CSC7052 - Database Report Submission

Lee Service - Student ID: 40312791

Overview of Report:

This report and accompanying video presentation propose a real-world flight booking system, created in PHPmyAdmin using SQL which has been modelled on that of easyJet.com.

With no access to easyJet's flight booking system in the first phase of this project, I worked iteratively on an Entity Relationship Diagram (ERD) within a group of students to provide a structural basis for our SQL database. Once this was achieved, I continually evolved my relational model further in order to lean closer to easyJet's booking model before utilising a variety of SQL statements to finalise all of the data values inside the database for demonstration.

Throughout this report, I will outline the rationale behind my database design decisions, design assumptions, selected data types and database conventions, then present possible improvements to enhance its functionality within the conclusion. An appendix will serve to provide the SQL statements which will be utilised on video, alongside additional images of the final result.

An introduction to the database ERD:

Peter Chen proposed the concepts of the Entity Relation (ER) Model and Entity Relationship Diagram (ERD) in his paper *The Entity-Relationship Model-Toward a Unified View of Data*[1]. He presented that the ER model takes a 'more natural view that the world consists of entities and relationships', incorporating semantic information about the real world when identifying logical views of data to provide a structured design for databases. Chen stated that there are four steps in designing a database using an ER model and that the designer must:

> "(1) identify the entity sets and the relationship sets of interest; (2) identify semantic information in the relationship sets...(3) define the value sets and attributes (4) organize data into entity/relationship relations and decide primary keys".

Chen defined "entity sets" as "thing[s] which can be identified distinctly… which exist within our minds or are pointed at with fingers". Entities may relate to several attributes, which define or describe the entity through their semantic relationships. For example, the value "Child" can be expressed as an attribute to the entity "PassengerType" or the value "12" can be expressed as an attribute to the entity "Age".

Chen stated that "it is always a difficult problem to maintain data consistency following insertion, deletion, and updating of data in the database". Recognising core relationships before the database is programmed helps the designer maintain better data integrity and consistency as they will better understand how to organise data using the ER model because the consequences of insertion, deletion, and updating operations will be more clearly defined.

Through designing the ERD, it was easier to dynamically scale the foundations of our database because entities and attributes can be considered or eliminated without the need to insert them via SQL and test them - this was enormously beneficial to a group which worked iteratively over multiple weeks to refine our ERD before programming the final result.

Design Assumptions and evolving the group ERD:

Imitating Chen's view that the ERD should be modelled on the requirements of users within the real world, my group and I listed assumptions about our database to help form the basis for the first version of our ERD, along with the fundamental entities of which our database would store data on.

It was unnecessary for us to consider every aspect of easyJet's site when building our ERD as we only needed to consider the intrinsic features of the flight booking process. To justify our decisions on the relevant entities, relationships, and attributes, we walked through the easyJet site to gain an understanding of where user inputs were required if they intended on making a booking.

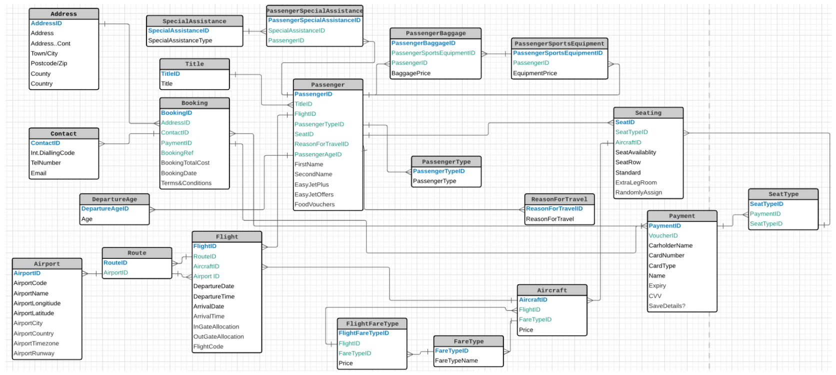
We believed that the following assumptions were core to the experiences of booking a flight using a site like easyJet.com:

- Browse a variety of **airport routes** to suit their needs.
- Make a **booking** for their selected destination and add multiple **passengers** if necessary.
- Understand **flight** details / the flight timetable to fit their schedule.
- Personalise their booking to their individual requirements (E.g. **Fare type, seating type**, their **luggage** needs, add an in flight meal) that would influence the cost of their booking.
- Understand the **cost** of their booking and be able to make a **payment** for it.

As we didn't have access to the easyJet database, or an understanding of their technical infrastructure, we made a number of other assumptions and provided alternative solutions to these in our databases:

- We could only assume that easyjet encrypts user card details in a secure database to Payment Card Industry (PCI) standards. To obscure confidential user details, we relied on AES encryption to demonstrate a simple form of encryption in PHP, covered in greater detail below.
- We considered if data needed to be stored in a multilingual system for European travellers visiting the easyJet site, however our flight booking system had to be completed in English, so it was deemed out of scope for this project.
- We assumed that randomly assigned seating (if a user does not select a seat) would be a seating type within the database. As we investigated this, we found that it was more likely to be conducted by a programming language so I removed this functionality from the ERD and instead, created a seat type called 'No Seat Selected' for demo purposes.
- We were unsure whether or not programming languages were responsible for live inventory (like seating) and price rates, or if there was something set in the database that determined this at milestones, for example - a Boolean in the database might detect if X% of seating was booked for a flight, then increase prices by Y%. As we needed a master price list for demonstration purposes, we decided to use a **LineItem** table to manually update prices for bookings.

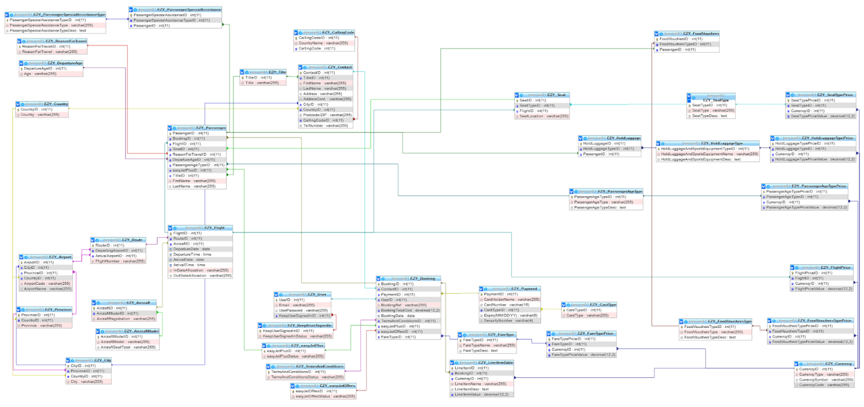
Once we understood the core entities involved with a flight booking system, we began to lay foundations for the cardinality of our relationships between entity tables and their attributes. The first draft of the ERD outlines these relationships but has roughly half of the amount of tables as the final version (see appendix), with no thought provided to data types. We normalised a number of tables such as the **Title** or **Passenger Type** table by reducing their size or by breaking them into smaller tables to reduce data redundancy and minimise inconsistent data dependencies. We were unaware of the extent to which we would have to do this in order to make the relational model more informative to users. The first draft of the group design can be seen below.



