

Ailine Ticketing System
(Technical Report)

Jose Leonardo Wong

Contents

1.	Airline Ticketing System	2
1.1	Introduction	2
1.2	Business Requirements	3
1.3	Database Design.....	4
1.3.1	Design Assumptions	4
1.3.2	Conceptual Design.....	5
1.3.2.1	Entities and Attributes.....	6
1.3.2.2	Conceptual Relationship Summary	7
1.3.2.3	Conceptual ER Diagram.....	8
1.3.3	Logical Design	9
1.3.3.1	Junction Tables	9
1.3.3.2	Enumerated Fields	9
1.3.3.3	Logical Entity Relationships.....	10
1.3.3.4	Normalisation.....	11
1.3.3.5	Data Types.....	12
1.3.4	Physical Design	13
1.3.5	Implementation.....	16
1.3.5.1	Data Integrity and Concurrency	16
1.3.5.2	Creating the Database.....	18
1.3.5.3	Creating the Tables.....	18
1.3.5.4	Populating the Tables	23
1.3.5.5	Database Security Implementation.....	30
1.3.5.6	Backup and Recovery Implementation	34
1.4	Database Objects and Business Logic Implementation	36
1.4.1	Reservation Date Constraint.....	36
1.4.2	Passenger Queries: Pending Reservations & Age Filtering.....	38
1.4.3	Stored Procedures for Passenger, Employee, and Reservation Management	39
1.4.4	Employee Revenue View per Flight.....	44
1.4.5	Seat Status Trigger.....	46
1.4.6	Checked-in Baggage Function	48
1.4.7	Extended Functionality: Reservation, Seat, and Ticket Procedures	49
1.5	Additional Recommendations.....	55
1.5.1	Data Integrity and Concurrency Recommendations	55

1.5.2	Security Recommendations.....	56
1.5.3	Backup and Recovery Recommendations	57
1.6	Conclusions	57

1. Airline Ticketing System

1.1 Introduction

This technical report presents the design and implementation of a database solution for an airline ticketing system that manages reservations, seat assignments, ticket issuance, and additional services. The system is required to have functionality to allow airport staff to handle passenger bookings and issue boarding passes securely and efficiently. The approach to tackle the development of the solution involves following a structured process in response to the gathered requirements, comprising conceptual, logical, and physical design stages and then implementing it and carrying out testing along the way using T-SQL in SQL Server Management Studio.

1.2 Business Requirements

Design and implement a relational database in Microsoft SQL Server that supports the full ticketing lifecycle, from reservations through to check-in and boarding.

Employee Access

- Ticketing staff and supervisors must log in with secure credentials.
- Staff can manage passenger bookings; supervisors can grant access if authentication fails.

Passenger Reservations

- Passengers book flights in advance, identified by a Passenger Name Record (PNR).
- A PNR includes flight number, itinerary, status (pending, confirmed, cancelled), and seat assignment.
- If a preferred seat is not available, the seat is stored as `NULL`.

Ticket Issuance

- When issuing a ticket, the system must generate an e-boarding number.
- This number is stored against the passenger's PNR along with the ID of the issuing employee.
- Additional fees must be calculated for optional services:
 - Excess baggage: £100/kg
 - Meal upgrade: £20/person
 - Preferred seat: £30/person

Flights & Tickets

- Each flight (FlightID, number, origin, destination, departure/arrival times) may have multiple tickets.
- Tickets (TicketID, issue date/time, fare, seat number, class) link passengers to flights and employees.
- Passengers can hold multiple tickets for different journeys.

Baggage

- The system must track baggage linked to tickets and flights.
- Each record includes baggage weight, status (checked-in, loaded, missing), and associated fees.

1.3 Database Design

1.3.1 Design Assumptions

The design included the following assumptions, beyond what the task's scenario mentions, and based on our interpretation:

- The airline operates only a single model of aircraft, which has the same seat configuration and layout. There are 10 seats in economy class, 5 in business class, and 5 in first class.
- Multi-leg flights, such as connections or multi-city flights, require one reservation per leg.
- A PNR involves only one single reservation (No group reservations).
- A reservation can result in only one ticket.
- A ticket is issued with the involvement of only one employee.
- Preferred seat is optional and can be selected at reservation time or later, including when confirming the reservation details for the emission of a ticket. There are no fares including a preferred seat for free.
- A meal preference (Vegetarian and Non-vegetarian) is optional, free of charge, and can be selected at reservation time or later, including when confirming the reservation details for the emission of a ticket. There are no more meal preferences, so no selection of Vegetarian assumes non-vegetarian preference. The meal upgrade fee does not refer to the meal preference. There are no fares including a free meal upgrade. The set fare applies across all fares.
- Each piece of baggage is limited to 20 kg. and charged at £100.00 per piece. The extra baggage fee, also at £100 charged by the kg, refers to the fee paid for each additional kg over 20kg. The set fee applies across all fares. The

limit on the number of pieces of baggage per passenger is handled at the application level, not at the database level.

- A PNR is a passenger-facing unique identifier of a reservation that allows locating and retrieving the reservation and the details contained in it.
- While the scenario mentions calculating additional charges when issuing the ticket, our interpretation is that this refers to the confirmation of charges already recorded during the reservation, given that in the real world, add-ons such as baggage, meal upgrades, and seat selection are specified at the reservation stage.
- Although the scenario suggests updating the seat's status when the ticket is issued, in our implementation, seat assignment is handled earlier during reservation to align with real-world practices. Therefore, the trigger that updates the seat status to Reserved is placed on the Reservation table when the seat is selected, in addition to the seat status update to Reserved carried out within a stored procedure for ticket issuance.

1.3.2 Conceptual Design

This section covers the conceptual database design, focusing on identifying the entities involved in the airline ticketing system, their relevant attributes, and the relationships between them, including cardinality and optionality, adopting a high-level perspective without considering implementation details or the enforcement of a relational schema. The objective is to capture the key business concepts described in the scenario, including passengers, employees, flights, reservations, tickets, seats, fares, and baggage, defining them as entities and their associated essential characteristics. Additionally, relationships are described conceptually to represent how the entities interact within the system, such as a passenger making a reservation or a ticket being issued by an employee.

1.3.2.1 Entities and Attributes

Table 1 below presents the conceptual entities that were identified.

Table 1: Entities and Attributes.

Entity	Attributes
Passenger	First Name, Last Name, Email, Date of Birth, Emergency Contact Number
Employee	Name, Email, Username, Password, Role
Flight	Flight Number, Origin Airport, Destination Airport, Departure Time, Arrival Time
Reservation	Booking Date, Status, PNR, Meal Upgrade Choice, Meal Preference
Ticket	Issue Date, E-boarding Number
Baggage	Weight, Cost, Status
Seat	Seat Number, Seat Cost
Fare	Fare Class, Base Price, Seat Selection Fee, Excess Baggage Fee Per Kg, Meal Upgrade Fee, Baggage Piece Fee

A Passenger represents individuals placing a reservation with the airline. A Flight refers to a scheduled instance of a route between an origin and a destination. The Flight Number attribute refers to a route between an origin and destination that occurs regularly. A Reservation is a passenger's booking for a specific flight. An Employee refers to the ticketing staff and supervisors responsible for managing bookings and issuing tickets. A Ticket is the document confirming the right to board the plane associated with a reservation. A Seat represents a physical seat in a Flight that can be selected. Baggage represents the luggage services linked to a reservation. Although the Fare entity is not mentioned in the scenario, it was introduced as a way to encapsulate all the rules and pricing structure that govern a flight reservation, including base fare, baggage fees, seat selection charges, and meal upgrades, with the added flexibility of allowing for different sets of rules to be introduced in the future.

Regarding the baggage fee, even though this could be calculated by multiplying its weight times the corresponding fee, we chose to store it as an explicit attribute, Fee, in the Baggage entity to preserve the fee associated with a piece of baggage, even if the airline pricing rules change in the future.

1.3.2.2 Conceptual Relationship Summary

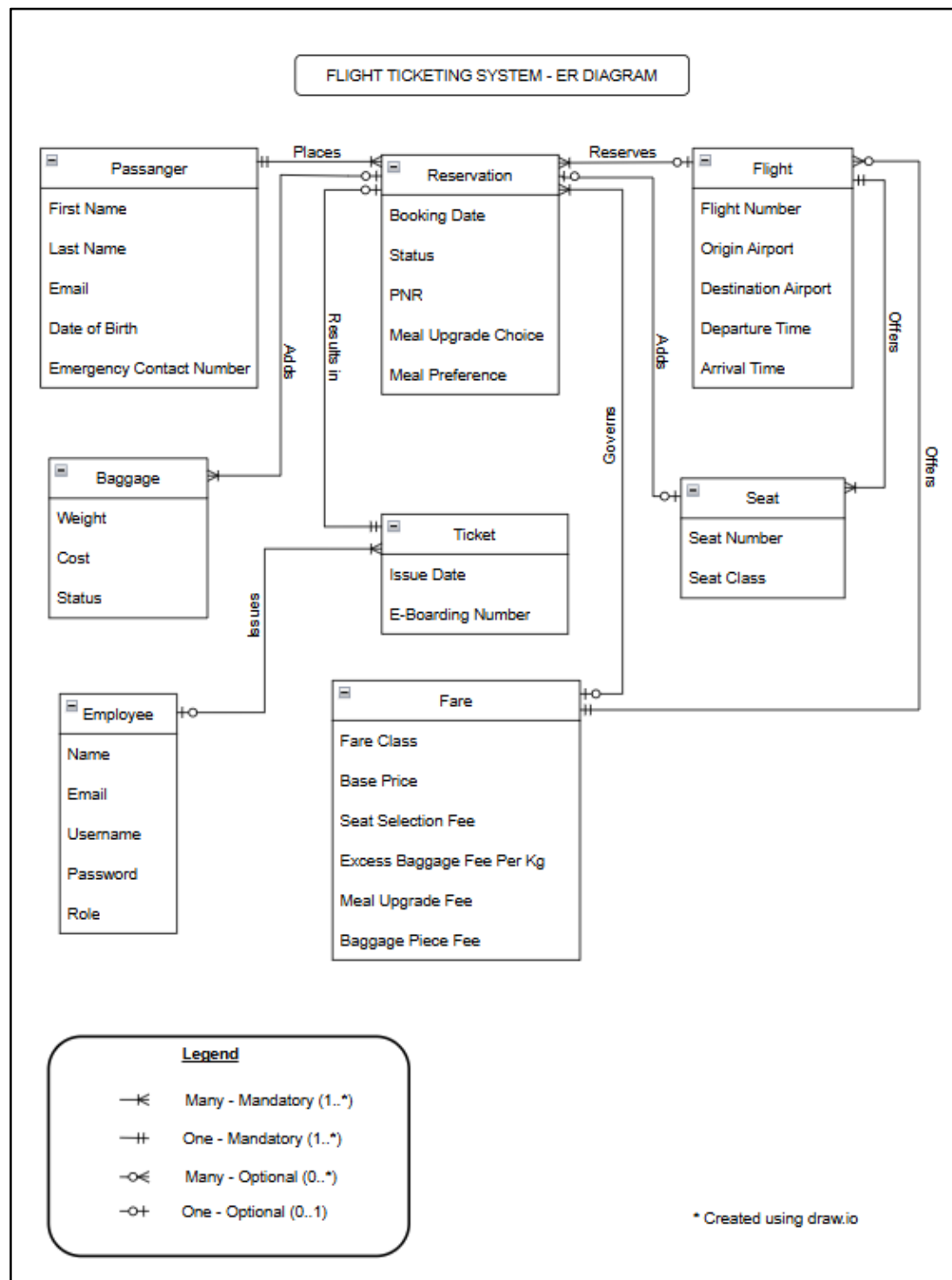
The following conceptual entity relationships were identified:

- Each Reservation is made by one Passenger.
- Each Passenger may make multiple Reservations, but only one per Flight.
- Each Reservation is for only one Flight.
- Each Flight may have many Reservations.
- Each Reservation uses exactly one Fare.
- Each Fare may be used in many Reservations.
- Each Reservation may result in one Ticket or none.
- Each Ticket must be associated with only one Reservation.
- Each Ticket is issued by only one Employee.
- Each Employee may issue many Tickets.
- Each Flight includes many Seats.
- Each Seat exists in the context of only one Flight.
- Each Reservation may include a selected seat (optional).
- Each Seat may be selected by only one Reservation for that Flight.
- Each Reservation may include multiple Baggage items.
- Each Baggage item is associated with exactly one Reservation.
- Each Fare applies to only one Flight and one Class (stored as an attribute)
- Each Flight may have multiple Fares, possibly within the same Class, depending on included services and pricing rules.

1.3.2.3 Conceptual ER Diagram

The final ER diagram visually represents the entities, their attributes, and the relationships described above. It uses Crow's Foot notation to show cardinality and optionality, reflecting the business rules interpreted from the scenario

Figure 1: Conceptual ER Diagram.



1.3.3 Logical Design

We now move to the logical design stage, focusing on normalisation (up to 3NF), relational model schema definition, including necessary junction tables, primary keys selection, the relationship between primary keys and foreign keys, finding and defining the necessary constraints and producing a Relational Schema Diagram.

During the conceptual design stage, natural language was used for naming entities and attributes, while in the logical model, these names were converted to PascalCase, for example, PassengerID and BookingDate), to follow database naming conventions and in preparation for implementation.

Moreover, the Employee entity's Password attribute was replaced by PasswordHash to indicate that this attribute refers to a secured version of the password.

1.3.3.1 Junction Tables

A FlightSeat junction table was introduced to represent the availability and assignment of each seat on a specific flight. Although the conceptual model shows a one-to-many relationship between Flight and Seat, in the logical model, the system needs to represent the per-flight status of each seat to allow for seat availability tracking.

1.3.3.2 Enumerated Fields

Several instances of enumerated attributes were identified, including employee roles, reservation status, fare class, meal preferences, flight seat status, seat class and baggage status, where the possible values the attribute can take are limited to a discrete set of options. These could be modelled using separate lookup tables linked via primary and foreign key relationships. However, given that none of these attributes require additional metadata fields or reuse on multiple areas of the system, for simplicity of design, we chose to use check constraints to enforce consistency and data validity instead of using relational tables. Table 2 below presents the possible values these enumerated fields can take.

Table 2: Enumerated Fields.

Field	Values
Employee Role	Ticketing Staff, Ticketing Supervisor
Reservation Status	Confirmed, Pending, Cancelled
Fare Class	Economy, Business, First
Seat Class	Economy, Business, First
Reservation Meal Preference	Vegetarian, Non-Vegetarian (Default)
Flight Seat Status	Available, Reserved
Baggage Status	Checked-In, Loaded

1.3.3.3 Logical Entity Relationships

During this stage, the first goal was to identify and assign primary keys for each entity in the logical model, considering that the primary key needs to be an attribute or set of attributes that uniquely identify all the entity's records, not change over time and use as few attributes as possible. Some candidates appeared to be natural keys at first sight, such as the Passenger's email, but upon further consideration, they could be changed over time. Similarly, a PNR seemed to be a good candidate to uniquely identify a Reservation, however, it might follow a user-facing external format that could change over time. For this reason, we introduced surrogate keys in all entities, such as PassengerID and FlightID, to store internally generated incremental values to guarantee the Primary Keys uniqueness and Schema's referential integrity.

Additionally, foreign keys were added across the model to represent relationships between entities, such as linking a Reservation to a Passenger and a Flight, or a Baggage record with a Ticket. In the particular case of the relationship between the junction table FlightSeat and the Reservation table, a one-to-one cardinality is established by defining foreign keys in both directions, so each table contains a

foreign key referencing the primary key of the other, creating a circular reference that allows mutual connection between a reservation and the specific seat assigned to it.

Figure 2 presents the logical (relational) schema derived from the conceptual design. Each table is listed along with its attributes. Attributes that are the primary key are shown with underlining, while foreign key attributes are shown in bold and marked with an asterisk (*).

Figure 2: Logical Entity Relationships.

Passenger (PassengerID, FirstName, LastName, Email, DateOfBirth, EmergencyContactNumber)

Flight (FlightID, FlightNumber, OriginAirport, DestinationAirport, DepartureTime, ArrivalTime)

Employee (EmployeeID, Name, Email, Username, PasswordHash, Role)

Seat (SeatID, SeatNumber, SeatClass)

Reservation (ReservationID, BookingDate, Status, PNR, MealUpgradeChoice, MealPreference, **PassengerID***, **FlightID***, **FareID***, **FlightSeatID***)

Ticket (TicketID, IssueDate, EboardingNumber, **ReservationID***, **EmployeeID***)

Baggage (BaggageID, Weight, Cost, Status, **ReservationID***)

Fare (FareID, FareClass, BasePrice, SeatSelectionFee, ExcessBaggageFeeKg, MealUpgradeFee, BaggagePieceFee, **FlightID***)

FlightSeat (FlightSeatID, Status, **FlightID***, **SeatID***, **ReservationID***)

1.3.3.4 Normalisation

All entities in the logical model were evaluated and confirmed to be in Third Normal Form (3NF), understanding that each normal form builds upon the previous. All tables satisfy First Normal Form (1NF) because their attributes hold atomic values, meaning each field contains only one item of data. For example, the DepartureTime attribute in Flight holds a single date-time value. There are no repeating groups, such as multiple emails in a single column.

In terms of Second Normal Form (2NF), none of the tables use composite primary keys, which means it is not possible for partial dependencies to exist. All non-key attributes depend fully on the single-attribute primary key of their respective table. For instance, in the Fare entity, attributes like SeatSelectionFee and BaggagePieceFee depend fully on FareID.

Lastly, all entities meet the requirements for the Third Normal Form (3NF) by ensuring that there are no cases where a non-key attribute depends on another non-key attribute. Each attribute depends directly on the table's primary key. For example, in the Employee entity, the role is not derived from or dependent on email or username but is stored independently.

During the normalisation stage, we found attributes such as phone numbers, email addresses, place names, and employee names, which could technically be decomposed further into more atomic components. For instance, a phone number could be decomposed into country code, area code and local number. However, the scenario does not require specific operations involving these components, and all are treated as single units within the system. Consequently, we chose to store them as atomic values. For instance, origin and destination are stored directly in the Flight entity without normalising them into a separate Airport table, and employee names are stored as full names rather than separated into components.

1.3.3.5 Data Types

At this stage, we identified suitable data types for all attributes in the logical model based on the nature of the data, considering general instead of platform-specific choices and without concern for storage cost. For example, identifiers such as PassengerID and ReservationID were defined as integers and textual attributes, not required to be involved in calculations, such as names, emails, and seat numbers, were defined as variable-length strings, to allow for flexible input. Moreover, dates of birth and booking dates were modelled as calendar dates, while attributes like DepartureTime and ArrivalTime were considered to include date and time. For all monetary values, including base fares and fees, fixed-point decimal types were considered to ensure the accuracy of financial calculations.

1.3.4 Physical Design

Before translating the schema into SQL code, we reviewed the physical design of the database, adapting the logical model to required system-level details, aligning with SQL Server. Entities' generic data types and the identified constraints were translated and expressed using T-SQL data types. Identifiers such as PassengerID, FlightID, and ReservationID were defined as INT with the IDENTITY property to allow automatic generation of unique values. Text fields were declared as NVARCHAR. Date fields such as BookingDate and DateOfBirth were stored as DATE, and DepartureTime and ArrivalTime were stored as DATETIME. DECIMAL(6,2) was used for all monetary values. Field sizes and types were adjusted based on the requirements of the data to be held, balancing usability and storage cost. For instance, NVARCHAR(50) for names and roles, NVARCHAR(100) for emails, and NVARCHAR(255) for hashed passwords.

