**Assessment info**

* No database file needed
* Marking based entirely on report and video

**Written report info**

* Pdf form
* Submitted through canvas
* No cover page or table of contents
* Sections
  + Overview of db design – designer view/ draw io
    - Start and final – allows to explain key design decisions
    - Database building process and test data entry
    - More diagrams can go in appendix
    - Talk through noramlisations and evolution of design
  + List of design assumptions that were made
    - What is in and out of scope?
    - Thisis about package holidays
    - Aircraft baggage etc/ meals out of scope
    - Package holiday: hotel room for x days and flights
    - Assume room and flight are tied, if you cang et one then you can get the other
    - Direct flights only
    - Enumerate the most key assumptions
  + Explanation of some important primary and fk constraints
    - Surrogates?
    - Composite keys?
    - Foreign keys to prouce 1:1, 1:N, N:M etc
    - Types and why they were chosen
    - Not interested in a dictionary of everything
  + Some Important design desicisons **(paired with below – same section)**
    - **Micro ones aren’t really necessary**
    - Key decisions around prices, availability, pricing etc
  + Some important normalisation decisions **(paired with above)**
    - Good pragmatic decisions
    - Not so interested in specific normal forms, but the pragmatic WHY of decisions
  + Improvements
    - Design improvements (own section maybe)
    - We are only building selected aspects of the system, so there will be reasons why some tables are empty, why some things are not implemented
  + SQL Queries in appendix
    - Demo of important aspects
    - Record all ones from video in report

**Introduction and Project Context**

This project is an attempt to reverse engineer the database layer of certain sections of the professional commercial booking system of the [Jet2Holidays](https://www.jet2holidays.com/) website, to create a relational database that enables its functionality. The Jet2Holiday website is vast and was developed iteratively over many years and contains much functionality that would not be possible to recreate within the timescale allowed for this project. Thus, the scope of this project will be limited by certain design assumptions listed below.

A high-level overview of the functionality of interest on the website is as follows:

1. Search for a holiday package using departure airport, holiday destination, holiday duration, and number of passengers and their ages.
2. Make a booking for a holiday based on selected departure airport, hotel, room type, board basis, holiday duration, number of passengers. A single passenger is nominated as the booking contact and that passenger requires extra information to be stored, namely address, email and telephone contact information.
3. A room availability model.
4. Variable flight pricing dependent on date and time.
5. Variable accommodation pricing dependent date.
6. Secure storage of payment information.
7. Create SQL queries and statements to demonstrate the various CRUD operations underpinning the above use cases.

The database is to serve the data storage needs of all these use cases, and additionally to store all data visible on the in-scope sections of the website, which are discussed below.

We approached this project by operated under the following design assumptions:

1. If a room is available to choose, a corresponding outbound and return flight is assumed to be available.
2. Financing and recurring monthly payment of the cost of a booking was considered out of scope, but multiple part payments or a single complete payment were considered in scope.
3. Any data required to implement the selection and pricing of fine grained flight booking were out of scope; namely specific seating on a flight, extra carry-on baggage, hold luggage, extra leg room seating, suitability of seating for infant passengers, sports equipment transport, and flight meals.
4. Flights are assumed to be direct to destination. Multiple leg flights are considered out of scope.
5. Transfers from airports to hotels were considered out of scope.
6. Car Hire was considered out of scope.
7. Any form of insurance was out of scope.
8. Scope is limited to the Jet2Holidays section of the website, excluding others – such as Agent Finder, Jet2Villas and Jet2CityBreaks, although the latter two have considerable overlap in functionality with Jet2Holidays.
9. We will not be able to implement full Payment Card Industry Security Standard Council (PCI SCC) policies and procedures regarding storage of payment card information. These are very detailed and prescriptive, instead we settled for implementing basic 2-way encryption of card details using AES encryption using a known Secret, adhering to the most basic principles of the Payment Card Industry Data Security Standard (PCI DSS), a subset of PCI SSC policies relating to payment card information storage, amongst a great many other things.
10. A country may use many currencies and languages, but I shall store only the countries primary currency and language data.
11. We will consider internal Jet2Holiday hotel reviews in scope, but trip advisor reviews are considered out of scope. As a note the “internal” Jet2Holiday reviews are served via trip-advisor, likely via Trip Advisor’s Restful [Content API](https://tripadvisor-content-api.readme.io/reference/overview), but we shall implement it via database storage as a naïve implementation.

Diagram

Description automatically generatedDiagram

Description automatically generated

Figure 1 Initial Entity Relationship Diagram

Figure 2 Final Entity Relationship Diagram

**Database Design Process**

Our process of database design can be divided into distinct stages. Initial entity discovery was undertaken individually. We separately analysed the Jet2Holidays website to try to deduce the entities and attributes underpinning data visible on in-scope sections of the website.

We then met as a group to combine our entities into one list, and as a group pinpointed which of these bits of information were entities, and which were attributes. We undertook normalisation as a pragmatic process intended to remove obvious repetition of data, expand composite and multi-valued attributes into their constituent attributes, and decide which were more appropriate as derived/ computed attributes. This normalisation process was ultimately intended to bring all the entities and their attributes up to the standard of the 3rd normal form as described by Edgar F Codd’s 1971 paper “*Further Normaliszation of the Data Base Relational Model*”, which developed upon his earlier 1970 paper “*A Relational Model of Data for Large Shared Data Banks*”, that is:

1. To free the collection of relations from undesirable insertion, update, and deletion dependencies.
2. To reduce the need for restructuring the collection of relations as new types of data are introduced, and thus increase the life span of application programs.
3. To make the relational model more informative to users.
4. To make the collection of relations neutral to the query statistics, where these statistics are liable to change as time goes by.

The fourth point above went somewhat beyond the scope of this project.

We did not move through the normal forms in steps, as to do so often involves contorting the data into unintuitive forms, but instead took the more pragmatic approach of approaching normalisation as a continuous process, by moving towards the more informative and natural schema that the 3rd normal form informs. The normal forms beyond the 3rd were not directly considered.

