward navigation is possible
  - reset() – Clears navigation history and resets the workflow

6. SEAT SELECTION

6.1. Visual Features

The Seat Selection module allows users to choose their preferred seat based on class and availability, closely resembling real-world airline seat maps.

- **Business Class Seats**: Rows 1–3
- **Economy Class Seats**: Rows 4–10
- **Seat Status Indicators**:
  - *Available*
  - *Selected*
  - *Occupied*

- **Seat Assignment Modes**:
    - Manual seat selection by the user
    - Random seat assignment based on seat class

This visual differentiation enhances user experience and prevents seat allocation conflicts.

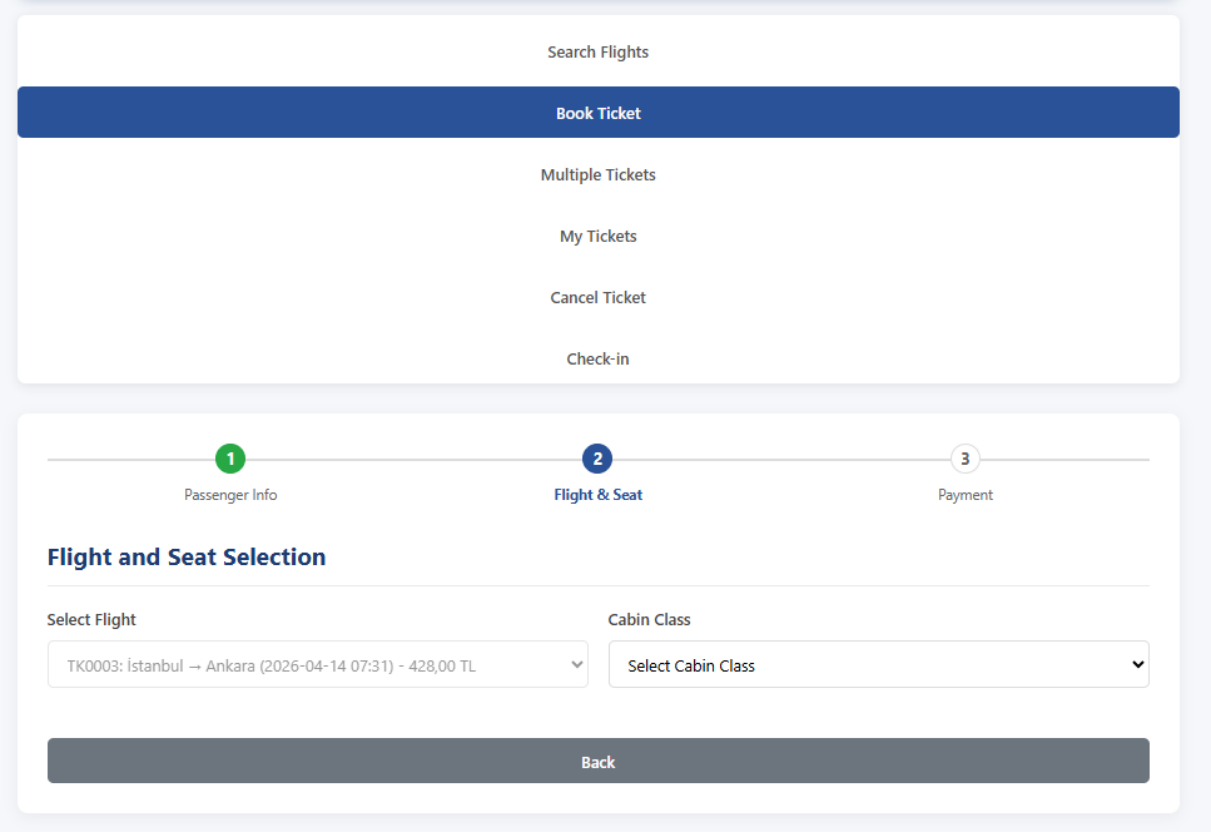

Figure 6.1.1: *Cabin class selection screen*

6.2. Backend Data Structure of Seat Selection

Seat management is handled within the Flight class using a HashMap-based structure.

```
private Map<String, Seat> seats;
this.seats = new HashMap<>();

public Seat getSeat(String seatNumber) {
    return seats.get(seatNumber);  // O(1) HashMap lookup
}

public boolean reserveSeat(String seatNumber) {
    Seat seat = seats.get(seatNumber);  // O(1) access
    if (seat != null && !seat.isOccupied()) {
        seat.setOccupied(true);
```

```
            return true;
        }
    return false;
}
```

Each seat is uniquely identified by its seat number (e.g., "5A"), which acts as the key for direct access.

6.3. Object-Oriented Design

The Seat Selection module follows core object-oriented design principles:

- **Class**: Flight
  Manages flight-specific data, including seat allocation.
- **Encapsulation**:
  The seats map is declared as private, restricting direct external access.
- **Composition**:
  The Flight class contains multiple Seat objects, forming a *has-a* relationship.
- **Behavioral Methods**:
  - reserveSeat(String seatNumber) – Reserves a specific seat
  - getSeat(String seatNumber) – Retrieves seat information
  - getAvailableSeatsByClass(SeatClass) – Filters seats by class
  - assignRandomSeat(SeatClass) – Automatically assigns a random available seat

This structure improves maintainability and ensures clear separation of responsibilities.

6.4. Rationale for Using HashMap

The HashMap data structure was selected to optimize seat lookup and reservation performance.

Advantages of HashMap Usage:

1. **Constant-Time Access (O(1))**
   Direct access to seat objects using seat numbers such as "5A".
2. **Fast Availability Checks**
   Seat occupancy status can be validated in constant time.
3. **Key-Value Mapping**
   Maps seat numbers directly to Seat objects.

4. **Scalability for Large Seat Counts**
   With approximately **174 seats**, an ArrayList would require O(n) searches, while HashMap maintains O(1) access.

5. **Efficient Manual Selection**
   When a user selects a seat (e.g., "5A"), the system retrieves and reserves it instantly.

6.5. Time Complexity Analysis

Operation	Time Complexity	Description
getSeat()	O(1)	HashMap lookup by seat number
reserveSeat()	O(1)	Seat lookup and status update
getAvailableSeatsByClass()	O(n)	Iterates over all seats
assignRandomSeat()	O(n)	Filters available seats before selection

This analysis confirms that the module prioritizes constant-time operations for user-triggered actions while maintaining acceptable linear-time processing for seat filtering tasks.

6.6. Detailed Time Complexity Analysis

The Seat Selection module is designed to optimize frequent user-driven operations by prioritizing constant-time access wherever possible. The following analysis evaluates each core method in detail.

getSeat(String seatNumber)

- **Time Complexity:** O(1)
- **Explanation:** The method retrieves a seat object directly from a HashMap using the seat number as the key. This operation involves hash computation followed by bucket access, both of which execute in constant time on average.

```
public Seat getSeat(String seatNumber) {
    return seats.get(seatNumber);
}
```

reserveSeat(String seatNumber)

- **Time Complexity:** O(1)
- **Explanation:** The method performs a HashMap lookup to retrieve the seat and then updates a boolean flag indicating occupancy. No iteration is required, ensuring constant-time execution.

```
public boolean reserveSeat(String seatNumber) {
    Seat seat = seats.get(seatNumber);
    if (seat != null && !seat.isOccupied()) {
        seat.setOccupied(true);
        return true;
    }
    return false;
}
```

getAvailableSeatsByClass(SeatClass seatClass)

- **Time Complexity:** O(n)
  (n = total number of seats, e.g., 174)
- **Explanation:** The method iterates over all seat objects stored in the HashMap to identify seats that match the specified class and are not occupied. Each seat is checked exactly once, resulting in linear time complexity.

```
public List<String> getAvailableSeatsByClass(SeatClass seatClass) {
    List<String> available = new ArrayList<>();
    for (Seat seat : seats.values()) {  // O(n) iteration
        if (!seat.isOccupied() && seat.getSeatClass() == seatClass) {
            available.add(seat.getSeatNumber());
        }
    }
    return available;
}
```

assignRandomSeat(SeatClass seatClass)

- **Time Complexity**: **O(n)**
- **Explanation:** This method first calls getAvailableSeatsByClass(), which performs a linear scan of all seats. After the list of available seats is generated, selecting a random seat from the list is a constant-time operation.

```
public String assignRandomSeat(SeatClass seatClass) {
    List<String> availableSeats = getAvailableSeatsByClass(seatClass);  // O(n)
    if (availableSeats.isEmpty()) {
        return null;
    }
    Random random = new Random();
```

```
    return availableSeats.get(
        random.nextInt(availableSeats.size())
    );  // O(1)
}
```

6.6.1. Overall Evaluation

The detailed analysis shows that:

- **User-triggered actions** such as manual seat selection and reservation operate in **O(1)** time.
- **Seat filtering operations** require **O(n)** time, which is acceptable given the fixed and relatively small number of seats.
- The combination of HashMap-based access and controlled iteration provides a balance between performance, clarity, and scalability.

This design ensures predictable behavior and efficient seat management in real-world airline reservation scenarios.

7. BUSINESS VS ECONOMY SEAT CLASS

7.1. Visual Features

The Seat Class selection mechanism enables users to choose between different comfort levels and pricing options during the ticket booking process. The system provides Business Class seats with premium comfort and higher pricing, Economy Class seats with standard pricing, and Premium Economy seats as an intermediate option. Each seat class is visually distinguished within the interface, allowing users to clearly understand the cost and service differences before completing their reservation. The selected seat class directly affects the final ticket price, ensuring transparency in pricing.

Figure 7.1: *Available cabin options displayed on the selection screen*

7.2. Backend Data Structure of Business vs Economy Seat Class

Seat class definitions and pricing logic are implemented using a Java enumeration. The SeatClass enum represents all available seating categories and associates each class with a predefined price multiplier.

```
public enum SeatClass {
    BUSINESS("Business", 2.5),
    ECONOMY("Economy", 1.0),
    PREMIUM_ECONOMY("Premium Economy", 1.5);
}
```

By encapsulating pricing-related data within the enum, the system ensures consistent and centralized price calculation logic.

7.3. Object-Oriented Design

The SeatClass enumeration follows key object-oriented programming principles. As an enum, it defines a fixed and controlled set of constants representing valid seat classes. Pricing information is stored using private final fields, enforcing encapsulation and immutability. The enum also includes behavior through methods such as getPriceMultiplier() and calculatePrice(double basePrice), allowing each seat class to participate directly in ticket price calculation. This approach demonstrates polymorphic behavior, as each enum constant applies the same pricing interface while maintaining its own multiplier value.

7.4. Rationale for Using Enum

The use of an enum for seat class management provides strong type safety by ensuring that only predefined seat classes can be used throughout the system. Since the set of seat classes is fixed, runtime modification is prevented, reducing the risk of invalid values. Embedding price multipliers within the enum centralizes business rules and eliminates the need for external conditional logic. This design improves maintainability, readability, and consistency across the pricing subsystem while fully adhering to object-oriented encapsulation principles.

7.5. Time Complexity Analysis

All operations related to seat class pricing execute in constant time. Accessing the price multiplier through getPriceMultiplier() is an O(1) operation, as it involves direct access to an enum field. Similarly, the calculatePrice() method performs a simple arithmetic multiplication, which also runs in O(1) time. As a result, seat class–based pricing introduces no measurable performance overhead and scales efficiently regardless of system size.

8. RANDOM SEAT ASSIGNMENT VS MANUAL SEAT SELECTION

8.1. Visual Features

The seat selection interface provides users with two distinct options during the booking process. Users may choose **Random Seat Assignment**, which is offered free of charge, or **Manual Seat Selection**, which incurs an additional fee of 100 TL. When manual selection is chosen, a detailed seat map is displayed, allowing users to visually inspect seat availability and select a specific seat. This dual-option design balances cost efficiency for price-sensitive users with flexibility and control for users who prefer a personalized seating choice.

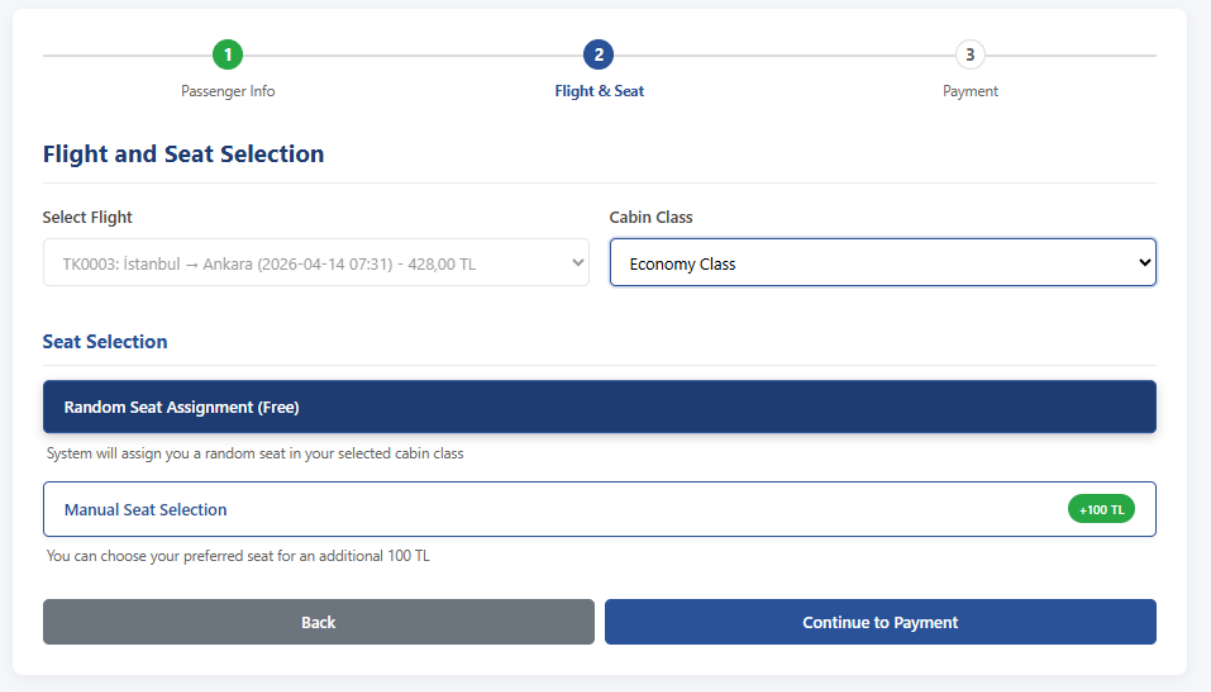

Figure 8.1.1: *User selecting the random seat assignment option*

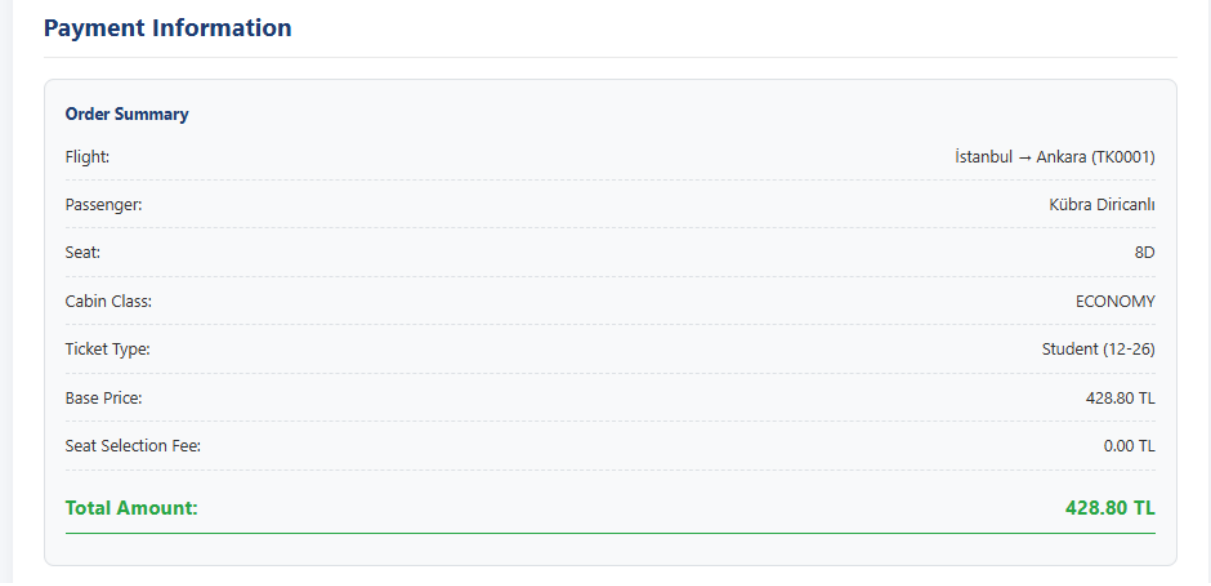

Figure 8.1.2: *Screen displaying the randomly assigned seat information*

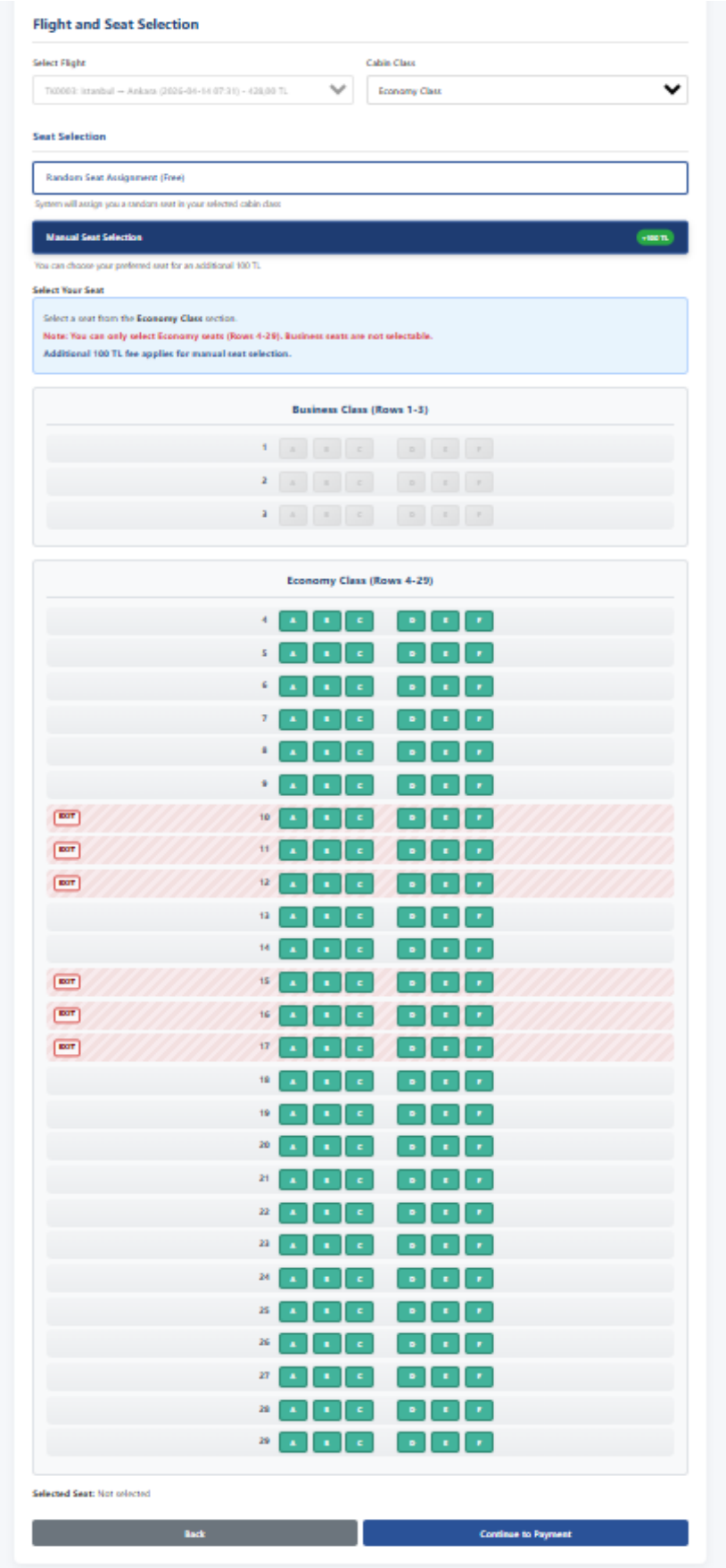

Figure 8.1.3: *User selecting a paid seat option in the economy cabin for an additional fee*

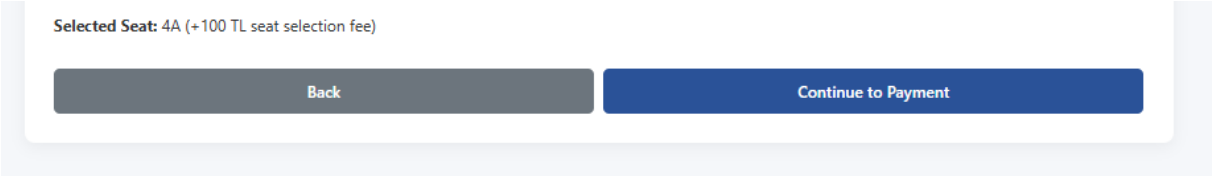

Figure 8.1.4: *Screen displaying the selected seat details*

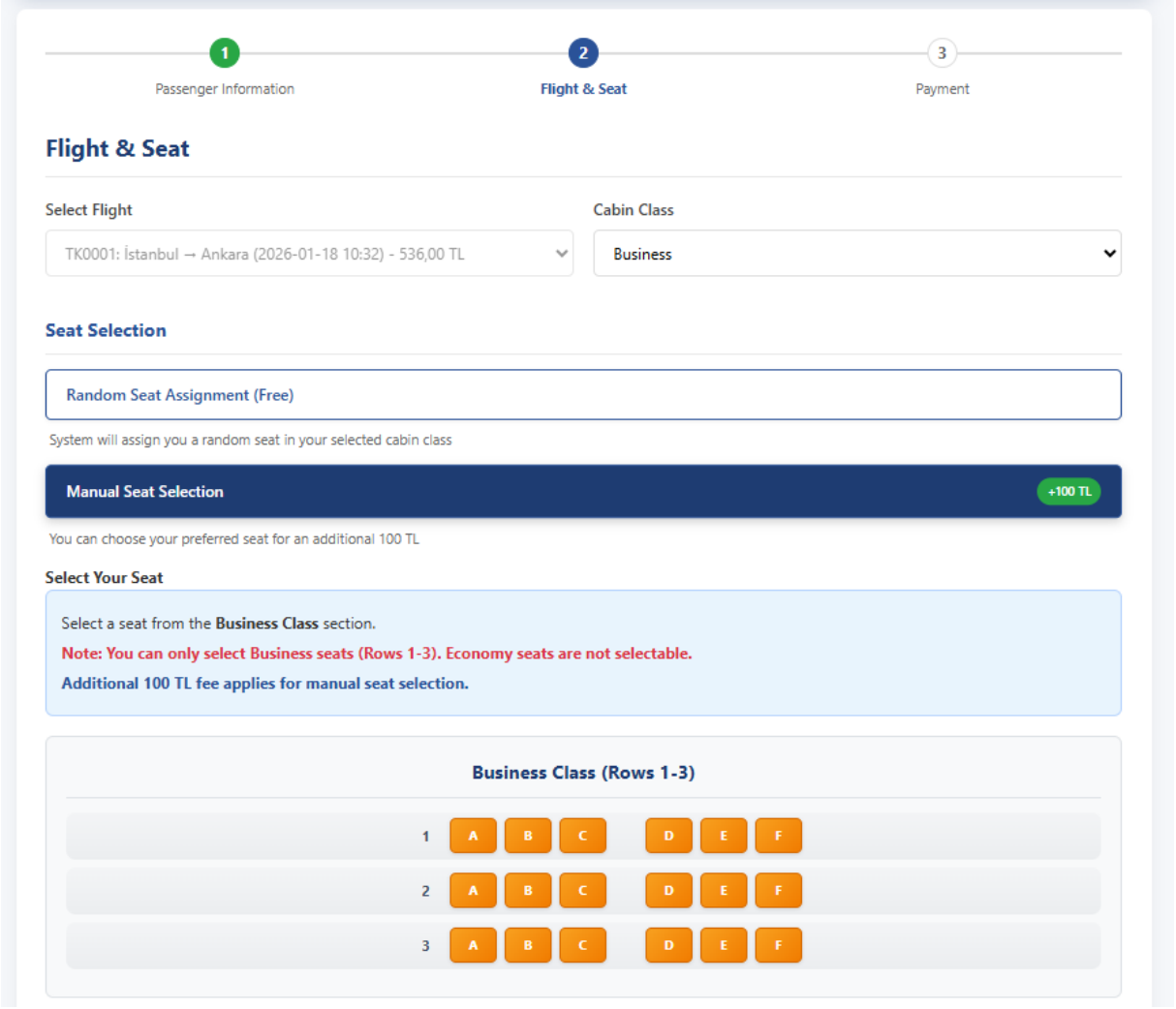

Figure 8.1.5: *User selecting a premium seat option in the business cabin for an additional fee*

8.2. Backend Data Structure of Random Seat Assignment vs Manual Seat Selection

Random seat assignment is implemented within the Flight class using a method that selects a seat from the list of available seats filtered by seat class.

```
public String assignRandomSeat(SeatClass seatClass) {
    List<String> availableSeats = getAvailableSeatsByClass(seatClass);
    Random random = new Random();
    return availableSeats.get(random.nextInt(availableSeats.size()));
}
```

The method relies on a dynamically generated list of available seats and selects one using a random index.

