BIRMINGHAM CITY University

Name:	Bibek Shah/Subresh Thakulla
Student ID:	23189619 / 23189651
Date:	June 23, 2024
Tutor:	Sumanta Silwal
Module:	Object Oriented Programming
Module Code:	CMP5332
CourseWork:	Flight Booking System
Word Count:	2971
Page Count:	26

Contents

Introduction

Our Flight Booking System is a cutting-edge solution designed to transform how flight bookings are managed in the airline industry. For customers, it offers an easy-to-use platform where they can quickly browse available flights, book their tickets with a few clicks, and customize their travel plans to suit their needs. Airline administrators, on the other hand, benefit from a powerful set of tools that make it simple to manage flight schedules, track seat availability, and handle customer bookings with accuracy and efficiency. By integrating all these features into one comprehensive system, our Flight Booking System not only streamlines operations and improves communication between airlines and passengers but also enhances the overall booking experience in the ever-evolving world of aviation.

Purpose and Functionality

The main goal of our Flight Booking System is to make booking flights easier and more efficient for both customers and airline staff. Customers can search for flights by destination, departure date, and other preferences, and once they find the right flight, they can book their tickets securely through the system. This streamlined process saves time and hassle for everyone involved.

For airline administrators, our system offers a range of tools to manage flight schedules, aircraft availability, customer bookings, and other operational tasks. They can add new flights, update current schedules, view passenger lists, and make adjustments as needed to handle changing demands or unexpected events. This helps ensure smooth and efficient airline operations.

Key Features:

1. **Flight Management:** The system keeps a detailed database of available flights, including information like flight numbers, origins, destinations, departure dates, and seat availability. This ensures that all necessary flight details are readily accessible and up to date.

2. **Customer Management:** Customers can create accounts, view their booking history, and manage their reservations easily. Administrators can access customer information as well, allowing them to assist with booking inquiries and provide better support.

3. **Booking Processing:** The system streamlines the booking process by allowing customers to search for flights, choose their preferred options, and make reservations securely. This ensures a smooth and hassle-free experience from start to finish.

Functionality Overview:

- **Add Flight:** This feature empowers administrators to easily add new flights to the system. By entering key details like flight number, origin, destination, departure date, number of seats, and price, airlines can efficiently expand their flight offerings and improve customer accessibility.

- **View Flight:** Users can easily browse through available flights and examine crucial details such as flight number, origin, and destination.

- **Delete Flight:** This functionality allows administrators to effortlessly remove unwanted or discontinued flights from the system. By entering the flight ID or other identifying details, airlines can efficiently manage their flight schedules, ensuring that only relevant and operational flights are displayed to customers.

- **Add Booking:** With the "Add Booking" feature, customers can easily reserve seats on their desired flights. By providing their details and flight preferences, users can secure their bookings seamlessly, enhancing their overall booking experience and ensuring a hassle-free travel journey.

- **Cancel Booking:** The "Cancel Booking" functionality empowers customers to manage their reservations flexibly. Whether due to a change of plans or unforeseen circumstances, users can easily cancel their bookings. This action helps free up seats for other potential passengers and optimizes flight occupancy efficiently.

- **Add Customer:** This feature enables administrators to seamlessly onboard new customers into the system. By capturing essential customer information such as name, contact details, and email address, airlines can efficiently expand their customer base and personalize services according to individual preferences.

- **View Customer:** The "View Customer" functionality gives administrators a comprehensive overview of customer profiles. By accessing details like names, contact information, booking history, and preferences, airlines can personalize services and improve customer satisfaction effectively.

- **Delete Customer:** This functionality enables administrators to effortlessly remove redundant or inactive customer profiles from the system. By streamlining customer databases, airlines can optimize resource allocation and maintain data integrity within the flight booking system.

Main.java

Main.java customers.txt bookings.txt

```java
package bcu.cmp5332.bookingsystem.main;

import bcu.cmp5332.bookingsystem.data.FlightBookingSystemData;

/**
 * The Main class is the entry point for the Flight Booking System application.
 * It loads the flight booking system data, accepts user input, parses commands,
 * and executes them in a loop until the user exits the application.
 *
 * @Author Subresh Thakulla / Bibek Shah
 */
public class Main {

    /**
     * The main method to start the Flight Booking System application.
     *
     * @param args Command line arguments (not used).
     * @throws IOException If an I/O error occurs while loading or storing data.
     * @throws FlightBookingSystemException If an error occurs within the Flight Booking System.
     */
    public static void main(String[] args) throws IOException, FlightBookingSystemException {
        // Load the FlightBookingSystem data
        FlightBookingSystem fbs = FlightBookingSystemData.load();

        // Create a BufferedReader to read input from the console
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        // Display welcome message and instructions
        System.out.println("Flight Booking System");
        System.out.println("Enter 'help' to see a list of available commands.");

        // Main loop for accepting user input and executing commands
        while (true) {
            System.out.print("> ");
            String line = br.readLine();
            // Exit the loop if the user enters 'exit'
            if (line.equals("exit")) {
                break;
            }
            try {
                // Parse and execute the command
                Command command = CommandParser.parse(line, fbs);
                command.execute(fbs);
            } catch (FlightBookingSystemException ex) {
                // Display any error messages
                System.out.println(ex.getMessage());
            }
        }

        // Store the FlightBookingSystem data before exiting
        FlightBookingSystemData.store(fbs);
        // Exit the application
        System.exit(0);
    }
}
```

Description

This Java code serves as the heart of a flight booking system application, facilitating interactions between users and the system via a command-line interface. Initially, it loads existing data from storage, which includes details about available flights, bookings, and other pertinent information. Users interact with the system by entering commands through the console, where they receive clear prompts for input. The system continuously reads and processes these commands, executing corresponding actions within the flight booking system. These actions might involve booking or canceling flights, retrieving flight details, or performing administrative tasks. Exception handling is integrated to manage errors that may arise during command execution, ensuring the system's reliability. The loop persists until the user opts to exit using the "exit" command, triggering the saving of any data modifications back to storage. Additionally, comprehensive javadoc documentation is generated for all command classes to enhance code clarity and maintainability.

