Database Design Report – easyJet Flight Booking System

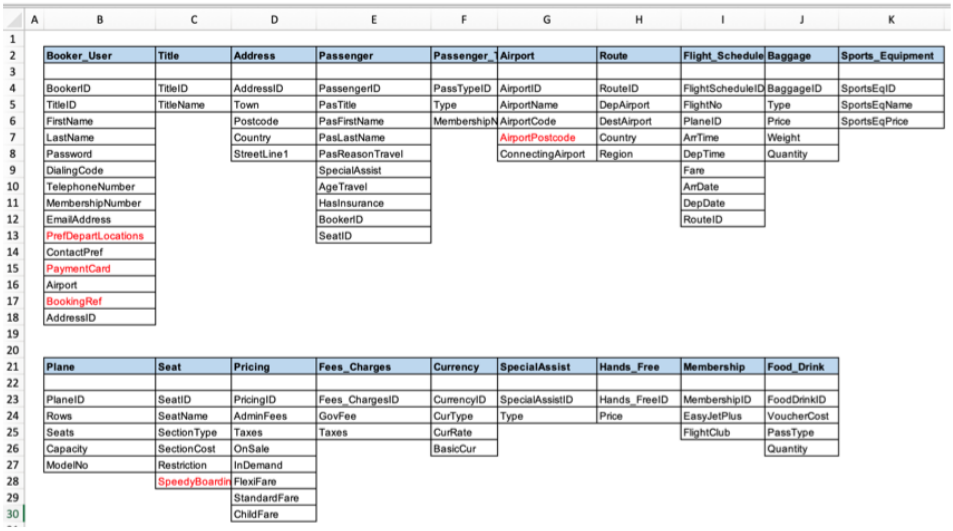
The focus of this project was to undertake the design and implementation of a database for the easyJet flight booking system, mirroring the booking aspects of the website by moulding the database around the front-end flight booking experience.

The progression of this project began with early investigations during group work research and data modelling through the process of reverse engineering the easyjet.com website, developing entities and attributes as well as relationships with thoughts around keys and normalisation. This project then developed individually which led to progression of database design decisions, assumptions, more detailed planning around normalisation and entity relationships, and finally the database implementation, addition of sample data and database functionality exploration through use of SQL queries and data mining techniques.

This project focuses on what Chen describes during his development of entity relationship models, that in order for a database to function to the highest degree, it 'adopts the more natural view that the real world consists of entities and relationships…It incorporates some of the important semantic information about the real world' [1]. With Chen's theories around modelling databases based on real world problems and real user interaction in mind, a database was created for this project based on real issues around holding and linking data in a real-life booking system, and holding the preservation of data integrity with utmost importance.

I. *Early Developments*

The project began focusing primarily on entity discovery based on the reverse engineering of the easyJet booking process as a group. We began by separating entities from attributes and bringing these together in table format as shown in Fig. 1:



Booker_User	Title	Address	Passenger	Passenger_T	Airport	Route	Flight_Schedule	Baggage	Sports_Equipment
BookerID	TitleID	AddressID	PassengerID	PassTypeID	AirportID	RouteID	FlightScheduleID	BaggageID	SportsEqID
TitleID	TitleName	Town	PasTitle	Type	AirportName	DepAirport	FlightNo	Type	SportsEqName
FirstName		Postcode	PasFirstName	MembershipN	AirportCode	DestAirport	PlaneID	Price	SportsEqPrice
LastName		Country	PasLastName		AirportPostcode	Country	ArrTime	Weight	
Password		StreetLine1	PasReasonTravel		ConnectingAirport	Region	DepTime	Quantity	
DialingCode			SpecialAssist				Fare		
TelephoneNumber			AgeTravel				ArrDate		
MembershipNumber			HasInsurance				DepDate		
EmailAddress			BookerID				RouteID		
PrefDepartLocations			SeatID						
ContactPref									
PaymentCard									
Airport									
BookingRef									
AddressID									