8.3. Object-Oriented Design

The seat assignment mechanism follows object-oriented design principles by encapsulating all seat allocation logic within the Flight class. The system distinguishes between random and manual seat selection through method behavior rather than external conditional logic. Method overloading is used to support different seat assignment strategies, while polymorphism is achieved by allowing the same conceptual operation—seat assignment—to produce different outcomes based on user choice. Encapsulation ensures that the random selection logic remains internal to the class, preventing misuse and maintaining consistency across the system.

8.4. Design Rationale

The use of an ArrayList to store available seats enables efficient iteration and compatibility with random index selection. Filtering seats by seat class ensures that random assignment respects class-based constraints such as Business or Economy seating. The random selection itself executes in constant time, providing a fast and fair seat allocation mechanism. Manual seat selection, on the other hand, leverages direct access to seat objects through a HashMap, allowing users to select a specific seat with minimal computational cost.

8.5. Time Complexity Analysis

- **assignRandomSeat()**: **O(n)** – Available seats are first filtered by iterating over all seats in the selected class, followed by an **O(1)** random index selection from the resulting list.
- **Manual Seat Selection**: **O(1)** – The selected seat is accessed directly from a HashMap using the seat number as the key, without any iteration.

9. PAYMENT

9.1. Visual Features

The Payment module provides a structured interface for securely completing ticket purchases. Users are required to select a card type (Visa or MasterCard) and enter essential card details, including cardholder name, card number, expiry date, and CVV. A payment summary section displays the final price, reflecting seat class and optional services. The system also visually communicates the payment outcome through clearly defined statuses such as Success, Failed, Pending, or Refunded, ensuring transparency throughout the transaction process.

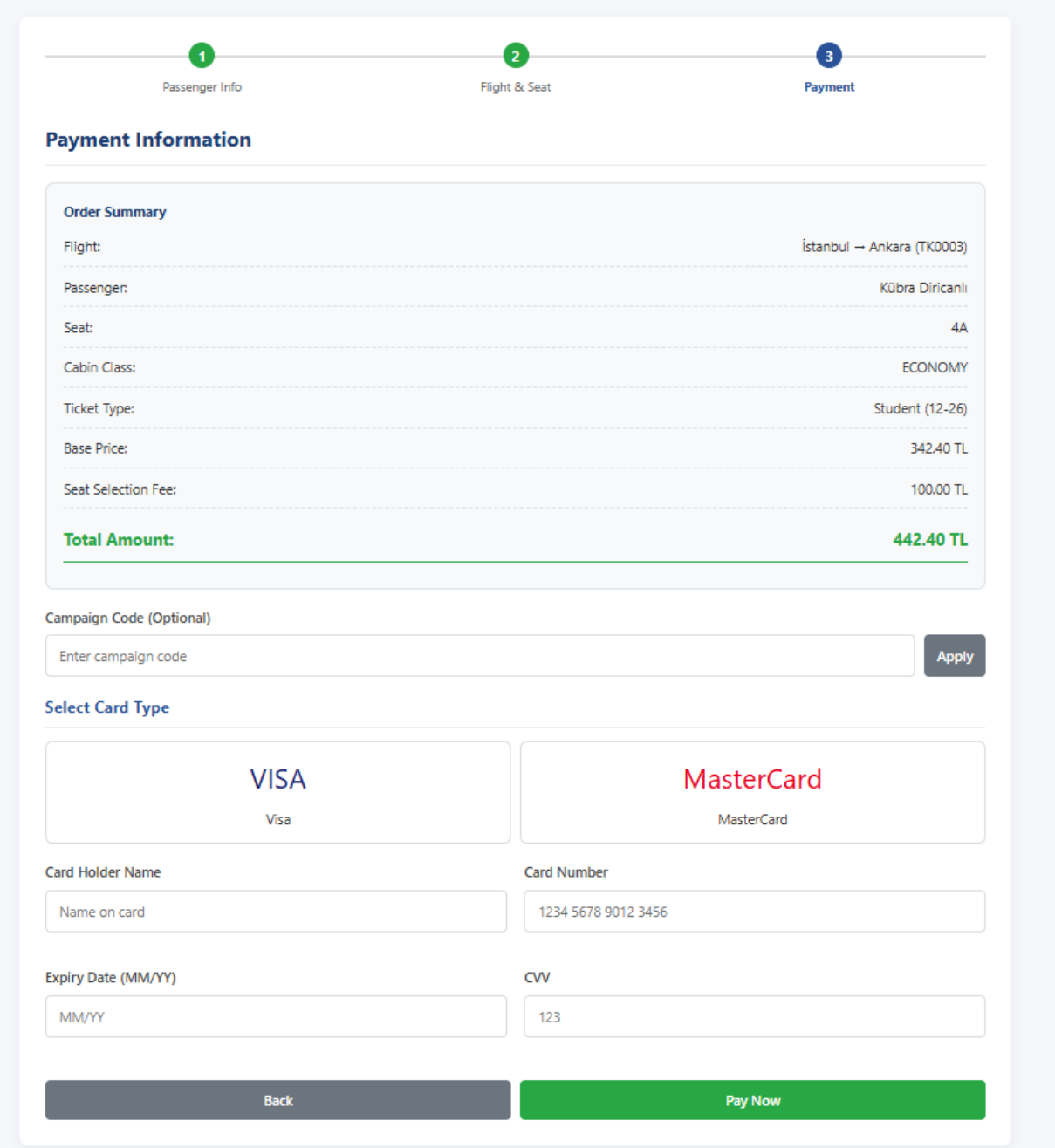

Figure 9.1.1: *View of the payment information screen*

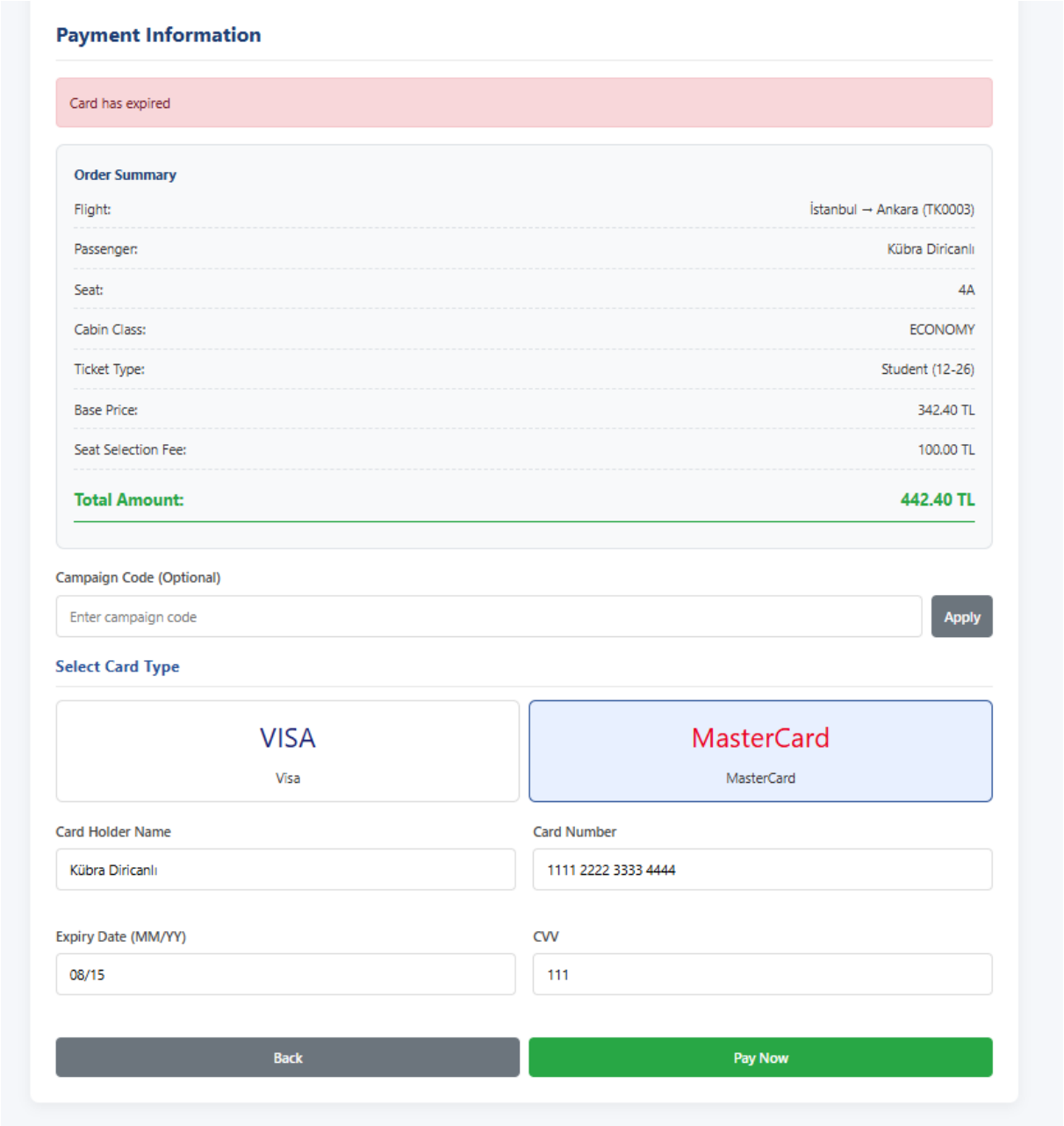

Figure 35: *Validation error for entering a past expiration year in the card details*

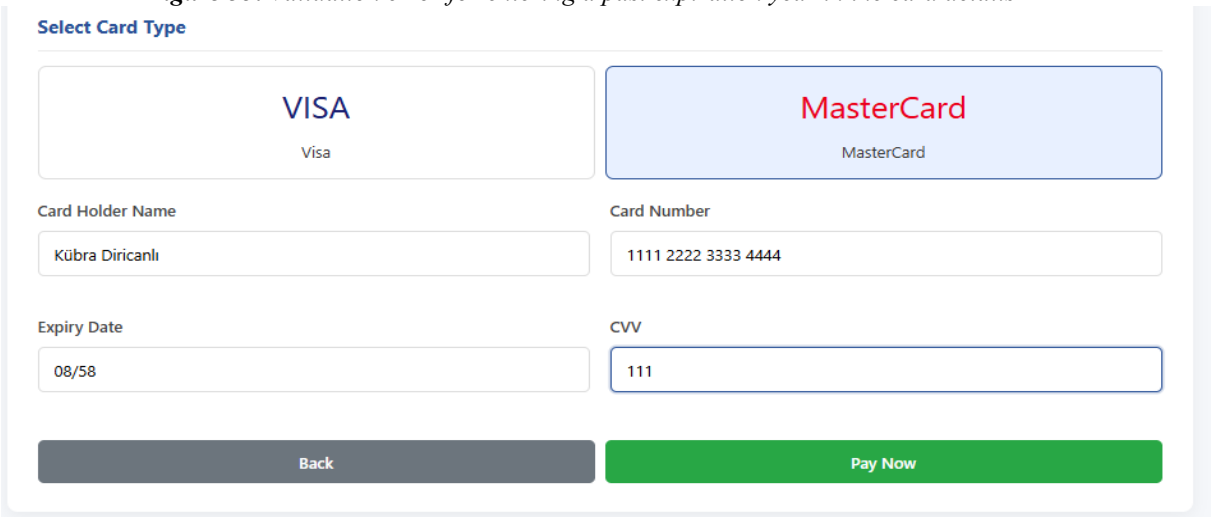

Figure 9.1.2: *Successful entry of valid credit card information*

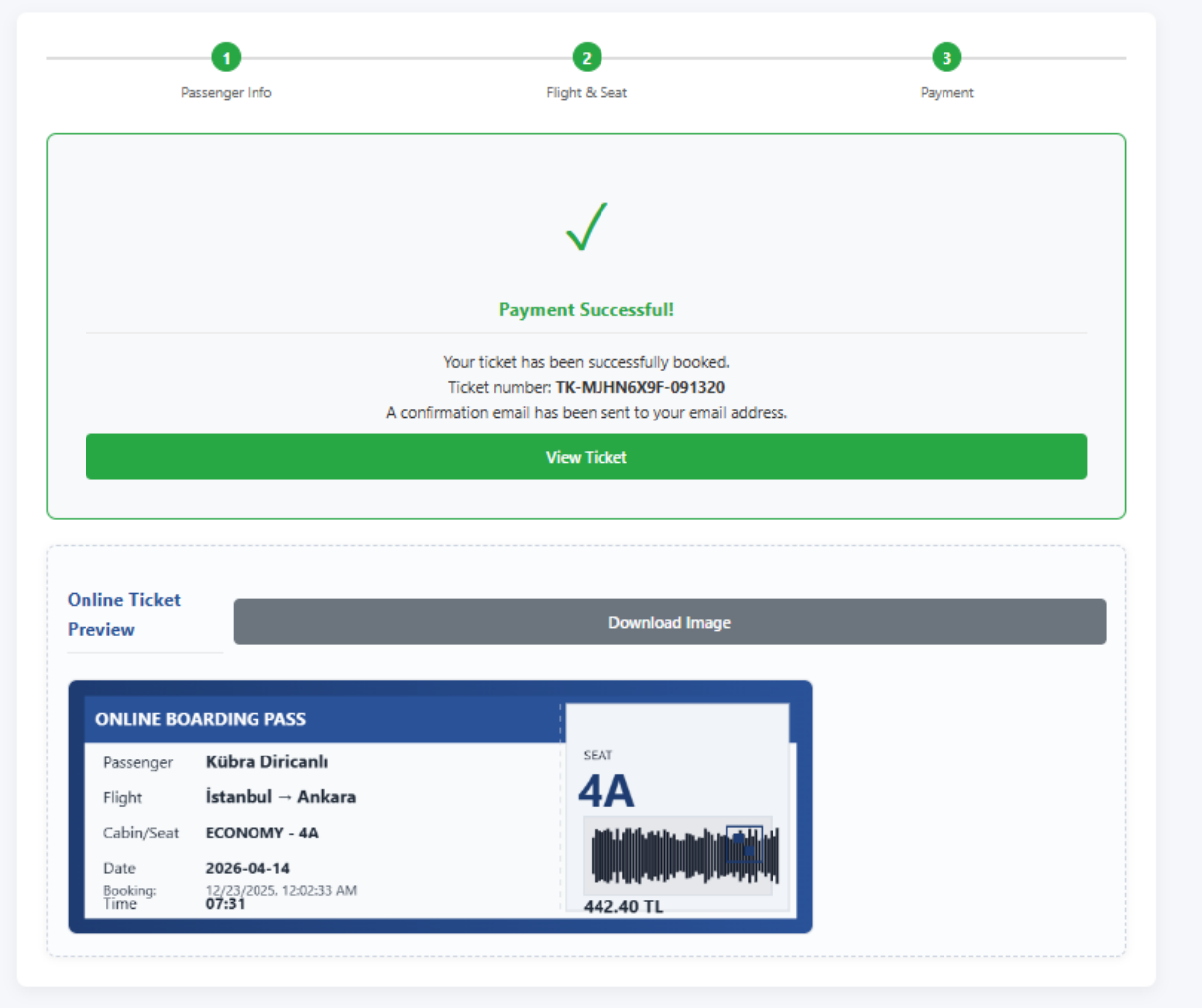

Figure 9.1.3: *Payment confirmation and display of the online e-ticket*

9.2. Backend Data Structure of Payment

Payment-related data is encapsulated within the PaymentInfo class, which represents a single payment transaction and its associated attributes.

```
public class PaymentInfo {
    private String cardNumber;
    private String cardHolderName;
    private String expiryDate;
    private String cvv;
    private String cardType;
    private Date paymentDate;
    private PaymentStatus status;

    public enum PaymentStatus {
        SUCCESS, FAILED, REFUNDED, PENDING
    }
```

```
    private String detectCardType(String cardNumber) {
        if (cardNumber.startsWith("4")) return "VISA";
        if (cardNumber.startsWith("5")) return "MASTERCARD";
        if (cardNumber.startsWith("3")) return "AMEX";
        return "UNKNOWN";
    }
}
```

The card type is automatically detected based on the card number prefix, eliminating the need for redundant user input and reducing potential errors.

9.3. Object-Oriented Design

The Payment module applies core object-oriented principles by encapsulating all payment-related fields within the PaymentInfo class. All attributes are declared as private, enforcing encapsulation and protecting sensitive data. Controlled access is provided through getter and setter methods. The PaymentStatus enum defines a restricted set of valid payment states, ensuring type safety and preventing invalid status assignments. The PaymentInfo object is used within the FlightBookingSystem through composition, representing a clear *has-a* relationship between a booking and its payment details.

9.4. Design Rationale

Using a dedicated class for payment information allows related data to be grouped logically and managed consistently. The enum-based representation of payment status ensures compile-time safety and improves code readability. Automatic card type detection simplifies the user experience while keeping validation logic centralized within the payment domain. This structure improves maintainability, security, and clarity in the payment processing workflow.

9.5. Time Complexity Analysis

Payment-related operations are computationally lightweight. The processPayment() operation executes in constant time, as it involves basic validation and state updates. The detectCardType() method also runs in O(1) time due to simple prefix checks using startsWith(). Similarly, operations such as retrieving the last four digits of a card number rely on substring operations and execute in constant time. As a result, the payment module introduces no performance bottlenecks.

10. MY TICKETS

10.1. Visual Features

The My Tickets module allows users to view and manage their existing reservations. Tickets are categorized into active and cancelled lists, enabling clear separation of valid and invalid bookings. Each ticket entry displays essential details such as ticket number, passenger information, flight details, assigned seat, and total price. This organized presentation enhances usability and simplifies post-booking management.

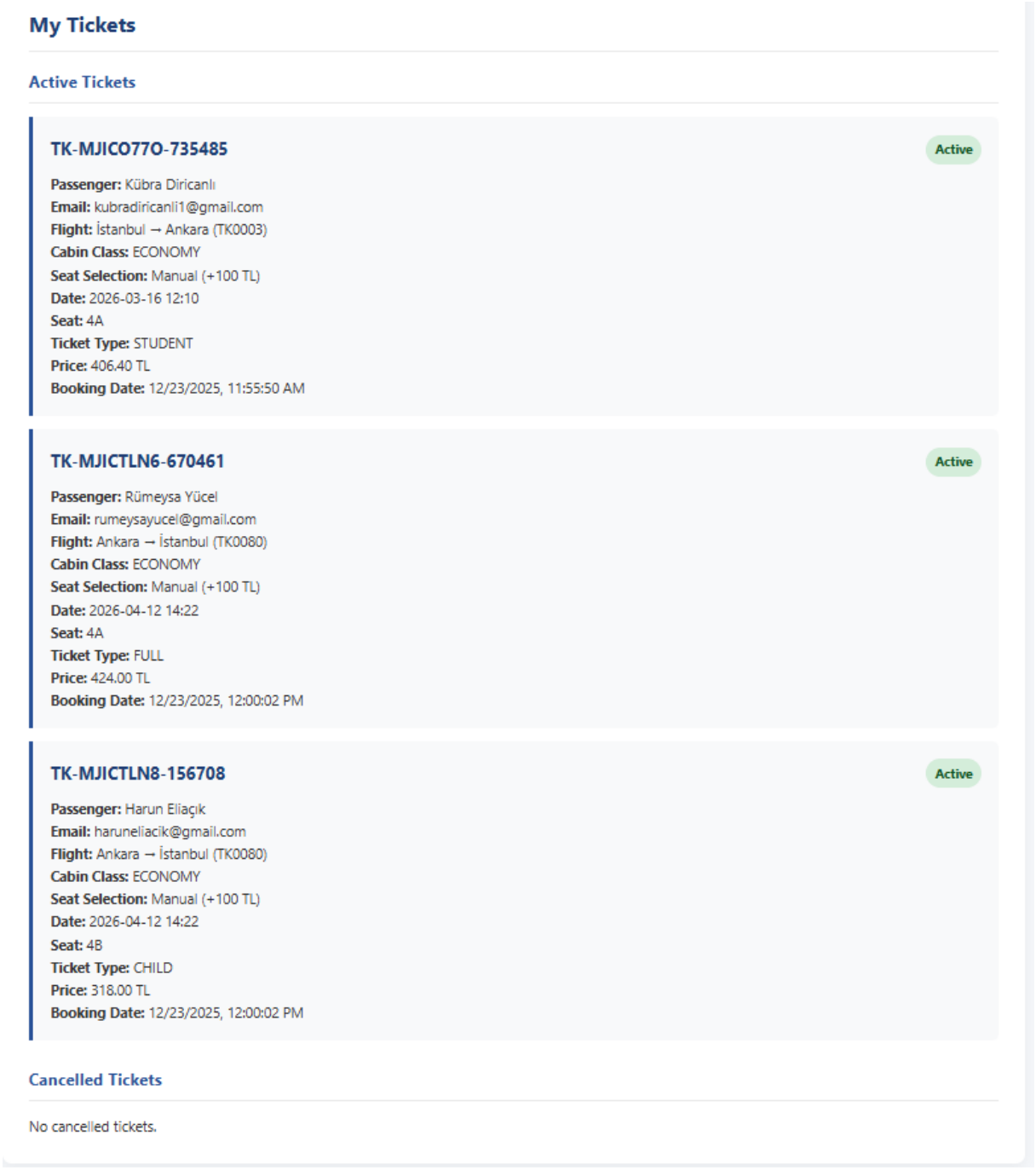

Figure 10.1: *Purchased tickets displayed in the "My Tickets" section of the user profile*

10.2. Backend Data Structure of My Tickets

Ticket management is implemented within the FlightBookingSystem class using a HashMap that maps ticket numbers to Ticket objects.

```
private Map<String, Ticket> ticketMap;
this.ticketMap = new HashMap<>();

public Ticket findTicketByNumber(String ticketNumber) {
    return ticketMap.get(ticketNumber);
}

public boolean cancelTicket(String ticketNumber) {
    Ticket ticket = ticketMap.get(ticketNumber);
    if (ticket == null) {
        return false;
    }
    return ticket.cancelTicket();
}
```

Each ticket number acts as a unique key, enabling direct access to ticket data.

10.3. Object-Oriented Design

The ticket management mechanism adheres to object-oriented principles by encapsulating the ticket collection within a private HashMap. The FlightBookingSystem class provides controlled access through dedicated methods for searching, cancelling, and listing tickets. The key-value relationship between ticket numbers and Ticket objects ensures a clear and efficient mapping. Additional methods such as retrieving tickets by passenger ID further support modular and extensible design.

10.4. Rationale for Using HashMap

HashMap was selected due to its constant-time access characteristics. Ticket lookup and cancellation operations are performed frequently, making O(1) access essential for performance. The key-value mapping between ticket numbers and ticket objects provides a natural and intuitive data representation. This structure ensures that ticket-related operations remain fast and scalable, even as the number of bookings increases.

10.5. Time Complexity Analysis

Finding a ticket by its number executes in O(1) time due to direct HashMap lookup. Ticket cancellation also operates in O(1) time, as it relies on immediate access to the ticket object followed by a state update. Retrieving tickets by passenger ID requires iterating over all stored tickets and therefore executes in O(n) time, where n represents the total number of tickets. Overall, the module is optimized for fast access to individual tickets while maintaining acceptable performance for list-based operations.

11. TICKET CANCELLATION (CANCEL TICKET)

11.1. User Interface Features

The ticket cancellation screen allows the user to enter a valid ticket number and initiate the cancellation process via a dedicated **"Cancel Ticket"** button. After the operation, the system provides a confirmation message indicating whether the cancellation was successful or failed. This interaction ensures clarity and prevents accidental cancellations.