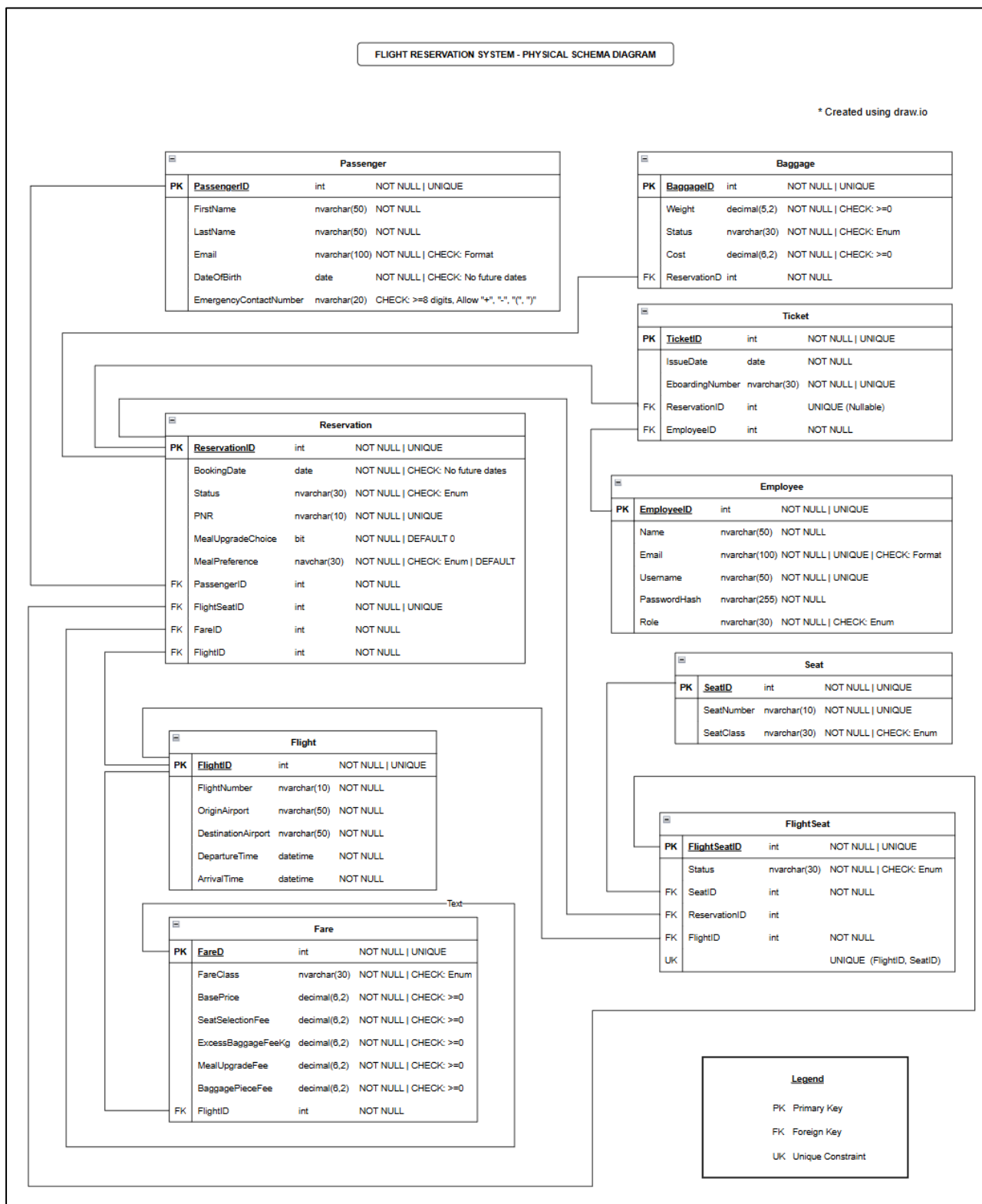
Additionally, several physical design features were considered during this stage, but were found beyond the scope of the current system development iteration and scale. For instance, the system uses the default SQL Server Management Studio indexing setting that creates indexes only in primary and foreign keys and UNIQUE constraints. If in the future, the system is required to provide heavy reporting based on non-key columns, manual indices could be implemented to speed up queries, particularly involving columns with high cardinality, string pattern searches and filtering by date.

Similarly, if the amount of data grows significantly, table partitioning could be considered to improve query performance by allowing SQL Server to scan only the relevant partition. Additionally, denormalisation may be used in the future to improve read performance, for instance by storing passenger names directly in the reservation table to reduce joins. Moreover, if multilingual deployment is required in the future, the default SQL Server collation setting the system currently uses would need to be adjusted to support sorting of names or special character handling appropriately. Lastly, Tools such as SQL Server's execution plan visualiser could be used to analyse slow queries and improve their performance, for instance, rewriting joins, enhancing query performance beyond what the SQL Server internal query

optimisation mechanisms can do. However, the current implementation only involves basic queries on small tables, so no inefficiencies were anticipated.

All the decisions made throughout the logical and physical design stages are represented in the Physical Relational Schema Diagram in Figure 3, which portrays the complete structure of the database.

Figure 3: Physical Schema Diagram.



1.3.5 Implementation

Based on the requirements gathered from the task's scenario and the decisions taken during the conceptual, logical, and physical design stages, we continued by performing the practical implementation, using T-SQL within SQL Server Management Studio, of the database schema, population of data, enforcement of constraints and business rules, implementation of security measures, transaction and concurrency control, as well as all required database objects such as stored procedures, functions, views, and triggers.

1.3.5.1 *Data Integrity and Concurrency*

In the database solution, several mechanisms were implemented to preserve data integrity. For example, the use of a relational schema with primary and foreign key constraints ensured relational consistency. Surrogate primary keys were implemented using INT PRIMARY KEY IDENTITY, and foreign keys were declared using FOREIGN KEY REFERENCES during the table creation process.

Moreover, the default SQL Server setting that restricts delete operations was maintained, preventing cascade deletion of parent rows if there are child rows that reference them, avoiding orphan records, such as a ticket without a corresponding employee. Furthermore, triggers were implemented, for instance, a BEFORE INSERT and BEFORE UPDATE trigger to ensure consistency of the seat allotments, automatically updating their status to Reserved when the ticket is issued.

Additionally, several attribute-level constraints were implemented, including NOT NULL constraints for required fields, preventing the absence of critical data such as names and booking dates. Similarly, CHECK constraints were used to validate input formats. For instance, numeric and monetary fields such as Weight and SeatSelectionFee, respectively, were restricted to accept only non-negative values. Email fields were validated using LIKE and NOT LIKE with pattern matching to ensure they include an @ symbol and a period and avoiding whitespace. Phone numbers, while not enforced to follow a strict format as passengers may be from

different countries, were validated to contain at least eight characters and allow digits, hyphens, and + signs. More complex input data format validation was expected to be handled at the application level. Similarly, constraints were also used to enforce domain-specific and business logic, such as not allowing reservations with past dates and, in the junction table SeatFlight, enforcing UNIQUE Seat and Flight pairs, explicitly allowing NULL ReservationID values for available seats not allocated to any reservation.

In addition, the enumerated fields identified during the logical design, such as Role, FareClass, and FlightSeatStatus, were constrained to only accept values from a set of predefined values and UNIQUE constraints were used for fields to be expected to be unique, such as PNR and to enforce one-to-one cardinality such as in the ReservationID foreign key in the Ticket table to prevent the creation of more than one ticket for the same reservation.

Similarly, through the implementation, atomicity was enforced and inconsistent data resulting from truncated transactions was prevented using transaction control with BEGIN TRANSACTION, COMMIT and ROLLBACK within TRY CATCH exception handling blocks. This approach was used in CRUD transactions, other than read operations, involving multiple tables or requiring conditional validation such as in the ReserveSeatForReservation stored procedure, involving a conditional validation to determine if the seat is still available and then making changes in both FlightSeat and Reservation tables, preventing partial updates and avoiding inconsistent database state resulting from only one of these tables being modified.

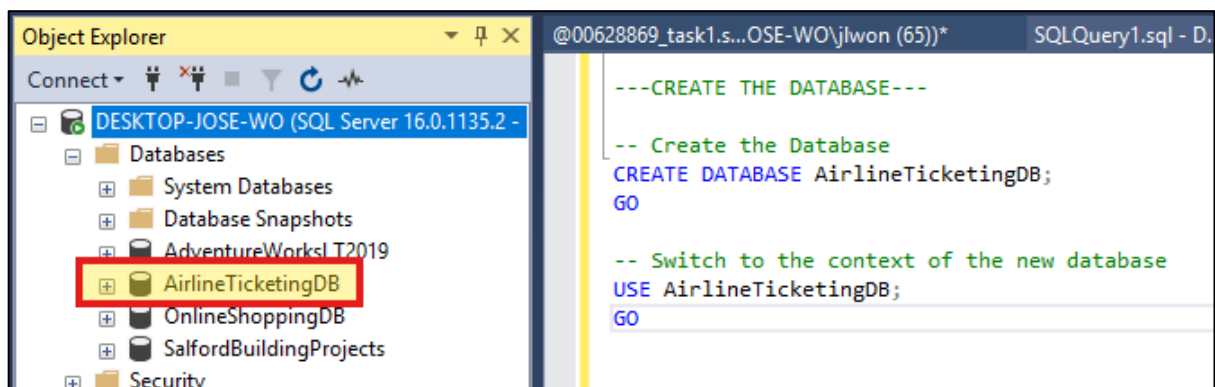
Furthermore, default SQL server transaction and concurrency control was reinforced in scenarios where multiple users could attempt to modify the same data simultaneously, particularly when involving sensitive data such as seat assignments, where the same row has to be first validated for availability and then updated, creating the risk of other transactions modifying the row in between. To achieve this, we used the SERIALIZABLE isolation level, which is stricter than the default SQL server READ COMMIT isolation level which only prevents reading rows while being updated by another transaction but does not protect from updates made by another transaction happening in between when a transaction needs to read first and then update the row. Instead, the SERIALIZABLE isolation level does not allow any other

transaction to insert, update, or even read rows while a transaction is running. To even further ensure transaction atomicity and data consistency in these cases, explicit row-level locking was implemented through WITH (UPDLOCK, HOLDLOCK), for instance, to lock the row corresponding to the seat being reserved, enhancing the default SQL Server's dynamic selection of lock granularity level.

1.3.5.2 Creating the Database

The implementation began with the creation of the AirlineTicketingDB database using the CREATE DATABASE statement (Figure 4).

Figure 4: Creating AirlineTicketingDB Database.



1.3.5.3 Creating the Tables

All tables were created using the CREATE TABLE clause, following the structure and data types defined during the physical design stage and including the elements identified in the Data Integrity and Concurrency section. Figures 5 to 13 below present the implementation of each of the tables.

Figure 5: Table Passenger.

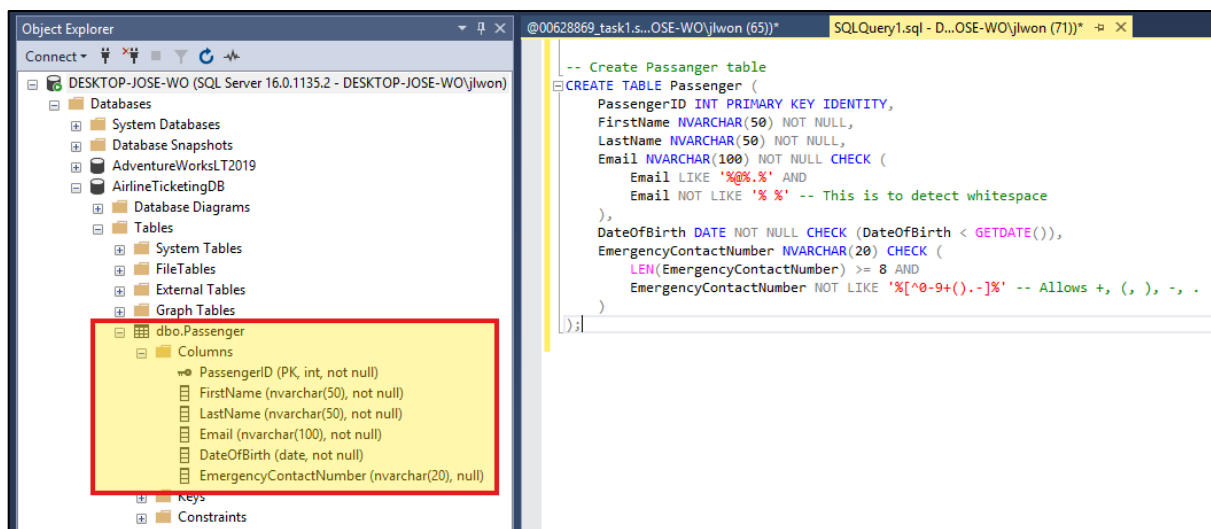


Figure 6: Table Flight.

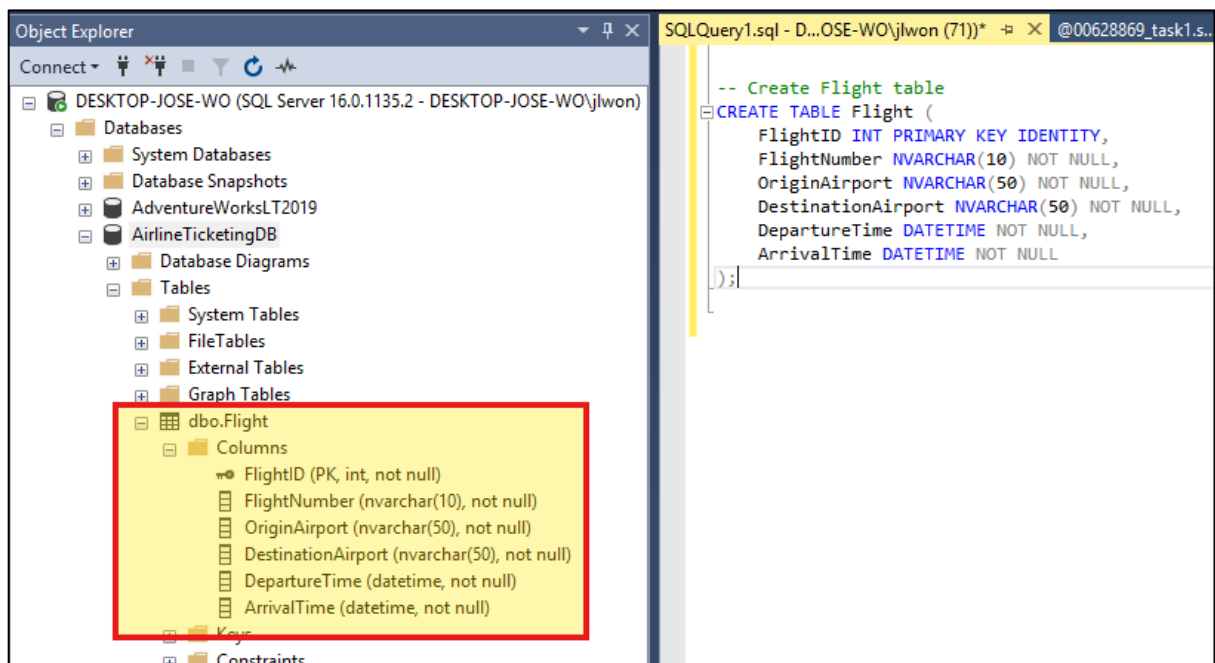


Figure 7: Table Employee.

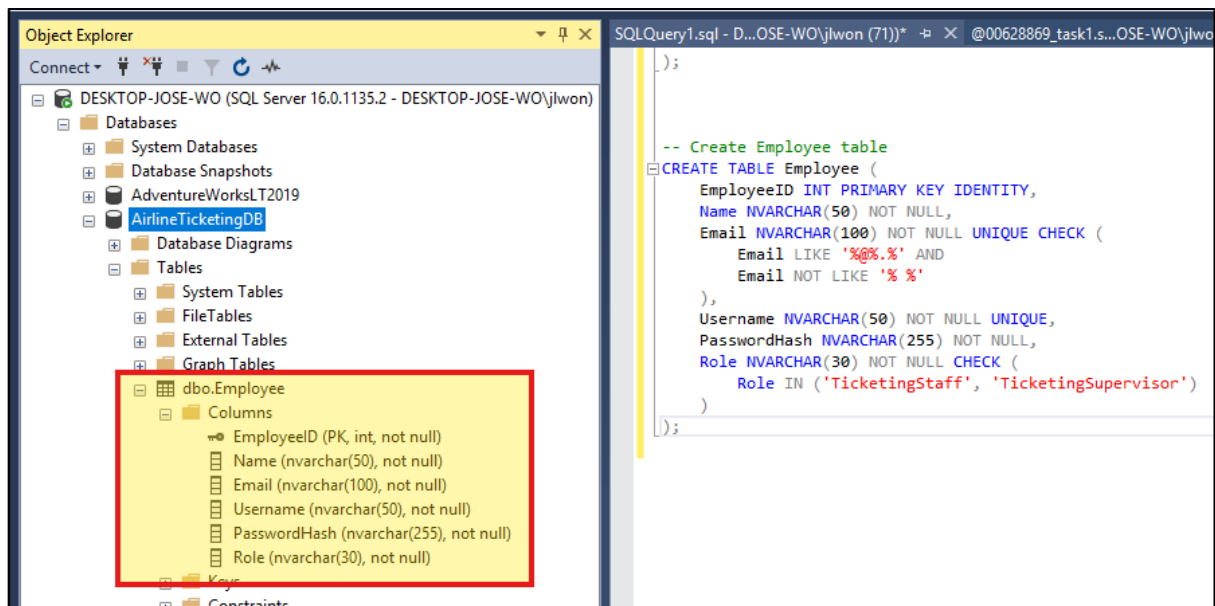


Figure 8: Table Seat.

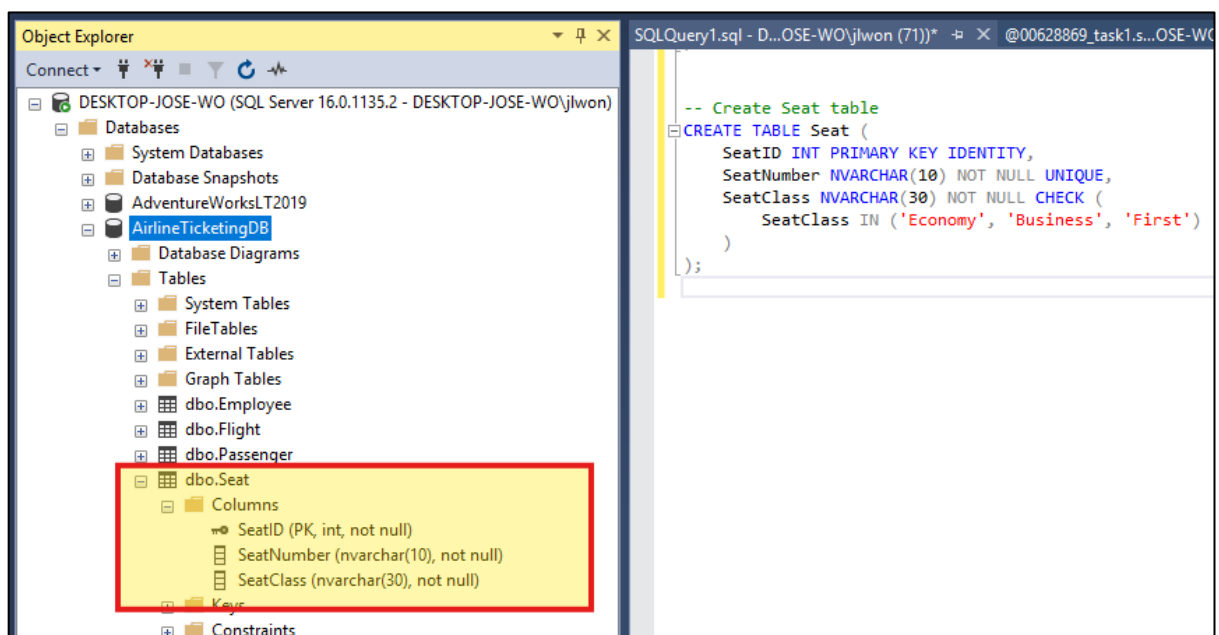


Figure 9: Table Fare.