First draft of group ERD, constructed in LucidChart. Zoom for detail or see appendix for PDF.

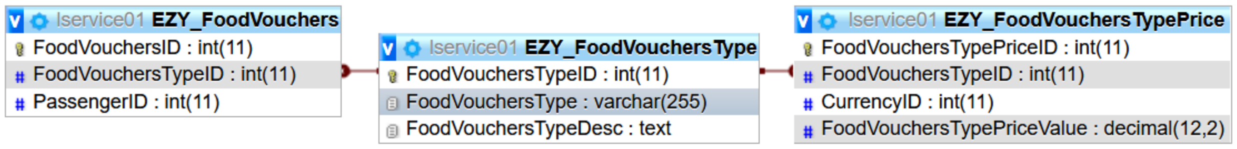
Changes made to Final ERD:

After group work concluded, I changed a number of design decisions in the ERD to address shortcomings in the final model of my database.



Final version of Database in PHP Designer view. Zoom for detail or see Appendix for PDF.

Normalisation of Food Vouchers:
I normalised Food Vouchers into their own table and linked this to the **Passenger** table to provide a 'menu' of different types of food vouchers rather than a Boolean status of whether or not the user has accepted a food voucher for their flight. I applied prices to each of these vouchers in different currencies, with the idea that the real world user could choose a convenient option like lunch or a kids meal at a reduced rate (-20%) before the flight to save money. This would also secure additional revenue for the airline.

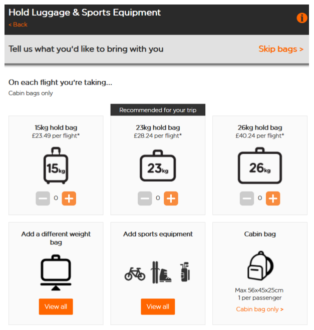


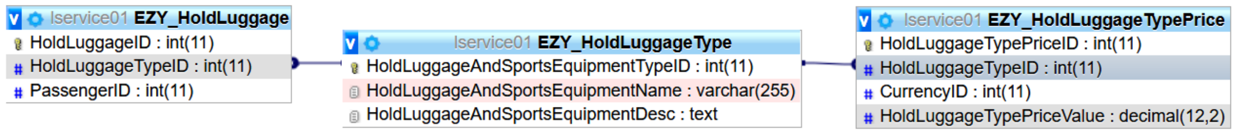
lservice01 **EZY_FoodVouchers**
FoodVouchersID : int(11)
FoodVouchersTypeID : int(11)
PassengerID : int(11)

lservice01 **EZY_FoodVouchersType**
FoodVouchersTypeID : int(11)
FoodVouchersType : varchar(255)
FoodVouchersTypeDesc : text

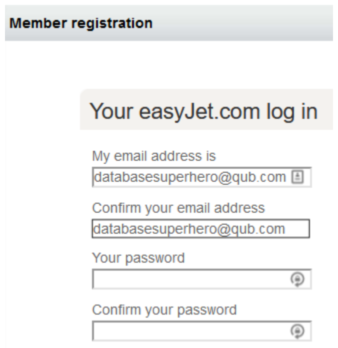
lservice01 **EZY_FoodVouchersTypePrice**
FoodVouchersTypePriceID : int(11)
FoodVouchersTypeID : int(11)
CurrencyID : int(11)
FoodVouchersTypePriceValue : decimal(12,2)

Reduced redundancy regarding Sports Baggage & Baggage Type:
Both sports baggage and baggage types like 26kg suitcases go straight into the hold at weigh-in. It made logical sense to group these together as a customer's total booking cost is provided from the **LineItem** table, not the Sports Baggage / Baggage Type tables. New **HoldLuggage** / **HoldLuggageType** / **HoldLuggagePrice** tables were created and the other 6 tables were deleted. Additionally, this emulates how easyJet models this section of their system whilst it also makes the database less redundant.




lservice01 EZY_HoldLuggage
HoldLuggageID : int(11)
HoldLuggageTypeID : int(11)
PassengerID : int(11)
lservice01 EZY_HoldLuggageType
HoldLuggageAndSportsEquipmentTypeID : int(11)
HoldLuggageAndSportsEquipmentName : varchar(255)
HoldLuggageAndSportsEquipmentDesc : text
lservice01 EZY_HoldLuggageTypePrice
HoldLuggageTypePriceID : int(11)
HoldLuggageTypeID : int(11)
CurrencyID : int(11)
HoldLuggageTypePriceValue : decimal(12,2)

Updated and encrypted User table:
The email attribute was removed from the contact table and replaced the ‘Username’ field. This small change constrains login details to email addresses rather than varbinary strings (e.g. ‘Lee123’), mirroring easyJet.com’s sign in or account registration processes. This update has the added benefit of the designer being able to query a much smaller table faster, as they just need to match the PK to the relevant email address.


Member registration

Your easyJet.com log in

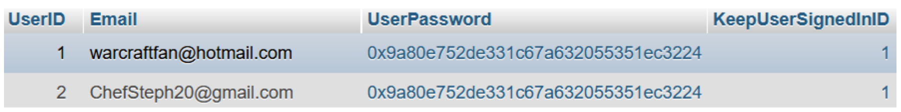
My email address is
databasesuperhero@qub.com

Confirm your email address
databasesuperhero@qub.com

Your password

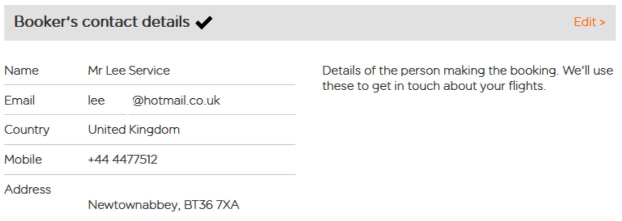
Confirm your password

The **UserPassword** field was changed to varbinary and encrypted using AES_ENCYPT to ensure a higher level of confidentiality and security for the user’s personal details. The key for this table is different from the key within the Payment table (covered below) as easyJet would be required to be GDPR compliant in the way they handle customer details.



