|  |  |
| --- | --- |
| COMP-1687 Web Application Development | Madalin Cristian Preda 000937119 |

Table of Contents

[1 – PART A 2](#_Toc34677675)

[1.1 Group Work Introduction 2](#_Toc34677676)

[1.2 Entity Relation Diagram (ERD) 4](#_Toc34677677)

[1.3 Architecture Diagram 5](#_Toc34677678)

[1.4 Individual Report 6](#_Toc34677679)

[2 – PART B 7](#_Toc34677680)

[2.1 Screenshots 12](#_Toc34677681)

[2.1.1 Implementation from Part A **Error! Bookmark not defined.**](#_Toc34677682)

[2.1.2 Functionality A 15](#_Toc34677683)

[2.1.3 Functionality B 17](#_Toc34677684)

[2.1.4 Functionality C 22](#_Toc34677685)

[2.1.5 Functionality D 24](#_Toc34677686)

[2.1.6 Functionality E 28](#_Toc34677687)

[2.1.7 Functionality F 29](#_Toc34677688)

# 1 – PART A

## Group Work Introduction

Group work done with Gabriel Ciortea-Pop, 000968052.

For the group was not the first time working together, and pair programming was a developing strategy already adopted in the past. Both members have always been keen to work together when facing new challenges as it speeds up the learning process. When pair programming the quality of the code also increases, and the developed solution becomes more robust. Besides, pair programming enables both team members to have an excellent understanding of the code as it requires engagement from both the driver and navigator.

The group started the assignment by designing the entity relationship diagram (ERD), which enabled the group to define the objects requiring abstract representation. After completing the initial ERD, it was decided not to use the university hosted SQL server as it would give less flexibility but to take advantage of the Microsoft Azure donation for students and to deploy a SQL Server instance on the cloud which would allow us to work from our devices from home or at university comfortably. To create an entity/model, the group decided to scaffold the database and use Entity Framework.

The use of Entity Framework extremely simplified the database management and enabled to quickly be able to develop a desktop application to perform create, read, update and delete (CRUD) operation for the employees. The entity framework scaffold was implemented as a separated project in the solution, and it enabled to require the entity/model on other projects in the same solution by adding a reference and the connection string the config file. This allowed updating the model in one point for all the projects each time the database was altered.

To handle the numerous windows required by the application, the group decided to take advantage of Winforms “User Controls” (UC) which enables to create views inside a single form dynamically. Each UC is controlled by a singleton pattern which restricts the application to only have a single instance of each user control at runtime. The advantage of the approach is to save memory and avoid a large number of user views instances. The approach has also enabled modulation of the front-end. The group has also decided to create two components: a new TextBox for password management, which inherits the properties of the standard TextBox but masks the input and enforces minimum and maximum chars. Also, a dynamic button was created, which was reused for different CRUD operations.

The group acknowledges that it is possible that the database and the design could further change, the trial and failure approach adopted is also a critical element in the learning process and what appeared correct in the initial analysis could turn out not to be ideal.

## Entity Relation Diagram (ERD)

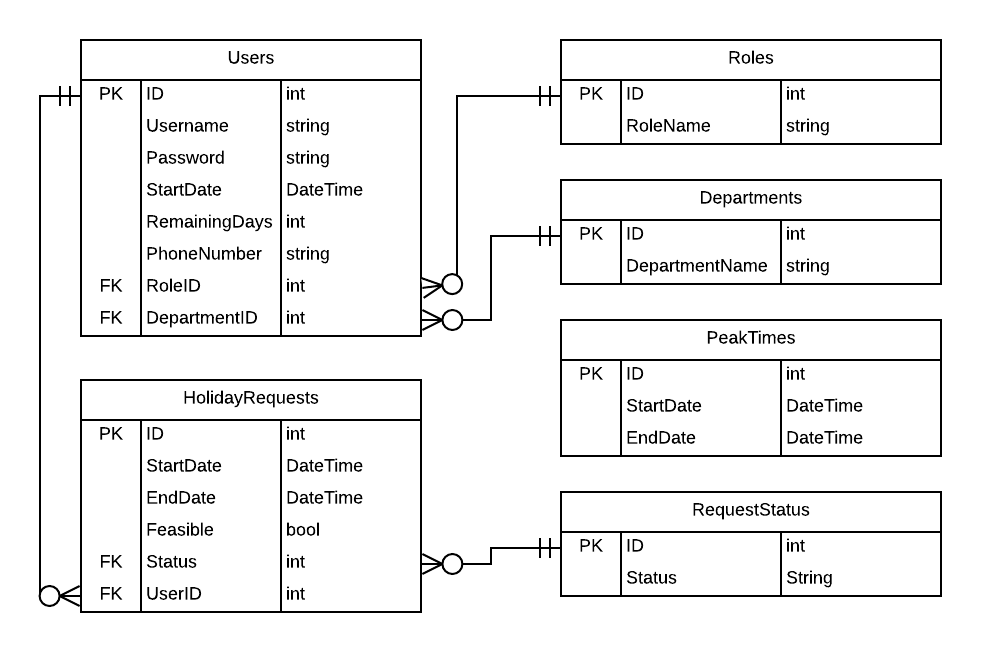


Figure 1 Initial Entity Relationship Diagram (ERD)

The above ERD diagram is the result of the group effort. The diagram was kept as simple as possible, for example users do not have a name or surname, but just their username which is a unique field. Additional details such as age, sex, address are not considered in the scope of this project. The main relationship is between Users and Holiday Requests, while the other linked table are simple look up table to manage a user role and department and the Request status for a Holiday Request.

PeakTimes is a table needed only to record the holiday requests peak time, it has no relationships and the purpose is to allow the admin users to edit the peak periods.

The group agrees that it is possible that the implementation could change while developing part B.