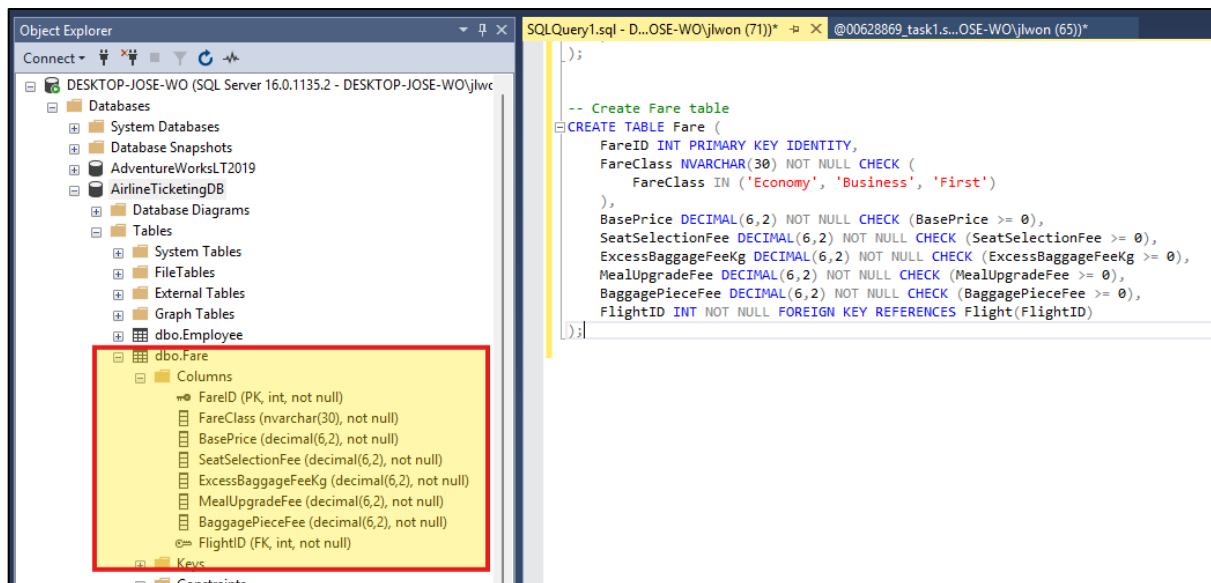


Figure 10: Table FlightSeat.

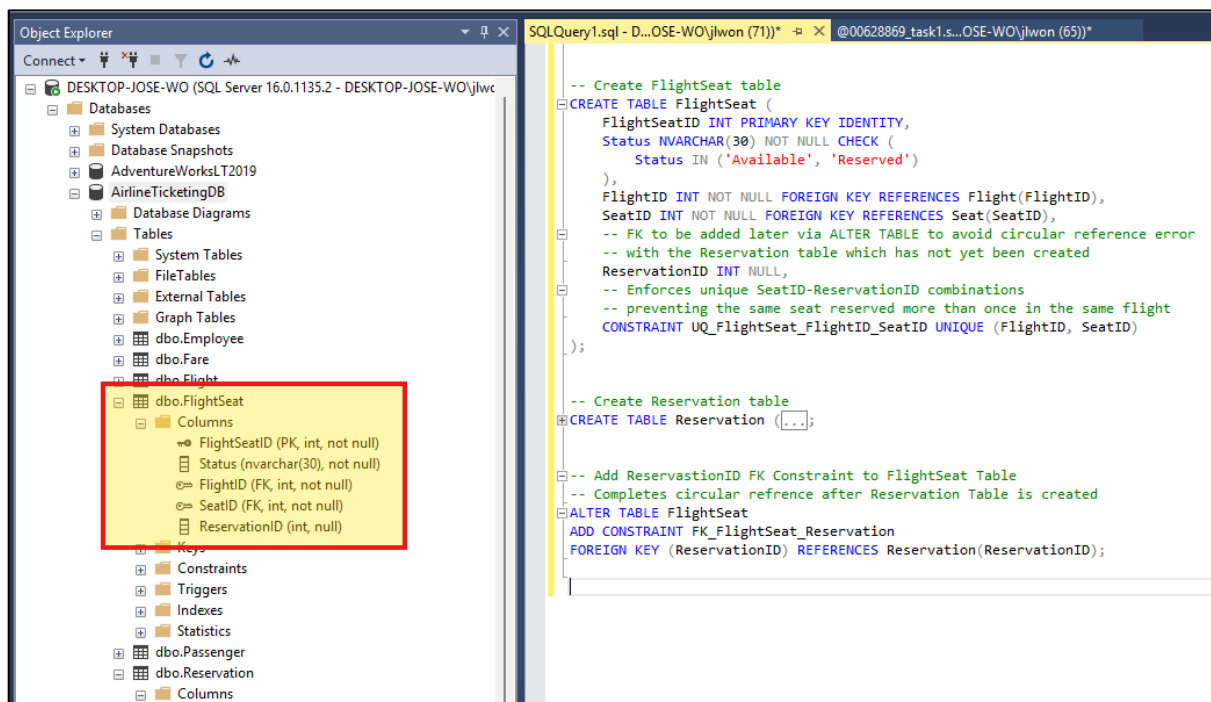


Figure 11: Table Reservation.

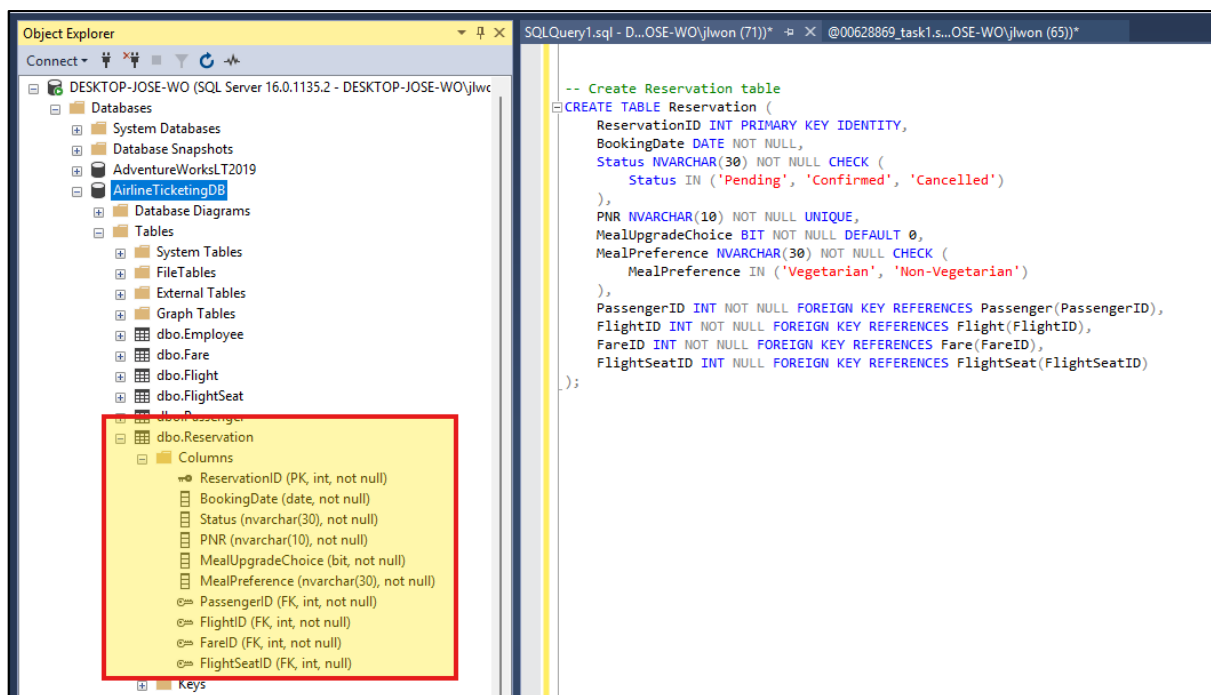


Figure 12: Table Ticket.

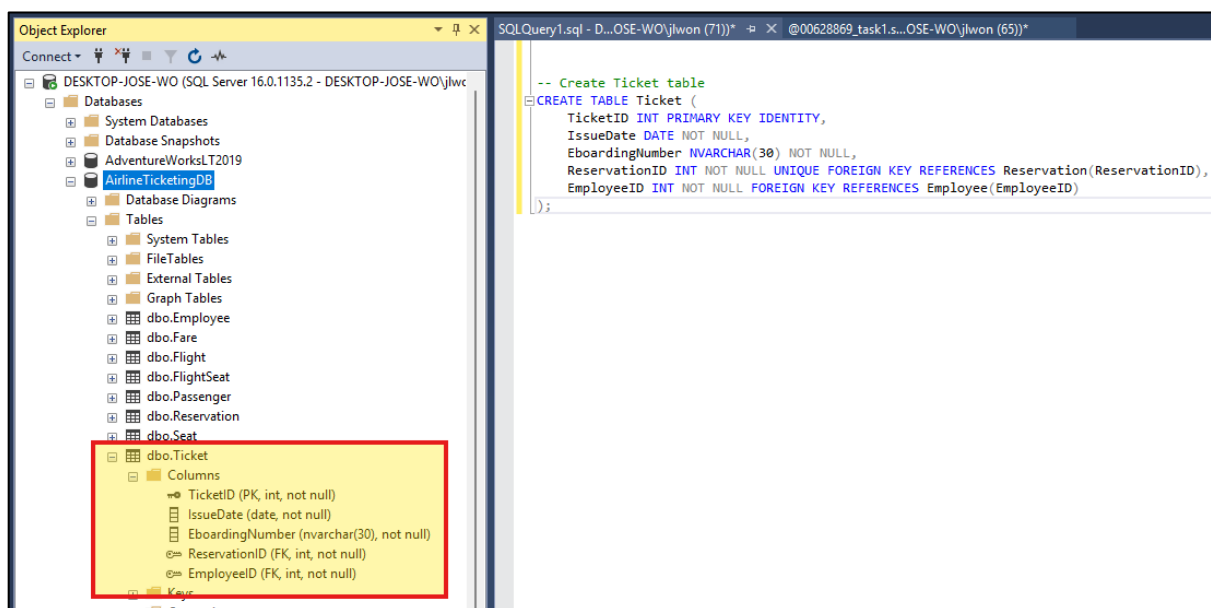
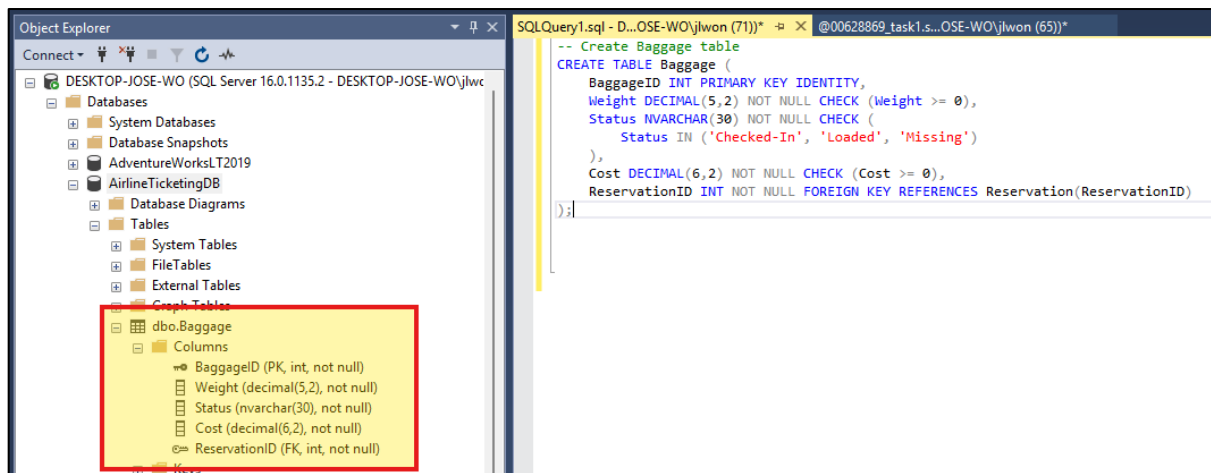


Figure 13: Table Baggage.



1.3.5.4 Populating the Tables

To populate the database with data, samples were inserted into each table using the INSERT INTO statement, specifying the column names and grouped VALUES clauses, as shown in Figures 14 to 23. Due to the junction table SeatFlight and the Reservation table having a circular reference, as each contains a foreign key linking to the primary key of the other, the SeatFlight table was first populated with entries containing NULL ReservationIDs, a process that realistically reflect the state of seat allocation when an airline just schedules a flight, and then ReservationIDs were updated in the SeatFlight table after the Reservation table was populated.

Figure 14: Populating Passenger Table.

```
v2.SQLQuery1.sql...OSE-WO\jlwon (71))* -p X @00628869_task1.s...OSE-WO\jlwon (65))*

-- Insert Data in Passenger table
BEGIN TRY
    BEGIN TRANSACTION;

    INSERT INTO Passenger (FirstName, LastName, Email, DateOfBirth, EmergencyContactNumber)
    VALUES
    ('Aisha', 'Khan', 'aisha.khan@mail.com', '1980-05-22', '+44-7700-900111'),
    ('Carlos', 'Ramirez', 'c.ramirez@yahoo.com', '1992-11-10', '+34-600-123456'),
    ('Zhang', 'Wei', 'zhang.wei@GmAIL.com', '2001-03-15', NULL),
    ('Liam', 'O'Connor', 'liam.oconnor@mail.nhs.com', '1983-07-09', '+353-850-112233'),
    ('Fatima', 'Al-Mansouri', 'fatima.m@un.org', '1998-12-01', NULL),
    ('Emily', 'Nguyen', 'emily.nguyen@msn.com', '2004-09-27', '+1-310-555-7890'),
    ('Rajesh', 'Kumar', 'rajesh.kumar@hotmail.com', '1990-02-18', NULL),
    ('Sophia', 'Müller', 's.muller@worldbank-HHRR.com', '1995-06-30', '+49-151-23456789'),
    ('James', 'Anderson', 'james.a@123.com', '2002-04-14', NULL),
    ('Hassan', 'Youssef', 'h.youssef@this.is.a.very.long.domain.with.many.subdomains.test', '1970-08-03', '+20-101-2223334');

    COMMIT TRANSACTION;
    PRINT 'Passenger data was inserted successfully.';
END TRY
BEGIN CATCH
    ROLLBACK TRANSACTION;
    PRINT 'An error has occurred.';
    PRINT ERROR_MESSAGE();
    DBCC CHECKIDENT ('Passenger', RESEED, 0);
END CATCH;
```

Figure 15: Populating Flight Table.

```
v2.SQLQuery1.sql...OSE-WO\jlwon (71))* -p X @00628869_task1.s...OSE-WO\jlwon (65))*

-- Insert Data into Flight table
BEGIN TRY
    BEGIN TRANSACTION;

    INSERT INTO Flight (FlightNumber, OriginAirport, DestinationAirport, DepartureTime, ArrivalTime)
    VALUES
    ('BA123', 'London', 'Brasilia', '2025-04-18 10:30', '2025-04-18 13:00'),
    ('AF456', 'Paris', 'Osaka', '2025-04-19 14:00', '2025-04-20 08:30'),
    ('DL789', 'New York', 'Denver', '2025-04-20 09:15', '2025-04-20 12:30'),
    ('EK202', 'Dubai', 'Sydney', '2025-04-21 22:00', '2025-04-22 16:00'),
    ('LH330', 'Frankfurt', 'Johannesburg', '2025-04-22 13:45', '2025-04-22 22:05'),
    ('AI101', 'Delhi', 'London', '2025-04-23 02:00', '2025-04-23 06:30'),
    ('CX999', 'Hong Kong', 'San Francisco', '2025-04-24 23:45', '2025-04-24 20:10'),
    ('QF302', 'Melbourne', 'Singapore', '2025-04-25 07:00', '2025-04-25 13:00'),
    ('KL888', 'Amsterdam', 'Toronto', '2025-04-26 15:30', '2025-04-26 18:00'),
    ('AZ321', 'Rome', 'Cairo', '2025-04-27 11:20', '2025-04-27 15:10');

    COMMIT TRANSACTION;
    PRINT 'Flight data was inserted successfully.';
END TRY
BEGIN CATCH
    ROLLBACK TRANSACTION;
    PRINT 'An error has occurred.';
    PRINT ERROR_MESSAGE();
    DBCC CHECKIDENT ('Flight', RESEED, 0);
END CATCH;
```

Figure 16: Populating Employee Table.

```

v2.SQLQuery1.sql...OSE-WO\jlwon (71))* X @00628869_task1.s...OSE-WO\jlwon (65))*

-- Insert Data into Employee table
BEGIN TRY
    BEGIN TRANSACTION;

    INSERT INTO Employee (Name, Email, Username, PasswordHash, Role)
    VALUES
    ('Ana María López', 'ana.lopez@emirates-HHRR.com', 'ana_lopez',
    '2a1c9edfea4f7b6a5d9c57a18e3291bc4f94de973f89bc801e4ab12e2a6fa5b3', 'TicketingStaff'),
    ('David Smith', 'david.smith@emirates.bookings.com', 'dsmith', 'pw_456', 'TicketingSupervisor'),
    ('Mei Ling', 'mei.ling@EmiRates.com', 'mei_ling88', 'pw_mei_ling', 'TicketingStaff'),
    ('Ahmed El-Sayed', 'ahmed.sayed@emirates.ticketing.complaints.com', 'ahmed_s', 'ahmed_pass', 'TicketingStaff'),
    ('Emma Johansson', 'emma.j@emirates.com', 'emma_j', 'emma_pw', 'TicketingSupervisor'),
    ('Carlos Mendes', 'c.mendes@emirates.com', 'cmendes77', 'carlos_pw', 'TicketingStaff'),
    ('Nobuo Tanaka', 'n.tanaka@emirates.com', 'nobuo_t', 'tanaka_pw', 'TicketingStaff'),
    ('Fatima Zahra', 'fatima.z@emirates.com', 'fzahra', 'fzahra_pw', 'TicketingSupervisor'),
    ('John O'Reilly', 'john.oreilly@emirates.com', 'john_o', 'john_pw123', 'TicketingStaff'),
    ('Sara Ibrahim', 'sara.ibrahim@emirates.com', 'sara_ibrahim', 'sara_pw', 'TicketingStaff');

    COMMIT TRANSACTION;
    PRINT 'Employee data was inserted successfully.';
END TRY
BEGIN CATCH
    ROLLBACK TRANSACTION;
    PRINT 'An error has occurred: ';
    PRINT ERROR_MESSAGE();
    DBCC CHECKIDENT ('Employee', RESEED, 0);
END CATCH;

```

Figure 17: Populating Seat Table.

```

v2.SQLQuery1.sql...OSE-WO\jlwon (71))* X @00628869_task1.s...OSE-WO\jlwon (65))*

-- Insert Data into Seat table
BEGIN TRY
    BEGIN TRANSACTION;

    INSERT INTO Seat (SeatNumber, SeatClass)
    VALUES
    ('E1', 'Economy'), ('E2', 'Economy'), ('E3', 'Economy'), ('E4', 'Economy'), ('E5', 'Economy'),
    ('E6', 'Economy'), ('E7', 'Economy'), ('E8', 'Economy'), ('E9', 'Economy'), ('E10', 'Economy'),
    ('B1', 'Business'), ('B2', 'Business'), ('B3', 'Business'), ('B4', 'Business'), ('B5', 'Business'),
    ('F1', 'First'), ('F2', 'First'), ('F3', 'First'), ('F4', 'First'), ('F5', 'First');

    COMMIT TRANSACTION;
    PRINT 'Seat data was inserted successfully.';
END TRY
BEGIN CATCH
    ROLLBACK TRANSACTION;
    PRINT 'An error has occurred: ';
    PRINT ERROR_MESSAGE();
    DBCC CHECKIDENT ('Seat', RESEED, 0);
END CATCH;

