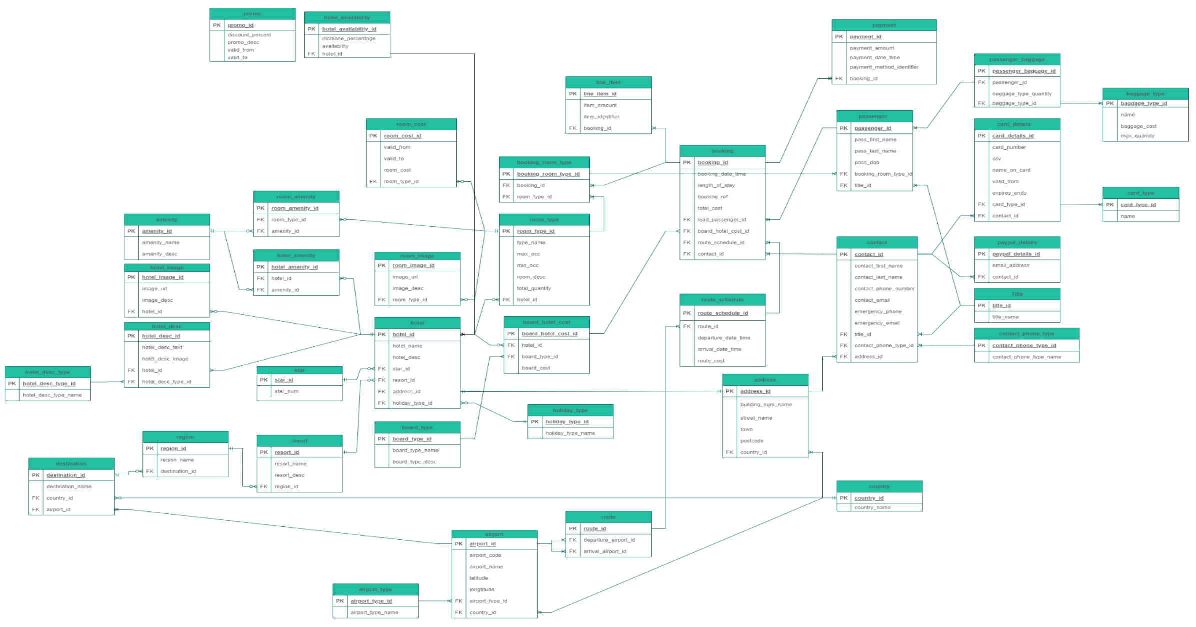
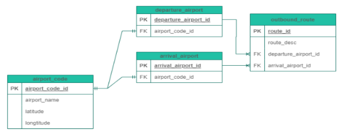
DESIGN OF A DATABASE FOR THE PACKAGE HOLIDAY BOOKING SYSTEM

Initial elaboration from the group ERD model and entity discoveries

As I was late to the entity discovery because of my late enrollment, I did not fully understand some of the key design decisions my group made so I had to recreate some of the tables from scratch and modify many aspects of them. I built the whole ERD system to have a better understanding and functionality of my personal version of the database, then I compared it with my group ERD. The objective of this new system should be to be fully autonomous and working with a deeper focus on the room booking process in comparison with the previous model. We suppose that the database is not connected with an external source for the management of the room booking system. The starting point of my project began with the analysis of the flight booking system from our initial ERD model, designed to represent jet2holidays.com.





The flight booking system, destinations and positions

I started by modifying the mechanism of the flight booking system. I am using a route table, a flight table and an airport table to manage the flights. Each flight is connected with the booking table with a one-to-many relationship to allow me to have more than one flight in a single booking. Once the booking is created with a transaction, we also need to store the flight choices, we cannot book anything without a flight. Because more customers in the real world should be allowed to take the same flight, each flight is connected to a route and the route with the airport table. The airport table lists all the airports, and with the route table we decide which route should be allocated to our flight with a one-to-many relationship, then the flight table connects booking and route in a many-to-many relationship. In the airport table we need to store information like the geolocation. For latitudes I decided to use: Decimal(8,6), and for longitudes use: Decimal(9,6) (6 figures of a total of 9 should be on the right of the decimal point) in order to obtain ~10cm of accuracy in the map I only need 6 decimal places of precision to store. I am also using route_description and route_name in order to include more details and security policies in the specific route selected. Instead of using the route table and the route_schedule table as we did in our group, I decided to create a flight table that contains all the information about a specific flight. To retrieve the total price we assume that the website gives you the opportunity to select a determinate type of flight. For example, for a flight with extra luggage the website gives you the opportunity to select a flight_type to meet all your needs and then retrieve the total price from the flight company website. We suppose that the price for each flight comes from the external service by connecting the database with an API. I also added the number of seats available at the beginning, but it should be also updated with the same method, because if we book the flight from an external website the number of seats may change and the price with it. In the flight_type we store all the information that the user needs to check the type of flight he booked. In this case we need to store for him the luggage information, meals and any other additional services included with that particular flight in order to allow the customer to retrieve all the needed information. We can use a many-to-many relationship to better allocate each luggage and service we need separately, to allow more flexibility, but we consider in our model that it is all included with the particular type of flight booked, and we do not want repetition of similar concepts. The total price of the flight is a snapshot of the actual price of the external provider at the moment of the booking, so is it not necessary to store this data, we can just sum it to the total price of the booking. There is data in our database that is added only at the end of the booking, because of the volatility of the information. We store the data in a virtual memory and use a transaction to save it in the disk only if the user succeeds with the payment and the final process of the booking. If the user waits too long the price may have changed, which is why the website forces you to put all your information again if you stay idle for too long on the booking page.

At the beginning of my model, I was using a table for all the addresses in order to avoid as many repetitions as possible in my database. I also connected this table with the city_id. It is possible to create another table only for postcode to avoid repetitions, but I think that is not required and can make the database more difficult to understand. Then I figured out that the address table is different on some occasions, for example for the room address I should separate the room number and other elements, whereas in the booking address for the lead person the address is stored as a whole in address_line_one and address_line_two, the same concept is used in the website. I added a city table to show all the different cities and which country, region and county they are connected to. The city is part of a bigger set and consequently, by establishing the set we can automatically find which region, county or country it is part of. We do not know if every country has a county or a region, so in the end I decided to separate the tables and connect county and region directly with the city table instead of connecting them between the city and the country table in a hierarchical order. Counties and regions are not mandatory fields, so I had to change the foreign keys of these 2 tables in the city table. To do so I changed the “default” field in phpmyadmin from None to NULL and I modified the “Constraint properties” on “relation view” from “RESTRICT ON UPDATE” to “SET NULL”, to allow me to leave the field null. Each country can have many counties or regions, each one with their own description. I realized that it could be possible to end up with 2 counties or regions from different countries with the same name. If there is no description on them it is possible to merge them, but in a practical world we would never do it, so I decided to add to both of them the foreign key country_id, in this way the database knows which region and county we are talking about in a filter or research function. I could also retrieve in the same way all the counties or regions connected with a particular country, or all the cities of that country without repeating the country, region or county for each city in the table, which is why I decided to use a different table to include cities and countries instead of using them in a common address table. The city name should help us to find the correct city when we insert data manually in phpmyadmin, but there is the possibility that we have 2 cities with the same name in different countries. We cannot simply select one of the repeated cities randomly, because each city has a different description, so we need to be careful and check the country_id of that city. The system could be improved by adding for similar cities the name of the city and the country in the city_name column, or to show city_name and country_name merged together in the selection menu if there is a way to do so. I also modified the tables to include a short description (city_description) for each country to give the possibility to add more details about the chosen destination, currency information or give the flexibility to include important current regulations, for example we can use it to check the current covid policies to respect in that specific country and all the mandatory documents you have to bring with you. I also included the possibility to add an URL to bring the user to the right website to be updated about the policies (for example uk.gov for visa regulations). I am using a varchar with length 2048 because it is the maximum length that is possible to use for a URL. Varchar can store up to 65535 characters in MySQL 5.0.3 and later versions, but before it the maximum was only 255 characters. In some cases, the description of the county or city or region can coincide with the one of the country, in which case we leave these fields empty and we take the higher one, by using a hierarchy concept. (check image 1).