UserID	Email	UserPassword	KeepUserSignedInID
1	warcraftfan@hotmail.com	0x9a80e752de331c67a632055351ec3224	1
2	ChefSteph20@gmail.com	0x9a80e752de331c67a632055351ec3224	1

The contact table has one central strong purpose:
The Address table was deleted and its contents were added to the contact table, so one table could be queried to find all customer contact information, rather than needing to query an address and contact table. This reflects how easyJet stores contact details in the booking section further, reducing redundancy.


Booker's contact details ✓ Edit >

Name	Mr Lee Service
Email	lee @hotmail.co.uk
Country	United Kingdom
Mobile	+44 4477512
Address	Newtownabbey, BT36 7XA

Details of the person making the booking. We'll use these to get in touch about your flights.

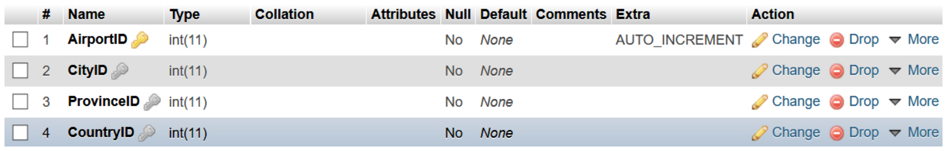
Various Entities were cleaned up to ensure data integrity:

- Airport Latitude and Longitude were two attributes not relevant to user booking so they were deleted.
- Airport Runway and Gate details are normally provided before departure when you arrive at the airport, so were removed because they didn’t hold value for a booking that might take place months in advance.
- Timezone was initially added to the database, but removed after sample data was added; it felt superfluous to the booking experience as long as accurate times were provided within the FlightArrival attribute.
- The Int.Dialling code table was simplified into the ‘CallingCode’ table and multiple columns were removed because they were redundant.
- The currency table was fleshed out for greater context for travellers who may pay in other currencies, extra context was added to each currency in order to reduce ambiguity.
- All Boolean functions of the group were normalised into tables, see varchar data type below.

Database Structure - Naming conventions, PK, AI and FK used throughout:

To distinguish each table from other databases, all tables are written in CamelCase and prefixed with ‘EZY_’ to distinguish them uniquely and prevent update / insert errors if another database was merged onto this one.

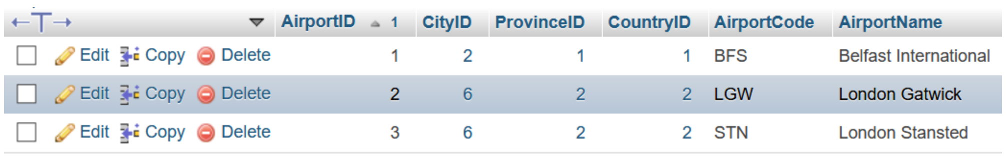
As mentioned above in step 4 of Chen’s method of designing databases, selecting Primary Keys (PKs) are an important part of database design when utilising an ERD. We might store a **Card Number**, **Security Number** and **Expiry Date** in our database to describe a **Payment** entity but we would still require a relationship for this **Payment** entity to link to a unique **Booking** entity which in turn will link to the unique **User** entity who will authorise the payment. There may be millions of users each year that use easyJet's flight booking system who may share the same card details across business or family accounts, or the same booker might make 10 bookings using various payments, so we need to uniquely identify these entities in order to ensure traceability from a user perspective and prevent anomalies from a database management perspective. This is why a Primary Key (PK) is so important in ensuring data integrity. In SQL, a PK must be a uniquely identifiable column which can be semantically linked to other entity tables through the usage of Foreign Key (FK) constraints. In each entity table within my database, the PK is the name of the table, affixed with the letters 'ID' for consistency.



	#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra	Action
☐	1	AirportID	int(11)			No	None		AUTO_INCREMENT	Change Drop More
☐	2	CityID	int(11)			No	None			Change Drop More
☐	3	ProvinceID	int(11)			No	None			Change Drop More
☐	4	CountryID	int(11)			No	None			Change Drop More

AirportID is this table's PK (marked with gold key). CityID, ProvinceID and CountryID are attributes which link back to their respective entities through FK constraints which constrain data input in this Airport table.

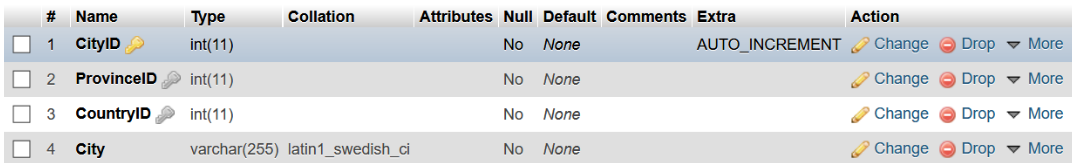
Every instance of a PK in the database is modified with an auto increment (AI) status to ensure uniqueness as each record is inserted. The PK is incremented with a consistent and unique numeric ID which we can use to take advantage of faster, more reliable queries when searching through high volumes of records with frequently altered data elsewhere in the table, for example, if airport routes are being deleted or added to the system every day, it would still be relatively simple for the database designer to query all of the routes by pulling back info from the primary key.



		AirportID ▲ 1	CityID	ProvinceID	CountryID	AirportCode	AirportName
☐	Edit Copy Delete	1	2	1	1	BFS	Belfast International
☐	Edit Copy Delete	2	6	2	2	LGW	London Gatwick
☐	Edit Copy Delete	3	6	2	2	STN	London Stansted

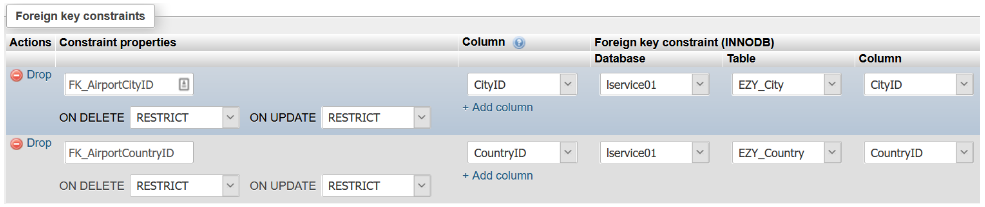
AirportID is being auto incremented, so even if new records are inserted, they will be stored as a unique int value for easy reference.

As a result of consistent naming conventions on every PK and that each one is auto incremented, FKs are easier to create because every PK has the same datatype of **int(11)** throughout the database. This mirrors each FK constraint that was set to the same datatype of **int(11)**. See the image below for an example of these same lengths across the PK and FK indexes in the City table.