## Architecture Diagram

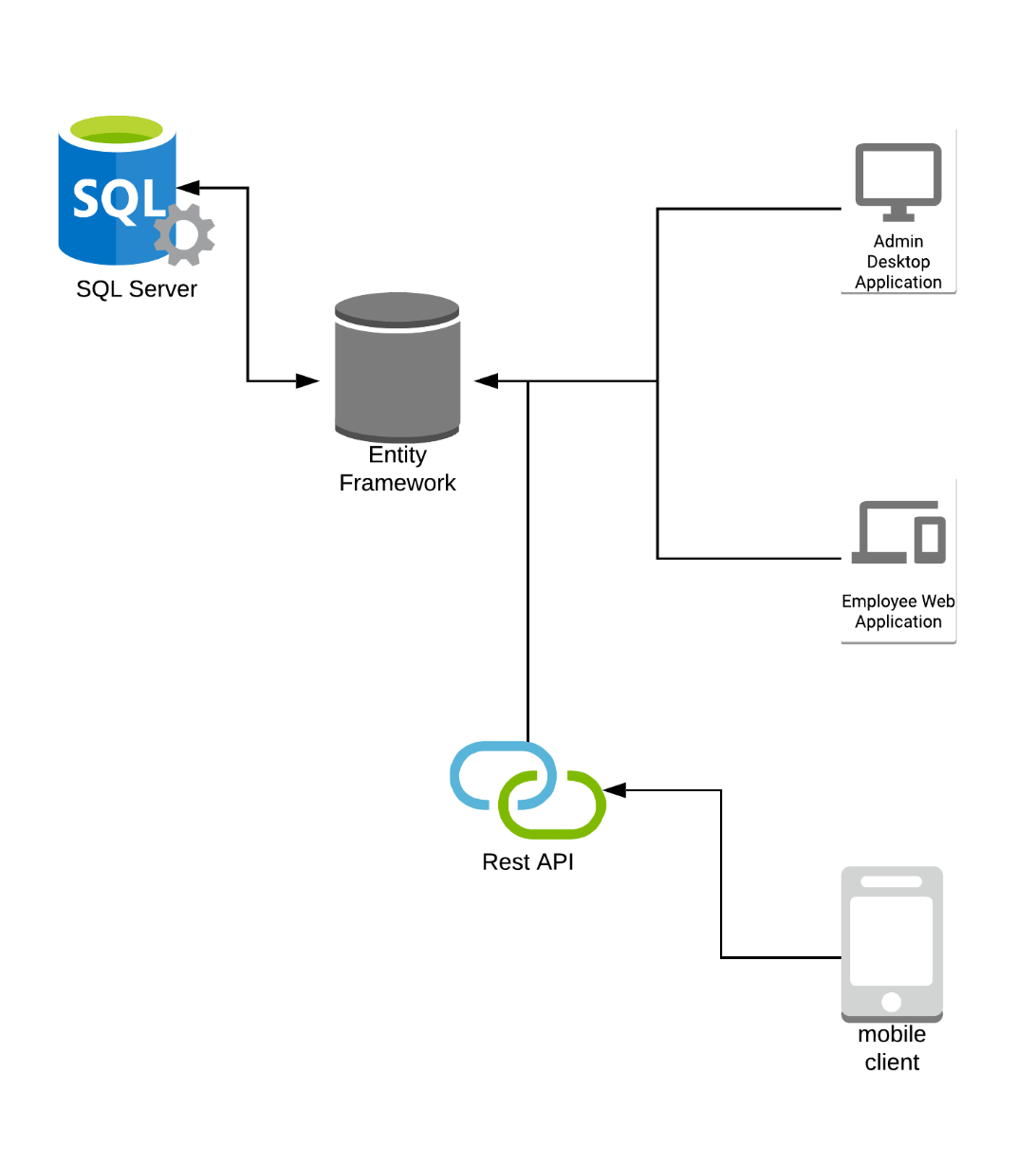


Figure 2 Initial envisioned architecture

The above diagram represents the architecture of the holiday booking system application. There is a SQL Server database stored on the cloud (MS Azure) and all operations are handled by the entity framework projects, which in fact operates as the middleman for all the other components in the diagrams. The Rest APIs will also provide a communication interface for the mobile client(s).

## Individual Report

Gabriel and I worked previously in many different projects since year one, thus each of knew how to each other’s strengths and weaknesses. I strongly believe that we had a great team dynamic and managed to provide a good solution to the required tasks. As we both previously worked with C# we concluded that we still have different skillsets, as Gabriel did projects with windows forms before he clearly did a better job than I could have done on putting the different User Interface(UI) components together, and myself, I tried to add a bit of a styling so that the fonts and colors are nicer.

We worked on everything together, database, backend and frontend, I assissted with the database design and building it. Additionally, I helped at sampling the solution structure together as we managed to scaffold the database into an ADO.NET Entity Model and linked it to both Manager Windows Form Login and Employees ASP.NET Forms web application. Additionally, I created the backend seed queries for populating the database with an admin and some default users, also worked on exception handling, and queries optimization as Gabriel had not use Entity Framework before to the extent I did.

Personally, I believe that our team dynamic did not disappoint, and we did good on pair programming, he was the main person typing and I was the one researching how to do things, supervise and assist him on writing code.

Since PART A did not seem very demanding we did not assign specific roles one to another, we did not discuss anything such as he was going to do the database and the frontend, and I would do the backend. However, if there were more requirements for this part, we would have been better off splitting the work accordingly, in different roles, not everyone working on everything. We mainly worked together on the day when the tutorial is scheduled and remained to carry on working after the tutorial has finished for few more hours.