Plane	Seat	Pricing	Fees_Charges	Currency	SpecialAssist	Hands_Free	Membership	Food_Drink
PlaneID	SeatID	PricingID	Fees_ChargesID	CurrencyID	SpecialAssistID	Hands_FreeID	MembershipID	FoodDrinkID
Rows	SeatName	AdminFees	GovFee	CurType	Type	Price	EasyJetPlus	VoucherCost
Seats	SectionType	Taxes	Taxes	CurRate			FlightClub	PassType
Capacity	SectionCost	OnSale		BasicCur				Quantity
ModelNo	Restriction	InDemand						
	SpeedyBoardin	FlexiFare						
		StandardFare						
		ChildFare						

Fig. 1 – Excel spreadsheet showing early entity discovery

As a group we discussed the intricacies of the website, in particular the scope of the website itself and therefore outlining the scope of our potential databases. We decided to focus on the user/booker, passenger, and flight detail aspects of the booking experience

whilst deciding other facets of the website such as travel insurance, hotel booking and car hire were out of scope and not relevant, primarily due to their third-party nature.

We then went on to ensure each entity had a unique identifier by creating a primary key in order to enforce entity integrity and provide a foreign key reference for other tables further down the line of Entity Relationship Diagram (ERD) and database creation. We considered using a combination of attributes to create a candidate key but agreed as a group that an auto-generated surrogate key was safer in terms of preventing duplication of attributes or data, especially when making assumptions about easyJet business rules. With this data, we then developed an ERD by considering relationships between these entities and attempting to conform to the rules of normalisation.

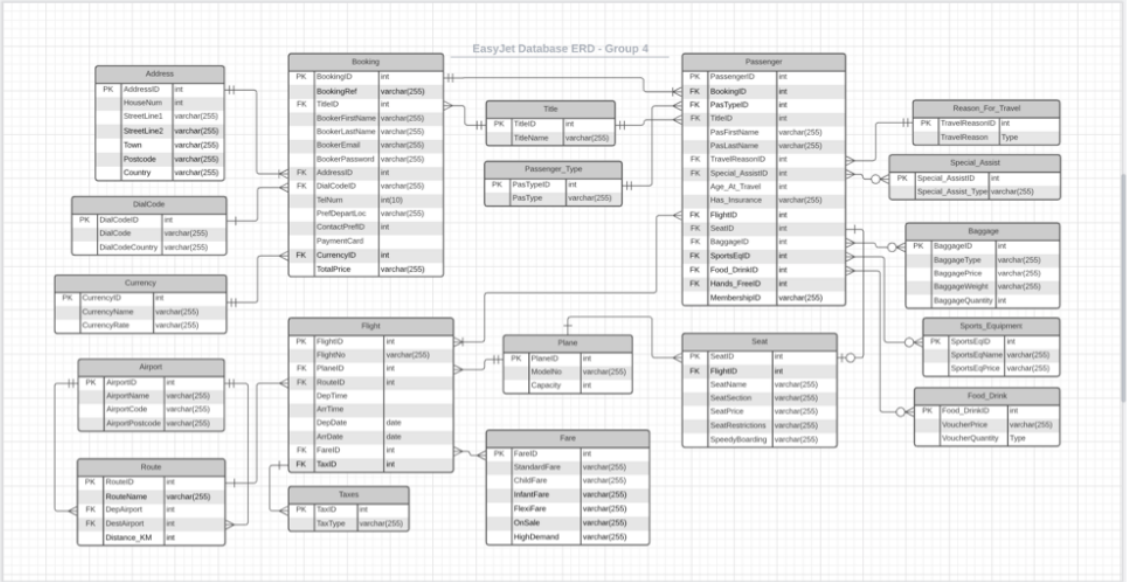


Fig. 2 – Group ERD containing developed entities, attributes, relationships and keys.

We began with considering what entities would naturally link with other entities, for example, Booking with Passenger, Address, Title and Passenger with Baggage, Food_Drink, Reason_For_Travel etc. From here we quickly developed the ERD shown in Fig. 2. It's worth noting that some entities and attributes were cut down as the ERD developed, Hands_Free and Membership tables were cut as they were more applicable to easyJet Plus on its own, and some attributes such as 'ConnectingAirport' and 'PrefDepartLocations' were similarly removed due to their inconsequential nature during the booking process.