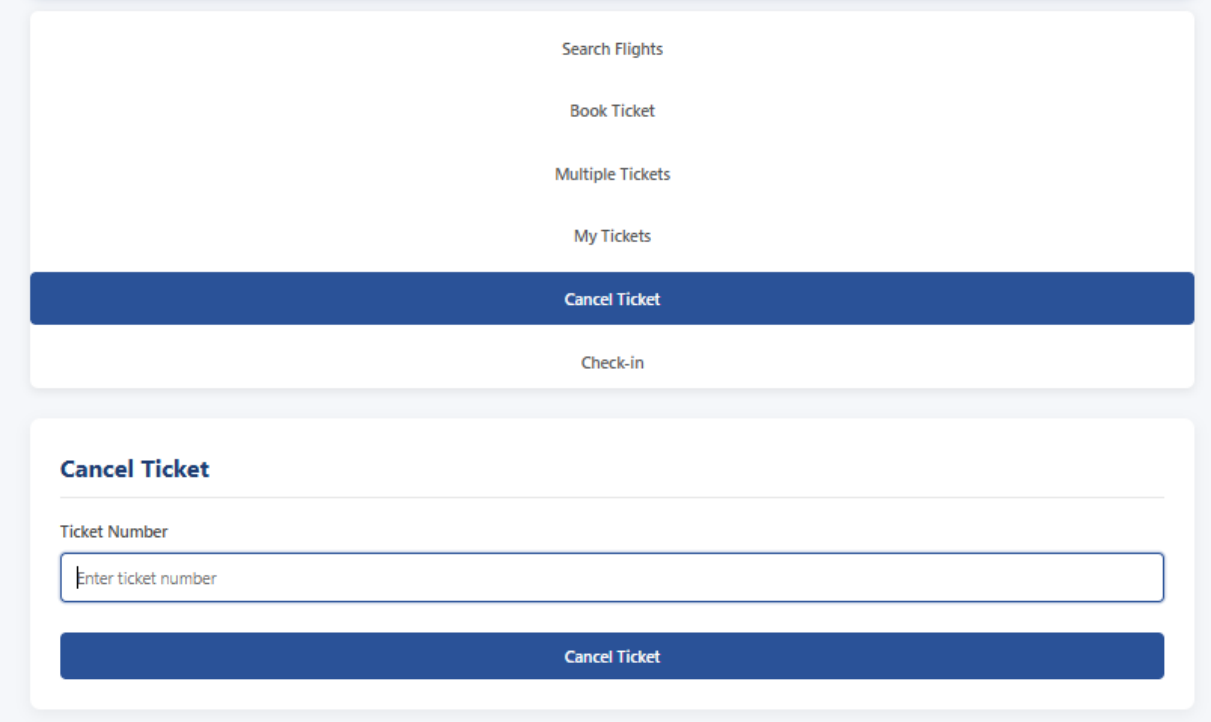

Figure 11.1.1: *Ticket cancellation screen*

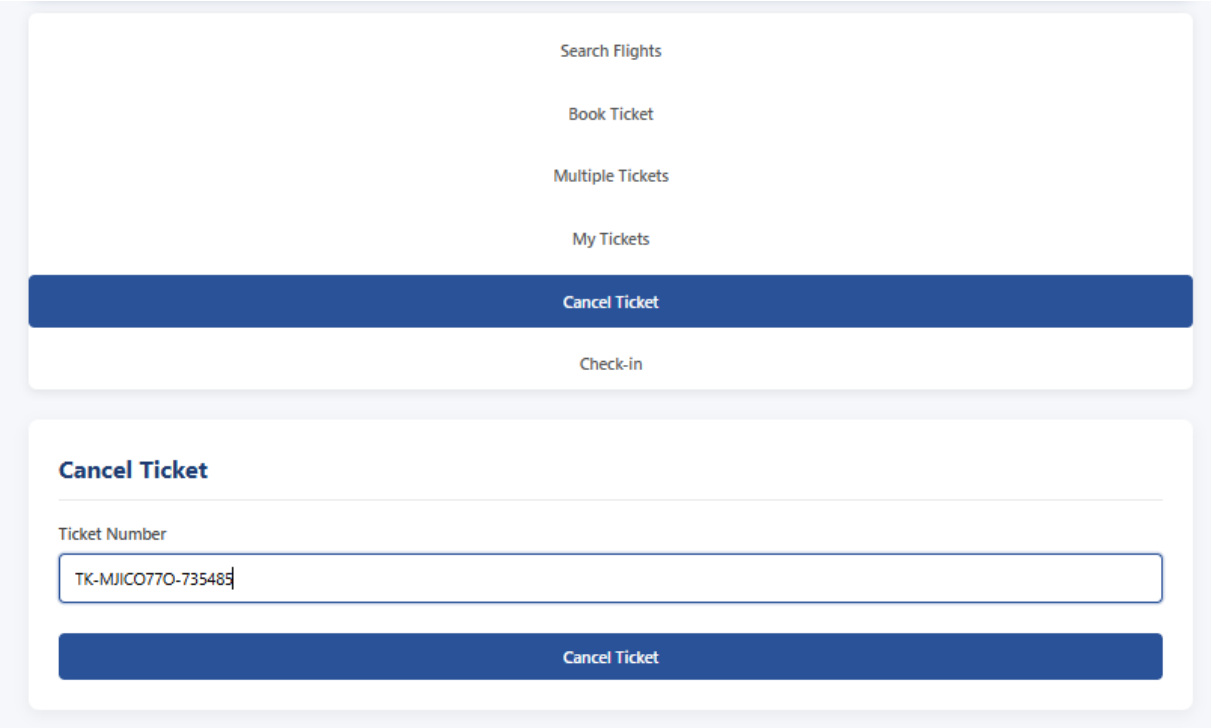

Figure 11.1.2: *Entry of the ticket number to be canceled*

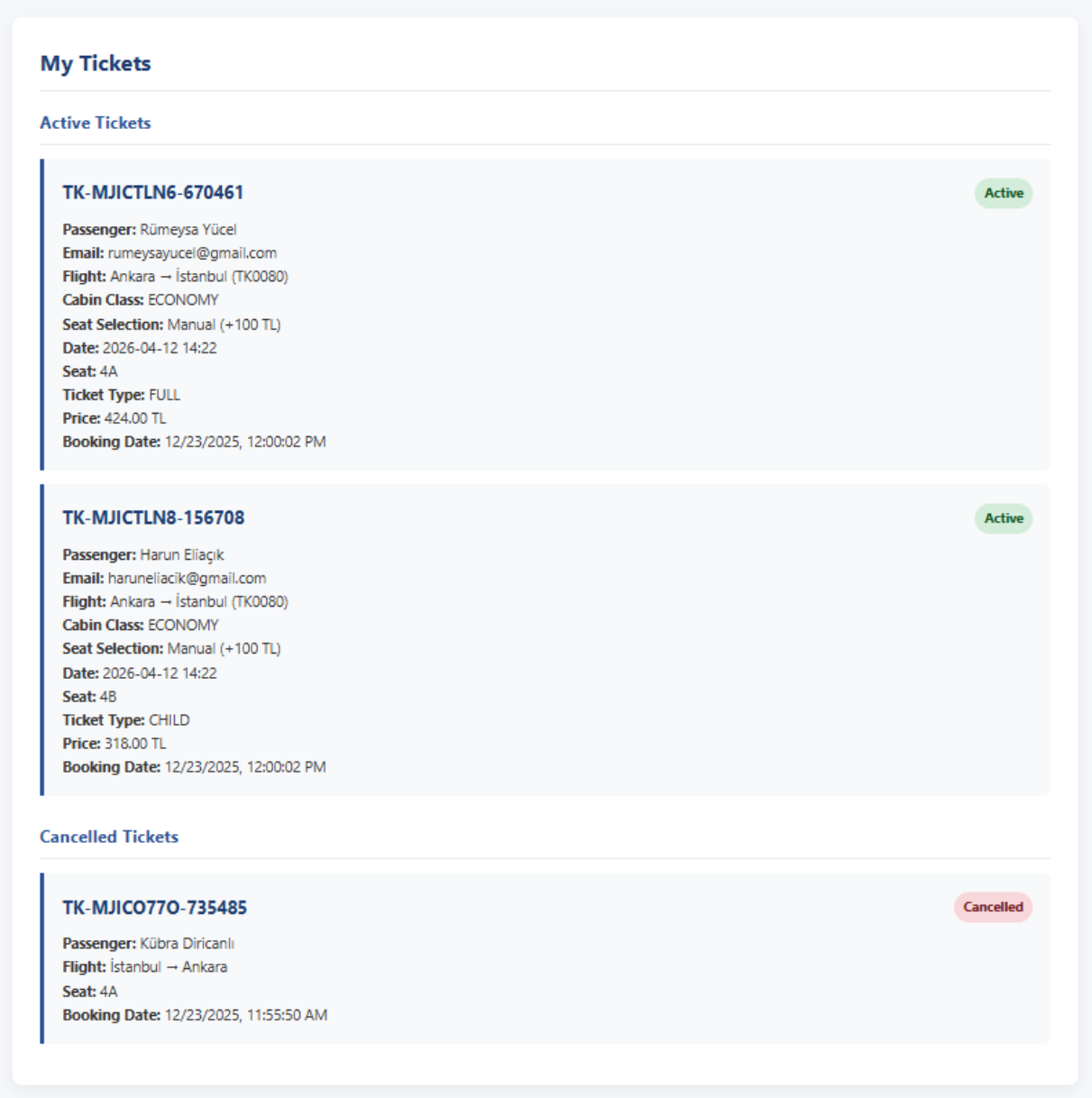

Figure 11.1.3: *View of the cancelled ticket displayed in the "My Tickets" section*

11.2. Backend Data Structure of Ticket Cancellation

The cancellation logic is implemented within the FlightBookingSystem class using a combination of a HashMap and a priority-based data structure.

```
private Map<String, Ticket> ticketMap;
private ReservationPriorityQueue reservationPriorityQueue;
```

The ticketMap stores ticket objects indexed by their unique ticket numbers, while the ReservationPriorityQueue maintains reservation order based on check-in or priority rules.

11.3. Object-Oriented Design

The cancellation feature follows core object-oriented principles. The FlightBookingSystem class encapsulates both the ticket storage and reservation priority management. Ticket lookup and cancellation logic are handled through a dedicated cancelTicket() method, ensuring that business rules are centralized and not exposed externally. The use of composition allows multiple data structures to cooperate within the same class without violating single-responsibility principles.

11.4. Design Rationale

A HashMap is used to provide direct access to ticket objects via their ticket numbers, which is critical for fast and frequent cancellation operations. The priority queue is used to maintain and update reservation order when a ticket is canceled, ensuring that system-level constraints such as priority-based reservations remain consistent. Combining these two structures allows the system to balance speed and correctness.

11.5. Time Complexity

The cancelTicket() operation runs in **O(1)** time for retrieving the ticket from the HashMap, followed by **O(log n)** time for removing the corresponding reservation from the priority queue. This results in an overall time complexity of **O(log n)**, where *n* represents the number of active reservations in the priority queue.

12. CHECK-IN OPERATION

12.1. User Interface Features

The check-in interface requires the user to enter the **ticket number** and **passenger surname**, followed by triggering the process via a **"Check-in"** button. Upon successful validation, the system displays a **check-in confirmation message**, ensuring clarity and immediate feedback for the user.

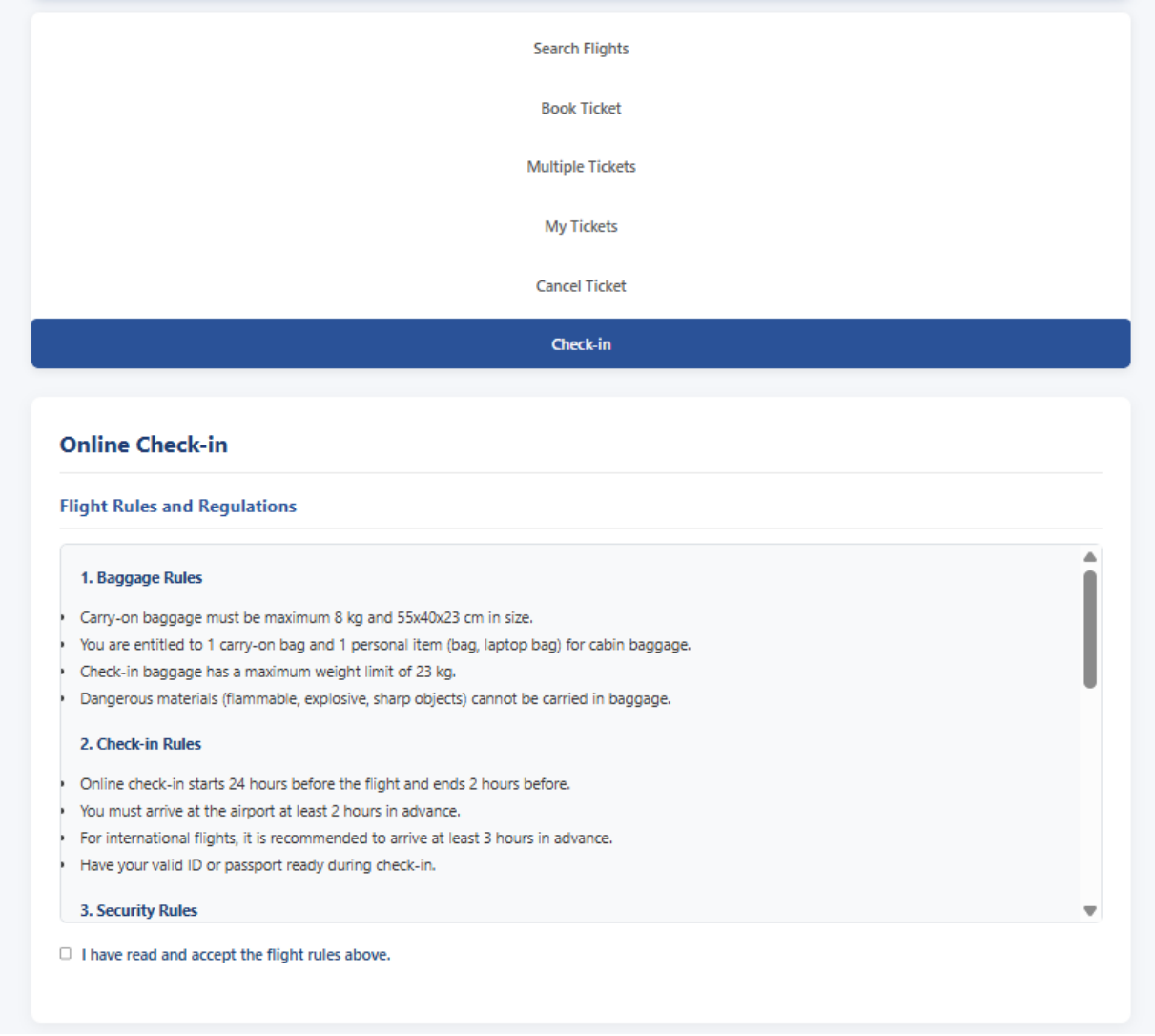

Figure 12.1.1: *View of the online check-in screen*

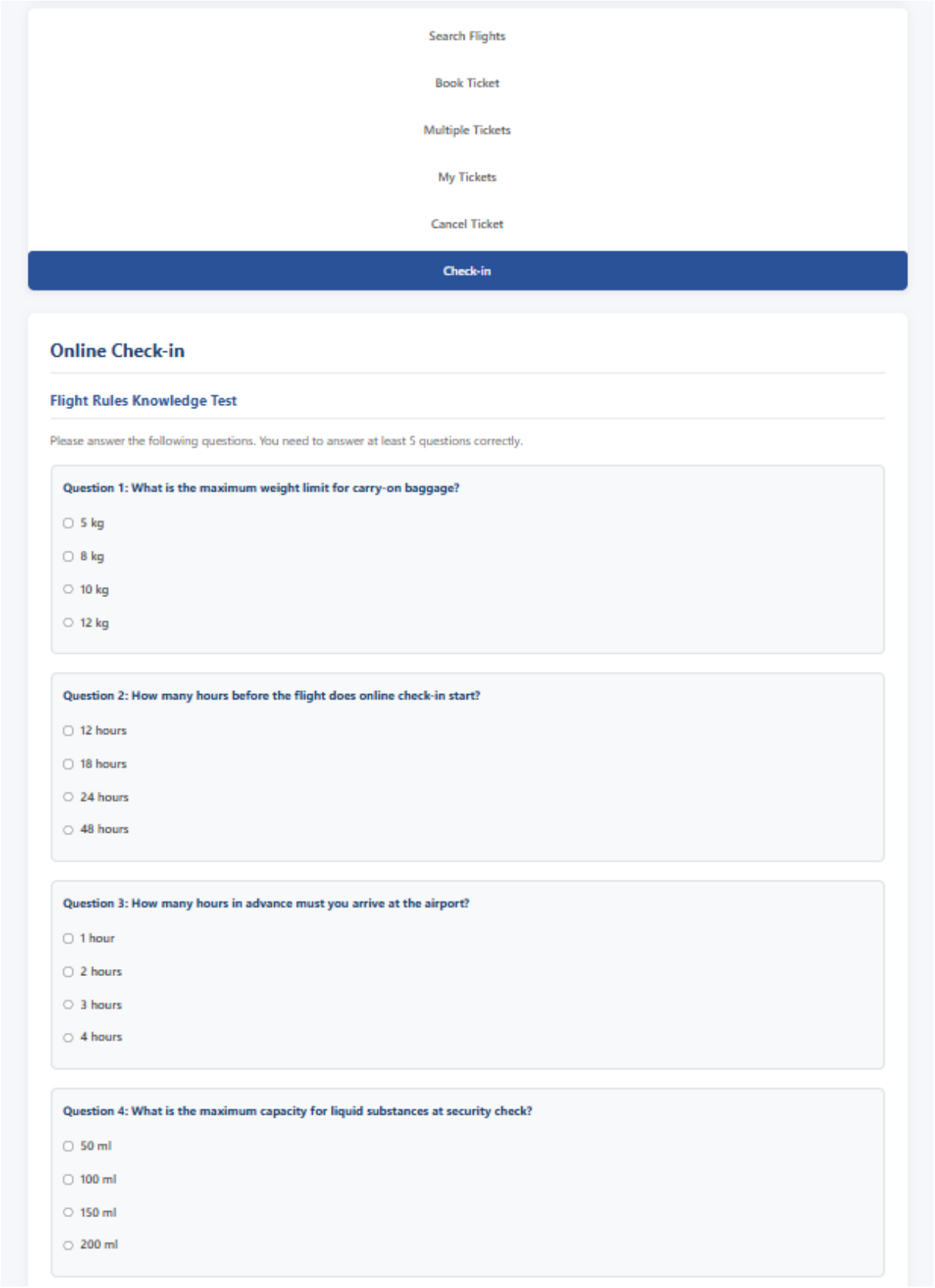

Figure 12.1.2: *Mini-quiz screen covering flight information and safety guidelines prior to check-in*

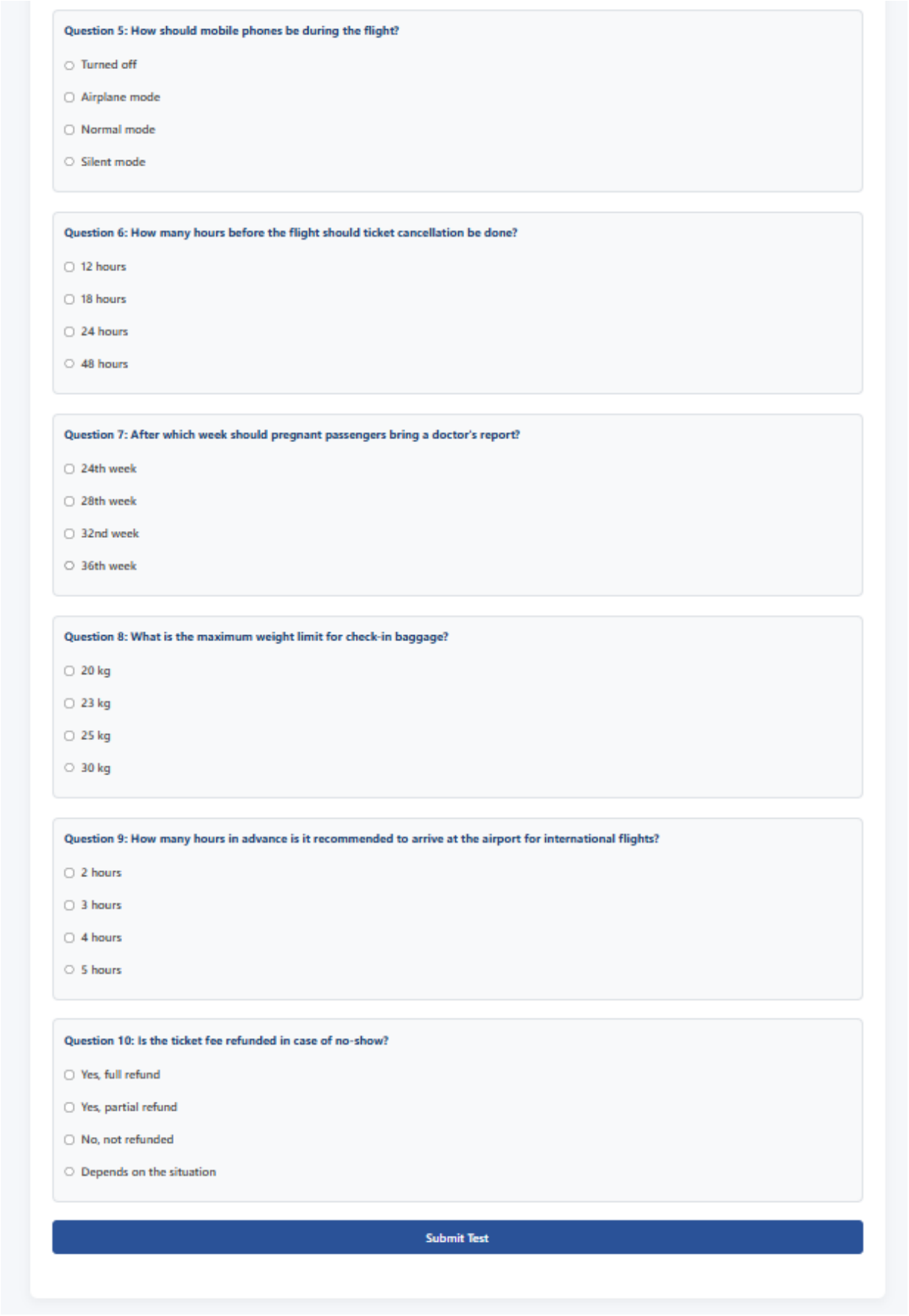

Figure 12.1.3: *Progression of the 10-question mini-quiz during the pre-check-in phase*

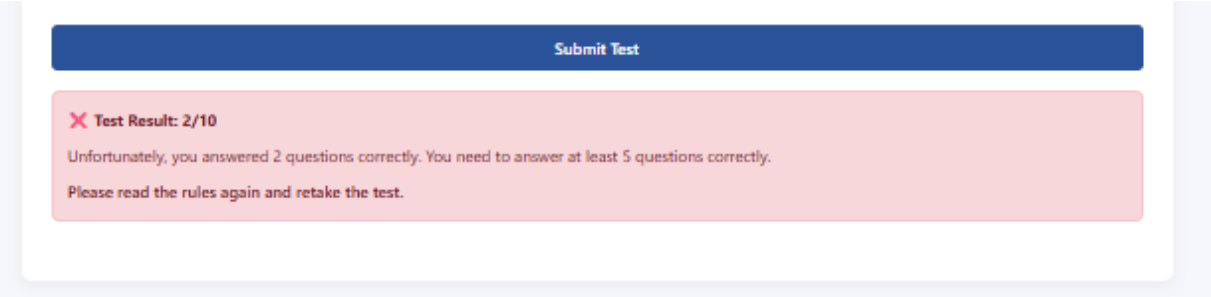

Figure 12.1.4: *Redirection to the quiz after failing to meet the minimum score (less than 5 correct answers)*

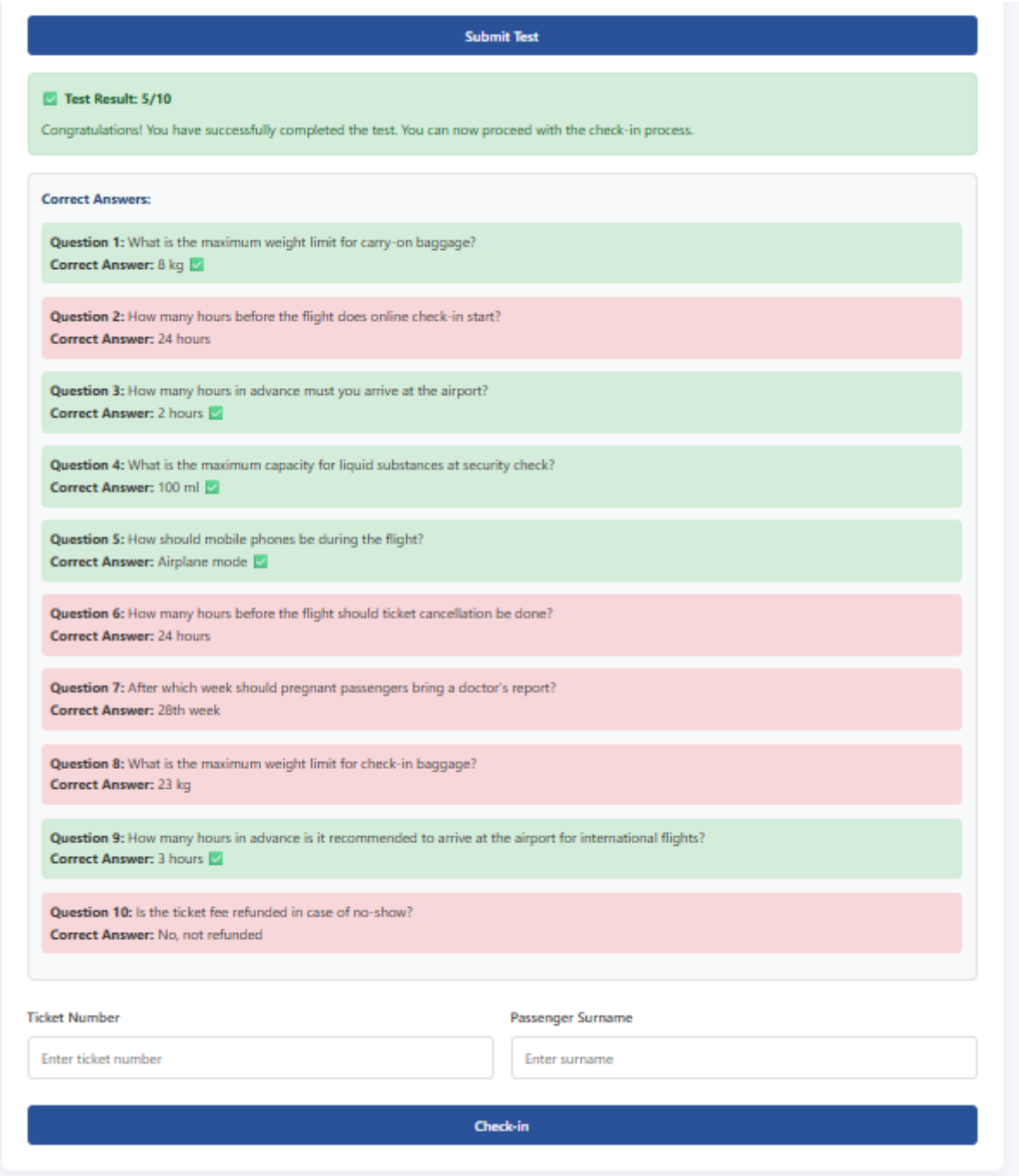

Figure 12.1.5: *Quiz results screen displaying correct answers and enabling the check-in button*