## Screenshots

### Login

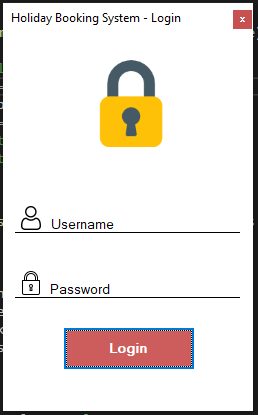


Figure 3 Desktop Login Form

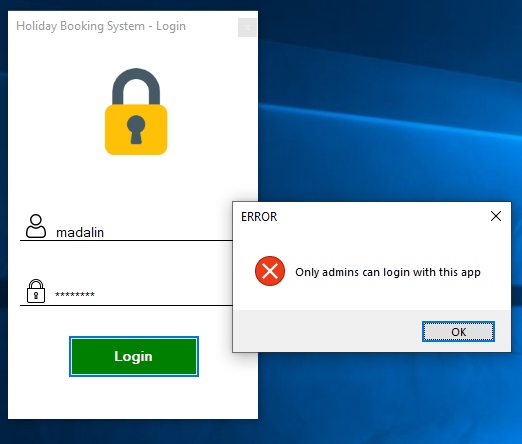


Figure 4 Attempting to sign in with employee account

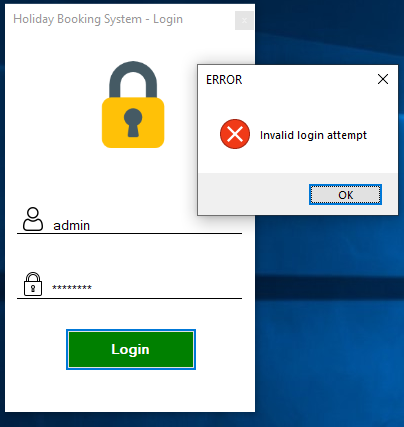


Figure 5 Typing wrong credentials on login

### Add Employee

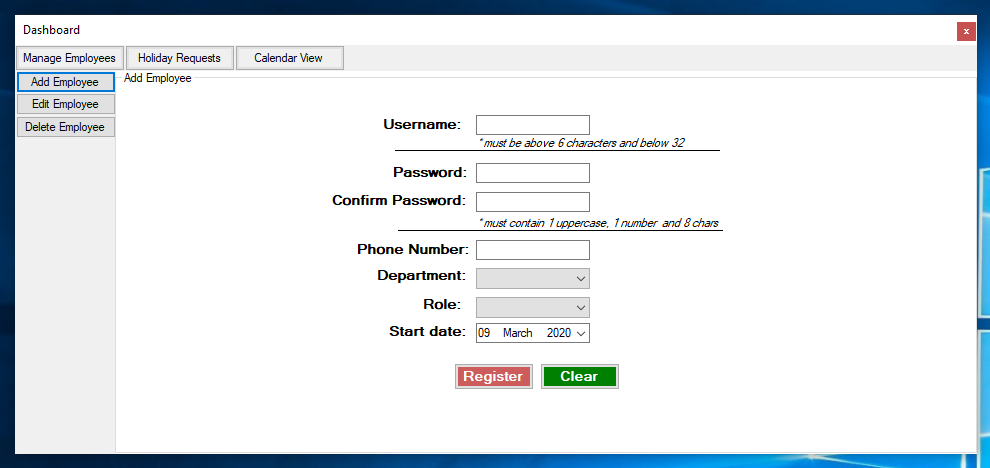


Figure 6 Desktop App Dashboard with Add Employee Form

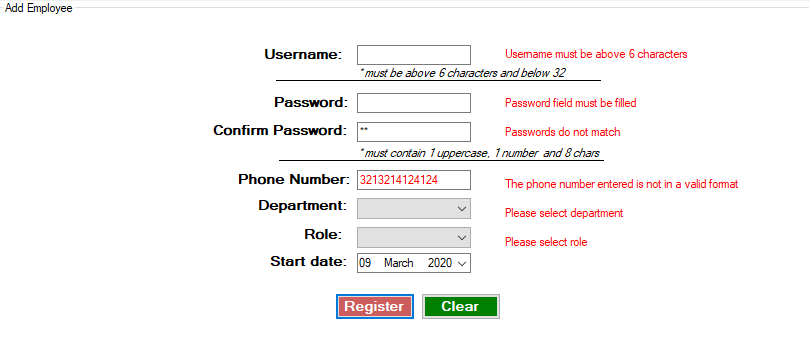


Figure 7 Create Employee Form Inputs Validation

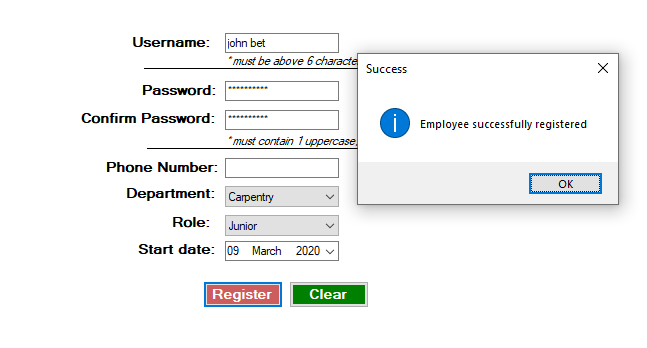


Figure 8 Successful employee registration

### Edit Employee

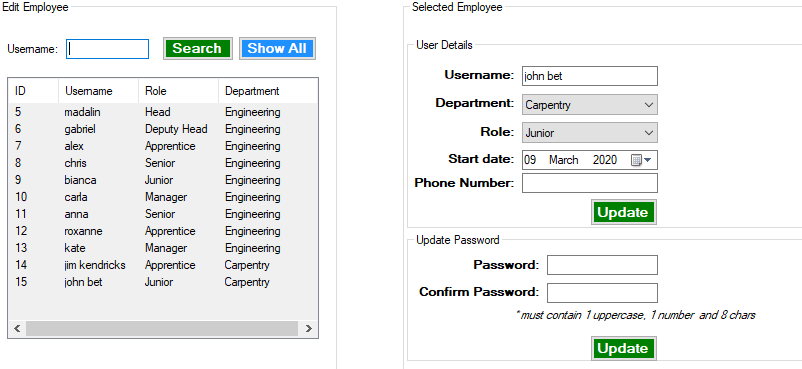


Figure 9 Edit employee details

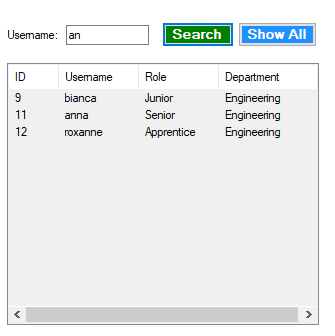


Figure 10 Search employee by Regular Expression match

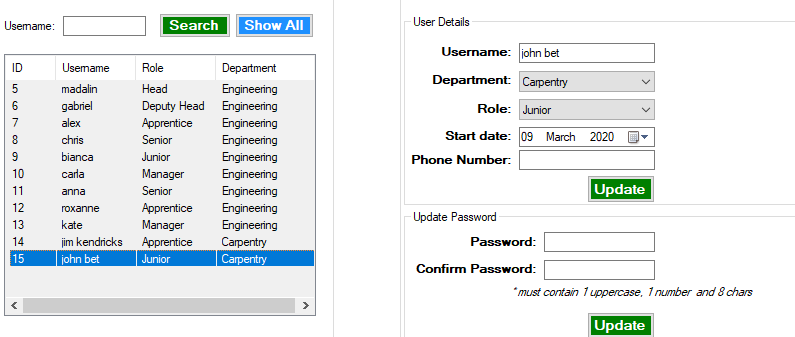


Figure 11 Editing selected employee details

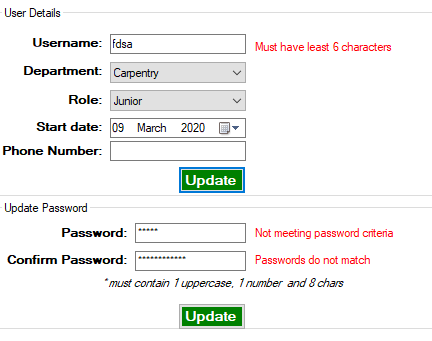


Figure 12 Input validation for editing employee details

### Delete Employee

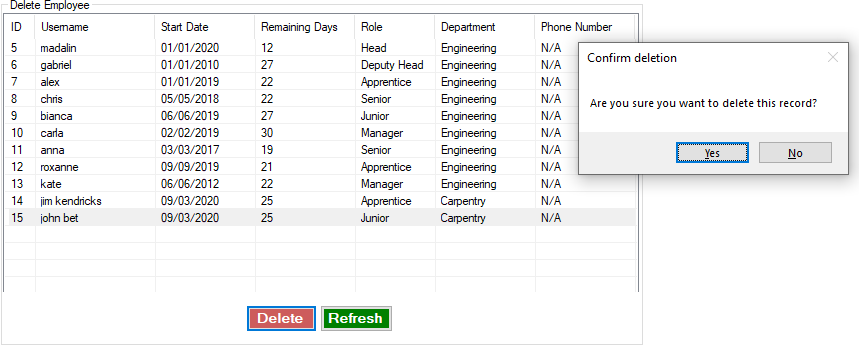