```

Figure 18: Populating Fare Table.

```
v2.SQLQuery1.sql...OSE-WO\jlwon (71))* X @00628869_task1.s...OSE-WO\jlwon (65))*

-- Insert Data into Fare table
-- All fares follow the same additional services fee structure specified in the scenario
BEGIN TRY
    BEGIN TRANSACTION;

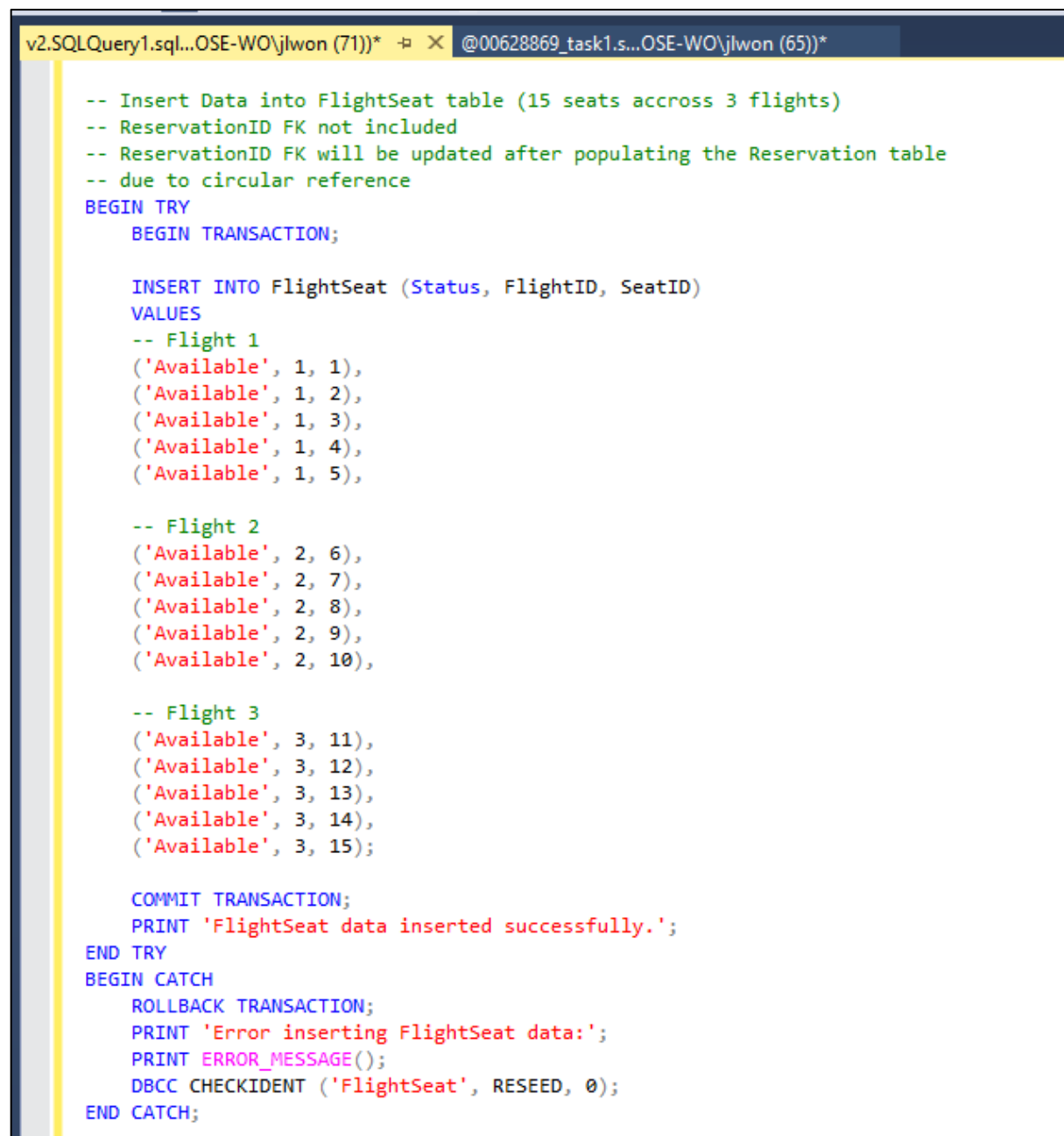
    INSERT INTO Fare (FareClass, BasePrice, SeatSelectionFee,
        ExcessBaggageFeeKg, MealUpgradeFee, BaggagePieceFee, FlightID)
    VALUES
        -- Fares for FlightID 1
        ('Economy', 150.00, 30.00, 100.00, 20.00, 100.00, 1),
        ('Business', 400.00, 30.00, 100.00, 20.00, 100.00, 1),
        ('First', 800.00, 30.00, 100.00, 20.00, 100.00, 1),

        -- Fares for FlightID 2
        ('Economy', 160.00, 30.00, 100.00, 20.00, 100.00, 2),
        ('Business', 420.00, 30.00, 100.00, 20.00, 100.00, 2),
        ('First', 850.00, 30.00, 100.00, 20.00, 100.00, 2),

        -- Fares for FlightID 3
        ('Economy', 140.00, 30.00, 100.00, 20.00, 100.00, 3),
        ('Business', 410.00, 30.00, 100.00, 20.00, 100.00, 3),
        ('First', 780.00, 30.00, 100.00, 20.00, 100.00, 3),

        -- more fares continue (truncated for the screenshot only)

    COMMIT TRANSACTION;
    PRINT 'Fare data was inserted successfully.';
END TRY
BEGIN CATCH
    ROLLBACK TRANSACTION;
    PRINT 'An error has occurred: ';
    PRINT ERROR_MESSAGE();
    DBCC CHECKIDENT ('Fare', RESEED, 0);
END CATCH;
```

Figure 19: Populating FlightSeat Table.

```
v2.SQLQuery1.sql...OSE-WO\jlwon (71))* X @00628869_task1.s...OSE-WO\jlwon (65))*  
  
-- Insert Data into FlightSeat table (15 seats accross 3 flights)  
-- ReservationID FK not included  
-- ReservationID FK will be updated after populating the Reservation table  
-- due to circular reference  
BEGIN TRY  
    BEGIN TRANSACTION;  
  
    INSERT INTO FlightSeat (Status, FlightID, SeatID)  
    VALUES  
        -- Flight 1  
        ('Available', 1, 1),  
        ('Available', 1, 2),  
        ('Available', 1, 3),  
        ('Available', 1, 4),  
        ('Available', 1, 5),  
  
        -- Flight 2  
        ('Available', 2, 6),  
        ('Available', 2, 7),  
        ('Available', 2, 8),  
        ('Available', 2, 9),  
        ('Available', 2, 10),  
  
        -- Flight 3  
        ('Available', 3, 11),  
        ('Available', 3, 12),  
        ('Available', 3, 13),  
        ('Available', 3, 14),  
        ('Available', 3, 15);  
  
    COMMIT TRANSACTION;  
    PRINT 'FlightSeat data inserted successfully.';  
END TRY  
BEGIN CATCH  
    ROLLBACK TRANSACTION;  
    PRINT 'Error inserting FlightSeat data:';  
    PRINT ERROR_MESSAGE();  
    DBCC CHECKIDENT ('FlightSeat', RESEED, 0);  
END CATCH;
```

Figure 20: Populating Reservation Table.

```

v2.SQLQuery1.sql...OSE-WO\jlwon (71))*  X  @00628869_task1.s...OSE-WO\jlwon (65))*
-- Insert data into Reservation table
BEGIN TRY
    BEGIN TRANSACTION;

    INSERT INTO Reservation (
        BookingDate, Status, PNR, MealUpgradeChoice, MealPreference,
        PassengerID, FlightID, FareID, FlightSeatID
    )
    VALUES
    ('2025-04-19', 'Confirmed', 'PNR001', 1, 'Vegetarian', 1, 1, 1, 1),
    ('2025-04-19', 'Pending', 'PNR002', 0, 'Non-Vegetarian', 2, 1, 1, 2),
    ('2025-04-20', 'Confirmed', 'PNR003', 1, 'Vegetarian', 3, 2, 2, 6),
    ('2025-04-21', 'Confirmed', 'PNR004', 0, 'Vegetarian', 4, 2, 2, 7),
    ('2025-04-21', 'Pending', 'PNR005', 1, 'Non-Vegetarian', 5, 2, 2, 8),
    ('2025-04-22', 'Confirmed', 'PNR006', 0, 'Vegetarian', 6, 3, 3, 11),
    ('2025-04-23', 'Confirmed', 'PNR007', 1, 'Non-Vegetarian', 7, 3, 3, 12),
    ('2025-04-23', 'Pending', 'PNR008', 1, 'Vegetarian', 8, 3, 3, 13),
    ('2025-04-24', 'Confirmed', 'PNR009', 0, 'Non-Vegetarian', 9, 1, 1, 3),
    ('2025-04-25', 'Confirmed', 'PNR010', 1, 'Vegetarian', 10, 1, 1, 4);

    COMMIT TRANSACTION;
    PRINT 'Reservation data inserted successfully.';
END TRY
BEGIN CATCH
    ROLLBACK TRANSACTION;
    PRINT 'Error inserting Reservation data:';
    PRINT ERROR_MESSAGE();
    DBCC CHECKIDENT ('Reservation', RESEED, 0);
END CATCH;

```

Figure 21: Updating FlightSeat Table with ReservationID Foreign Keys.

```

v2.SQLQuery1.sql...OSE-WO\jlwon (71))*  X  @00628869_task1.s...OSE-WO\jlwon (65))*

-- Updating ReservationID FK in FlightSeat junction table after creating the reservations
BEGIN TRY
    BEGIN TRANSACTION;

    UPDATE FlightSeat SET ReservationID = 1 WHERE FlightSeatID = 1;
    UPDATE FlightSeat SET ReservationID = 2 WHERE FlightSeatID = 2;
    UPDATE FlightSeat SET ReservationID = 3 WHERE FlightSeatID = 6;
    UPDATE FlightSeat SET ReservationID = 4 WHERE FlightSeatID = 7;
    UPDATE FlightSeat SET ReservationID = 5 WHERE FlightSeatID = 8;
    UPDATE FlightSeat SET ReservationID = 6 WHERE FlightSeatID = 11;
    UPDATE FlightSeat SET ReservationID = 7 WHERE FlightSeatID = 12;
    UPDATE FlightSeat SET ReservationID = 8 WHERE FlightSeatID = 13;
    UPDATE FlightSeat SET ReservationID = 9 WHERE FlightSeatID = 3;
    UPDATE FlightSeat SET ReservationID = 10 WHERE FlightSeatID = 4;

    COMMIT TRANSACTION;
    PRINT 'FlightSeat records successfully updated with ReservationID.';
END TRY
BEGIN CATCH
    ROLLBACK TRANSACTION;
    PRINT 'Error updating FlightSeat ReservationID:';
    PRINT ERROR_MESSAGE();
END CATCH;

```

Figure 22: Populating Ticket Table.

```

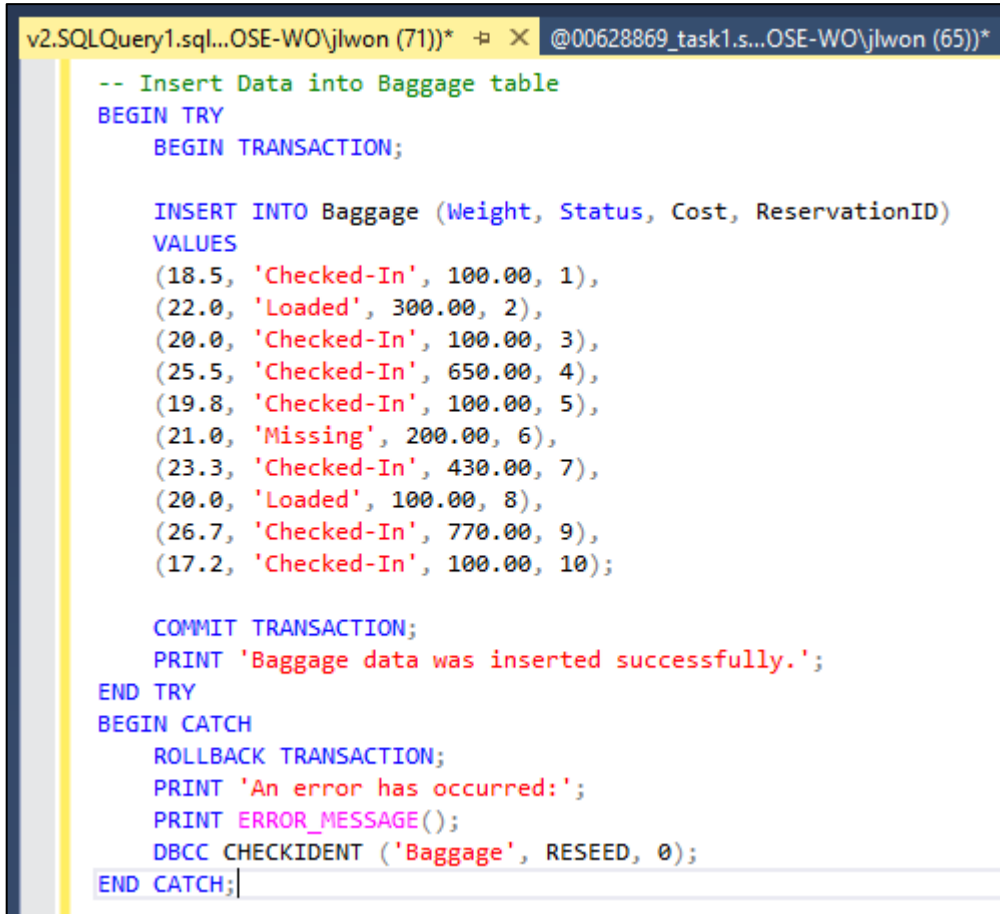
v2.SQLQuery1.sql...OSE-WO\jlwon (71))*  X  @00628869_task1.s...OSE-WO\jlwon (65))*

-- Insert data into Ticket table
BEGIN TRY
    BEGIN TRANSACTION;

    INSERT INTO Ticket (IssueDate, EboardingNumber, ReservationID, EmployeeID)
    VALUES
    ('2025-04-15', 'EBD12345601', 1, 1),
    ('2025-04-15', 'EBD12345602', 2, 2),
    ('2025-04-16', 'EBD12345603', 3, 3),
    ('2025-04-16', 'EBD12345604', 4, 4),
    ('2025-04-17', 'EBD12345605', 5, 5),
    ('2025-04-17', 'EBD12345606', 6, 6),
    ('2025-04-18', 'EBD12345607', 7, 7),
    ('2025-04-18', 'EBD12345608', 8, 8),
    ('2025-04-19', 'EBD12345609', 9, 9),
    ('2025-04-19', 'EBD12345610', 10, 10);

    COMMIT TRANSACTION;
    PRINT 'Ticket data was inserted successfully.';
END TRY
BEGIN CATCH
    ROLLBACK TRANSACTION;
    PRINT 'An error has occurred:';
    PRINT ERROR_MESSAGE();
    DBCC CHECKIDENT ('Ticket', RESEED, 0);
END CATCH;

```

Figure 23: Populating Baggage Table.


```

v2.SQLQuery1.sql...OSE-WO\jlwon (71))* @00628869_task1.s...OSE-WO\jlwon (65))*
-- Insert Data into Baggage table
BEGIN TRY
    BEGIN TRANSACTION;

    INSERT INTO Baggage (Weight, Status, Cost, ReservationID)
    VALUES
    (18.5, 'Checked-In', 100.00, 1),
    (22.0, 'Loaded', 300.00, 2),
    (20.0, 'Checked-In', 100.00, 3),
    (25.5, 'Checked-In', 650.00, 4),
    (19.8, 'Checked-In', 100.00, 5),
    (21.0, 'Missing', 200.00, 6),
    (23.3, 'Checked-In', 430.00, 7),
    (20.0, 'Loaded', 100.00, 8),
    (26.7, 'Checked-In', 770.00, 9),
    (17.2, 'Checked-In', 100.00, 10);

    COMMIT TRANSACTION;
    PRINT 'Baggage data was inserted successfully.';
END TRY
BEGIN CATCH
    ROLLBACK TRANSACTION;
    PRINT 'An error has occurred: ';
    PRINT ERROR_MESSAGE();
    DBCC CHECKIDENT ('Baggage', RESEED, 0);
END CATCH;

```

1.3.5.5 Database Security Implementation

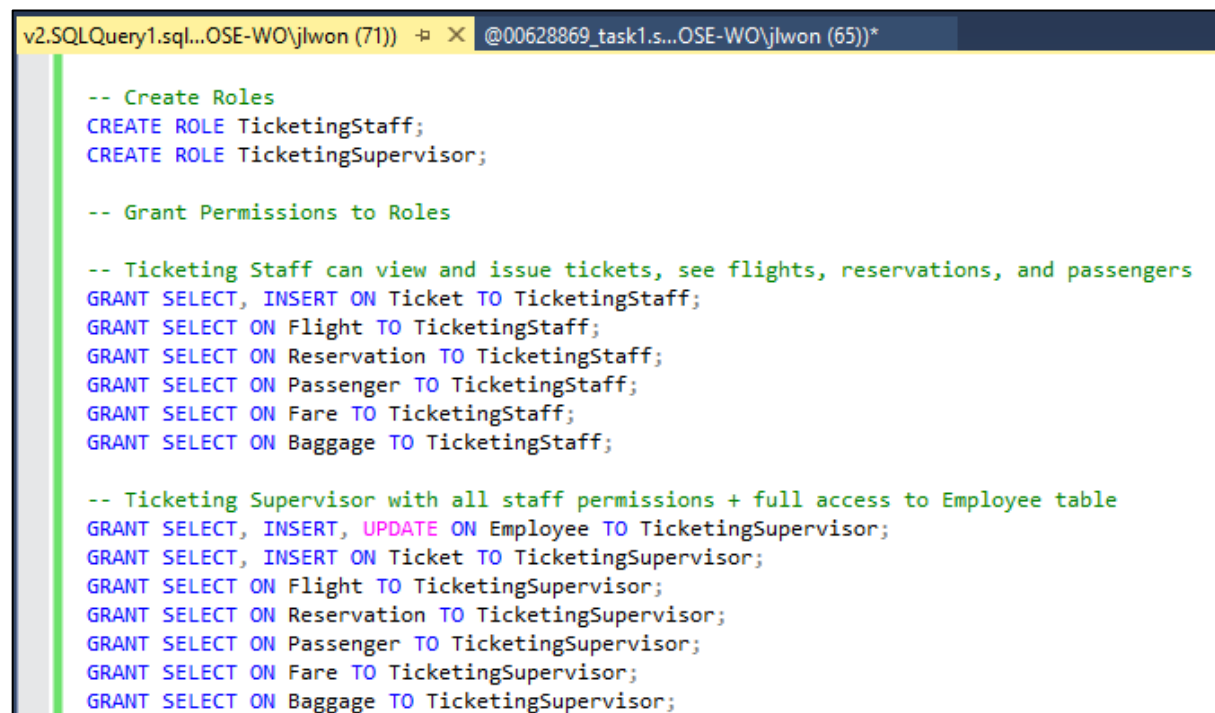
The database implementation benefits from full recovery model, the default setting when a new database is created in SQL Server, enhancing its ability to prevent accidental loss or corruption of data by allowing for point-in-time recovery.

Regarding access control, authentication, roles, authorisations and privileges, in a real-world environment, Ticketing Staff and Ticketing Supervisors would not access the database directly through SQL Server Management Studio. Instead, they would interact with the database through an application interface, in this case a ticketing system, which would authenticate users and apply role-based permissions at the

application level. The application would create a connection to the database using a single user permission while restricting or granting access to different areas of the application and its functionality based on the user's role, retrieved by querying the Employee table. However, for the purpose of this task, SQL logins, users, and roles were implemented directly within SQL Server.

Since the scenario describes that the Ticketing Supervisor intervenes in cases where the Ticketing Staff is not authenticated and that SQL Server requires users to be authenticated to perform any operation, we interpreted this as implying authorisation instead of authentication and assume that the Ticketing Staff should be allowed to perform write operations on ticket data such as inserting new tickets, and read operations such as viewing reservations, and accessing flight and passenger information, while a Supervisor should have higher privileges, such as updating employee records.

Consequently, two database roles TicketingStaff and TicketingSupervisor, were defined using CREATE ROLE, and then permissions were granted with the GRANT clause as portrayed in Figure 24. The TicketingStaff role was allowed to SELECT and INSERT into the Ticket table but had no access to modify employee data, while the TicketingSupervisor role was granted SELECT, INSERT, and UPDATE on the Employee table.