We divided up into sub-groups to consider the various major sections of the functionality we were mirroring with our database, with myself being part of the customer subgroup. I argued against dividing ourselves into subgroups as it would be difficult to simultaneously work on different sections, considering the relation model requires the sub sections to be intimately related to each other through foreign key/ primary key relationships. The group however decided to go this route and so I honoured the decision. The different subgroups worked on customer, flight, hotel and booking sections of our entity discovery list. We produced initial, and very naïve, entity relation diagrams of the isolated parts of the systems (viewable in appendix X). After this stage I worked mostly independently.

The differences between the subgroup ERD diagrams, the initial combined diagram, and the finished diagram are myriad, and a period of extensive normalisation and identification of key relationships had to be completed before our entity relationship model was able to mirror the Jet2Holiday’s website data storage requirements.

I implemented my entity relationship model as a MariaDB (v10.4.24) relational database – an open-source fork of MySQL created by some of its original developers. This process was straight forward and allowed me the opportunity to consider each relationship as it was implemented. Much further continuous normalisation was undertaken at this stage, as limitations in our model became more apparent. Changes enacted at this stage are reflected in the final entity relationship diagram.

After which, test data was inserted into every table to allow us to exercise the various relationships which identified many more shortcomings in our assumptions on how our relationships would interact to form the overall relational structure of my database.

The final stage of the design process was the construction of SQL queries and statements which would demonstrate the ability of my entity-relationship model to serve the in-scope sections Jet2Holiday website. These are viewable in the Appendix X and in my video report.

**Design Decision Overview**

Figure X describes our initial entity-relationship model; however, it is naïve in many aspects within the specified scope of the project, including no consideration of data types – these issues were mostly addressed in the final diagram (figure x).

**Naming Conventions**

Initial naming conventions of entities and their attributes are inconsistent. There is a lively and ongoing debate on the appropriateness of pluralisation in naming entities, but I decided to keep the naming conventions simple and in the end I opted for lower-case and snake-case (words separated with an underscore), with no pluralisation. Notable examples are:

1. The initial combined design has an entity named Hotel\_images, but the final design settled on hotel\_image, removing the capitalisation and pluralisation in the name
2. Board Types in the initial design became board\_type in the final design, removing capitalisation and internal whitespace from the entity name.
3. Hotels entity became hotel, removing capitalisation and pluralisation.

**Currency and Language**

I made the decision that, for my design, all prices in the database where to be stored in Great British Pounds sterling (GBP), as Jet2Holidays is specifically a British package holiday provider. I investigated this by visiting the website via a virtual private network that spoofed my connection’s origin as another country. The prices were always shown as GBP. I also searched for any other top-level domains Jet2Holiday instances may reside in but could find none. The prices are therefore based in British tender.

If the website wished to expand to other markets, it would seem likely to me that either the new market would be served by a separate instance of the Jet2Holday system under a different top-level domain, or the system would use an external service’s API to convert the GBP prices recorded in the database into a localised currency based on exchange rates.

I acknowledge that this design decision may cause unfavourable restructuring decisions in the future if the website would like to expand its service to operate natively under other currencies, as attributes would either need renamed (e.g. total\_cost\_gbp attribute of the booking entity may need renamed total\_cost\_eur), or foreign key attributes linking the currency entity may need to be added to all price containing entities (route\_price, room\_type\_price and hotel\_board\_type). This goes against the 2nd point of Codd’s principles of further normalisation discussed above, as it may require some restructuring of entities and attributes at a later date.

1. Different currencies are still stored within the currency entity, as this data is visible on the website at the level of the region (e.g., the *Costa Blanca* region page). I chose to store my currencies at the level of country, specifically in the form of a primary\_currency\_id foreign key in the country entity, which then allows currency data to be accessible via a chain of 1:N relationships from region entity up to the currency entity.

Figure 3 1:N relationship chain from currency to region

**Pricing Model**

The initial design does not consider pricing at all (flights, hotel rooms and board type selected)! Subsequently it does not consider time/date dependent pricing, nor how to tabulate payments made nor balanced owed on a booking. The final design addresses this as discussed in the bullet points below. These pricing models will be demonstrated in my video report.

1. **Hotel Room Pricing:**

Graphical user interface, text, application, email

Description automatically generated A room\_type\_price entity was created that allows a hotel room – which we have priced at the hotel’s room\_type level as they appear to be on the website – to have individual pricing across date ranges via the room\_type\_price\_valid\_from\_date and room\_type\_price\_valid\_to\_date attributes.

A price of a booked room must also consider the board type selected. All-inclusive, after all, will be priced differently from self-catering. Board bases are defined in the board\_type entity, and the hotel specific price and description of a board type is defined by a 1:N relationship relating each board type with the hotel\_board\_type entity.

Figure 4 Room availability and pricing model

These two levels of pricing (hotel\_board\_type and room\_type) for a room are then related together by the room\_booking table via a N:M relationship using room\_booking as a linking/ join table This allows us to enumerate the rooms chosen for a booking, and to calculate the total price across dates and times, room types and hotel board types selected, and associate this information to a specific booking (via a 1:N relationship between the booking and room\_booking entities).

A hotel room’s price therefore is based upon several factors: the hotel to which it belongs, the room type of the room, the date upon which the room is required, and the board basis selected. I made the design decision to limit the room type pricing to dates and not times, as a hotel room is booked by the day and therefore time is not a relevant consideration.

1. **Flight Pricing:**

**Graphical user interface, text, application

Description automatically generated**A route\_price entity was created that allows a flight to be priced depending on both the route upon which it flies, and the date and time upon which the flight is to depart. The reason I implemented both time and date dependent pricing is due to how certain flights are more desirable and so therefore command a higher price. For example, a flight at 9am on a Monday will be priced differently (due to different demand) than a flight at 1am on a Sunday morning. Including the time in the pricing model allows us to account for this. Therefore a flights price is based upon the route it takes, and the datetime dependent price of that route.