Figure 13 Delete Employee asking for confirmation

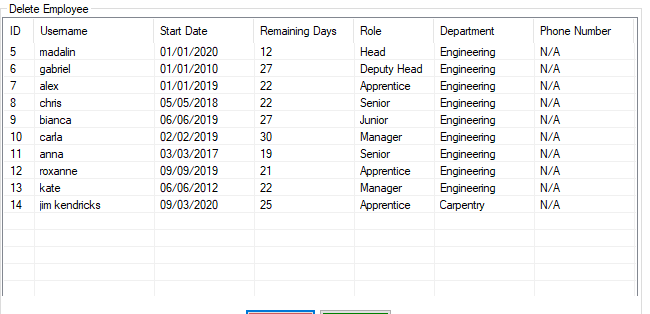


Figure 14 Once employee deleted the list refreshes

# – PART B

## Implementation introduction

During the development process of the individual part of the implementation for this module it became obvious that the previously solution built as a group is not suitable to fulfill all the requirements expected to be accomplished for this coursework as the full functionality was not thoroughly assessed. Therefore, changes have occurred and they have been illustrated in the adapted diagrams that can be found in the following section. There were not major changes to be done, but rather small changes. As it can be seen in the ERD (Figure 19), PeakTimes table was removed, and ConstraintsBroken table was added, and there were also changes in the Users and HolidayRequests tables. Also, the architecture diagram was changed (Figure 20), now the Web Service for employee app resides on the Web application, meaning that they could run together on the same server.