	#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra	Action
☐	1	CityID	int(11)			No	None		AUTO_INCREMENT	Change Drop More
☐	2	ProvinceID	int(11)			No	None			Change Drop More
☐	3	CountryID	int(11)			No	None			Change Drop More
☐	4	City	varchar(255)	latin1_swedish_ci		No	None			Change Drop More

Each FK is linked back to a table using a clear naming convention to indicate that they exist as a primary key, allowing the designer to easily understand where data anomalies occurred when sample data was input. See below for an example of this naming convention.



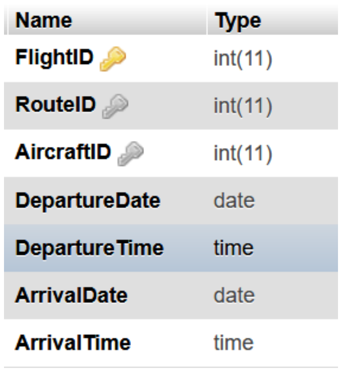
Foreign key constraints

Actions	Constraint properties	Column	Foreign key constraint (INNODB) Database	Table	Column
Drop	FK_AirportCityID ON DELETE RESTRICT ON UPDATE RESTRICT	CityID + Add column	lservice01	EZY_City	CityID
Drop	FK_AirportCountryID ON DELETE RESTRICT ON UPDATE RESTRICT	CountryID + Add column	lservice01	EZY_Country	CountryID

Database Structure - Prefixing & DataTypes:

Like PKs and FKs having a consistent data structure, data types throughout the database are also consistent to ensure that all data input by the user or designer is usable and within logical constraints. The following info explains the rationale behind the usage of each data type.

int(11) - The data type consistently used for all PKs and numeric values, to a range of 11 integer digits which comfortably provides room for all inserted sample data.



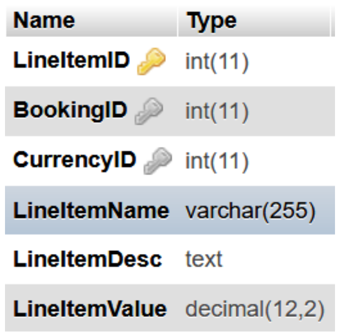
Name	Type
FlightID	int(11)
RouteID	int(11)
AircraftID	int(11)
DepartureDate	date
DepartureTime	time
ArrivalDate	date
ArrivalTime	time

date - Date format datatype used for **BookingDate** attribute and inside the flight table to provide details on flight schedule in the format MM/DD/YY.

time - Used inside the flight table to provide clarity on flight schedule in the format HH:MM:SS. Seconds were chosen to reduce ambiguity.

varchar(255)* - Datatype consistently selected for all short string values such as Seat Types ("Upfront", "Extra Legroom Upfront") and phone numbers where int would be unsuitable due to numbers starting with a 0 infront of the phone number.

- In situations where Boolean functions may have been required - e.g. Has the user accepted terms and conditions? True / False - this was instead normalised into its own table with the varchar data type used for 'Yes' or 'No' in case additional conditions needed to be added in the future by the database designer.



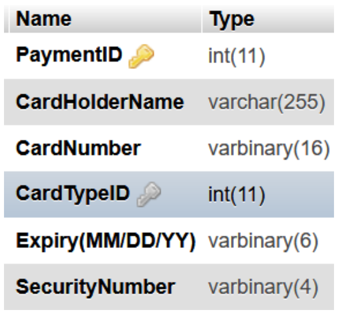
Name	Type
LineItemID	int(11)
BookingID	int(11)
CurrencyID	int(11)
LineItemName	varchar(255)
LineItemDesc	text
LineItemValue	decimal(12,2)

text (2550) - Consistently used for anything which may have a lengthy description, such as an item that may be promoted during a flight such as cologne or the special assistance types available to passengers.

decimal (12,2) - Consistently used for pricing items in tables like the **LineItem** Table - rounds to 2 decimal places to ensure values are rounded to full penny / cent.

varbinary* - Used in the **User** and **Payment** tables when confidential details need to be encrypted using AES encryption queries. *Length was set to different length values for the following:

- Set to (16) for **CardNumber** as this is universally the max number of characters for a debit / credit card. Allowing a user to include more would cause transaction problems.
- Set to (6) for **Expiry(MM/DD/YY)** for encryption purposes as this was the simplest form to enable AES_ENCRYPT on this attribute.
- Set to (4) for **SecurityNumber** due to standard cards (E.g. Visa, Mastercard) having 3 security numbers, whereas AMEX has 4.



Name	Type
PaymentID	int(11)
CardHolderName	varchar(255)
CardNumber	varbinary(16)
CardTypeID	int(11)
Expiry(MM/DD/YY)	varbinary(6)
SecurityNumber	varbinary(4)

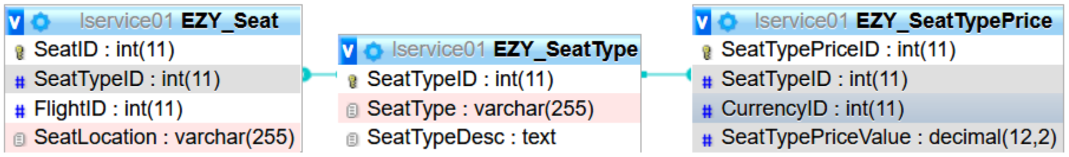
Database Normalisation:

It was important to consider normalisation when tables were being created to reduce data duplication and further enhance data integrity.

Each table was set up so that all column names were unique and any values that were stored in each table were relevant to the entity. This meant that each table served one strong purpose and data could be retrieved easily. It didn't matter what order the data was stored in, helping minimize potential data anomalies. These features follow the first normal form (1NF) of normalisation.

Attributes were cut from tables which didn't serve a strong purpose within the table, reducing partial dependencies. If these attributes were important in their own right, they were added to their own table and normalised, in order to fulfill the second normal form (2NF). This minimised repetition of data in each table.

Conveniently, with each FK constraint referencing a PK of another table, rather than an attribute from the same table, this meant that my tables also followed the third normal form (3NF) as there were no transitive dependencies.



lservice01 EZY_Seat
SeatID : int(11)
SeatTypeID : int(11)
FlightID : int(11)
SeatLocation : varchar(255)

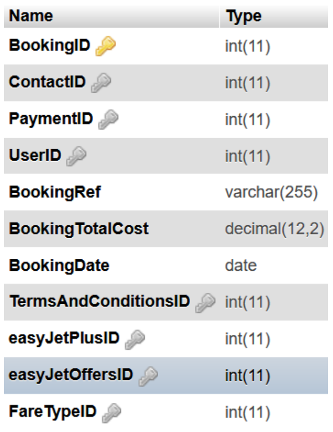
lservice01 EZY_SeatType
SeatTypeID : int(11)
SeatType : varchar(255)
SeatTypeDesc : text

lservice01 EZY_SeatTypePrice
SeatTypePriceID : int(11)
SeatTypeID : int(11)
CurrencyID : int(11)
SeatTypePriceValue : decimal(12,2)

An example - The Seat table provides seat locations (E.g. Row 1, Seat A).
SeatType sits as a service table between the two to define what type of seat the user is selecting.
SeatTypePrice indicates the price of the seat selected in the users selected currency.