Output

```
Flight Booking System
Enter 'help' to see a list of available commands.
> help
Commands:
        listflights                             print all flights
        listcustomers                           print all customers
        addflight                               add a new flight
        addcustomer                             add a new customer
        showflight [flight id]                  show flight details
        showcustomer [customer id]              show customer details
        addbooking [customer id] [flight id]    add a new booking
        cancelbooking [customer id] [flight id] cancel a booking
        editbooking [booking id] [flight id]    update a booking
        loadgui                                 loads the GUI version of the app
        help                                    prints this help message
        exit                                    exits the program
>
```

Commands

1. AddCustomer

```java
/**
 * Command to add a new customer to the flight booking system.
 */
package bcu.cmp5332.bookingsystem.commands;

import java.io.BufferedWriter;

/**
 * Command to add a new customer to the flight booking system.
 */
public class AddCustomer implements Command {

    private final String name; // Name of the customer
    private final String phone; // Phone number of the customer
    private final String email; // Email address of the customer

    /**
     * Constructs an AddCustomer command with the specified name, phone, and email.
     *
     * @param name  the name of the customer
     * @param phone the phone number of the customer
     * @param email the email address of the customer
     */
    public AddCustomer(String name, String phone, String email) {
        this.name = name;
        this.phone = phone;
        this.email = email;
    }

    /**
     * Executes the command to add a new customer to the flight booking system.
     *
     * @param fbs The flight booking system.
     * @throws FlightBookingSystemException If an error occurs while executing the command.
     */
    @Override
    public void execute(FlightBookingSystem fbs) throws FlightBookingSystemException {
        // Get the maximum customer ID currently in the system
        int maxId = 0;
        if (!fbs.getCustomers().isEmpty()) {
            maxId = fbs.getCustomers().stream().mapToInt(Customer::getId).max().orElse(0);
        }

        // Create a new customer object with the next available ID
        Customer customer = new Customer(++maxId, name, phone, email);
        fbs.addCustomer(customer); // Add the customer to the flight booking system
        System.out.println("Customer #" + customer.getId() + " added.");

        // Write customer data to a file
        try (BufferedWriter writer = new BufferedWriter(new FileWriter("resources/data/customers.txt", true))) {
            writer.write(customer.getId() + "," + customer.getName() + "," + customer.getPhone() + "," + customer.getEmail());
            writer.newLine();
        } catch (IOException e) {
            throw new FlightBookingSystemException("Error writing to customers.txt: " + e.getMessage());
        }
    }
}
```

Description

This Java code defines a command class called AddCustomer, implementing the Command interface to handle adding new customers to a flight booking system. It contains instance variables for the customer's name, phone number, and email, initialized during object creation. The execute method, which overrides the Command interface, manages the process of adding the customer. It begins by validating the customer's name to ensure it does not contain digits, raising an exception if it does. Assuming validation passes, the method proceeds to fetch the next available customer ID from the system. Using the provided details, including the email, a new Customer object is then created and added to the system. Finally, a confirmation message is printed, detailing the successful addition of the customer along with their assigned ID, ensuring clarity and transparency in system operations.

Output

```
Console
Main (7) [Java Application] C:\Program Files\Java\jdk-21\bin\javaw.exe  (Jun 2
Flight Booking System
Enter 'help' to see a list of available commands.
> addcustomer
Customer Name: Bibek Shah
Customer Phone: 9821685754
Customer Email: bikkishah57@gmail.com
Customer #5 added.
> viewcustomer
Invalid command.
> addcustomer
Customer Name: Subrace Thakulla
Customer Phone: 9845678934
Customer Email: subrace@gmail.com
Customer #6 added.
>
```

2. ListCustomers

```java
 * Command to list all customers in the flight booking system.
package bcu.cmp5332.bookingsystem.commands;

import bcu.cmp5332.bookingsystem.main.FlightBookingSystemException;

/**
 * Command to list all customers in the flight booking system.
 */
public class ListCustomers implements Command {

    /**
     * Executes the command to list all customers in the flight booking system.
     *
     * @param flightBookingSystem the flight booking system
     * @throws FlightBookingSystemException if an error occurs while executing the command
     */
    @Override
    public void execute(FlightBookingSystem flightBookingSystem) throws FlightBookingSystemException {
        List<Customer> customers = readCustomersFromFile("resources/data/customers.txt");
        for (Customer customer : customers) {
            System.out.println(customer.getDetailsShort());
        }
        System.out.println(customers.size() + " customer(s)");
    }

    /**
     * Reads the list of customers from the specified file.
     *
     * @param filename the name of the file containing customer data
     * @return a list of customers
     * @throws FlightBookingSystemException if an error occurs while reading the file
     */
    private List<Customer> readCustomersFromFile(String filename) throws FlightBookingSystemException {
        List<Customer> customers = new ArrayList<>();
        try (BufferedReader reader = new BufferedReader(new FileReader(filename))) {
            String line;
            while ((line = reader.readLine()) != null) {
                String[] parts = line.split(",");
                int id = Integer.parseInt(parts[0]);
                String name = parts[1];
                String phone = parts[2];
                String email = parts[3];
                Customer customer = new Customer(id, name, phone, email);
                customers.add(customer);
            }
        } catch (IOException | NumberFormatException e) {
            throw new FlightBookingSystemException("Error reading customers from file: " + e.getMessage());
        }
        return customers;
    }
}
```

Description

This Java code defines a command class called ListCustomers, which implements the Command interface to handle the task of listing all customers stored in the flight booking system. The main objective of this class is to execute a command that retrieves customer data from a specific file named customers.txt. The execute method, overriding the Command interface, is responsible for reading the data from the file using a BufferedReader to process each line individually. It parses each line to extract essential customer details such as ID, name, phone number, and optionally their email if it's included. The method ensures thorough handling by checking the length of split parts to determine if an email exists in the line. Any IOException encountered during the file reading process is caught and triggers the throwing of a FlightBookingSystemException, which provides a clear error message detailing the issue with reading the customer data file. Ultimately, this command facilitates the systematic listing of customers stored in the flight booking system, aiding administrators in managing and monitoring customer information effectively.

Output

```
Customer #5 - Bibek Shah - 9821685754 - bikkishah57@gmail.com
Customer #6 - Subrace Thakulla - 9845678934 - subrace@gmail.com
6 customer(s)
>
```