## Final Diagrams

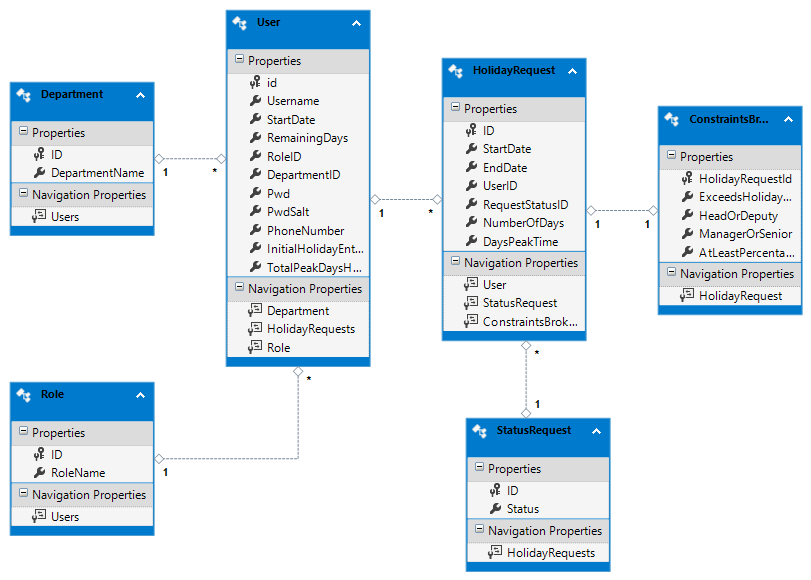


Figure 15 Entity Framework generated model of database

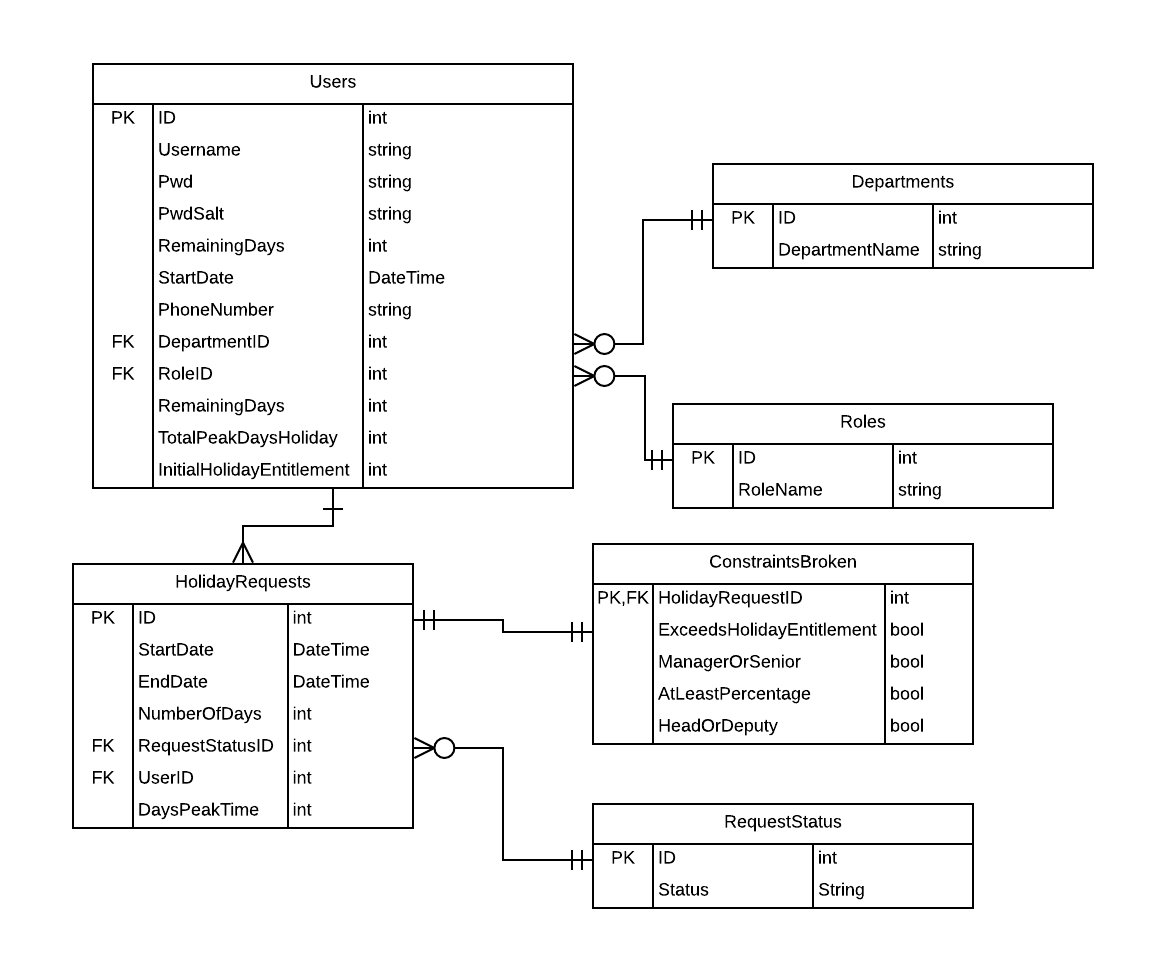


Figure 16 Final Entity Relationship Diagram

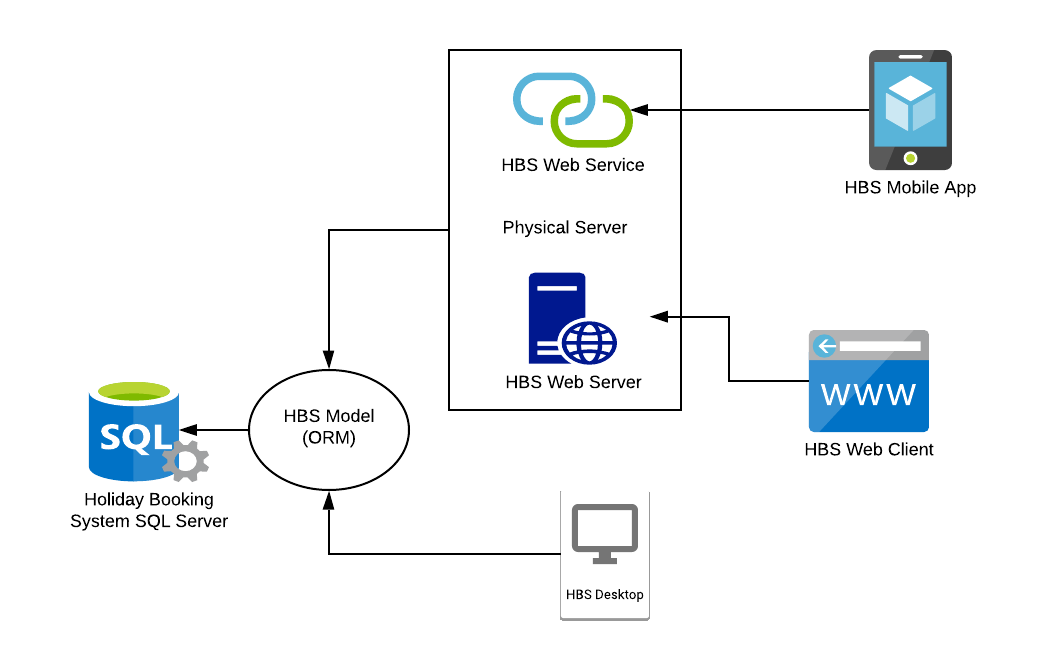


Figure 17 Final Architecture Diagram

## Screenshots

This are screenshots taken from the final version of the coursework.

### Functionality A

#### Outstanding Requests

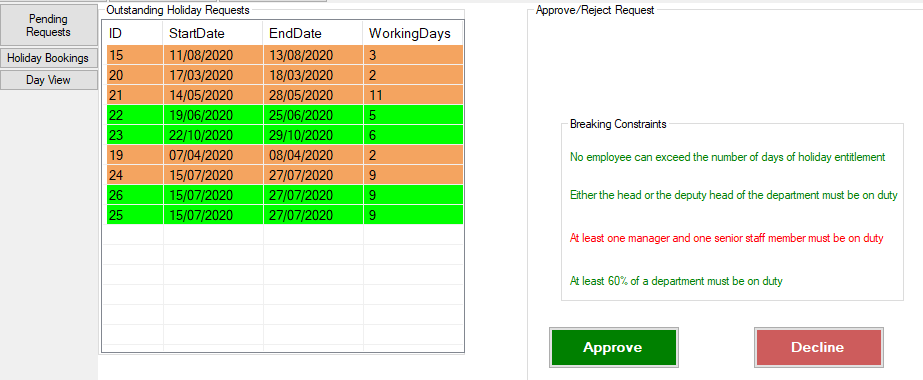


Figure 18 Outstanding Holiday Requests (includes the functionality with prioritization and constraints checking)