Customer information management and protection of sensitive data

My initial design from the group has a table called contact that connects the contact information with the passenger table by using the booking table. I could not find information about the passengers that are not leading by using the booking reference in this way and I cannot associate to my booking all the involved persons. Then there was also repetition in the title, name and surname that was not necessary, because the leader already had this information in the table passenger. I am not sure about the choice of this structure so I decided to rebuild it from scratch without going deeper in other aspects and problems I could find along the way. For my model I decided to split the table into 2 different tables that store the information separately for the customer_lead and the customer. The customer lead has different parameters like the number, emergency number, email, address and postcode that are mandatory on the jet2holidays.com website. I decided to store the number variables as a Varchar instead of int, otherwise it will ruin the formatting of the number, for example by removing preceding 0 or the impossibility to store symbols like the + char or the spaces that help a better readability. I split the customer_lead and the customer tables to avoid confusion in the data. If a customer is in the customer_lead table I will add new information mandatory only for the customer lead, as I’ve seen from the website. In this way it doesn't matter which customer I select, they are all in the same table initially, and I can collect their information jointly. Every customer is connected with the booking table, which means that if I need to search all the customers from a particular booking I can do it by selecting the booking I am interested in and consequently collect all the information about customers and see who is the lead one. I believe it is easier to manage the tables and update them in this way. I could not see mandatory information like the passport or id card number in the website, so I did not include it in my project.

Differently from my initial group project I am connecting the customer table and the booking table with a many-to-many relationship by using the customer_booking table to show the number of customers and the customer_lead between them, connected with that specific booking. I also included the foreign key customer_lead_id in the booking table to retrieve the customer leader directly. In this way it is easier to retrieve all the information about customers for that booking. One customer can be connected with more than one booking and one booking can have many customers and only one customer_lead. The booking table is comprehensive of flights, rooms and other services, and the final price will be the total price by including all these products.The programmer will manage the customer_leads, because if the age is less than 18 the system should not allow you to allocate a customer_lead to that specific customer_id. In my model I suppose that the system will store only the birth date of the customer and not if it is a child or an adult. The programmer should count the number of adults and children where their age is less than 18.

In my booking table I added a discount code that is unique for each booking. Once the discount code is stored it is impossible to include the same code for another booking, It should be unique. If a discount code was already used in a previous booking, we need to store it to check if someone else already used it. In this case I should force the database to avoid repetitions for this field by rendering it UNIQUE, but I decided to not bother with it because I want to reuse a discount code or a reference number if the previous booking was already concluded years ago. For marketing reasons, it is very important to have the possibility to apply discounts when required and store this data for the customer. In my model I also included a payment_method_id in the booking table. The service needs to store which type of payment method was used in order to solve any kind of payment issues such as a canceled booking or a dispute opened by the client with PayPal. I have also included a booking_payment_reference to allow the system to find the transaction in the payment method chosen by the customer. We can join the booking table with the customer table and after finding the customer_lead we can retrieve the card number connected with him. If the payment method used is only PayPal we cannot find any card details connected with that particular customer lead. If we use PayPal as a payment method, PayPal will manage the payment system and so, for this type of method we do not need to store any data, only the payment type used and the reference. The booking has a creation_timestamp in timestamp format and a end_contract_date in date format, we use the date format to store the time in our timezone (Greenwich Mean Time) because we are using this value internally to understand when we can clean our system from old data of previous bookings, but the creation_timestamp is used by the customer, so we need to convert it in its timezone, which is why we store it as a timestamp.

Each customer_lead has one or many credit/debit cards stored in the system. The system needs to store this information to be allowed to process the payment or to (if it is a credit card) withdraw the money from the card in a different moment, or for example to let you split the payment monthly. That's why for the purpose of this model I am using a card_details table. I will store both the card number and the CVV number to allow the system automatic withdrawal. For the purpose of this project, we will manage the payment information but in the real world we would have used an external service to manage this sensitive data because the legal rules or new systems of security evolve very fast, and they should be up to date on a daily basis. A website that we should consult for this purpose is PCI Security Standards Council (PCI SSC): https://www.pcisecuritystandards.org/. In a real system we would use a token to connect our payment details with the external service website to manage the payment information of the customer, and we would store only the token instead of the real data. The external service will do the rest for us. Sensitive data like passwords or credit card details usually should be encrypted to avoid a hacker or one of the employees of the company using it and logging in with it in other websites. For example, for what concerns the login process, if we use the same password in more than one website, the hacker can use it to access our accounts from other websites. We can use tools like https://haveibeenpwned.com/ to find if a website lost our data and what kind of data. If the data was not encrypted and we used it to log in to other websites, we are not safe anymore. Even if the customer data was encrypted, we are still not safe, because the website itself can steal our data. We should also take into consideration the type of connection that we are using on the website page. If the protocol used on the website is HTTP, we should never type sensitive data like a password on it, it should be used only if it uses an SSL (Secure Sockets Layer) connection to encrypt HTTP, the S at the end stands for secure, so the URL should start with HTTPS. In image 1 below we can see how the customer tables are connected together, this is a portion of our whole final ERD.