Figure 5 Flight Pricing Model

1. **Payments and Balance Owed**

The total calculated price of the booking is stored primary as a summation of the booking\_line\_item entities booking\_line\_item\_price\_gbp attribute, that is inserted during a booking’s finalisation. A payment table was added that allowed us to record each the total of each separate payment made and its associated card details, and therefore (summation of all line items price for a booking) – (total payments made for a booking) can be used to calculate the outstanding balance of a booking.

1. **Limitations of Pricing Model:**

One issue with our room/ flight pricing model is that I understand that holiday pricing is dynamic in that the price of a room on any given day, not only depends on the date the booking is for, but also the date that the booking is made – and perhaps more ethically grey criteria such as how many times a customer has checked a booking search result.

Booking a room for tomorrow might incur additional “scarcity” costs, over booking a room further in the future, assuming that all other costs are equal. We have isolated our pricing models into their own tables and so a way of having this dynamic pricing would be to update the respective valid\_from and valid\_to attributes of the route\_price and room\_type\_price entities as a scheduled job during regular maintenance down time. These pricing tables are the source of truth for pricing, and so updating (via UPDATE or DELETE/INSERT statements) these tables will update the price for all queries using the updated records.

The booking\_line\_item records related to a booking and subsequently the total\_cost\_gbp attribute in the booking entity (via a N:1 relationship based on the booking\_id primary key) record the price of a booking at the time it was made, so updating the pricing tables would not affect the cost of a previously finalised booking.

I acknowledge that the total\_cost\_gbp attribute of the booking entity stores redundant information that is already stored as a summation of all line item costs for a booking. A simple solution would be to remove this attribute entirely and rely on the summation.

**Room Availability Model**

My room availability model (figure x) operates by assigning a base number of each room type that a hotel has (represented by the room\_type\_base\_quantity attribute of the room\_type entity). This is the maximum number of bookings that may utilise this room type in this hotel on a given day. The number of available rooms is calculated by in the following manner (I will demonstrate this in my Video Report – query viewable in *Appendix X*):

|  |  |  |
| --- | --- | --- |
| (room\_type\_base\_quantity) | – | (Number of room\_bookings utilising that room\_type on a given day) |

A given room\_type, is related to a single hotel via a 1:N relationship.

Table

Description automatically generatedThere is a glaring shortcoming with room my availability model (figure x), in that it is possible that a room type is available on all given days of a booking, but that a particular room itself of this room type is not available for the full duration of the booking. As in figure x, if a hotel has 3 rooms of a specific type, 2 may be booked on any given day inside a range of dates, which would give the misleading appearance that there is one free room able to be booked continuously for this entire date range. In reality, there may be a number of shorter bookings of this room type. This means if a 3rd booking utilised the last free room over the date range, they may have to change rooms during their booking. This would be a negative experience for customers, and unacceptable in a real system.

Figure 6 Room availability model shortcoming

Diagram

Description automatically generatedWith a little more forethought this could have been solved by enumerating all of the given rooms of a room\_type in a separate room entity, removing the room\_type\_base\_quantity attribute from room\_type, and instead linking the new room entity to the room\_type by placing a foreign key in the room entity relating to room\_type’s primary key in a 1:N relationship, which would have allowed more fine grain querying of the rooms available for booking (figure x). After which the room\_booking entity would be updated to remove its foreign key relationship to room\_type, and instead link directly to room. The base quantity of a room type available would then be calculated by using a COUNT() aggregate function to tally the number of rooms of a type.

Figure 7 Proposed solution to room availability shortcoming

My original design ERD (figure x) started with room as an entity and was subsequently dropped, with the room\_type\_base\_quantity added to room\_type in its stead. In the end, this design decision had unforeseen consequences, and if it had been noticed earlier in the design process, would have been remedied.

**Encryption and Secure Data Storage**

It is worth noting that I recognise that in a real-world system this payment card information would be stored by a specialised business that implements the full set of Payment Card Industry Security Standards Council’s Data Security Standard (PCI SSC/ DSS) policies and procedures and would abstract the legal, financial, and reputational liability of storing payment card information from Jet2Holidays to said payment processing business. The initial design does not consider how to store payment card information at all, nor the PCI SSC/ PCI DSS requirements of doing so.

1. My final design settled on encrypting the payment card’s long number (*primary account number* or *PAN* as per PCI DSS documentation) and expiry date. These are classified as *Card Holder Data* by PCI SSC policies and may be stored if they are encrypted in line with their requirements. Our encryption method falls short of the stringent guidelines of PCI SSC/ DSS, in that they are not stored using “*One-way hash functions based on strong cryptography”,* whereas we implemented storage using a 2-way AES encryption method using a known secret decryption key. Although our method of secure storage falls short of full PCI DSS compliance, it is an acknowledgement of their existence and importance.
2. I chose not to store any *Sensitive Authentication Data* (SAD) as defined by PCI DSS relevant to our storage needs, namely the cards security code/ CVV number. As per PCI SSC policies and procedures SAD may be used for initial authentication of the payment card, but it must only be stored by certain legitimate authorisation entities such as the card issuer, which Jet2Holidays is not.
3. It does not consider the ability for a user to create an account and save provisional booking information to a short list.

**Geographical Information**

Graphical user interface, text, application, chat or text message

Description automatically generatedOur initial ERD does not adequately describe the set of relationships between the geographic entities destination, region, resort, and hotel visible on the website. The region entity is missing entirely! These were later added in a “chain” of 1:N relationships that relates all this data together. From the *figure 1* example destination *Spain* contains many regions, region *Costa Blanca* contains many resorts, resort *Benidorm* contains many hotels, our final design mirrors this structure (figure x).

Due to the nature of the way Jet2Holdays store their destinations, it is likely that the country Spain (a geographical location) and the destination Spain (a group of related regions resorts and hotels) would contain redundant data in the form of the string “Spain” in their respective name attributes.