Figure 24: Implement Roles and Privileges.A screenshot of a SQL query window. The title bar shows two tabs: 'v2.SQLQuery1.sql...OSE-WO\jlwon (71))' and '@00628869_task1.s...OSE-WO\jlwon (65))*'. The query text is as follows:

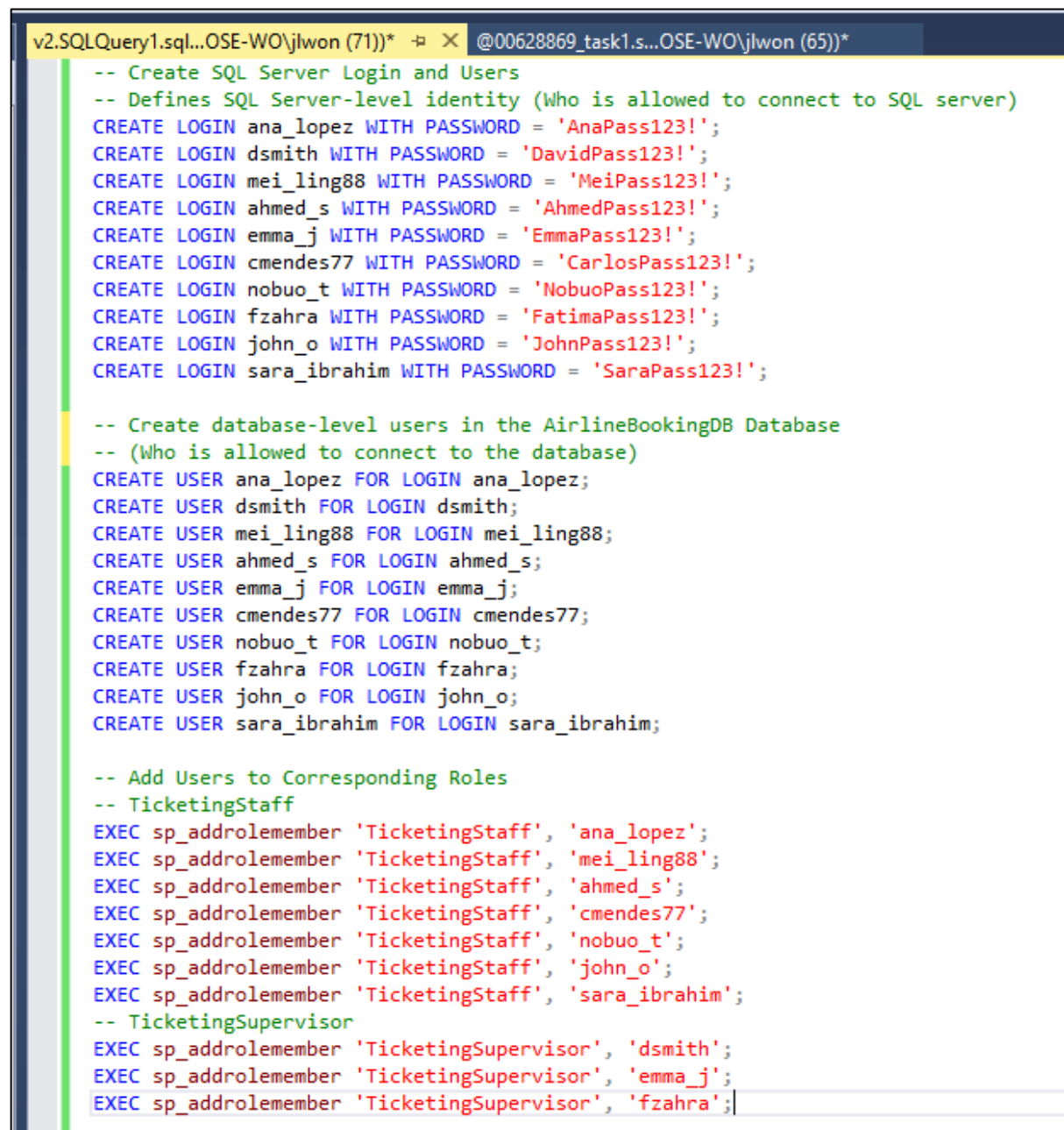
```
-- Create Roles
CREATE ROLE TicketingStaff;
CREATE ROLE TicketingSupervisor;

-- Grant Permissions to Roles

-- Ticketing Staff can view and issue tickets, see flights, reservations, and passengers
GRANT SELECT, INSERT ON Ticket TO TicketingStaff;
GRANT SELECT ON Flight TO TicketingStaff;
GRANT SELECT ON Reservation TO TicketingStaff;
GRANT SELECT ON Passenger TO TicketingStaff;
GRANT SELECT ON Fare TO TicketingStaff;
GRANT SELECT ON Baggage TO TicketingStaff;

-- Ticketing Supervisor with all staff permissions + full access to Employee table
GRANT SELECT, INSERT, UPDATE ON Employee TO TicketingSupervisor;
GRANT SELECT, INSERT ON Ticket TO TicketingSupervisor;
GRANT SELECT ON Flight TO TicketingSupervisor;
GRANT SELECT ON Reservation TO TicketingSupervisor;
GRANT SELECT ON Passenger TO TicketingSupervisor;
GRANT SELECT ON Fare TO TicketingSupervisor;
GRANT SELECT ON Baggage TO TicketingSupervisor;
```

Additionally, as presented in Figure 25, to simulate session access control, each employee listed in the Employee table was assigned a SQL Server login using CREATE LOGIN with a corresponding secure password, followed by CREATE USER FOR LOGIN to associate them with the database. These users were then added to the appropriate role with EXEC sp_addrolemember.

Figure 25: Implementing SQL Server and Database Access.


```

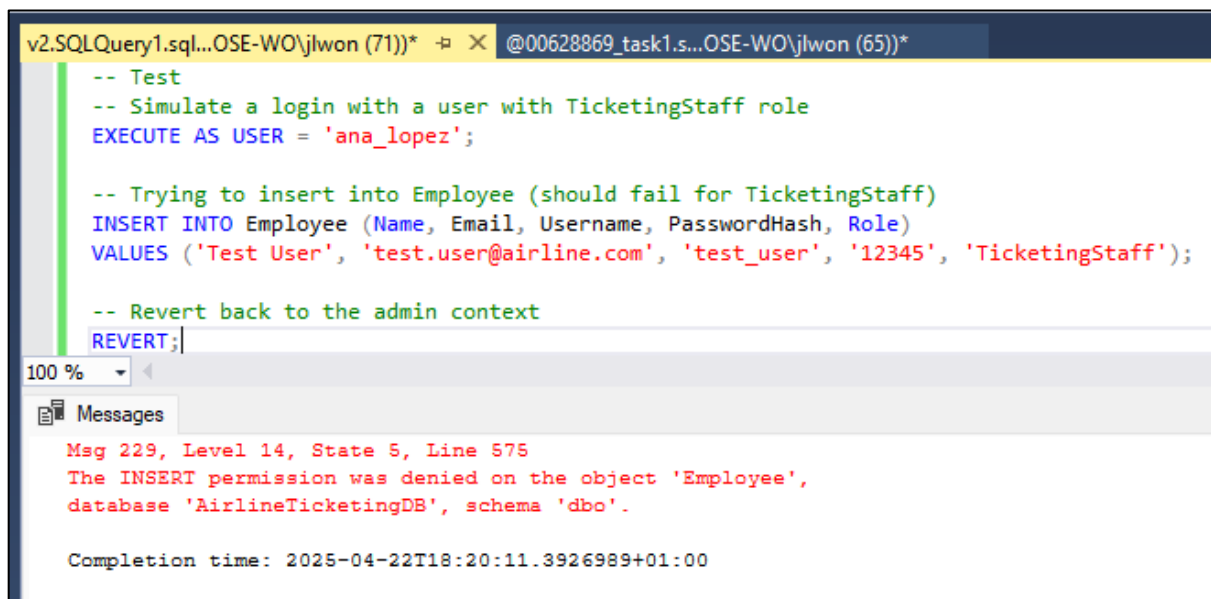
v2.SQLQuery1.sql...OSE-WO\jlwon (71))* X @00628869_task1.s...OSE-WO\jlwon (65))*
-- Create SQL Server Login and Users
-- Defines SQL Server-level identity (Who is allowed to connect to SQL server)
CREATE LOGIN ana_lopez WITH PASSWORD = 'AnaPass123!';
CREATE LOGIN dsmith WITH PASSWORD = 'DavidPass123!';
CREATE LOGIN mei_ling88 WITH PASSWORD = 'MeiPass123!';
CREATE LOGIN ahmed_s WITH PASSWORD = 'AhmedPass123!';
CREATE LOGIN emma_j WITH PASSWORD = 'EmmaPass123!';
CREATE LOGIN cmendes77 WITH PASSWORD = 'CarlosPass123!';
CREATE LOGIN nobuo_t WITH PASSWORD = 'NobuoPass123!';
CREATE LOGIN fzahra WITH PASSWORD = 'FatimaPass123!';
CREATE LOGIN john_o WITH PASSWORD = 'JohnPass123!';
CREATE LOGIN sara_ibrahim WITH PASSWORD = 'SaraPass123!';

-- Create database-level users in the AirlineBookingDB Database
-- (Who is allowed to connect to the database)
CREATE USER ana_lopez FOR LOGIN ana_lopez;
CREATE USER dsmith FOR LOGIN dsmith;
CREATE USER mei_ling88 FOR LOGIN mei_ling88;
CREATE USER ahmed_s FOR LOGIN ahmed_s;
CREATE USER emma_j FOR LOGIN emma_j;
CREATE USER cmendes77 FOR LOGIN cmendes77;
CREATE USER nobuo_t FOR LOGIN nobuo_t;
CREATE USER fzahra FOR LOGIN fzahra;
CREATE USER john_o FOR LOGIN john_o;
CREATE USER sara_ibrahim FOR LOGIN sara_ibrahim;

-- Add Users to Corresponding Roles
-- TicketingStaff
EXEC sp_addrolemember 'TicketingStaff', 'ana_lopez';
EXEC sp_addrolemember 'TicketingStaff', 'mei_ling88';
EXEC sp_addrolemember 'TicketingStaff', 'ahmed_s';
EXEC sp_addrolemember 'TicketingStaff', 'cmendes77';
EXEC sp_addrolemember 'TicketingStaff', 'nobuo_t';
EXEC sp_addrolemember 'TicketingStaff', 'john_o';
EXEC sp_addrolemember 'TicketingStaff', 'sara_ibrahim';
-- TicketingSupervisor
EXEC sp_addrolemember 'TicketingSupervisor', 'dsmith';
EXEC sp_addrolemember 'TicketingSupervisor', 'emma_j';
EXEC sp_addrolemember 'TicketingSupervisor', 'fzahra';

```

To test that the security enforcement works correctly, the command `EXECUTE AS USER = 'ana_lopez'` was used to simulate a login as a Ticketing Staff member, and an `INSERT` into the Employee table was attempted, as demonstrated in Figure 26. As expected, the attempt was denied, confirming that the permissions were correctly enforced. The `REVERT` command was then used to exit the simulation context back to the admin context.

Figure 26: Security Test.


The screenshot shows a SQL Server Enterprise Manager query window. The title bar indicates the connection is to 'v2.SQLQuery1.sql...OSE-WO\jlwon (71))' and the user is '@00628869_task1.s...OSE-WO\jlwon (65))'. The query text is as follows:

```
-- Test
-- Simulate a login with a user with TicketingStaff role
EXECUTE AS USER = 'ana_lopez';

-- Trying to insert into Employee (should fail for TicketingStaff)
INSERT INTO Employee (Name, Email, Username, PasswordHash, Role)
VALUES ('Test User', 'test.user@airline.com', 'test_user', '12345', 'TicketingStaff');

-- Revert back to the admin context
REVERT;
```

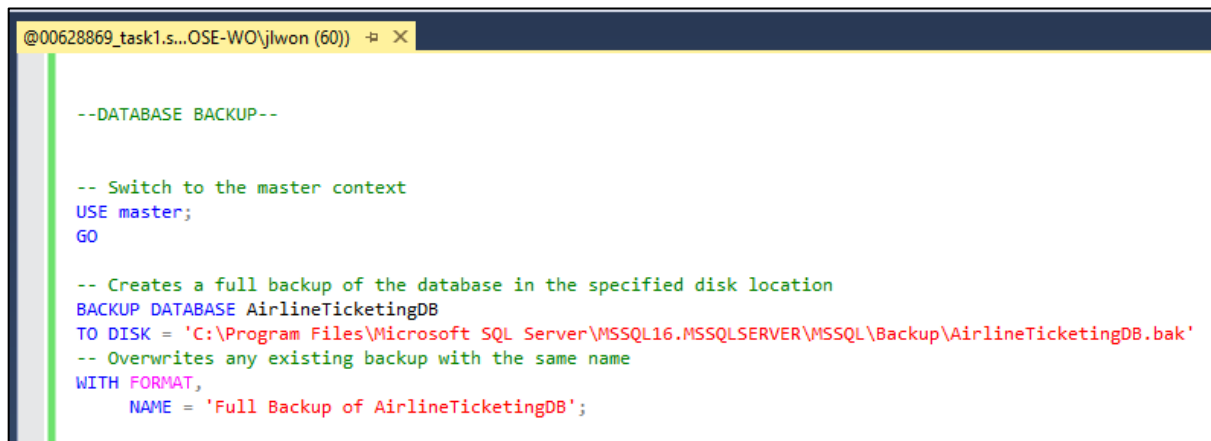
Below the query text, the 'Messages' pane shows the following error message:

```
Msg 229, Level 14, State 5, Line 575
The INSERT permission was denied on the object 'Employee',
database 'AirlineTicketingDB', schema 'dbo'.
```

The completion time is displayed as: 2025-04-22T18:20:11.3926989+01:00.

1.3.5.6 Backup and Recovery Implementation

Regarding backup and recovery, the implementation, as portrayed in Figure 27, uses the BACKUP DATABASE clause to manually generate a .bak file, which is provided as part of the deliverables to allow for the recovery of the schema and data in case of failure. Backup automation advice was provided to the client. Within the implementation, the WITH CHECKSUM clause enables, during the creation of the .bak file, the calculation of checksums based on the database content, which are then compared to the checksums stored in the database as default by SQL Server, throwing an error if a difference is found allowing to identify if the database is damaged.

Figure 27: Generating Backup File.

```
@00628869_task1.s...OSE-WO\jlwon (60))  X

--DATABASE BACKUP--

-- Switch to the master context
USE master;
GO

-- Creates a full backup of the database in the specified disk location
BACKUP DATABASE AirlineTicketingDB
TO DISK = 'C:\Program Files\Microsoft SQL Server\MSSQL16.MSSQLSERVER\MSSQL\Backup\AirlineTicketingDB.bak'
-- Overwrites any existing backup with the same name
WITH FORMAT,
    NAME = 'Full Backup of AirlineTicketingDB';
```

To restore the database using the .bak file, the RESTORE DATABASE clause can be used, and, optionally, the WITH CHECKSUM clause can also be used to identify if the database is damaged.

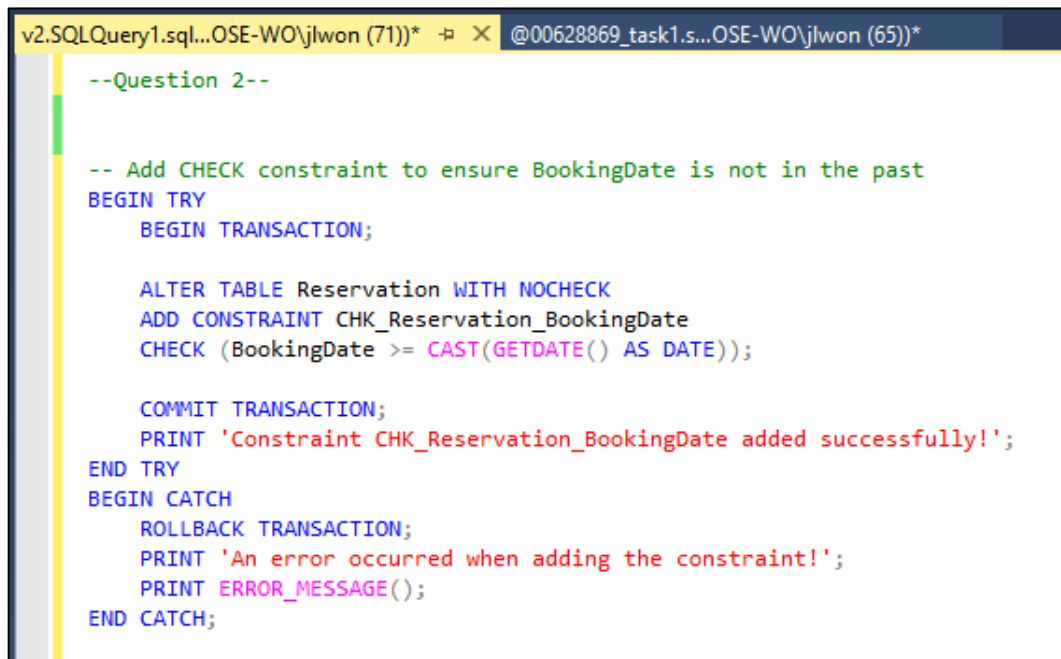
1.4 Database Objects and Business Logic Implementation

Following the database design and implementation the required database objects were also implemented.

1.4.1 Reservation Date Constraint

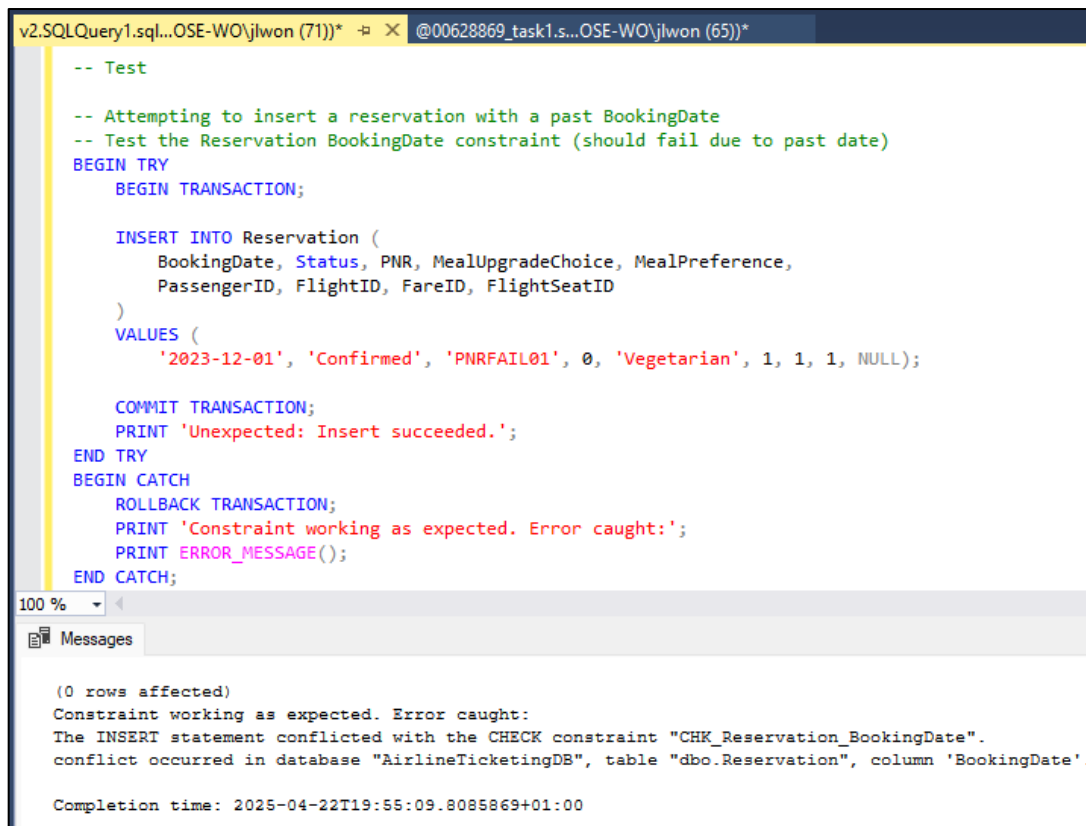
To ensure that booking dates cannot be created with past dates, a constraint was added to the Reservation table using a CHECK constraint, as presented in Figure 28, which validates that the BookingDate is greater than or equal to the current system date. The GETDATE() function was cast to DATE to remove the time component. The WITH NOCHECK clause was used so that existing records, which were inserted before this rule was enforced, would not be validated retroactively.