Once we'd finished cutting our tables down and considered relationships, we began adding potential foreign keys into our tables by labelling them with a foreign key label, which would enable us to link entities together and insisting on referential integrity. This went hand in hand with normalisation in our tables, which then began to expand and simplify our entities and attributes. We began to look the rules of Normalisation described by Edgar Codd [2], beginning with First Normal Form (1NF) enforcing the criteria of eliminating repeated data, creating separate tables for related data, and creating a primary key for each table. During the entity discovery stage, we'd already normalised tables such as the Title table as we knew it contained attributes that would be used in both the Booking and Passenger tables multiple times, and therefore by creating the Title table we could enter the specific data used on the easyJet site and put the Title foreign keys into both

Booking and Passenger tables for use there. We also used 1NF to create separate tables such as PassengerType, DialCode and Currency to again allow specific data to be input to increase data reliability, and prevent repetition in both the Passenger and Booker tables. Originally, we created two Airport tables to allow for a destination and arrival airport, which we were then able to condense into a singular airport table that could be linked by its FK to both DeptAirport and DestAirport in the Route table which could be joined in an SQL query using an alias for each join. This proved useful as the database developed, as it allowed us to create specific data allowing only airports used by easyJet in particular countries to be used as arrival and destination airports, as well as minimising space used by in the database by having multiple airport tables.

We also considered 2NF and 3NF in our ERD, particularly paying attention to the 'no dependency or reliance of any columns on the primary key' rule, hence why our tables each have their own surrogate primary key with no dependency between columns. Similarly with the 3NF directive, we avoided creating entities where a non-key field becomes a fact about another non-key field. For example, we split Fare from the Flight table and created its own table so a price wouldn't depend on each flight. In this case, if a flight fare became more expensive as it got closer to the departure date and the price needed to be changed, it would only have to be changed in the Fare table and be linked to the particular Flight rather than change the fare for each different type of fare for the flight in the Flight table.

Our final challenge as a group was to begin to look at the types of relationships between entities, and decide on what cardinality constraints to apply to our model. This task went hand in hand with primary and foreign key design and took a significant amount of discussion and theoretical consideration to factor in particular constraints to apply logic to the database. We began with the 'many to one and only one' constraint and began drawing constraint arrows between entities such as Passenger and Title, Passenger and Reason_For_Travel, Booking and Address, Booking and Currency etc. This was partially to ensure some common-sense rules were set up, that an individual Booking could only be linked to one and only one address and dial code in order to prevent repetition or irrelevant data from being entered. It also allowed us to enforce some business logic in our diagram. By making assumptions about business logic from the easyJet website, it was obvious that there were some restrictions on data being entered. For example, we were able to ensure that a passenger could only select one reason for travel, Business or Leisure, for their booking which stands as the business logic on the website by adding a 'many to one and only one' relationship. Similarly, we applied this logic with 'many to zero or many' relationships. For example, allowing a passenger to have multiple bags, sports equipment, special assistance or food and drink options by creating many to many relationships between these entities.

Some issues did occur when deciding how to enforce a many to many table, as Fig. 2 shows that whilst a many to many relationship arrow between these tables exist, it still only allowed a passenger to have one bag, one special assistance option etc. Similarly, there were many discussions around relationships between Flight, Plane, Seat and Passenger tables in order to ensure seats were able to be mapped to rows and seat names, but also allow the option to link a seat to a passenger, plane or flight. With discussions, alternative options and design decisions considered, we left this ERD as our final group contribution and pursued these routes individually.

II. *Individual design – ERD development*

To make my ERD as clear and distinct as possible, I began with adapting the group ERD and adding some superficial design alterations. To save page space, I removed the Key column and opted instead for the more visual colour-coded key, with blue underline highlighting primary keys and red bold representing foreign keys (see Fig. 3. At this point I made the decision to give foreign keys the same ID name as primary keys and place them as the first rows in every single table in the ERD to make the implementation of the database in phpMyAdmin as seamless as possible. Some minor changes were also made to attribute and entity names, for example 'Booking' becoming 'booker', 'Flight' becoming 'flightDetails' to differentiate between it, 'plane' and 'route, 'taxFee' instead of 'Taxes' and 'Food_Drink' becoming 'mealVouch'. Again, all minor and superficial changes to prevent confusion and make the database properties easier to read. As a group, we'd also had different preferences for case types used, so our ERD was using a mixture of Pascal case and Snake case. At this point I opted for my own preference of using Camel case for table names and Pascal case for table attributes in order to make them distinct from one another during database creation and to avoid confusion during the SQL query stage.

