DEVELOPERS

AIRPORT MANAGEMENT

COMPANY NAME: C SHARP DEVELOPERS
TEAM LEADER: EMINE MELISSA CANDEMIR
SECRETARY: COURTNEY
DEVELOPER 1: ZEEL YAGNIK
DEVELOPER 2: DANA GUZMAN
TESTER: AMIN HEIDARI

INTRODUCTION

This project is an airport management system which manages the departing and arriving flights to Stansted Airport, London, preventing the planes from crashing. The availability of planes. Determining the safety of flying a plane. Estimating travel times. Booking flights for customers. Finally the project is also responsible for the display screens in the airport displaying flight details for onlooking passengers.

REPORT LAYOUT

THE PLAN

Booking Flights (Passenger Interface)
The booking process begins with passengers selecting their travel route from a dropdown menu, with Stansted set as the default departure location. Once a destination, and a date is chosen, the system displays selectable available flights, including key details such as departure time, duration, price, and flight ID.

Passengers have the option to add additional baggage beyond the standard 2kg carry-on allowance for a fee of £15. A visual representation of the aircraft seating layout is presented, allowing them to select a preferred seat before proceeding.

Upon confirming their selections, passengers have to complete a personal information form. This form collects essential details, including passport number (used as the primary key in the database), full name, phone number, date of birth, and address details. The system also records the selected plane seat and baggage preferences.

Next passengers are directed to a mock payment section, where they enter their payment details. Once completed, the system generates a unique four-character booking ID, which is stored in the database. Their booking confirmation is emailed to them if they choose to press a button that does so along with all selected flight details and their passport number.

Check-in System
The check-in system is accessible to administrators via a secure login. Once logged in, admins can search for passengers using their passport number, verify their full name, and check them in.

The admin can modify plane, passenger, employee, and flight details. This modification is seen in the database.

A separate page allows administrators to manage baggage modifications. If a passenger exceeds the baggage limit, additional charges are processed via an integrated payment form. Conversely, if baggage is reduced, a refund is manually processed. These updates ensure the baggage weight is accurately recorded in the database.

Assigning Flights

Flight assignments are managed through an admin login, granting access to a flights page displaying a weekly timetable. This timetable presents flights with details such as flight number, destination, and terminal number. To ensure safe air traffic management.

The system automates flight assignments using SQL, ensuring aircrafts are allocated based on availability. Flight records include flight number, gate number, destination, date/time, assigned aircraft, and pilot crew.

Aircraft availability is stored as a boolean property, and seating capacity is tracked through an integer field. Similarly, pilots are assigned based on their availability status, which is managed as a boolean value. The system checks for conflicting schedules before finalizing assignments; if a conflict is detected, it moves on to the next available aircraft or pilot.

DESIGN

THE PSEUDOCODES

Baggage handling by developer Dana.

```
SET free_weight, paid_weight, baggage_fee, total_baggage_weight = 2, 20, 0, 0
SET user_answer = ""
SET user_add_baggage = true

WHILE user_add_baggage IS true DO
    DISPLAY "Enter baggage weight: "
    INPUT INT baggage_weight
    IF baggage_weight <= free_weight THEN
        baggage_fee = 0
        total_baggage_weight = baggage_weight
    ELSE IF baggage_weight > free_weight AND baggage_weight <= paid_weight THEN
        baggage_fee = 20
        DISPLAY "Maximum 20kg allowed for additional baggage"
    END IF

    DISPLAY "Your baggage fee is £" + baggage_fee
    DISPLAY "Total baggage weight is " + total_baggage_weight +"kg"
    DISPLAY "Do you want to add more baggage (yes/no)?"
    INPUT STRING user_answer

    IF user_answer = "no" THEN
        user_add_baggage = false
    END IF
END WHILE

DISPLAY "Your total baggage weight is " + total_baggage_weight + "kg"
```