This potential repitition of data was solved via a normalisation decision, which resulted resulted in an entity named location\_name being created. The destination and country spain could share a name without this name being entered into two separate tables. This also allows destinations such as Balearic Islands and Canary Islands (both parts of the country Spain in reality) to use the location\_name of Spain via location\_name\_id’s 1:0 relationship with country, which was enacted via a combination of foreign key constraints an a uniquenss constraint on location\_name\_id in country.

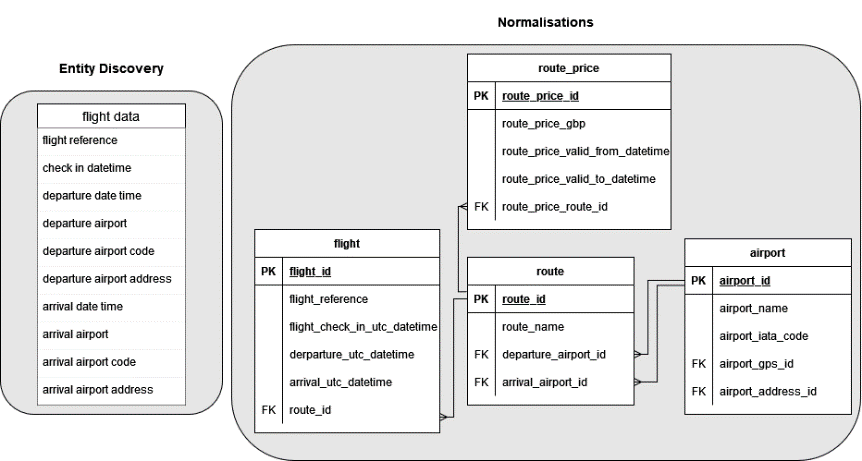
**Flight tables**

Figure 9 Flight Data Normalisation

One of the most important sets of normalisations arose from considering the data storage requirements for a flight. During initial entity discovery I tried to record all the data visible on the website that was tied to a flight entity (with respect to the design assumptions laid out on the first page).

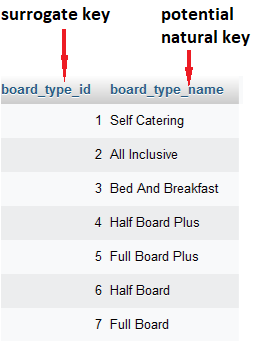
*Figure 4* is a representation of all this data in its initial and final form. To store multiple flights, it would be required to have much repetition of data in the form of repetitive airport information across many flight records. The data that describes the flight itself and the airport it flies to and from were normalised out into their own tables (flight and airport), and surrogate primary keys were added to enforce uniqueness of each record. While it would have been possible to store route information as two foreign keys in the flight table (departure and arrival airport ids), early on I identified my preferred method of pricing a flight as to price by route, date and time, and so this information was further normalised out into route and route\_price tables, and surrogate primary keys added to both.

This left our flight tables normalised up to the 3rd normal form, and importantly adhered to the Codd’s principles of further normalisation in that it:

* freed the collection of relations from undesirable insertion, update and deletion dependencies, for example:
  + an airport may be updated with a single update SQL statement to the airport table and this change would be propagated to all entities that use the changed record’s primary key as a foreign key automatically.
  + the price of a route may be amended with a single update to the route\_price table.
  + The route of a flight may be amended at the route level with a single update to the route table, or at the flight level by changing the value of the route\_id foreign key in the route table.
* and made the relational model more informative to users in that it more accurately represented the way a flight is considered – a close relationship between price, route, airport and the flight itself.

Another notable change from the initial diagram to the final diagram was that I dropped the estimated\_time attribute of the flight entity (which was intended to describe the time duration of a flight). This is instead implemented as a derived attribute calculated as (arrival datetime) – (departure datetime). This removed unfavourable update dependencies: for example, if a flight arrival time is updated, the duration of the flight is implicitly updated as the calculation will reflect this new duration.

**Primary Keys, Foreign Keys, and their Constraint Considerations**

**Unique Primary Keys**

Throughout the project I chose to use exclusively surrogate primary keys. That is, PRIMARY KEY attributes of type INTEGER used to uniquely identify each record in a table by means of AUTO INCREMENT flag.

At several points I considered using natural primary keys. As in the example (figure x)*,* at first the board\_type\_name seemed like a good candidate for a natural primary key, but to ensure that this table’s design would enjoy a long lifespan, I chose a surrogate key.

Figure 10 Natural Key Consideration in board\_type table.

I did so as I wanted to isolate the data being stored from its method of unique identification, so that it would remain trivial to edit the board\_type\_name without also violating foreign key constraints across all tables which potentially used the natural primary key board\_type\_name as a foreign key (bearing in mind my strict adherence to RESTRICT ON UPDATE and DELETE constraints discussed in the following section). This consideration was also apparent in several other “list” entities containing only a single piece of data, for example language, email, card\_vendor, telephone\_type.

The one of the main downsides to using a surrogate primary key is that extra unique constraints may have to be added on attributes that might otherwise be candidates for a natural primary key, to ensure that uniqueness is maintained for such attributes that require uniqueness for their business meaning.

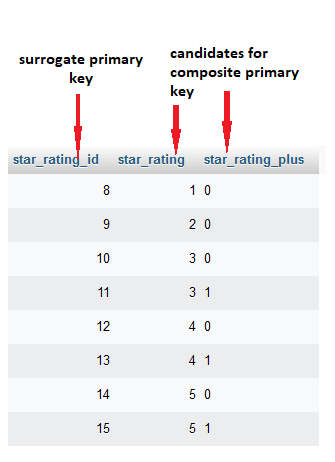
At several points I also considered using composite primary keys, one example is the the star\_rating entity (*figure 5*). On the website I noted that the star rating of a website was a combination of integer values (1-5 inclusive), with an optional “plus” attribute that may or may not be present for a given hotel. The plus was a Boolean value, and so I implemented it in our database as a non-null bit(1) type, allowing only true (1) / false (0) values to be present. This a relevant place to note that the hotel pages on the website had this star rating stored as a decimal number without this flag (plus flag being equivalent to a 0.5 decimal faction of the star rating, 3 plus = 3.5), so perhaps this attribute structure does not mirror the Jet2Holiday implementation as much as it initially seemed to. Given more time this change would have been enacted.