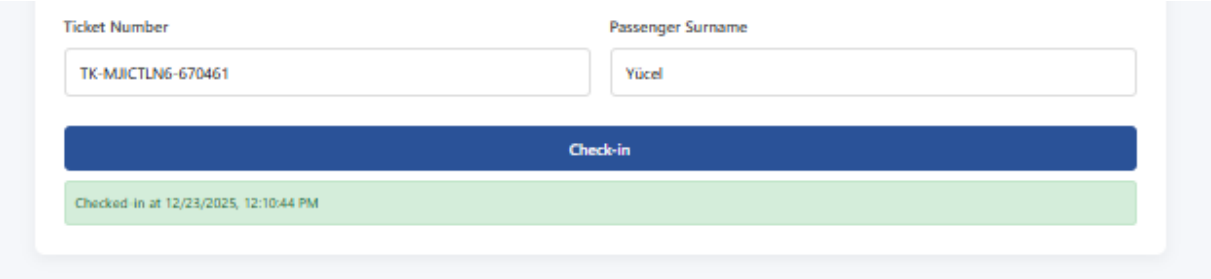

Figure 12.1.6: *Entry of booking details for the check-in process*

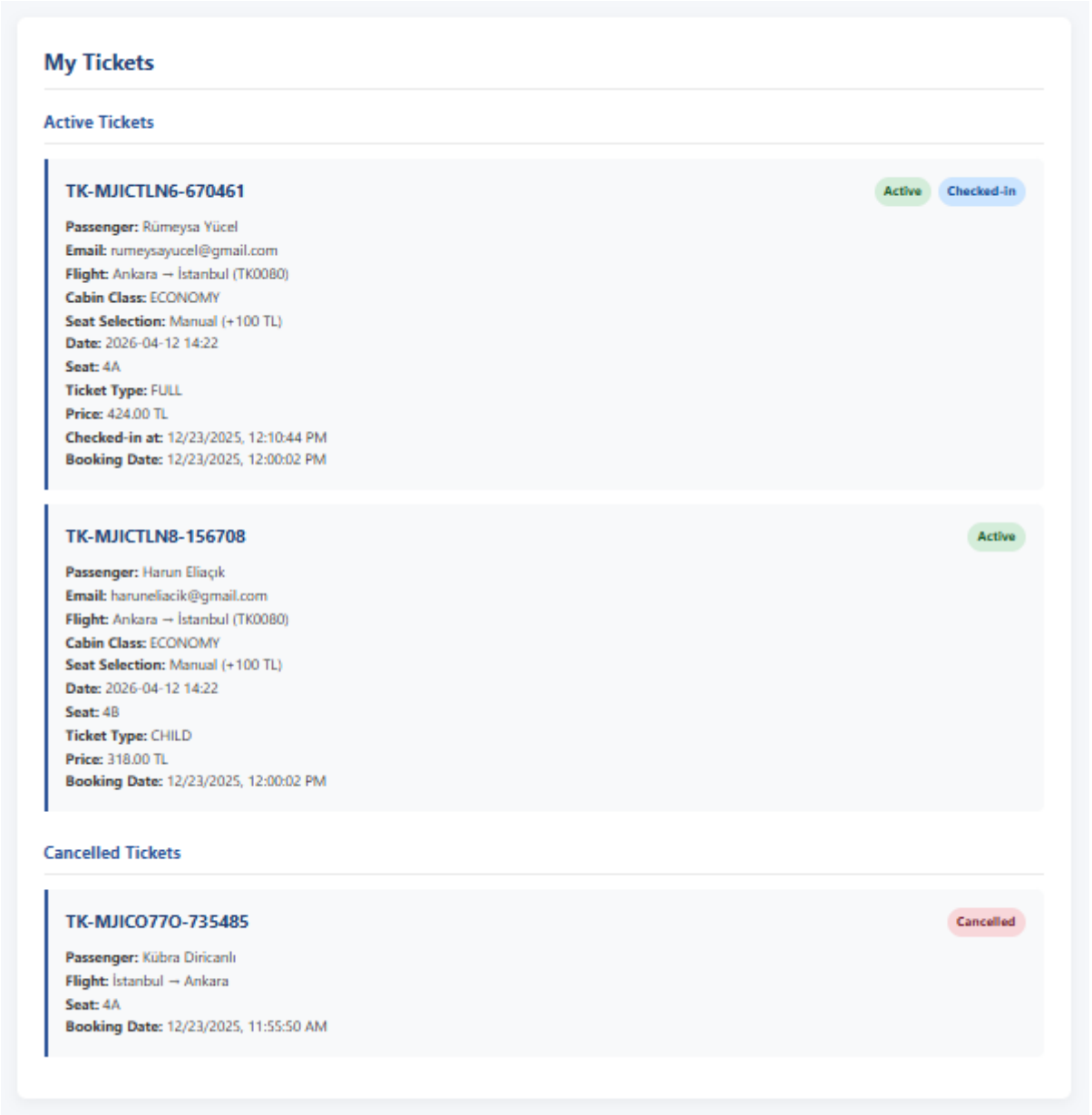

Figure 12.1.7: *View of the checked-in ticket displayed in the "My Tickets" section*

12.2. Backend Data Structure of Check-in Operation

The check-in mechanism is implemented using a **Binary Heap–based Priority Queue**, defined in ReservationPriorityQueue.java. The heap is implemented with an ArrayList to maintain a compact array-based structure, while a HashMap is used to provide constant-time access to reservation indices.

```
private List<Reservation> heap;
private Map<String, Integer> reservationIndexMap;
```

The add() method inserts a new reservation into the heap and restores heap order using a heapify-up operation. The poll() method retrieves and removes the highest-priority reservation (earliest check-in) and restores the heap using heapify-down.

12.3. Object-Oriented Design

The ReservationPriorityQueue class encapsulates the entire heap structure, preventing external modification and enforcing controlled access. Composition is used by combining a List<Reservation> with a Map<String, Integer> to achieve both ordered priority handling and fast index lookup. Core operations such as add, poll, and remove are implemented as class methods, ensuring cohesion and clear responsibility separation within the system.

12.4. Design Rationale

A binary heap is chosen to efficiently manage reservations based on **check-in priority**, where reservations with earlier check-in times are processed first. The array-based structure provided by ArrayList offers optimal memory locality and fast access, while the auxiliary HashMap eliminates the need for linear searches when removing a specific reservation. This hybrid design balances performance with implementation simplicity and aligns well with real-world airline check-in logic.

12.5. Time Complexity Analysis

The **add()** operation runs in **O(log n)** time. Although inserting into the end of the ArrayList and updating the HashMap are both O(1), the heapify-up process may traverse up to log n levels to restore heap order.

The **poll()** operation also has a time complexity of **O(log n)**. Accessing the minimum element (heap root) occurs in O(1) time, and removing it from the HashMap is also O(1). However, restoring heap order through heapify-down requires O(log n) time in the worst case.

The **remove(String reservationId)** operation executes in **O(log n)** time. The index of the reservation is obtained from the HashMap in O(1), and removal from the heap is performed in constant time. The subsequent heapify-up or heapify-down operation ensures heap consistency and dominates the overall complexity.

13. MULTIPLE TICKET RESERVATION

13.1. User Interface Features

The multiple ticket reservation interface allows users to specify the **number of passengers** and enter **separate personal information for each passenger** through individual forms. All tickets are booked for the **same flight**, while allowing **different seat assignments** per passenger. The system supports **bulk payment processing**, followed by sending a **single confirmation email** that summarizes all reserved tickets.

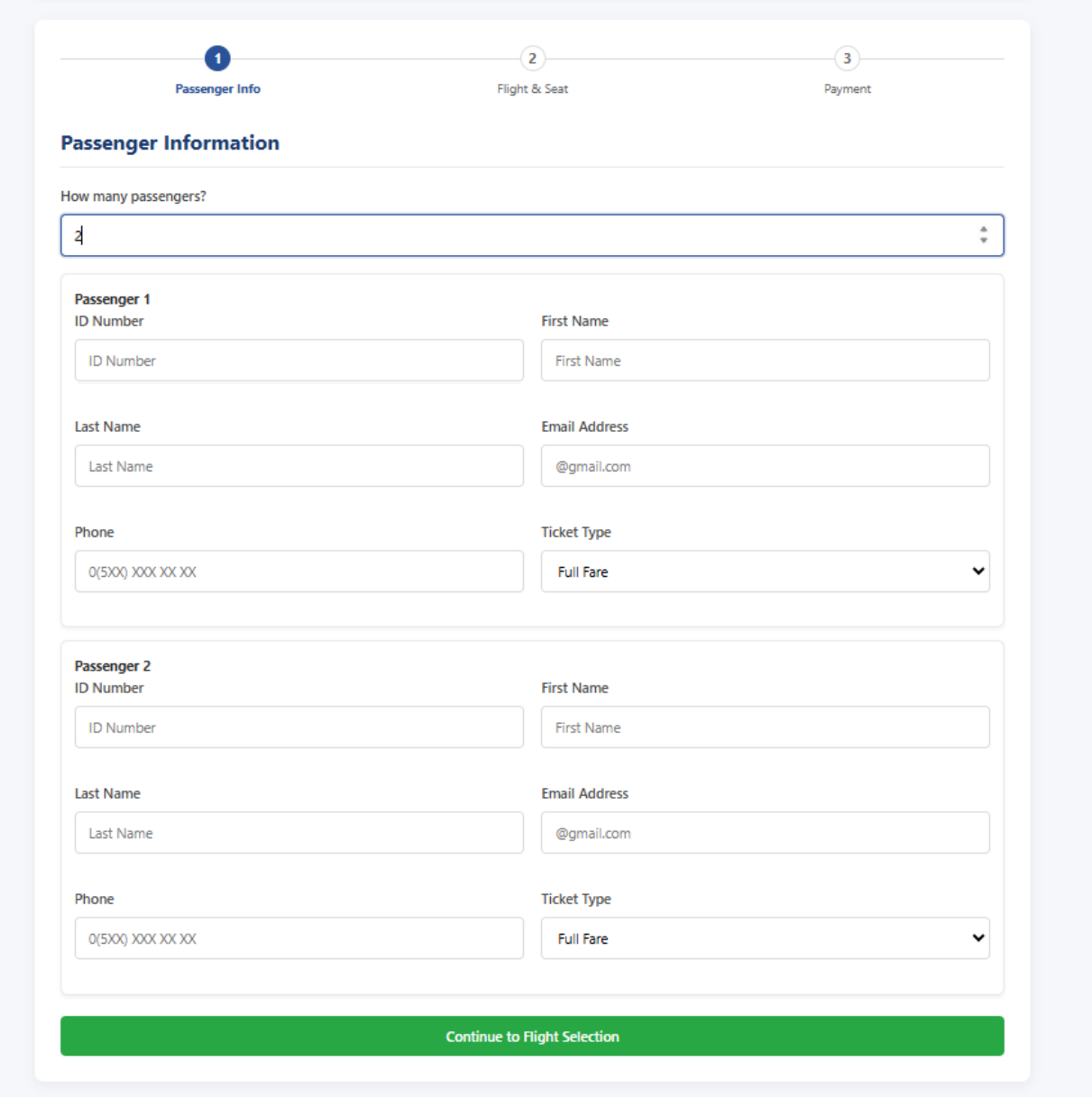

Figure 13.1.1: *View of the multi-passenger booking screen*

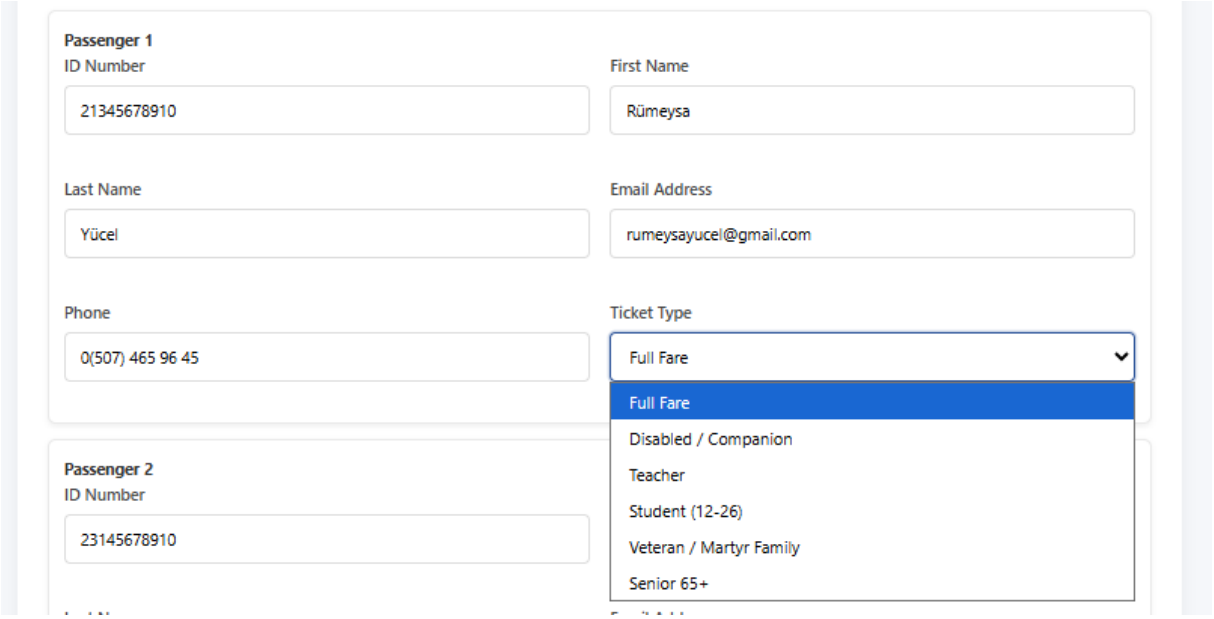

Figure 13.1.2: *Screen for selecting the ticket type (fare class) for the first passenger*

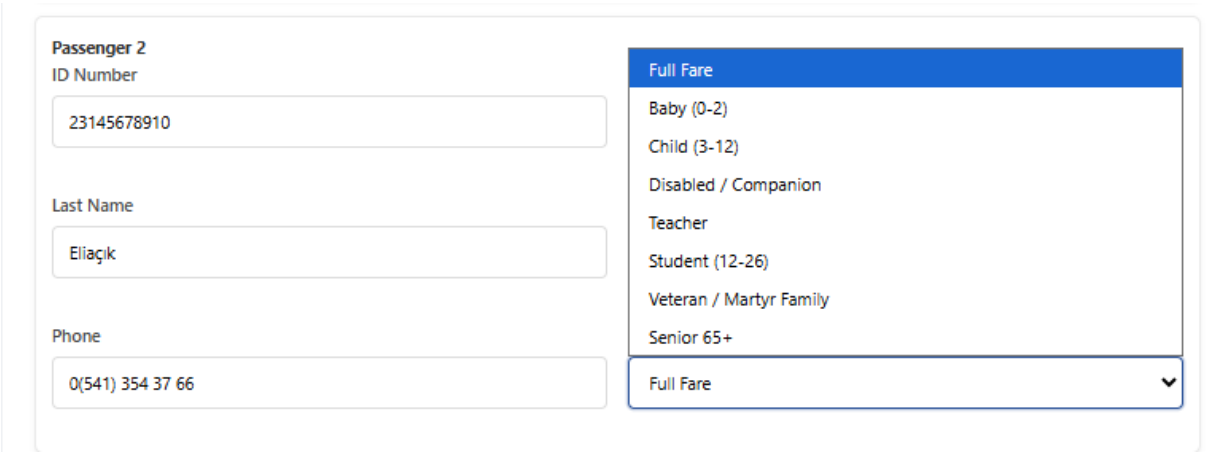

Figure 13.1.3: *Selection of fare types for subsequent passengers and dynamic update of the selection count*

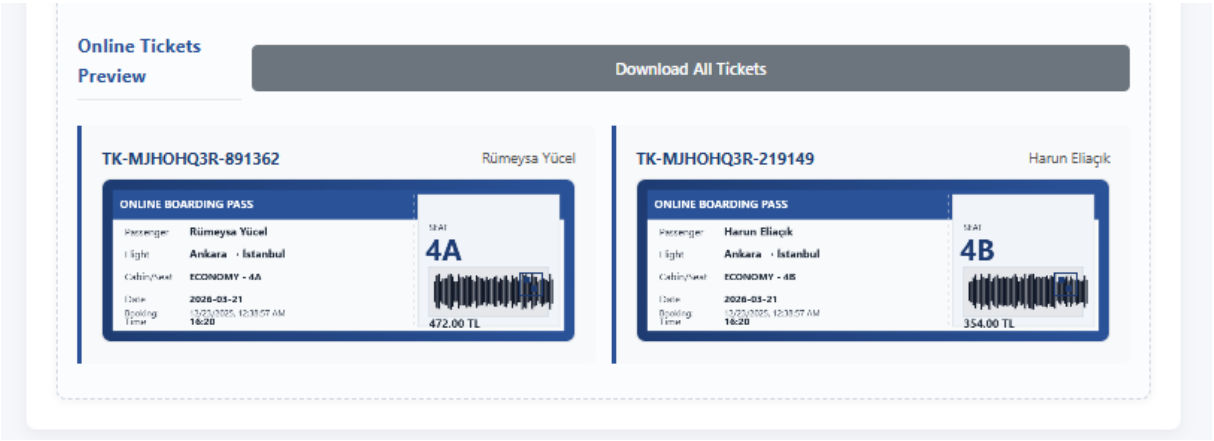

Figure 13.1.4: *Display of the issued online e-tickets for all passengers*

13.2. Backend Data Structure of Multiple Ticket Reservation

Multiple ticket reservations are managed using an ArrayList<Ticket> within FlightBookingSystem.java. Each passenger is processed individually in a loop, and a distinct Ticket object is created for every passenger while maintaining association with the same flight.

```
private List<Ticket> tickets;
this.tickets = new ArrayList<>();

for (Passenger passenger : passengers) {
   Ticket ticket = bookTicket(passenger, flightNumber, cabinClass, ...);
   tickets.add(ticket);
}
```

For post-booking communication, the EmailService handles confirmation delivery. The sendMultipleTicketsConfirmation method generates a single HTML-based email containing a consolidated summary and detailed information for all tickets associated with the reservation.

```
public boolean sendMultipleTicketsConfirmation(String toEmail, List<Ticket>
tickets) {
   // HTML email with multiple ticket details
}
```

13.3. Object-Oriented Design

Each ticket is represented as an independent Ticket object, demonstrating **composition**, where a reservation consists of multiple ticket instances rather than a single aggregated structure. The use of an ArrayList enables flexible and ordered storage of tickets. The booking logic is encapsulated within the FlightBookingSystem, while email-related responsibilities are delegated to the EmailService, ensuring clear separation of concerns and high cohesion.

13.4. Design Rationale

ArrayList is selected due to its **dynamic resizing capability**, which allows an arbitrary number of tickets to be added without predefined limits. Tickets are appended sequentially, preserving booking order, which is beneficial for both payment summaries and email presentation. Additionally, constant-time index-based access supports efficient iteration when generating ticket summaries for confirmation emails. This structure aligns naturally with real-world group bookings where passengers share the same flight but hold distinct tickets.

13.5. Time Complexity Analysis

The multiple booking process operates in **O(n)** time, where *n* represents the number of passengers. Each passenger requires the creation of a ticket, which involves constant-time operations such as HashMap insertion for ticket storage and **O(log n)** insertion into the reservation priority queue when applicable.

The sendMultipleTicketsConfirmation() method also executes in **O(n)** time, as it iterates through the list of tickets to generate the email content. No nested iterations are involved, ensuring scalability for group reservations with a higher number of passengers.

14. WEATHER WIDGET

14.1. User Interface Features

The weather widget displays **real-time weather information** relevant to the flight or departure location. It includes a **weather condition icon**, **current temperature**, a short **weather description**, and **warning messages** for adverse conditions such as rain or snow. This feature enhances user experience by providing contextual environmental awareness without affecting the booking workflow.

Figure 14.1.1: *Interface displaying rainy weather conditions for the selected destination*

Figure 14.1.2: *Interface displaying sunny weather conditions for the selected destination*

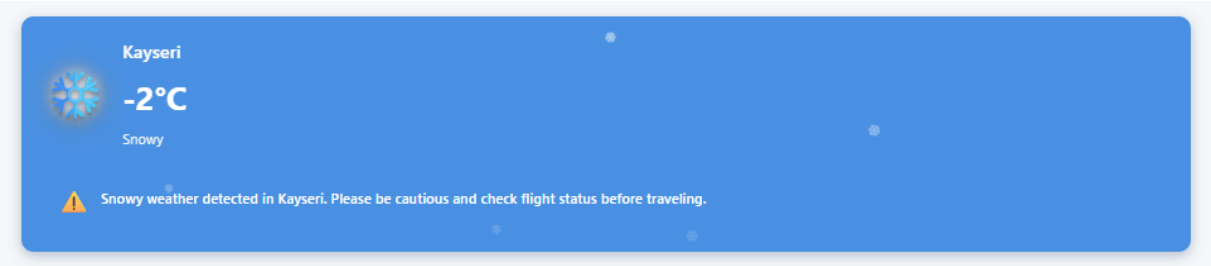

Figure 14.1.3: *Interface displaying snowy weather conditions for the selected destination*

14.2. Backend Data Structure of Weather Widget

This component is implemented as a **frontend-only feature** and does not interact with the backend system. Weather data is retrieved via an **external weather API** using HTTP requests. All processing and rendering occur on the client side.

14.3. Object-Oriented Design

Weather data is represented using a **JavaScript object literal**, which holds attributes such as temperature, condition, icon URL, and alert messages. API integration is handled through asynchronous requests, ensuring non-blocking UI updates. Since no business logic or persistent data is involved, backend classes or data structures are intentionally avoided to reduce system complexity.

14.4. Design Rationale

Keeping the weather widget independent from the backend improves **modularity** and **system performance**. External API integration allows real-time updates without increasing server load. This design choice follows the principle of **separation of concerns**, where auxiliary informational features are decoupled from core booking logic.

15. CAMPAIGNS

15.1. User Interface Features

The campaign management feature allows administrators to define discount campaigns that can be applied during the ticket purchase process. Each campaign consists of a unique campaign code, a discount percentage, a validity period (start and end dates), and an active/passive status.

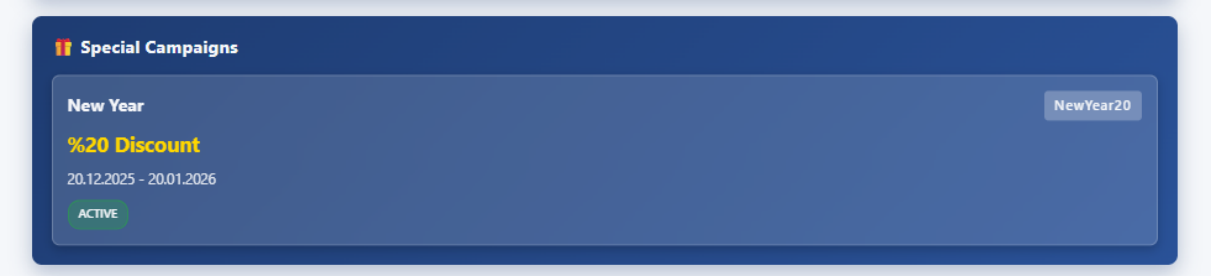

Figure 15.1: *View of the promotions and special offers screen*

15.2. Backend Data Structure of Compaigns

On the backend side, campaigns are stored using a HashMap<String, Campaign> data structure within the AdminPanel class. The campaign code is used as the key, while the corresponding Campaign object is stored as the value. This design enables direct access to campaign information based on the entered campaign code.