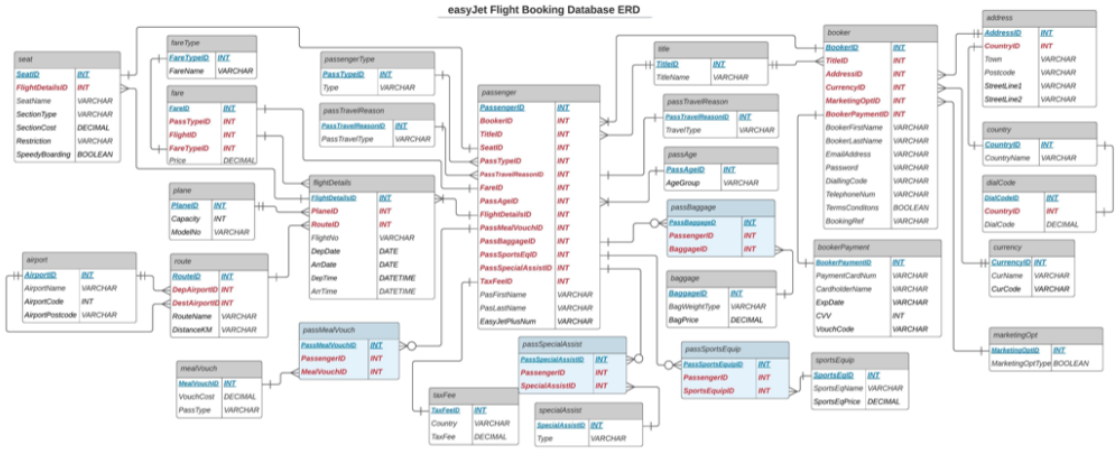


Fig. 3 – Personal ERD developed with many to many tables ready for database implementation

Once these superficial changes were made, I began to do some further investigations into the easyJet website. From there came the additions of some additional tables such as marketingOpt which would enable a booker to select the option for receiving easyJet marketing or easyJet partner marketing, both or neither. I decided on a BOOLEAN datatype that would become a TINYINT with a limit of 1 indicating 'Yes' and 0 indicating 'No' for each checkbox on the website (Fig. 4). The only option unavailable to the booker was to check 'No' for easyJet offers and 'Yes' for partner offers, however I made the decision that this rule could not be implemented at the database level and would instead be executed with a programming language such as PHP during website development.

The 'many to many' tables were also developed into two 'one to many' tables using a third bridge table (blue tables, Fig. 3), which solved the problem of allowing passengers to select multiple bags, sports equipment, special assistance and meal vouchers. For example, originally the passenger table had a 'many to many' relationship with the baggage table,

which then developed into the passenger table and baggage table linking to the passBaggage bridge table that only holds the PassengerID and BaggageID foreign keys. By creating a bridge table that holds these foreign keys, a passenger can be linked by their ID to many bags rather than only one, and can be presented in the passBaggage table separately. The passBaggage table on its own isn't very informative, but when used in an SQL join to passenger and baggage it becomes much easier to tell which passengers booked which bags.

Through reconsideration of the process of Normalisation, the fareType table was created to link to fare in order to eliminate repetition in the fare table of selecting adult, child or infant each time a fare had to be added. This was then linked in to the fare table, which connected to the flight table to map each fare to each flight, mirroring the dynamic pricing model of the easyJet website. Implementing a fareType table also disabled the ability to enter a free text option not available on the easyJet website (e.g. 'Pensioner', 'Teenager') and restricted the fares being entered to strictly 'Adult', 'Child' or 'Infant', thus maintaining data integrity in the fare table. Similarly, some changes were made to the country and taxFee tables. It became clear from the easyJet website that they maintain a specific list of countries in their database that can be selected when entering address or dial code (Fig. 5, Fig. 6), so it made sense to normalise the 'Country' attribute into a country table, with CountryID added into the address and dialCode tables as a foreign key. Again, this provided a defined list of countries for selected and therefore maintaining data consistency during the booking process.

Finally, the 'one to many' relationship between flight and taxes was removed and instead a 'one to one' relationship was created between the passenger and taxFee tables as it became clear from the booking process that tax was considered on a 'per passenger per flight' rule, rather than an overall tax fee per flight. With this modification made, there was a tangible plan to go forward with the database implementation.