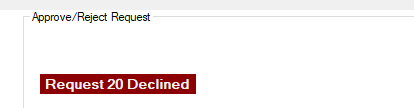


Figure 19 Declining a request

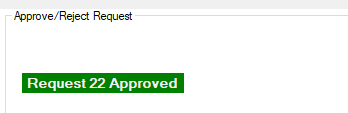


Figure 20 Approving a request

#### All holiday bookings and filter by employee

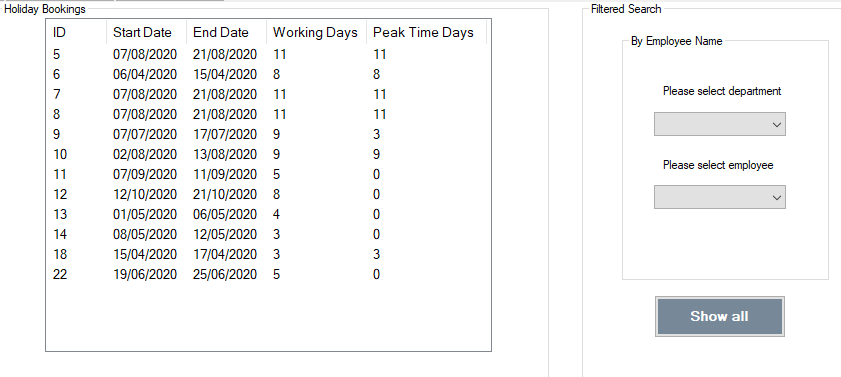


Figure 21 All holiday bookings

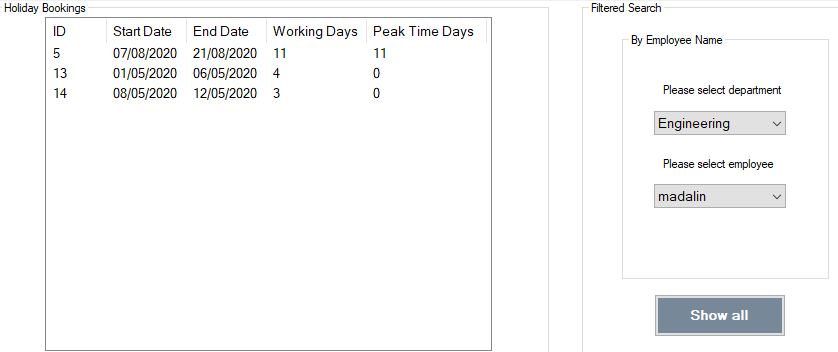


Figure 22 Holiday Requests filtered by employee

#### Employees working and on leave on a selected date

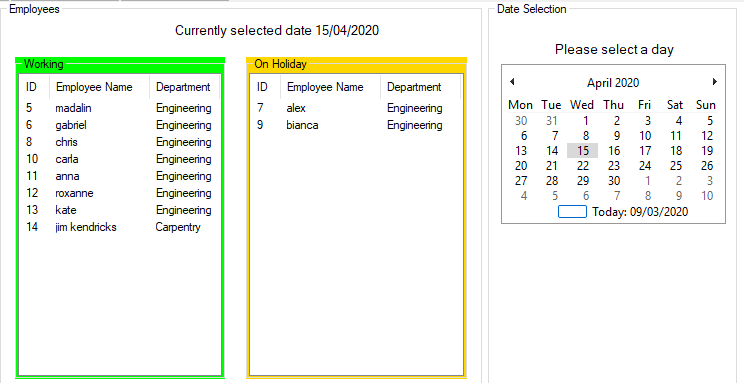


Figure 23 Employees working and employees on holiday

### Functionality B

#### Phone Text Box Component



Figure 24 Text becomes red if invalid input and only numbers allowed

#### ASP.NET Web forms Login

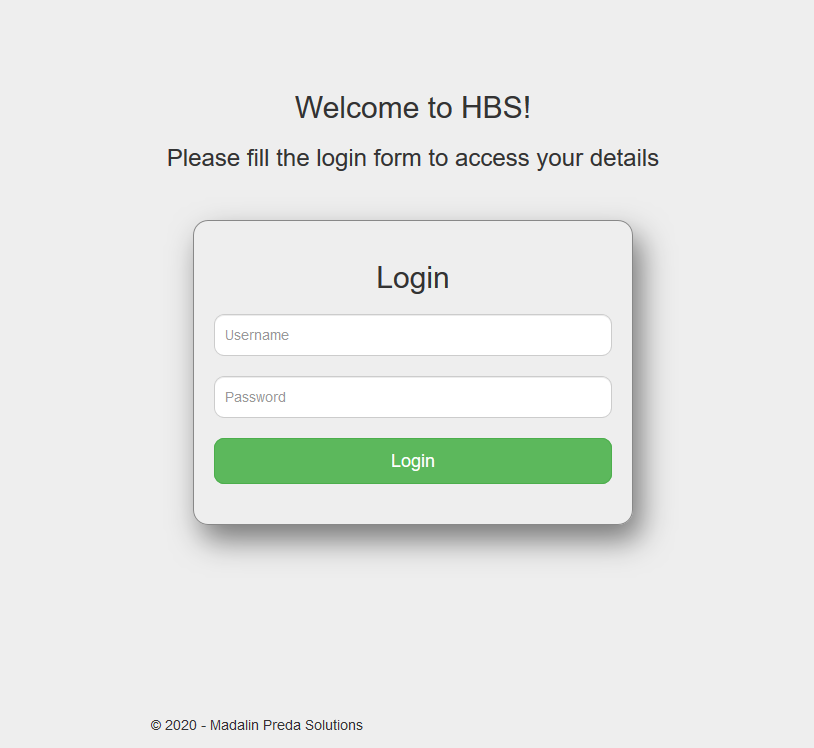


Figure 25 Login Form