From an object-oriented programming perspective, the Campaign entity is implemented as an inner class, ensuring encapsulation and logical grouping within the administration module. The HashMap itself is declared as a private field, preventing unauthorized external modification.

The choice of HashMap provides constant-time access performance. Retrieving a campaign using its code and applying the discount to the ticket price both operate in O(1) time complexity, making this structure highly suitable for real-time pricing operations during the booking flow.

16. ANNOUNCEMENTS

16.1. User Interface Features

The announcements section presents **system-wide messages** such as updates, maintenance notifications, or promotional information. Each announcement includes a **title**, **message content**, and **date information**, displayed in chronological order to ensure clarity and relevance for users.

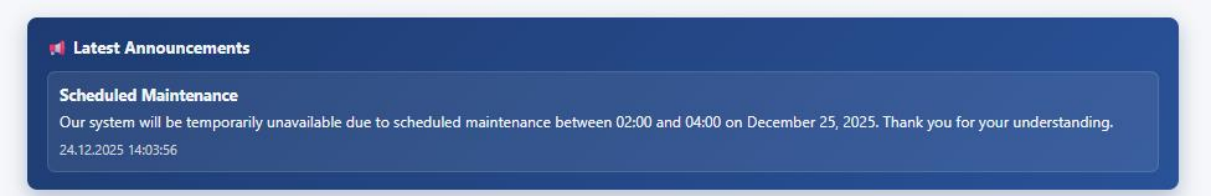

Figure 16.1: *View of the announcements and notifications screen*

16.2. Backend Data Structure of Announcements

Announcements are managed within the AdminPanel.java class using an ArrayList<Announcement>. This structure allows administrators to dynamically add new announcements, which are then retrieved and displayed on the user interface.

```
private List<Announcement> announcements;
this.announcements = new ArrayList<>();
```

16.3. Object-Oriented Design

Each announcement is represented by an Announcement class, implemented as an inner class within the admin panel module. The list of announcements is encapsulated as a private field, ensuring controlled access and modification. This design supports clean separation between administrative functionality and user-facing components.

16.4. Design Rationale

ArrayList is chosen due to its **ordered structure**, which preserves insertion order for chronological display. Adding a new announcement is an **O(1)** operation when appended to the end of the list, making it efficient for frequent updates. The simplicity of the data model aligns with the lightweight nature of announcements, where complex search or prioritization mechanisms are unnecessary.

16.5. Time Complexity Analysis

Adding a new announcement via addAnnouncement() executes in **O(1)** time, as it involves appending an element to the end of the ArrayList. Retrieving announcements using getAnnouncements() is also **O(1)**, since the method returns a reference to the existing list without iteration or additional processing.

17. CHATBOT

17.1. User Interface Features

The chatbot component provides an interactive communication interface for users through chat messages, predefined quick-question buttons, and a text input field. Its primary purpose is to assist users with frequently asked questions and basic navigation guidance.

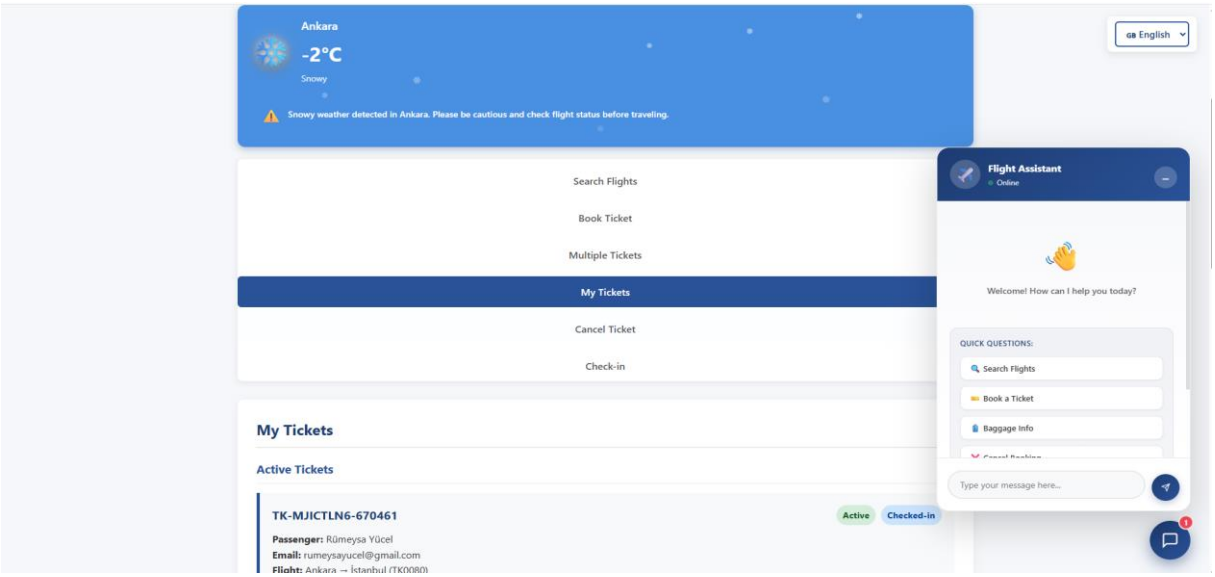

Figure 17.1.1: *View of the AI chatbot assistant on the main dashboard*

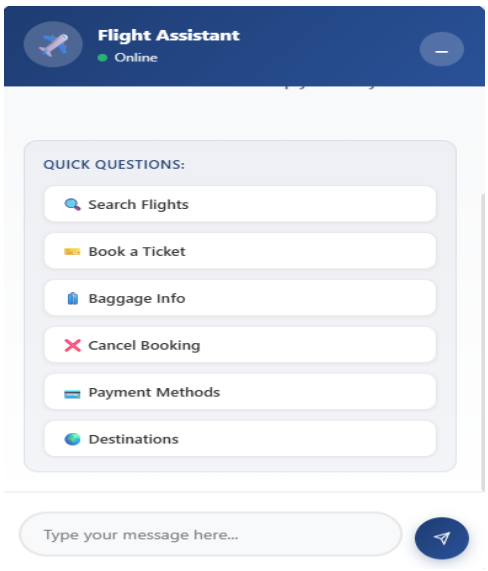

Figure 17.1.2: *View of the quick-reply options and suggested questions within the chatbot interface*

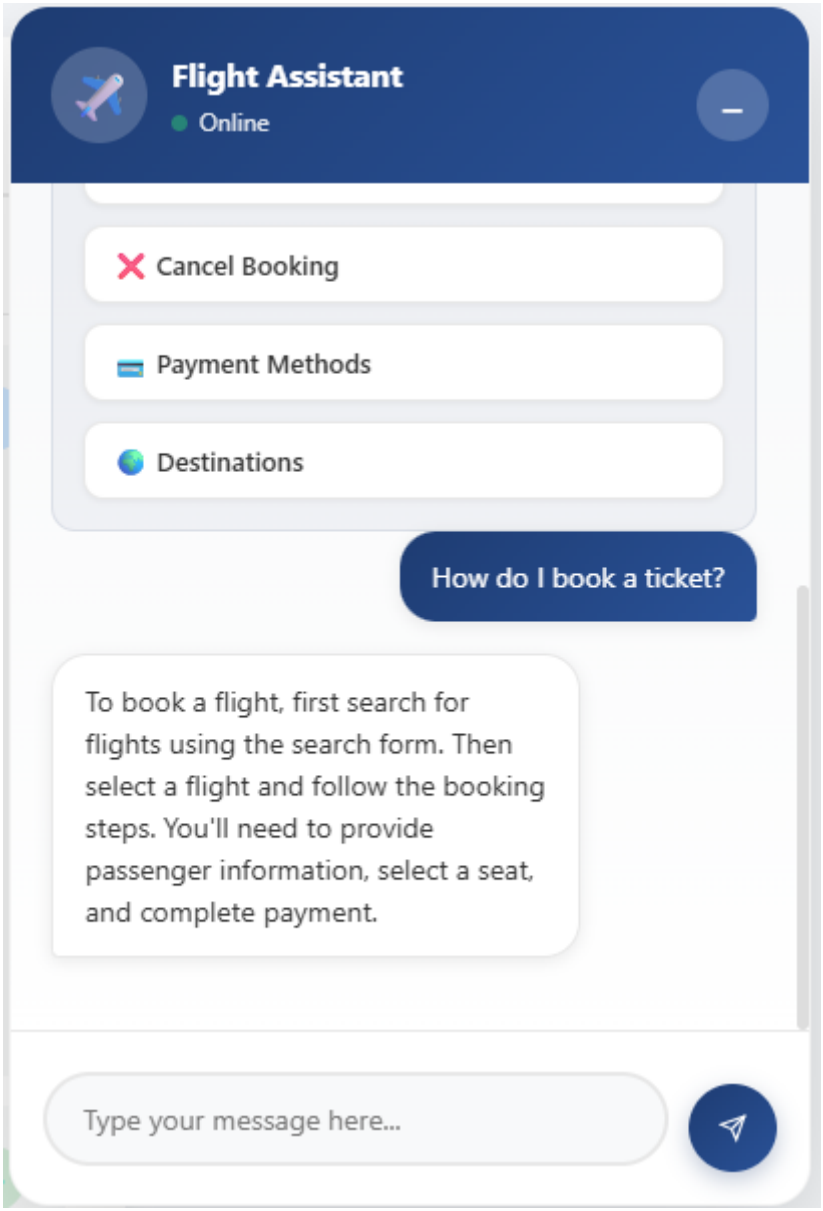

Figure 17.1.3: *Chatbot responding to a custom user query outside the predefined quick-reply options*

17.2. Backend Data Structure of Chatbot

This component is implemented entirely on the frontend side and does not rely on backend data structures. The chatbot operates using pattern matching techniques, where predefined user inputs are matched against known intents and responses. Since no persistent data storage or backend processing is involved, object-oriented backend structures are not required for this feature.

18. ADMIN LOGIN & AUTHENTICATION

18.1. Admin Interface Features

The Admin Login and Authentication module serves as the primary security gateway for the administrative dashboard. The user interface consists of a dedicated login screen that includes a username input field, a password input field, a login button, and an error message display mechanism for failed authentication attempts. In addition, the system implements role-based access control (RBAC), supporting three predefined administrator roles: Super Admin, Operations, and Accounting. These roles determine which administrative sections and functionalities are accessible after successful login.

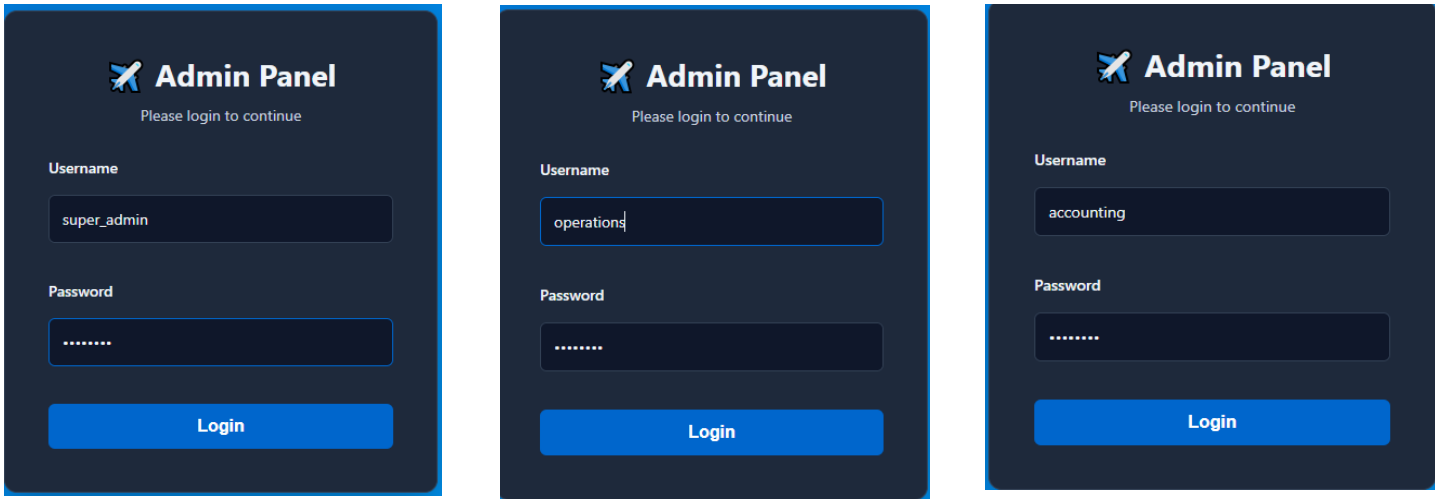

Figure 18.1: *Demonstration of role-based authentication where different administrator roles are identified during the login process.*

18.2. Backend Data Structure of Admin Login & Authentication

18.2.1. Frontend (JavaScript)

File: admin-dashboard-real.js
Structure: ADMIN_CREDENTIALS (Object – HashMap-like)

```
const ADMIN_CREDENTIALS = {
    'super_admin': { password: 'admin123', role: 'super_admin', name: 'Super
Admin' },
    'operations': { password: 'ops123', role: 'operations', name: 'Operations
Manager' },
    'accounting': { password: 'acc123', role: 'accounting', name: 'Accounting
Manager' }
};
```

Key-Value Definition:

- **Key:** username
- **Value:** Credential object { password, role, name }

This structure provides direct key-value access, functioning similarly to a HashMap data structure.

18.2.2. Backend (Java)

18.2.2.1. AdminPanel.java – Admin User List

```
private List<Admin> adminUsers;
this.adminUsers = new ArrayList<>();
Authentication Method:
public boolean authenticateAdmin(String username, String password) {
    Admin admin = adminUsers.stream()
            .filter(a -> a.getUsername().equals(username))
            .findFirst()
            .orElse(null);

    if (admin != null && admin.authenticate(password) && admin.isActive()) {
        this.currentAdmin = admin;
        admin.updateLoginTime();
        return true;
    }
    return false;
}
```

This method authenticates the admin by performing a linear search over the admin user list and validating credentials.

18.2.2.2. Admin.java – Object Class with Enum

```
public class Admin {
    private String username;
    private String password;
    private String email;
    private AdminRole role;
    private Date lastLogin;
    private List<AdminLog> activityLogs;
    private boolean isActive;

    public enum AdminRole {
        SUPER_ADMIN,
        OPERATIONS,
        ACCOUNTING
    }
```

```
    public boolean authenticate(String password) {
        return this.password.equals(password);
    }

    public void updateLoginTime() {
        this.lastLogin = new Date();
        logActivity("LOGIN", "Admin logged in");
    }
}
```

18.3. Object-Oriented Design

18.3.1. Frontend Design

- **Object Literal:** Used as a HashMap-like structure for storing credentials
- **Encapsulation:** All credential-related data is grouped within a single object
- **Session Management:** localStorage is used to persist the authenticated admin session

18.3.2. Backend Design

- **Class:** Admin represents an administrative user
- **Enum:** AdminRole defines fixed and type-safe role values
- **ArrayList:** Stores admin users sequentially
- **Inner Class:** AdminLog maintains activity records
- **Encapsulation:** All class fields are declared as private
- **Composition:** AdminPanel contains and manages Admin objects

18.4. Rationale for Data Structure Choices

18.4.1. Why Use an Object on the Frontend?

- **O(1) Lookup Time:** Direct access using the username ADMIN_CREDENTIALS[username]
- **Key-Value Mapping:** Username mapped directly to credentials
- **Simple Architecture:** No database or backend call required
- **Fast Authentication:** Constant-time validation
- **Static Data:** Admin accounts are predefined and rarely change

18.4.2. Why Use an ArrayList on the Backend?

- **Ordered Storage:** Admin users are stored sequentially
- **Easy Iteration:** Compatible with Java Stream API
- **Low Data Volume**: Number of admin users is typically small
- **Simple Management**: Adding and removing users is straightforward

18.5 Time Complexity Analysis

The time complexity of the admin login and authentication module varies depending on whether the operation is performed on the frontend or the backend, as well as on the data structures employed at each layer.

On the frontend, the authentication process operates in constant time, **O(1).** This efficiency is achieved by storing administrator credentials in a JavaScript object that behaves as a HashMap-like structure. During a login attempt, the system directly accesses the credential object using the provided username as a key. Since property access in JavaScript objects is executed in constant time, both the credential lookup and the subsequent password comparison are completed in **O(1)** time. Furthermore, persisting session data using localStorage also incurs constant-time overhead. As a result, the overall frontend authentication process remains highly efficient and independent of the number of administrator accounts.

On the backend, the authentication process exhibits linear time complexity, **O(n)**, where *n* represents the total number of administrator users stored in the system. The authenticateAdmin method performs a sequential search over an **ArrayList<Admin>** using the Java Stream API. Specifically, the **filter()** operation iterates through the list to locate an administrator whose username matches the input value, while **findFirst()** returns the first matching result. In the worst-case scenario, the entire list must be traversed, leading to **O(n)** time complexity.

Once the corresponding Admin object is found, all subsequent operations are executed in constant time. The **authenticate()** method performs a direct string comparison for password verification, which is considered **O(1)** under typical conditions. Similarly, the **updateLoginTime()** method updates a date field and records the login event by appending a new entry to the activity log. Since appending an element to an ArrayList occurs in amortized constant time, this logging operation does not introduce any additional computational overhead.

Overall, the dominant performance cost of the authentication workflow is the linear search performed during backend user lookup. Given that the number of administrative users is generally small and tightly controlled, this design choice represents a reasonable trade-off between simplicity, code readability, and performance. However, in systems where the number of administrator accounts grows significantly, replacing the ArrayList with a HashMap keyed by username would reduce the backend authentication time complexity from **O(n)** to **O(1),** thereby improving scalability.

19. DASHBOARD (STATISTICS)

19.1. Admin Interface Features

The dashboard interface provides administrators with a comprehensive overview of system performance and operational metrics. Displayed information includes **total reservations**, **active flights**, **occupancy rate**, **daily revenue**, **monthly revenue**, and **cancellation rate**. Additionally, graphical components such as charts are used to visualize trends and statistical distributions, enabling faster decision-making and system monitoring.

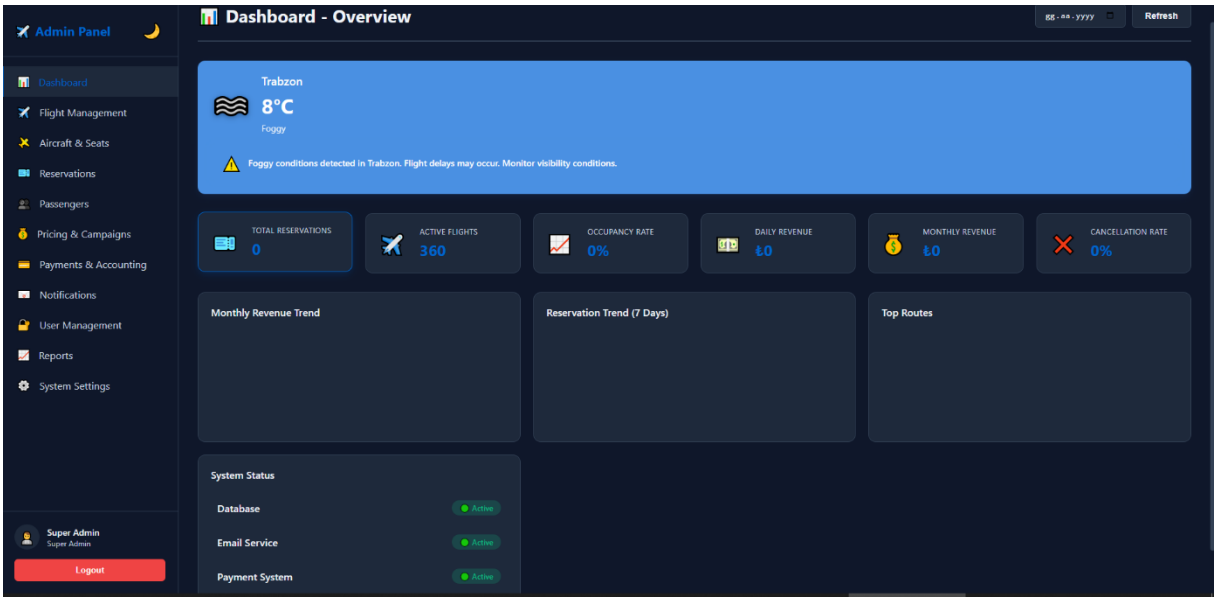

Figure 19.1: *Overview of the administrative dashboard presenting aggregated operational data, including active flights, reservation statistics, revenue metrics, and system health indicators.*

19.2. Backend Data Structure of Dashboard (Statistics)

Dashboard-related statistics are generated within the AdminPanel.java class using a dedicated inner class named DashboardData. This inner class acts as a structured data container that aggregates all calculated statistical values and returns them as a single object to the presentation layer.

```
public DashboardData getDashboardData() {
    DashboardData dashboard = new DashboardData();
    // Statistics calculation
}
```

19.3. Object-Oriented Design

The DashboardData class is implemented as an **inner class** within AdminPanel, ensuring that it is tightly coupled with administrative functionality while remaining hidden from unrelated system modules. All statistical attributes are defined as private fields and accessed through public getter methods, enforcing encapsulation. The dashboard data object is composed of multiple calculated metrics, each derived from existing system entities such as tickets, flights, and reservations.

Java Stream API operations such as **filter(), map(), reduce(),** and **groupingBy()** are extensively used during the calculation phase to process collections in a declarative and readable manner. This approach improves code clarity while supporting functional-style data processing.

19.4. Design Rationale for Inner Class Usage

Using an inner class for dashboard statistics enables logical grouping of closely related data and prevents unnecessary exposure of dashboard-specific structures to other parts of the system. This design improves code organization by keeping administrative reporting concerns encapsulated within the AdminPanel. It also aligns with object-oriented best practices by ensuring cohesion between data and the component responsible for producing it.

19.5. Time Complexity Analysis

The **getDashboardData()** method operates in **O(n)** time complexity, as it iterates over all relevant flights, tickets, and reservations to compute aggregate statistics, where *n* represents the total number of processed data elements. Calculating the occupancy rate similarly requires traversing all flights, resulting in **O(n)** complexity. Revenue-related calculations, such as daily and monthly revenue, rely on Stream API filtering and aggregation operations, which also execute in **O(n)** time due to single-pass iteration over the underlying collections.

20. FLIGHT MANAGEMENT (FLIGHT ADMINISTRATION)

20.1. Admin Interface Features

The flight management module allows administrators to manage all flight-related operations through a centralized control panel. The interface presents flights in a tabular list format and supports actions such as adding new flights, editing existing flight details, deleting flights, searching flights, filtering by flight status, and updating flight statuses (ACTIVE, CANCELLED, DELAYED). This module enables efficient operational control and real-time flight administration.