Figure 28: Booking Date Constraint Test.



```
v2.SQLQuery1.sql...OSE-WO\jlwon (71))* X @00628869_task1.s...OSE-WO\jlwon (65))*  
  
--Question 2--  
  
-- Add CHECK constraint to ensure BookingDate is not in the past  
BEGIN TRY  
    BEGIN TRANSACTION;  
  
    ALTER TABLE Reservation WITH NOCHECK  
    ADD CONSTRAINT CHK_Reservation_BookingDate  
    CHECK (BookingDate >= CAST(GETDATE() AS DATE));  
  
    COMMIT TRANSACTION;  
    PRINT 'Constraint CHK_Reservation_BookingDate added successfully!';  
END TRY  
BEGIN CATCH  
    ROLLBACK TRANSACTION;  
    PRINT 'An error occurred when adding the constraint!';  
    PRINT ERROR_MESSAGE();  
END CATCH;
```

To test the constraint, a reservation insert operation was attempted using a BookingDate in the past, which was correctly rejected as seen in Figure 29.

Figure 29: Booking Date Constraint Test.

The screenshot displays the SQL Server Enterprise Manager interface. The top pane shows a T-SQL query designed to test a booking date constraint. The query uses a TRY-CATCH block to attempt an insert of a reservation with a past booking date ('2023-12-01'). If the insert succeeds, it prints an unexpected message. If it fails, it prints a message indicating the constraint is working as expected. The bottom pane shows the execution results, which include a message stating that the insert failed due to a CHECK constraint violation on the 'BookingDate' column.

```
-- Test
-- Attempting to insert a reservation with a past BookingDate
-- Test the Reservation BookingDate constraint (should fail due to past date)
BEGIN TRY
    BEGIN TRANSACTION;

    INSERT INTO Reservation (
        BookingDate, Status, PNR, MealUpgradeChoice, MealPreference,
        PassengerID, FlightID, FareID, FlightSeatID
    )
    VALUES (
        '2023-12-01', 'Confirmed', 'PNRFAIL01', 0, 'Vegetarian', 1, 1, 1, NULL);

    COMMIT TRANSACTION;
    PRINT 'Unexpected: Insert succeeded.';
END TRY
BEGIN CATCH
    ROLLBACK TRANSACTION;
    PRINT 'Constraint working as expected. Error caught: ';
    PRINT ERROR_MESSAGE();
END CATCH;
```

100 %

Messages

(0 rows affected)

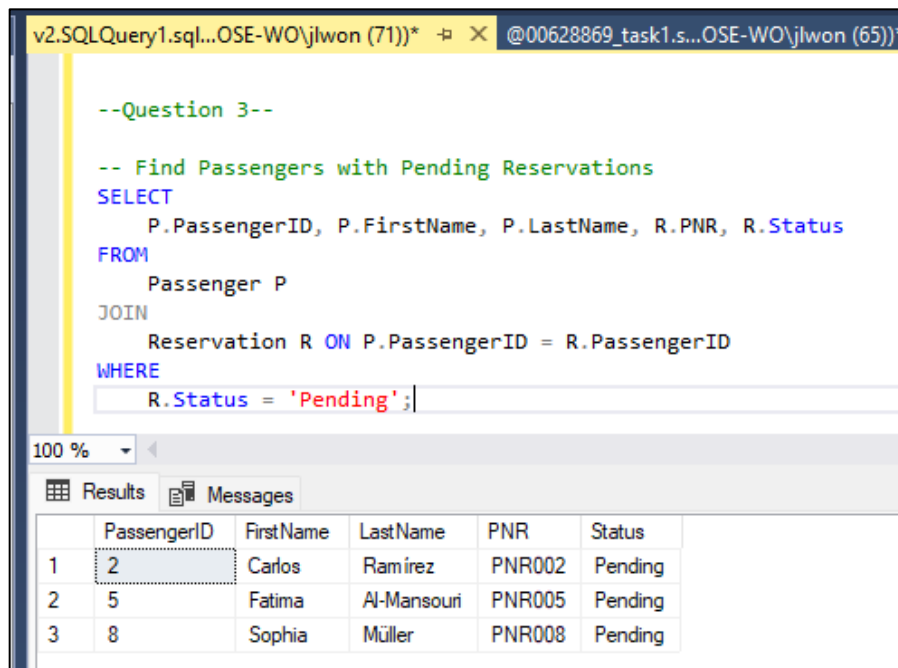
Constraint working as expected. Error caught:
The INSERT statement conflicted with the CHECK constraint "CHK_Reservation_BookingDate".
conflict occurred in database "AirlineTicketingDB", table "dbo.Reservation", column 'BookingDate'.

Completion time: 2025-04-22T19:55:09.8085869+01:00

1.4.2 Passenger Queries: Pending Reservations & Age Filtering

To identify passengers with pending reservations, a SELECT query was written using a JOIN between the Passenger and Reservation tables, filtering on Status = 'Pending', as presented in Figure 30.

Figure 30: Find Passengers with Pending Reservations Query.



The screenshot shows a SQL query window with the following text:

```
--Question 3--

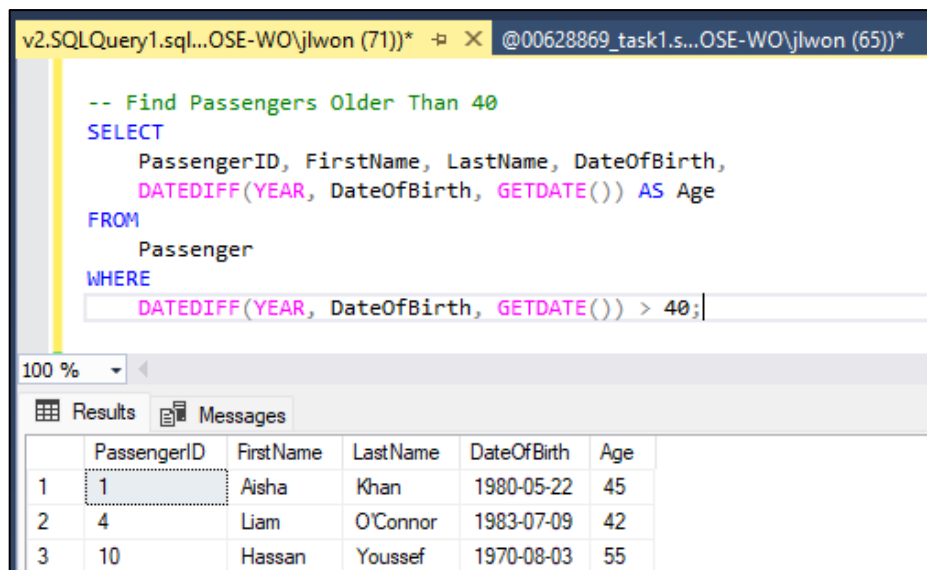
-- Find Passengers with Pending Reservations
SELECT
    P.PassengerID, P.FirstName, P.LastName, R.PNR, R.Status
FROM
    Passenger P
JOIN
    Reservation R ON P.PassengerID = R.PassengerID
WHERE
    R.Status = 'Pending';
```

Below the query, the 'Results' tab is active, displaying a table with 6 columns: PassengerID, FirstName, LastName, PNR, and Status. The table contains 3 rows of data.

	PassengerID	FirstName	LastName	PNR	Status
1	2	Carlos	Ramírez	PNR002	Pending
2	5	Fatima	Al-Mansouri	PNR005	Pending
3	8	Sophia	Müller	PNR008	Pending

For retrieving passengers over the age of 40, the SQL Server's DATEDIFF function was used to calculate the age based on the current date and the passenger's date of birth and a WHERE clause was used to return only those records where the result exceeds 40 years, as demonstrated in Figure 31.

Figure 31: Finding Passengers Older than 40 Query.



```

v2.SQLQuery1.sql...OSE-WO\jlwon (71))* @00628869_task1.s...OSE-WO\jlwon (65))*

-- Find Passengers Older Than 40
SELECT
    PassengerID, FirstName, LastName, DateOfBirth,
    DATEDIFF(YEAR, DateOfBirth, GETDATE()) AS Age
FROM
    Passenger
WHERE
    DATEDIFF(YEAR, DateOfBirth, GETDATE()) > 40;

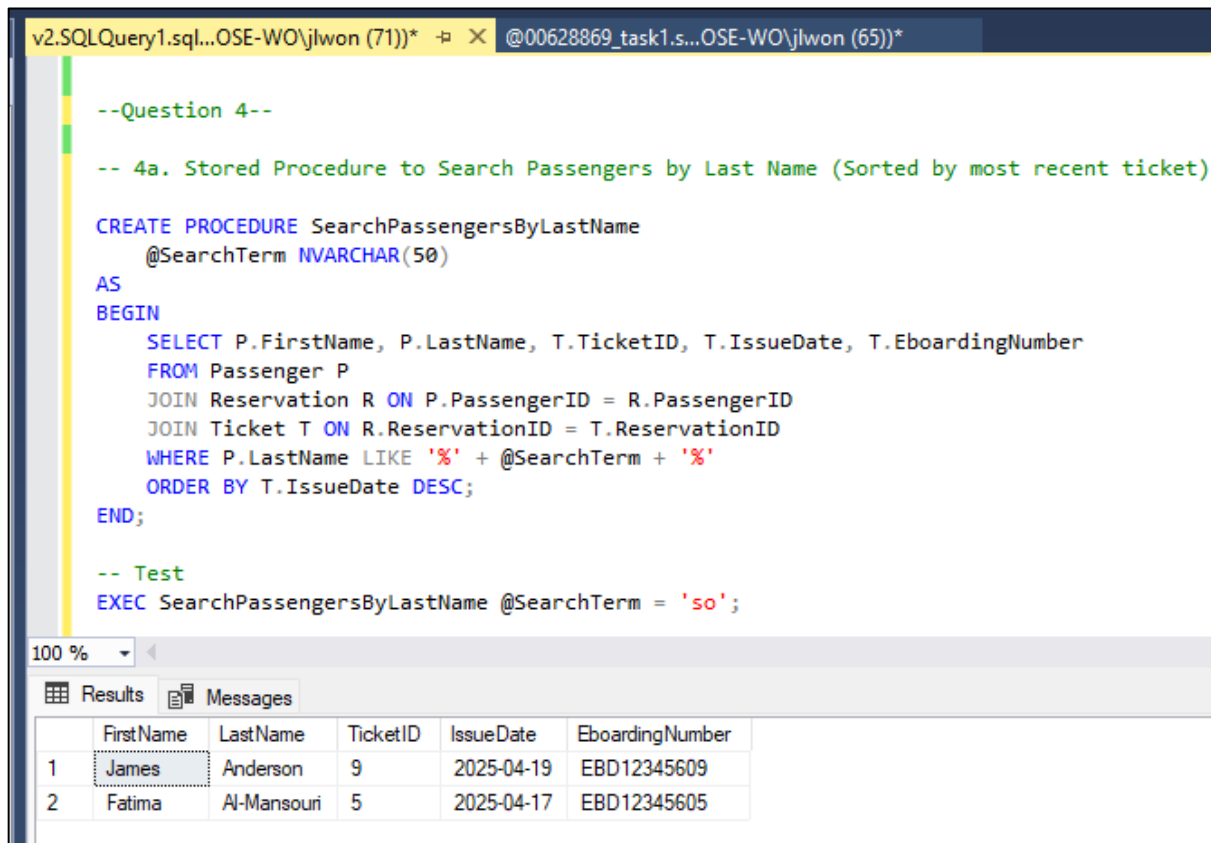
```

	PassengerID	FirstName	LastName	DateOfBirth	Age
1	1	Aisha	Khan	1980-05-22	45
2	4	Liam	O'Connor	1983-07-09	42
3	10	Hassan	Youssef	1970-08-03	55

1.4.3 Stored Procedures for Passenger, Employee, and Reservation Management

a) Search Passenger by Last Name

To help staff locate passenger records quickly, a stored procedure, presented in Figure 32, was implemented to search for partial matching character strings in the LastName field of the Passenger table. The procedure uses the LIKE operator with wildcards to match partial strings, allowing queries such as searching for “so” to return passengers whose last name contains an instance of that string literal, like Al-Mansouri or Anderson. The query sorts results so that the most recently issued ticket appears first by joining the Passenger table to the Reservation and Ticket tables and using the ORDER BY IssueDate DESC clause.

Figure 32: Find Passenger by Last Name Stored Procedure.


```
--Question 4--

-- 4a. Stored Procedure to Search Passengers by Last Name (Sorted by most recent ticket)

CREATE PROCEDURE SearchPassengersByLastName
    @SearchTerm NVARCHAR(50)
AS
BEGIN
    SELECT P.FirstName, P.LastName, T.TicketID, T.IssueDate, T.EboardingNumber
    FROM Passenger P
    JOIN Reservation R ON P.PassengerID = R.PassengerID
    JOIN Ticket T ON R.ReservationID = T.ReservationID
    WHERE P.LastName LIKE '%' + @SearchTerm + '%'
    ORDER BY T.IssueDate DESC;
END;

-- Test
EXEC SearchPassengersByLastName @SearchTerm = 'so';
```

	FirstName	LastName	TicketID	IssueDate	EboardingNumber
1	James	Anderson	9	2025-04-19	EBD12345609
2	Fatima	Al-Mansouri	5	2025-04-17	EBD12345605

b) List today's business class passengers and meal preferences

A second stored procedure, portrayed in Figure 33, was created to retrieve all passengers flying in business class today, based on ticket issue date, along with their specific meal preferences. The procedure performs joins across the Ticket, Reservation, and Passenger tables and filters by both the current system date (GETDATE()) and the Business value in the TravelClass field. CAST(GETDATE() AS DATE) is used to remove the time component from the system date to allow for comparison with the ticket issue date. The output includes the full name of each passenger and their declared meal preference.

Figure 33: Listing Passengers Flying Today in Business Class Stored Procedure.

```

v2.SQLQuery1.sql...OSE-WO\jlwon (71))* X @00628869_task1.s...OSE-WO\jlwon (65))*

--4b. Stored procedure to list of passengers flying today in business class
-- and meal preference

CREATE OR ALTER PROCEDURE usp_GetBusinessClassPassengersFlyingToday
AS
BEGIN
    SELECT
        P.FirstName, P.LastName, R.MealPreference, F.FareClass, FL.DepartureTime
    FROM Reservation R
    INNER JOIN Passenger P ON R.PassengerID = P.PassengerID
    INNER JOIN Fare F ON R.FareID = F.FareID
    INNER JOIN Flight FL ON R.FlightID = FL.FlightID
    WHERE
        F.FareClass = 'Business'
        AND CAST(FL.DepartureTime AS DATE) = CAST(GETDATE() AS DATE);
END;

-- Test

-- Create a Reservation for a flight departing today in business class
INSERT INTO Reservation (
    BookingDate, Status, PNR, MealUpgradeChoice, MealPreference,
    PassengerID, FlightID, FareID, FlightSeatID
)
VALUES ('2025-04-22', 'Confirmed', 'FWICUF806C', 0, 'Vegetarian',
    3, 5, 14, NULL);

-- Execute the store procedure
EXEC usp_GetBusinessClassPassengersFlyingToday;

```

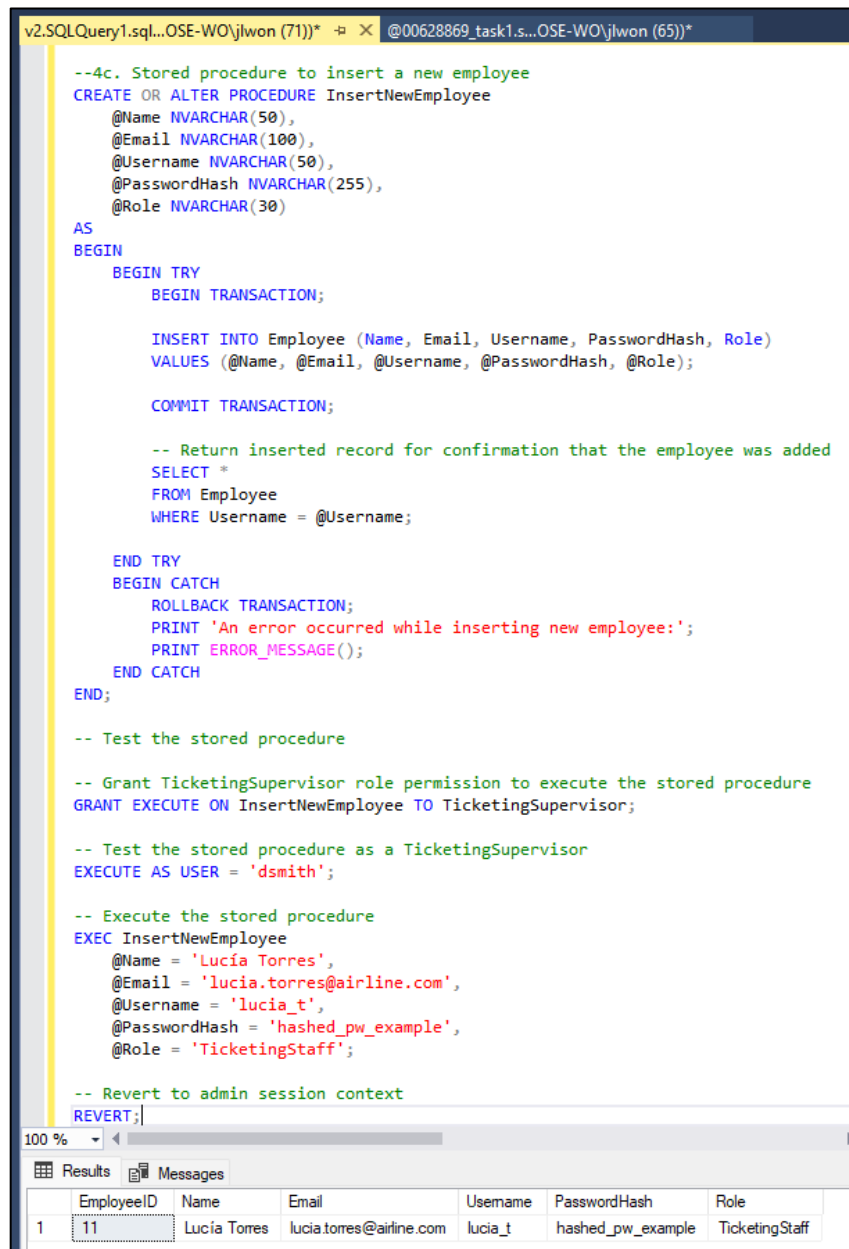
100 %

Results Messages

	FirstName	LastName	MealPreference	FareClass	DepartureTime
1	Zhang	Wei	Vegetarian	Business	2025-04-22 13:45:00.000

c) Insert new employees (restricted to supervisors)

A stored procedure, portrayed in Figure 34, was created to allow supervisors to insert new employee records. This stored procedure accepts the employee's name, email, username, hashed password, and role as parameters and performs an INSERT into the Employee table. To enforce role-based access, the procedure was granted only to the TicketingSupervisor role and wrapped in a transaction for consistency. The procedure was tested using EXECUTE AS USER to confirm that only supervisors are authorised to insert employee records.

