|  |
| --- |
|  |
| ***Group 8 Report***  ***CIS016-1 Assignment 3 - Hotel Booking System*** |
| |  |  | | --- | --- | |  | CIS016-1 Assignment 3 2016/2017 | |

Contents

[List of Group Members 2](#_Toc481708422)

[1. Introduction 2](#_Toc481708423)

[2. System Design 3](#_Toc481708424)

[2.1. UML Use Case Diagram 3](#_Toc481708425)

[2.2. UML Class Diagram 4](#_Toc481708426)

[2.3. Database Design 5](#_Toc481708427)

[3. Implementation 7](#_Toc481708428)

[4. Testing 13](#_Toc481708429)

[5. Reflection and Responsibilities 15](#_Toc481708430)

[6. Conclusion 16](#_Toc481708431)

[7. References and Resources 16](#_Toc481708432)

[8. Appendix 17](#_Toc481708433)

[8.1. Appendix A 17](#_Toc481708434)

[8.2. Appendix B 19](#_Toc481708435)

[8.3. Appendix C 30](#_Toc481708436)

[8.4. Appendix D 31](#_Toc481708437)

[8.5 Appendix E 48](#_Toc481708438)

List of Group Members

# Introduction

In this report we will discuss the solution that we produced for the given scenario. The scenario required us to consider a Hotel Booking System, which was to have a Graphical User Interface for the users to use and a database for data to be stored even after the program has been closed. In the task brief, two different types of users could be identified, the customers and the hotel manager, which we refer to as the admin in our program. These users both needed to be able to log in, which meant login credentials had to be stored, and the customers needed to be able to register to use the program, meaning they had to provide us with some details which could be stored in the database. As the task outlines, registered customers can book rooms in the hotel and look at bookings they had already made, as well as deleting and or changing existing bookings. To book a room, a customer has to provide us with the room type they require and the dates they need it for. Room bookings are also to be stored in the database, and bookings should not be allowed if there is not enough space in the hotel for the requested dates. Once they are logged in, the hotel manager will see a different screen to the one seen by a customer, and they will be able to view the bookings per room and assign the bookings made by customers to a room. For our system, we opted to use a fictional hotel called ‘Group 8 Hotel’, which had 3 floors with 8 rooms per floor, 4 of which are doubles and 4 of which are singles. The task description suggested the use of a client-server architecture, which we have implemented within our system.

1. System Design
   1. UML Use Case Diagram

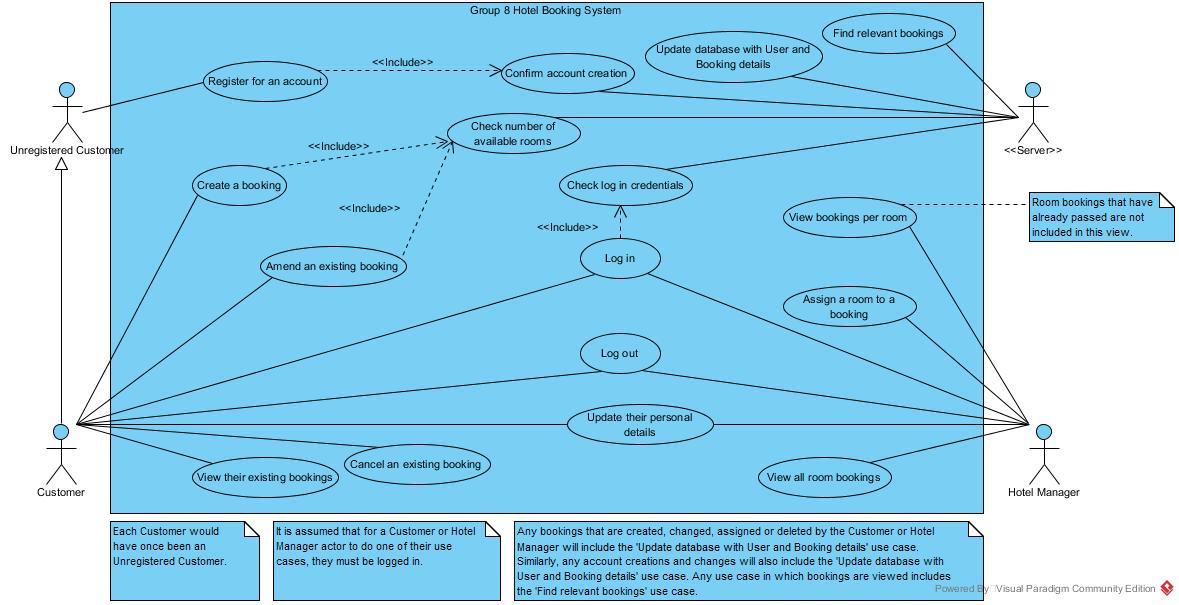


Figure 1: Use Case Diagram for the Group 8 Hotel Booking System, demonstrating the functionality and roles that are present within the system.

The use case diagram provided in Figure 1 demonstrates the core functionality of the program at the sea level, which is also known as the system level. This level of abstraction shows the main actors we have proposed in our system, the Hotel Manager (referred to in the Java code as Admin), the Customers, the Server (even though this is not a physical entity like the other actors listed, this actor does have its own roles within the system which are required to implement the functionality, and hence it deserves an actor in its own right) and finally the Unregistered Customer. A Customer can be seen as a derivative from the Unregistered Customer, since at some point in time, each Customer would have had to register, meaning that prior to that point, they were an Unregistered Customer. From the use case diagram, it can be seen that an Unregistered Customer can only register for an account, which the server must confirm, before they can become a Customer and use the system to its full functionality. The use case diagram highlights that the Customer creating and amending a booking is dependent on the Server checking the availability of rooms, and this is crucial for the system as it would be unrealistic and poor design if our hotel booking system allowed customers to book rooms when the hotel had no rooms available. The diagram also highlights the capability for both the Customer and Hotel Manager actors to log in and out of the system. The comments within the diagram outline how this is a necessity for the users to be able to carry out the rest of their use cases, which is one of the things that was required by the brief. The diagram also shows how the log in use case for the users relies on the Server actor checking the login credentials provided against those that exist in the database to allow the users to log in.

It may seem strange that the actors provided in the diagram does not include a Client actor, however we felt that this did not provide functionality in terms of the system, it instead provided a way of carrying out the functions outlined in the use cases.

* 1. UML Class Diagram

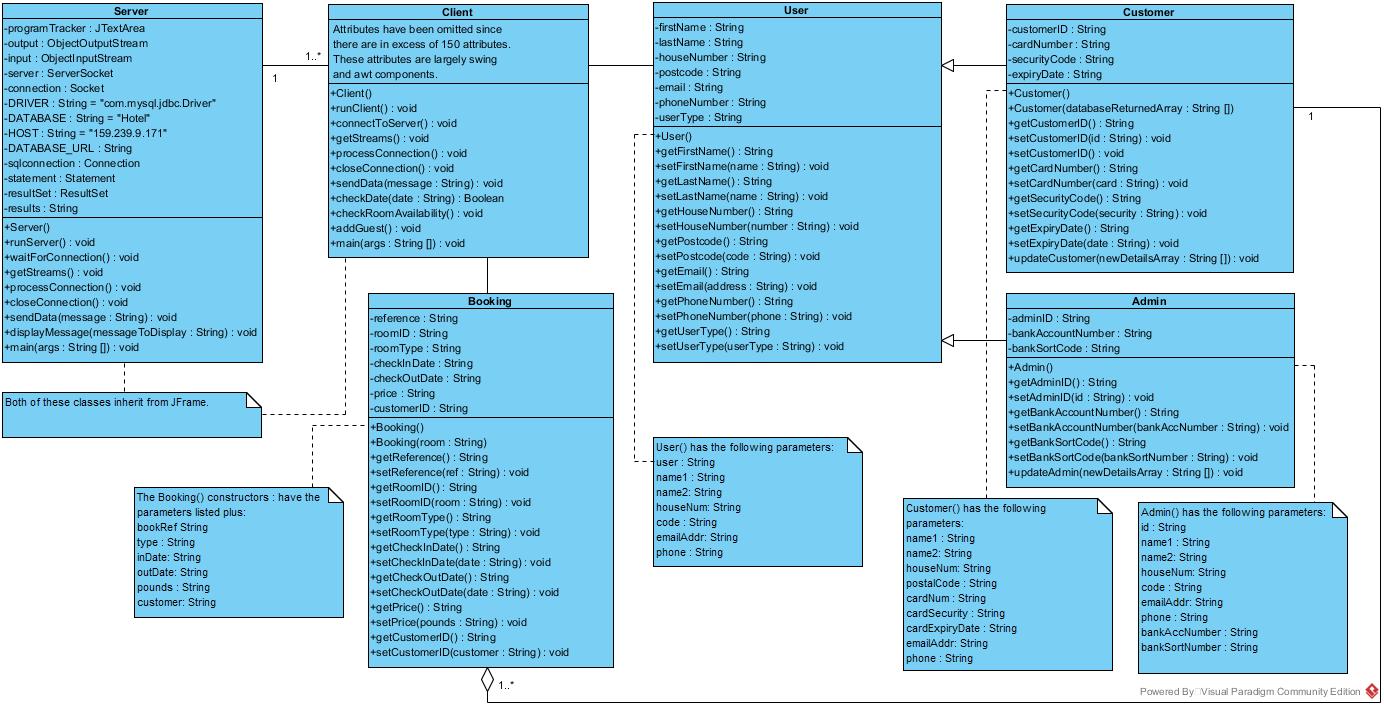


Figure 2: A class diagram which identifies all of the classes within the Group 8 Hotel Booking System and highlights the relationships that exist between those classes.

In the class diagram shown in Figure 2, we have highlighted the classes that exist in our program, and the relationships that exist between these classes. For each of the classes we have outlined the methods within that class, including the constructor(s) for those classes, and the attributes that every instance of these classes shall have (with the exception of the Client class, where a full list of attributes is provided in appendix A). Identifying the attributes and functions in this way allowed us to work on different parts of the project separately, whilst having an understanding of the end goal for that section of the code.

One of the relationships shown in the class diagram is the generalisation relationship that exists between both the Customer and the Admin classes and the User class. This is because there are many similar attributes and methods that both the Admin and Customer user types have in common, and so it makes sense for objects of these types to derive from a superclass called User, which contains all of the attributes and methods that it’s subclasses will have.

Another relationship that is highlighted in the class diagram is the aggregation relationship existing between the Customer class and the Booking class. We felt that this relationship is necessary since a Booking cannot exist without a Customer object being associated with it, hence you can say that a Customer, whilst not entirely, makes up a Booking and a Booking cannot exist without it. We decided not to use a composition relationship since the Customer object can exist independently of the Booking, and deleting the Booking will not delete the Customer, as the Customer can exist on its own without any bookings.

The remaining relationships shown within the diagram are simple association relationships. We decided to include these relationships since we felt that the classes work in conjunction with each other, and whilst they use each other, neither of the classes could be considered to be a part to the bigger whole which was the other class within the relationship, and hence felt that aggregation and composition relationships did not exist in these cases. We also felt that both of the classes within each relationship can exist in their own right, for example a Client object can exist fine prior to a User logging in, and similarly there is no reason to say that a User object can only exist if the Client does, it could be created elsewhere.