Core Entities and Relationships:

Booking:
The core table of the database concerning flight booking, holding booking information for each user.

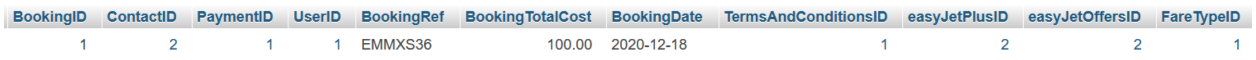


Name	Type
BookingID	int(11)
ContactID	int(11)
PaymentID	int(11)
UserID	int(11)
BookingRef	varchar(255)
BookingTotalCost	decimal(12,2)
BookingDate	date
TermsAndConditionsID	int(11)
easyJetPlusID	int(11)
easyJetOffersID	int(11)
FareTypeID	int(11)

Regarding FKs, the Booking table draws on **contact** information, **payment** information, the unique **user** account of the booker, their **easyJetPlus status,** whether or not they have accepted the **terms and conditions** of flying and **easyJetOffers** and, if the whole booking is a Flexi (provides users with various perks) or standard booking via the **FareType** FK. It also provides the user with a total **cost**, and details of their booking itself (the **reference** and the **date** the booking was made).

Considering normalisation on this table, the booking ref provides the **BookingID** with a string value which will be pulled to the **Passenger** table (see below) which uses the BookingID as a FK, allowing many passengers to be added to one booking. The advantage of this setup is that bookings can also be made on behalf of other people, where the booker doesn't need to declare themselves as a passenger.

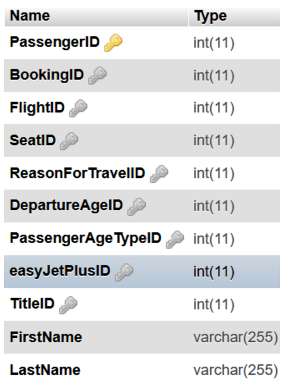
For demonstration purposes, the **BookingTotalCost** can be updated via a SQL query directly from the **LineItem** table which has a one to one relationship to the BookingID for the final cost to the booker. In real life, I assume that prices would be provided to the **Booking** table dynamically using programming languages. A demonstration of how important the LineItem table is to the Booking table can be seen in video query [2]+[3].



BookingID	ContactID	PaymentID	UserID	BookingRef	BookingTotalCost	BookingDate	TermsAndConditionsID	easyJetPlusID	easyJetOffersID	FareTypeID
1	2	1	1	EMMXS36	100.00	2020-12-18	1	2	2	1

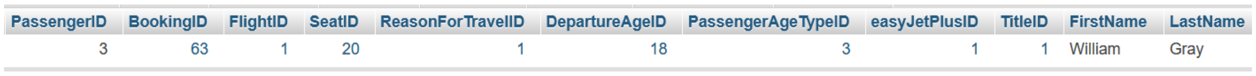
Sample record of the Booking table, updated with a price from LineItem table via SQL update query.

Passenger:
As each passenger has unique requirements when flying, the Passenger table is important because it is central for multiple many-to-many relationship tables such as **PassengerSpecialAssistance** which caters to passengers with reduced mobility or disabilities, **HoldLuggage** which allows users to select a variety of luggage types and, **FoodVouchers** which provides in-flight consumables at a reduced rate before boarding. These tables use the **PassengerID** as an FK, allowing users to personalise their flying experience before departure.



Name	Type
PassengerID	int(11)
BookingID	int(11)
FlightID	int(11)
SeatID	int(11)
ReasonForTravelID	int(11)
DepartureAgeID	int(11)
PassengerAgeTypeID	int(11)
easyJetPlusID	int(11)
TitleID	int(11)
FirstName	varchar(255)
LastName	varchar(255)

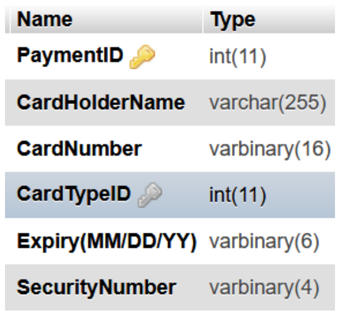
We can quickly reference the **first name** and **last name** of each passenger, their **title** and what **flight** and **seat** they've selected through queries tied to the **PassengerID**. This table can be particularly useful for analytical or marketing purposes as easyJet can track things like how many business users they may have flying by simply querying the **ReasonForTravel** table, or querying the **easyJetPlusID** to understand how many users passengers are utilising easyJetPlus for additional revenue. Crucially, a many to one relationship exists between the passenger table and the booking table. One booking can contain multiple passengers, but not vice-versa, which is why the **BookingID** FK is present within this table.



PassengerID	BookingID	FlightID	SeatID	ReasonForTravelID	DepartureAgeID	PassengerAgeTypeID	easyJetPlusID	TitleID	FirstName	LastName
3	63	1	20	1	18	3	1	1	William	Gray

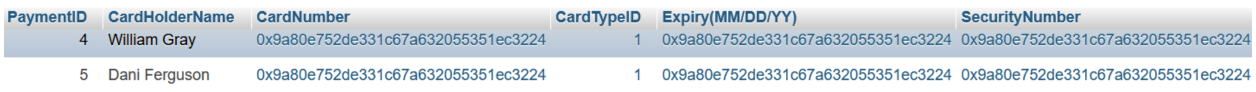
Payment:

Customer card details are stored in the Payment table in order for users to pay for their booking. The Payment table contains the PK **PaymentID** which is an FK constraint in the **Booking** table. Credit card type was normalised into the **CardType** table as this is a standard e-commerce practice to ensure users may only make payments with card providers the airline accepts (E.g. Visa).



Name	Type
PaymentID	int(11)
CardHolderName	varchar(255)
CardNumber	varbinary(16)
CardTypeID	int(11)
Expiry(MM/DD/YY)	varbinary(6)
SecurityNumber	varbinary(4)

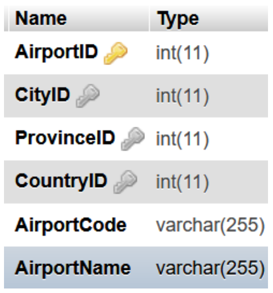
Sensitive payment information such as the 16 digit card number, 3-4 digit security number and the expiry date are all encrypted via AES encryption with a key unique to that table to enhance security of these confidential details. AES limitations will be discussed below in the improvements section.