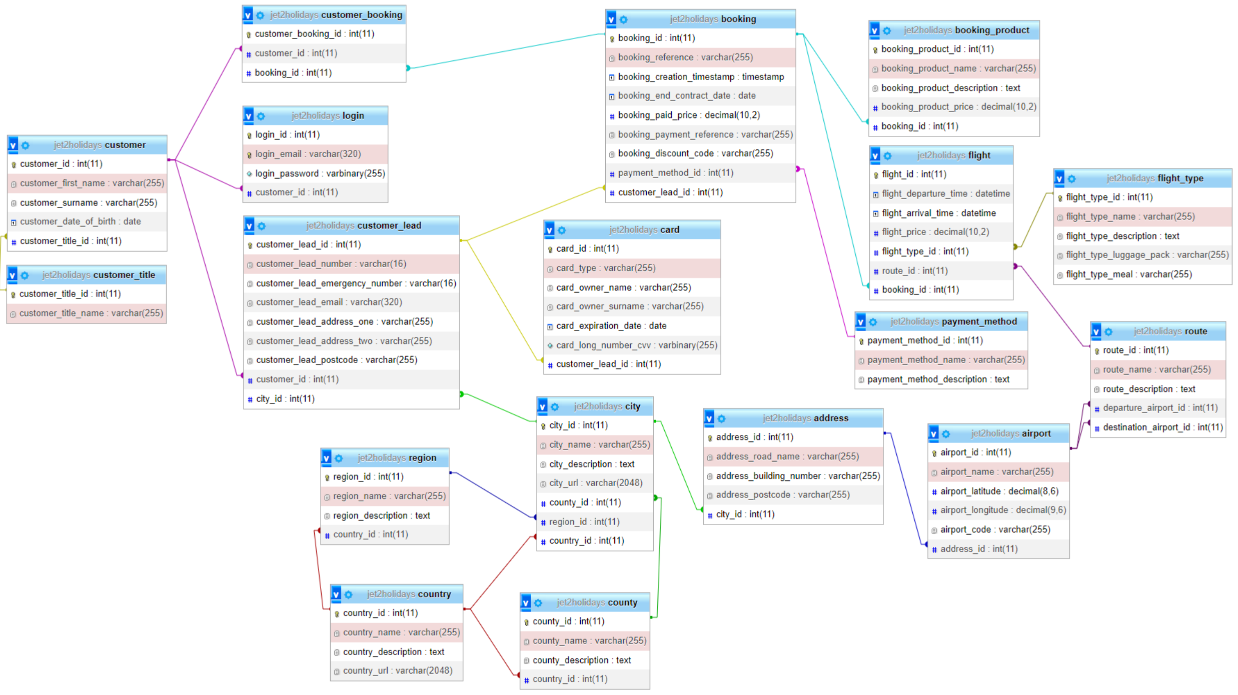

jet2holidays customer_booking
customer_booking_id : int(11)
customer_id : int(11)
booking_id : int(11)
jet2holidays booking
booking_id : int(11)
booking_reference : varchar(255)
booking_creation_timestamp : timestamp
booking_end_contract_date : date
booking_paid_price : decimal(10,2)
booking_payment_reference : varchar(255)
booking_discount_code : varchar(255)
payment_method_id : int(11)
customer_lead_id : int(11)
jet2holidays booking_product
booking_product_id : int(11)
booking_product_name : varchar(255)
booking_product_description : text
booking_product_price : decimal(10,2)
booking_id : int(11)
jet2holidays login
login_id : int(11)
login_email : varchar(320)
login_password : varbinary(255)
customer_id : int(11)
jet2holidays customer
customer_id : int(11)
customer_first_name : varchar(255)
customer_surname : varchar(255)
customer_date_of_birth : date
customer_title_id : int(11)
jet2holidays customer_title
customer_title_id : int(11)
customer_title_name : varchar(255)
jet2holidays flight
flight_id : int(11)
flight_departure_time : datetime
flight_arrival_time : datetime
flight_price : decimal(10,2)
flight_type_id : int(11)
route_id : int(11)
booking_id : int(11)
jet2holidays flight_type
flight_type_id : int(11)
flight_type_name : varchar(255)
flight_type_description : text
flight_type_luggage_pack : varchar(255)
flight_type_meal : varchar(255)
jet2holidays customer_lead
customer_lead_id : int(11)
customer_lead_number : varchar(16)
customer_lead_emergency_number : varchar(16)
customer_lead_email : varchar(320)
customer_lead_address_one : varchar(255)
customer_lead_address_two : varchar(255)
customer_lead_postcode : varchar(255)
customer_id : int(11)
city_id : int(11)
jet2holidays card
card_id : int(11)
card_type : varchar(255)
card_owner_name : varchar(255)
card_owner_surname : varchar(255)
card_expiration_date : date
card_long_number_cvv : varbinary(255)
customer_lead_id : int(11)
jet2holidays payment_method
payment_method_id : int(11)
payment_method_name : varchar(255)
payment_method_description : text
jet2holidays route
route_id : int(11)
route_name : varchar(255)
route_description : text
departure_airport_id : int(11)
destination_airport_id : int(11)
jet2holidays city
city_id : int(11)
city_name : varchar(255)
city_description : text
city_url : varchar(2048)
county_id : int(11)
region_id : int(11)
country_id : int(11)
jet2holidays address
address_id : int(11)
address_road_name : varchar(255)
address_building_number : varchar(255)
address_postcode : varchar(255)
city_id : int(11)
jet2holidays airport
airport_id : int(11)
airport_name : varchar(255)
airport_latitude : decimal(8,6)
airport_longitude : decimal(9,6)
airport_code : varchar(255)
address_id : int(11)
jet2holidays region
region_id : int(11)
region_name : varchar(255)
region_description : text
country_id : int(11)
jet2holidays country
country_id : int(11)
country_name : varchar(255)
country_description : text
country_url : varchar(2048)
jet2holidays county
county_id : int(11)
county_name : varchar(255)
county_description : text
country_id : int(11)

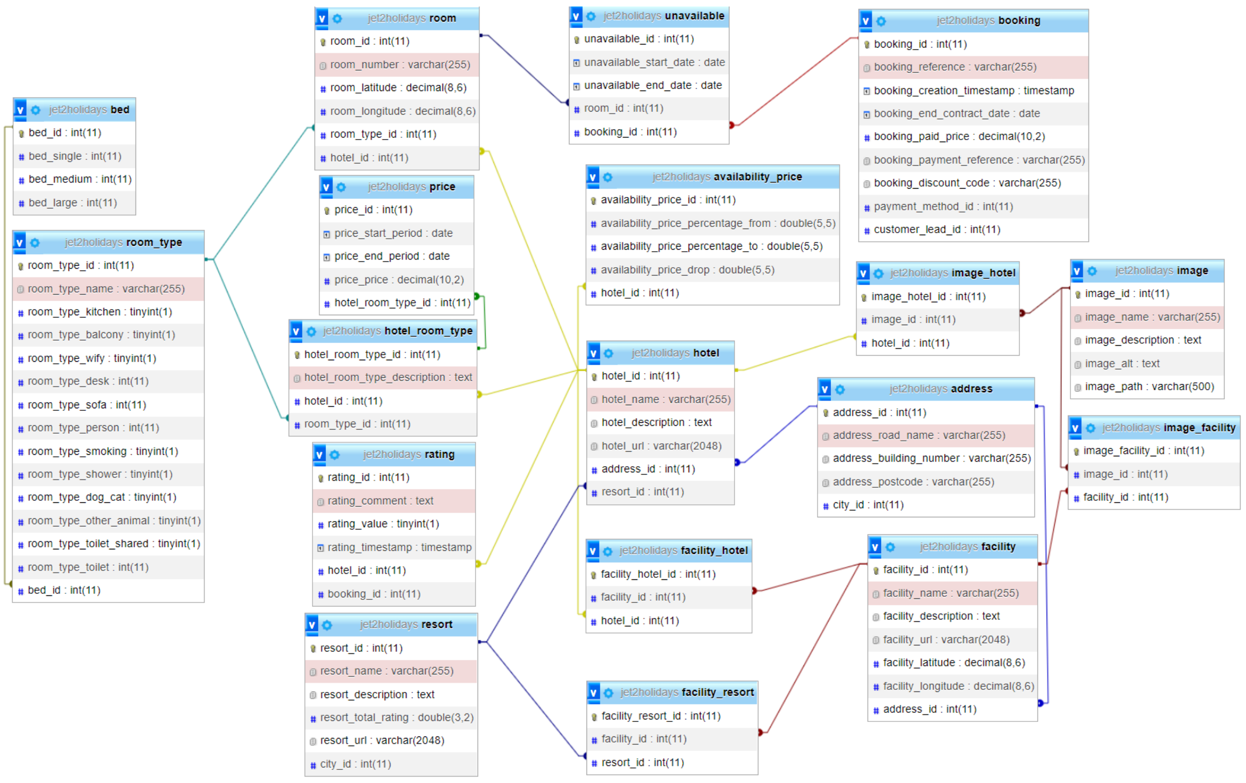
Image 1 - portion of the ERD we explained so far

For the purpose of this project I decided to store the CVV and the card number directly in the same column and use the function CONCAT() before the encryption of them. I decided to encrypt them together with the AES_ENCRYPT() function that uses the Advanced Encryption Standard to encrypt it with a password stored in our system. This method is not secure and should be avoided in a real-world system. We can fully recover our card_long_number and CVV by using the password, and a hacker that has access to that password can steal all the card details in our database. With the SUBSTRING() function we are able to split the CVV and the card number in two different variables, which will increase the security of our system and use less code to retrieve it. The result of the encryption should be stored as a varbinary because it is represented in a hexadecimal value. To obtain the encrypted CVV and card number we use the same password and the function called AES_DECRYPT(), then we only need to separate the strings. Each card has a unique customer_lead_id, which is how we connect the card to its owner and how we can retrieve this data from the booking table. Our model should use these information to make automatic payments, but systems like Google Pay ask the user to put the CVV number every time a transaction is made, which will increase the security, because we do not need to pay attention to store this information and at the same time we are sure that the user that is going to pay is the owner of the card.