* 1. Database Design

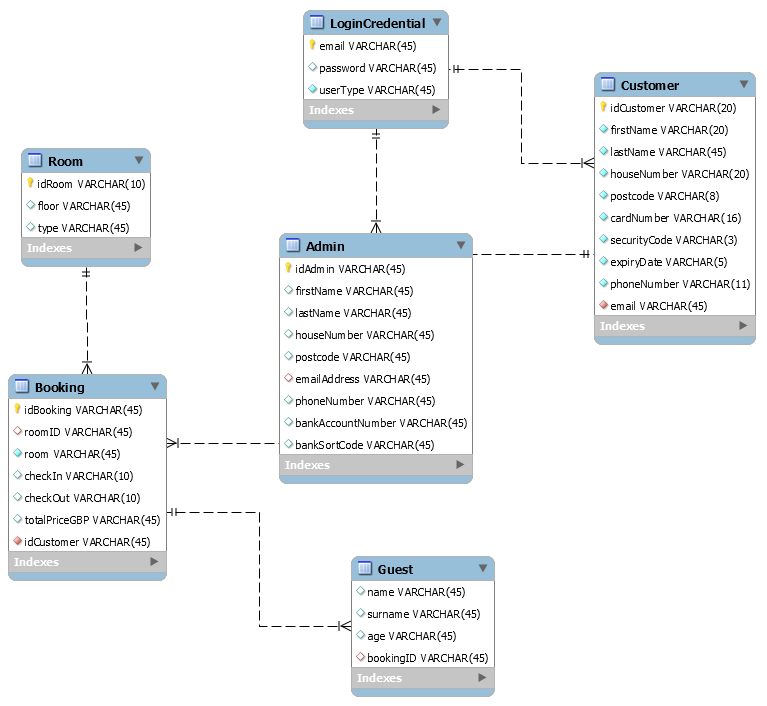


Figure 3: An entity-relationship model which shows the tables within our database, the relationships between tables and the columns within the tables.

In the figure above, an entity relationship model is provided for the database that we have used within our project. From the model, 6 tables can be identified, which contain within them all of the data points necessary to allow the system to run properly. Starting with the Booking table, you can see that within this table we store a data point that relates to each of the attributes within the Booking class outlined above. Essentially, every Booking object that is instantiated will translate to a row within this table – whenever a booking is made or changed, a Booking object is created and the values of that object’s attributes are written into the database. The primary key for the Booking table is the idBooking field, since this must be unique (it is generated and checked within our system) and is the field that will allow you to identify and select the correct booking. The table has two foreign keys, idCustomer which it references from the Customer table which allows us to identify which Customer it was that made the booking and idRoom which it references from the Room table, which allows us to ensure that the room that is assigned to a booking is one that exists within our hotel.

For the Room table, only three columns are needed, idRoom, which is the primary key and is the unique identifier which we can use to select any given room, along with floor and type which gives us some further details about the room. Within the Room table, one of the rows has a value for idRoom of ‘Unassigned’. Whilst this room is not a room that exists within our hotel, this ‘room’ had to be included within the table since this is the ‘room’ that is given to all bookings upon creation before the admin has gone into the system and decided which room should be allocated to which booking.

The Customer and Admin tables within our database are very similar, with only a couple of fields being different. This is because the information stored in these tables are the values of the attributes for the Customer and Admin objects that exist within our program respectively. Since both of the Customer and Admin classes derive from the User superclass, it will come as no surprise that many of the columns within the tables are the same. For each of the tables, a primary key exists. For Customer, that primary key is the idCustomer column, whilst for the Admin table it is the idAdmin table. These columns were used as the primary key since they are unique to each row in the table and will never change. Our system generates the values put in for these fields and ensures that the value does not exist prior to entering it in to the table and will generate a new value if necessary. The email associated with one of these Customer or Booking objects is used as the foreign key within each of the tables, represented by email in the Customer table and emailAddress in the Booking table. These columns are the foreign keys in each table since they reference the LoginCredential table, which stores every User’s user name and password.

The LoginCredential table contains three columns, email, password and user type, and it is this table that the system uses to identify whether a user can log in. The email is the primary key for this table, and whilst it is always unique (the register code will not allow the same email to be used for multiple accounts), unlike normal primary keys, this value can change as the user may forget the details for that email account or may simply need to change emails. There was a potential for problems to arise here in that if the User changes their email, we had to ensure that this change was reflected in the LoginCredential table and the Customer or Admin table depending on their user type. This was done by setting the action for the foreign keys email and emailAddress from the Customer and Admin tables respectively to on update cascade and on delete cascade, meaning changes only needed to be made in the referenced table for them to be reflected in the other table.

In the Guest table, if a booking was cancelled, and hence removed from the Booking table, we had to ensure that the rows in the Guest table that were associated with this Booking were also removed from the table. This was done by setting the on delete action for the bookingID column in Guest, which references the idBooking column from Booking to cascade on delete.

The aims of structuring our database in the way we have done was to make it easy for anyone to read, even someone who has no prior experience with our system or even databases. The relationships that exist between the tables in the forms of foreign keys has allowed us to store all of the information we wanted without duplicating data, which is something that we wanted to avoid since it is not good practice and we could have easily fell into the trap of changing the values in one location and not the other. Appendix B provides some examples of the data that is in the database. From these examples, you can see how each table links together and bookings/users can be traced throughout the different tables, without the need for writing the details for each user in various locations, thanks to the use of foreign keys.

1. Implementation

To develop the system, we decided to use Eclipse as our IDE. Despite this being our first time using Eclipse, we felt that it was worth our time in learning how to use it because Eclipse seemed to be more capable in handling large scale projects than BlueJ which we had been using previously. One of the key features that made Eclipse much easier to use is that we could run the Client and the Server classes simultaneously without having to open multiple Eclipse windows like we would have had to with BlueJ. Another useful thing with Eclipse is that it points out errors on the lines as you type and tells you where they exist within the code with suggestions on how to fix the problem, which is helpful in saving time when it comes to debugging the code.

The system we have produced uses a Client-Server system, which is also known as Sockets Programming. We decided to use this method of connecting with the database as we felt that it was more applicable for the real world, as the database can be changed without a bearing on the Client, there would be no reason for a front end user to even realise that the change had happened. Another reason that we felt this system architecture was good to use is that a user will not have any details stored locally on their computer and all of the data will be stored in a single, central location. Additionally, as Maryam Khan (2014) outlines, this approach would allow any back-ups to be taken in a single place, and allows for any system security to be implemented in a single place, both of which contribute to ways of minimising data loss. On a small scale project like ours, or a wider project in the real world, such an approach has allowed us to add and update the data that is in the database from different computers easily and conveniently, and has allowed us to export all of the data within the database from one place. Without this functionality, we would have struggled as each computer that we used to develop the code on would have ended up with different databases, something which is not ideal for a booking system. We do however recognize that there are some drawbacks to using a client-server architecture. For example, as Maryam Khan describes, the system entirely depends on the Server running, if the Server goes down, the rest of the System is rendered useless. This impacted on the development of our systems when our VPN to connect to the server (since it was hosted at work) failed on one of the days we were at University, meaning we wasted time. As we have mentioned, we decided to implement a database within our system. We felt that this was a good way of storing all of the details needed within our system in a manner that was manageable and easy to expand and update. By using a database, we found it easy to add new data points where necessary by simply adding new columns to existing tables or adding new tables all together. The database we used is on a Server at our workplace, and is a MySQL database. We opted to use a MySQL database since the syntax is very simple and easy to use, something which was very important to us given that we have no prior experience of working with databases. To manage and view the contents on the database we used a program called MySQL Workbench (available here: <https://www.mysql.com/products/workbench/> ) which allowed us to create and visualise the database in an easy manner. In order to connect to the database from within Java, we had to install a JDBC driver (which can be found here online at <https://breo.beds.ac.uk/webapps/blackboard/content/listContent.jsp?course_id=_61910465_1&content_id=_2671639_1> ) and install this into Eclipse.

For us to produce a solution to the task, there were some assumptions and decisions that had to be made by the group. One of the initial assumptions we made was that rooms were to be charged on a flat rate throughout the year depending on what room type they selected. We decided on charges of £80 for a double room per night and £60 for a single room per night. Another assumption we made was that the admin will only assign rooms to a booking on the day before arrival, and they will always do this without fail, in order to maximize the amount of time a customer has to change or cancel a booking, as a booking can only be amended or cancelled prior to being assigned to the room in our system, without contacting the admin directly. Within our system it is also assumed that whenever a booking is created, or any changes or cancellations are made to an existing booking, any payments or refunds will be handled automatically, and payment details do not need to be checked every time a booking is made, only upon account creation. We anticipate that an Admin would take a backup of the live database at the end of each working day in order to prevent data loss and allow a backup Server to be run if necessary, however this is beyond the scope of this project to implement such a feature. A final assumption that has been made is that a user cannot book a room in the hotel for a day, and must have an overnight stay, as the check-out time for our hotel is in the morning, and the check-in time is in the late afternoon. This assumption is what would happen in a real hotel, as a customer can vacate a room in the morning, and this room will become available to another customer later in that day.

The group agreed to use a Client-Server right from the start, and so the first steps were to set up a connection between the Client and the Server and then the Server and the Database. Setting up the connections were straightforward, however we had to consider ways of transferring data between the Client and the Server, so that the Client could send information to the Server which could be used to generate the correct SQL statement, and the Server could return information returned from the database. We decided to base our system on the chat server example provided here: <https://breo.beds.ac.uk/webapps/blackboard/content/listContent.jsp?course_id=_61910465_1&content_id=_2671653_1> , and we used strings to send ‘messages’ between the Client and Server which could be translated into something useful at the other end. The way we agreed on doing this was to have a string which identified the kind of message being sent, for example “Login”, which was followed by a ‘~’. Following the ‘~’ symbol, any additional information could be provided, such as constraints needed for an SQL statement, as highlighted in the code snippet below. Each of the pieces of additional information was separated by a ‘~’ symbol, meaning that when a message was received by the Client or Server, the program could split the message into the array and do the desired computation. Originally, we had been using a hyphen to separate the different parts of the string, however we had to change from this as the date formats in Java and SQL rely on a hyphen separating the different parts of the date, meaning that when we were splitting a received message, we were splitting up any dates that had been sent across too which was undesired.

String messageAction **=** "Login~" **+** user **+** "~" **+** pass**;**

sendData**(**messageAction**);**

An example of a string being sent from the Client to the Server is provided in the code snippet above. This code passes the string which is to be sent across to the sendData method, which writes the string to the Server using an ObjectOutputStream. This method also exists within the Server, to send data to the Client. We decided to use this way of exchanging data between the Client and Server as we felt it was an easy to understand method, and anyone reading the code could probably understand it, even without prior knowledge of network programming, and it made it very easy to handle the different types of message that were sent and received.

The strings that were exchanged between the Client and the Server are split into arrays, as mentioned above. The system identifies what type of commands are expected to be executed by using a switch statement on the first item within the array of the split message. The code below shows how the message string is read from Client using the ObjectInputStream, and is then split into an array of smaller strings using the predefined split function. This code exists within the processConnection method, which contains a do-while loop which continues to run until the Server sends “---“ as a message, which causes the loop to end and eventually results in the program terminating. The element at the first index position in the array, that is position 0, is assigned the identifier command, which is used as the condition for the switch statement. Each of the cases below has had the code removed from the blocks in order to save space, however the full code can be found in Appendix C part A. It can be seen that we have identified multiple different cases, each of which can be used to do what the Client requested, which normally involves generating and executing an SQL statement, and sending the results back in a String format.