Figure 11 Composite Primary Key Consideration in star\_rating table

Each star rating was therefore defined by me a composite of its star\_rating integer and the boolean/ non-null bit(1) star\_rating\_plus value. Initially I intended to use star\_rating/ star\_rating\_plus as a composite primary key, but as in the considerations around board\_type above, I wanted to isolate the data itself from its unique method of identification, so the proposition of using this composite primary key was discarded for the use of a surrogate primary key. Although star rating values are unlikely to be changed in the future, it seems prudent to consider the fact that perhaps in the future further subtypes or rating beyond the integer and bit value combinations may be added, and I did not wish to rely on the composite key for uniqueness after potential changes in the future.

**Non-Unique Foreign Keys, their Cardinality and Constraints**

As in all relational databases, relationships between entities are established using references to unique primary keys by non-unique foreign keys. Where a primary key is used to enforce uniqueness of a record in a table, a foreign key leverages the primary key’s uniqueness to create a relationship between a record containing a primary key in one table, to a record containing a foreign key in another table.

In my database I set up all foreign keys with foreign key constraints that RESTRICT the primary key record from being deleted (ON DELETE) or having the primary key value itself updated (ON UPDATE). This means that if a primary key is referenced by a foreign key in another entity, the record that the primary key references may not be removed nor have the value of the primary key updated whilst it is being referenced by the foreign key in any record. In this manner we ensure referential integrity is maintained between our tables, and avoids records being orphaned in the table containing the foreign key. There are other constraints that MySQL permits, such as SET NULL, NO ACTION (an alias for RESTRICT in MySQL) and CASCADE, however I chose to work exclusively with RESTRICT as I did not wish to accidentally remove – or set to NULL – records in referenced tables unless I intended to do so explicitly.

Foreign keys are very powerful, not only in that they allow us to create the relationships between tables, but in that they can allow us to define the cardinality of a relationship – such as 1:1 (strict and 1:0), 1:N and N:M relationships.

One-to-one relationships (1:1 and 1:0)

I found no compelling reason to use a strict 1:1 relationship anywhere in the database. This relationship is very restrictive and only finds use in very specific circumstances. What we do have in our table is several 1:0 relationships.

An example of these is the relationship between a booking\_contact table record and its corresponding review table record. This 1:0 relation describes the relationship between these two entities as follows: a booking\_contact can leave one, or no reviews for a booking (via the review.reviewer\_id foreign key to booking\_contact.booking\_contact\_id primary key relationship). This 1:0 cardinality is implemented using a standard 1:N relationship, with an additional uniqueness constraint on the foreign key, which restricts the booking\_contact\_id from appearing more than once in the review table.

Our intent for records in the booking\_contact table was they were not to be reused, as it is a key link in the relationships that defines a single booking. If the same person were to make another booking, we would create a new booking\_contact record, so that the original booking is untouched. Potentially the email\_id and address\_id for this new booking\_contact may be reused however, to reduce repetition of data. In this way, it seemed appropriate enact a 1:0 relationship here.

Another example of 1:0 relationship is the hotel to payment.booking\_id relationship. This can be described as a booking may

One-to-many relationships (1:N)

By far the most common type of relationship cardinality is the venerable 1:N relationship. In such relationships a record in one table may be related to (via its primary key) many records in other tables (via foreign keys that reference this primary key).

One of the most important 1:N relationships in the entire database is that between the booking (via the booking\_id primary key attribute) and room\_booking tables (via the room\_booking\_booking\_id foreign key attribute). The is relationship is central to how our database associates a booking with its booked rooms. This relationship is also a component of several N:M relationships discussed in the next section.

There is a further N:1 relationship between room\_booking (via room\_booking\_room\_type\_id foreign key attribute) and room\_type (via the room\_type\_id primary key attribute). In a similar manner there is another N:1 between the room\_type and hotel tables. This chain of 1:N relationships is characteristic of how many, tangentially, related tables in a database can be related together through a series relationships. In this example the booking table has an “indirect” relation to the hotel table, which can be exploited using joins when executing queries/ statements.

A good demonstration of this in our database is the important of the hotel\_id primary key of the hotel entity. This primary key is an instrumental one and defines the direct relationships of the hotel table to: hotel\_facility, hotel\_bullet, hotel\_board\_type, room\_type and hotel\_image. And through these tables to a great number of others indirectly

Many-to-many relationships (N:M)

N:M relationships are a more complex relationship than a 1:N, as is shown in the extra entity complexity required in implementing them. N:M’s are a combination of two 1:N/N:1 relationships linked together by a third joining table.

The simplest N:M relationship is that between room\_type and room\_facility, which is maintained by 2 separate 1:N/N:1 relationships in the room\_type\_facility joining table. A room\_type (such as Double Room) can have many room\_facilities (such as Balcony or Air Conditioning), and a room\_facility can be present in many room\_types. The room\_type\_facility table contains the foreign key constraints that define this relationship.

There are more complex N:M relationships present in my database. The room\_booking entity is the joining table for two separate but related N:M relationships. These are the booking to hotel\_board\_type and booking to booking to room\_type relationships. Semantically this can be described as: A booking can have many room\_types and hotel\_board\_types and conversely room\_types and hotel\_board\_types can be present in many bookings.

The room\_booking table can be said to be a list of cross-references between various booking/ room\_types and booking/ hotel\_board\_types. Foreign key constraints are then placed on the room\_booking table that enforce the actual references between these tables.