In my version of the database, I wanted also to include the possibility to sign in on the website and log in with your credential. I added a new table called login and I connected it to the customer table. Every login_id has only one customer_id. It is not mandatory to log in to the system but it will be easier to find your personal information and bookings. To do so I decided to use the one-way hash function encryption, salted with the corresponding hashed random number between 0 and 1 generated with the function RAND() that is added to the beginning of the password. For this project I will use the SHA2(data, 256) function. SHA256 uses a 256-bit long key to encrypt data and will ensure a stronger encryption for the password. The final length of the hash stored should be 64 characters long, so we use a substring from 7 to 70 to divide it from its salt of 6 characters. The reason we encrypt the password with this method is because we do not want to lose the sensitive information of the customer, because it can be used on other websites. In this case the password should be used only as a proof of login and we never want to store this information raw in the database. As I mentioned before these systems are updated constantly and it will be safer to use an external service to manage it. After generating the salt with SHA1 (160-bit long key), we take a small portion of it, 6 characters, and then we concatenate it with the password typed by the user at the sign in process. Then we concatenate the hashed-salted password with the salt and we store it in the database. When we log in again, we just use again the same hashing function to hash again the password concatenated with the same salt stored in our database, and we compare it with the hashed salted password we stored previously in the database. We used the salting system to increase the security of the encryption because if we use only the one-way hashing it is easy to decrypt it by using a dictionary of hashes that we can find in https://crackstation.net/, we just need to copy paste the hash in this website and the hash can be decrypted. In the login table we are also restricting the repetition of the email in the column login_email, because we want a unique email for each registration. To do so we changed the index to UNIQUE. The maximum length of an email is 320 characters, so we are using a varchar 320.

Booking process, accommodation and facilities

In my initial model I decided to use one table for all the facilities. I modified our group model from scratch to allow me a better understanding of the whole mechanism. I connect hotel facilities, room facilities and resort facilities with many-to-many relationships. We suppose that for each flight there is always a room with the number of nights needed for our stay. We cannot book the flight without the room or the room without the flight, the website should give us a package all included with everything we need for our stay, to simplify the booking for the customer. The customer should be able to filter the research for the room he needs. In this case the board type can be something included with the facilities of the room, or the hotel, or the resort, which are already included in these tables. We can join the tables together to allow us to book the right room. For the room_type we have a separate table, where we include the number of beds, if the toilet is shared with other rooms, how many toilets, if there are kitchens shared or not, number of beds etc. For the specific room facilities, we can consider the possibility that on a particular floor there are services allowed only for that particular type of room, so we use the facility_room table to add these services. The facilities have a hierarchy level, as we did for countries and cities. In this way we can include a particular facility for the whole resort or only for a couple of rooms or hotels in it. In this way we have a wide flexibility in our model for filtering any single possible variable we need. As we did for cities and countries, we also have a description and an URL relevant only for that specific subset, and it is not a mandatory field. Successively I was not sure if I had to include facilities for rooms, because usually the facilities are for all the rooms of a particular hotel. In the end I decided to connect it with the hotel_room_type instead of the room table directly, with a many-to-many relationship facility_hotel_room_type. This decision is more reasonable, because all the rooms of the same type should share the same facility for the specific hotel, but I am still not sure about this decision. Anyway, the table can be left empty without causing any problem. In our model the address table doesn’t have latitude and longitude coordinates because I decided to include them in the specific room or airport. The address table should include only the building address, number and connect it with the city. We suppose that it is not practical to assign an address to a resort because there are many buildings on it. At the same time the address should refer only to a building, like a hotel, and to avoid repetition I decided to add the foreign key only in the hotel table. I already have the coordinates of my specific room and its number. To find the building or the address of the room I refer to the hotel table and find the address_id connected to it. We have a hierarchical concept of room, hotel and resort, so if our hotel doesn’t have a resort our system is not limited to it, because we start from the hotel table the filtering process. If we do not have a resort for a group of hotels, we simply list all the hotels that are not connected with a resort_id to that particular country. To simplify this process, I decided to modify my previous model by adding into the resort table the city_id. By searching the city we can find the list of resorts connected with that city and then we can just join all the hotels that don't have a resort_id (because the foreign key resort_id is NULL as default), in this way we can use the UNION ALL operator to combine and show in one column all the list of resort and 'hotel without a resort' very easily and retrieve all the data we need for the filtering process. Check image 2 for the accommodation ERD.


jet2holidays bed
bed_id : int(11)
bed_single : int(11)
bed_medium : int(11)
bed_large : int(11)
jet2holidays room_type
room_type_id : int(11)
room_type_name : varchar(255)
room_type_kitchen : tinyint(1)
room_type_balcony : tinyint(1)
room_type_wify : tinyint(1)
room_type_desk : int(11)
room_type_sofa : int(11)
room_type_person : int(11)
room_type_smoking : tinyint(1)
room_type_shower : tinyint(1)
room_type_dog_cat : tinyint(1)
room_type_other_animal : tinyint(1)
room_type_toilet_shared : tinyint(1)
room_type_toilet : int(11)
bed_id : int(11)
jet2holidays room
room_id : int(11)
room_number : varchar(255)
room_latitude : decimal(8,6)
room_longitude : decimal(8,6)
room_type_id : int(11)
hotel_id : int(11)
jet2holidays price
price_id : int(11)
price_start_period : date
price_end_period : date
price_price : decimal(10,2)
hotel_room_type_id : int(11)
jet2holidays hotel_room_type
hotel_room_type_id : int(11)
hotel_room_type_description : text
hotel_id : int(11)
room_type_id : int(11)
jet2holidays rating
rating_id : int(11)
rating_comment : text
rating_value : tinyint(1)
rating_timestamp : timestamp
hotel_id : int(11)
booking_id : int(11)
jet2holidays resort
resort_id : int(11)
resort_name : varchar(255)
resort_description : text
resort_total_rating : double(3,2)
resort_url : varchar(2048)
city_id : int(11)
jet2holidays unavailable
unavailable_id : int(11)
unavailable_start_date : date
unavailable_end_date : date
room_id : int(11)
booking_id : int(11)
jet2holidays availability_price
availability_price_id : int(11)
availability_price_percentage_from : double(5,5)
availability_price_percentage_to : double(5,5)
availability_price_drop : double(5,5)
hotel_id : int(11)
jet2holidays hotel
hotel_id : int(11)
hotel_name : varchar(255)
hotel_description : text
hotel_url : varchar(2048)
address_id : int(11)
resort_id : int(11)
jet2holidays facility_hotel
facility_hotel_id : int(11)
facility_id : int(11)
hotel_id : int(11)
jet2holidays facility_resort
facility_resort_id : int(11)
facility_id : int(11)
resort_id : int(11)
jet2holidays booking
booking_id : int(11)
booking_reference : varchar(255)
booking_creation_timestamp : timestamp
booking_end_contract_date : date
booking_paid_price : decimal(10,2)
booking_payment_reference : varchar(255)
booking_discount_code : varchar(255)
payment_method_id : int(11)
customer_lead_id : int(11)
jet2holidays image_hotel
image_hotel_id : int(11)
image_id : int(11)
hotel_id : int(11)
jet2holidays address
address_id : int(11)
address_road_name : varchar(255)
address_building_number : varchar(255)
address_postcode : varchar(255)
city_id : int(11)
jet2holidays facility
facility_id : int(11)
facility_name : varchar(255)
facility_description : text
facility_url : varchar(2048)
facility_latitude : decimal(8,6)
facility_longitude : decimal(8,6)
address_id : int(11)
jet2holidays image
image_id : int(11)
image_name : varchar(255)
image_description : text
image_alt : text
image_path : varchar(500)
jet2holidays image_facility
image_facility_id : int(11)
image_id : int(11)
facility_id : int(11)
Image 2 - initial model of the accommodation system

For the rating we have a separate table and we only show the rating only for a particular hotel, so I included in every resort a column for the rating that is not connected directly with the rating table, because the customer will never give a rating for the whole resort but for that particular hotel. In this case to retrieve the rating of a resort we just use an arithmetic average. there is nonetheless the possibility that the hotel has more rooms than another hotel in the same resort, in that case I consider it enough to use the number of ratings to weight the average of that particular resort, if a hotel has more rooms than another hotel, we have more ratings, and the average should be correct for the whole resort. There is no need for the rating of a singular room in the real world. The rating should also include a personal comment from the customer, and the customer for our model should be anonymous. We need to understand what can be improved and we need the customer to be honest about it. The rating is also connected with a limited number of images. There is no possibility that a customer will use the same photo multiple times and we do not want the user to overload our system with images, that's why we suppose the programmer to not allow in our database more than 5 photos for each review. After revising my model I thought that it is necessary to avoid the same person giving more than one review, so I modified the table again and I added a relation with the booking table, the programmer should allow more than one review only for different buildings that the same customer booked. To do so we use the booking_id and hotel_id to connect the review. Next, I also decided to add a timestamp to show the reviews in order of publication. The rating format is a value between 0 and 5, so we use a TINYINT to store the value. The data we show is of length 1 in this case. We never store the total rating, because it may change every second if someone adds a rating in the system, so instead of creating a hotel_total_rating or a resort_total_rating, we calculate it when it is required. We decided to not include rating in our initial model with the group, but I added it only internally, we do not want to store the rating already existing from external websites like TripAdvisor.