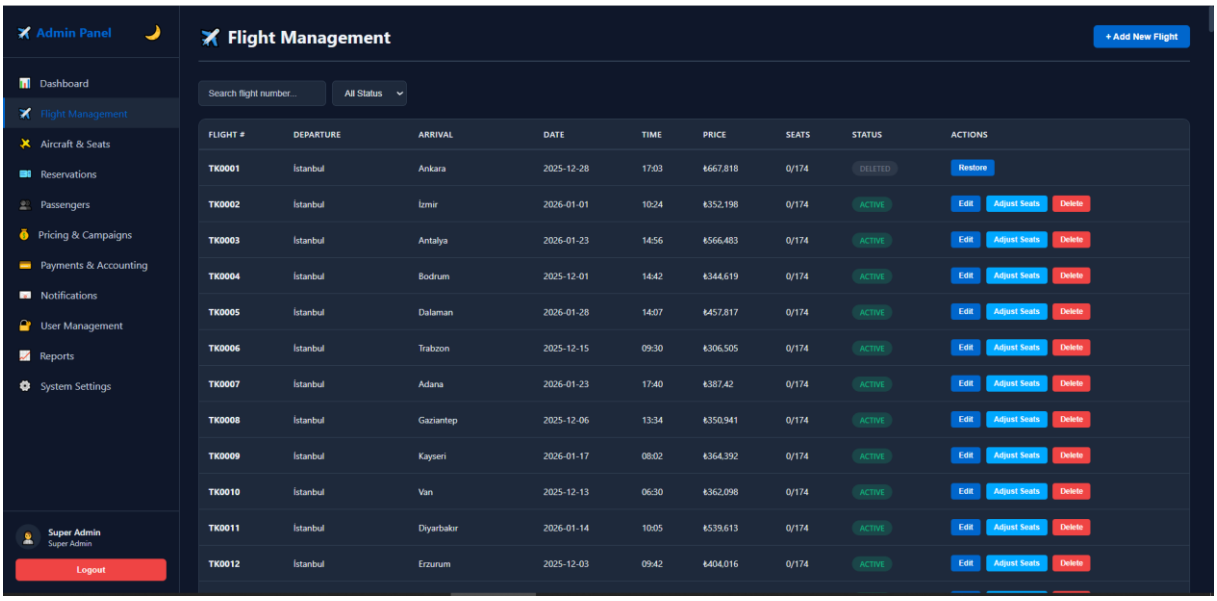

Figure 20.1: *Flight management interface displaying the list of scheduled flights with detailed attributes such as route, date, time, price, seat capacity, status, and administrative actions.*

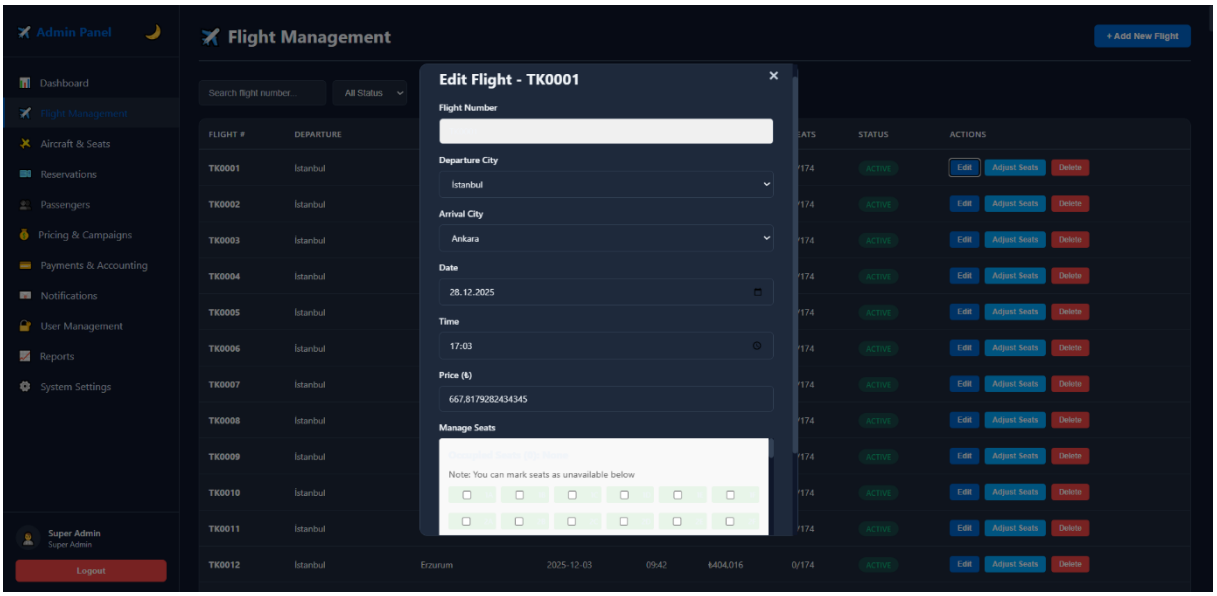

Figure 20.1.2: *Flight editing dialog enabling administrators to modify flight details including route information, schedule, pricing, and seat availability.*

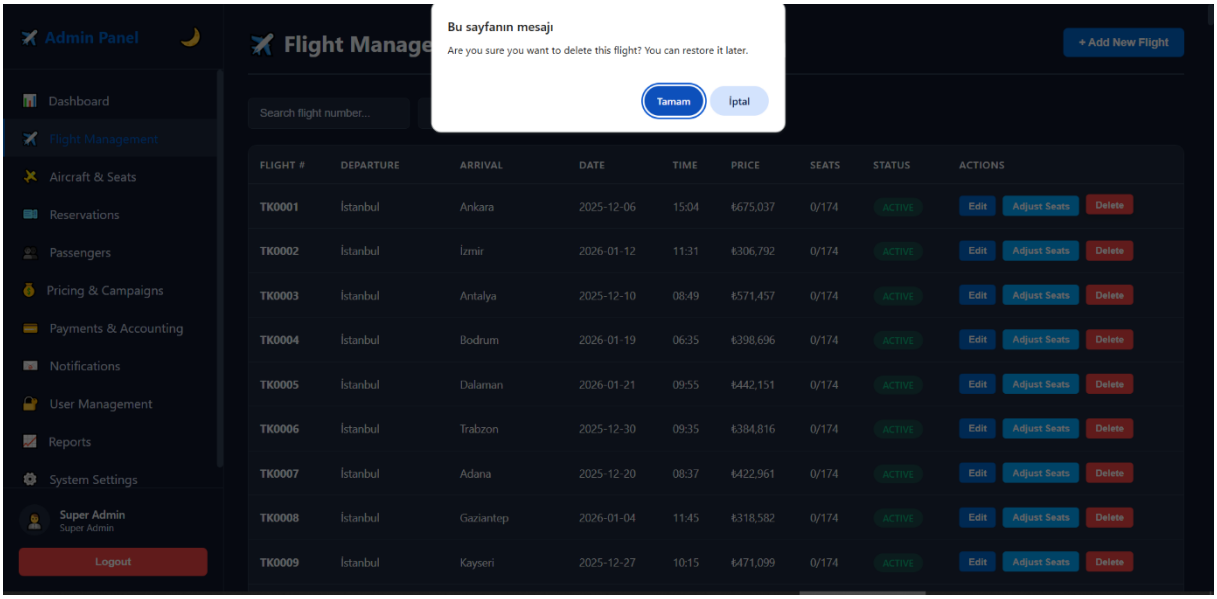

Figure 20.1.3:*User confirmation mechanism applied before executing critical delete operations.*

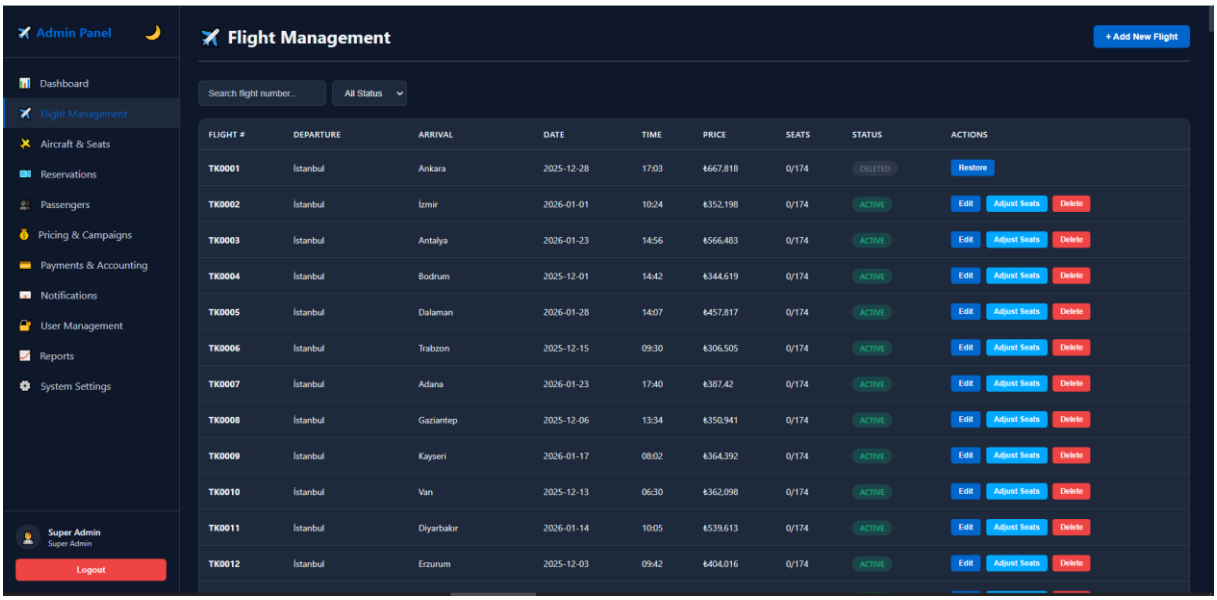

Figure 20.1.4: *Flight management interface demonstrating the soft delete mechanism, where deleted flights are marked inactive and can be restored without permanent data loss.*

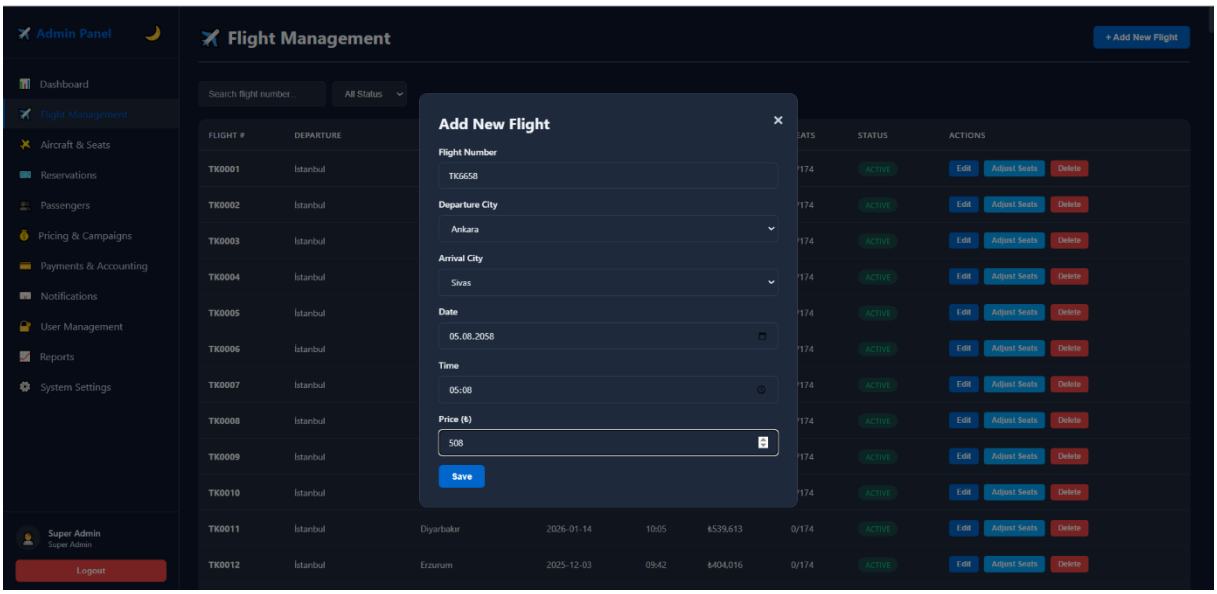

Figure 20.1.5: *Add New Flight interface allowing administrators to create new flight records by specifying route details, schedule information, and pricing parameters.*

20.2. Backend Data Structure of Flight Management (Flight Administration)

All flight records are stored in the FlightBookingSystem.java class using an ArrayList<Flight>. This structure serves as the primary container for flight objects and supports both administrative and user-side flight operations.

```
private List<Flight> flights;
this.flights = new ArrayList<>();

public void addFlight(Flight flight) {
    flights.add(flight);
}
Flight status updates are performed in the AdminPanel.java
class by iterating over the flight list and updating the
status of the matching flight using Stream API operations.
public void updateFlightStatus(String flightNumber, String
status) {
    bookingSystem.getFlights().stream()
        .filter(f ->
f.getFlightNumber().equals(flightNumber))
        .forEach(f -> f.setFlightStatus(status));
}
```

This implementation assumes that the Flight class provides **getFlightStatus()** and **setFlightStatus()** methods. If these methods are not present, they should be implemented to ensure proper encapsulation and status management.

20.3. Object-Oriented Design

The FlightBookingSystem class encapsulates the flight list as a private field, ensuring controlled access to flight data and enforcing the principle of encapsulation. Each flight is modeled as an independent Flight object, demonstrating composition within the booking system. Administrative operations such as adding, editing, deleting, and searching flights are implemented as dedicated methods, which maintains a clear separation between business logic and presentation logic.

The use of Java Stream API (**filter()**, **collect()**) enables declarative data processing and improves code readability. This functional-style approach reduces boilerplate code while maintaining consistency with modern Java development practices.

20.4. Rationale for Using ArrayList

ArrayList is chosen as the underlying data structure due to its suitability for ordered collections and frequent iteration, which are essential requirements for flight management. Since flights are displayed in a table and processed sequentially, preserving insertion order is critical. ArrayList provides **O(1)** index-based access, allowing direct retrieval of flight records when necessary.

Additionally, flight management operations involve frequent traversal for searching, filtering, and status updates. ArrayList offers excellent performance for such sequential access patterns and integrates seamlessly with the Java Stream API. Its dynamic resizing capability simplifies the addition and removal of flights without requiring manual memory management. Compared to alternative structures such as LinkedList or HashMap, ArrayList provides a balanced solution with lower memory overhead and sufficient performance for moderate-sized flight datasets**.**

20.5. Time Complexity Analysis

Adding a new flight to the system executes in **O(1)** time, as the flight is appended to the end of the list. Searching, editing, deleting, and updating flight status operations require iterating over the flight list, resulting in **O(n)** time complexity, where *n* represents the total number of flights. Stream-based filtering and status updates also operate in **O(n)** time due to linear traversal of the list.

20.5.1. Trade-off Analysis

The use of ArrayList in flight management introduces a trade-off between **readability and iteration efficiency** versus **update performance**. Edit and delete operations require linear traversal of the list, resulting in **O(n)** time complexity. However, this cost is acceptable within the context of the system. The total number of flights is typically limited to a moderate range (approximately 100–1000 entries), and administrative edit or delete operations are performed relatively infrequently compared to read and search operations.

Furthermore, the benefits of using ArrayList, such as seamless integration with tabular UI representations and the Java Stream API, outweigh the drawbacks of linear update operations. This design choice prioritizes maintainability, clarity, and efficient sequential access over constant-time updates, which are not critical in this domain.

20.5.2. Time Complexity Analysis (Detailed)

- **addFlight()**: **O(1)** – Appending a flight to the end of the ArrayList executes in amortized constant time.
- **editFlight()**: **O(n)** – A linear search is performed to locate the flight by its flight number, followed by an **O(1)** replacement operation.

```
for (int i = 0; i < flights.size(); i++) {
    if
(flights.get(i).getFlightNumber().equals(flightNumber)) {
        flights.set(i, updatedFlight);  // O(1) set
operation
        break;
    }
}
```

- **deleteFlight()**: **O(n)** – The flight list is traversed to identify the matching flight, and element removal requires shifting subsequent elements.

```
flights.removeIf(f -> f.getFlightNumber().equals(flightNumber));
// removeIf: O(n) search + O(n) shift = O(n)
```

- **searchFlights()**: **O(n)** – All flights are processed using Stream API filtering to match departure and destination criteria.

```
return flights.stream()
    .filter(f ->
f.getDepartureCity().equalsIgnoreCase(departure) &&
                f.getDestinationCity().equalsIgnoreCase(des
tination))
    .collect(Collectors.toList());
```

- **getAllFlights()**: **O(1)** – Returns a reference to the existing flight list without additional processing.

20.6. Real Code Example

```
// FlightBookingSystem.java
private List<Flight> flights;
this.flights = new ArrayList<>();

public void addFlight(Flight flight) {
    flights.add(flight);  // O(1) amortized
}

// AdminPanel.java
public void editFlight(String flightNumber, Flight
updatedFlight) {
    List<Flight> flights = bookingSystem.getFlights();
    for (int i = 0; i < flights.size(); i++) {  // O(n)
iteration
```

```
        if
(flights.get(i).getFlightNumber().equals(flightNumber)) {  //
O(1) comparison
            flights.set(i, updatedFlight);  // O(1) update
            break;
        }
    }
    // Total complexity: O(n) - linear search
}
```

21. AIRCRAFT & SEAT MANAGEMENT

21.1. Admin Interface Features

The aircraft and seat management module allows administrators to manage the airline's aircraft fleet and define seat configurations for each aircraft. The interface presents an **aircraft list**, supports detailed **seat configuration management**, and provides an **Add Aircraft** action for registering new aircraft into the system. This module ensures that flight-seat relationships are accurately defined and maintained.

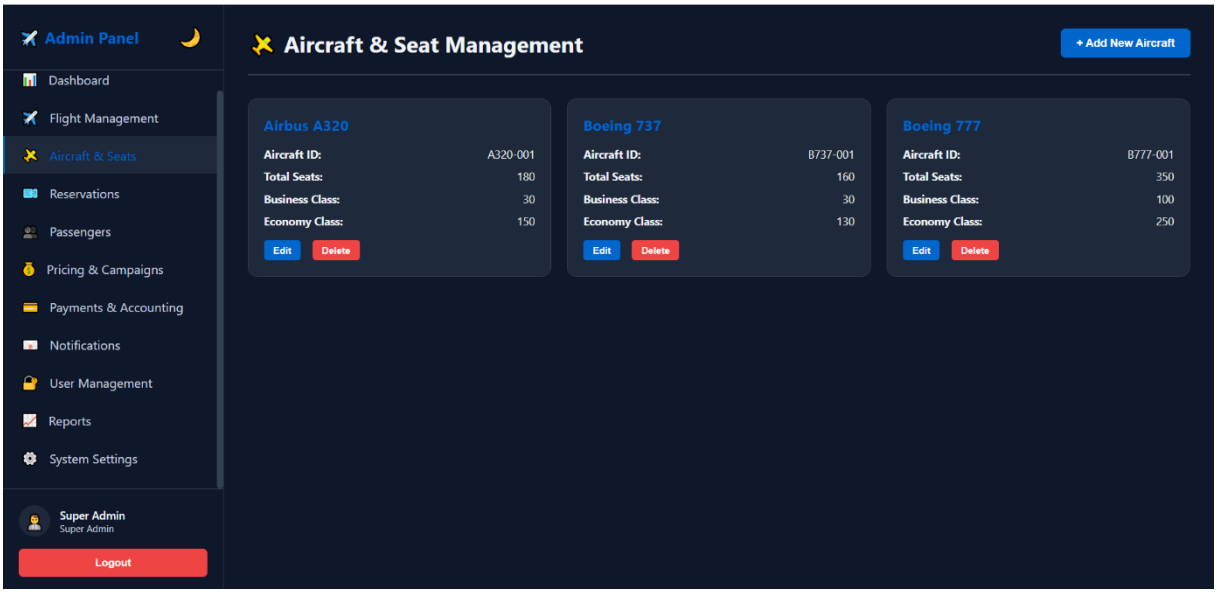

Figure 21.1.1: *Aircraft and seat management interface displaying registered aircraft models along with their total, business class, and economy class seat capacities*

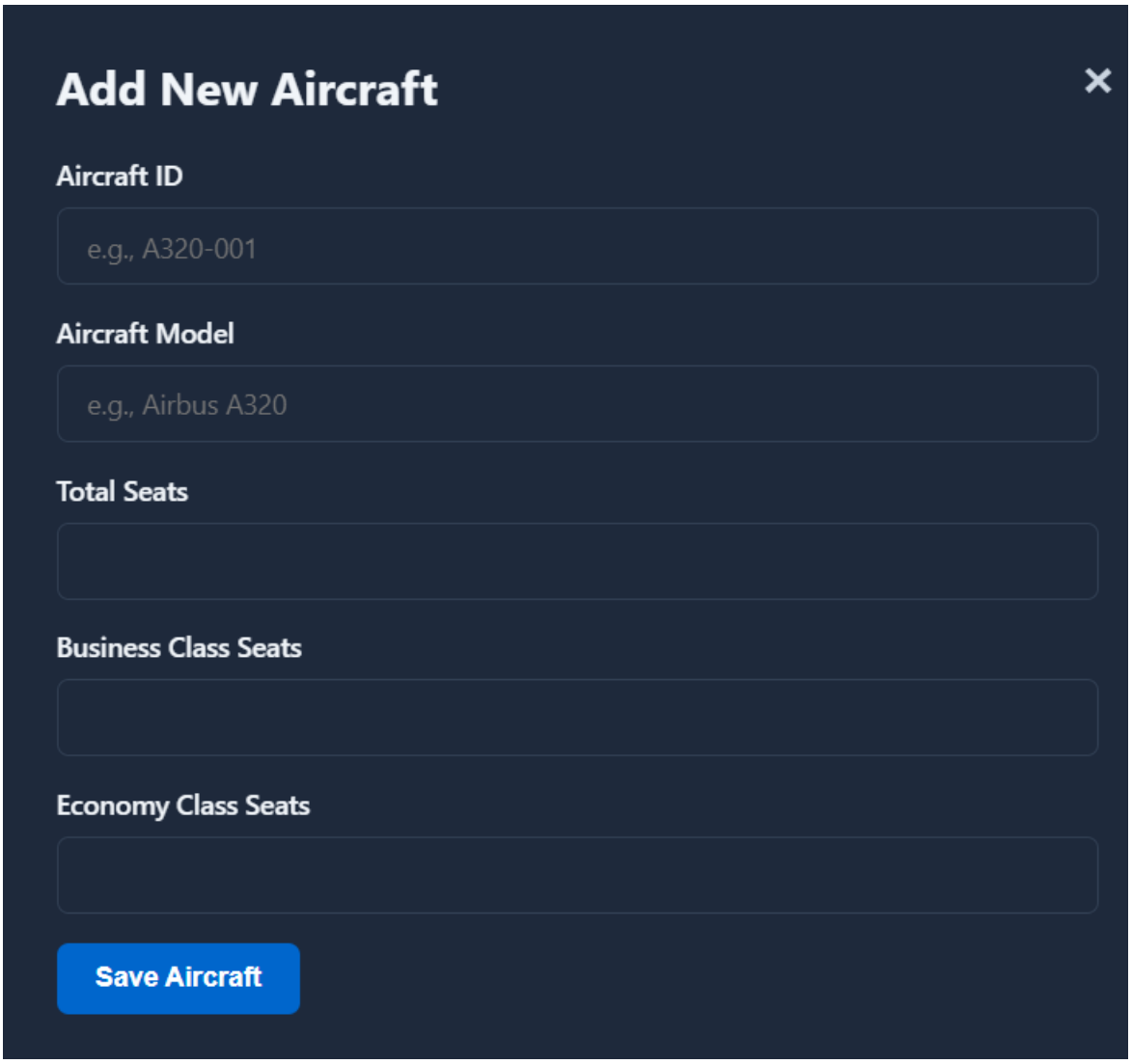

Figure 21.1.2: *Add New Aircraft dialog enabling administrators to define aircraft identifiers, models, and class-based seat distributions.*

21.2. Backend Data Structure of Aircraft & Seat Management

Aircraft data is stored within the AdminPanel.java class using a HashMap<String, Aircraft> named aircraftFleet. Each aircraft is uniquely identified by an aircraft ID, which serves as the key in the map, while the corresponding Aircraft object is stored as the value.

```
private Map<String, Aircraft> aircraftFleet;
this.aircraftFleet = new HashMap<>();
```

21.3. Object-Oriented Design

The Aircraft entity is implemented as an inner class within the AdminPanel, reflecting its exclusive use in administrative contexts. The aircraft fleet is encapsulated as a private map, ensuring controlled access and modification. This design follows the principles of **encapsulation** and **composition**, where the admin panel composes and manages multiple aircraft objects as part of the system configuration.

21.4. Design Rationale for Using HashMap

HashMap is selected as the underlying data structure due to its ability to provide **constant-time access** to aircraft records based on their unique identifiers. Since aircraft IDs such as "A320-001" are known and unique, they naturally serve as efficient keys. This enables administrators to retrieve, update, or configure an aircraft and its seat layout in **O(1)** time without requiring iteration over the entire fleet.

Compared to list-based structures, HashMap eliminates the need for linear searches when accessing aircraft data. This is particularly beneficial as seat configuration operations are frequently performed and must respond instantly to administrative actions.

21.5. Time Complexity Analysis

Retrieving an aircraft using its ID executes in **O(1)** time due to direct HashMap lookup. Adding a new aircraft to the fleet also operates in **O(1)** time, as it involves a single put() operation into the map. These constant-time operations ensure scalability and responsiveness of the aircraft management module even as the fleet size grows.

22. RESERVATION MANAGEMENT

22.1. Admin Interface Features

The reservation management module enables administrators to monitor and control all existing reservations through a structured interface. Reservations are displayed in a **tabular list**, supporting **PNR or reservation code-based search**, **status-based filtering**, and a **Cancel Reservation** action. This module allows fast access to reservation details and efficient handling of cancellation operations.