Selecting travel routes (default start location is Stansted, dropdown searchable destinations) by team leader.

```
STRING start_location = "Stansted"
//SQL table
STRING ARRAY destinations = ["Delhi", "Istanbul", "Madrid", "Paris", "Tokyo"]
//HTML dropdown linked to SQL and to passenger table (assigning destination)
LIST destinations
```

The pseudocode for recieving the passengers details with data types.

```
DISPLAY "Passenger details form"

INPUT INTEGER passport_number Display "Passport number"
INPUT STRING first_name DISPLAY "First name"
INPUT STRING last_name DISPLAY "Last name"
INPUT INTEGER phone_number Display "Phone number"
INPUT DATE date_of_birth Display "Date of birth"
INPUT STRING postcode DISPLAY "Postcode"
INPUT STRING street_line1 DISPLAY "Street Line 1"
INPUT STRING street_line2 DISPLAY "Street Line 2"
INPUT STRING city DISPLAY "City"

FOR EACH INPUT
    IF INPUT == BLANK
        DISPLAY "Please enter details for said " INPUT " "
    ELSE DISPLAY checkbox

BUTTON = "Submit"
SAVE details TO SQL_DATABASE under PASSENGER
DISPLAY "Your details have been registered you will now be taken to the payment page"
```

Displaying available flights for selected routes (take off time, duration, price) by team leader.

```
FOR destinations[x]
    IF LIST destinations[x] == true
    STRING chosen_destination = destinations[x]
//HTML
TICKLIST free_weight, paid_weight
//C#
INTEGER displayed_flights = 0
FOR flight_table[x]
    IF flight_table.destination == chosen_destination
    DISPLAY flight_table[x].time, flight_table[x].duration, flight_table[x].price
    INTEGER displayed_flights += 1
IF displayed_flights == 0
    DISPLAY STRING "Sorry no flights available for this combination right now"
```

The pseudocode for the tables and fields we decided the database should contain

```
TABLE Passenger = [PassengerID PRIMARY_KEY, FullName, PassportNumber, Email, Phone, DOB, Address, FlightID]
TABLE Flight = [FlightID PRIMARY_KEY, Destination, DepartureTime, ArrivalTime, GateNumber, Price]
TABLE Employee = [EmployeeID PRIMARY_KEY, FullName, Role, AssignedFlightID]
TABLE Gate = [GateID PRIMARY_KEY, FlightID]
```

The pseudocode for the tables and fields we decided the database should contain.

Developer Zeel decided to use the data structures of lists and arrays for the database's making. The reason being, arrays and lists are simple to use. We needed to store items in order and access them by index therefore arrays and lists were the option she chose.

```
DISPLAY "Admin Login Page"
INPUT Username DISPLAY "Enter Admin ID"
INPUT Password DISPLAY "Enter Password"

IF AdminID == "AdminFromDatabase"
    IF password == "AdminPassword"
        DISPLAY "Login Successful!"
        NAVIGATE TO "Passenger check-in page"
    ELSE DISPLAY "Password is incorrect"
ELSE
    DISPLAY "Login Unsuccessful - Admin does not exist"

After LOGIN -->
DISPLAY "Passenger Search"

INPUT PassportNumber DISPLAY "Enter passenger's passport number"
    SEARCH DATABASE(PassportNumber)

    IF Passenger EXISTS THEN
        DISPLAY Passenger.FullName
        DISPLAY "Check-in checklist"

        PROMPT admin to tick passenger as "Arrived"

        IF admin_ticks_checkbox THEN
            UPDATE Passenger SET Status = "Arrived"
            DISPLAY "Passenger succesfully checked in"
        ELSE
            DISPLAY "Passenger not checked in"
        ENDIF
    ELSE
        DISPLAY "Passenger not found"
    ENDIF
```

The pseudocode above was made by developer Zeel and represents the process in which the admin login will use to login to the website.