3. ShowCustomer

```java
* Command to display details of a specific customer in the flight booking system.
package bcu.cmp5332.bookingsystem.commands;

import bcu.cmp5332.bookingsystem.main.FlightBookingSystemException;

/**
 * Command to display details of a specific customer in the flight booking system.
 */
public class ShowCustomer implements Command {

    private final int customerId;

    /**
     * Constructs a new ShowCustomer command with the specified customer ID.
     *
     * @param customerId the ID of the customer to display details for
     */
    public ShowCustomer(int customerId) {
        this.customerId = customerId;
    }

    /**
     * Executes the command to display details of the specified customer.
     *
     * @param fbs the flight booking system
     * @throws FlightBookingSystemException if there is an error displaying customer details
     */
    @Override
    public void execute(FlightBookingSystem fbs) throws FlightBookingSystemException {
        Customer customer = fbs.getCustomerByID(customerId);
        if (customer == null) {
            throw new FlightBookingSystemException("Customer with ID " + customerId + " not found.");
        }

        System.out.println("Customer ID: " + customer.getId());
        System.out.println("Name: " + customer.getName());
        System.out.println("Phone: " + customer.getPhone());
        System.out.println("Email: " + customer.getEmail());

        List<Booking> bookings = customer.getActiveBookings(); // Get only active bookings
        if (bookings.isEmpty()) {
            System.out.println("This customer has not made any bookings.");
        } else {
            System.out.println("Bookings:");
            for (Booking booking : bookings) {
                Flight flight = booking.getFlight();
                System.out.println("Booking ID: " + booking.getId());
                System.out.println("Flight Number: " + flight.getFlightNumber());
                System.out.println("Origin: " + flight.getOrigin());
                System.out.println("Destination: " + flight.getDestination());
                System.out.println("Date: " + flight.getDepartureDate().format(DateTimeFormatter.ofPattern("yyyy-MM-dd")));
                System.out.println("Price: " + booking.getPrice());
                System.out.println();
            }
        }
    }
}
```

Description

This Java class, `ShowCustomer`, is designed as a command within a flight booking system application. Its primary function is to provide comprehensive information about a particular customer stored in the system. This includes displaying the customer's unique ID, their full name, contact information, and details of any bookings they have made. For customers with bookings, the command lists each booking with specifics such as the flight number, origin, destination, date, and price. In cases where a customer has not made any bookings, the output clearly indicates this absence. Overall, this command serves to assist users in viewing detailed customer profiles and their associated booking information within the flight booking system.

Output

```
Main (7) [Java Application] C:\Program Files\Java\jdk-21\bin\javaw.ex
Flight Booking System
Enter 'help' to see a list of available commands.
> showcustomer 6
Customer ID: 6
Name: Subrace Thakulla
Phone: 9845678934
Email: subrace@gmail.com
This customer has not made any bookings.
>
```

4. ShowFlight

```java
 * Command to display details of a specific flight in the flight booking system.
package bcu.cmp5332.bookingsystem.commands;

import bcu.cmp5332.bookingsystem.main.FlightBookingSystemException;

/**
 * Command to display details of a specific flight in the flight booking system.
 */
public class ShowFlight implements Command {

    private final int flightId;

    /**
     * Constructs a new ShowFlight command with the specified flight ID.
     *
     * @param flightId the ID of the flight to display details for
     */
    public ShowFlight(int flightId) {
        this.flightId = flightId;
    }

    /**
     * Executes the command to display details of the specified flight.
     *
     * @param fbs the flight booking system
     * @throws FlightBookingSystemException if there is an error displaying flight details
     */
    @Override
    public void execute(FlightBookingSystem fbs) throws FlightBookingSystemException {
        Flight flight = fbs.getFlightByID(flightId);
        if (flight == null) {
            throw new FlightBookingSystemException("Flight with ID " + flightId + " not found.");
        }

        System.out.println("Flight Number: " + flight.getFlightNumber());
        System.out.println("Origin: " + flight.getOrigin());
        System.out.println("Destination: " + flight.getDestination());
        System.out.println("Departure Date: " + flight.getDepartureDate());
        System.out.println("Number of Seats: " + flight.getNumberOfSeats());
        System.out.println("Price: " + flight.getPrice());

        List<Customer> passengers = flight.getPassengers();
        if (passengers.isEmpty()) {
            System.out.println("No passengers booked for this flight.");
        } else {
            System.out.println("Passengers:");
            for (Customer passenger : passengers) {
                System.out.println("Name: " + passenger.getName());
                System.out.println("Phone Number: " + passenger.getPhone());
                System.out.println();
            }
        }
    }
}
```

Description

This Java class, `ShowFlight`, functions as a command within a flight booking system application. Its role is to present comprehensive details about a chosen flight. This includes displaying essential information such as the flight's unique ID, departure origin, destination, departure date, total number of available seats, and ticket price. Moreover, the command lists all passengers booked for the flight, providing their names and contact phone numbers. In instances where no passengers are booked for the flight, the output clearly states this absence. Overall, this command is designed to aid users in viewing detailed flight information and understanding the passenger composition associated with each flight stored in the system.

Output

```
> showflight 7
Flight Number: 1
Origin: kathmandu
Destination: dheli
Departure Date: 2024-06-23
Number of Seats: 10
Price: 20000.0
No passengers booked for this flight.
>
```

5. AddBooking

Main.java | customers.txt | bookings.txt | AddBooking.java | MainWindow.java

```java
* Class description goes here.
package bcu.cmp5332.bookingsystem.commands;

import bcu.cmp5332.bookingsystem.main.FlightBookingSystemException;


public class AddBooking implements Command {

    private final int customerId; // ID of the customer
    private final int flightId; // ID of the flight


    public AddBooking(int customerId, int flightId, LocalDate localDate) {
        this.customerId = customerId;
        this.flightId = flightId;
    }


    @Override
    public void execute(FlightBookingSystem fbs) throws FlightBookingSystemException {
        LocalDate today = LocalDate.now();
        LocalDate twoYearsFromToday = today.plusYears(2);

        if (today.isAfter(twoYearsFromToday)) {
            throw new FlightBookingSystemException("Bookings more than 2 years in advance are not allowed.");
        }

        Customer customer = fbs.getCustomerByID(customerId);
        if (customer == null) {
            throw new FlightBookingSystemException("Customer with ID " + customerId + " not found.");
        }

        Flight flight = fbs.getFlightByID(flightId);
        if (flight == null) {
            throw new FlightBookingSystemException("Flight with ID " + flightId + " not found.");
        }

        if (!flight.hasNotDeparted(today)) {
            throw new FlightBookingSystemException("Cannot book a flight that has already departed.");
        }

        if (flight.getPassengers().size() >= flight.getNumberOfSeats()) {
            throw new FlightBookingSystemException("The flight is full. Booking cannot be made.");
        }

        int price = flight.calculatePrice(today);

        int bookingId = fbs.generateBookingId();
        LocalDate bookingDate = LocalDate.now();
        Booking booking = new Booking(bookingId, customer, flight, bookingDate, price);
        customer.addBooking(booking);
        flight.addPassenger(customer);

        if (!booking.isCancelled()) {
            try (BufferedWriter writer = new BufferedWriter(new FileWriter("resources/data/bookings.txt", true))) {
                writer.write(booking.getId() + "," + customer.getId() + "," + flight.getId() + "," + booking.getBookingDate() + "," + booking.getPrice());
                writer.newLine();
            } catch (IOException e) {
                throw new FlightBookingSystemException("Error writing to bookings.txt: " + e.getMessage());
            }
        }

        System.out.println("Booking was issued successfully to the customer.");
    }
}
```

Description

The `AddBooking` class in a flight booking system acts as a command that facilitates the creation of bookings for customers on specific flights. It ensures that bookings are made within a two-year window from the current date and verifies seat availability for the chosen flight. Once these validations are completed, it calculates the booking cost, generates a unique booking ID, and creates a new booking object. This object is then linked to both the customer and the flight, updating their respective records accordingly. Additionally, the command ensures that all booking details are saved to a file for future reference, ensuring data persistence. Overall, this command simplifies the process of adding bookings to the system while ensuring data accuracy and providing confirmation of successful booking creation.