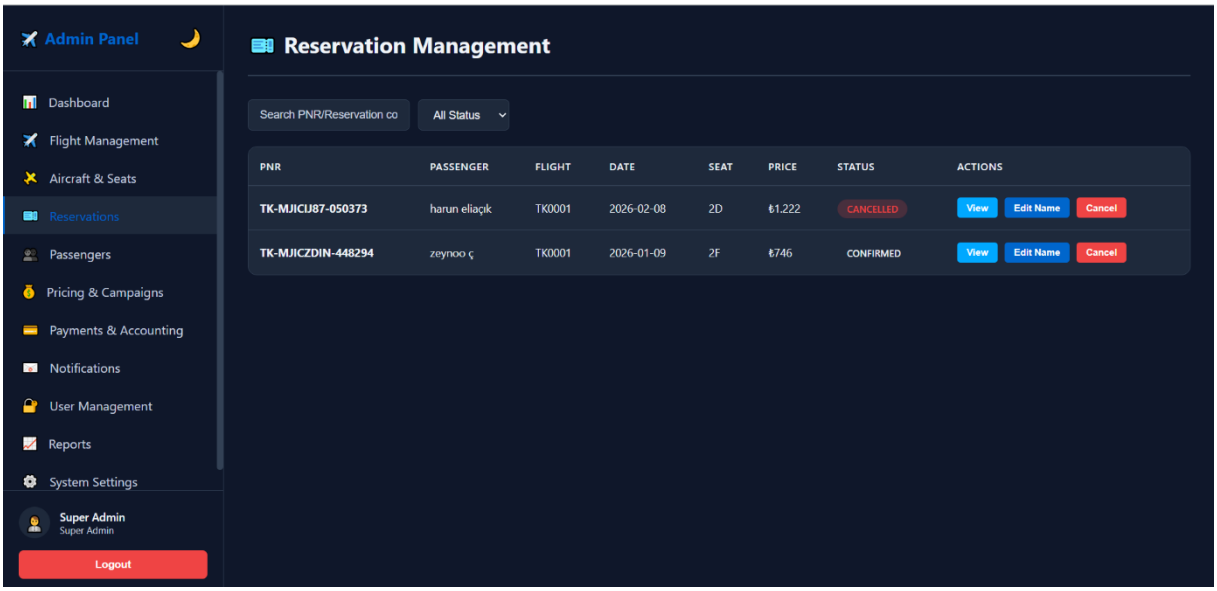

Figure 22.1.1: *Reservation management interface displaying passenger reservations with PNR codes, flight details, seat information, pricing, reservation status, and administrative actions.*

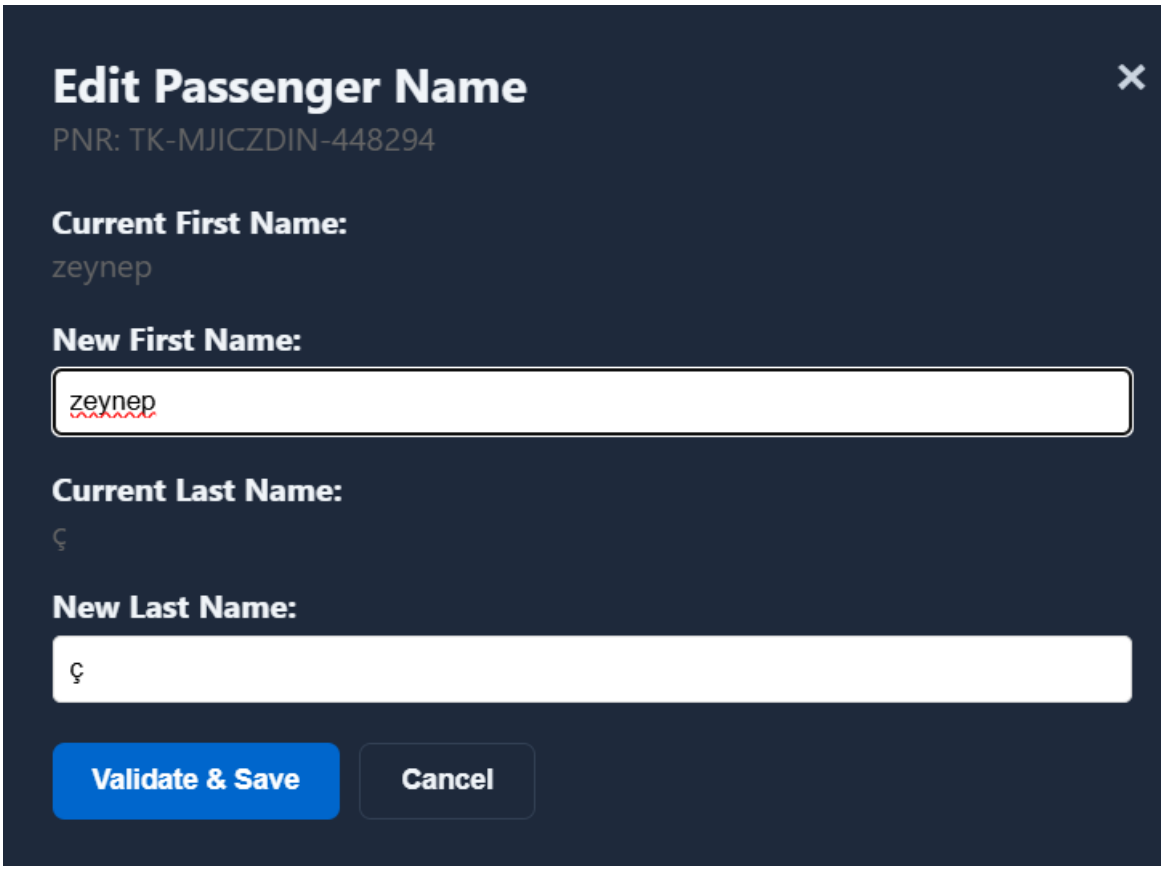

Figure 22.1.2: *Validation mechanism applied during passenger information update to ensure data consistency before saving changes.*

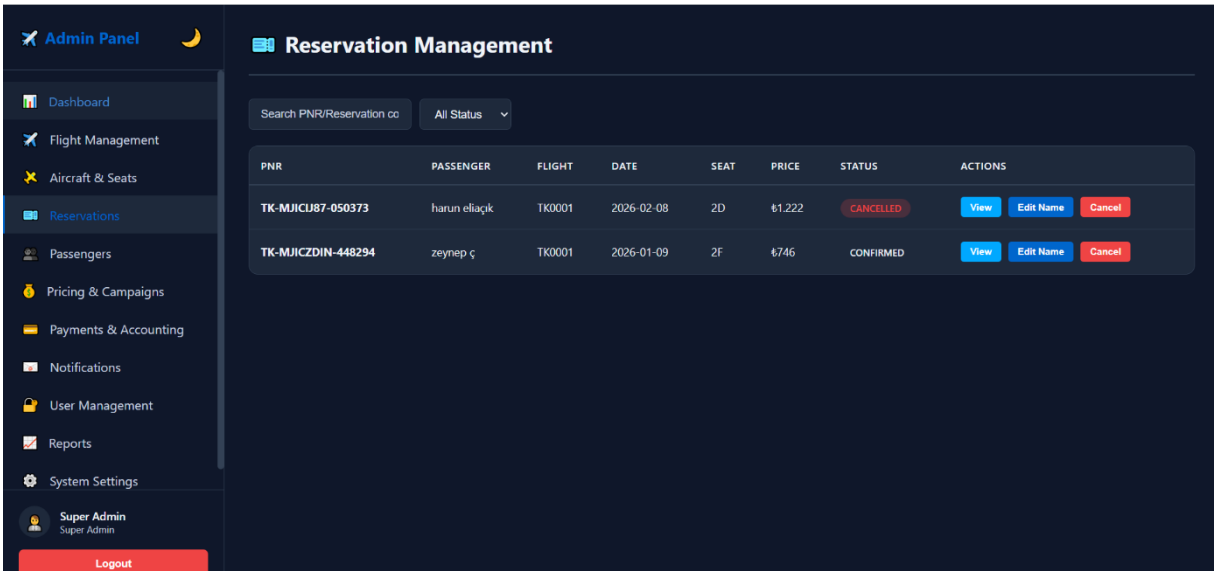

Figure 22.1.3: *Reservation management view illustrating different reservation states, such as confirmed and cancelled bookings*

22.2. Backend Data Structure of Reservation Management

Reservation data is managed within the FlightBookingSystem.java class using two complementary data structures: an **ArrayList<Ticket>** and a **HashMap<String, Ticket>.** While both structures store references to the same Ticket objects, each serves a distinct functional purpose.

```
private List<Ticket> tickets;
private Map<String, Ticket> ticketMap;
```

22.3. Object-Oriented Design

Each reservation is represented by a Ticket object. The booking system composes multiple ticket objects using both list-based and map-based collections. Encapsulation is maintained by declaring both data structures as private fields and exposing controlled access through dedicated methods. This dual-structure approach demonstrates a practical application of composition and data structure optimization within an object-oriented system.

22.4. Rationale for Combined Use of ArrayList and HashMap

The combined use of ArrayList and HashMap addresses different operational requirements of reservation management. The ArrayList preserves insertion order and supports sequential iteration, making it ideal for displaying reservations in a table format within the admin interface. In contrast, the HashMap enables **constant-time access** to reservations using unique reservation identifiers such as PNR codes.

Although this approach introduces a slight increase in memory usage due to duplicated references, it significantly improves system performance for frequent search and cancellation operations. This trade-off favors speed and responsiveness, which are critical in administrative workflows.

22.5. Time Complexity Analysis

Retrieving all reservations through getAllReservations() operates in **O(1)** time, as it returns a reference to the existing reservation list. Searching for a reservation using findReservation() executes in **O(1)** time due to direct HashMap lookup. Cancelling a reservation also runs in **O(1)** time, as it involves a constant-time map lookup followed by a status update or removal operation.

23. PASSENGER MANAGEMENT

23.1. Admin Interface Features

The passenger management module enables administrators to monitor and manage passenger-related information across the system. The interface displays passengers in a **tabular list format**, supports **passenger search**, shows **total flights per passenger**, **registration date**, and allows **editing passenger names** with built-in **name validation mechanisms**. These features ensure both data accuracy and administrative control over passenger records.

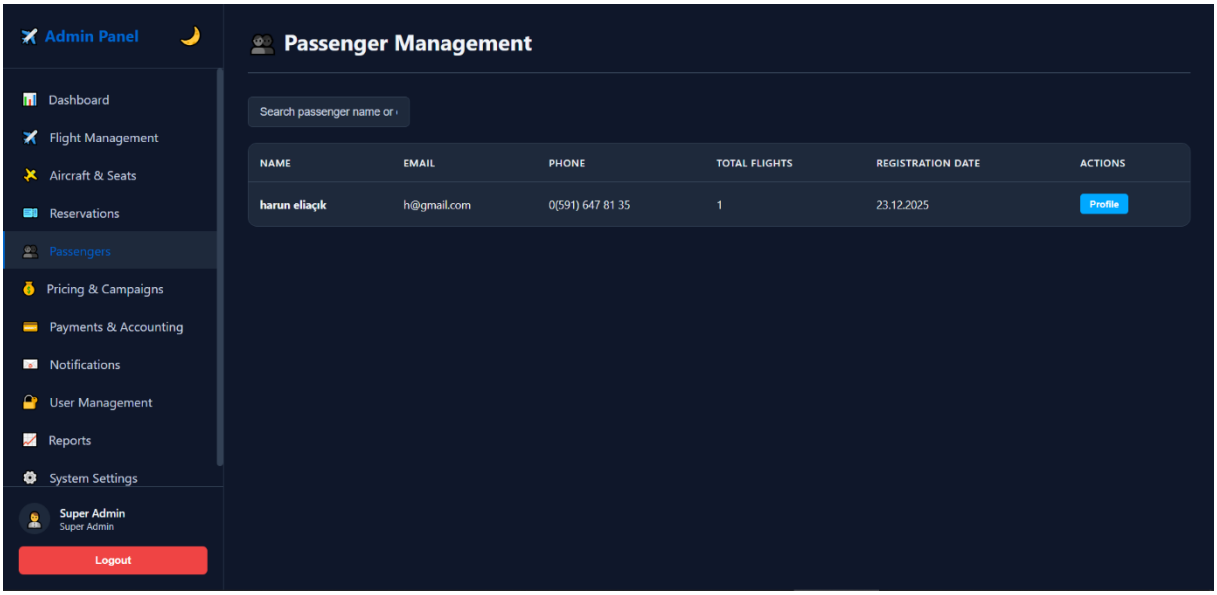

Figure 23.1:*Passenger management interface displaying registered users with contact information, total flight count, registration date, and profile access.*

23.2. Backend Data Structure of Passenger Management

Passenger-related data is derived indirectly from the ticket records stored in the FlightBookingSystem.java class. Passenger information is processed by traversing the **List<Ticket>** collection and grouping tickets based on passenger identifiers using Stream API operations.

```
public List<Passenger> getTopSpendingPassengers(int limit) {
    Map<String, PassengerSpending> passengerSpending = new
HashMap<>();
    // Grouping by passenger ID
}
Name correction and validation logic is isolated in a
dedicated utility class named PassengerNameValidator, which
evaluates name change requests based on predefined rules.
public static String validateNameCorrection(String
oldFirstName, String newFirstName,
```

```
                                                String
oldLastName, String newLastName) {
    // Returns: APPROVE, REJECT, or MANUAL_REVIEW
}
```

23.3. Object-Oriented Design

Passenger management relies on a combination of **HashMap-based grouping**, **inner helper classes**, and **static utility classes**. Tickets are grouped by passenger ID using a HashMap, where each entry represents aggregated passenger-related data. An inner class named PassengerSpending is used to encapsulate calculated metrics such as total spending and flight count per passenger.

The Stream API is utilized for grouping, sorting, and limiting passenger records (groupingBy(), sorted(), limit()), enabling concise and expressive data processing. Name validation logic is intentionally separated into a static utility class to maintain separation of concerns and prevent business logic from polluting core system classes.

23.4. Rationale for Using HashMap and Inner Class

HashMap is selected to efficiently group tickets by passenger ID and enable constant-time access during aggregation operations. This structure allows the system to compute per-passenger metrics such as total spending and number of flights without repeated linear searches.

The PassengerSpending inner class serves as a lightweight data container for intermediate aggregation results. Its use improves code clarity and encapsulates auxiliary computation data without exposing it to unrelated modules. This design balances performance, readability, and encapsulation.

23.5. Name Validation Design

The name validation mechanism is designed to prevent fraudulent or incorrect passenger name modifications while allowing minor corrections. The validator supports **Turkish character normalization**, recognizing variations such as *ü/u*, *ç/c*, and *ı/i*. It also provides **typo tolerance**, allowing up to two character differences between old and new names.

Security considerations are enforced by rejecting name changes that resemble entirely different identities. A **confidence score** mechanism is used to evaluate similarity levels and determine whether a request should be automatically approved, rejected, or flagged for manual review. This approach ensures both flexibility and data integrity.

23.6. Time Complexity Analysis

The getTopSpendingPassengers() method operates in **O(n log n)** time complexity, where *n* represents the number of tickets. This includes **O(n)** grouping by passenger ID followed by **O(n log n)** sorting operations. Retrieving all passengers via getPassengers() executes in **O(n)** time, as it requires traversing the ticket list. The validateNameCorrection() method runs in **O(n)** time relative to the length of the passenger name, as it performs character-by-character similarity comparisons using a Levenshtein-like distance calculation.

24. PRICING & CAMPAIGNS

24.1. Admin Interface Features

The pricing and campaigns module enables administrators to define and manage discount campaigns that can be applied during the ticket booking process. The interface displays a **campaign list** including **campaign codes**, **discount percentages**, and **start/end dates**. Administrators can add new campaigns through the **Add Campaign** action, allowing dynamic pricing strategies and promotional control.

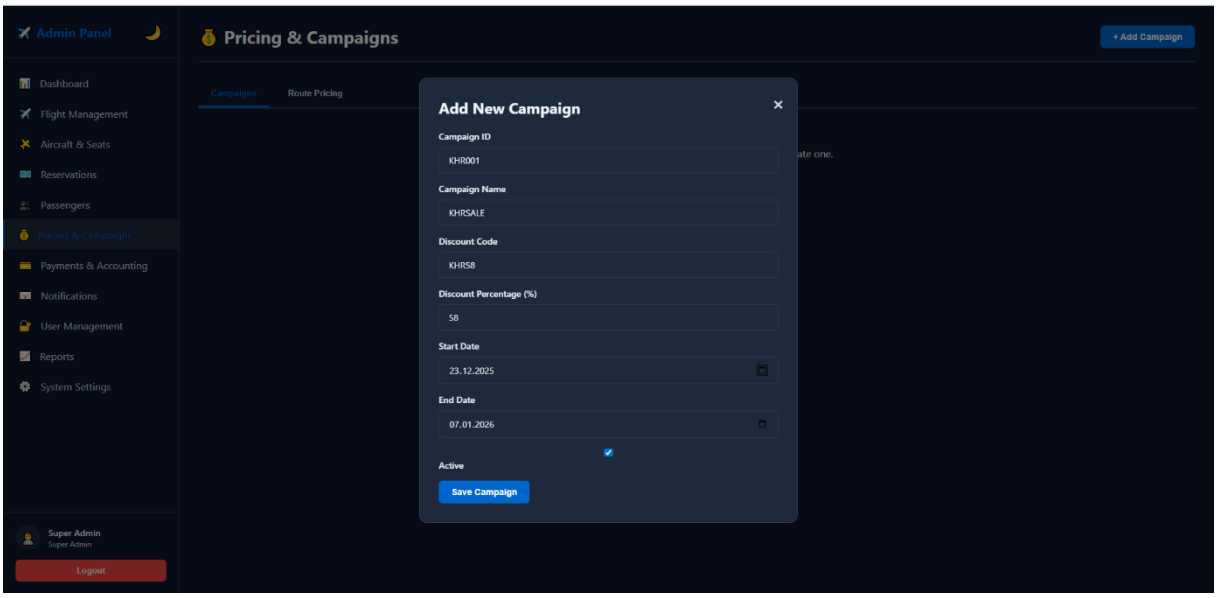

Figure 24.1: *Campaign creation interface allowing administrators to define discount codes, percentage-based discounts, validity periods, and activation status.*

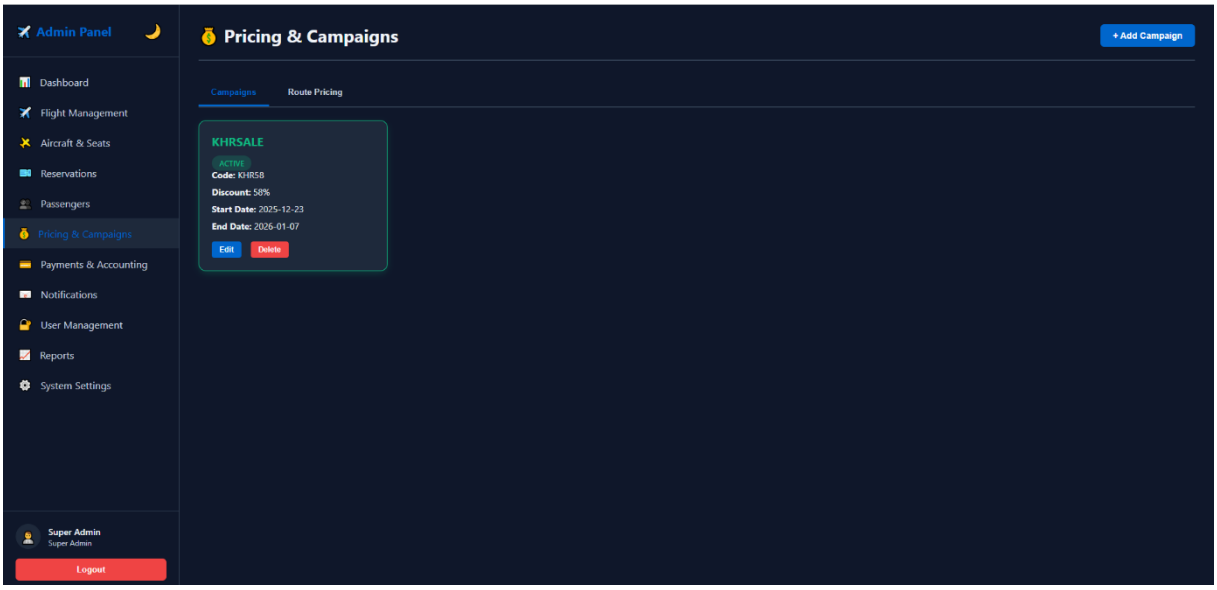

Figure 24.1.2: *Pricing and campaign management view displaying active campaigns with discount details, validity dates, and administrative actions.*

24.2. Backend Data Structure of Pricing & Campaigns

Campaign data is managed within the AdminPanel.java class using a **HashMap<String, Campaign>.** Each campaign is uniquely identified by a campaign code, which serves as the key in the map, while the corresponding Campaign object is stored as the value.

```
private Map<String, Campaign> campaigns;
this.campaigns = new HashMap<>();
```

24.3. Object-Oriented Design

The Campaign entity is implemented as an inner class within the AdminPanel, reflecting its exclusive relevance to administrative pricing operations. The campaign collection is encapsulated as a private map, ensuring that campaign data can only be accessed or modified through controlled administrative methods. This design follows object-oriented principles such as **encapsulation** and **cohesion**, keeping pricing logic isolated within the admin domain.

24.4. Design Rationale for Using HashMap

HashMap is selected as the primary data structure for campaign management due to its ability to provide **constant-time access** based on campaign codes. Since campaign codes are unique and known at the time of application, they serve as ideal keys for direct lookup. This eliminates the need for linear searches and ensures that discount application does not introduce performance overhead during the booking process.

Compared to list-based structures, HashMap significantly improves responsiveness for campaign validation and application. This is particularly important because campaign checks occur in real time during price calculation and payment workflows.

24.5. Time Complexity Analysis

Retrieving a campaign using its code via getCampaign() operates in **O(1)** time due to direct HashMap lookup. Adding a new campaign using addCampaign() also executes in **O(1)** time, as it involves a single put() operation. Applying a campaign discount through applyCampaign() runs in **O(1)** time as well, since it consists of constant-time retrieval followed by a simple arithmetic price calculation.

25. PAYMENTS & ACCOUNTING

25.1. Admin Interface Features

The payments and accounting module provides administrators with a comprehensive view of all financial transactions within the system. The interface displays a **transaction list**, **payment-related statistics**, and supports **date range filtering** to analyze financial activity over specific periods. Additionally, the **Generate Report** functionality enables administrators to produce summarized financial reports for accounting and auditing purposes.

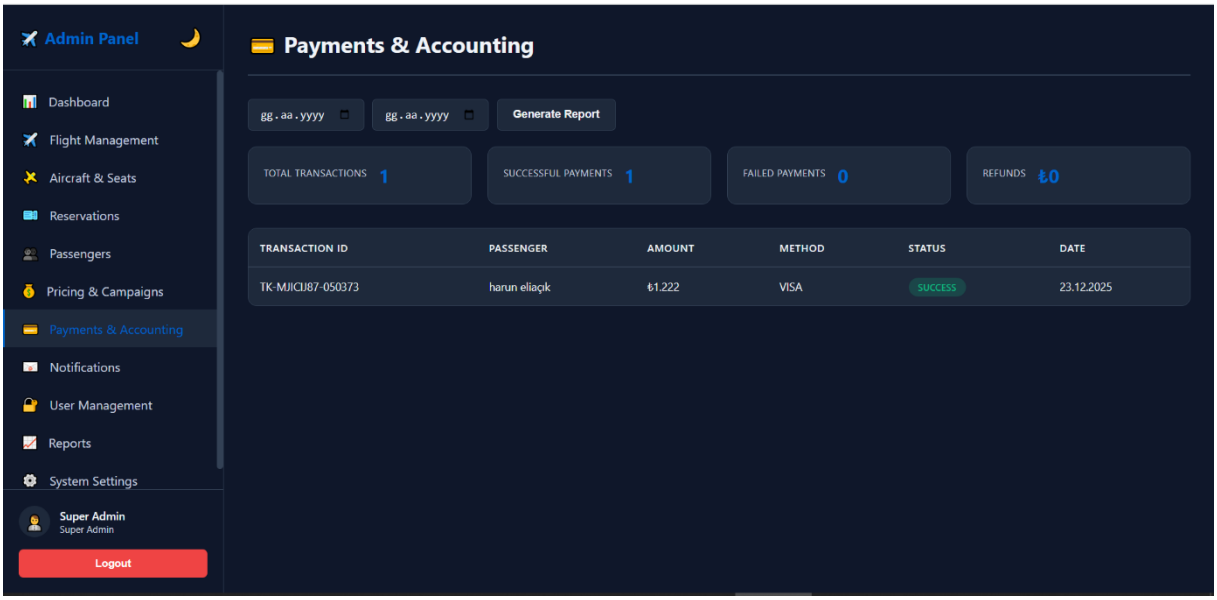

Figure 25.1: *Payments and accounting dashboard presenting transaction summaries, payment success statistics, refund information, and detailed transaction records.*

25.2. Backend Data Structure of Payments & Accounting

Payment and accounting data is managed within the AdminPanel.java class using a **HashMap<String**, **PaymentAccount>**. Each payment account is uniquely identified by an account ID, which serves as the key in the map, while the corresponding PaymentAccount object contains transaction records and aggregated financial data.

```
private Map<String, PaymentAccount> paymentAccounts;
this.paymentAccounts = new HashMap<>();
```

25.3. Object-Oriented Design

The PaymentAccount entity is implemented as an inner class within the AdminPanel, reflecting its exclusive role in administrative financial management. All payment accounts are encapsulated within a private map, ensuring controlled access and modification. The module leverages the Java Stream API to filter transactions by date range and compute financial statistics, enabling expressive and maintainable data processing.