**Data Types**

**VARCHAR**

Any short text strings, such as name or titles storage attributes were set as VARCHAR(255) (for example,address\_line\_1 and address\_line\_2 attributes of the address entity, hotel\_name attribute of hotel entity, or region\_descriptor\_title of the region\_descriptor entity).

This seemed appropriate as these types of strings should not exceed the 255 maximum character limit. In the early stages of the project, I considered trying to be more restrictive with my character limits by reducing the maximum size at declaration for these types of attributes for memory and storage utilisation reasons. Upon more reading I discovered that varchar types are true to their name. Unlike CHAR types, VARCHAR types do not right-pad their string data up to the length specified upon declaration. They store their values length information as a 1 or 2 byte prefix and subsequently uses only the required memory/ storage capacity to store the actual value. In this manner VARCHARs only uses the necessary amount of storage required to store the *actual* value, and not the theoretical maximum value stated upon declaration.

There were several points at which I chose to use a VARCHAR character limit other than 255. These were for short string data that is known to not exceed a certain size. Generally, I tried to avoid these decisions as VARCHAR is a dynamically sized type, but there were several attributes where it seemed appropriate. Notable examples of this are:

1. currency\_symbol attribute of the currency entity, which was set to VARCHAR(10). There are several currencies that use more than 1 character for their symbol. The longest I found was the Serbia Dinar which uses 4 characters (Дин. – from xe.com), which although not part of Jet2Holidays destinations, was worth considering. Bearing all this in mind I chose VARCHAR(10) to allow this entity’s design to enjoy a long life without the need for restructuring as per Codd’s principles of normalisation, while being able to suggest to users via the design of the database that this string data should not be particularly long.
2. airport\_iata\_code from the airport entity, which was set to VARCHAR(3). Jet2Holidays seems to use IATA standard airport codes, which are always 3 characters long.

**TEXT**

The TEXT type was used to store text values of indeterminate, but potentially long length. As with VARCHARs, TEXT type attributes are stored using nonbinary character sets which is appropriate for long plain text data. The choice of TEXT over VARCHAR can be simplified down to a question of length of string data, but there are also considerations to be made about collation and searching of data from TEXT attributes, which go beyond the scope of the project. In essence any character type that may be used in queries that inspect the text data itself, it best not stored as TEXT type. The TEXT types used in my final database were used in places where data is to be stored and retrieved for display, but not likely to be part of any actual queries WHERE or HAVING clauses.

Notable uses of the TEXT type in my final database:

1. hotel\_description attribute in the hotel entity, which is a string value of possibly very long length.
2. All image URL storage attributes (like hotel\_image\_url or room\_image\_url from hotel\_image and room\_image entities respectively) as the Internet Society and Internet Engineering Task Force Request for Comments Document 2616 states that “*The HTTP protocol does not place any a priori limit on the length of a URI [and SHOULD be able to handle URIs of unbounded length]”*, and so it seemed prudent to store these potentially very long strings using the TEXT type.

I recognise that the Jet2Holiday system likely enforces character limits on the various entities description-style attributes, to the point where the TEXT type may not be appropriate in a real system, and potentially a hinderance due to the downsides discussed above. For the purposes of this project, I decided that TEXT was still appropriate, as there was no way for me to discern these limits, if they existed, by analysing the website’s front-end.

In this section I will discuss some of the key normalisation decisions taken on the road to producing our database schema. This list of key decisions is far from exhaustive.

**DECIMAL**

This data type was utilised by all attributes that describe monetary values. DECIMAL type is particularly suited to monetary values as it is a fixed-point decimal number, that preserves precision up to the point specified in the declaration of the attribute’s type. DECIMALS allow definition of a numbers precision and its scale. When declaring a decimal you may define the total number of digits the number may represent, and (optionally) the amount of numbers after the decimal point.

Floating point decimal values such as FLOAT/ DOUBLE are unsuitable for storage of monetary information due to floating point numbers being approximate values with indeterminate precision. Precision is incredibly important when dealing with monetary values, especially for a system serving as many customers as Jet2Holidays. Where the loss of a fraction of a penny may not seem important for a single person day to day, potentially this could be a huge legal, financial and ethical liability for a commercial business over time and with increasing number of users.

I chose to represent all monetary data as DECIMAL(13,4). I could not find a completely authoritative source for how many decimal places a monetary value should store, but I found much information from several sources: Generally Accepted Accounting Principles (United States) state 4 decimal places, while the Federal Accounting Standards Advisory Board (also United States) state 3 decimal places is sufficient. I chose 4 decimal places to be careful. The 13 total digits was chosen because it represents a colossal currency value (~999 million when factoring in the 4 decimal places), likely outside the amounts that this database would need to handle. However I have seen advisories suggesting up to DECIMAL(28,6)! I acknowledge that this design decision could prove financially, and legally costly for the system using this database if my assumptions are incorrect.

Although normalisation was approached as a pragmatic on-going process in the design of this database, certain key normalisations were undertaken early on to bring our tables up to the 3rd normal form.

**Improvements**

Booking\_passenger could be replaced by adding a room\_booking\_passenger table, that links multiple passengers to a single room\_booking, which would record the same data as booking\_passenger, but allow an association between passengers and specific rooms. The functionality is not apparent on the website as we cannot progress pass payment stage