The pseudocode to the right was made by developer Dana and represents the process in which a booking will be made.

The O(1) access time for arrays and lists means it is efficient.

```
START UserBookingProcess

    // Step 1: Flight Selection
    DISPLAY "Select a flight from Stansted to your destination"
    DISPLAY available destinations in a dropdown list (fetched from database)
    DISPLAY selected flight details (Time, Duration, Price, Flight ID)

    // Step 2: Baggage Selection
    DISPLAY "Select baggage options"
    DISPLAY checkbox for checked baggage (Flat fee: £20, 23kg)
    DISPLAY included free items (Carry-on, 2kg)
    IF user selects checked baggage THEN
        ADD £20 to total cost
    END IF

    // Step 3: Collect User Information
    DISPLAY "Enter your details"
    PROMPT user for:
        STRING Name
        STRING Email
        INTEGER Passport ID
        INTEGER Phone Number
        DATE Date of Birth
        STRING Address

    IF any required field is missing or invalid THEN
        DISPLAY "Please complete all required fields"
        RETURN to Step 3
    END IF

    // Step 4: Payment Process (simulate payment)
    DISPLAY "Proceed to payment"
    PROMPT user for payment details (e.g., card info)
    IF payment is successful THEN
        DISPLAY "Payment successful"
    ELSE
        DISPLAY "Payment failed, try again"
        RETURN to Step 4
    END IF

    // Step 5: Booking Confirmation
    GENERATE a unique booking reference number (e.g., BKG-123456)
    SAVE user booking information in the Booking Table in the database
    SEND confirmation email with booking details and reference number to the user

    // Step 6: Display Confirmation
    DISPLAY "Your booking is confirmed"
    DISPLAY booking reference number
    DISPLAY flight details

END UserBookingProcess
```

CODING

The team leader produced a step-by-step coding plan inspired by research from Courtney and themselves from which the team leader derived and produced a database structure to be coded by developer Zeel.

Passengers	Flights	DestinationsPrices	Bookings	Planes	Employee	Baggage
passport_id (PK)	flight_id (PK)	destination (PK)	Booking_id (PK)	plane_id (PK)	employee_id (PK)	baggage_id (PK)
full_name	plane_id (FK)	price	passport_id (FK)	seat_capacity	role	passport_id (FK)
phone_number	destination (FK)	airport_name	flight_id (FK)	availability - boolean	employee_name	
dob	departure_dateTime		payment_status - boolean	weight	flight_id (FK)	
address	return_dateTime		seat_number			
baggage - boolean	gate					
checked_in - boolean	duration					
email						
flight_type						
flight_id (FK)						

The PK referring to primary key and FK referring to foreign key.

Courtney, the secretary, has made a HTML prototype website for us to develop the airport flight management system functionality. Simultaneously developing the CSS as the developers worked on coding the previously mentioned pseudocodes.

Developer Zeel used a Code First approach in Entity Framework to design the database schema to manage the database directly through code, rather than using a database-first method. Allowing to create model classes that represent tables, define relationships using navigation properties, and use migrations to keep the database schema in sync with the code. By using the Package Manager Console, developers can generate and apply migrations, which convert class changes into SQL commands that update the database.

TESTING

The following are the MS and unit testing that were done.
Testing the endpoints for the Baggage table. The time complexities are assumed.