**try**

**{**

message **=** **(**String**)** input**.**readObject**();**

**this.**displayMessage**(** "\n" **+** message **);**

String **[]** splitMessage **=** message**.**split**(**"~"**);**

command **=** splitMessage**[**0**];**

System**.**out**.**println**(**command**);**

**try**

**{**

statement **=** sqlconnection**.**createStatement**();**

**switch** **(**command**)**

**{**

**case** ""**:**

**...**

**break;**

**case** "Login"**:**

**...**

**break;**

**case** "Register"**:**

**...**

**break;**

**case** "Create"**:**

**...**

**break;**

**case** "View"**:**

**...**

**break;**

**case** "ViewBookingIDOnly"**:**

**...**

**break;**

**case** "DetailsAboutThisBooking"**:**

**...**

**break;**

**case** "DeleteThisBooking"**:**

**...**

**break;**

**case** "AdminView"**:**

**...**

**break;**

**case** "Amend"**:**

**...**

**break;**

**case** "CompileListOfUnassignedBookings"**:**

**...**

**break;**

**case** "RoomTypeRequired"**:**

**...**

**break;**

**case** "CheckRoomAvailability"**:**

**...**

**break;**

**case** "CheckHowManyRooms"**:**

**...**

**break;**

**case** "Assign"**:**

**...**

**break;**

**case** "UpdateDetails"**:**

**...**

**break;**

**case** "UpdateBooking"**:**

**...**

**break;**

**case** "FloorRoomTypes"**:**

**...**

**break;**

**case** "GetAllBookingsForRoom"**:**

**...**

**break;**

**case** "GetUsersPassword"**:**

**...**

**break;**

**case** "UpdateAdminDetails"**:**

**...**

**break;**

**case** "AddGuest"**:**

**...**

**break;**

**case** "Check"**:**

**...**

**break;**

**}**

**}**

**catch** **(**SQLException sql**)**

**{**

sql**.**printStackTrace**();**

**}**

**catch** **(**ParseException pe**)**

**{**

pe**.**printStackTrace**();**

**}**

**}**

**catch** **(**ClassNotFoundException classNotFoundException **)**

**{**

displayMessage**(** "\nUnknown object type received" **);**

**}**

**catch(**NullPointerException npe**)**

**{**

npe**.**printStackTrace**();**

**}**

**catch(**SocketException socketExc**)**

**{**

**this.**closeConnection**();**

message **=** "CLIENT>>> TERMINATE"**;**

**}**

**}**

Similar code exists in the Client, where the message that is split means the flow of control is diverted to the relevant case. The cases in the switch statement for the Client side are statements that act in response to the results that have been retrieved from the database by the Server and returned to the Client. These responses can do a variety of things, from changing the appearance of the GUI, allowing or preventing a user from logging in and allowing or preventing a customer from creating/changing a booking.

One of the most crucial pieces of code to our program was the code which checked what rooms were available to be booked. For this, we identified two scenarios, one which was to calculate the number of rooms that were booked for a particular date, and another which was to identify which rooms were booked for a particular date. The system needs to know how many rooms are booked for each date to ensure that all of the rooms are not booked when a customer makes a booking or changes the dates of an existing one. This code was not negotiable for us as it is important that the system does not take more bookings than the hotel can handle, as in a real life application, the company would end up with a lot of unhappy customers. The code for finding out the number of rooms booked is provided in appendix C part B.

The code takes in the second and third parts of the string which are separated by the ‘~’ character and as the system knows, these will be dates, it changes the string into the desired format, which is yyyy-MM-dd, (y is year, M is months and d is day). The program then creates an empty array list, which will store the dates the customer wants to stay at the hotel for, including the check in and check out dates. Using the SimpleDateFormat and Calendar classes from the java.text and java.util libraries respectively, the check in date found within the string that was sent to the Server can be used to set the time of the holidayRequestedCalendar Calendar object. A while loop is then used to increment the date by a day using the add method from Calendar, until the string object that is retrieved from the Calendar’s date matches the string which represents the check out date. At this point, the program now has an array list full of the dates that the customer will be staying in the hotel. Two for loops then select the roomID’s from the Booking table for each of the dates in the array list that corresponds with a check in in the table. One of the for loops handles the entries where the room column is ‘Double’ and the other ‘Single’. The for loops put their results into an array list which relates to the room type that the loop was looking at. The size of this array list, obtained using the size method, is subtracted from 12, to give the number of rooms left available within the hotel of a given type (we have 12 rooms of each type in our hotel). These values are then concatenated with a string and the ‘~’ characters separating the values which is returned to the Client, allowing the Client to respond in the appropriate manner and allow the customer to make their booking if there are enough rooms available.

However, during final testing we have noticed a limitation of this code, in that all room bookings may not be picked up, and so there is the potential that the hotel could over book. This is due to the fact that in the database design, we have only included the check in and check out dates for the bookings, and at no point have we calculated the days in between. This means that a booking that is checked in before the requested date that we are checking (and could check out after the requested check out date), would be occupying a room, but this would not be picked up by our system.

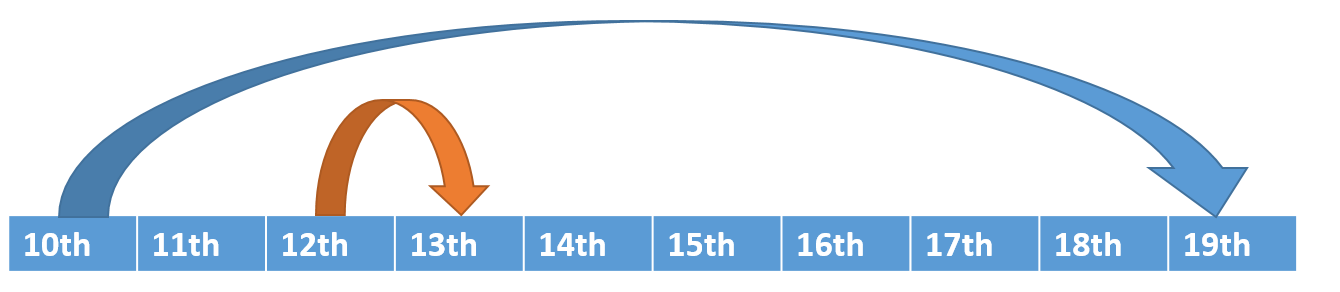


Figure 4: Visual example of holiday bookings.

If you consider the example in Figure 4, when checking the number of rooms available for the booking represented by the arrow in orange, the booking represented by the arrow in blue would not be detected, and therefore an extra room would appear to be available. This is due to the fact that in the database and within the program, there are no points where these bookings overlap, meaning the SQL statement would not detect them. The solution that we proposed to this problem, but unfortunately did not have time to implement, is that the array list containing all of the dates that are going to be booked is created, as we have already done. Then following this, every booking where room = ‘double’ is selected from the Booking table, and a similar array list is created for the dates contained within these bookings. The program could then compare each of the dates in the requested dates array list against each of the values in each of the array lists generated from the bookings in the database. A counting variable could be used to find the number of lists where a match is found, and hence the number of booked rooms for the requested dates can be deduced. This process could then be repeated for the bookings where room = ‘Single’, before putting a string message together containing the results that can be sent to the Client. When creating a booking, this checking of the room availability is invoked using a ‘Check Availability’ button, and if there are rooms available, the button that allows the user to proceed with the booking becomes enabled. This button becomes disabled again after 5 seconds however, and the customer would have to check the availability of rooms again if they wanted to proceed with the booking. This is because in the time between checking the availability and proceeding with the booking, another customer could have created a booking which took up the final available room. We felt that 5 seconds was a long enough comprise in giving the user chance to click proceed and ensuring that the rooms they were about to book hadn’t been taken in the meantime.

The other scenario was the one where the program needed to know which rooms were booked, in order for the Hotel Manager to put rooms to a booking. The brief outlined how the Manager, also known as the Admin, needed to be able to view the bookings per room, which we have implemented through a JTable in a JOptionPane dialog box, in order to be able to assign each booking a room. Within our project however, we made this process much easier, by only allowing the Hotel Manager to select the rooms that are available at that time. The code that handles this functionality is shown in appendix C part C.

To only allow the Manager to select available rooms when assigning them, a request is sent to the Server with the string “CheckRoomAvailability~” followed by the booking ID for the booking they have selected in the program window and a number which represents the floor selected in the JComboBox within the Client window. The Server then uses this bookingID to select the check in and check out dates, represented in the Booking table by checkIn and checkOut for this booking, by adding the constraint ‘WHERE idBooking = splitMessage[1]’, in which splitMessage[1] is the booking reference sent to the Server. No loops are surrounding the SQL statements this time as the booking ID is unique, meaning only one booking will be found in the table. The Server then repeats what was done for checking the number of rooms available and compiles an array list for all of the dates that the user has booked for. The Server then executes two SQL statements on the database, which are contained within for loops and the results are added to a new array list. The SQL statements select the roomID for all entries in the Booking table where checkIn matches one of the values in the array list which contains the dates for the booking that has been selected, roomID is not equal to ‘Unassigned’ and the checkIn value is not equal to the checkOutResult variable. This is then repeated to select the roomID from records where where checkOut matches one of the values in the array list which contains the dates for the booking that has been selected, roomID is not equal to ‘Unassigned’ and the checkOut value is not equal to the checkInResult variable, which is the check in date retrieved from the table for the booking selected by the Manager. The Server then goes through each of the values in the array list that contains the result and ensures that the rooms found are on the floor selected by the Hotel Manager, by using a regular expression and the predefined matches method which allows the system to identify which floor the room was on. This works because each of the roomID entries follows the same format in that it is the room, followed by the floor number, followed by a ‘.’ and finally the room number.

1. Testing

The objective of this testing was to iron out any unidentified bugs and ensure that the program’s behavior was as we intended it to be, ensuring that the system was fit for purpose in a range of circumstances. For the testing, our approach was to conduct as many tests as possible with multiple combinations being tested in the fields, and incorrect entries were tested to ensure that the program handled them cleanly. In the test screenshots provided in the Test tables within the appendix for this report, we have only provided one combination of the test. In these final tests we have not found any syntax errors and at no point has an unhandled error been thrown. The test conditions were connecting to the database through a VPN connection whilst at University, and whilst connected directly to the workplace network at work. We have found that the testing conditions does not have an impact on the performance of our program, other than the fact that a database connection can be achieved very slightly quicker when connected to the workplace network. In the initial stages of the development, we used many print statements to track the values of variables, which enabled us to trace where any strange behavior was happening. For the final testing provided here, we have removed such print statements and have shown the window for both the Server and the Client, as we would not expect any of our end users to rely on the console. In order to carry out the final tests for the functionality of our hotel booking system, we have made sure to go through each of the cards in the cardLayout and use the system as we believe a user would, and test any of the situations where potential errors could arise. To check that we had caught any errors which could be raised by the user’s input, we deliberately made the user input incorrect, and this confirmed that the error is identified to the user by the appropriate means, and the errors do not cause the program to terminate unexpectedly. Testing has been carried out on the system throughout the development in the same way as below, as changes were made, but the testing discussed below and in the appendix are only screenshots of our testing on our final program.

The first part of our program we needed to make sure was working correctly was the login page. The login pages’ tests are shown in appendix D under ‘Test 1 Login Page’. The login page functionality was essential as problems at this stage would possibly prevent users from using the system completely. When testing the login page, we had to test whether the user’s input into the email and password fields matched an email and password in a row of the ‘LoginCredential’ table in the database. We needed to ensure that if the user’s input was not found in the table, a message appeared letting them know of the error that occurred, part a) of ‘Test 1 Login Page’ in appendix D shows this error. After testing this, we have discovered, that the email and password fields are not case sensitive, but we did not see this a problem for the functionality of our program.

Tests on whether the right home page was presented to the user depending on the user type associated to their login details in the database were carried out during all testing but shown in ‘Test 1 Login Page’ and ‘Test 6 Admin Login’ of appendix D. To test the Customer’s actions, we logged in as a Customer and to test the Admin’s functionality we would log in as an Admin/Hotel Manager.

The brief stated we had to have the functionality of being able to register as a Customer to use the hotel booking system. To make sure the user enters correct and valid information into the register fields, we had coded a lot of data validation within the action listener of the Create Account button. ‘Test 2 Register’ of appendix D show how different entries raise different errors appropriately. For example, in ‘Test 2 Register’ part b) of appendix D, the error ‘Emails do not match’ is shown. It is essential this error is caught as the system takes the first email enters as the one to enter the database and if this is not the email the user intended then they would not be able to login, as in our system the users log in using their email address.

While testing the Create Booking page, shown in ‘Test 3 Create Booking’ of appendix D, we had to make sure all our data validation was working correctly by entering values which should throw an error. Other crucial features we needed to check were whether required rooms were available for given date, there were enough rooms selected for the given number of guests, as we allowed our Customers to book multiple rooms at once, and the number of the ‘Enter Guest Details’ pages which were produced is equal to the number of guests the user entered. As previously mentioned, checking room availability is not to the working order of which we originally intended because of the error shown in ‘Test 3 Create Booking’ part o) in appendix D. However, the testing for the other considerations mentioned were successful. Amend booking tests in ‘Test 4 ‘Amend Booking’ part of appendix D followed the same logic and testing methods which were used to test the Create Booking.