**APPENDIX X: Initial Entity Relationship Diagrams of Sub-Groups**

Figure 9 Booking ERD

Figure 8 Flight ERD

**Diagram

Description automatically generated**Timeline

Description automatically generated

Figure 12 Booking ERD ERD

Figure 13 Customer ERD

Diagram

Description automatically generated

**Diagram, schematic

Description automatically generated**

Figure 14 Airport ERD

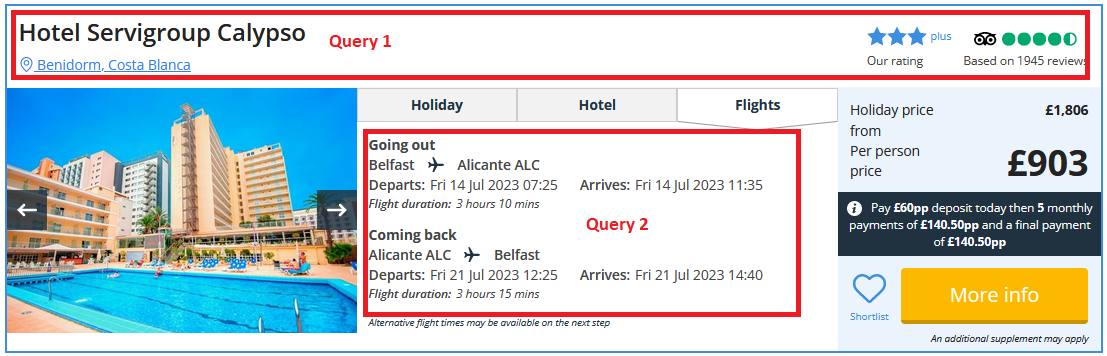
Figure 15 Accommodation ERD

**APPENDIX 1 : QUERIES**

Note Test data does not necessarily match website data

**Use Case - 1:**

**Generate the data visible on a hotel “card” on the website after using main search form**



**Use Case 1 – Queries 1:**

SET @HotelName = 'Spanish Hotel 1';

SELECT

  hotel.hotel\_name AS HotelName,

    region.region\_name AS RegionName,

    resort.resort\_name AS ResortName,

    star\_rating.star\_rating AS OurRating,

    star\_rating.star\_rating\_plus AS OurRatingPlusFlag,

    CAST(SUM(review\_rating.review\_rating) / COUNT(review.review\_id) AS DEC(2,1)) AS ReviewScore,

    COUNT(DISTINCT review.review\_id) AS NumberOfReviews

FROM hotel

INNER JOIN resort ON resort.resort\_id = hotel.hotel\_resort\_id

INNER JOIN region ON region.region\_id = resort.resort\_id

INNER JOIN star\_rating ON star\_rating.star\_rating\_id = hotel.hotel\_star\_rating\_id

INNER JOIN room\_type ON room\_type.room\_type\_hotel\_id = hotel.hotel\_id

INNER JOIN room\_booking ON room\_booking.room\_booking\_room\_type\_id = room\_type.room\_type\_id

INNER JOIN booking ON booking.booking\_id = room\_booking.room\_booking\_booking\_id

INNER JOIN booking\_contact ON booking\_contact.booking\_contact\_id = booking.booking\_contact\_id

INNER JOIN review ON review.reviewer\_id = booking\_contact.booking\_contact\_id

INNER JOIN review\_rating ON review\_rating.review\_rating\_id = review.review\_rating\_id

WHERE hotel.hotel\_name = @HotelName

GROUP BY hotel.hotel\_id;

**Use Case 1 – Results 1:**

Graphical user interface, application, website

Description automatically generated

**Use Case 1 – Queries 2:**

-- parameters gathered from jet2Holidays main search feature form

SET @OutboundDate = '2023-01-01';

SET @Duration = 2;

SET @HomeAirportId = 5; -- Belfast Internation Airport ID

-- parameter from website card shown above, changes for each card

SET @HotelId = 11; -- 'Spanish Hotel 1' Id

-- get route ids & compute return date

SET @AwayAirportId = (SELECT hotel.hotel\_serving\_airport\_id FROM hotel WHERE hotel.hotel\_id = @HotelId);

SET @OutboundRouteId = (SELECT route.route\_id FROM route WHERE route.departure\_airport\_id = @HomeAirportId AND route.arrival\_airport\_id = @AwayAirportId);

SET @ReturnRouteId = (SELECT route.route\_id FROM route WHERE route.departure\_airport\_id = @AwayAirportId AND route.arrival\_airport\_id = @HomeAirportId);

SET @ReturnDate = DATE\_ADD(@OutboundDate, INTERVAL @Duration DAY);

-- outbound flight info query

SELECT

  dep\_airport.airport\_name AS DepartureAirport,

    dep\_airport.airport\_iata\_code AS DepartureAirportCode,

    arr\_airport.airport\_name AS ArrivalAirport,

    arr\_airport.airport\_iata\_code AS ArrivalAirportCode,

    flight.departure\_utc\_datetime AS DepartureTime,

    flight.arrival\_utc\_datetime AS ArrivalTime,

    TIMEDIFF(flight.arrival\_utc\_datetime, flight.departure\_utc\_datetime) AS Duration

FROM flight

INNER JOIN route ON route.route\_id = flight.route\_id

INNER JOIN airport AS dep\_airport ON dep\_airport.airport\_id = route.departure\_airport\_id

INNER JOIN airport AS arr\_airport ON arr\_airport.airport\_id = route.arrival\_airport\_id

WHERE flight.route\_id = @OutboundRouteId LIMIT 1;

-- return flight info query

SELECT

  dep\_airport.airport\_name AS DepartureAirport,

    dep\_airport.airport\_iata\_code AS DepartureAirportCode,

    arr\_airport.airport\_name AS ArrivalAirport,

    arr\_airport.airport\_iata\_code AS ArrivalAirportCode,

    flight.departure\_utc\_datetime AS DepartureTime,

    flight.arrival\_utc\_datetime AS ArrivalTime,

    TIMEDIFF(flight.arrival\_utc\_datetime, flight.departure\_utc\_datetime) AS Duration

FROM flight

INNER JOIN route ON route.route\_id = flight.route\_id

INNER JOIN airport AS dep\_airport ON dep\_airport.airport\_id = route.departure\_airport\_id

INNER JOIN airport AS arr\_airport ON arr\_airport.airport\_id = route.arrival\_airport\_id

WHERE flight.route\_id = @ReturnRouteId LIMIT 1;

**Use Case 1 – Results 2:**





**Use Case - 2:**

**Demonstrate Time & Date Dependent Flight Pricing**

**Use Case 2 – Queries 1** *(ran once with each datetime variable below)***:**

SET @DateTimeToCheck = '2023-01-01 03:00:00';