I also included a table for the image connected in a many-to-many relationship with hotels, rooms and rating only. I did this because for each resort we have a selected series of images, so we do not need a many-to-many relationship, a one-to-many is enough, so I tried to connected it directly with the image table, but the only way to connect it was with another table for a many-to-many relationship, so it is possible to use a many-to-many relationship for our resort but it is not needed. For the other tables, one hotel could share more than one image with other hotels, for example 2 hotels inside a resort could have the same view of the swimming pool in the same resort, same could be for the rooms in the same flat or with the same type. At the beginning I wanted to use the room_type table for the relationship, but even rooms of the same type can have a different image. For the image table we have included an image_alt, this is a short description of what the image shows, which can be useful if the user has a slow internet connection and the webpage needs to load the image, or (I suppose) it can be used in case of a browser search from a Linux server shell, in which case the user knows the content of the image without the need to see the actual image. I use TEXT to have better flexibility about the length of the description. I included the path of the image instead of the actual image, as I do not want to overload my database. The path could also be a link that connects an external cloud where we store the image. For the facilities the scheme is different, we have many-to-many relationships that connect not only rooms and hotels but also resorts, there is the possibility to have multiple resorts with the same facility (for example a swimming pool), and all of them should be connected with the table facility. The hierarchical system should also be applied for it, we do not want to include a facility at the resort or hotel level if that facility is specific only for the room, in that case we should include it only at the room level. In the same way if all the rooms of our hotel have that specific facility, we should include it only at the hotel level, if all the hotels in that particular resort have it we should include it only at the resort level. Again, we do not want to repeat the same data multiple times, which is why we need to respect relational database normalization.

We do not include the price of the room for each room_type on the room table, but instead we create a table called hotel_room_type. This table is very important because we set all the prices for each room type on it. It connects hotels and room_type in a many-to-many relationship. The concept of this choice of structure was built from scratch to allow a better organization of the system from the original group project, because I believe that exemplifies the system, and it works as it should in a real-world example. This table allows us to assign for each hotel a different price related to the room_type. In the real world we never have different prices for the same room type in the same hotel (same size, same number of beds, same balcony), that's why we set a starting price at the hotel level. The hotel_room_type store all the different prices for each hotel, for each room_type, so if we have a double room in hotel1 that costs 100$, and the same room_type in another hotel (hotel2) has a price of 150$, we select the foreign keys hotel_id, room_type_id and we set a different price for 2 different hotels by adding a foreign key to the table price. The table price contains the price valid for periods of time, like seasons, and another table that is connected to the hotel table modifies that price based on the availability of the room. We can check the availability of the room by counting the number of rooms available depending on the type and the period of the booking in the unavailable table. If a room has low availability in that season, the system automatically raises the price of the final booking by using a percentage. In the hotel_room_type we also store the description of the room_type, to have a different description for a different hotel that uses the same room_type. I decided to connect the room with the room_type and not hotel_room_type, because I need only the id of room_type, then I can retrieve the description or the price from the hotel_room_type.

Calculation of the final price, booking conclusion and ‘commit’

The total booking price is the sum of the calculated room price, additional services and the flight price. In my system I decided to take the price of the flight from the flight company database, with an API in real time, so we do not have to calculate this price. We do not want to repeat for the flights the same concept used for the rooms, and it is probably not how the website works. Probably the website uses external websites for the room price, but we are building it anyway to demonstrate how this system could work. For the availability as we said before we check the unavailable table. This concept reorganizes our system in a harmonious way, because it allows us to set every time we book a particular room, the start and the end date when that room is not available. In my version of the database, if the user completes the booking, a transaction in SQL inserts (commits) the data only if all the queries inside the transaction block have success. In this case we add a row in the unavailable table and at the same time we add another row in the booking (with the calculated paid price), flight and booking_product table. Once the transaction is complete the database saves the data in memory. This concept is used especially in transactions with money. For example, in the database of a bank we never want to use auto-commit in our database to update our bank account after a transaction, otherwise if the server runs out of power before the second line is committed, we end up with a loss of money. So we need to rollback to our previous state in case of a failure. For our purposes we use it only to simulate the booking process. If during the booking process someone else has already booked before us the last room of that type in one of the days we needed during our payment process, the booking is reverted and the payment aborted. My system works as follows:
Let's say we have 5 rooms. When I have to check the availability for that type of room, I will also write a query to count all the rooms that are available on that specific date. First I will use a join to take all the rooms for that particular type and I'll use something similar to:

```
SELECT * FROM unavailable WHERE
        (unavailable_start_date BETWEEN @start_date_reservation AND @end_date_reservation) OR
        (unavailable_end_date BETWEEN @start_date_reservation AND @end_date_reservation);
```

to check if the room is available for the specific set of days by looking in the unavailable table. if the result of it is null, it means that the room is available. The next step is to confirm the transaction where I add a row in the unavailable table with the new reservation (add a row with start and end date to keep in memory the date when the room is not available anymore) and add also a row in the booking table etc. The system checks the dates for each room and takes the first available room of that type to allocate it to the user (for example room 27 if the other 26 rooms are already booked for that period). The rows inside the unavailability table represent the times the booking is not available. Every time a booking is complete we can delete the old data inside the database or leave it and schedule a clean-up every year to free up space in memory. The system uses this concept not only to check if the room is available, but also to understand what price to apply to the room. In fact, with the table price that we connected with a one-to-many relationship to the table hotel_room_type, and the table availability_price connected to the whole hotel, we are able to check which price to apply based on the availability_price_drop percentage that is applied to the whole hotel. For example, if we have set a 30% price drop for an availability between 60 and 80%, the price of that specific room type in that specific period of time will drop and will change the total price to pay. The table price contains different periods of time. For each period we have a different price and we are supposed to add it line by line every year (decimal(10,2)). So now we need only to retrieve the price based on the period we booked the room. To do so we check the period in the price table and we count all the days that coincide with the days we booked, and multiply the number of days for the price of that specific period. To do so we need to use the function DATEDIFF() to count all the days inside the start and end period and sum the daily price that corresponds to that period. In this way our system is completely autonomous without the need for loops. Check the code in the appendix for a better understanding of the process.

The availability_price table contains all the variations of the final price of the room, and it is applied for the whole hotel. If a room type is more available than before the price will be reduced based on this table. It is not mandatory to use this table and doesn't block the functionality of our system. It is up to the programmer if he needs to use it or not, otherwise he can just set the price without variation for the room. The variable used for the percentage is a double(5,5), in case we need more precision for the program that controls the database. For both the price table and the unavailable table, we should plan a clean-up after that period of time is concluded. To connect each booking with the room selected I decided to use only the unavailable table, where we store the booked room date_start and date_end. I connected the room with a one-to-many relationship with the booking table, no other bookings can have the same room, but more than one room can be allocated to the same booking by its id. In this way we have all the information about the rooms and also the history of previous bookings if we do not delete the previous records, everything connected with only one foreign key. In conclusion, I added another table to add other additional products in our booking (booking_product) to take a snapshot of the current price of the additional service or product. We are in control of the price for the rooms and hotels in our model but we cannot control the other services. For each additional product we have a description, name and price that represent a snapshot of the price paid at the time we booked it from the provider of the additional service. The booking_price_paid in the booking table is the sum of these products plus the price of the room/rooms and the flight/flights booked. I used https://www.mockaroo.com/ to help myself to fill the data needed to test my final version of the database. The image 3 below is my final ERD model