Figure 34: Insert New Employee Stored Procedure.


```

v2.SQLQuery1.sql...OSE-WO\j\won (71))* X @00628869_task1.s...OSE-WO\j\won (65))*
--4c. Stored procedure to insert a new employee
CREATE OR ALTER PROCEDURE InsertNewEmployee
    @Name NVARCHAR(50),
    @Email NVARCHAR(100),
    @Username NVARCHAR(50),
    @PasswordHash NVARCHAR(255),
    @Role NVARCHAR(30)
AS
BEGIN
    BEGIN TRY
        BEGIN TRANSACTION;

        INSERT INTO Employee (Name, Email, Username, PasswordHash, Role)
        VALUES (@Name, @Email, @Username, @PasswordHash, @Role);

        COMMIT TRANSACTION;

        -- Return inserted record for confirmation that the employee was added
        SELECT *
        FROM Employee
        WHERE Username = @Username;

    END TRY
    BEGIN CATCH
        ROLLBACK TRANSACTION;
        PRINT 'An error occurred while inserting new employee: ';
        PRINT ERROR_MESSAGE();
    END CATCH
END;

-- Test the stored procedure

-- Grant TicketingSupervisor role permission to execute the stored procedure
GRANT EXECUTE ON InsertNewEmployee TO TicketingSupervisor;

-- Test the stored procedure as a TicketingSupervisor
EXECUTE AS USER = 'dsmith';

-- Execute the stored procedure
EXEC InsertNewEmployee
    @Name = 'Lucia Torres',
    @Email = 'lucia.torres@airline.com',
    @Username = 'lucia_t',
    @PasswordHash = 'hashed_pw_example',
    @Role = 'TicketingStaff';

-- Revert to admin session context
REVERT;

```

EmployeeID	Name	Email	Username	PasswordHash	Role
11	Lucia Torres	lucia.torres@airline.com	lucia_t	hashed_pw_example	TicketingStaff

d) Update passenger details (only if a reservation exists)

A stored procedure, shown in Figure 35, was implemented to update the details of a passenger, but only if the passenger has previously booked a flight. It was created using CREATE PROCEDURE with parameters for PassengerID, Email, MealPreference, and EmergencyContactNumber. The procedure first uses an IF EXISTS clause to check whether the given PassengerID appears in the Reservation

table. If it does, the passenger's record is updated in the Passenger table using the UPDATE statement.

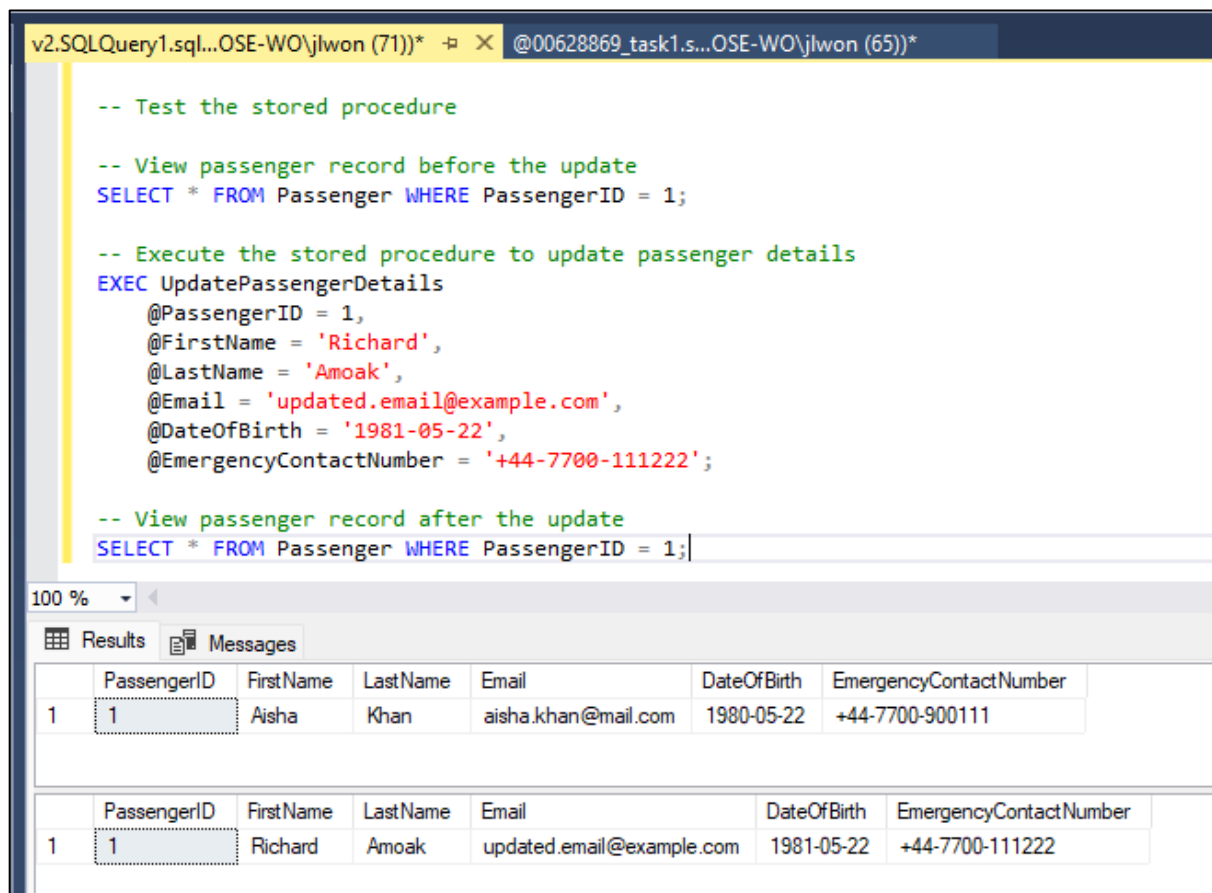
Figure 35: Upgrade Passenger Details Stored Procedure.

```
v2.SQLQuery1.sql...OSE-WO\jlwon (71))* -R X @00628869_task1.s...OSE-WO\jlwon (65))*
-- 4d. Create stored procedure to update passenger details (only if they have a reservation)
CREATE OR ALTER PROCEDURE UpdatePassengerDetails
    @PassengerID INT,
    @FirstName NVARCHAR(50),
    @LastName NVARCHAR(50),
    @Email NVARCHAR(100),
    @DateOfBirth DATE,
    @EmergencyContactNumber NVARCHAR(20)
AS
BEGIN
    BEGIN TRY
        BEGIN TRANSACTION;

        -- Check if passenger has at least one reservation
        IF EXISTS (
            SELECT 1 FROM Reservation WHERE PassengerID = @PassengerID
        )
        BEGIN
            UPDATE Passenger
            SET FirstName = @FirstName,
                LastName = @LastName,
                Email = @Email,
                DateOfBirth = @DateOfBirth,
                EmergencyContactNumber = @EmergencyContactNumber
            WHERE PassengerID = @PassengerID;

            COMMIT TRANSACTION;
            PRINT 'Passenger details updated successfully.';
        END
        ELSE
        BEGIN
            ROLLBACK TRANSACTION;
            PRINT 'No update performed. Passenger has no reservation.';
        END
    END TRY
    BEGIN CATCH
        ROLLBACK TRANSACTION;
        PRINT 'An error occurred: ';
        PRINT ERROR_MESSAGE();
    END CATCH
END;
```

The procedure was tested by executing it with modified values for a passenger and checking the passenger's record before and after to confirm the update was successful, as seen in Figure 36.

Figure 36: Test Upgrade Passenger Details Stored Procedure.


```

v2.SQLQuery1.sql...OSE-WO\jlwon (71))* X @00628869_task1.s...OSE-WO\jlwon (65))*

-- Test the stored procedure

-- View passenger record before the update
SELECT * FROM Passenger WHERE PassengerID = 1;

-- Execute the stored procedure to update passenger details
EXEC UpdatePassengerDetails
    @PassengerID = 1,
    @FirstName = 'Richard',
    @LastName = 'Amoak',
    @Email = 'updated.email@example.com',
    @DateOfBirth = '1981-05-22',
    @EmergencyContactNumber = '+44-7700-111222';

-- View passenger record after the update
SELECT * FROM Passenger WHERE PassengerID = 1;

```

100 %

Results Messages

	PassengerID	FirstName	LastName	Email	DateOfBirth	EmergencyContactNumber
1	1	Aisha	Khan	aisha.khan@mail.com	1980-05-22	+44-7700-900111

	PassengerID	FirstName	LastName	Email	DateOfBirth	EmergencyContactNumber
1	1	Richard	Amoak	updated.email@example.com	1981-05-22	+44-7700-111222

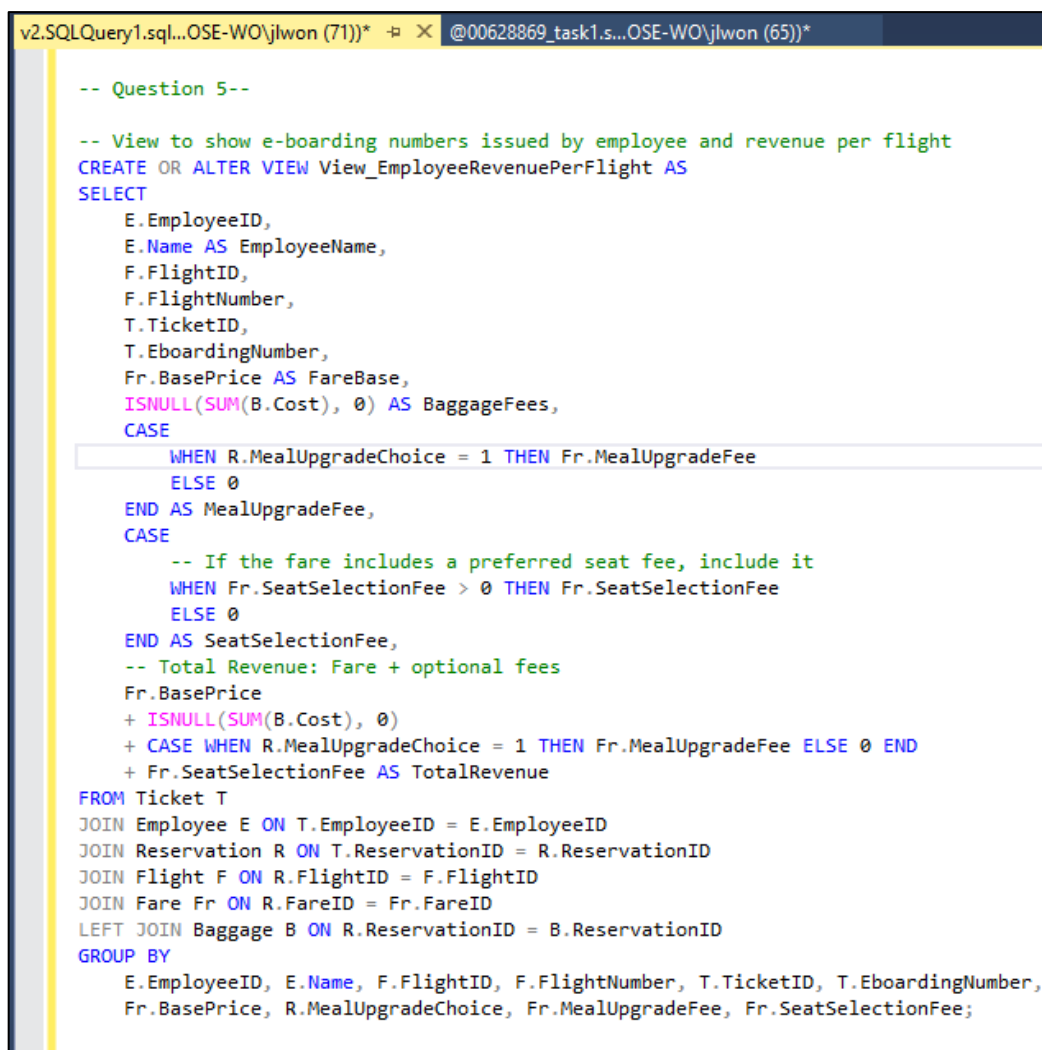
1.4.4 Employee Revenue View per Flight

A view, presented in Figure 37, was created to report the e-boarding numbers issued by each employee and the corresponding revenue generated per flight. This view, named `View_EmployeeRevenuePerFlight`, consolidates information from several tables including `Ticket`, `Employee`, `Reservation`, `Flight`, and `Baggage`.

The `Ticket` table provides the base fare and e-boarding number issued by each employee. The `Reservation` table links each ticket to a specific flight, while the `Flight` table contributes the flight number and identifier, providing the necessary link to distinguish which flight each ticket and its associated revenue belong to. The `Employee` table is joined to identify the issuing staff member. The `Baggage` table is included through a `LEFT JOIN`, which ensures tickets without baggage are still displayed in the results.

To calculate total revenue, the view adds the base fare from the ticket to the total baggage cost associated with the reservation. Since not all passengers check in baggage, the `ISNULL(SUM(B.Cost), 0)` construct ensures that nulls are replaced with zero when computing the final revenue figure. The `GROUP BY` clause is applied on a per-ticket basis alongside employee and flight information, allowing for reporting of revenue by staff member and flight in an itemised manner.

Figure 37: View to Show E-Boarding Numbers.



```
-- Question 5--

-- View to show e-boarding numbers issued by employee and revenue per flight
CREATE OR ALTER VIEW View_EmployeeRevenuePerFlight AS
SELECT
    E.EmployeeID,
    E.Name AS EmployeeName,
    F.FlightID,
    F.FlightNumber,
    T.TicketID,
    T.EboardingNumber,
    Fr.BasePrice AS FareBase,
    ISNULL(SUM(B.Cost), 0) AS BaggageFees,
    CASE
        WHEN R.MealUpgradeChoice = 1 THEN Fr.MealUpgradeFee
        ELSE 0
    END AS MealUpgradeFee,
    CASE
        -- If the fare includes a preferred seat fee, include it
        WHEN Fr.SeatSelectionFee > 0 THEN Fr.SeatSelectionFee
        ELSE 0
    END AS SeatSelectionFee,
    -- Total Revenue: Fare + optional fees
    Fr.BasePrice
    + ISNULL(SUM(B.Cost), 0)
    + CASE WHEN R.MealUpgradeChoice = 1 THEN Fr.MealUpgradeFee ELSE 0 END
    + Fr.SeatSelectionFee AS TotalRevenue
FROM Ticket T
JOIN Employee E ON T.EmployeeID = E.EmployeeID
JOIN Reservation R ON T.ReservationID = R.ReservationID
JOIN Flight F ON R.FlightID = F.FlightID
JOIN Fare Fr ON R.FareID = Fr.FareID
LEFT JOIN Baggage B ON R.ReservationID = B.ReservationID
GROUP BY
    E.EmployeeID, E.Name, F.FlightID, F.FlightNumber, T.TicketID, T.EboardingNumber,
    Fr.BasePrice, R.MealUpgradeChoice, Fr.MealUpgradeFee, Fr.SeatSelectionFee;
```

To verify the view's functionality, a `SELECT` query was executed using specific `EmployeeID` and `FlightID` values, confirming that the view returns accurate, flight-specific revenue information issued by each employee as seen in Figure 38.

Figure 38: Test View to Show E-Boarding Numbers.

SQL Query Editor: v2.SQLQuery1.sql...OSE-WO\jlwon (71))* @00628869_task1.s...OSE-WO\jlwon (65))*

```
-- Test the view with employee with ID 1 and flight with ID 1
SELECT *
FROM View_EmployeeRevenuePerFlight
WHERE EmployeeID = 1 AND FlightID = 1;
```

100 %

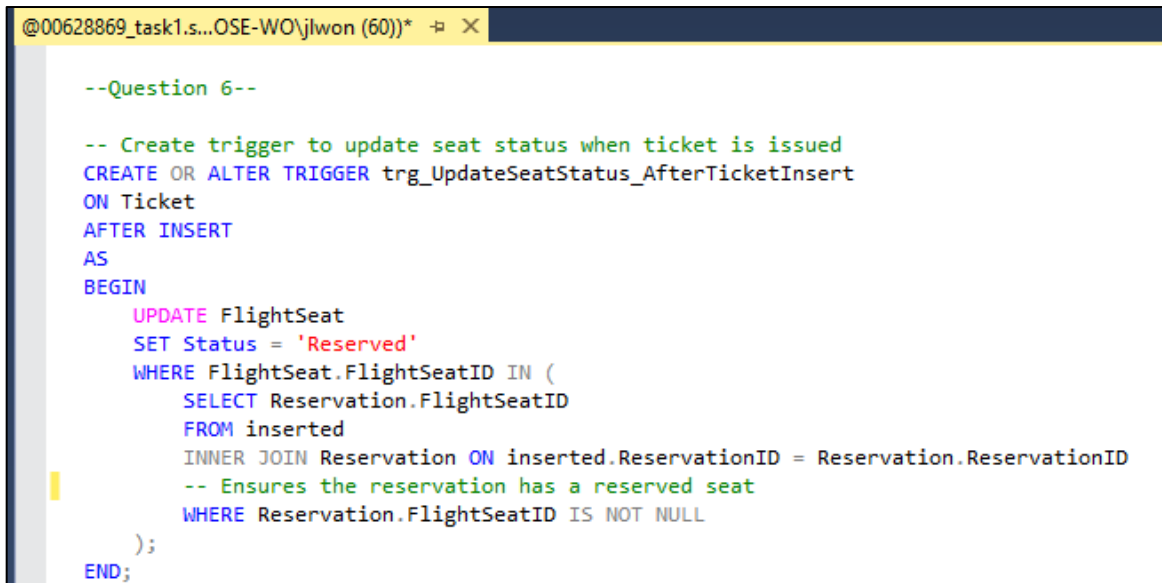
Results Messages

	EmployeeID	EmployeeName	FlightID	FlightNumber	TicketID	EboardingNumber	FareBase	BaggageFees	MealUpgradeFee	SeatSelectionFee	TotalRevenue
1	1	Ana María López	1	BA123	1	EBD12345601	150.00	0.00	20.00	30.00	200.00

1.4.5 Seat Status Trigger

To meet the requirement of automatically marking a seat as reserved once a ticket is issued, a trigger, presented in Figure 39, was implemented using the AFTER INSERT event on the Ticket table. In the current schema, seat assignments are represented through the FlightSeat table, where each record links a seat in a flight to a reservation. When a ticket is issued, it references a reservation that, in turn, may reference a flight seat. The trigger works by first joining the Reservation table with the “inserted” pseudo table, which is a temporary table that holds the inserted rows, in this case a ticket, during the execution of the AFTER INSERT trigger, and selects the reservation matching the ReservationID in the ticket only if the reservation has a FlightSeatID foreign key value not set to null, indicating that the reservation is linked to a FlightSeat representing an entry in the table that contains the seats of a flight. Then, once the Reservation is selected, the FlightSeat row with the matching ReservationID is updated to Reserved.

Figure 39: Trigger to Update Seat Status when Ticket is Issued.



```
--Question 6--

-- Create trigger to update seat status when ticket is issued
CREATE OR ALTER TRIGGER trg_UpdateSeatStatus_AfterTicketInsert
ON Ticket
AFTER INSERT
AS
BEGIN
    UPDATE FlightSeat
    SET Status = 'Reserved'
    WHERE FlightSeat.FlightSeatID IN (
        SELECT Reservation.FlightSeatID
        FROM inserted
        INNER JOIN Reservation ON inserted.ReservationID = Reservation.ReservationID
        -- Ensures the reservation has a reserved seat
        WHERE Reservation.FlightSeatID IS NOT NULL
    );
END;
```

A test was performed by first inserting a new reservation record with a linked FlightSeat already existing in the FlightSeat table and having an Available status. The new ReservationID, obtained using SCOPE_IDENTITY(), was assigned to a variable @NewResID. A new Ticket was then created for the corresponding ReservationID. To check that the trigger had worked as expected, a SELECT statement was used to retrieve the status of the FlightSeat where the ReservationID matched @NewResID, confirming that the status had been updated as expected, as seen in Figure 40.

Figure 40: Test Trigger to Update Seat Status when Ticket is Issued.