-- SET @DateTimeToCheck = '2023-01-01 14:00:00';

-- SET @DateTimeToCheck = '2023-01-01 21:00:00';

-- SET @DateTimeToCheck = '2023-01-02 12:20:00';

SET @RouteId = 10; -- 'Spain to Belfast'

SELECT

  route\_price.route\_price\_gbp AS FlightPrice,

    @DateTimeToCheck AS FlightDateTime,

    route\_price.route\_price\_valid\_from\_datetime AS ValidFrom,

    route\_price.route\_price\_valid\_to\_datetime AS ValidTo FROM route\_price

INNER JOIN route ON route.route\_id = route\_price.route\_price\_route\_id

WHERE route\_price.route\_price\_route\_id = @RouteId

AND @DateTimeToCheck BETWEEN route\_price.route\_price\_valid\_from\_datetime AND route\_price.route\_price\_valid\_to\_datetime

AND route.route\_id = @RouteId;

**Use Case 2 – Results 1**

**Text

Description automatically generated with low confidence**

**Graphical user interface

Description automatically generated with medium confidence**

**Graphical user interface

Description automatically generated with medium confidence**

**A picture containing graphical user interface

Description automatically generated**

**Use Case - 3:**

**Hotel Reviews**

Graphical user interface, text, application, email

Description automatically generated

**Use Case 3 – Queries 1:**

SET @HotelName = 'Spanish Hotel 1';

SELECT

    hotel.hotel\_name AS HotelName,

    town\_city.town\_city\_name AS ReviewerTown,

    CountryName.location\_name AS ReviewerCountry,

    passenger.passenger\_first\_name AS ReviewerFirstName,

    passenger.passenger\_last\_name AS ReviewerLastName,

    DATE(review.review\_timestamp) AS ReviewDate,

    review\_rating.review\_rating AS ReviewRating,

    review.review\_content AS Review

FROM review

INNER JOIN booking\_contact ON booking\_contact.booking\_contact\_id = review.reviewer\_id

INNER JOIN passenger ON passenger.passenger\_id = booking\_contact.booking\_contact\_passenger\_id

INNER JOIN booking ON booking.booking\_contact\_id = booking\_contact.booking\_contact\_id

INNER JOIN review\_rating ON review\_rating.review\_rating\_id = review.review\_rating\_id

INNER JOIN address ON address.address\_id = booking\_contact.booking\_contact\_address\_id

INNER JOIN town\_city ON town\_city.town\_city\_id = address.town\_city\_id

INNER JOIN country ON country.country\_id = town\_city.town\_city\_country\_id

INNER JOIN location\_name AS CountryName ON CountryName.location\_name\_id = country.country\_location\_name\_id

INNER JOIN room\_booking ON room\_booking.room\_booking\_booking\_id = booking.booking\_id

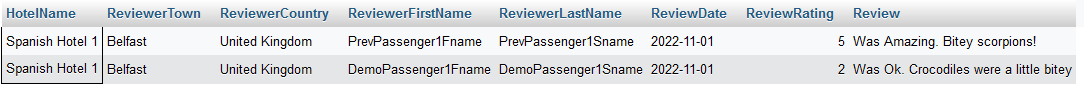
INNER JOIN room\_type ON room\_type.room\_type\_id = room\_booking.room\_booking\_room\_type\_id

INNER JOIN hotel ON hotel.hotel\_id = room\_type.room\_type\_hotel\_id

WHERE hotel.hotel\_name = @HotelName

GROUP BY review.review\_id;

**Use Case 3 – Results 1:**



**CUTS**

The amount of work to piece these diagrams together was significant, and many changes had to be made. The entirety the booking subgroup’s ERD tables had to be dropped for several reasons. The flight\_info, travellers and guests entities were dropped as they had been implemented in other diagrams already, and were not directly related to booking data. The guest health, transfer, and transfer\_option entities were out of scope as per our initial design assumptions and so were also dropped. This left the booking and booking\_info tables, which were also completely redesigned because the relationships therein were not indicative of the real world system we were reverse engineering. For example the booking and booking\_info tables were arbitrarily normalised apart, when the data in both tables were attributes wholly dependent on the booking\_id primary key. These initial 2 booking entities also made no attempt at managing dates, nor duration of the package holiday.

The airport ERD was more useful, but it also included out of scope information (for example the plane and ticket entities, the hand\_luggage attribute of the flight entity) and contained several attributes that were previously implemented in other subgroups ERD’s (for example the country/region and town/area attributes of the Airport table) which was a shortcoming of splitting into subgroups.

The splitting of flight data into 3 distinct tables of route, flight and airport was maintained through to the end of the project, although the attributes therein underwent significant changes, and a route\_price entity was later added to allow for route date and time dependent pricing. The route table attempted to describe both the outbound and return flights for a booking, which violated the 1st and 2nd normal forms. The 1st normal form was violated because the dep\_airport\_out, dep\_airport\_in, arr\_airport\_out, arr\_airport\_in entities were not foreign keys and so would create much repetition of airport data through each record of the table. Additionally, these 4 attributes violated the 2nd normal form by introducing partial dependencies on the primary key route\_id. These 4 attributes variously describe 2 separate routes, and so were subsequently changed to be departure\_aiport\_id and arrival\_airport\_id in subsequent revisions, to more naturally describe a single route an aircraft may travel.

The hotel/ accommodation ERD provided was a great basis to start with. Many changes were implemented in this as further normalisation was undertaken. Examples of this are the Board Types entity (board\_type in the final design) which had an unsatisfactorily defined 1:N relationship with hotels. After further analysis of the Jet2Holidays website, it was discovered that although board bases were shared across hotels, each hotel had its own unique price and description and so a linking entity named hotel\_board\_type was added to allow us the ability to use the board\_types between many hotels, whilst allowing each hotel its own description and pricing of the board\_type

**APPENDIX 2 : FINAL DRAW IO DIAGRAM**

**Put it here**