Output

```
Console
Main (7) [Java Application] C:\Program Files\Java\jdk-21\bin\javaw.ex
Flight Booking System
Enter 'help' to see a list of available commands.
> addbooking 6 1
Customer ID: 6
Flight ID: 1
```

6. AddFlight

```java
/* Command to add a new flight to the flight booking system.[]
package bcu.cmp5332.bookingsystem.commands;

import bcu.cmp5332.bookingsystem.main.FlightBookingSystemException;[]

/**
 * Command to add a new flight to the flight booking system.
 */
public class AddFlight implements Command {

    private final String flightNumber; // Flight number
    private final String origin; // Origin airport
    private final String destination; // Destination airport
    private final LocalDate departureDate; // Departure date
    private final int numberOfSeats; // Number of seats available on the flight
    private final int price; // Price of the flight

    /**
     * Constructs an AddFlight command with the specified flight details.
     *
     * @param flightNumber the flight number
     * @param origin the origin airport
     * @param destination the destination airport
     * @param departureDate the departure date of the flight
     * @param numberOfSeats the number of seats available on the flight
     * @param price the price of the flight
     */
    public AddFlight(String flightNumber, String origin, String destination, LocalDate departureDate, int numberOfSeats, int price) {
        this.flightNumber = flightNumber;
        this.origin = origin;
        this.destination = destination;
        this.departureDate = departureDate;
        this.numberOfSeats = numberOfSeats;
        this.price = price;
    }

    /**
     * Executes the command to add a new flight to the flight booking system.
     *
     * @param flightBookingSystem the flight booking system
     * @throws FlightBookingSystemException if an error occurs while executing the command
     */
    @Override
    public void execute(FlightBookingSystem flightBookingSystem) throws FlightBookingSystemException {
        LocalDate currentDate = LocalDate.now();
        if (departureDate.isBefore(currentDate)) {
            throw new FlightBookingSystemException("Cannot add flight with a departure date in the past.");
        }

        // Get the maximum flight ID currently in the system
        int maxId = flightBookingSystem.getFlights().stream().mapToInt(Flight::getId).max().orElse(0);

        // Create a new flight object with the next available ID
        Flight flight = new Flight(++maxId, flightNumber, origin, destination, departureDate, numberOfSeats, price);
        flightBookingSystem.addFlight(flight); // Add the flight to the flight booking system
        System.out.println("Flight #" + flight.getId() + " added.");

        // Write the new flight data to the flights.txt file
        try (BufferedWriter writer = new BufferedWriter(new FileWriter("resources/data/flights.txt", true))) {
            DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyy-MM-dd");
            String formattedDate = departureDate.format(dtf);
            writer.write(flight.getId() + "," + flight.getFlightNumber() + "," + flight.getOrigin() + "," + flight.getDestination() + "," + formattedDate + "," + flight.getNumberOfSeats() + "," + flight.getPrice());
            writer.newLine();
        } catch (IOException ex) {
            throw new FlightBookingSystemException("Error writing to flights.txt: " + ex.getMessage());
        }
    }
}
```

Description

The `AddFlight` class in a flight booking system application serves as a command that enables the addition of new flights. It accepts parameters such as flight number, origin, destination, departure date, number of seats, and price to create a new flight object. Before adding the flight, the command ensures that the departure date is not in the past, validating the timing of the new flight. Once validated, the command updates the flight booking system with the newly added flight and notifies users with a confirmation message. Furthermore, it ensures the persistence of flight details by writing them to a file named `flights.txt`, ensuring that all flight information is securely stored for future reference. This command simplifies the management of flights within the booking system by providing a straightforward way to add new flights with accurate and essential details.

Output

```
Main (7) [Java Application] C:\Program Files\Java\jdk-21\bin\javaw.exe (
Flight Booking System
Enter 'help' to see a list of available commands.
> addflight
Flight Number: 2
Origin: ktm
Destination: newyork
Departure Date ("YYYY-MM-DD" format): 2024-06-23
Number of Seats: 5
Price: 25000
Flight #8 added.
>
```

7. ListFlights

```java
/* Command to list all upcoming flights in the flight booking system.
package bcu.cmp5332.bookingsystem.commands;

import bcu.cmp5332.bookingsystem.main.FlightBookingSystemException;

/**
 * Command to list all upcoming flights in the flight booking system.
 */
public class ListFlights implements Command {

    /**
     * Executes the command to list all upcoming flights in the flight booking system.
     *
     * @param flightBookingSystem the flight booking system
     * @throws FlightBookingSystemException if an error occurs while executing the command
     */
    @Override
    public void execute(FlightBookingSystem flightBookingSystem) throws FlightBookingSystemException {
        List<Flight> flights = readFlightsFromFile("resources/data/flights.txt");
        LocalDate today = LocalDate.now();
        flights = filterFlights(flights, today);
        for (Flight flight : flights) {
            System.out.println(flight.getDetailsShort());
        }
        System.out.println(flights.size() + " flight(s)");
    }

    /**
     * Reads the list of flights from the specified file.
     *
     * @param filename the name of the file containing flight data
     * @return a list of flights
     * @throws FlightBookingSystemException if an error occurs while reading the file
     */
    private List<Flight> readFlightsFromFile(String filename) throws FlightBookingSystemException {
        List<Flight> flights = new ArrayList<>();
        try (BufferedReader reader = new BufferedReader(new FileReader(filename))) {
            String line;
            while ((line = reader.readLine()) != null) {
                String[] parts = line.split(",");
                int id = Integer.parseInt(parts[0]);
                String flightNumber = parts[1];
                String origin = parts[2];
                String destination = parts[3];
                LocalDate departureDate = LocalDate.parse(parts[4]);
                int numberOfSeats = Integer.parseInt(parts[5]);
                double price = Double.parseDouble(parts[6]);

                Flight flight = new Flight(id, flightNumber, origin, destination, departureDate, numberOfSeats, price);
                flights.add(flight);
            }
        } catch (IOException | NumberFormatException e) {
            throw new FlightBookingSystemException("Error reading flights from file: " + e.getMessage());
        }
        return flights;
    }

    /**
     * Filters the list of flights to include only those that depart after the specified date.
     *
     * @param flights the list of flights to filter
     * @param today the date to compare against
     * @return a list of flights that depart after the specified date
     */
    private List<Flight> filterFlights(List<Flight> flights, LocalDate today) {
        return flights.stream()
                .filter(flight -> flight.getDepartureDate().isAfter(today))
                .collect(Collectors.toList());
    }
}
```