III. Individual design – Database development

During the database building stage, entities, attributes and relationships were constructed according to the plan ERD in Fig. 3, however throughout the process there were still many deviations and changes that occurred before it developed into the database in Fig. 7:

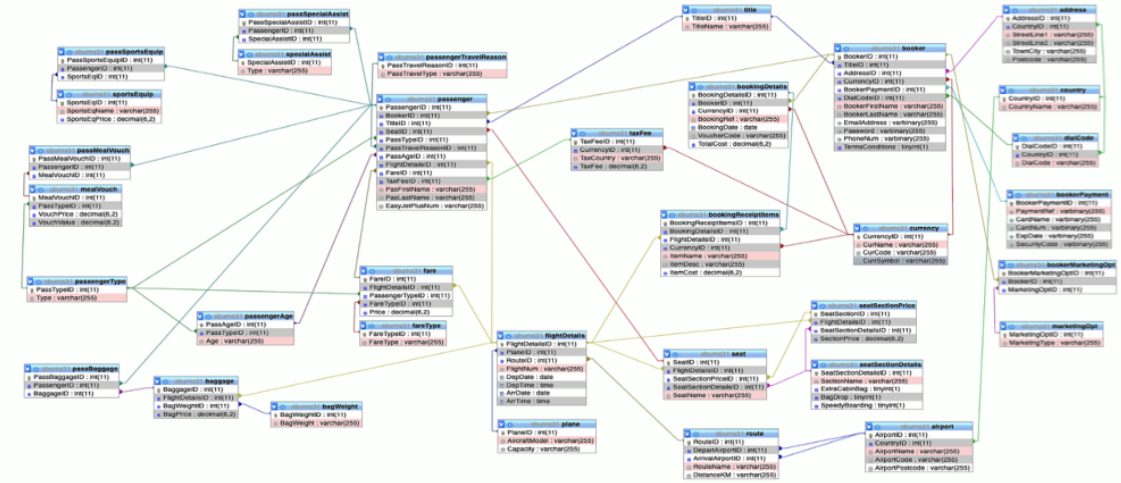


Fig. 7 – ERD from database created in phpMyAdmin

The primary concern was around seat pricing, and consideration of how to map each seat to a passenger, have a map of seats per flight, and allow each section of seats to be priced and named differently per flight. To combat these issues, I added two extra tables seatSectionDetails and seatSectionPrice to allow the section details to be normalised and prevent repetitive free text data in the seat table, and have its foreign key in the seatSectionPrice table to enable a price to be given per section. These then linked through foreign constraints to the seat table, which in turn was linked to both the flightDetails and passenger tables allowing these seats and their prices to be mapped according to flight and passenger, as seen in Fig. 8, 9 and 10. This also allowed the seatSectionDetails table to be populated with information about benefits of each seat assigned to a passenger, such as an extra cabin bag allowance and eligibility for bag drop and Speedy Boarding benefits (Fig. 11).

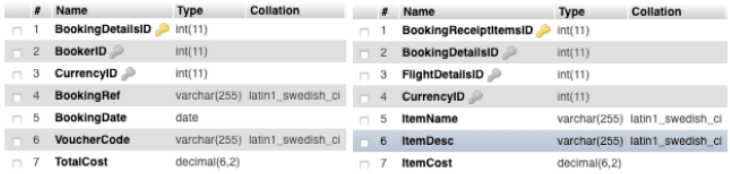
SeatSectionDetailsID	SectionName	ExtraCabinBag	BagDrop	SpeedyBoarding
1	Extra Legroom - Front Row	1	1	1
2	Up Front	1	1	1
3	Standard - Middle	0	0	0
4	Extra Legroom - Middle	1	1	1
5	Standard - Back	0	0	0

Fig. 10 – seatSectionDetails table showing benefits each passenger has according to their seat selection.

With fare prices and seat prices build to accommodate easyJet's real world dynamic pricing model, it became clear from the website that the prices for meal vouchers and sports equipment were the same for each flight regardless of route, however baggage also fell under the dynamic pricing model. In order to change the database to fit this scenario, I linked the baggage table to flightDetails, and also added a bagWeight table through the process of normalisation to prevent repetitive data appearing in the baggage table and allow passengers to select only baggage weight options defined by the easyJet website as seen in Fig. 11.