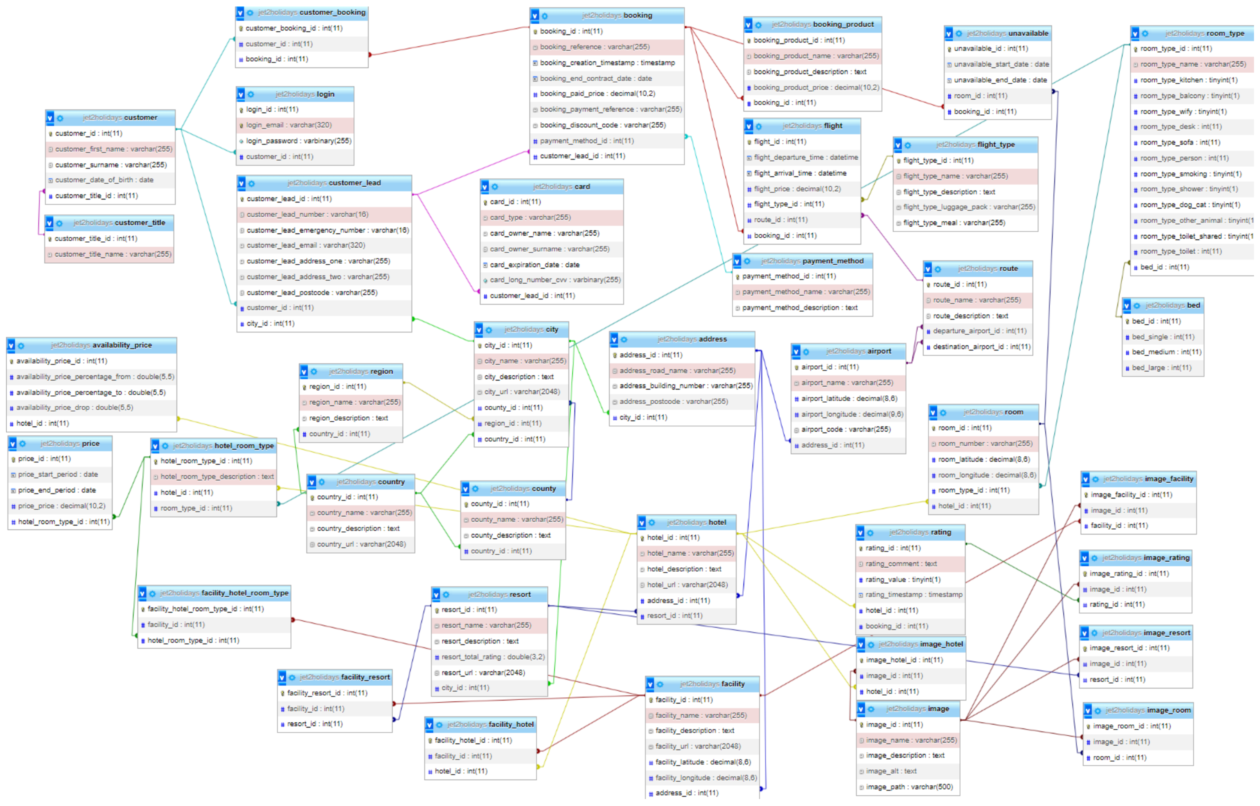

jet2holidays customer_booking
customer_booking_id : int(11)
customer_id : int(11)
booking_id : int(11)
jet2holidays booking
booking_id : int(11)
booking_reference : varchar(255)
booking_creation_timestamp : timestamp
booking_end_contract_date : date
booking_paid_price : decimal(10,2)
booking_payment_reference : varchar(255)
booking_discount_code : varchar(255)
payment_method_id : int(11)
customer_lead_id : int(11)
jet2holidays booking_product
booking_product_id : int(11)
booking_product_name : varchar(255)
booking_product_description : text
booking_product_price : decimal(10,2)
booking_id : int(11)
jet2holidays unavailable
unavailable_id : int(11)
unavailable_start_date : date
unavailable_end_date : date
room_id : int(11)
booking_id : int(11)
jet2holidays room_type
room_type_id : int(11)
room_type_name : varchar(255)
room_type_kitchen : tinyint(1)
room_type_balcony : tinyint(1)
room_type_wify : tinyint(1)
room_type_desk : int(11)
room_type_sofa : int(11)
room_type_person : int(11)
room_type_smoking : tinyint(1)
room_type_shower : tinyint(1)
room_type_dog_cat : tinyint(1)
room_type_other_animal : tinyint(1)
room_type_toilet_shared : tinyint(1)
room_type_toilet : int(11)
bed_id : int(11)
jet2holidays customer
customer_id : int(11)
customer_first_name : varchar(255)
customer_surname : varchar(255)
customer_date_of_birth : date
customer_title_id : int(11)
jet2holidays customer_title
customer_title_id : int(11)
customer_title_name : varchar(255)
jet2holidays login
login_id : int(11)
login_email : varchar(320)
login_password : varbinary(255)
customer_id : int(11)
jet2holidays customer_lead
customer_lead_id : int(11)
customer_lead_number : varchar(16)
customer_lead_emergency_number : varchar(16)
customer_lead_email : varchar(320)
customer_lead_address_one : varchar(255)
customer_lead_address_two : varchar(255)
customer_lead_postcode : varchar(255)
customer_id : int(11)
city_id : int(11)
jet2holidays card
card_id : int(11)
card_type : varchar(255)
card_owner_name : varchar(255)
card_owner_surname : varchar(255)
card_expiration_date : date
card_long_number_cvv : varbinary(255)
customer_lead_id : int(11)
jet2holidays flight
flight_id : int(11)
flight_departure_time : datetime
flight_arrival_time : datetime
flight_price : decimal(10,2)
flight_type_id : int(11)
route_id : int(11)
booking_id : int(11)
jet2holidays flight_type
flight_type_id : int(11)
flight_type_name : varchar(255)
flight_type_description : text
flight_type_luggage_pack : varchar(255)
flight_type_meal : varchar(255)
jet2holidays payment_method
payment_method_id : int(11)
payment_method_name : varchar(255)
payment_method_description : text
jet2holidays route
route_id : int(11)
route_name : varchar(255)
route_description : text
departure_airport_id : int(11)
destination_airport_id : int(11)
jet2holidays bed
bed_id : int(11)
bed_single : int(11)
bed_medium : int(11)
bed_large : int(11)
jet2holidays availability_price
availability_price_id : int(11)
availability_price_percentage_from : double(5,5)
availability_price_percentage_to : double(5,5)
availability_price_drop : double(5,5)
hotel_id : int(11)
jet2holidays region
region_id : int(11)
region_name : varchar(255)
region_description : text
country_id : int(11)
jet2holidays city
city_id : int(11)
city_name : varchar(255)
city_description : text
city_url : varchar(2048)
county_id : int(11)
region_id : int(11)
country_id : int(11)
jet2holidays address
address_id : int(11)
address_road_name : varchar(255)
address_building_number : varchar(255)
address_postcode : varchar(255)
city_id : int(11)
jet2holidays airport
airport_id : int(11)
airport_name : varchar(255)
airport_latitude : decimal(8,6)
airport_longitude : decimal(9,6)
airport_code : varchar(255)
address_id : int(11)
jet2holidays room
room_id : int(11)
room_number : varchar(255)
room_latitude : decimal(8,6)
room_longitude : decimal(8,6)
room_type_id : int(11)
hotel_id : int(11)
jet2holidays price
price_id : int(11)
price_start_period : date
price_end_period : date
price_price : decimal(10,2)
hotel_room_type_id : int(11)
jet2holidays hotel_room_type
hotel_room_type_id : int(11)
hotel_room_type_description : text
hotel_id : int(11)
room_type_id : int(11)
jet2holidays country
country_id : int(11)
country_name : varchar(255)
country_description : text
country_url : varchar(2048)
jet2holidays county
county_id : int(11)
county_name : varchar(255)
county_description : text
country_id : int(11)
jet2holidays hotel
hotel_id : int(11)
hotel_name : varchar(255)
hotel_description : text
hotel_url : varchar(2048)
address_id : int(11)
resort_id : int(11)
jet2holidays rating
rating_id : int(11)
rating_comment : text
rating_value : tinyint(1)
rating_timestamp : timestamp
hotel_id : int(11)
booking_id : int(11)
jet2holidays image_facility
image_facility_id : int(11)
image_id : int(11)
facility_id : int(11)
jet2holidays image_rating
image_rating_id : int(11)
image_id : int(11)
rating_id : int(11)
jet2holidays image_resort
image_resort_id : int(11)
image_id : int(11)
resort_id : int(11)
jet2holidays image_room
image_room_id : int(11)
image_id : int(11)
room_id : int(11)
jet2holidays facility_hotel_room_type
facility_hotel_room_type_id : int(11)
facility_id : int(11)
hotel_room_type_id : int(11)
jet2holidays resort
resort_id : int(11)
resort_name : varchar(255)
resort_description : text
resort_total_rating : double(3,2)
resort_url : varchar(2048)
city_id : int(11)
jet2holidays image_hotel
image_hotel_id : int(11)
image_id : int(11)
hotel_id : int(11)
jet2holidays facility_resort
facility_resort_id : int(11)
facility_id : int(11)
resort_id : int(11)
jet2holidays facility_hotel
facility_hotel_id : int(11)
facility_id : int(11)
hotel_id : int(11)
jet2holidays facility
facility_id : int(11)
facility_name : varchar(255)
facility_description : text
facility_url : varchar(2048)
facility_latitude : decimal(8,6)
facility_longitude : decimal(8,6)
address_id : int(11)
jet2holidays image
image_id : int(11)
image_name : varchar(255)
image_description : text
image_alt : text
image_path : varchar(500)