Description

The `ListFlights` class functions as a command within a flight booking system, designed to list all available flights. When executed, it retrieves flight information stored in a file named `flights.txt`. Each line of this file is parsed to create individual Flight objects. These flights are filtered based on whether their departure date occurs after the current date, ensuring that only future flights are included in the listing. Once filtered, details of each flight are displayed on the console using the `getDetailsShort()` method, accompanied by a total count of the flights listed. If any issues arise during the reading or parsing of the file, such as errors in data format, the command handles these scenarios by throwing a FlightBookingSystemException with a relevant error message. Overall, this command provides users with a clear overview of upcoming flights available in the system, facilitating informed flight selection and effective management within the booking system.

Output

```
> listflights
Flight #5 - 1234 - ASD to HGFD on 30/06/2024 - Price: $200.0 - Seats: 150
Flight #6 - 1 - ktm to uk on 24/06/2024 - Price: $20000.0 - Seats: 2
Flight #8 - 7 - ktm to england on 23/07/2024 - Price: $30000.0 - Seats: 1
3 flight(s)
>
```

8. CancelBooking

```java
 * Command to cancel an existing booking in the flight booking system.
package bcu.cmp5332.bookingsystem.commands;

import bcu.cmp5332.bookingsystem.main.FlightBookingSystemException;

/**
 * Command to cancel an existing booking in the flight booking system.
 */
public class CancelBooking implements Command {

    private final int customerId; // ID of the customer
    private final int flightId; // ID of the flight

    /**
     * Constructs a CancelBooking command with the specified customer ID and flight ID.
     *
     * @param customerId the ID of the customer
     * @param flightId the ID of the flight
     */
    public CancelBooking(int customerId, int flightId) {
        this.customerId = customerId;
        this.flightId = flightId;
    }

    /**
     * Executes the command to cancel an existing booking in the flight booking system.
     *
     * @param fbs the flight booking system
     * @throws FlightBookingSystemException if an error occurs while executing the command
     */
    @Override
    public void execute(FlightBookingSystem fbs) throws FlightBookingSystemException {
        Customer customer = fbs.getCustomerByID(customerId);
        if (customer == null) {
            throw new FlightBookingSystemException("Customer not found for ID: " + customerId);
        }

        Flight flight = fbs.getFlightByID(flightId);
        if (flight == null) {
            throw new FlightBookingSystemException("Flight not found for ID: " + flightId);
        }

        Booking booking = null;
        for (Booking b : customer.getBookings()) {
            if (b.getFlight().getId() == flightId) {
                booking = b;
                break;
            }
        }

        if (booking == null) {
            throw new FlightBookingSystemException("No booking found for customer ID: " + customerId + " and flight ID: " + flightId);
        }

        // Cancel the booking
        booking.cancelBooking();

        // Store updated data
        BookingDataManager dataManager = new BookingDataManager();
        try {
            dataManager.storeData(fbs);
        } catch (IOException e) {
            e.printStackTrace();
        }

        System.out.println("Booking successfully canceled for customer ID: " + customerId + " and flight ID: " + flightId);
    }
}
```

Description

The CancelBooking class, part of a flight booking system, implements the Command interface to facilitate the cancellation of specific customer bookings on designated flights. It accepts parameters such as customer and flight IDs, using them to retrieve corresponding customer and flight objects. Upon execution, the command verifies the existence of both entities and throws exceptions if either is not found. Subsequently, it searches for the booking associated with the identified customer and flight; if found, it removes the booking from the customer's records and the customer from the flight's passenger list. Additionally, it updates the booking data file to reflect the cancellation, incorporating robust exception handling for potential file operation errors. Overall, this class enhances the system's capability to efficiently manage and process booking cancellations, ensuring accurate reflection of customer reservations and flight occupancy status.

Output

```
> cancelbooking 5 8
Customer ID: 5
Flight ID: 8
Booking successfully canceled for customer ID: 5 and flight ID: 8
>
```

Data of txt file

All the information is extracted from their respective classes and stored in text files accordingly

1. **Booking.txt**

```
1,2,6,2024-06-23,40150.0
2,4,6,2024-06-23,40200.0
3,1,5,2024-06-23,200.0
4,3,5,2024-06-23,200.0
```

2. **Customers.txt**

```
1,Anushna Chaulagain,9861571677,anushnachaulagain@gmail.com
2,Sugam Adhikari,984444444,sugam@gmail.com
3,sugam,123456,sugam
4,sugam,123456,dssdsd
5,Bibek Shah,9821685754,bikkishah57@gmail.com
6,Subrace Thakulla,9845678934,subrace@gmail.com
```

3. **Flights.txt**

```
2,1234,Kathmandu,Dading,2024-06-18,150,200.0
3,4455,Maitidevi,Boudha,2024-06-19,200,450.0
4,123456,asd,gfd,2024-06-19,200,100.0
5,1234,ASD,HGFD,2024-06-30,150,200.0
6,1,ktm,uk,2024-06-24,2,20000.0
7,1,kathmandu,dheli,2024-06-23,10,20000.0
8,2,ktm,newyork,2024-06-23,5,25000.0
```

Model

1. **Flight.java**

   The Flight class forms the core of the flight booking system, representing individual flights characterized by attributes such as flight number, departure origin, destination, departure date, available seats, and ticket price. It supports essential functionalities including passenger management, booking handling, and calculation of booking costs based on factors such as remaining days until departure and seat availability. By encapsulating its properties and providing methods for controlled access and modification, the class ensures the integrity of flight data. Overall, the Flight class plays a pivotal role in the system, facilitating efficient management of flights and reservations while upholding consistency and reliability in all flight-related operations.

2. **Customer.java**

   The `Customer` class in the flight booking system represents an individual customer and stores essential details like their unique ID, full name, phone number, email address, and a list of bookings they have made. This class provides methods to access and modify these attributes, including getters and setters for customer information and functions to handle bookings such as adding, removing, and retrieving them. Additionally, the class includes functionality to check if a customer has been deleted from the system and whether any of their bookings have been cancelled. Overall, the `Customer` class plays a crucial role in the booking system by managing and storing customer data and their associated bookings efficiently.