When testing the admin pages, we had to make sure the changes the Admin made to the database affecting the bookings table was correctly carried out and therefore the changes would be visible by the customers also. For example, in ‘Test 8 Admin Assign Rooms to Bookings’ in appendix D, rooms are assigned to bookings the Customers have made, and we had to test whether the Customers can see the assigned rooms in their account also. So after the we had made these changes as an Admin we logged in as a Customer and viewed bookings to see whether we could see the assigned room to the booking (‘Test 9 Customer View Bookings’).

From this testing we are pleased to see that almost all of the program has acted as we intended it to. We have found no places where the code needs to be changed for what the user inputs, for example error handling, and we do not see any obvious changes needing to be made to the graphical user interfaces. The layout we present forms in to the users makes them easy to understand and use, and avoids overloading the user and prevents them having to jump around the page too much, as everything is presented in a logical manner.

1. Reflection and Responsibilities

On reflection we feel that for the most part, this project went well. We are disappointed by the fact that the logic we have implemented for the checking the number of rooms available when creating and amending a booking is flawed, however putting that aside we have a working system that successfully implements a Client-Server relationship, and demonstrates a JDBC connection.

The group work for this project went well, as there were only two of us in the group it was easy to delegate tasks between us without the need for a group leader and we have managed to come to agreements easily. When dividing the tasks we tried to split them in a manner which allowed people to work on things which were functionally similar, so that the logic could be reapplied and avoided us both creating very similar code, which could have used up a lot of time. As we work together, we have had regular contact which has made making decisions on how to move the project forward easy. At the times it has been needed, we have both spent time looking at each other’s code to provide one another with help and/or advice.

In terms of the system, we feel that the development has gone well and we have managed to achieve almost all of what we intended to do, and we are confident that what we have produced is an easy to use system, which is what we wanted to provide. The system we have produced also has a nice aesthetic appeal to it, which hopefully gives rise to a positive user experience.

On the server side however we are frustrated by the fact that the Server we have produced is only a single-threaded Server, and as such only allows a single Client to connect to it at any time. If we had had more time, we would have been able to implement a multi-threaded Server which would have allowed multiple Clients to connect to it at any one time. These multi-threaded Servers are what would typically be seen in industry, since you do not normally have a Server running for every single Client you want to connect; you would have a single Server that would handle multiple connections and would shift any incoming connections to other ports.

For the database, we shared the work load in creating and managing it, since it was completely new to both of us, having not taken the module in which databases are studied yet. This maybe took a little longer than expected, but we are happy with the database we have produced, as the correct relationships exist between the tables, and the tables store the data in a manner which is clear and avoids repeating entries.

*Table 1 Roles and Responsibilities*

|  |  |
| --- | --- |
| **Name and ID** | **Roles** |
|  | * UML Class and Use Case Diagrams * Creating Hotel Logo * Setting up the initial server code and SQL connection * View Bookings(Customer/Admin) * Amend Bookings(Customer) * Update Personal Details (Customer/Admin) * Assign Bookings(Admin) * View Bookings Per Room(Admin) * Checking Room Availability * Log Out * Admin class * Report Supervisor |
|  | * Create a Booking (Customer) * Add Guests to a Booking(Customer) * Log In * Booking class * User class * Customer class * Register(Customer) * Tidying and checking all code * Data validation for entering personal details for registering and updating(Customer/Admin) * Data validation regarding the ratio of rooms to people and whether dates were in the future and the check-out date was after the check-in date * Testing Supervisor |

1. Conclusion

This report has explored the design and implementation for a Hotel Booking System based on a given brief. The system we modelled in the initial stages of this project, demonstrated through the use of UML techniques and the planning photos shown in appendix E, has translated well into the system that we have produced. The system that we have produced performs data validation on all entries that it takes in, and ensures that they are within a legal format for the program. Where possible, we have tried to take into account any errors and handled them using try and catch clauses, or added a throws declaration to the method heading. If we were to produce this system again, or deploy it in the real world, we would make some changes to the way it was set up. After correcting the problem with room availability and adding the Server’s capability to connect to multiple clients, we would have enabled functionality within the system in which the admin could automate assigning rooms to bookings, which may be more appropriate to a full scale hotel/hotel group. In a deployed system, the Client program would have allowed an existing Hotel Manager/Admin to create more users of this type, to enable multiple users to share their responsibilities, rather than having to manipulate the database directly. Aside from creating additional Admins, there should be no reasons to directly manipulate the system which is what we wanted to achieve. Within this report we have tried to demonstrate as much of the functionality of the system as we can and have tried to provide an insight into how we came to this solution, from the planning stages, development and all the way through to testing, and what could be done moving forward.

1. References and Resources

This section includes the references for sources used in the report, and links to the resources that we have used in the development of our code.

1. Maryam Khan (2014) *Client server network*. Available at: <https://getrevising.co.uk/grids/client_server_network> (Last accessed 4 May 2017)
2. Winston Lievsay (2010) *Java GUI tutorial 19 – JTable*. Available at: <https://www.youtube.com/watch?v=wniqpx8OQxo> (Last accessed 28 April 2017)
3. Oracle (2015) *Java tutorials code sample – ListDemo.java*. Available at: <https://docs.oracle.com/javase/tutorial/displayCode.html?code=https://docs.oracle.com/javase/tutorial/uiswing/examples/components/ListDemoProject/src/components/ListDemo.java> (Last accessed 26 April 2017)
4. Oracle (2016) *ArrayList Java platform SE 7.* Available at: <https://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html> (Last accessed 26 April 2017)
5. Various Authors (2011) *The getSource() and getActionCommand().* Available at: <http://stackoverflow.com/questions/8214958/the-getsource-and-getactioncommand> (Last Accessed 27 April 2017)
6. Various Authors (2013) *How to update on cascade in MySQL.* Available at: <http://stackoverflow.com/questions/16779552/how-to-update-on-cascade-in-mysql> (Last Accessed 2 May 2017)
7. Various Authors (2009) *Cannot delete or update a parent row: a foreign key constraint fails.* Available at: <http://stackoverflow.com/questions/1905470/cannot-delete-or-update-a-parent-row-a-foreign-key-constraint-fails> (Last Accessed 2 May 2017)
8. Various Authors (2013) *How to convert current date into string in java?* Available at: <http://stackoverflow.com/questions/2942857/how-to-convert-current-date-into-string-in-java> (Last accessed 18 April 2017)
9. Various Authors (2013) *Changing Panels using the Card layout* Available at:

<http://stackoverflow.com/questions/14821952/changing-panels-using-the-card-layout>

1. Appendix

## Appendix A

The attributes that have been omitted from the Client class in the Class diagram are listed here, as they appear within the Java Client class:

private Container con**;**

private ObjectOutputStream output**;**

private ObjectInputStream input**;**

private String message **=** ""**;**

private String hbsServer**;**

private Socket client**;**

private CardLayout cardLayout**;**

private JPanel cards**;**

private BorderLayout fullLayout**;**

private GridLayout centerLayout**;**

private JPanel inputPanel**;**

private JPanel top**;**

private ImageIcon imageicon**;**

private JLabel header**;**

private JPanel bottom**;**

private JPanel loginPage**;**

private JLabel userName**;**

private JLabel password**;**

private JTextField userNameF**;**

private JPasswordField passwordF**;**

private JButton login**;**

private JButton register**;**

private JPanel registerPage**;**

private JLabel name1**;**

private JLabel name2**;**

private JLabel houseNum**;**

private JLabel postcode**;**

private JLabel email**;**

private JLabel email2**;**

private JLabel pass**;**

private JLabel password2**;**

private JLabel phoneNum**;**

private JLabel cardNum**;**

private JLabel security**;**

private JLabel expiryDate**;**

private JTextField name1F**;**

private JTextField name2F**;**

private JTextField houseNumF**;**

private JTextField postcodeF**;**

private JTextField emailF**;**

private JTextField email2F**;**

private JPasswordField password2F**;**

private JPasswordField passF**;**

private JTextField phoneNumF**;**

private JTextField cardNumF**;**

private JTextField securityF**;**

private JTextField expiryF**;**

private JButton confirm**;**

private JButton backLogin**;**

private JPanel updatePanel**;**

private JLabel updateFirstName**;**

private JLabel updateSurname**;**

private JLabel updateHouseNumber**;**

private JLabel updatePostcode**;**

private JLabel updateEmail**;**

private JLabel updateEmailConfirm**;**

private JLabel updatePassword**;**

private JLabel updatePasswordConfirm**;**

private JLabel updatePhoneNumber**;**

private JLabel updateCardNumber**;**

private JLabel updateCardSecurityCode**;**

private JLabel updateCardExpiryDate**;**

private JTextField updateFirstNameField**;**

private JTextField updateSurnameField**;**

private JTextField updateHouseNumberField**;**

private JTextField updatePostcodeField**;**

private JTextField updateEmailField**;**

private JTextField updateEmailConfirmField**;**

private JPasswordField updatePasswordField**;**

private JPasswordField updatePasswordConfirmField**;**

private JTextField updatePhoneNumberField**;**

private JTextField updateCardNumberField**;**

private JTextField updateCardSecurityCodeField**;**

private JTextField updateCardExpiryDateField**;**

private JButton updateCancel**;**

private JButton updateConfirm**;**

private String **[]** newDetails**;**

private JPanel updateAdminPanel**;**

private JLabel updateAdminFirstName**;**

private JLabel updateAdminSurname**;**

private JLabel updateAdminHouseNumber**;**

private JLabel updateAdminPostcode**;**

private JLabel updateAdminEmail**;**

private JLabel updateAdminEmailConfirm**;**

private JLabel updateAdminPassword**;**

private JLabel updateAdminPasswordConfirm**;**

private JLabel updateAdminPhoneNumber**;**

private JLabel UpdateAdminBankAccountNumber**;**

private JLabel UpdateAdminBankAccountSortCode**;**

private JTextField updateAdminFirstNameField**;**

private JTextField updateAdminSurnameField**;**

private JTextField updateAdminHouseNumberField**;**

private JTextField updateAdminPostcodeField**;**

private JTextField updateAdminEmailField**;**

private JTextField updateAdminEmailConfirmField**;**

private JPasswordField updateAdminPasswordField**;**

private JPasswordField updateAdminPasswordConfirmField**;**

private JTextField updateAdminPhoneNumberField**;**

private JTextField UpdateAdminBankAccountNumberField**;**

private JTextField UpdateAdminBankAccountSortCodeField**;**

private JButton updateAdminCancel**;**

private JButton updateAdminConfirm**;**

private String **[]** newAdminDetails**;**

private JPanel customerHomeScreen**;**

private JButton createBookingButton**;**

private JButton customerViewBookingsButton**;**

private JButton updateDetailsButton**;**

private JButton customerLogOut**;**

private JPanel adminHomeScreen**;**

private JButton assignBookings**;**

private JButton adminViewBookings**;**

private JButton adminUpdateDetails**;**

private JButton adminLogOut**;**

private String bookingRef**;**

private String roomID**=**"Unassigned"**;**

private String roomType**;**

private String inDate**;**

private String outDate**;**

private String price**;**

private String custID**;**

private int numGuest **=**0**;**

private int guestCount **=** 0**;**

private int doubleNumber **=** 0**;**

private int singleNumber **=** 0**;**

private int totalPrice **=** 0**;**

private boolean refOkay **=** **false;**

private int refCount **=** 1**;**

private int singleCount **=** 0**;**

private int totalRoomNum**;**

private int doubleCount **=** 0**;**

private int doubleGuestCount **=** 2**;**

private JButton backToHome**;**

private JPanel createB1**;**

private JLabel checkIn**;**

private JLabel checkOut**;**

private JLabel numOfGuests**;**

private JLabel numOfDouble**;**

private JLabel numOfSingle**;**

private JTextField checkInF**;**

private JTextField checkOutF**;**

private JTextField numOfGuestsF**;**

private JComboBox**<**String**>** numOfDoubleF**;**

private JComboBox**<**String**>** numOfSingleF**;**

private JTextField error**;**

private JButton nextBooking**;**

private int numberOfDoubleRoomsAvailable**;**

private int numberOfSingleRoomsAvailable**;**

private JButton checkAvailabilityButton**;**

private JLabel checkAvailabilityLabel**;**

private JPanel createB2**;**

private JLabel guestNumber**;**

private JLabel guestRoom**;**

private JLabel guestName**;**

private JLabel guestSurname**;**

private JLabel guestAge**;**

private JTextField guestNameF**;**

private JTextField guestSurnameF**;**

private JTextField guestAgeF**;**

private JButton enterGuest**;**

private JButton noGuestB**;**

private JLabel blank**;**

private final String DATE\_FORMAT **=** "dd-MM-yyyy"**;**

private Booking booking**;**

private JPanel customerViewBookingsScreen**;**

private JLabel myBookingsJTableHeader**;**

private JTable customerBookingsTable**;**

private JButton viewBackToMenu**;**

private JScrollPane customerTableScrollPane**;**

private DefaultTableModel customerModel**;**

private JButton customerAmendBooking**;**

private JPanel customerAmendBookingPanel**;**

private JLabel customerAllBookingsComboBoxExplanationLabel**;**

private JComboBox**<**String**>** customerAllBookingsComboBox**;**

private JLabel assignedRoomsExplanation**;**

private JLabel assignedRoomsExplanation2**;**

private JLabel assignedRoomsExplanation3**;**

private JPanel assignedRoomsExplanationPanel**;**

private JLabel customerAmendRoomLabel**;**

private JComboBox**<**String**>** customerAmendRoomTypes**;**

private JLabel customerAmendCheckInLabel**;**

private JTextField customerAmendCheckInField**;**

private JLabel customerAmendCheckOutLabel**;**

private JTextField customerAmendCheckOutField**;**

private JLabel customerAmendPriceLabel**;**

private JTextField customerAmendPriceField**;**

private JPanel customerAmendBookingPanelAmendComponents**;**

private JPanel customerAllBookingsComboBoxPanel**;**

private JButton customerAmendConfirm**;**

private JButton customerAmendDelete**;**

private JButton amendBackToHome**;**

private JPanel adminViewBookingsScreen**;**

private JLabel adminBookingsJTableHeader**;**

private JTable adminBookingsTable**;**

private JButton adminViewBackToMenu**;**

private JScrollPane adminTableScrollPane**;**

private DefaultTableModel adminModel**;**

private JPanel adminAssignBookingsScreen**;**

private JButton adminAssignBackToMenu**;**

private DefaultListModel**<**String**>** listModel**;**

private JList**<**String**>** unassignedBookingsList**;**

private JLabel unassignedBookingsLabel**;**

private JScrollPane unassignedBookingsListScrollPane**;**

private JLabel unassignedRoomTypeLabel**;**

private JLabel unassignedRoomType**;**

private JPanel chooseBookingPanel**;**

private JLabel chooseFloorLabel**;**

private JComboBox**<**String**>** chooseFloor**;**

private JPanel hotelRoomsPanel**;**

private JPanel room1Panel**;**

private JPanel room2Panel**;**

private JPanel room3Panel**;**

private JPanel room4Panel**;**

private JPanel room5Panel**;**

private JPanel room6Panel**;**

private JPanel room7Panel**;**

private JPanel room8Panel**;**

private JLabel room1RoomTypeLabel**;**

private JLabel room2RoomTypeLabel**;**

private JLabel room3RoomTypeLabel**;**

private JLabel room4RoomTypeLabel**;**

private JLabel room5RoomTypeLabel**;**

private JLabel room6RoomTypeLabel**;**

private JLabel room7RoomTypeLabel**;**

private JLabel room8RoomTypeLabel**;**

private JButton room1Assign**;**

private JButton room2Assign**;**

private JButton room3Assign**;**

private JButton room4Assign**;**

private JButton room5Assign**;**

private JButton room6Assign**;**

private JButton room7Assign**;**

private JButton room8Assign**;**

private JLabel **[]** roomTypeLabelsArray**;**

private JButton **[]** roomAssignButtonsArray**;**

private JLabel loadingLabel**;**

private JButton adminViewBookingsPerRoom**;**

private JTable bookingsPerRoom**;**

private DefaultTableModel bookingsPerRoomModel**;**

private JScrollPane bookingsPerRoomTableScrollPane**;**

private JLabel bookingsPerRoomSelectorLabel**;**

private JComboBox**<**String**>** bookingsPerRoomSelector**;**

private JPanel bookingsPerRoomDialogPanel**;**

private Customer customer**;**

private Admin admin**;**

## Appendix B

The following Appendix provides some of the project code, which has been discussed in this report.

Part A)

private void processConnection**()** **throws** IOException

**{**

String message **=** "Connection successful"**;**

**this.**displayMessage**(**message**);**

sendData**(**message**);**

String command **=** ""**;**

**do**

**{**

**try**

**{**

message **=** **(**String**)** input**.**readObject**();**

**this.**displayMessage**(** "\n" **+** message **);**

String **[]** splitMessage **=** message**.**split**(**"~"**);**

command **=** splitMessage**[**0**];**

System**.**out**.**println**(**command**);**

**try**

**{**

statement **=** sqlconnection**.**createStatement**();**

**switch** **(**command**)**

**{**

**case** ""**:**

System**.**out**.**println**(**"No command"**);**

**break;**

**case** "Login"**:**

**try**

**{**

resultSet **=** statement**.**executeQuery**(**"SELECT userType FROM LoginCredential WHERE email = '" **+** splitMessage**[**1**]** **+** "' AND password = '" **+** splitMessage**[**2**]** **+** "'"**);**

ResultSetMetaData metaData **=** resultSet**.**getMetaData**();**

int numberOfColumns **=** metaData**.**getColumnCount**();**

**this.**sendData**(**"SQL Statement executed successfully"**);**

**if** **(!**resultSet**.**next**())**

**{**

**this.**sendData**(**"Denied"**);**

**}**

**else**

**{**

String userType **=** resultSet**.**getObject**(**1**).**toString**();**

**this.**sendData**(**userType**);**

**if** **(**userType**.**equals**(**"Customer"**))**

**{**

resultSet **=** statement**.**executeQuery**(**"SELECT \* FROM Customer WHERE email = '" **+** splitMessage**[**1**]** **+** "'"**);**

String results **=** "CustomerObjectInitialising~"**;**

metaData **=** resultSet**.**getMetaData**();**

numberOfColumns **=** metaData**.**getColumnCount**();**

**while** **(**resultSet**.**next**())**

**{**

**for** **(**int i **=** 1**;** i **<=** numberOfColumns**;** i**++)**

**{**

results **+=** resultSet**.**getObject**(**i**).**toString**()** **+** "~"**;**

**}**

**}**

**this.**sendData**(**results**);**

**}**

**else** **if** **(**userType**.**equals**(**"Admin"**))**

**{**

resultSet **=** statement**.**executeQuery**(**"SELECT \* FROM Admin WHERE emailAddress = '" **+** splitMessage**[**1**]** **+** "'"**);**

String results **=** "AdminObjectInitialising~"**;**

metaData **=** resultSet**.**getMetaData**();**

numberOfColumns **=** metaData**.**getColumnCount**();**

**while** **(**resultSet**.**next**())**

**{**

**for** **(**int i **=** 1**;** i **<=** numberOfColumns**;** i**++)**

**{**

results **+=** resultSet**.**getObject**(**i**).**toString**()** **+** "~"**;**

**}**

**}**

**this.**sendData**(**results**);**

**}**

**}**

**}**

**catch** **(**ArrayIndexOutOfBoundsException arrayExc**)**

**{**

**this.**sendData**(**"Denied"**);**

**}**

**break;**

**case** "Register"**:**

**try**

**{**

String sql **=** "INSERT INTO LoginCredential VALUES('" **+** splitMessage**[**10**]** **+** "', '" **+** splitMessage**[**11**]** **+** "', '" **+** splitMessage**[**12**]** **+** "');"**;**

**this.**displayMessage**(**sql**);**

int value **=** statement**.**executeUpdate**(**sql**);**

**this.**displayMessage**(**value **+** " rows inserted."**);**

sql **=** "INSERT INTO Customer VALUES('" **+** splitMessage**[**1**]** **+** "', '" **+** splitMessage**[**2**]** **+** "', '" **+** splitMessage**[**3**]** **+** "', '" **+** splitMessage**[**4**]** **+** "', '" **+** splitMessage**[**5**]** **+** "', '" **+** splitMessage**[**6**]** **+** "', '" **+** splitMessage**[**7**]** **+** "', '" **+** splitMessage**[**8**]** **+** "', '" **+** splitMessage**[**9**]** **+** "', '" **+** splitMessage**[**10**]** **+** "');"**;**

**this.**displayMessage**(**sql**);**

value **=** statement**.**executeUpdate**(**sql**);**

**this.**displayMessage**(**value **+** " rows inserted."**);**

**this.**sendData**(**"Registered"**);**

**}**

**catch** **(**MySQLIntegrityConstraintViolationException duplicationError**)**

**{**

sendData**(**"DuplicationError"**);**

**}**

**break;**

**case** "Create"**:**

String sql **=** "INSERT INTO Booking VALUES('" **+** splitMessage**[**1**]** **+** "', '" **+** splitMessage**[**2**]** **+** "', '" **+** splitMessage**[**3**]** **+** "', '" **+** splitMessage**[**4**]** **+** "', '" **+** splitMessage**[**5**]** **+** "','" **+** splitMessage**[**6**]+**"','" **+** splitMessage**[**7**]+**"')"**;**

**this.**displayMessage**(**sql**);**

int value **=** statement**.**executeUpdate**(**sql**);**

**this.**displayMessage**(**value **+** " rows inserted."**);**

**this.**sendData**(**"SQL Statement executed successfully"**);**

**break;**

**case** "View"**:**

resultSet **=** statement**.**executeQuery**(**"SELECT idBooking, roomID, room, checkIn, checkOut, totalPriceGBP FROM Booking WHERE idCustomer = '" **+** splitMessage**[**1**]** **+** "'"**);**

ResultSetMetaData metaData **=** resultSet**.**getMetaData**();**

int numberOfColumns **=** metaData**.**getColumnCount**();**

String results **=** "ThisCustomersBookings~"**;**

**while** **(**resultSet**.**next**())**

**{**

**for** **(**int i **=** 1**;** i **<=** numberOfColumns**;** i**++)**

**{**

results **+=** resultSet**.**getObject**(**i**).**toString**()** **+** "~"**;**

**}**

**}**

System**.**out**.**println**(**results**);**

**this.**sendData**(**results**);**

**break;**

**case** "ViewBookingIDOnly"**:**

SimpleDateFormat todaysFormat **=** **new** SimpleDateFormat**(**"yyyy-MM-dd"**);**

Calendar todaysCalendar **=** Calendar**.**getInstance**();**

String todayString **=** todaysFormat**.**format**(**todaysCalendar**.**getTime**());**

resultSet **=** statement**.**executeQuery**(**"SELECT idBooking FROM Booking WHERE idCustomer = '" **+** splitMessage**[**1**]** **+** "' AND roomID = 'Unassigned' AND checkOut > '" **+** todayString **+** "'"**);**

String idResults **=** "ThisCustomersBookingsIDOnly~"**;**

**while** **(**resultSet**.**next**())**

**{**

idResults **+=** resultSet**.**getObject**(**1**).**toString**()** **+** "~"**;**

**}**

System**.**out**.**println**(**idResults**);**

**this.**sendData**(**idResults**);**

**break;**

**case** "DetailsAboutThisBooking"**:**

resultSet **=** statement**.**executeQuery**(**"SELECT checkIn, checkOut, room, totalPriceGBP FROM Booking WHERE idBooking = '" **+** splitMessage**[**1**]** **+** "'"**);**