Image 3 - final version of the ERD and the database

APPENDIX

Code for the booking process:

```sql
-- now we set the autocommit off with the transaction. in case we have an error the transaction will fail and the data rolled back to the previous status
START TRANSACTION;
    SET @flightPriceOne = 210;
    SET @flightPriceTwo = 150;
    SET @busPrice = 34;
    SET @start_date_reservation = '2023-01-16';
    SET @end_date_reservation = '2023-02-14';
    SET @bookingRoomNumber = 1;
    SET @hotelNumber = 1;

    -- select all the bookings for room number 2
    SELECT * FROM unavailable WHERE room_id = @bookingRoomNumber;

    -- select all the bookings that collide with the date of our booking. if the rows are empty it means that we can book that room, otherwise we should select a different room
    SELECT unavailable_id FROM unavailable WHERE
        (unavailable_start_date BETWEEN @start_date_reservation AND @end_date_reservation) OR
        (unavailable_end_date BETWEEN @start_date_reservation AND @end_date_reservation);
    -- if the columns are empty we can continue the booking process, otherwise we should raise an error and decline the transaction, we don't modify the data in the database if an error is raised.

    -- check the price of the room:
    -- select the correct room_type_id of that room
    SELECT @roomType := room_type_id FROM room WHERE hotel_id = @hotelNumber AND room_id = @bookingRoomNumber;

    SELECT * FROM hotel_room_type;

    -- take the id to find the price
    SELECT @hotelRoomType := hotel_room_type_id FROM hotel_room_type WHERE hotel_id = @hotelNumber AND room_type_id =
@roomType;

    -- find all the prices for the room we have chosen
    SELECT * FROM price WHERE hotel_room_type_id = @hotelRoomType;

    -- find all the prices for that room in the period we chosen
    SELECT * FROM price WHERE hotel_room_type_id = @hotelRoomType AND (
        (price_start_period BETWEEN @start_date_reservation AND @end_date_reservation) OR
        (price_end_period BETWEEN @start_date_reservation AND @end_date_reservation)
    );

    -- now we need to count all the days between the 2 different interval of dates and multiply all the different prices
    -- first let's sum up all the periods in between our dates. We use DATEDIFF to count the number of days between two dates
    SELECT SUM(price_price * DATEDIFF(price_end_period, price_start_period)) FROM price WHERE hotel_room_type_id =
@hotelRoomType AND (
        (price_start_period BETWEEN @start_date_reservation AND @end_date_reservation) OR
        (price_end_period BETWEEN @start_date_reservation AND @end_date_reservation)
    );

    -- we should change the condition to OR in AND otherwise we count the price for the whole period. we added IFNULL function to assign 0 if the result is NULL, we need it for the sum
    SELECT @priceStartPeriodBetween := IFNULL(SUM(price_price * DATEDIFF(price_end_period, price_start_period)), 0) FROM
```

```sql
        price WHERE hotel_room_type_id = @hotelRoomType AND (
            (price_start_period BETWEEN @start_date_reservation AND @end_date_reservation) AND
            (price_end_period BETWEEN @start_date_reservation AND @end_date_reservation)
    );

    -- now let's sum first from the beginning of the period and then to the end with the price changed
    SELECT @priceStartPeriodLeft := IFNULL(SUM(price_price * DATEDIFF(price_end_period, @start_date_reservation)), 0) FROM
        price WHERE hotel_room_type_id = @hotelRoomType AND (
            (price_start_period NOT BETWEEN @start_date_reservation AND @end_date_reservation) AND
            (price_end_period BETWEEN @start_date_reservation AND @end_date_reservation)
    );

    SELECT @priceStartPeriodRight := IFNULL(SUM(price_price * DATEDIFF(@end_date_reservation, price_start_period)), 0) FROM
        price WHERE hotel_room_type_id = @hotelRoomType AND (
            (price_start_period BETWEEN @start_date_reservation AND @end_date_reservation) AND
            (price_end_period NOT BETWEEN @start_date_reservation AND @end_date_reservation)
    );
    -- with the NOT BETWEEN we collect only the portion of data that we need for the sum. If we have more than one period with
a different price this system returns the correct total price.

    -- return the total price of the room or rooms
    SELECT @priceRoom := @priceStartPeriodBetween + @priceStartPeriodLeft + @priceStartPeriodRight;

    -- return the total price of the booking
    SELECT @priceBooking := @priceRoom + @flightPriceOne + @flightPriceTwo + @busPrice;


    -- SAVEPOINT savepoint1;

    -- insert the data in the tables or replace them
    SET @bookingId = 3;

    -- insert the booking in the booking table
    REPLACE INTO `booking` (`booking_id`, `booking_reference`, `booking_creation_timestamp`, `booking_end_contract_date`,
`booking_paid_price`, `booking_payment_reference`, `booking_discount_code`, `payment_method_id`, `customer_lead_id`) VALUES
        (@bookingId, 'DGWHW3246', current_timestamp(), @start_date_reservation, @priceBooking, 'si3bfu579dc', '263577', '2',
'1');

    -- insert the booking of the room in the unavailable table
    INSERT INTO `unavailable` (`unavailable_id`, `unavailable_start_date`, `unavailable_end_date`, `room_id`, `booking_id`)
VALUES
        (NULL, @start_date_reservation, @end_date_reservation, @bookingRoomNumber, @bookingId);

    -- insert the booking of the flight in the flight table. we use concat to concatenate the date with the time
    INSERT INTO `flight` (`flight_id`, `flight_departure_time`, `flight_arrival_time`, `flight_price`, `flight_type_id`,
`route_id`, `booking_id`) VALUES
        (NULL, CONCAT(@start_date_reservation, ' 06:33:00'), CONCAT(@start_date_reservation, ' 10:06:00'), @flightPriceOne,
'1', '3', @bookingId),
        (NULL, CONCAT(@end_date_reservation, ' 13:30:00'), CONCAT(@end_date_reservation, ' 16:03:00'), @flightPriceTwo, '1',
'4', @bookingId);

    -- insert the additional product booking in the booking_product table
    INSERT INTO `booking_product` (`booking_product_id`, `booking_product_name`, `booking_product_description`,
`booking_product_price`, `booking_id`) VALUES
        (NULL, 'Bus transport', 'Departure from the entrance of the airport direct to Catania', @busPrice, @bookingId);
    -- ROLLBACK TO savepoint1;
COMMIT;
```