3. **Booking.java**

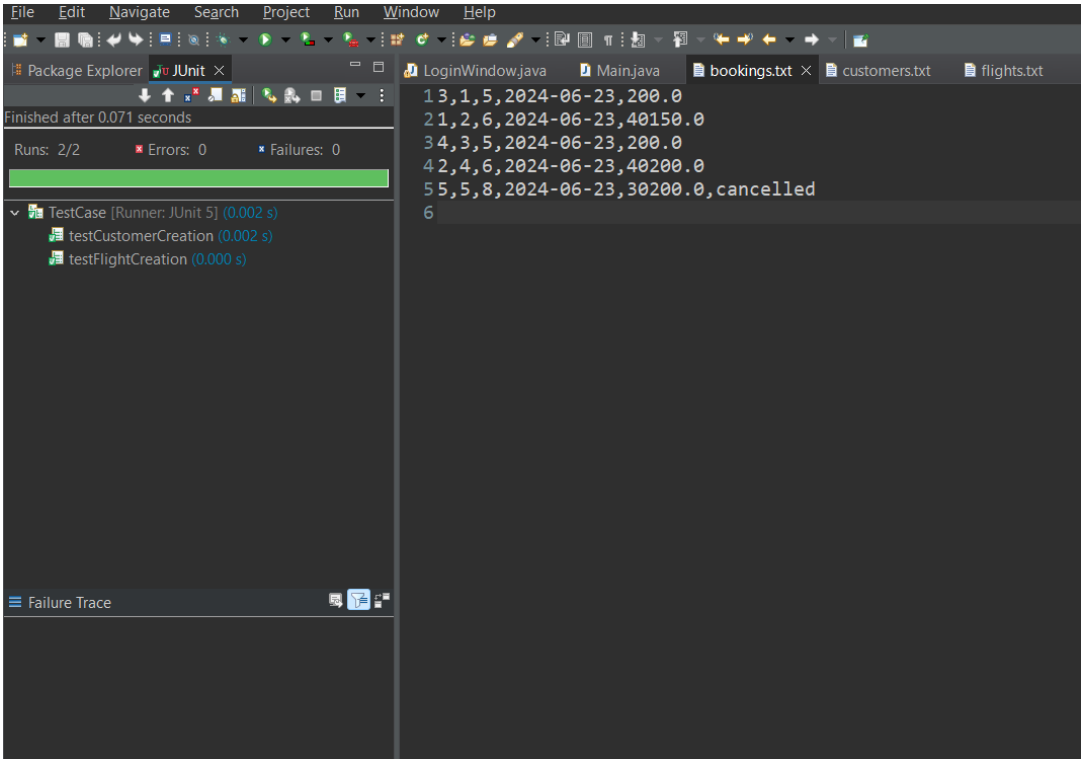
   The Booking class within the flight booking system represents a reservation made by a customer for a specific flight. It includes properties such as an ID assigned to the booking, references to the associated customer and flight, the booking price, the date when the booking was made, and a flag indicating whether the booking has been cancelled. This class encapsulates essential functionalities for handling bookings, offering methods like getters and setters to access and modify booking details. Each booking links a customer directly to a specific flight, establishing a clear connection between the two. The booking date signifies when the reservation was confirmed, while the price indicates its cost. Additionally, the class provides methods to verify if a booking has been cancelled and to update its cancellation status as needed. Overall, the Booking class serves a pivotal role in the flight booking system, effectively representing individual reservations and facilitating their management and monitoring throughout the system.

4. Flight Booking System

The `FlightBookingSystem` class serves as the central hub of the flight booking system, overseeing the management of flights, customers, and bookings. It utilizes maps to organize collections of flights, customers, and bookings, providing essential methods for adding, retrieving, and removing these entities. The class features functionalities such as generating unique booking IDs, retrieving flights, customers, and bookings based on their respective IDs, and accessing bookings associated with specific customers or flights. It also includes operations for deleting flights and customers from the system, along with their associated bookings. Additionally, the class offers utility methods for retrieving bookings using customer and flight IDs, as well as for canceling bookings as needed. Overall, the `FlightBookingSystem` class encapsulates the fundamental operations of the flight booking system, facilitating efficient management and interaction among flights, customers, and their reservations.

Junit Testing

The `FlightAndCustomerTests` class contains JUnit tests designed to validate the creation of Flight and Customer objects within the flight booking system. In the `testFlightCreation` method, it initiates a new Flight object using predefined parameters including ID, flight number, origin, destination, departure date, seat count, and price. The test then verifies that the created Flight object is not null and confirms that its attributes precisely match the provided values through assertion checks. Similarly, in the `testCustomerCreation` method, it creates a Customer object with specified parameters such as ID, name, phone number, and email. The test ensures that the resulting Customer object is not null and accurately reflects the provided attribute values. These tests are essential in ensuring that the constructors of both the Flight and Customer classes initialize objects correctly with the intended attributes.

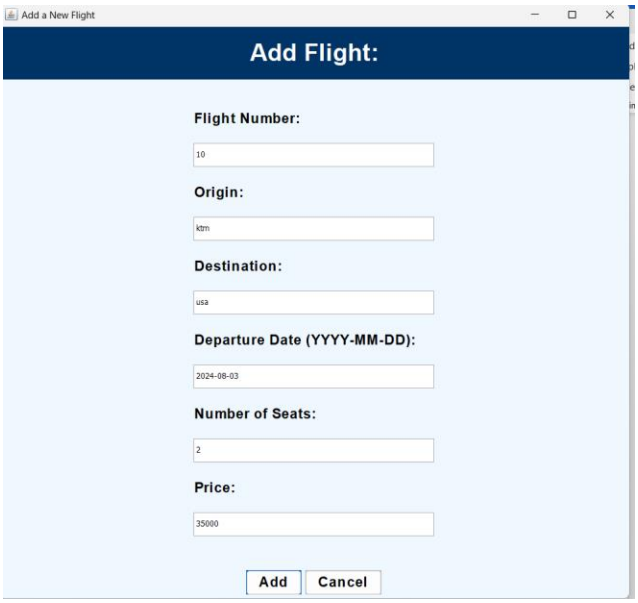


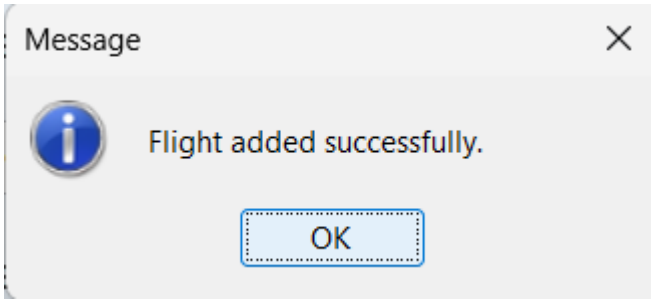
Graphical User Interface (GUI)

This initial interface of our GUI serves as the gateway for users to access different sections such as Admins, Flights, Bookings, and Customers. From here, users can perform actions like booking flights or managing flights, bookings, and customer details based on their preferences and needs.