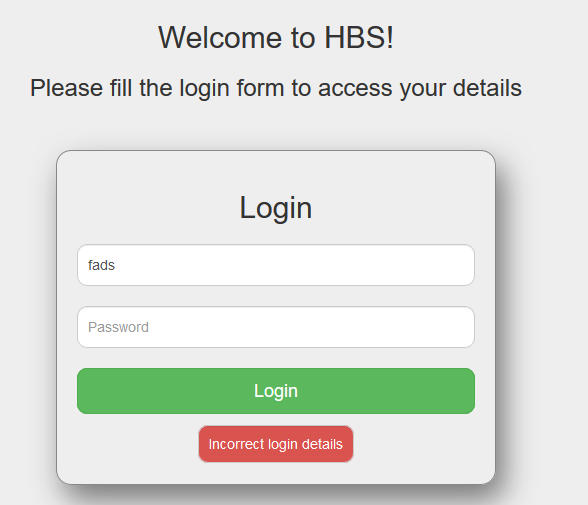


Figure 26 Failed login

#### Submit holiday request

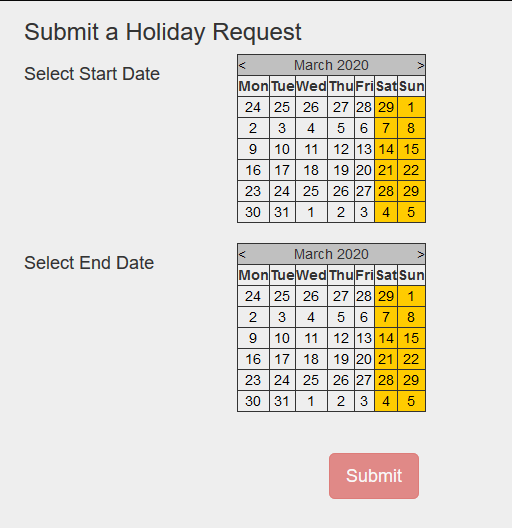


Figure 27 Submit Holiday Request

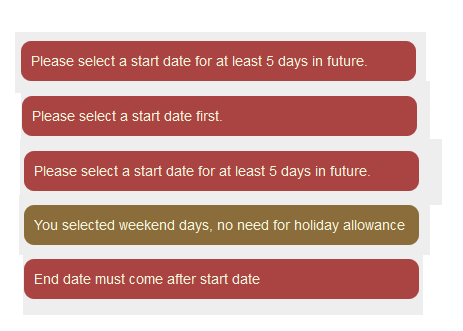


Figure 28 Types of error messages

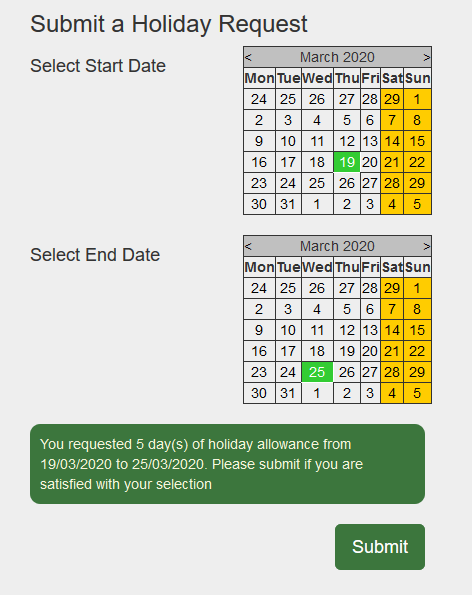


Figure 29 Valid holiday request submission

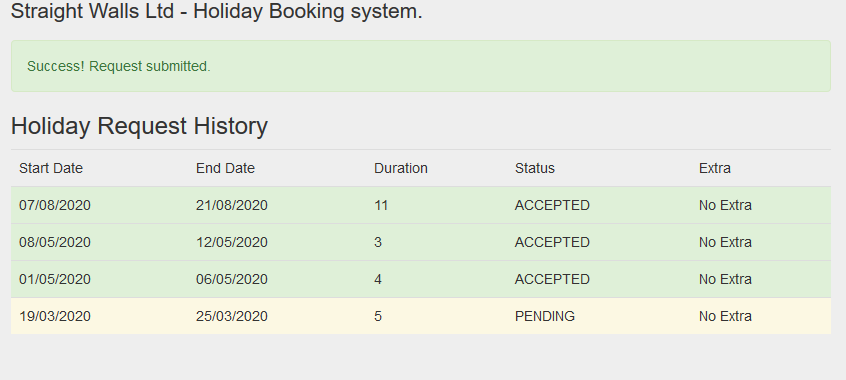


Figure 30 Confirmation request submitted

#### List of employee holiday requests and their status

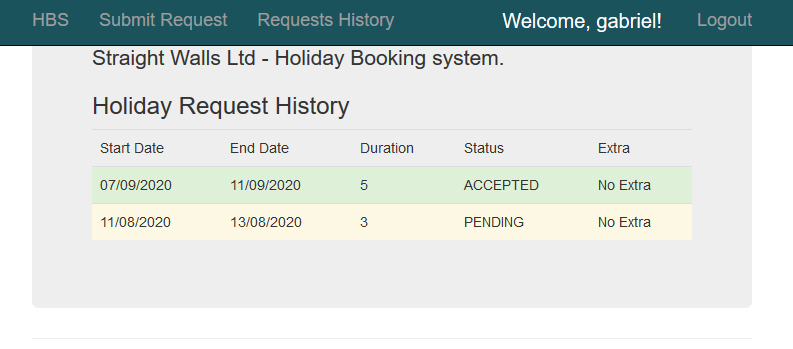


Figure 31 List of employee's holiday requests

### Functionality C

#### Constraint Checking Errors Display

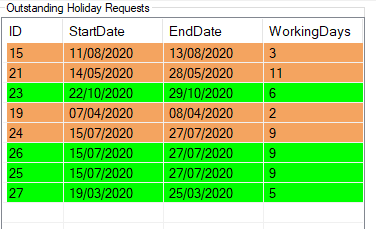


Figure 32 Requests breaking constraints (red), valid requests (green)