String thisBookingsResults **=** "ThisBookingsDetails~"**;**

ResultSetMetaData thisBookingsMetaData **=** resultSet**.**getMetaData**();**

int thisBookingsNumberOfColumns **=** thisBookingsMetaData**.**getColumnCount**();**

int resultIndex **=** 1**;**

**while** **(**resultSet**.**next**())**

**{**

**for** **(**int i **=** 1**;** i **<=** thisBookingsNumberOfColumns**;** i**++)**

**{**

thisBookingsResults **+=** resultSet**.**getObject**(**i**).**toString**()** **+** "~"**;**

**}**

**}**

System**.**out**.**println**(**thisBookingsResults**);**

**this.**sendData**(**thisBookingsResults**);**

**break;**

**case** "DeleteThisBooking"**:**

sql **=** "DELETE FROM Booking WHERE idBooking = '" **+** splitMessage**[**1**]** **+** "'"**;**

**this.**displayMessage**(**sql**);**

value **=** statement**.**executeUpdate**(**sql**);**

**this.**displayMessage**(**value **+** " rows deleted."**);**

**this.**sendData**(**"BookingCancelled"**);**

**break;**

**case** "AdminView"**:**

resultSet **=** statement**.**executeQuery**(**"SELECT \* FROM Booking"**);**

ResultSetMetaData adminMetaData **=** resultSet**.**getMetaData**();**

int adminNumberOfColumns **=** adminMetaData**.**getColumnCount**();**

String adminResults **=** "AllBookings~"**;**

**while** **(**resultSet**.**next**())**

**{**

**for** **(**int i **=** 1**;** i **<=** adminNumberOfColumns**;** i**++)**

**{**

adminResults **+=** resultSet**.**getObject**(**i**).**toString**()** **+** "~"**;**

**}**

**}**

System**.**out**.**println**(**adminResults**);**

**this.**sendData**(**adminResults**);**

**break;**

**case** "Amend"**:**

sql **=** "UPDATE Booking SET checkIn = '" **+** splitMessage**[**2**]** **+** "', checkOut = '" **+** splitMessage**[**3**]** **+** "', totalPriceGBP ='" **+** splitMessage**[**4**]** **+** "' WHERE idBooking = '" **+** splitMessage**[**1**]** **+** "'"**;**

**this.**displayMessage**(**sql**);**

value **=** statement**.**executeUpdate**(**sql**);**

**this.**displayMessage**(**value **+** " rows ammended."**);**

**this.**sendData**(**"SQL Statement executed successfully"**);**

**break;**

**case** "CompileListOfUnassignedBookings"**:**

resultSet **=** statement**.**executeQuery**(**"SELECT idBooking FROM Booking WHERE roomID = 'Unassigned'"**);**

ResultSetMetaData unassignedMetaData **=** resultSet**.**getMetaData**();**

int unassignedNumberOfColumns **=** unassignedMetaData**.**getColumnCount**();**

String unassignedResults **=** "AllUnassignedBookings~"**;**

**while** **(**resultSet**.**next**())**

**{**

**for** **(**int i **=** 1**;** i **<=** unassignedNumberOfColumns**;** i**++)**

**{**

unassignedResults **+=** resultSet**.**getObject**(**i**).**toString**()** **+** "~"**;**

**}**

**}**

System**.**out**.**println**(**unassignedResults**);**

**this.**sendData**(**unassignedResults**);**

**break;**

**case** "RoomTypeRequired"**:**

resultSet **=** statement**.**executeQuery**(**"SELECT room FROM Booking WHERE idBooking = '" **+** splitMessage**[**1**]** **+** "'"**);**

String roomReqResult **=** "RoomRequiredResult~"**;**

**if** **(**resultSet**.**next**())**

**{**

roomReqResult **+=** resultSet**.**getObject**(**1**).**toString**()** **+** "~"**;**

**}**

System**.**out**.**println**(**roomReqResult**);**

**this.**sendData**(**roomReqResult**);**

**break;**

**case** "CheckRoomAvailability"**:**

resultSet **=** statement**.**executeQuery**(**"SELECT checkIn FROM Booking WHERE idBooking = '" **+** splitMessage**[**1**]** **+** "'"**);**

String checkInResult **=** ""**;**

**if** **(**resultSet**.**next**())**

**{**

checkInResult **+=** resultSet**.**getObject**(**1**).**toString**();**

**}**

resultSet **=** statement**.**executeQuery**(**"SELECT checkOut FROM Booking WHERE idBooking = '" **+** splitMessage**[**1**]** **+** "'"**);**

String checkOutResult **=** ""**;**

**if** **(**resultSet**.**next**())**

**{**

checkOutResult **+=** resultSet**.**getObject**(**1**).**toString**();**

**}**

ArrayList**<**String**>** holidayDaysArrayList **=** **new** ArrayList**<**String**>(**0**);**

SimpleDateFormat holidayDates **=** **new** SimpleDateFormat**(**"yyyy-MM-dd"**);**

Calendar holidayCalendar **=** Calendar**.**getInstance**();**

holidayCalendar**.**setTime**(**holidayDates**.**parse**(**checkInResult**));**

String day **=** holidayDates**.**format**(**holidayCalendar**.**getTime**());**

holidayDaysArrayList**.**add**(**day**);**

**while** **(!**day**.**equals**(**checkOutResult**))**

**{**

holidayCalendar**.**add**(**Calendar**.**DATE**,** 1**);**

day **=** holidayDates**.**format**(**holidayCalendar**.**getTime**());**

holidayDaysArrayList**.**add**(**day**);**

**}**

System**.**out**.**println**(**holidayDaysArrayList**);**

ArrayList**<**String**>** allBookedRoomsArrayList **=** **new** ArrayList**<**String**>(**0**);**

**for** **(**String each **:** holidayDaysArrayList**)**

**{**

resultSet **=** statement**.**executeQuery**(**"SELECT roomID FROM Booking WHERE checkIn = '" **+** each **+** "' AND roomID != 'Unassigned' AND checkIn != '" **+** checkOutResult **+** "'"**);**

**while** **(**resultSet**.**next**())**

**{**

allBookedRoomsArrayList**.**add**(**resultSet**.**getObject**(**1**).**toString**());**

**}**

resultSet **=** statement**.**executeQuery**(**"SELECT roomID FROM Booking WHERE checkOut = '" **+** each **+** "' AND roomID != 'Unassigned' AND checkOut != '" **+** checkInResult **+** "'"**);**

**while** **(**resultSet**.**next**())**

**{**

allBookedRoomsArrayList**.**add**(**resultSet**.**getObject**(**1**).**toString**());**

**}**

**}**

System**.**out**.**println**(**allBookedRoomsArrayList**);**

String bookedRooms **=** "BookedRoomsOnSelectedFloor~"**;**

String regexExpression **=** "room" **+** splitMessage**[**2**]** **+** "\\.([1-8])"**;**

**for** **(**String each **:** allBookedRoomsArrayList**)**

**{**

**if** **(**each**.**matches**(**regexExpression**))**

**{**

bookedRooms **+=** each **+** "~"**;**

**}**

**}**

System**.**out**.**println**(**bookedRooms**);**

sendData**(**bookedRooms**);**

**break;**

**case** "CheckHowManyRooms"**:**

String **[]** date1 **=** splitMessage**[**1**].**split**(**"-"**);**

String checkInDateRequested **=** date1**[**2**]** **+** "-" **+** date1**[**1**]** **+** "-" **+** date1**[**0**];**

String **[]** date2 **=** splitMessage**[**2**].**split**(**"-"**);**

String checkOutDateRequested **=** date2**[**2**]** **+** "-" **+** date2**[**1**]** **+** "-" **+** date2**[**0**];**

sendData**(**checkInDateRequested**);**

sendData**(**checkOutDateRequested**);**

ArrayList**<**String**>** holidayDaysRequestedArrayList **=** **new** ArrayList**<**String**>(**0**);**

SimpleDateFormat holidayRequestedDates **=** **new** SimpleDateFormat**(**"yyyy-MM-dd"**);**

Calendar holidayRequestedCalendar **=** Calendar**.**getInstance**();**

sendData**(**holidayRequestedCalendar**.**toString**());**

holidayRequestedCalendar**.**setTime**(**holidayRequestedDates**.**parse**(**checkInDateRequested**));**

sendData**(**checkInDateRequested**);**

sendData**(**holidayRequestedCalendar**.**toString**());**

String dayRequested **=** holidayRequestedDates**.**format**(**holidayRequestedCalendar**.**getTime**());**

holidayDaysRequestedArrayList**.**add**(**dayRequested**);**

sendData**(**"requested" **+** dayRequested**);**

sendData**(**checkOutDateRequested**);**

**while** **(!**dayRequested**.**equals**(**checkOutDateRequested**))**

**{**

holidayRequestedCalendar**.**add**(**Calendar**.**DATE**,** 1**);**

System**.**out**.**println**(**"Day Added"**);**

dayRequested **=** holidayRequestedDates**.**format**(**holidayRequestedCalendar**.**getTime**());**

System**.**out**.**println**(**dayRequested**);**

System**.**out**.**println**(**"Got time"**);**

holidayDaysRequestedArrayList**.**add**(**dayRequested**);**

**}**

System**.**out**.**println**(**holidayDaysRequestedArrayList**);**

ArrayList**<**String**>** allDoubleBookedRoomsArrayList **=** **new** ArrayList**<**String**>(**0**);**

**for** **(**String each **:** holidayDaysRequestedArrayList**)**

**{**

resultSet **=** statement**.**executeQuery**(**"SELECT roomID FROM Booking WHERE checkIn = '" **+** each **+** "' AND room = 'Double'"**);**

**while** **(**resultSet**.**next**())**

**{**

allDoubleBookedRoomsArrayList**.**add**(**resultSet**.**getObject**(**1**).**toString**());**

**}**

**}**

ArrayList**<**String**>** allSingleBookedRoomsArrayList **=** **new** ArrayList**<**String**>(**0**);**

**for** **(**String each **:** holidayDaysRequestedArrayList**)**

**{**

resultSet **=** statement**.**executeQuery**(**"SELECT roomID FROM Booking WHERE checkIn = '" **+** each **+** "' AND room = 'Single'"**);**

**while** **(**resultSet**.**next**())**

**{**

allSingleBookedRoomsArrayList**.**add**(**resultSet**.**getObject**(**1**).**toString**());**

**}**

**}**

System**.**out**.**println**(**allDoubleBookedRoomsArrayList**.**size**());**

System**.**out**.**println**(**allDoubleBookedRoomsArrayList**);**

int doublesRemaining **=** 12 **-** allDoubleBookedRoomsArrayList**.**size**();**

int singlesRemaining **=** 12 **-** allSingleBookedRoomsArrayList**.**size**();**

**this.**sendData**(**"ThisManyRoomsAreAvailable(d-s)~" **+** doublesRemaining **+** "~" **+** singlesRemaining**);**

**break;**

**case** "Assign"**:**

sql **=** "UPDATE Booking SET roomID = '" **+** splitMessage**[**2**]** **+** "' WHERE idBooking = '" **+** splitMessage**[**1**]** **+** "'"**;**

**this.**displayMessage**(**sql**);**

value **=** statement**.**executeUpdate**(**sql**);**

**this.**displayMessage**(**value **+** " rooms assigned."**);**

**this.**sendData**(**"SQL Statement executed successfully"**);**

**break;**

**case** "UpdateDetails"**:**

sql **=** "UPDATE LoginCredential SET email = '" **+** splitMessage**[**2**]** **+** "', password = '" **+** splitMessage**[**3**]** **+** "' WHERE email = '" **+** splitMessage**[**1**]** **+** "'"**;**

**this.**displayMessage**(**sql**);**

value **=** statement**.**executeUpdate**(**sql**);**

**this.**displayMessage**(**value **+** " logins updated."**);**

sql **=** "UPDATE Customer SET firstName = '" **+** splitMessage**[**4**]** **+** "', lastName = '" **+** splitMessage**[**5**]** **+** "', houseNumber = '" **+** splitMessage**[**6**]** **+** "', postcode = '" **+** splitMessage**[**7**]** **+** "', cardNumber = '" **+** splitMessage**[**8**]** **+** "', securityCode = '" **+** splitMessage**[**9**]** **+** "', expiryDate = '" **+** splitMessage**[**10**]** **+** "', phoneNumber = '" **+** splitMessage**[**11**]** **+** "' WHERE idCustomer = '" **+** splitMessage**[**12**]** **+** "'"**;**