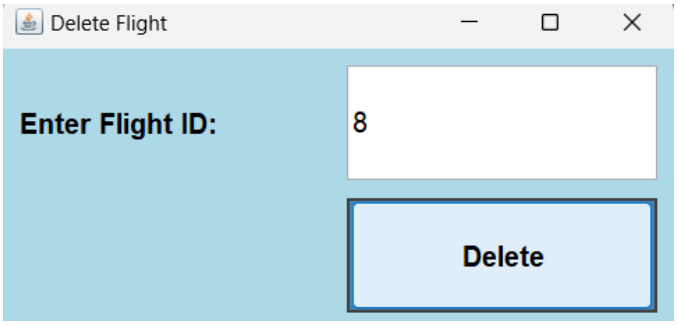
1) Flight

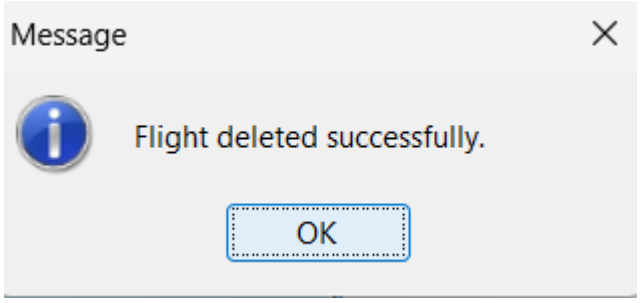
- **Add**



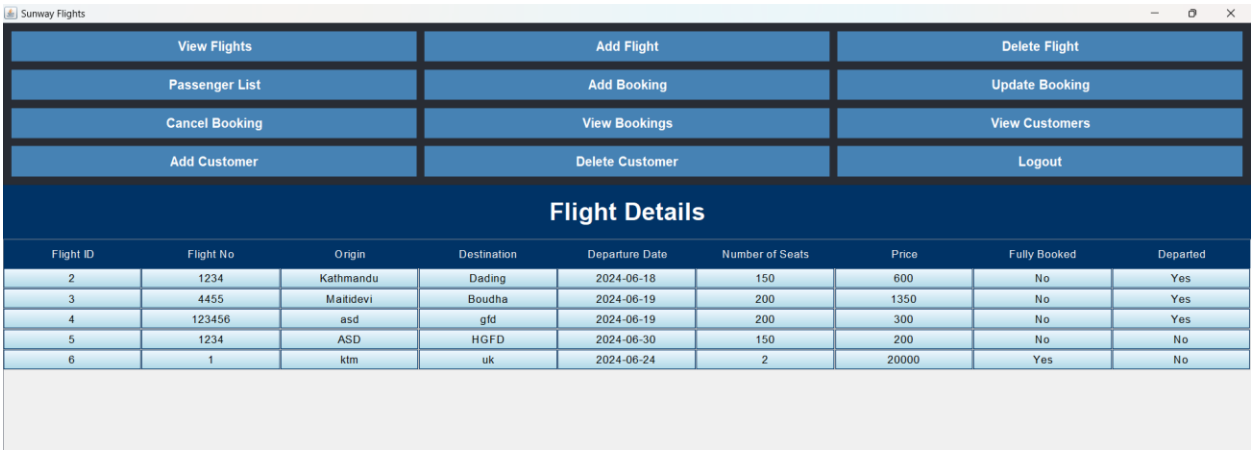


- **Delete**



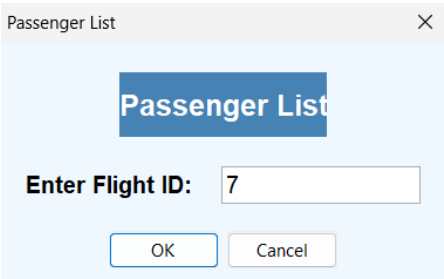


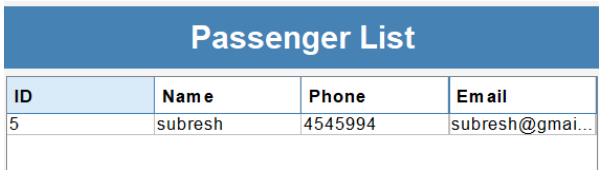
- **View**



Flight ID	Flight No	Origin	Destination	Departure Date	Number of Seats	Price	Fully Booked	Departed
2	1234	Kathmandu	Dading	2024-06-18	150	600	No	Yes
3	4455	Maitidevi	Boudha	2024-06-19	200	1350	No	Yes
4	123456	asd	gfd	2024-06-19	200	300	No	Yes
5	1234	ASD	HGFD	2024-06-30	150	200	No	No
6	1	ktm	uk	2024-06-24	2	20000	Yes	No

- **Passenger List**

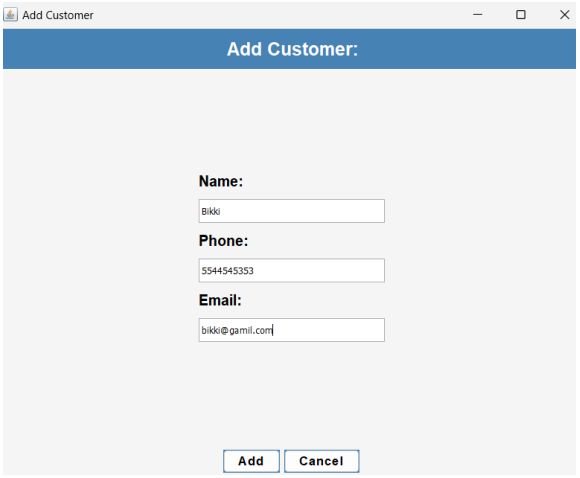


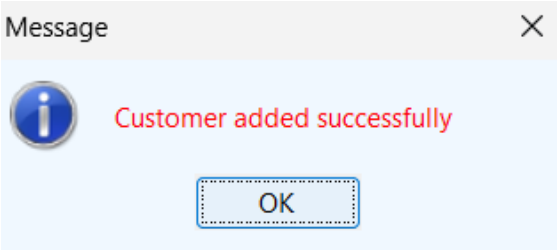


ID	Name	Phone	Email
5	subresh	4545994	subresh@gmai...