Test Case	Description	Input	Expected Output	Actual Output	Time Complexity	Status
GET /api/Baggage	Retrieve all baggage records	GET `/api/Baggage`	HTTP 200, list of all baggage	Successfully returned all records	O(n)	✅ PASS
GET /api/Baggage/{BaggageID}						✅ PASS
→ Valid BaggageID	Retrieve baggage by valid ID		HTTP 200, baggage details	Correct baggage record returned	O(1)	✅ PASS
→ Invalid BaggageID	Retrieve baggage by invalid ID	GET `/api/Baggage/999`	HTTP 404	Proper 404 response	O(1)	✅ PASS
POST /api/Baggage						✅ PASS
→ Valid request	Add baggage with valid passenger	POST `{ "BaggageID": 101, "PassportId": "P123456" }`	HTTP 200, baggage created	Baggage added successfully	O(1)	✅ PASS
→ Invalid passenger	Add baggage with invalid passenger	POST `{ "BaggageID": 102, "PassportId": "INVALID" }`	HTTP 400, error message	Correct error response	O(1)	✅ PASS
DELETE /api/Baggage/{BaggageID}						✅ PASS
→ Valid deletion	Delete baggage with valid ID	DELETE `/api/Baggage/1`	HTTP 200, baggage removed	Successful deletion	O(1)	✅ PASS
→ Invalid ID	Delete baggage with invalid ID	DELETE `/api/Baggage/999`	HTTP 404	Proper 404 response	O(1)	✅ PASS

Testing the endpoints for the Bookings table.

Test Case	Description	Input	Expected Output	Actual Output	Time Complexity	Status
GET /api/Bookings	Retrieve all bookings	GET `/api/Bookings`	HTTP 200, all bookings	Successfully returned all records	O(n)	✅ PASS
GET /api/Bookings/{BookingID}						✅ PASS
→ Valid BookingID	Retrieve booking by valid ID	GET `/api/Bookings/BK123` (where BK123 exists)	HTTP 200, booking details	Correct booking returned	O(1)	✅ PASS
→ Invalid BookingID	Retrieve booking by invalid ID	GET `/api/Bookings/INVALID_ID`	HTTP 404	Proper 404 response	O(1)	✅ PASS
POST /api/Bookings						✅ PASS
→ Valid booking creation	Add valid booking	POST `{ "PassportId": "P123456", "FlightID": "FL100", "PaymentStatus": "Paid", "SeatNumber": "A12" }`	HTTP 200, booking created	Booking added with correct details	O(1)	✅ PASS
→ Missing required fields	Missing required field (SeatNumber)	POST `{ "PassportId": "P123456", "FlightID": "FL100", "PaymentStatus": "Paid" }`	HTTP 400	Correctly rejected	O(1)	✅ PASS

System Under Test: The PassengersController class in the Controllers namespace is responsible for managing passenger data via the following endpoints:

Test Case	Input	Expected Output	Time Complexity	Status
Get all passengers	GET `/api/Passengers`	200 OK, list of all	O(n)	✅ PASS
Get by PassportId	Valid & Invalid ID	200 OK / 404 Not Found	O(1)	✅ PASS
Add passenger	Valid AddPassengerDto	200 OK	O(1)	✅ PASS
Update passenger	Valid + Invalid PassportId	200 OK / 404 Not Found	O(1)	✅ PASS
Delete passenger	Valid + Invalid PassportId	200 OK / 404 Not Found	O(1)	✅ PASS

Testing the endpoints for the DestinationsPrices table.

Test Case	Description	Input	Expected Output	Actual Output	Time Complexity	Status
GET /api/DestinationsPrices	Retrieve all destinations	GET `/api/DestinationsPrices`	HTTP 200, all destinations	Returned successfully	O(n)	✅ PASS
GET /api/DestinationsPrices/{Destination}						
→ Valid destination	Retrieve valid destination	GET `/api/DestinationsPrices/Paris`	HTTP 200, destination details	Correct destination returned	O(1)	✅ PASS
→ Invalid destination	Retrieve invalid destination	GET `/api/DestinationsPrices/InvalidCity`	HTTP 404	Proper 404 response	O(1)	✅ PASS
POST /api/DestinationsPrices						✅ PASS
→ Valid destination creation	Add valid destination	POST `{ "Destination": "Tokyo", "Price": 899.99, "AirportName": "Narita International" }`	HTTP 200	Destination added	O(1)	✅ PASS
→ Missing required fields	Missing required field (Price)	POST `{ "Destination": "Berlin", "AirportName": "Brandenburg" }`	HTTP 400	Proper rejection	O(1)	✅ PASS
PUT /api/DestinationsPrices/{Destination}						✅ PASS
→ Valid price update	Update valid destination	PUT `/api/DestinationsPrices/Paris` `{ "Price": 459.99 }`	HTTP 200, updated record	Price updated successfully	O(1)	✅ PASS
→ Update non-existent destination	Update non-existent destination	PUT `/api/DestinationsPrices/Atlantis` `{ "Price": 999.99 }`	HTTP 404	Correct error	O(1)	✅ PASS