**this.**displayMessage**(**sql**);**

value **=** statement**.**executeUpdate**(**sql**);**

**this.**displayMessage**(**value **+** " accounts updated."**);**

**this.**sendData**(**"CustomerUpdateCompleted"**);**

**break;**

**case** "UpdateBooking"**:**

sql **=** "UPDATE Booking SET checkIn = '" **+** splitMessage**[**2**]** **+** "', checkOut = '" **+** splitMessage**[**3**]** **+** "', totalPriceGBP = '" **+** splitMessage**[**4**]** **+** "', room = '" **+** splitMessage**[**5**]** **+** "' WHERE idBooking = '" **+** splitMessage**[**1**]** **+** "'"**;**

**this.**displayMessage**(**sql**);**

value **=** statement**.**executeUpdate**(**sql**);**

**this.**displayMessage**(**value **+** " bookings updated."**);**

**this.**sendData**(**"BookingUpdateCompleted"**);**

**break;**

**case** "FloorRoomTypes"**:**

resultSet **=** statement**.**executeQuery**(**"SELECT type FROM Room WHERE floor = '" **+** splitMessage**[**1**]** **+** "'"**);**

String floorRoomTypes **=** "FloorRoomTypeResult~"**;**

**while** **(**resultSet**.**next**())**

**{**

floorRoomTypes **+=** resultSet**.**getObject**(**1**).**toString**()** **+** "~"**;**

**}**

System**.**out**.**println**(**floorRoomTypes**);**

**this.**sendData**(**floorRoomTypes**);**

**break;**

**case** "GetAllBookingsForRoom"**:**

SimpleDateFormat todayFormat **=** **new** SimpleDateFormat**(**"yyyy-MM-dd"**);**

Calendar todayCalendar **=** Calendar**.**getInstance**();**

String today **=** todayFormat**.**format**(**todayCalendar**.**getTime**());**

resultSet **=** statement**.**executeQuery**(**"SELECT roomID, idBooking, room, checkIn, checkOut, totalPriceGBP, idCustomer FROM Booking WHERE roomID = '" **+** splitMessage**[**1**]** **+** "' AND checkOut > '" **+** today **+** "'"**);**

ResultSetMetaData roomMetaData **=** resultSet**.**getMetaData**();**

int roomResultsNumberOfColumns **=** roomMetaData**.**getColumnCount**();**

String roomBookings **=** "ThisRoomsBookings~"**;**

**while** **(**resultSet**.**next**())**

**{**

**for** **(**int i **=** 1**;** i **<=** roomResultsNumberOfColumns**;** i**++)**

**{**

roomBookings **+=** resultSet**.**getObject**(**i**).**toString**()** **+** "~"**;**

**}**

**}**

System**.**out**.**println**(**roomBookings**);**

**this.**sendData**(**roomBookings**);**

**break;**

**case** "GetUsersPassword"**:**

resultSet **=** statement**.**executeQuery**(**"SELECT password, userType FROM LoginCredential WHERE email = '" **+** splitMessage**[**1**]** **+** "'"**);**

String password **=** "UsersPassword~"**;**

**while** **(**resultSet**.**next**())**

**{**

**for** **(**int i **=** 1**;** i **<=** 2**;** i**++)**

**{**

password **+=** resultSet**.**getObject**(**i**).**toString**()** **+** "~"**;**

**}**

**}**

System**.**out**.**println**(**password**);**

**this.**sendData**(**password**);**

**break;**

**case** "UpdateAdminDetails"**:**

sql **=** "UPDATE LoginCredential SET email = '" **+** splitMessage**[**2**]** **+** "', password = '" **+** splitMessage**[**3**]** **+** "' WHERE email = '" **+** splitMessage**[**1**]** **+** "'"**;**

**this.**displayMessage**(**sql**);**

value **=** statement**.**executeUpdate**(**sql**);**

**this.**displayMessage**(**value **+** " logins updated."**);**

sql **=** "UPDATE Admin SET firstName = '" **+** splitMessage**[**4**]** **+** "', lastName = '" **+** splitMessage**[**5**]** **+** "', houseNumber = '" **+** splitMessage**[**6**]** **+** "', postcode = '" **+** splitMessage**[**7**]** **+** "', bankAccountNumber = '" **+** splitMessage**[**8**]** **+** "', bankSortCode = '" **+** splitMessage**[**9**]** **+** "', phoneNumber = '" **+** splitMessage**[**10**]** **+** "' WHERE idAdmin = '" **+** splitMessage**[**11**]** **+** "'"**;**

**this.**displayMessage**(**sql**);**

value **=** statement**.**executeUpdate**(**sql**);**

**this.**displayMessage**(**value **+** " accounts updated."**);**

**this.**sendData**(**"AdminUpdateCompleted"**);**

**break;**

**case** "AddGuest"**:**

sql **=** "INSERT INTO Guest VALUES('" **+** splitMessage**[**1**]** **+** "', '" **+** splitMessage**[**2**]** **+** "', '" **+** splitMessage**[**3**]** **+**"', '" **+** splitMessage**[**4**]+** "');"**;**

**this.**displayMessage**(**sql**);**

value **=** statement**.**executeUpdate**(**sql**);**

**this.**displayMessage**(**"Guest added for booking reference " **+** splitMessage**[**4**]);**

**this.**sendData**(**"Guest Added"**);**

**break;**

**case** "Check"**:**

sql **=**"SELECT \* FROM " **+** splitMessage**[**1**]+** " WHERE "**+** splitMessage**[**2**]** **+** " RLIKE '"**+**splitMessage**[**3**]+**"\*';"**;**

**this.**displayMessage**(**sql**);**

resultSet **=** statement**.**executeQuery**(**sql**);**

**if** **(!**resultSet**.**next**())**

**{**

**this.**sendData**(**"Not Found"**);**

**}**

**break;**

**}**

**}**

**catch** **(**SQLException sql**)**

**{**

sql**.**printStackTrace**();**

**}**

**catch** **(**ParseException pe**)**

**{**

pe**.**printStackTrace**();**

**}**

**}**

**catch** **(**ClassNotFoundException classNotFoundException **)**

**{**

displayMessage**(** "\nUnknown object type received" **);**

**}**

**catch(**NullPointerException npe**)**

**{**

npe**.**printStackTrace**();**

**}**

**catch(**SocketException socketExc**)**

**{**

**this.**closeConnection**();**

message **=** "CLIENT>>> TERMINATE"**;**

**}**

**}** **while** **(!**message**.**equals**(** "CLIENT>>> TERMINATE" **)** **);**

**}**

## Appendix C

Database Screenshots



Figure 5: Snapshot of the Admin table.

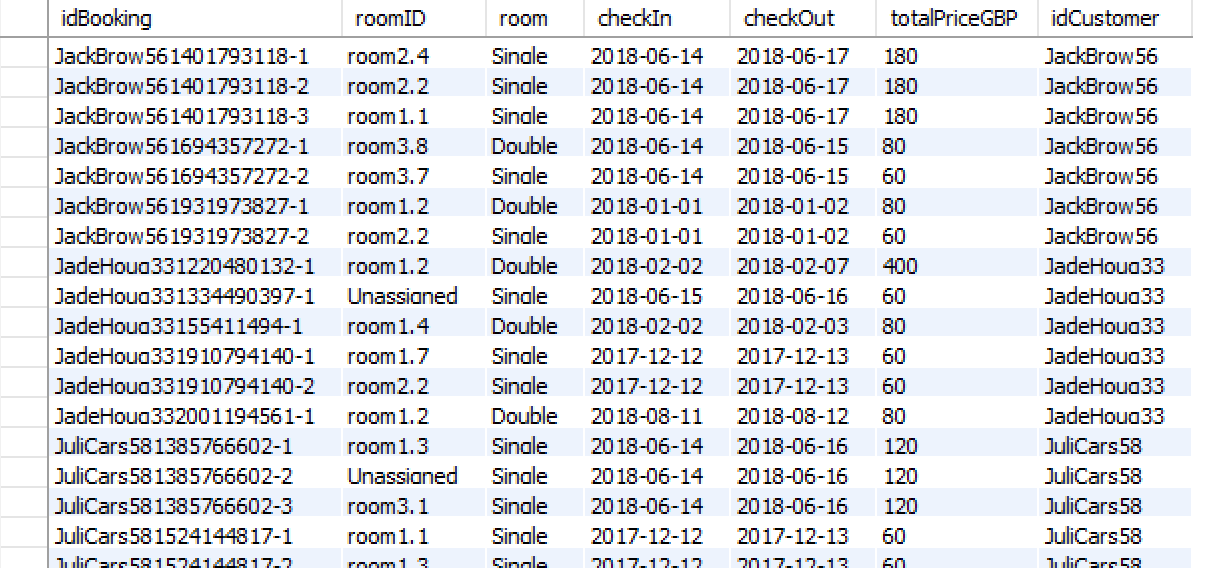


Figure 6: Snapshot of the Booking table.

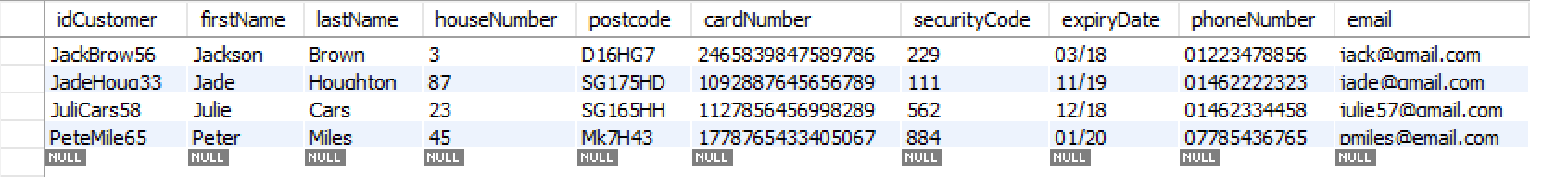


Figure 7: Snapshot of the Customer table.

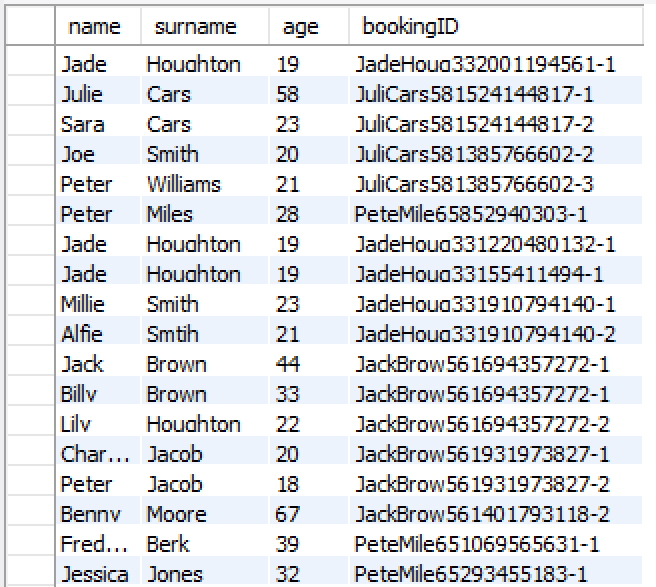


Figure 8: Snapshot of the Guest table.

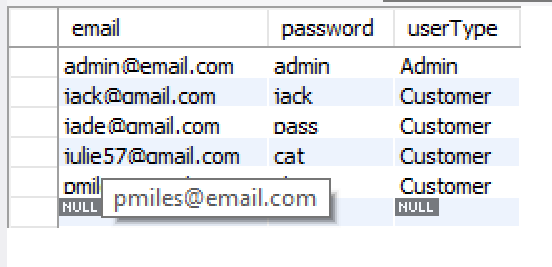


Figure 9: Snapshot of the LoginCredential table.