The screenshot shows a SQL Server Enterprise Manager window with two tabs: 'SQLQuery2.sql - D:\...OSE-WO\jwlon (60))' and 'v2.SQLQuery1.sql...OSE-WO\jwlon (57))*'. The active tab displays a SQL script designed to test a trigger. The script includes comments and SQL statements to create a reservation, insert a ticket, and verify the seat status. Below the script, the 'Results' tab is active, showing a single row with the status 'Reserved'.

```

-- Test the trigger
-- Create a new reservation linked to a seat
INSERT INTO Reservation (
    BookingDate, Status, PNR, MealUpgradeChoice, MealPreference,
    PassengerID, FlightID, FareID, FlightSeatID
)
VALUES (
    CAST(GETDATE() AS DATE), 'Confirmed', 'TRGPNR01', 1, 'Vegetarian',
    3, 2, 4, 9
);

-- Find the new ReservationID
DECLARE @NewResID INT = SCOPE_IDENTITY();

-- Insert the ticket to fire the trigger
INSERT INTO Ticket (
    IssueDate, EboardingNumber, ReservationID, EmployeeID
)
VALUES (
    GETDATE(), 'EBD-TRG-20250422', @NewResID, 1
);

-- Confirm that the seat status is now 'Reserved'
SELECT Status FROM FlightSeat WHERE ReservationID = @NewResID;

```

	Status
1	Reserved

1.4.6 Checked-in Baggage Function

A user-defined function was implemented to allow tracking of the total number of baggage items with the status Checked-In for a specific flight and date, as portrayed in Figure 41. The function declares a variable to hold the required output and be able to return it. The variable is assigned a value calculated by joining the Baggage, Reservation, and Flight tables to filter records by flight and departure date and counting only those baggage items whose status is Checked-In. The function was tested using a SELECT statement passing a sample flight ID and date, and it successfully returned the correct baggage count.

Figure 41: User-Defined Function to Count Checked-In Baggage.

The screenshot shows a SQL Server Enterprise Manager query window with two tabs. The active tab is titled "v2.SQLQuery1.sql...OSE-WO\jlwon (57))*" and contains the following SQL code:

```
-- User-defined function to count checked-in baggage for a specific flight and date
CREATE FUNCTION dbo.ufn_CountCheckedInBaggage
(
    @FlightID INT,
    @DepartureDate DATE
)
RETURNS INT
AS
BEGIN
    -- Variable to hold the number of checked-in bags that match the criteria
    DECLARE @BaggageCount INT;

    SELECT @BaggageCount = COUNT(*)
    FROM Baggage B
    JOIN Reservation R ON B.ReservationID = R.ReservationID
    JOIN Flight F ON R.FlightID = F.FlightID
    WHERE
        B.Status = 'Checked-In'
        AND F.FlightID = @FlightID
        AND CAST(F.DepartureTime AS DATE) = @DepartureDate;

    RETURN @BaggageCount;
END;

-- Test the UDF
SELECT dbo.ufn_CountCheckedInBaggage(3, '2025-04-20') AS CheckedInBaggage;
```

Below the query editor, the "Results" tab is active, displaying a single row of data:

CheckedInBaggage
1

1.4.7 Extended Functionality: Reservation, Seat, and Ticket Procedures

a) Reserve a seat for a reservation

A stored procedure, displayed in Figure 42, was created to enable the allocation of a specific seat to a passenger's reservation. This procedure can be used during the booking process or after a booking has already been made, for instance, by the ticketing staff when the reservation details are confirmed for the ticket issuance, if the passenger has not reserved a seat previously.

The procedure accepts as parameters the ReservationID and the SeatID. Internally, the procedure declares two variables, one to hold the FlightID associated with the reservation and another to hold the FlightSeatID, which uniquely identifies a specific seat on a specific flight in the FlightSeat junction table. Then the procedure retrieves the FlightID associated with the reservation by querying the Reservation table and then uses the combination of FlightID and the provided SeatID to identify the corresponding FlightSeatID in the FlightSeat table.

If the seat is found and confirmed to be available, the procedure proceeds to update the FlightSeat table by assigning the reservation and changing the seat's Status to 'Reserved'. It also updates the corresponding reservation to store the FlightSeatID, to maintain consistency between the reservation and seat assignment. If the seat is not available or does not exist for the flight, the transaction is rolled back, and an error message is raised.

Row-level lock and SERIALIZABLE isolation level are applied to ensure the seat remains available during the transaction, avoiding concurrency issues.

To verify correct operation, the procedure was tested with ReservationID 13 and SeatID 14. The reservation record was retrieved afterwards, confirming that the FlightSeatID was successfully updated to 16, indicating that the seat reservation logic worked as intended, as shown in Figure 43.

Figure 42: Stored Procedure to Reserve Seat.

```

v2.SQLQuery1.sql...OSE-WO\jlwon (68))*  X
--8a. Stored procedure to reserve a seat for a reservation
CREATE OR ALTER PROCEDURE ReserveSeatForReservation
    @ReservationID INT,
    @SeatID INT
AS
BEGIN
    DECLARE @FlightID INT;
    DECLARE @FlightSeatID INT;

    BEGIN TRY
        SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
        BEGIN TRANSACTION;
        -- Retrieve the flight associated with the reservation
        SELECT @FlightID = FlightID
        FROM Reservation
        WHERE ReservationID = @ReservationID;

        -- Identify the corresponding FlightSeatID
        -- Lock the seat row to prevent concurrency issues
        SELECT @FlightSeatID = FlightSeatID
        FROM FlightSeat WITH (UPDLOCK, HOLDLOCK)
        WHERE FlightID = @FlightID AND SeatID = @SeatID AND Status = 'Available';

        -- If no match found, the seat is already reserved or does not exist for that flight
        IF @FlightSeatID IS NULL
        BEGIN
            RAISERROR('The selected seat is either not available or does not exist for this flight.', 16, 1);
            ROLLBACK TRANSACTION;
            RETURN;
        END

        -- Update the FlightSeat table to reflect the reservation
        UPDATE FlightSeat
        SET Status = 'Reserved', ReservationID = @ReservationID
        WHERE FlightSeatID = @FlightSeatID;

        -- Also update the Reservation to store the seat reference
        UPDATE Reservation
        SET FlightSeatID = @FlightSeatID
        WHERE ReservationID = @ReservationID;

        COMMIT TRANSACTION;
    END TRY
    BEGIN CATCH
        ROLLBACK TRANSACTION;
        PRINT ERROR_MESSAGE();
    END CATCH
END;

```

Figure 43: Test Stored Procedure to Reserve Seat.

```

v2.SQLQuery1.sql...OSE-WO\jlwon (68))*  X

-- Test the stored procedure

EXEC ReserveSeatForReservation
    @ReservationID = 13,
    @SeatID = 14;

-- View reservation record after the update
SELECT * FROM Reservation WHERE ReservationID = 13;

```

100 %

Results Messages

	ReservationID	BookingDate	Status	PNR	MealUpgradeChoice	MealPreference	PassengerID	FlightID	FareID	FlightSeatID
1	13	2025-04-22	Confirmed	FWICUFB06C	0	Vegetarian	3	5	14	16

b) Create reservations with optional seat selection

A stored procedure, shown in Figure 44, was created to allow booking a flight for a passenger, inserting a record in the Reservation table, specifying meal preferences and class of service, and optionally selecting a seat. The Reservation date is retrieved from the system's current timestamp (seconds and milliseconds), and a unique PNR is generated using a combination of the system time and passenger ID. If a SeatID is provided, it invokes the ReserveSeatForReservation to reserve the seat and link it to the reservation.

Figure 44: Stored Procedure to Reserve a Flight.

```
v2.SQLQuery1.sql...OSE-WO\jlwon (68))* -> X
--8b. Stored procedure to reserve a flight and seat (optional)
-- Uses stored procedure 8c. as helper for seat reservation
-- Generates unique PNRs via a combination of timestamp and passengerID
CREATE OR ALTER PROCEDURE CreateReservationWithOptionalSeat
    @PassengerID INT,
    @FlightID INT,
    @FareID INT,
    @MealPreference NVARCHAR(30),
    @MealUpgradeChoice BIT,
    @SeatID INT = NULL -- Seat optional
AS
BEGIN
    -- Variable to store the PNR
    -- Variable gets assigned a generates PNR using a timestamp + PassengerID for uniqueness
    DECLARE @PNR NVARCHAR(10) = 'PNR' + RIGHT(CAST(DATEPART(SECOND, SYSDATETIME()) AS VARCHAR) +
        CAST(DATEPART(MILLISECOND, SYSDATETIME()) AS VARCHAR) + CAST(@PassengerID AS VARCHAR), 7);
    BEGIN TRY
        SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
        BEGIN TRANSACTION;
        -- Variable to hold the generated reservationID
        DECLARE @ReservationID INT;
        -- Insert reservation with null FlightSeatID initially
        INSERT INTO Reservation (
            BookingDate, Status, PNR, MealUpgradeChoice, MealPreference,
            PassengerID, FlightID, FareID, FlightSeatID)
        VALUES (
            CAST(GETDATE() AS DATE), 'Pending', @PNR, @MealUpgradeChoice, @MealPreference,
            @PassengerID, @FlightID, @FareID, NULL
        );
        -- Gets the generated ReservationID and assigns its value to @ReservationID variable
        SET @ReservationID = SCOPE_IDENTITY();
        -- If a seat was selected, calls the stored procedure ReserveSeatForReservation in 8a.
        IF @SeatID IS NOT NULL
        BEGIN
            EXEC ReserveSeatForReservation @ReservationID, @SeatID;
        END
        COMMIT TRANSACTION;
    END TRY
    BEGIN CATCH
        ROLLBACK TRANSACTION;
        PRINT ERROR_MESSAGE();
    END CATCH
END;
```

To verify the implementation, the procedure was executed using valid input parameters for a business-class seat on Flight 5. The result was confirmed by checking both the Reservation and FlightSeat tables to ensure that the reservation was created, and the seat status was updated to Reserved as shown in Figure 45.

Figure 45: Test Stored Procedure to Reserve a Flight.

The screenshot displays a SQL query window titled 'v2.SQLQuery1.sql...OSE-WO\jilwon (68)' with the following SQL code:

```
-- Test the stored procedure
-- Execute the stored procedure to create a reservation and reserve a seat
EXEC CreateReservationWithOptionalSeat
    @PassengerID = 10,
    @FlightID = 1,
    @FareID = 1,
    @MealPreference = 'Non-Vegetarian',
    @MealUpgradeChoice = 1,
    @SeatID = 5;

-- Verify that the new reservation has been created and linked to a seat
SELECT TOP 1 *
FROM Reservation
ORDER BY ReservationID DESC;

-- Check FlightSeat table to confirm the seat was marked as reserved
SELECT *
FROM FlightSeat
WHERE SeatID = 5 AND FlightID = 1;
```

Below the query window, the 'Results' tab shows two tables. The first table is the result of the first SELECT statement, and the second table is the result of the second SELECT statement.

ReservationID	BookingDate	Status	PNR	MealUpgradeChoice	MealPreference	PassengerID	FlightID	FareID	FlightSeatID
17	2025-04-23	Pending	PNR3115410	1	Non-Vegetarian	10	1	1	5

FlightSeatID	Status	FlightID	SeatID	ReservationID
5	Reserved	1	5	17

c) Issue tickets with concurrency protection

A stored procedure, presented in Figure 46, was created to handle Ticket issuance. It takes two parameters, the ReservationID and the EmployeeID. The procedure first verifies that a ticket has not already been issued for the given reservation. If a ticket already exists, an error is raised, and the transaction is rolled back. If not, a unique e-boarding number is generated by combining the current timestamp (seconds and

milliseconds) with the reservation ID. Next, the reservation's status is updated to Confirmed in case it was still marked as Pending. Finally, a ticket is inserted into the Ticket table using the system date as the issue date.

The procedure uses transaction control and row-level locking with the SERIALIZABLE isolation level to protect against concurrency issues.

Figure 46: Stored Procedure to Issue a Ticket.

```
v2.SQLQuery1.sql...OSE-WO\jlwon (68))* X
--8c. Stored procedure to issue a ticket
CREATE PROCEDURE IssueTicket
    @ReservationID INT,
    @EmployeeID INT
AS
BEGIN
    BEGIN TRY
        SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
        BEGIN TRANSACTION;

        -- Check if ticket already exists for this reservation
        IF EXISTS (
            SELECT 1 FROM Ticket WHERE ReservationID = @ReservationID
        )
        BEGIN
            RAISERROR('Ticket has already been issued for this reservation.', 16, 1);
            ROLLBACK TRANSACTION;
            RETURN;
        END

        -- Generate a unique EboardingNumber using timestamp and reservation ID
        DECLARE @EboardingNumber NVARCHAR(30);
        SET @EboardingNumber = 'EBD' + CAST(DATEPART(SECOND, SYSDATETIME()) AS NVARCHAR) +
            CAST(DATEPART(MILLISECOND, SYSDATETIME()) AS NVARCHAR) +
            CAST(@ReservationID AS NVARCHAR);

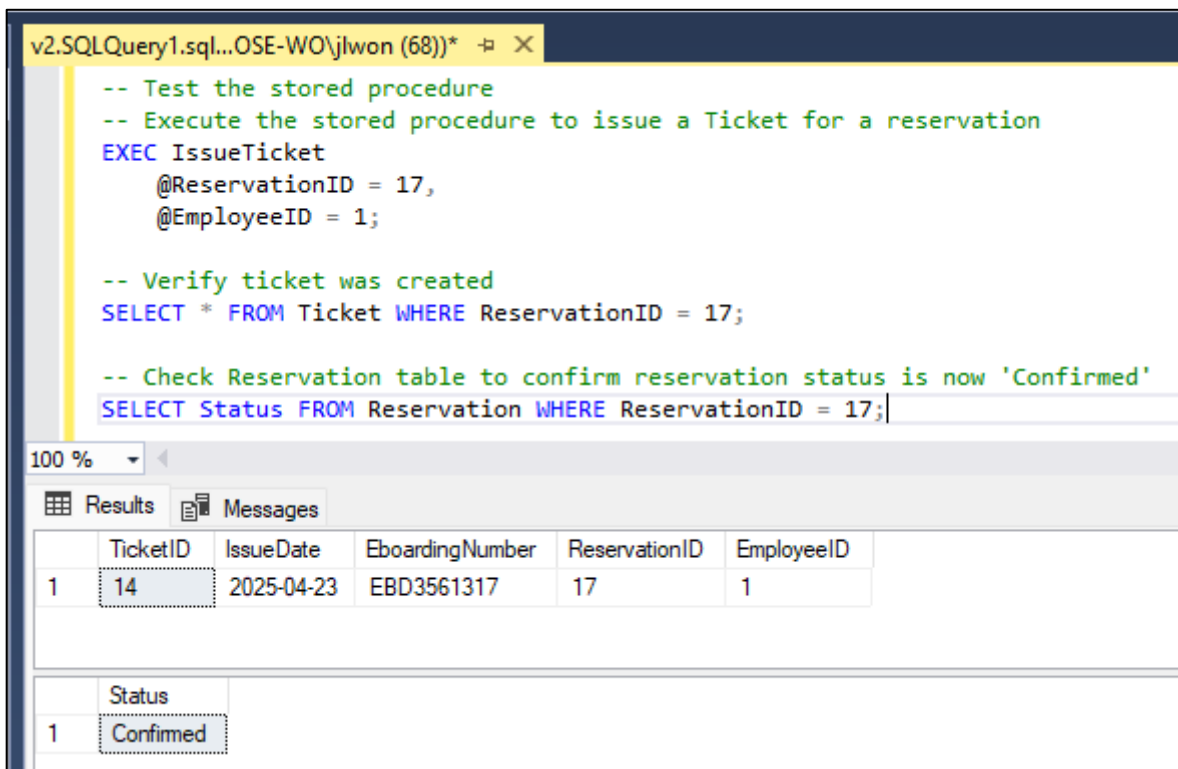
        -- Marks the reservation as Confirmed
        UPDATE Reservation
        SET Status = 'Confirmed'
        WHERE ReservationID = @ReservationID AND Status != 'Confirmed';

        -- Insert the new ticket
        INSERT INTO Ticket (
            IssueDate, EboardingNumber, ReservationID, EmployeeID
        )
        VALUES (
            CAST(GETDATE() AS DATE), @EboardingNumber, @ReservationID, @EmployeeID
        );

        COMMIT TRANSACTION;
    END TRY
    BEGIN CATCH
        ROLLBACK TRANSACTION;
        PRINT ERROR_MESSAGE();
    END CATCH
END;
```


The procedure was tested by issuing a ticket for an existing reservation and verifying that the ticket was inserted and the reservation status updated accordingly, as seen in Figure 47.

Figure 47: Test Stored Procedure to Issue a Ticket.



The screenshot displays a SQL query window with the following code:

```
-- Test the stored procedure
-- Execute the stored procedure to issue a Ticket for a reservation
EXEC IssueTicket
    @ReservationID = 17,
    @EmployeeID = 1;

-- Verify ticket was created
SELECT * FROM Ticket WHERE ReservationID = 17;

-- Check Reservation table to confirm reservation status is now 'Confirmed'
SELECT Status FROM Reservation WHERE ReservationID = 17;
```

Below the query, the 'Results' tab shows two tables. The first table, 'Ticket', contains one row with the following data:

	TicketID	IssueDate	EboardingNumber	ReservationID	EmployeeID
1	14	2025-04-23	EBD3561317	17	1

The second table, 'Reservation', contains one row showing the status update:

	Status
1	Confirmed

1.5 Additional Recommendations

1.5.1 Data Integrity and Concurrency Recommendations

Regarding data integrity, the database implementation contains features that support it, such as the enforcement of a relational model, transaction control mechanisms that prevent incomplete operations resulting in inconsistent data, triggers and constraints. All these features should be relied upon to maintain integrity, avoiding manual data manipulation, when extending functionality or scaling the solution. Moreover, the client should not rely only on these features but should

complement them with strict input validation at the application level, such as using appropriate front-end components like date pickers to select reservation dates and client and server-side input validation and enforcement of business rules, such as validating email formats and avoiding reservation dates in the past. Moreover, data quality auditing should be implemented to monitor input anomalies.

Concerning concurrent access to data, although the database implementation includes transaction control, row-level locking, and the use of the `SERIALIZABLE` isolation level to ensure data integrity under concurrent usage, these mechanisms could result in the SQL Server's internal deadlock detection terminating one or more concurrent transactions or even throwing a system exception error message, for instance if two users attempt transactions to reserve the same seat at the same time. To gracefully handle these situations and enhance the user experience, the application layer should implement mechanisms to catch these errors and retry the transaction after some wait time, for instance, giving a first seat reservation transaction time to complete, resulting in a second user either seeing that the seat has been reserved and selecting another, or successfully reserving the same seat if it is still available, instead of receiving low-level database error messages.

1.5.2 Security Recommendations

Even though security was implemented using SQL Server roles, in production, the database should be accessed using a service account from an application layer and rely on this layer for role-based authentication and authorisation. Additionally, hashing and encryption should be considered for sensitive fields such as Employee passwords. A password rotation policy should also be in place.

If the system and number of roles grow and become more complex, database objects like tables, views, and procedures could be grouped into different SQL Server namespace schemas limiting access according to what each area or department, such as the Ticketing Department, needs to access, enhancing organisation, access control, and security. Also, network operating system user groups could be linked and mapped to particular database logins, roles and database users for this purpose.

Lastly, SSL encryption should be used for securing communications between the database and the application, for instance, between a head office server or cloud instance where the database is hosted and the application running on a computer at the airport counter, accessing it.

1.5.3 Backup and Recovery Recommendations

Although a .bak file is provided, it is advisable, for long-term deployment and system scaling, to implement a formal backup policy, which should include an automated backup strategy, including full and differential backups, stored in secure, ideally off-site, locations, along with transaction log backups and a retention policy to indicate how long backups are kept. Additionally, recovery testing should be scheduled periodically to ensure data can be restored quickly in the event of failure.

1.6 Conclusions

In conclusion, through the use of the relational model and well-defined constraints, concurrency management, transaction control, and triggers to enforce business logic, ensuring data integrity, security, and consistency, the implemented database system provides a successful solution for the functionality required, including core business operations such as reservations, issuance of tickets, and seat management.

Thanks to the flexibility and scalability of the design, the system is ready for further development, implementing the recommendations provided, to support additional functionality as it becomes required.