Testing the endpoints for the Planes table.

Test Case	Description	Input	Expected Output	Time Complexity	Status
GET /api/Planes	Retrieve all planes	GET `/api/Planes`	HTTP 200, all plane objects	O(n)	✅ PASS
GET /api/Planes/{PlaneID}					
→ Existing aircraft	Valid plane ID	GET `/api/Planes/1`	HTTP 200, plane details	O(1)	✅ PASS
→ Non-existent aircraft	Invalid plane ID	GET `/api/Planes/999`	HTTP 404	O(1)	✅ PASS
POST /api/Planes					
→ Valid aircraft creation	Valid aircraft creation	POST `{ "planeId": 101, "availability": true, "seatCapacity": 220, "weightCapacity": 25000 }`	HTTP 200	O(1)	✅ PASS
→ Invalid capacity values	Invalid capacities	POST `{ "planeId": 102, "seatCapacity": -50, "weightCapacity": 0 }`	HTTP 400	O(1)	✅ PASS
PUT /api/Planes/{PlaneID}					
→ Valid update	Valid update	PUT `/api/Planes/1` `{ "availability": false, "seatCapacity": 150, "weightCapacity": 18000 }`	HTTP 200	O(1)	✅ PASS
→ Update non-existent plane	Update non-existent plane	PUT `/api/Planes/999` `{ "availability": true }`	HTTP 404	O(1)	✅ PASS
DELETE /api/Planes/{PlaneID}					
→ Valid deletion	Valid deletion	DELETE `/api/Planes/2`	HTTP 200	O(1)	✅ PASS
→ Delete non-existent plane	Delete non-existent plane	DELETE `/api/Planes/999`	HTTP 404	O(1)	✅ PASS
Business logic validation					
→ Seat capacity limits	Seat capacity > 500	Input rejected		O(1)	✅ PASS
→ Weight capacity floor	Weight capacity < 1000	Input rejected		O(1)	✅ PASS
→ Unique PlaneID enforcement	Duplicate plane IDs	Rejected		O(1)	✅ PASS

System Under Test: The Payment class in the GroupCoursework.Functionality namespace handles payment-related functionality, including:

- Calculation of total payment based on destination price and baggage fees.
- Confirmation of payment status with a unique booking ID.

Test Case	Destination Price Initialization	Baggage Fee Handling	Total Payment Calculation	Payment Confirmation Logic	Booking ID Generation
Result	✅ Passed	✅ Passed	✅ Passed	✅ Passed	✅ Passed
Time Complexity	O(1)	O(1)	O(1)	O(1)	O(1)

Testing the endpoints for the Employees table.