It also proved useful to review the datatypes of attributes as they were entered into the database. Originally, the majority of the datatypes I chose were int, varchar and boolean/tinyint. Most of these remained during the database implementation, however I did make some further changes and decisions based on assumptions about the easyJet model. For example, instead of adding all number-based fields as int type, it became clear that attributes such as EasyJetPlusNum, PhoneNum and VoucherNum could simply be saved as varchar types due to the assumption that there wouldn't be a cause to use functions such as SUM or COUNT, but were best seen in a plain text varchar form. Boolean/tinyint(1) types proved suitable for tick box instances, such as TermsConditions and the seatSectionDetails attributes to determine simple yes or no options in order to give definitive results in searches and queries. Other number-based fields based on price were given DECIMAL types to enable the SUM function to be used when adding booking prices together.

When considering datatypes for pricing, it became apparent that a line items type table was needed in order to allow details of each booking to be generated as invoice style data. This also meant that whilst the dynamic pricing model was in place, flight prices at the time of booking could be recorded before they changed, and so each booking wouldn't update to the 'real time' prices as they changed. I created the bookingDetails table and linked it to the booker table and currency table in order to record key details about the booking including a booking reference and date (Fig.12).



#	Name	Type	Collation
1	BookingDetailsID	int(11)	
2	BookerID	int(11)	
3	CurrencyID	int(11)	
4	BookingRef	varchar(255)	latin1_swedish_ci
5	BookingDate	date	
6	VoucherCode	varchar(255)	latin1_swedish_ci
7	TotalCost	decimal(6,2)	

#	Name	Type	Collation
1	BookingReceiptItemsID	int(11)	
2	BookingDetailsID	int(11)	
3	FlightDetailsID	int(11)	
4	CurrencyID	int(11)	
5	ItemName	varchar(255)	latin1_swedish_ci
6	ItemDesc	varchar(255)	latin1_swedish_ci
7	ItemCost	decimal(6,2)	

Fig. 12 – bookingDetails table linked to Booker and Currency holding key details of booking made
Fig. 13 – bookingReceiptItems table linked to BookingDetails, FlightDetails and Currency

From there I created a bookingReceiptItems (Fig. 13) that linked to BookingDetails and Currency, and contained attributes that would allow a receipt to generate with items and prices paid for during the time of booking. After adding some sample data, I also added the FlightDetails constraint to the table in order to show for which flight each item was purchased, to avoid confusion when return flights were booked by a single booker. When the all data was entered for each booking, it was then useful to use a SUM query on the ItemCost attribute and insert the final figure for each booker in the bookingDetails table manually (Fig 14, 15). Because the ItemCost table had a datatype of decimal, the voucher code expressed in the bookingDetails table had functionality in the bookingReceiptItems table by allowing minus numbers to be entered, therefore taking the voucher amount away during the SUM query.

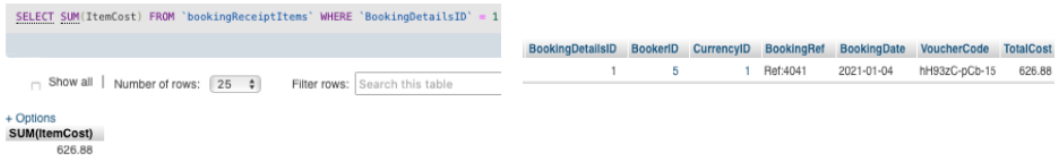


Fig. 14 – SUM function in query from bookingReceiptItems table for BookingDetails 1 returning 626.88
Fig. 15 – bookingDetails table showing BookingDetails 1 for Booker 5 with total cost input.

When the database was ready after these modifications and revisions, it was then populated with a small amount of sample data. Some tables were much more intricate than others, however it was helpful to use bulk insert statements in order to generate and update data much more quickly than to do so individually. The bookingReceiptItems table contained the most data, as well as some repetitive input and therefore it made the most sense to populate it using a bulk insert statement as seen in Fig. 16.