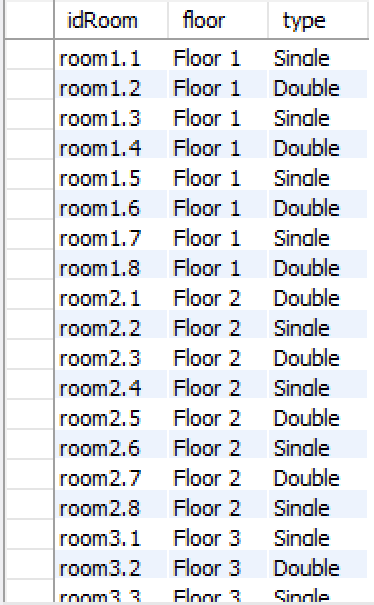


Figure 10: Snapshot of the Room table.

## Appendix D

Testing:

The tables displayed here go vertically down the page.

|  |  |
| --- | --- |
| **Test 1 Login Page** | **Test 2 Register** |
| a)When the only user is an admin with email- ‘[Admin@email.com](mailto:Admin@email.com)’ and password- ‘admin’ | C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Clicked Register.PNGa)When you click the button labelled ‘Create Account’ on the Login Page then following appears |
| C:\Users\jade.houghton\AppData\Local\Microsoft\Windows\INetCache\Content.Word\Login not on system.pngb)Email does not exist in the database therefore login is failed | C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Register-Emails do not match.PNGb)Customer enters different email addresses |
| c)This test is after the user ‘JadeHoug33’ is created and in the database. | C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Register - email invalid format.PNGc)Emails entered are the same but not in an email format |
| d)When the button labelled ‘Login’ is clicked after the login fields are enters as they are above the customer home page is raised as user type in the database is ‘customer’ | C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Register- password does not match.PNGd)Password Fields are different from one another |
|  | C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Register - invalid phone format.PNGe)Phone number is not 11 digits |
|  | C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Register- invalid security.PNGf)Security Code is not 3 digits |
|  | C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Register - card out of date.PNGg)Card Expiry Date year is in the past |
|  | C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Regisiter - invalid month.PNGh)Card Expiry Date month is not valid |
|  | C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Register - Successful.PNGi)Customer details are added to the database as shown below: |
|  | C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Home Page After Customer Registered.PNGj)After the account has been created and the customer has clicked okay on the message above, they are taken to the customer home page. |

|  |  |
| --- | --- |
| **Test 3 Create Booking** | **Test 4 Amend Booking** |
| C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Clicked Create Booking.PNGa)Clicked ‘Create Booking’ button on the customer home page | C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Clicked Ammend.PNGa)Clicked ‘Amend Booking’ button on the customer home page |
| b)User fill fields and selects options from drop down list. Date is incorrect as it is in the past. Clicked Check Availability. | C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Ammend- Drop Down.PNGb)All unassigned bookings can be view in the drop down arrow by their booking reference. |
| C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Create - Date incorrect format.PNGc)Date is not in dd-mm-yyyy format as required.. Clicked Check Availability | C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Ammend- Date error.PNGc)Booking chosen and fields are edited but check in date entered is before check in date. Clicked Check Availability and Confirm |
| C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Create - incorrect month.PNGd)Invalid month entered in date fields.. Clicked Check Availability. | C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Ammend - Change date message.PNGd)Correct date is entered into the fields and the customer clicked Check Availability and Confirm |
| C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Create - out date before in.PNGe)Check in date entered is before check in date. Clicked Check Availability | C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Ammend- Confirmed.PNGe)Changed booking confirmation |
| C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Create- Check Availability.PNGf)Correctly filled fields and clicked Check Availability and then Next | C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Ammend -single shown.PNGf)Selected a single room booking |
| C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Create - clicked next.PNGg)Customer is taken to guest entering fields. Guest 1, Room 1. | C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Ammend- change from single to double.PNGg)Changed room type from single to double and clicked Check Availability and Confirm. |
| C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Guest- name wrong.PNGh)Name entered has number and clicked Enter | C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Ammend - single to double confirmed.PNGh)After clicking the yes button above confirmation of the change is shown.  C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\database-bookings after ammend.PNGThe database which has changed is shown below: |
| C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Guest- age wrong.PNGi)Age entered contains letters and clicked Next | C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Ammend - delete.PNGi)Booking is selected and Delete is clicked |
| C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Guest - Guest 1 Added.PNGj)Correct fields and clicked Next | C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Ammend- delete confirmed.PNGj)Deletion confirmation message.  C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Database - bookings after delete.PNGDatabase after the selected booking has been deleted. |
| C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Guest - 2nd guest.PNGk)Customer take to Guest 2, Room 2 guest detail page. |  |
| C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Guest - guest 2 added.PNGl)Correct fields and click Next. |  |
| C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Guest - confirmation.PNGm)No more guest to add so confirmation message appears |  |
| Database once customer has completed created a booking.  C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Database- Bookings.PNGBooking table:  C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Database- Guest.PNGGuest table: |  |
| C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\amend error.PNGo)Checking if an eroor is shown to the user if they are trying to book room which are not available on the given dates when Check Availability is clicked  An error has not come up although the customer is trying to book 10 single rooms on the 15-06-2018 and there are only 3 single rooms available on that date. The database below shows the current bookings. |  |

|  |  |
| --- | --- |
| **Test 5 Update Details** | **Test 6 Admin Login** |
| C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Clicked Update.PNGa)Clicked Update Personal Details from the customer home page or admin home page (test shown below are from a customer user) | a)Same as Customer login page  C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Admin Login.PNG |
| C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Database Customer - email changed.PNGC:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Update - email changed.PNGb)Email fields have been changed and Update Details button clicked. Confirmation of the change is shown.  Database is updated.  Customer Table:  C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Database LoginCredential email changed.PNGLoginCredentials Table: | C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Admin Home.PNGb)After correct login in credentials were enter and login was clicked the admin home page was raised. |

|  |  |
| --- | --- |
| **Test 7 Admin View Booking** | **Test 8 Admin Assign Rooms to Bookings** |
| C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Admin Clicked View Bookings.PNGa)Clicked on View Booking Button on the admin Home Page. After bookings and changes shown above had occurred. | a))Clicked on Assign Bookings Button on the admin Home Page. C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Clicked Admin Assign.PNG |
| C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Bookings which are used for Admin Screen shots.PNGb)Booking from database which are used for all the following test. | C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Admin Assign Select Options.PNGb)Selecting a booking reference and a floor from the drop down arrows. The buttons enabled depends on these options. |
| C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Admin Clicked View Bookings2.PNGc)Clicked on View Booking Button on the admin Home Page. Booking are from database above. | C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Admin Assing- Assigned a room.PNGc)Clicked Assign button for Room 2 on floor one and confirmation is shown. |
|  | C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Admin Assign- View Booking per room.PNGd)Clicked on View Booking Per Room on Assign Booking Page. Booking assigned above is shown when room 1.2 is selected. |
|  | C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Admin Assign - Booked on day part 1.PNGe)Showing booking for room 1.1 for next test shown below |
|  | C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Admin Assign - Booked on day part 2.PNGf)Selecting an unassigned booking and choosing floor 1. Room 1.1 is unavailable to book as it is book for the same date(shown above) |
| C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Admin Clicked View Bookings2.PNGd)Clicked View Bookings from admin home page after rooms have been assigned | C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Database- Booking after assigned rooms.PNGg)Updated database. Room id is changed to the assigned room when a room is assigned to a booking by the admin. |

|  |  |
| --- | --- |
| **Test 9 Customer View Bookings** | **Test 10 Customer Amend Bookings** |
| C:\Users\jade.houghton\OneDrive - Unilever\Uni\Prinicples of Programming\Group Work\Tests\Customer Clicked View Bookings After rooms been assigned.PNGa)Clicked View Bookings on customer home page after rooms have been assigned to the customer bookings. Note Room ID has been changed. | b)Clicked Amend Booking on customer home page after rooms have been assigned to the customer bookings. Note no assigned booking are shown in drop down box. |

## Appendix E

This appendix provides examples of some of the initial planning that we did before producing any code or UML diagrams related to the system. It was at this stage that discussions took place to decide what we have to include in our system, like some of the non-negotiable features in the brief, and additional functionality we should aim to include within our system.

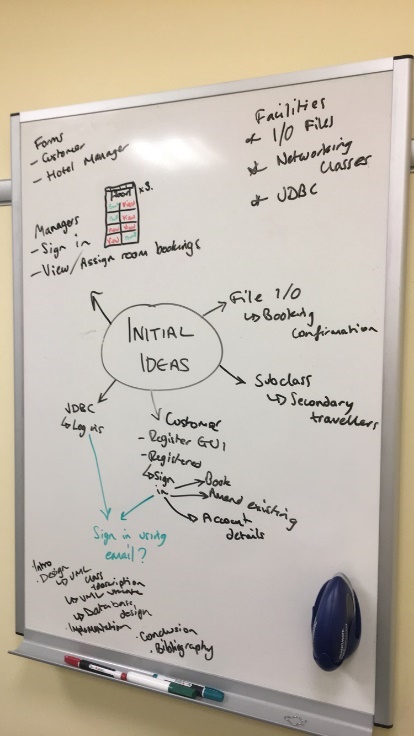


Figure 11: Mindmap of initial ideas of things to be implemented and considered within our system.

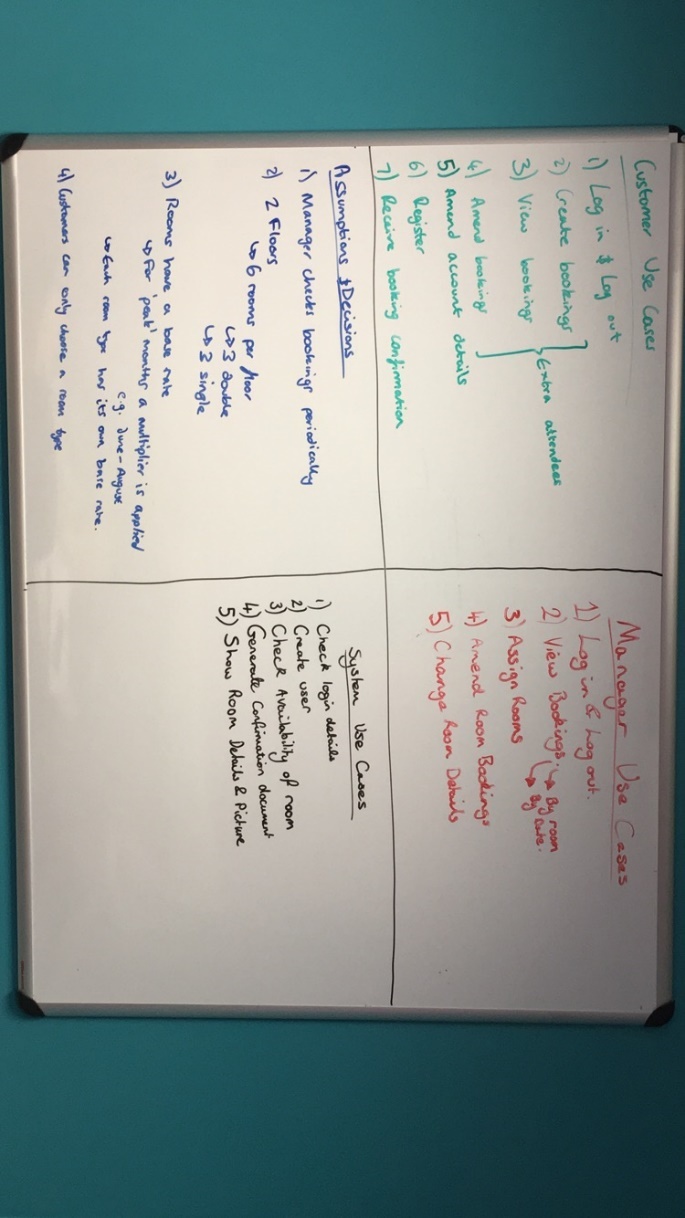


Figure 12: Some initial assumptions and use case ideas we had. Some of these have since been changed.