Code for the filter function for departure and destination:

```sql
-- filter function for departure cities
SET @countryName = 'United Kingdom';

SELECT * FROM country;

SELECT country_id FROM country WHERE country_name = @countryName;

SELECT * FROM city;

-- subqueries
SELECT * FROM city WHERE country_id IN (
    SELECT country_id FROM country WHERE country_name = @countryName
);

-- subqueries
SELECT city_name FROM city WHERE country_id IN (
    SELECT country_id FROM country WHERE country_name = @countryName
);
```

```sql
-- filter function for destination country, city, resort and hotels that have a particular facility and room type
SET @countryName = 'Italy';
SET @cityName = 'Catania';
SET @resortName = 'Sicily Country House & Beach';
SET @hotelName = 'Hotel & Residence Costa del Sol';
SET @facilityName = 'Swimming pool';
SET @roomType = 'Medium room for 3 people, animals allowed';

SELECT @countryId := country_id FROM country WHERE country_name = @countryName;

SELECT @cityId := city_id FROM city WHERE country_id = @countryId AND city_name = @cityName;

-- inner join
SELECT * FROM resort INNER JOIN city
    ON resort.city_id = city.city_id WHERE
    resort.city_id = @cityId;

-- multiple join
-- check if all the resorts have hotels in that city
SELECT
    city_name AS Cities,
    resort_name AS Resorts,
    hotel_name AS Hotels,
    hotel_description AS 'Hotel Description',
    hotel_url AS URL
FROM resort INNER JOIN city ON
    resort.city_id = city.city_id
LEFT JOIN hotel ON
    resort.resort_id = hotel.resort_id WHERE
    resort.city_id = @cityId;
```

```sql
-- multiple join
-- check if all the hotels are connected to a resort in that city. if not, the resort columns should be NULL
SELECT
    resort_name AS Resorts,
    hotel_name AS Hotels,
    hotel_description AS 'Hotel Description',
    hotel_url AS URL
FROM resort INNER JOIN city ON
    resort.city_id = city.city_id
RIGHT JOIN hotel ON
    resort.resort_id = hotel.resort_id WHERE
    address_id IN (SELECT address_id FROM address WHERE city_id = @cityId);

-- show all the hotels with a particular facility
SELECT * FROM hotel WHERE hotel_id IN (
    SELECT hotel_id FROM facility_hotel WHERE facility_id IN (
        SELECT facility_id FROM facility WHERE facility_name = @facilityName
    )
);

-- show all the rooms of a particular type
SELECT * FROM room WHERE room_type_id IN (
    SELECT room_type_id FROM room_type WHERE room_type_name = @roomType
);

-- shows all the hotel name, description and resort name of the hotels that provide that particular facility and that
            particular room type in the chosen resort
SELECT
    city_name AS Cities,
    resort_name AS Resorts,
    hotel_name AS Hotels,
    hotel_description AS 'Hotel Description',
    hotel_url AS URL
FROM resort INNER JOIN city ON
    resort.city_id = city.city_id
RIGHT JOIN hotel ON
    resort.resort_id = hotel.resort_id WHERE
    address_id IN (SELECT address_id FROM address WHERE city_id = @cityId) AND
    hotel_id IN (
        SELECT hotel_id FROM hotel WHERE hotel_id IN (
            SELECT hotel_id FROM facility_hotel WHERE facility_id IN (
                SELECT facility_id FROM facility WHERE facility_name = @facilityName
            )
        )
    ) AND hotel_id IN (
        SELECT hotel_id FROM room WHERE room_type_id IN (
            SELECT room_type_id FROM room_type WHERE room_type_name = 'Medium room for 3 people, animals allowed'
    ) AND resort_name = @resortName
);
```

Code for storing the card details of the user and retrieve them:

```sql
SET @cardType = 'Visa';
SET @cardOwnerName = 'Roberto';
SET @cardOwnerSurname = 'Lo Duca';
SET @cardLongNumber = '5274 3258 9126 1322';
SET @cardCVV = '513';
SET @cardExpirationDate = '2024-10-01';
SET @customerLead = '1';

SET @password = 'MK4o4kqd1%ctvmmy1nhvgK$8BN8L%G';

-- we never store the cvv or card long number in our system because it's easy for a hacker to steal it. We do it only to
demonstrate a way to encrypt data. It is advisable to have this work done by an external service
-- concatenate card long number and cvv, then encrypt it with a password
SET @cardLongNumberAndCVV = AES_ENCRYPT(CONCAT(@cardLongNumber, @cardCVV), @password);

-- insert the results
INSERT INTO `card` (`card_id`, `card_type`, `card_owner_name`, `card_owner_surname`, `card_expiration_date`,
`card_long_number_cvv`, `customer_lead_id`) VALUES
    (NULL, @cardType, @cardOwnerName, @cardOwnerSurname, @cardExpirationDate, @cardLongNumberAndCVV, @customerLead);

-- show the table with the values decrypted. we use SUBSTRING function to divide the card long number from the cvv
SELECT
    `card_id`,
    `card_type`,
    `card_owner_name`,
    `card_owner_surname`,
    `card_expiration_date`,
    SUBSTRING(AES_DECRYPT(`card_long_number_cvv`, @password), 1, 19) AS 'Card Long Number',
    SUBSTRING(AES_DECRYPT(`card_long_number_cvv`, @password), 20, 22) AS CVV,
    `customer_lead_id`
FROM card WHERE customer_lead_id = @customerLead
```

Code for registration and login process of the user:

```sql
-- we have setted the index for the email in UNIQUE, so we cannot register with the same email 2 times, otherwise we get an
error
SET @email = 'neutronixyz@gmail.com';
SET @password = 'elGC%8*MED9ObHD01ImAgZBHD0%J';
SET @customer = '1';

-- again, better to use external and updated services for this job, there are always more updated methods and a hacker can have
access to your password and log in other websites if you used the same password
-- always check if the website use the https secured and a SSL (Secure Sockets Layer) before type your password on it
-- first we generate randomly the salt because we need it to make the one-way hashing impossible to reverse
SELECT @salt := SUBSTRING(SHA1(RAND()), 1, 6);

-- use SHA256 that has a 160 bit long key to encrypt data. Concatenate the password with the salt to make a salted hash
SELECT @saltedHash := SHA2(CONCAT(@salt, @password), 256);

-- concatenate the salt and the salted hash together to store the data needed for the log in
SELECT @saltSaltedHash := CONCAT(@salt, @saltedHash);

-- store the data encrypted in the database
INSERT INTO `login` (`login_id`, `login_email`, `login_password`, `customer_id`) VALUES
    (NULL, @email, @saltSaltedHash, @customer);
```

```sql
-- insert login email and password, they need both to be correct
SET @loginEemail = 'neutronixyz@gmail.com';
SET @loginPassword = 'elGC%8*MED9ObHD01ImAgZBHD0%J';

-- first we need to take the salt concatenated before from the database related to our email. if the email is wrong the salt
will be probably NULL
-- we need the same salt used in the registration process to allow us to compare the hash generated before with the new hash.
-- We can generate the same salted hash only if we type the same password. In this way our password will be never stored and we
compare only the salted hash value generate with it
SELECT @salt := SUBSTRING(login_password, 1, 6) FROM login WHERE login_email = @loginEemail;

-- let's generate the salted hash with the same salt value but with the login password. we should use the same hashing function
to make it work, in this case SHA256
SELECT @saltedHash := SHA2(CONCAT(@salt, @loginPassword), 256);

-- now concatenate again the salt before the comparison. we could also compare only the salted hash but for practical reasons
we prefer to compare salt + salted hash together
SELECT @saltSaltedHash := CONCAT(@salt, @saltedHash);

-- compare the result with the stored value. if the selection is NULL it means that the password or the email are wrong
SELECT * FROM login WHERE login_password = @saltSaltedHash AND login_email = @loginEemail
```