IV. Protection and Encryption of data

The bookerPayment table originally didn't feature in the group database design, as a group we decided that payment would be handled by a third party and that this would therefore be out of scope for our design. As I implemented my own database, I decided to include a basic payment table in order to show how the data could be stored in a database with some safety precautions set in place through use of encryption in mySQL.

The bookerPayment entity was created, and its foreign key constraint placed in the booker table, to allow mapping between a booker and their payment details. Attributes such as payment reference, card name and number, expiry date and security code were then added, and it became clear when inputting sample data that due to the sensitive nature of the data being entered it needed to be protected and avoid being shown in plain

text. Through research it became evident that mySQL is very limited in terms of storing details securely, and for cyber security purposes it would be much safer to allow a third party to hold these details with separate security procedures. However, for the purpose of this project it was beneficial to show this part of the booking process being stored, and to show that basic encryption could be applied to the data using mySQL.

I decided that all fields in the bookerPayment table were sensitive, and therefore all attributes (with the exception of the primary key) needed to be encrypted so all fields needed to be changed to VARBINARY datatype. I used the Advanced Encryption Standard (AES) algorithm, which encrypts a string of sensitive information using a secret key string and returns a binary string containing the encrypted output. To access the sensitive data, I used the AES_DECRYPT() to decrypt the encrypted string using the secret key, which then returns the original string in plaintext. I applied the AES algorithm in bulk for the bookerPayment table, and then decided it was beneficial to encrypt sensitive data in the booker table as well, which can be seen in Fig.17, 18 and 19.

There are many issues using the AES_ENCRYPT() function, especially through use of secret keys that may be shared with other developers during the creation process, and secret keys that could be guessed, intercepted or reused in a database. According to the MySQL Documentation Manual, if an SSL isn't used when connecting to MySQL, the encrypted information is still delivered to the server as plaintext, making it even more prone to interception [3]. Therefore, it's best to implement both encryption and decryption in the actual application development using a programming language such as PHP on the server side and leave the database to solely handle storing and retrieving data. From researching the PCI Security Standards Council site [4], it's also clear that easyJet would not be able to comply with these standards of storing cardholder data using the AES algorithm, and specifically states that companies that don't store cardholder data themselves automatically provide stronger protection from data thieves. For the purposes of this project it has been useful to show how the data can be protected at database level through the AES algorithm, however for a real-world booking database for a global company such as easyJet, it would be much more secure and compliant to use a third-party payment processor and encrypt data using a programming language outside the database.

V. Improvements

Throughout the development of this database, I've tried to keep Chen's theories around creating and manipulating a database based on real-world scenarios at the forefront of the database design decisions during group work and individual development. However, there is always room for improvement, and the database implemented in this project is no exception. With more time to deliberate over the design decisions, it would be beneficial to the model to create a boardingPass table for example (see Fig.20), which could generate boarding pass information including passenger name, seat name, route, flight number, flight date, seat section options, plane boarding side etc. after the booking had been completed. This would have been valuable in terms of bringing all this data together in a single table per passenger automatically, instead of having to join this data together in an SQL query manually.

Another flaw in this database is the inability to see what seats have already been booked and applied to a passenger, so to avoid double booking a seat. It's possible that in the execution of the website, a programming language could be used to disallow to option

to book a seat if this seat has a passenger linked to it already, however for the purpose of this database it would have been valuable to see this at database level, especially when entering sample data. This could have been provided by adding another attribute to the seat table such as SeatBooked with a boolean datatype, which could be updated as bookings were placed by passengers, avoiding double booking.

VI. Final Outcome

Overall, this database functions successfully for storing data processed during the booking stage on the easyJet website. It conforms to the needs of a booker by providing a sensical approach to storing data through the process of normalisation, avoiding repetitive data and disallowing free text data where a table of options can be provided instead. I'm confident that this database has been developed in accordance with Chen's ideologies around moulding databases around real life problems and scenarios through the use of bridge tables and foreign key restraints. As described in chapter VI, there is room for significant improvements in the database, as well as enhanced data security that would have to take place during the programming of the website to conform to payment security standards. However, for the purpose of this project, the database conforms to the needs of a booker and of the data storage needs of the easyJet website in order to make a booking with multiple passengers, with multiple booking items and preferences for multiple flights and routes whilst successfully implementing a dynamic pricing model for fares, seats and baggage.