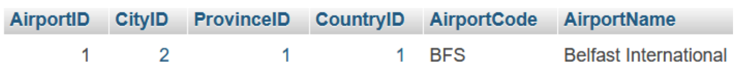
PaymentID	CardHolderName	CardNumber	CardTypeID	Expiry(MM/DD/YY)	SecurityNumber
4	William Gray	0x9a80e752de331c67a632055351ec3224	1	0x9a80e752de331c67a632055351ec3224	0x9a80e752de331c67a632055351ec3224
5	Dani Ferguson	0x9a80e752de331c67a632055351ec3224	1	0x9a80e752de331c67a632055351ec3224	0x9a80e752de331c67a632055351ec3224

Airport Table + Route Table:

The Airport table uses various FKs to link to the **City**, **Province** and Country tables to provide location details to the user when they are selecting their destination. It is important that each airport is assigned a unique **airport code** and **name** to eliminate ambiguity for users as there may be multiple airports within one city or multiple airports with the same name within a region or country.

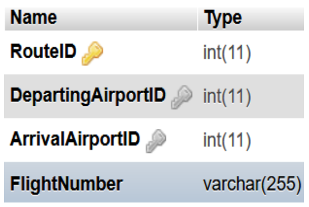


Name	Type
AirportID	int(11)
CityID	int(11)
ProvinceID	int(11)
CountryID	int(11)
AirportCode	varchar(255)
AirportName	varchar(255)

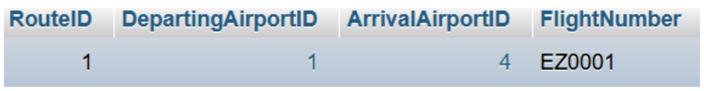


AirportID	CityID	ProvinceID	CountryID	AirportCode	AirportName
1	2	1	1	BFS	Belfast International

The Route table uses this **AirportID** to reference a **departing AirportID** and **arrival AirportID** and establishes a **RouteID** with a **flight number** in order to uniquely identify this route.

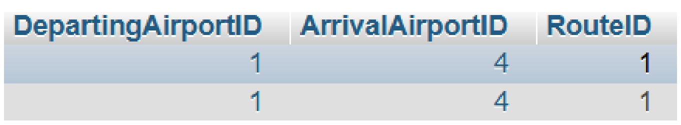


Name	Type
RouteID	int(11)
DepartingAirportID	int(11)
ArrivalAirportID	int(11)
FlightNumber	varchar(255)



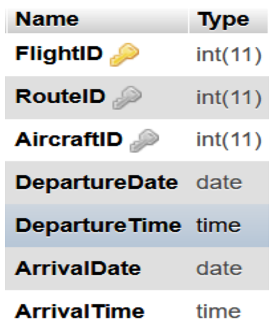
RouteID	DepartingAirportID	ArrivalAirportID	FlightNumber
1	1	4	EZ0001

The **Route** table is then called on by the **Flight** table's **RouteID** FK, see below. By using a JOIN, it is possible to retrieve the departing and arrival airport on a route [See appendix section additional queries for SQL query].

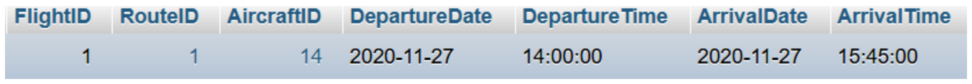


DepartingAirportID	ArrivalAirportID	RouteID
1	4	1
1	4	1

Flight Table:
FlightID is used as an FK within the **Passenger** table to connect what flight the passenger is going on and how much they will have to pay for the base cost of the journey. The Flight table utilises an **AircraftID** FK to provide information on the **registration** and **model** of aircraft the passenger will be boarding. Flight **departure** and **arrival times** and **dates** are listed within this table to allow real world users to plan accordingly around easyJet's flight schedule. This table can be searched or filtered so an individual could search for flights departing or arriving at specific times depending on their preference. The **Seat** table uses a **FlightID** FK so that many seats may be applied to one flight.



Name	Type
FlightID	int(11)
RouteID	int(11)
AircraftID	int(11)
DepartureDate	date
DepartureTime	time
ArrivalDate	date
ArrivalTime	time



FlightID	RouteID	AircraftID	DepartureDate	DepartureTime	ArrivalDate	ArrivalTime
1	1	14	2020-11-27	14:00:00	2020-11-27	15:45:00

Potential Improvements:

This project relies on phpMyAdmin's AES_ENCRYPT feature in order to remove clear text values and encrypt them in a more secure varbinary format. As mentioned previously, this was applied to the **User** tables **Password** attribute and **CardNumber**, **ExpiryDate** and, **SecurityNumber** in the **Payment** table to present an awareness of real-world PCI requirements and because it was within the limitations of phpMyAdmin to demonstrate. Whilst each of these tables have their own unique encryption key, a problem lies in that the key is applied to the whole table and not on an individual basis. If a table's encryption is cracked, then all of the data can be easily decrypted.

According to PCI standards, if you intend to store cardholder data (CHD), you must do so with a legitimate business reason because this information can be targeted by cybercriminals.[2] Whilst AES encryption is a method of cryptography that is accepted by the PCI, additional protection can be applied to the database by:

- Encrypting each record with a separate unique key, allowing for a much higher level of security if the key of one record is cracked by an attacker.
- Truncating confidential data that is entered into the database, for example, recording only the last four digits of the users 16 digit card number for each payment.
- Utilising a one way encryption method like hashing. Even if only one person knows the unique encryption key, this can be considered a weakness of the database as that person can decrypt the confidential information.

If security issues are tackled, then considerations for alternative payment types like Paypal details could be stored within the database both to emulate easyJet's site and for convenience to the user.

In the Booking table, the **BookingRef** could be used as the PK rather than **BookingID.** In its current form, passengers will know if they are passenger 1, 2, 3, etc. As the Booking Ref has to be uniquely generated per booking - this would mimic closer to the easyJet booking reference system. This principle could also be applied to **FlightNumber** in the **Route** table as this route must be unique each time it is generated, so could be used as the PK in lieu of **RouteID**.

In the real world, easyJet Plus membership provides benefits like free seating allocation and a free additional cabin bag, however the **easyJetPlus** table only stores that the user has, or has not accepted, easyJet Plus, with no bearing on other tables. Whilst it was deemed not to be key to a flight booking system, it would be a feature that would be good to expand on in improvements to the database.