Test Case	Description	Input	Expected Output	Actual Output	Time Complexity	Status
GET /api/Employees	Retrieve all employees	GET `/api/Employees`	HTTP 200, list of employees	Returned successfully	O(n)	✅ PASS
GET /api/Employees/{EmployeeID}						✅ PASS
→ Valid EmployeeID	Valid employee ID	GET `/api/Employees/101`	HTTP 200, employee details	Correct record returned	O(1)	✅ PASS
→ Invalid EmployeeID	Invalid employee ID	GET `/api/Employees/9999`	HTTP 404	Proper 404 response	O(1)	✅ PASS
POST /api/Employees						✅ PASS
→ Valid employee creation	Valid employee creation	POST `{ "EmployeeId": 105, "EmployeeName": "John Smith", "Role": "Flight Attendant", "FlightID": "FL205" }`	HTTP 200	Employee added	O(1)	✅ PASS
→ Missing required fields	Missing fields	POST `{ "EmployeeId": 106, "Role": "Pilot" }`	HTTP 400	Correct rejection	O(1)	✅ PASS
PUT /api/Employees/{EmployeeID}						✅ PASS
→ Valid employee update	Valid employee update	PUT `/api/Employees/101` `{ "Role": "Senior Pilot", "FlightID": "FL101" }`	HTTP 200	Record updated	O(1)	✅ PASS
→ Update non-existent employee	Update non-existent employee	PUT `/api/Employees/9999` `{ "Role": "Manager" }`	HTTP 404	Correct error	O(1)	✅ PASS
DELETE /api/Employees/{EmployeeID}						✅ PASS
→ Valid employee deletion	Valid employee deletion	DELETE `/api/Employees/103`	HTTP 200	Deletion successful	O(1)	✅ PASS
→ Delete non-existent employee	Delete non-existent employee	DELETE `/api/Employees/9999`	HTTP 404	Proper error	O(1)	✅ PASS

Testing the whole system

Test Case	Database Connectivity	CORS Policy	Swagger UI	HTTPS Redirection	Static Files	Controller Mapping
Result	✅ Passed	✅ Passed	✅ Passed	✅ Passed	✅ Passed	✅ Passed
Time Complexity	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)

The UpdatePassengerDto class

Test Case	Required Property Validation	Optional Fields (DOB)	Boolean Fields	Data Consistency	Edge Cases (Long Strings, Invalid Emails)
Result	✅ Passed	✅ Passed	✅ Passed	✅ Passed	✅ Passed
Time Complexity	O(1)	O(1)	O(1)	O(n)	O(n)

The DestinationManager Tests

Test Case	Initialization Values	ShowDestinations() Output	Immutability Enforcement	Empty List Behaviour
Result	✅ Passed	✅ Passed	✅ Passed	✅ Passed
Time Complexity	O(1)	O(n)	O(1)	O(1)

CONCLUSION

To summarise we made a database with all the tables an airport would need to be managed. A website was also made to reflect the needs of both passengers and airport staff. Airport staff can view and modify in a controlled manner, elements of the database. Email confirmation can be sent albeit in a limited manner through swagger . This limitation was caused by not being comfortable with handling the database from the backend code. Another limitation we had was the lack of set working times.

Start working from a shared repository sooner, be more active in starting to code despite lack of knowledge as the internet can help fill that void of knowledge. Be sure to test for every small change that we make so that debugging becomes easier. Be cautious of other teammates work. Next time we have a similar project, we shall establish a common working time schedule for the whole team and the scrum master should make sure the team follows it.

REFERENCES

Airport Management System Research References

AeroCloud Systems, n.d. Flight Management System | Airport Operations. [online] Available at: https://aerocloudsystems.com/airport-operations-system/flight-management/ [Accessed 17 Apr. 2025].

Code References

Bro Code, 2022. C# Full Course for Beginners | Learn C#. [YouTube playlist] Available at: https://www.youtube.com/watch?v=Et2khGnrIqc&list=PLLWMQd6PeGY3b89Ni7xsNZddi9wD5Esv2 [Accessed 17 Apr. 2025].

Microsoft, 2023. Create a web API with ASP.NET Core. [online] Available at: https://learn.microsoft.com/en-us/aspnet/core/web-api/?view=aspnetcore-7.0 [Accessed 17 Apr. 2025].

Microsoft, 2023. SmtpClient Class (System.Net.Mail). [online] Available at: https://learn.microsoft.com/en-us/dotnet/api/system.net.mail.smtpclient [Accessed 10 Apr. 2025].