This design follows object-oriented principles such as **encapsulation**, **composition**, and **separation of concerns**, keeping financial logic isolated from unrelated system components.

25.4. Design Rationale for Using HashMap

HashMap is chosen as the underlying data structure due to its ability to provide **constant-time access** to payment accounts based on unique account identifiers. This is particularly important in accounting operations where rapid retrieval of account data is required for reporting and analysis. Using a map-based structure eliminates the need for linear searches and improves system responsiveness.

Compared to list-based alternatives, HashMap offers superior performance for lookup-heavy operations, which are common in financial reporting and transaction management workflows.

25.5. Time Complexity Analysis

Retrieving a payment account using getPaymentAccount() executes in **O(1)** time, as it involves a direct HashMap lookup. Generating financial reports using generateReport() operates in **O(n)** time, where *n* represents the total number of transactions processed. This complexity arises from filtering transactions within a specified date range using Stream API operations, which require a single-pass traversal of the transaction collection.

26. NOTIFICATIONS & ANNOUNCEMENTS

26.1. Admin Interface Features

The notifications and announcements module enables administrators to communicate system-wide messages to users through a structured interface. The module provides a **notification creation form**, supports **recipient type selection**, and includes inputs for **subject** and **message content**. Additionally, the interface displays a **list of sent notifications and announcements**, allowing administrators to review previously delivered messages.

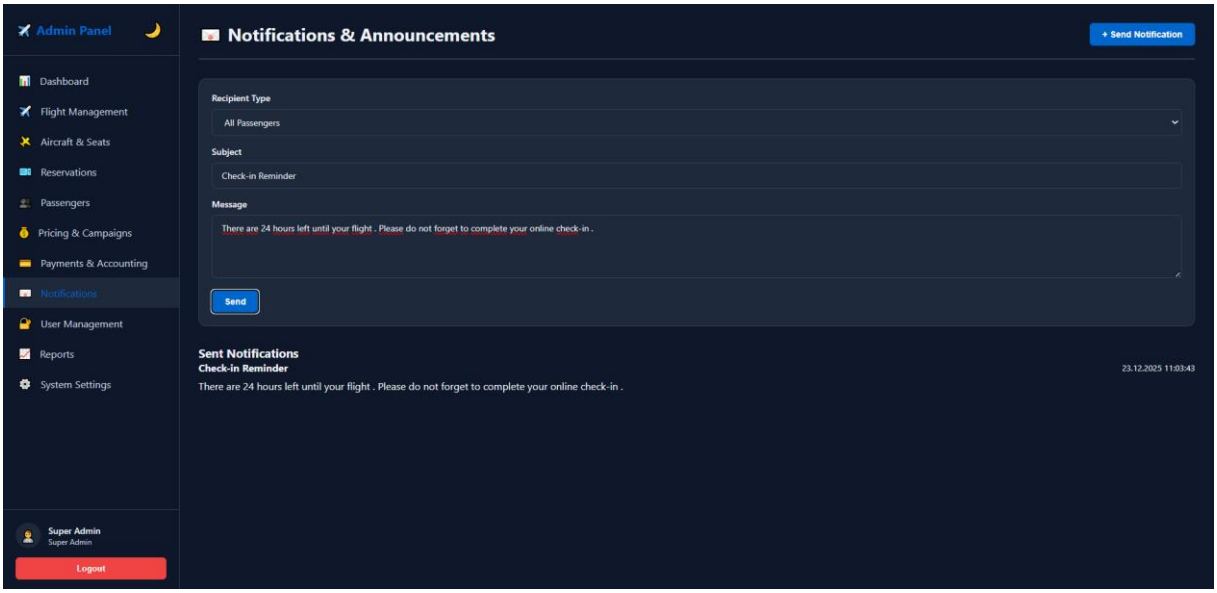

Figure 26.1: *Notification and announcement interface allowing administrators to send system-wide messages to passengers, including reminders and informational alerts.*

26.2 Backend Data Structure of Notifications & Announcements

Notification and announcement data is managed within the AdminPanel.java class using two separate ArrayList collections. One list stores notification records, while the other maintains announcement entries. Each list contains instances of their respective inner classes.

```
private List<Notification> notifications;
private List<Announcement> announcements;
```

26.3. Object-Oriented Design

Both Notification and Announcement are implemented as inner classes within the AdminPanel, reflecting their administrative scope and limited usage context. The collections are encapsulated as private fields, ensuring controlled access and modification through designated administrative methods. This design promotes high cohesion and clear separation between communication-related features and other administrative modules.

26.4. Design Rationale for Using ArrayList

ArrayList is selected due to its suitability for maintaining **ordered collections** of notifications and announcements. Messages are displayed in the order they are created, which aligns naturally with the insertion-order preservation of ArrayList. Adding new notifications or announcements is efficient, as elements are appended to the end of the list in **O(1)** time.

Given that notification and announcement records are primarily appended and read sequentially, list-based storage provides an optimal balance between simplicity, readability, and performance.

26.5. Time Complexity Analysis

Adding a new notification via addNotification() executes in **O(1)** time, as it involves appending an element to the end of the list. Retrieving notifications through getNotifications() also operates in **O(1)** time, since the method returns a reference to the existing collection without additional processing.

27. USER MANAGEMENT (ADMIN ADMINISTRATION)

27.1. Admin Interface Features

The user management module provides administrators with full control over system administrators and their access privileges. The interface displays an **admin list** including **username**, **email**, and **role** information, along with **last login timestamps** and **activity logs**. Administrators can add new admin users, activate or deactivate existing accounts, and monitor administrative actions through detailed log records. This module ensures accountability, traceability, and secure system administration.

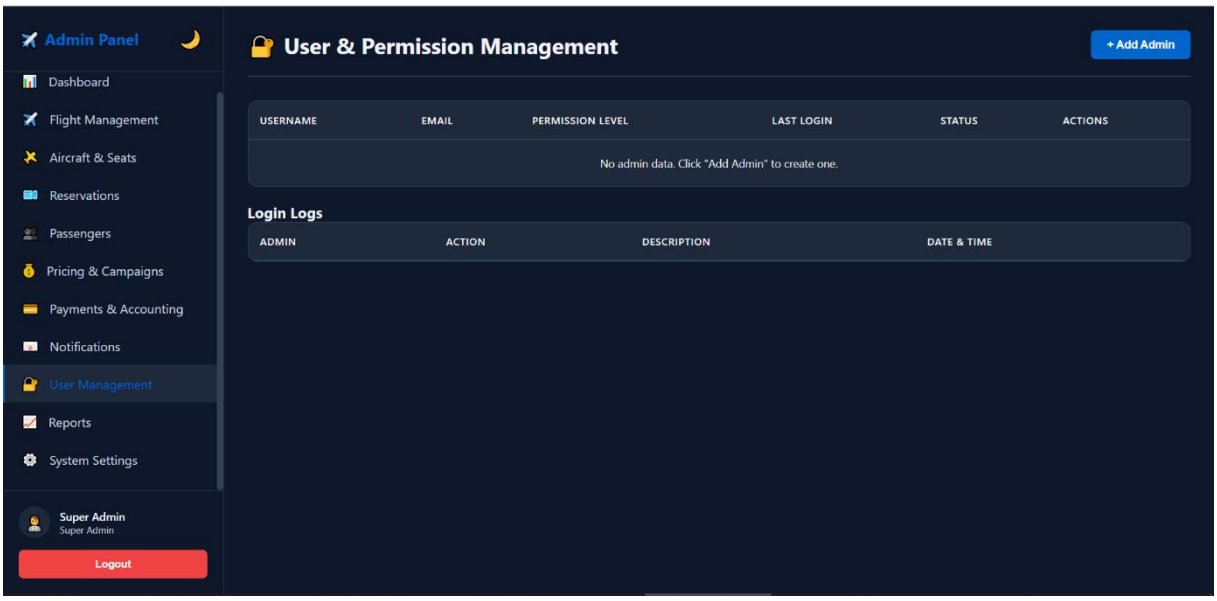

Figure 27.1.1: *User and permission management interface displaying administrative users, permission levels, account status, and login activity logs*

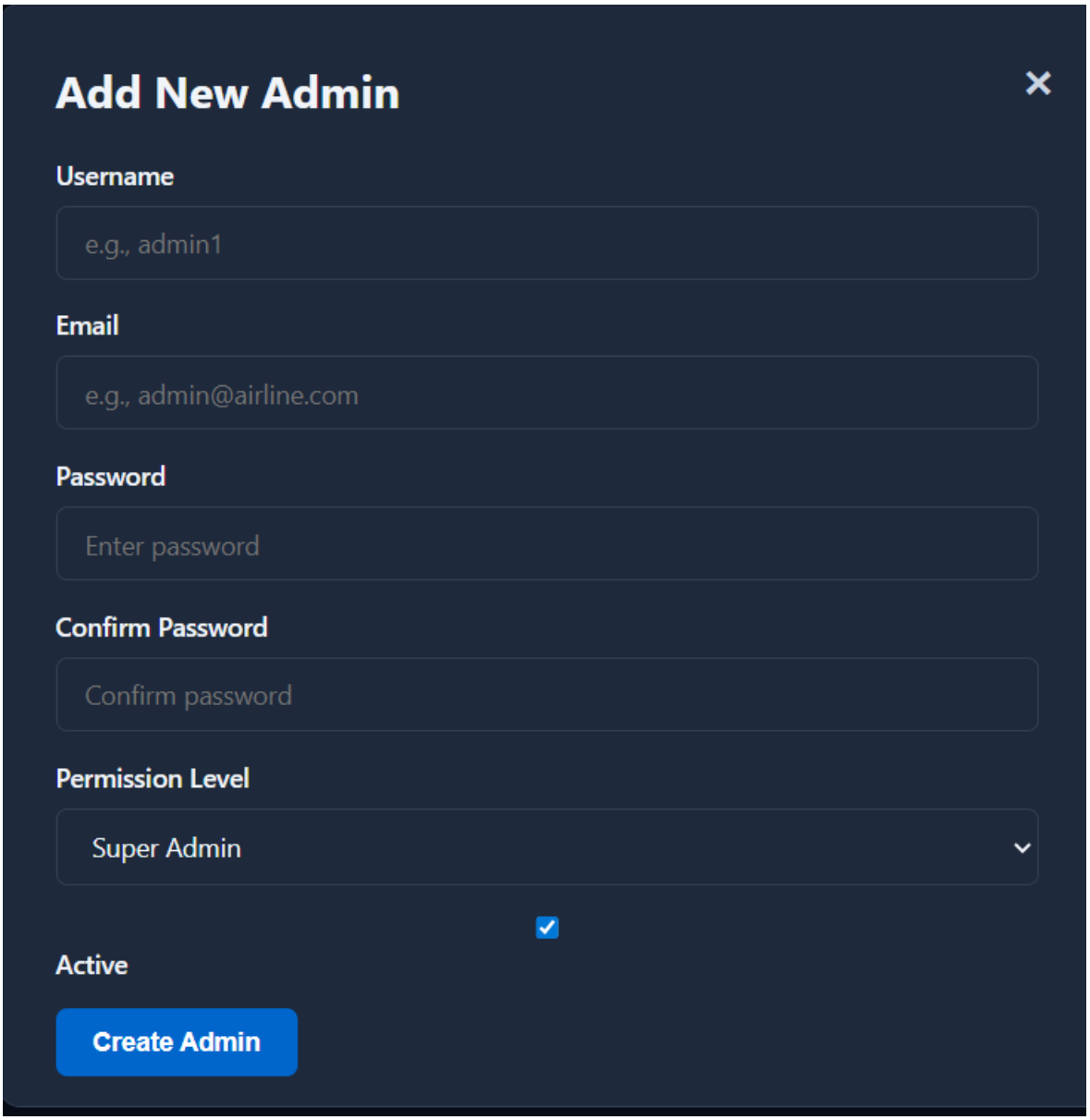

Figure 27.1.2: *Add New Admin dialog enabling the creation of administrative accounts with defined credentials, permission levels, and activation status*

27.2.Backend Data Structure of User Management (Admin Administration)

Administrative user data is modeled using the Admin class, which combines object-oriented constructs such as enums and inner classes to represent roles and activity logs. Each admin user maintains personal credentials, role information, login timestamps, activation status, and a history of performed actions.

```
public class Admin {
    private String username;
    private String password;
    private String email;
    private AdminRole role;
    private Date lastLogin;
    private List<AdminLog> activityLogs;
    private boolean isActive;

    public enum AdminRole {

        SUPER_ADMIN,
        OPERATIONS,
        ACCOUNTING
    }

    public static class AdminLog {
        private Date timestamp;
        private String adminUsername;
        private String action;
        private String description;
    }

    public void logActivity(String action, String
description) {
        AdminLog log = new AdminLog(username, action,
description);
        activityLogs.add(log);
    }
}
```

All admin users are stored within the FlightBookingSystem.java class using an ArrayList<Admin>, which serves as the central repository for administrative accounts.

```
private List<Admin> adminUsers;
this.adminUsers = new ArrayList<>();
```

27.3. Object-Oriented Design

The Admin class encapsulates all administrator-related data as private fields, enforcing strict access control through method-based interaction. Administrative roles are represented using the AdminRole enum, while activity tracking is handled through the AdminLog inner class. The system composes multiple admin objects within the FlightBookingSystem, demonstrating object composition and centralized user management.

The activityLogs field is implemented as an ArrayList, preserving the chronological order of administrative actions. This allows administrators to review actions in the exact sequence in which they occurred.

27.4. Rationale for Using Enum (AdminRole)

The use of an enum for administrative roles ensures **type safety** and prevents invalid role assignments at compile time. By restricting roles to a fixed set of values—*SUPER_ADMIN*, *OPERATIONS*, and *ACCOUNTING*—the system enforces consistent role-based access control. This structure also simplifies permission checks, as role comparisons can be performed directly without string-based validation.

27.5. Rationale for Using Inner Class (AdminLog)

The AdminLog inner class is used to tightly couple activity records with the admin entity they belong to. Since activity logs are never accessed independently of an admin user, encapsulating them within the Admin class improves cohesion and code organization. This design also prevents accidental misuse of log objects outside the administrative context.

27.6. Rationale for Using ArrayList

ArrayList is selected to store both admin users and activity logs due to its ordered nature and efficient append operations. Admin users are typically displayed in a sequential list, while activity logs must be preserved in chronological order. Adding a new admin or logging an activity executes in **O(1)** time, making the structure efficient for frequent insert operations.

27.7 Time Complexity Analysis

The authenticateAdmin() operation executes in **O(n)** time, where *n* represents the total number of admin users, as it requires iterating through the admin list to validate username and password combinations. Adding a new admin user via addAdmin() runs in **O(1)** time, as the admin is appended to the end of the list. Logging an administrative action using logActivity() also executes in **O(1)** time due to constant-time insertion into the activity log list. Updating the last login timestamp with updateLoginTime() operates in **O(1)** time, as it involves a direct field update.

28. REPORTS & ANALYTICS

28.1. Admin Interface Features

The reports and analytics module provides administrators with comprehensive analytical insights into system performance and operational efficiency. The interface includes **revenue reports**, **occupancy reports**, and **cancellation reports**, supported by **graphical visualizations** such as charts. These reports enable data-driven decision-making by presenting aggregated historical data in an interpretable and structured format.

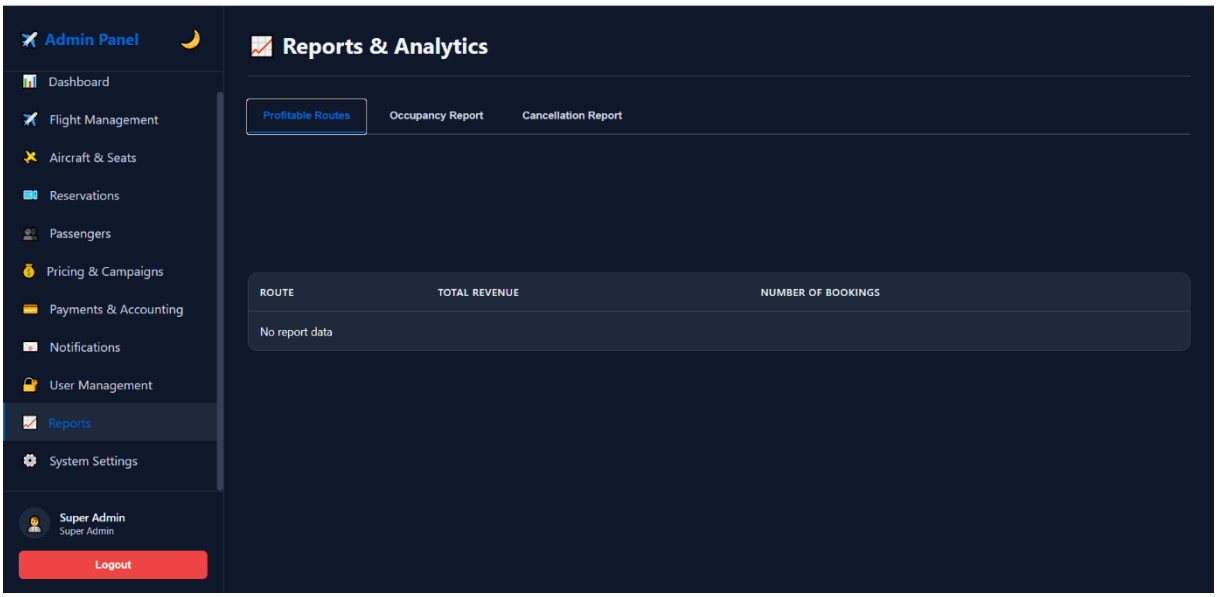

Figure 28.1: *Reports and analytics interface providing summarized operational insights such as route profitability, occupancy analysis, and cancellation statistics.*

28.2. Backend Data Structure of Reports & Analytics

Analytical data is generated within the AdminPanel.java class using a method that returns a **Map<String, Double>** representing monthly revenue values. A LinkedHashMap is used to store aggregated revenue data, where each key corresponds to a specific month and the associated value represents the total revenue for that month.

```
private Map<String, Double> getMonthlyRevenueData() {
    Map<String, Double> monthlyData = new LinkedHashMap<>();
    // Stream groupingBy
}
```

28.3. Object-Oriented Design

The analytics module leverages object-oriented principles by encapsulating report generation logic within the admin panel. A LinkedHashMap is used to maintain ordered analytical data, while Java Stream API operations such as groupingBy() and summingDouble() are employed to aggregate financial metrics efficiently. Additionally, an inner class named RouteData is used to represent route-based statistical information, enabling structured handling of route occupancy and performance metrics.

28.4. Design Rationale for Using LinkedHashMap

LinkedHashMap is chosen over HashMap because it preserves **insertion order**, which is critical when presenting monthly revenue data in chronological sequence. This ensures that months are displayed in the correct order within charts and tabular reports. The key-value structure naturally maps **month → revenue**, providing clarity and consistency in report generation.

Compared to unordered map structures, LinkedHashMap improves the interpretability of analytical outputs without sacrificing lookup efficiency, as it still provides constant-time access on average.

28.5. Time Complexity Analysis

The **getMonthlyRevenueData()** method operates in **O(n)** time complexity, where *n* represents the total number of tickets processed. This complexity arises from grouping tickets by month and summing their revenue values using Stream API operations. The **getTopRoutes()** operation executes in **O(n log n)** time, as it involves grouping route data followed by sorting based on performance metrics such as total revenue or occupancy rate.

29. SYSTEM SETTINGS

29.1. Admin Interface Features

The system settings module allows administrators to configure global application parameters that affect overall system behavior. The interface provides options for **currency selection**, **timezone selection**, **tax rate input**, and a **maintenance mode toggle**. These settings enable administrators to adapt the system to regional requirements, financial regulations, and operational conditions without modifying the application code.

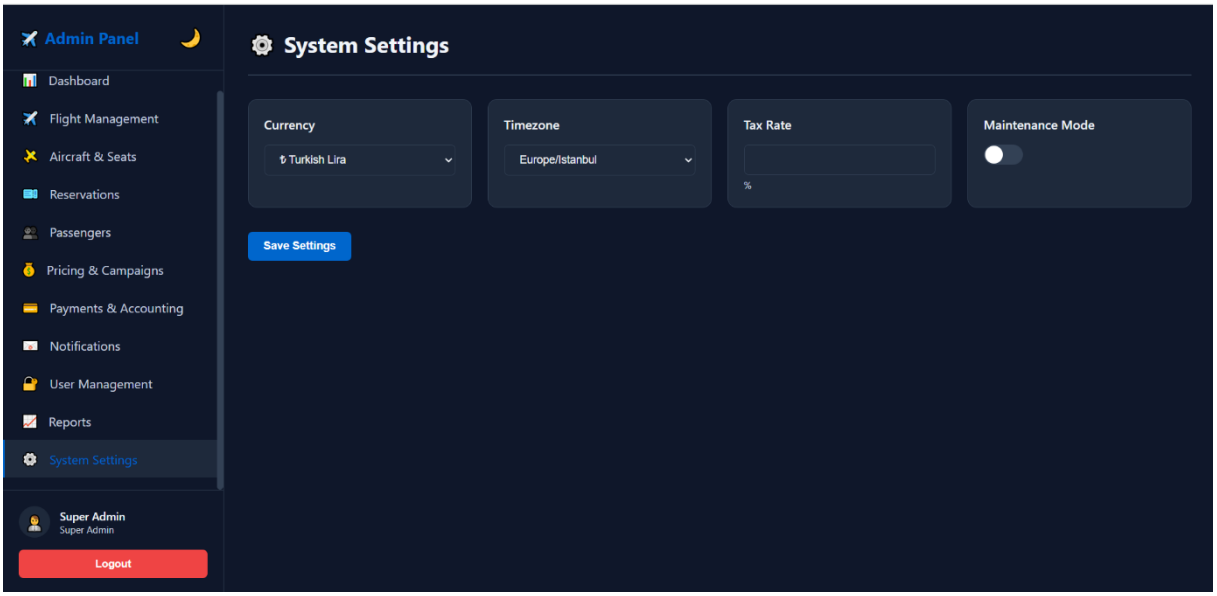

Figure 29.1: *System settings interface allowing administrators to configure global application parameters such as currency, timezone, tax rate, and maintenance mode.*

29.2. Backend Data Structure of System Settings

System configuration data is managed within the **AdminPanel.java** class using a **HashMap<String, SystemSetting>.** Each system setting is identified by a unique setting key, while the corresponding value is stored as a SystemSetting object containing the configuration details.

```
private Map<String, SystemSetting> systemSettings;
this.systemSettings = new HashMap<>();
```

29.3. Object-Oriented Design

The SystemSetting entity is implemented as an inner class within the AdminPanel, reflecting its exclusive role in administrative configuration management. All system settings are encapsulated within a private map, ensuring that configuration values can only be accessed or modified through controlled administrative methods. This design adheres to object-oriented principles such as **encapsulation**, **cohesion**, and **separation of concerns**, keeping system configuration logic isolated from business operations.

29.4. Design Rationale for Using HashMap

HashMap is selected as the underlying data structure due to its ability to provide **constant-time access** to configuration values based on unique setting keys. Keys such as "currency" and "taxRate" naturally represent configuration identifiers and allow direct lookup without requiring iteration over all settings.

This structure is particularly suitable for system settings, as configuration values are frequently read and occasionally updated. Using a map-based approach ensures fast retrieval while maintaining flexibility for adding new settings in the future.

29.5. Time Complexity Analysis

Retrieving a system setting via **getSetting()** executes in **O(1)** time, as it involves a direct HashMap lookup. Updating or adding a setting using **updateSetting()** also operates in **O(1)** time, since it consists of a single **put()** operation. These constant-time operations ensure efficient access to configuration data regardless of the number of defined system settings.

30. REFERENCES

1. Oracle. (2025). Java Platform, Standard Edition Documentation (JDK 17). Oracle Help Center. Available at: https://docs.oracle.com/en/java/javase/17/
2. Mozilla Developer Network (MDN). (2025). JavaScript Guide: Working with Objects and Arrays. MDN Web Docs. Available at: https://developer.mozilla.org/en-US/docs/Web/JavaScript/
3. Stack Overflow. (2024). Best practices for Java Object-Oriented Design and Class Structures. Available at: https://stackoverflow.com/questions/tagged/java+oop
4. W3Schools. (2025). CSS Flexbox and Responsive Web Design Layouts. Available at: https://www.w3schools.com/css/css3_flexbox.asp
5. Apache Maven. (2024). Maven Project Structure and Dependency Management. Apache Software Foundation. Available at: https://maven.apache.org/guides/
6. GeeksforGeeks. (2024). Implementation of Queue and Priority Queue Data Structures in Java. Available at: https://www.geeksforgeeks.org/
7. Mozilla Developer Network (MDN). (2025). HTML Forms and Client-Side Form Validation. MDN Web Docs. Available at: https://developer.mozilla.org/en-US/docs/Web/HTML/Element/form
8. W3Schools. (2025). JavaScript Form Validation and Basic DOM Manipulation. Available at: https://www.w3schools.com/js/js_validation.asp