This database operates on a one-way system, which does not represent real-world flight booking services; to do so, most details need to be stored twice (e.g. two different booking references and two different payments would need to be made) for a user to travel to and from their destination. In further improvements, a **returning airport**, **returning route** and, **returning flight** table would need to be related to each passenger to enable this.

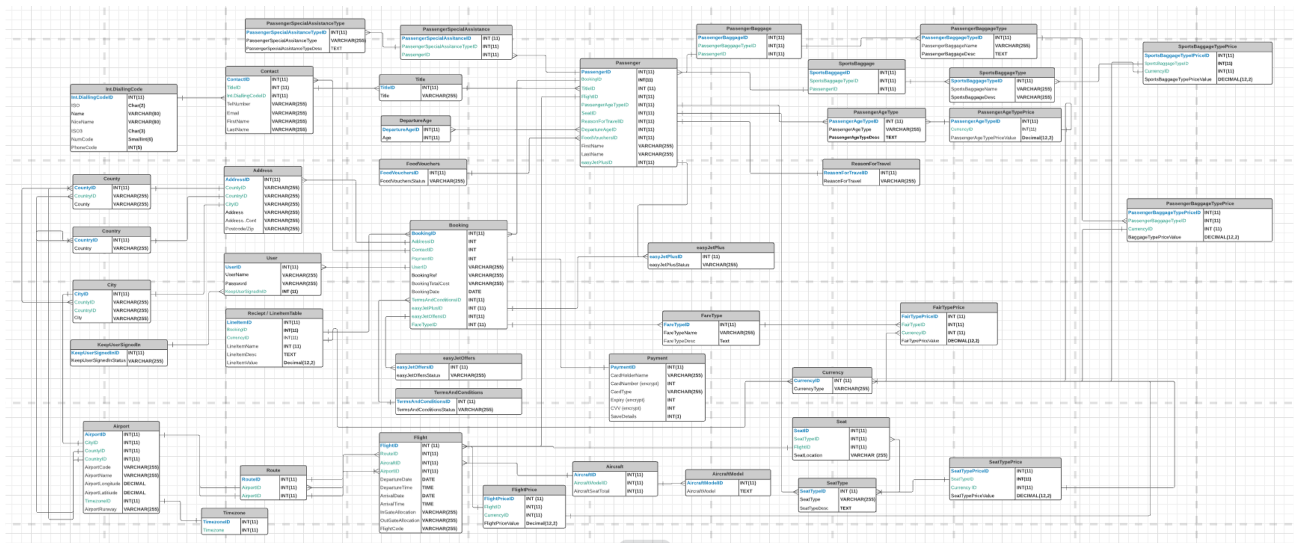
Conclusion:

To summarise, the goal of this database system is to replicate a functional flight booking system, and this has been achieved with a variety of aspects which allow users to create a booking on the front end of the system and have that information stored within the backend database. Without access to easyJet's backend system we have made reasonable assumptions about storing user information in a database. Users are successfully able to personalise their journey and assign multiple passengers, receive a total cost and pay for their booking using card details which are encrypted in a secure way which fulfills our core design assumptions.

These tables can be used in a variety of functions, from creating user accounts to providing marketing or analytics to business management through customer statuses being tracked through tables like **easyJetOffers** and **ReasonForTravel**, giving easyJet the option to provide additional value to their users or target opportunities to increase revenue. Through querying how many customers are selecting food vouchers, seat types or fare types, easyJet can get a better idea of their customer demographic and tailor the business to their users needs.

Additionally, the use of PKs, AI and the consistent setup of FK constraints ensures minimum data redundancy and while further improvements can be made to the system and additional security should be considered for confidential details before real-world usage, the database structure has been designed so that it can flexibly incorporate future improvements without requiring significant alterations because the tables have been normalised to the 3NF.

Appendix & SQL Queries:



Final draft of group ERD, constructed in LucidChart

Click here for the full scale PDF of our groups first basic ERD.

Click here for the full scale PDF of the final group ERD.

Click here for the full scale PDF of my final Lucid ERD.

Click here for the full scale of my Final Database in phpMyAdmin Designer View.

References used within database report:

[1] - Peter Chen - *The Entity-Relationship Model-Toward a Unified View of Data:* http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.526.369&rep=rep1&type=pdf

[2] PCI Rules for storing credit card numbers in a database: https://www.globalpaymentsintegrated.com/en-us/blog/2019/11/25/pci-rules-for-storing-credit-card-numbers-in-a-database

---

Video Queries:

[1] Transaction of adding a booking and two passengers.

START TRANSACTION;

INSERT INTO EZY_User (UserID, Email, UserPassword, KeepUserSignedInID) VALUES (NULL, 'JoeBiden@yahoo.com', AES_ENCRYPT('password', 'mySecretKey'), '1');

SET @userid = LAST_INSERT_ID();

INSERT INTO EZY_Payment(PaymentID,CardHolderName, CardNumber, CardTypeID, Expiry(MM/DD/YY), SecurityNumber) VALUES (NULL, AES_ENCRYPT('Joe Biden','mySecretKey1'), AES_ENCRYPT('111111111111111','mySecretKey1'), '3', AES_ENCRYPT('040421','mySecretKey2'), AES_ENCRYPT('8080','mySecretKey3'));

SET @paymentid = LAST_INSERT_ID();

INSERT INTO EZY_Contact (ContactID, TitleID, FirstName, LastName,Address,AddressCont.,CityID,CountryID,Postcode/ZIP, CallingCodeID, TelNumber) VALUES (NULL, '1', 'Joe', 'Biden','2 Malone Road','Belfast City','2','1','AB1 5GH','225', '7751250000');

SET @contactid = LAST_INSERT_ID();

INSERT INTO EZY_Booking (BookingID, ContactID, UserID ,PaymentID,FareTypeID, easyJetOffersID, easyJetPlusID, TermsAndConditionsID, BookingDate, BookingRef, BookingTotalCost) VALUES (NULL, @contactid, @userid, @paymentid,'1', '2', '1', '1', '2020-12-03', 'EZTZT', '0.00');

SET @bookingid = LAST_INSERT_ID();

INSERT INTO EZY_Passenger (PassengerID, BookingID, TitleID, FirstName, LastName, FlightID, SeatID, PassengerAgeTypeID, DepartureAgeID, ReasonForTravelID, easyJetPlusID) VALUES (NULL, @bookingid, '1', 'Joe', 'Biden', '21', '4', '3', '18', '1', '2');

INSERT INTO EZY_Passenger (PassengerID, BookingID, TitleID, FirstName, LastName, FlightID, SeatID, PassengerAgeTypeID, DepartureAgeID, ReasonForTravelID, easyJetPlusID) VALUES (NULL, @bookingid, '2', 'Mary', 'Biden', '21', '5', '3', '18', '1', '2');

COMMIT;

[2] Inserting multiple items into the line item table.