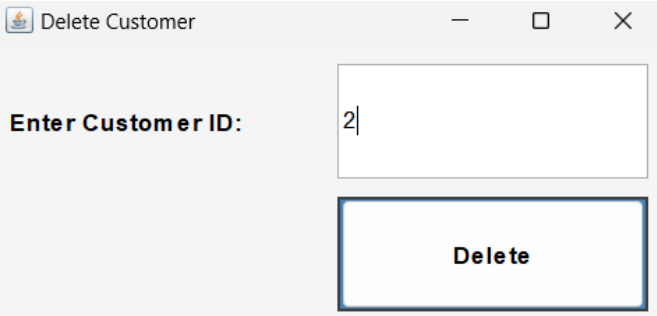
2) Customer

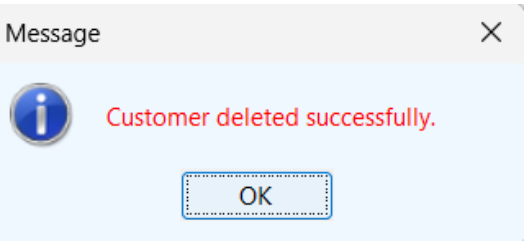
- **Add**





- **Delete**



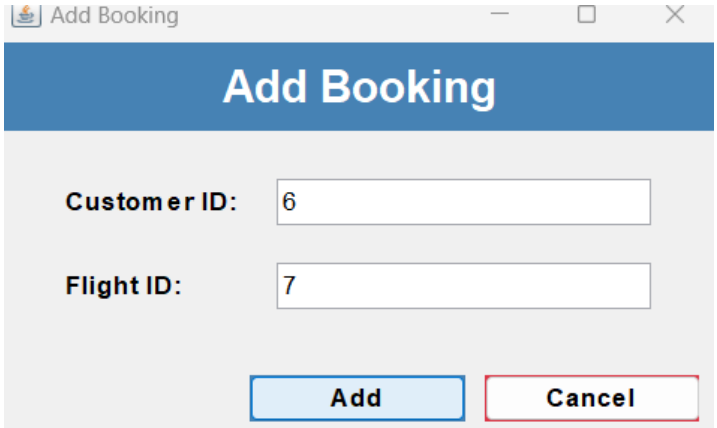


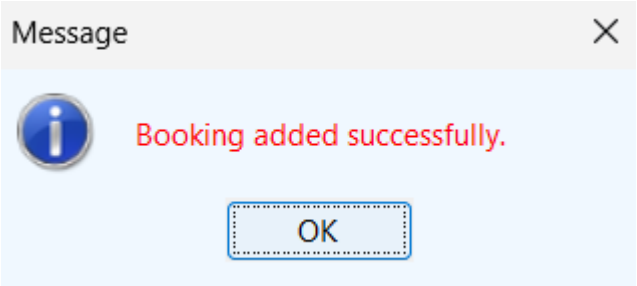
- **View**

ID	Name	Phone	Email	Number of Active Bookings
1	Anushna Chaulagain	9861571677	anushnachaulagain@gmail.com	1
2	Sugam Adhikari	984444444	sugam@gmail.com	1
3	sugam	123456	sugam	1
4	sugam	123456	dssdsd	1
5	subresh Thakulla	986792001	tsubresh@gmail.com	0

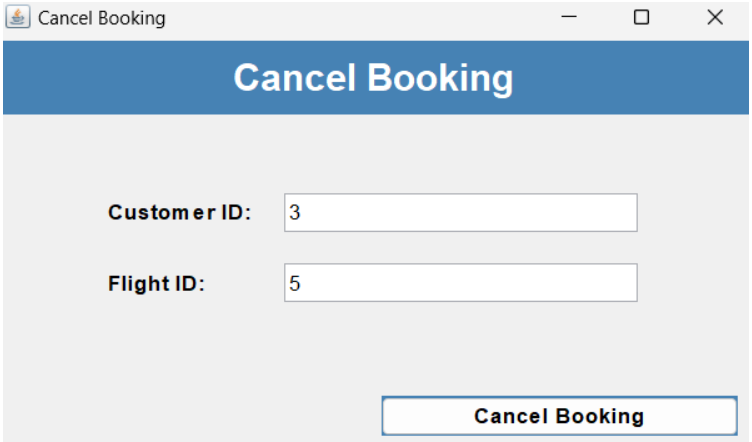
3) Booking

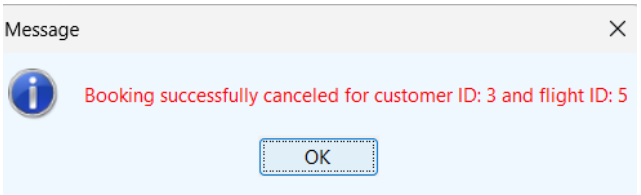
- **Add**


Add Booking
Add Booking
Customer ID:
6
Flight ID:
7
Add
Cancel


Message
Booking added successfully.
OK

- **Cancel**


Cancel Booking
Cancel Booking
Customer ID:
3
Flight ID:
5
Cancel Booking


Message
Booking successfully canceled for customer ID: 3 and flight ID: 5
OK

- **View All**

Booking Details

Booking ID	Customer Name	Flight Number	Booking Date	Price	Status
4	Customer 3	Flight 5	2024-06-23	200.0	Cancelled
2	Customer 4	Flight 6	2024-06-23	40200.0	Active
5	Customer 5	Flight 7	2024-06-23	50150.0	Active
6	Customer 6	Flight 7	2024-06-23	50200.0	Active

Java Documentation

The JavaDocs for our Flight Booking System offer thorough documentation that covers every aspect of the system's functionality, architecture, and usage. Each class, method, and variable is meticulously explained, providing clear details about their purpose, input parameters, return values, and potential exceptions that may occur. This documentation provides valuable insights into how the system operates internally, enabling developers to grasp and utilize the codebase efficiently. Moreover, the JavaDocs include comprehensive explanations of the command-line interface, outlining all available commands and their specific functionalities. This documentation serves as a crucial resource for developers, supporting them in tasks such as system development, maintenance, and troubleshooting. By presenting information in a clear and concise manner, the JavaDocs ensure transparency and facilitate a deeper understanding, empowering developers to confidently extend and improve the Flight Booking System.

Conclusion

In conclusion, the flight booking system offers a user-friendly platform designed to efficiently manage flight reservations. It provides a variety of functions such as adding, listing, and canceling flights, as well as managing bookings for customers, catering to diverse user requirements. Each command within the system is carefully designed to execute specific tasks smoothly, ensuring a seamless user experience. Additionally, robust error handling mechanisms bolster the system's reliability and stability, mitigating potential disruptions effectively. Utilizing file input/output operations enhances data persistence and integrity, further solidifying the system's dependability. With its intuitive interface and comprehensive features, the flight booking system proves invaluable to both customers and administrators in the airline industry, simplifying the reservation process and optimizing operational efficiency.