INSERT INTO EZY_LineItemTable (LineItemID, BookingID, CurrencyID, LineItemName, LineItemDesc, LineItemValue) VALUES (NULL, '86', '1', 'Pint of beer','Harp Ice','4.99'); INSERT INTO EZY_LineItemTable (LineItemID, BookingID, CurrencyID, LineItemName, LineItemDesc, LineItemValue) VALUES (NULL, '86', '1', 'Ski bag','Ski bag stored in hold','44.99'); INSERT INTO EZY_LineItemTable (LineItemID, BookingID, CurrencyID, LineItemName, LineItemDesc, LineItemValue) VALUES (NULL, '86', '1', 'Upfront Seat','Sit comfortably upfront','18.99'); INSERT INTO EZY_LineItemTable (LineItemID, BookingID, CurrencyID, LineItemName, LineItemDesc, LineItemValue) VALUES (NULL, '86', '1', 'Fare Price','Standard adult ticket','29.99'); INSERT INTO EZY_LineItemTable (LineItemID, BookingID, CurrencyID, LineItemName, LineItemDesc, LineItemValue) VALUES (NULL, '86', '1', 'Ski bag','Ski bag stored in hold','44.99'); INSERT INTO EZY_LineItemTable (LineItemID, BookingID, CurrencyID, LineItemName, LineItemDesc, LineItemValue) VALUES (NULL, '86', '1', 'Upfront Seat','Sit comfortably upfront','18.99'); INSERT INTO EZY_LineItemTable (LineItemID, BookingID, CurrencyID, LineItemName, LineItemDesc, LineItemValue) VALUES (NULL, '86', '1', 'Fare Price','Standard adult ticket','29.99');

SELECT * FROM EZY_LineItemTable;

[3] Update Booking Total Cost from Line Item table

SET @bookerid = 85; SET @sumTotalCost = (SELECT SUM(LineItemValue) FROM EZY_LineItemTable WHERE BookingID = @bookerid); UPDATE EZY_Booking SET BookingTotalCost = @sumTotalCost WHERE EZY_Booking.BookingID = @bookerid; SELECT * FROM EZY_Booking;

[4] Count distinct countries a user can fly to

SELECT * from EZY_Country; SELECT COUNT(DISTINCT ArrivalAirportID) FROM EZY_Route;

[5] All flights on a specific route, within 7 days of user specified date

SET @dateselected = '2020-11-25', @userroute = '1';

SELECT EZY_Flight.RouteID, EZY_Airport.AirportName, EZY_FlightPrice.CurrencyID, EZY_FlightPrice.FlightPriceValue, EZY_Flight.DepartureDate, EZY_Flight.DepartureTime, DATEDIFF(EZY_Flight.DepartureDate, @dateselected)FROM EZY_Flight INNER JOIN EZY_FlightPrice ON EZY_Flight.FlightID = EZY_FlightPrice.FlightID INNER JOIN EZY_Route ON EZY_Flight.RouteID = EZY_Route.RouteID INNER JOIN EZY_Airport ON EZY_Route.ArrivalAirportID = EZY_Airport.AirportID

WHERE EZY_Flight.RouteID = @userroute AND DATEDIFF(EZY_Flight.DepartureDate, @dateselected)<= '7';

[6] Display a sorted list of all flights leaving a specific airport on a specific date and what their departure times are.

SELECT EZY_Flight.FlightID, EZY_Flight.DepartureDate,EZY_Flight.DepartureTime FROM EZY_Flight LEFT JOIN EZY_Route ON EZY_Flight.RouteID = EZY_Route.RouteID RIGHT JOIN EZY_Airport ON EZY_Airport.AirportID = EZY_Route.DepartingAirportID WHERE EZY_Airport.AirportName = 'London Gatwick' AND EZY_Flight.DepartureDate = '2020-11-27' ORDER BY EZY_Flight.DepartureTime ASC;

[7] Display all passengers that bought a food voucher on a specific flight.

SELECT EZY_Passenger.PassengerID, FirstName, LastName, EZY_FoodVouchersType.FoodVouchersType FROM EZY_Passenger INNER JOIN EZY_FoodVouchers ON EZY_Passenger.PassengerID = EZY_FoodVouchers.PassengerID INNER JOIN EZY_FoodVouchersType ON EZY_FoodVouchers.FoodVouchersTypeID = EZY_FoodVouchersType.FoodVouchersTypeID

WHERE FlightID = 1;

Additional Queries:

Queries that were not conducted on video.

Present passenger names in uppercase:
SELECT UCASE (FirstName), UCASE (LastName)
FROM EZY_Passenger
ORDER BY LastName;

Return items greater than the average price from the food voucher table:
SELECT EZY_Flight.FlightID, EZY_Route.FlightNumber FROM EZY_Flight INNER JOIN EZY_Route ON EZY_Flight.RouteID=EZY_Route.RouteID

Display how many airports there are in a specific country:
SELECT EZY_Country.Country, COUNT('Country') AS 'Number of Airports' FROM EZY_Country INNER JOIN EZY_Airport ON EZY_Country.CountryID = EZY_Airport.CountryID WHERE Country = 'Northern Ireland';

Retrive the sum of a booking using the 'AS' function:
SELECT BookingID, SUM(LineItemValue) AS 'Total Value of Booking ID' FROM EZY_LineItemTable WHERE BookingID = 1
GROUP BY BookingID = 1

Decrypting AES Encryption:

```
SELECT AES_DECRYPT (UserPassword,'mySecretKey1') FROM EZY_User WHERE UserID = 4;
```

Insert into flight table:

```
INSERT INTO EZY_Flight (`FlightID`, `RouteID`, `AircraftID`, `DepartureDate`, `DepartureTime`, `ArrivalDate`, `ArrivalTime`)
VALUES (NULL, '1', '1', 'EZY0001', '2020-11-23', '15:00', '4', '2020-11-23', '16:00', '2'), (NULL, '2', '2', 'EZY888', '2020-11-29',
'08:00:00', '2020-11-29', '09:00:00');
```

Pull back departing airport and arrival airport with route using a JOIN:

```
SELECT EZY_Route.DepartingAirportID,EZY_Route.ArrivalAirportID,EZY_Route.RouteID
FROM EZY_Route
INNER JOIN EZY_Flight ON EZY_Route.RouteID = EZY_Flight.RouteID
```