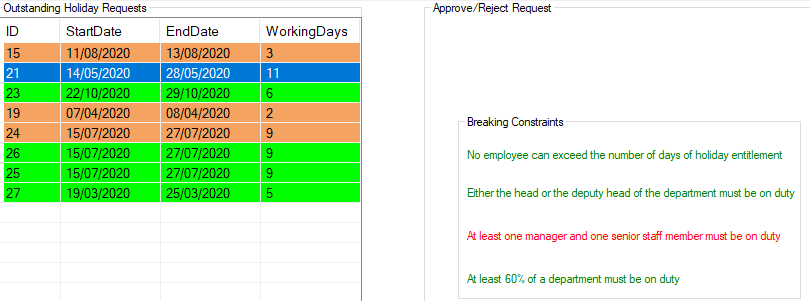


Figure 33 Constraints Broken Highlights

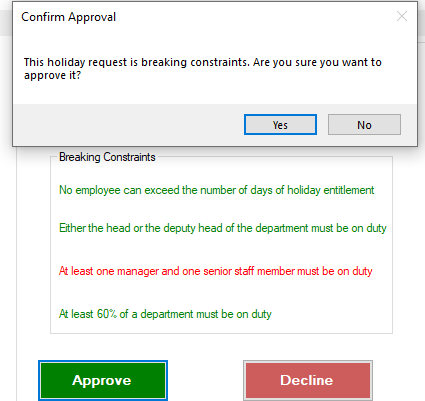


Figure 34 Approving holidays that break constraints

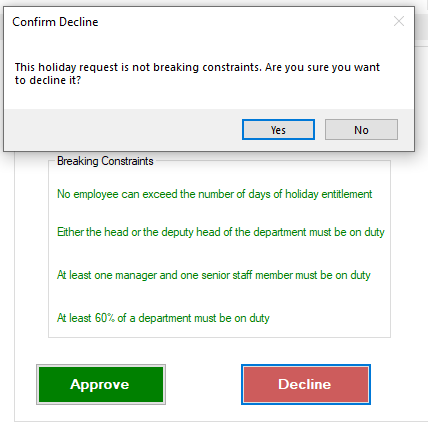


Figure 35 Declining holidays that do not break constraints

### Functionality D

#### Web Service

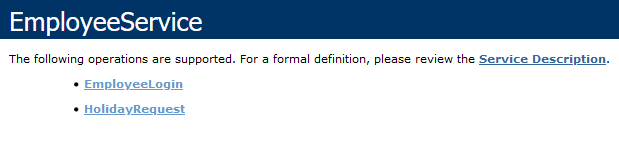


Figure 36 SOAP Web Service exposed methods

#### Mobile App Employee Login

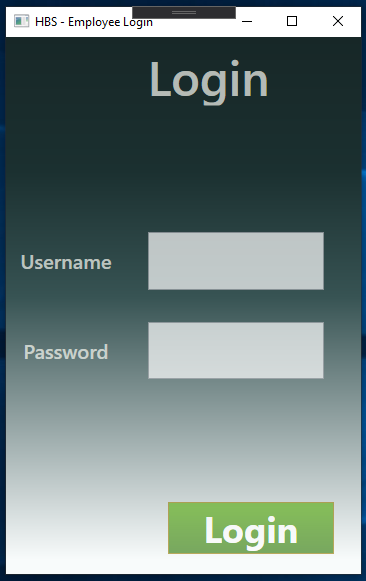


Figure 37 Login form employee app

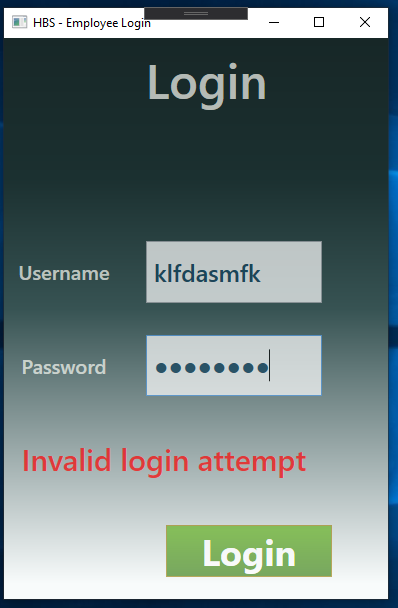


Figure 38 Attempt to login with wrong credentials

#### Employee app make holiday request

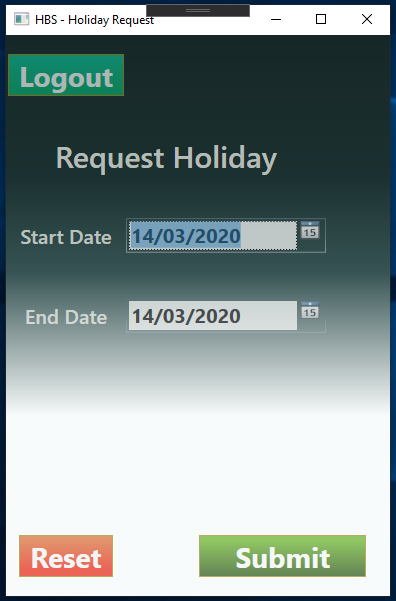


Figure 39 Submit holiday request

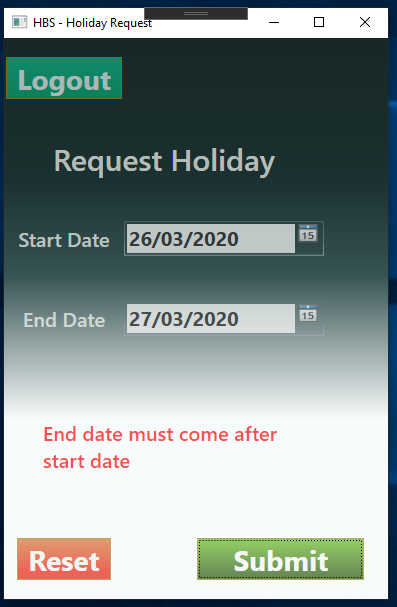


Figure 40 Making an invalid request

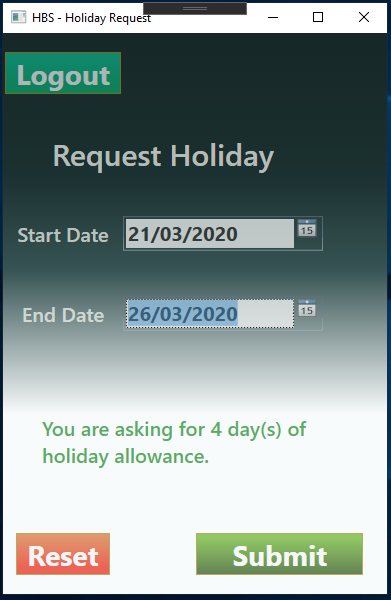


Figure 41 App checks how many days user asked for

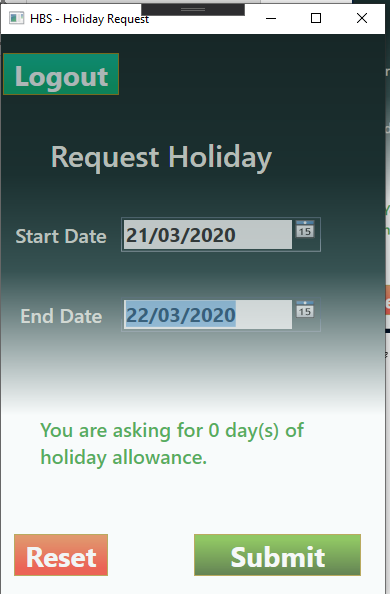


Figure 42 App lets user know if they select weekend days

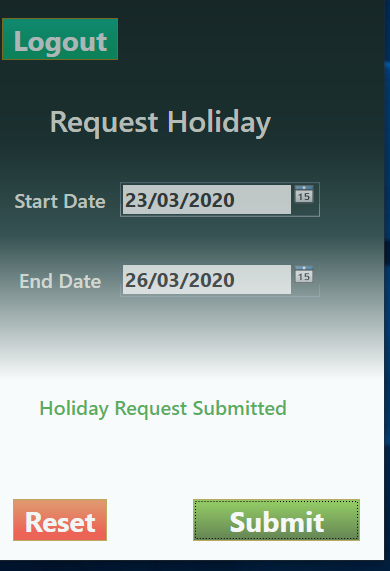


Figure 43 Feedback on request submission

### Functionality E

#### Prioritization of Holiday Requests

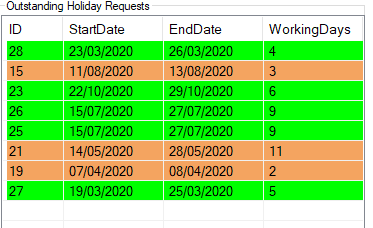


Figure 44 Rows are ordered by priority

#### Suggestions of Suitable Holiday

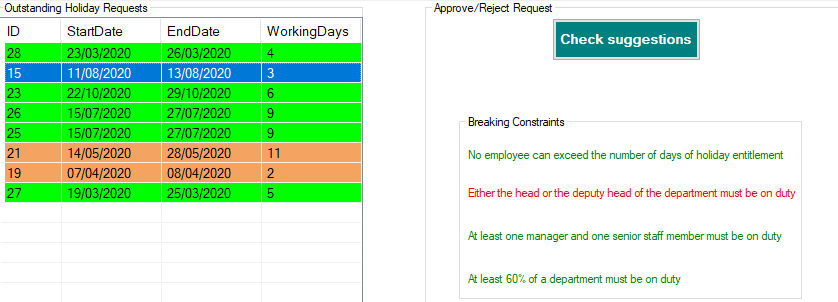


Figure 45 Suggestions button shows if request falls during peak times

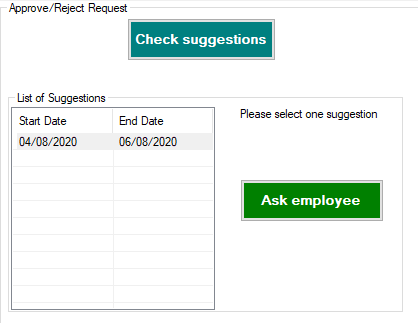


Figure 46 If suggestions available a list is returned

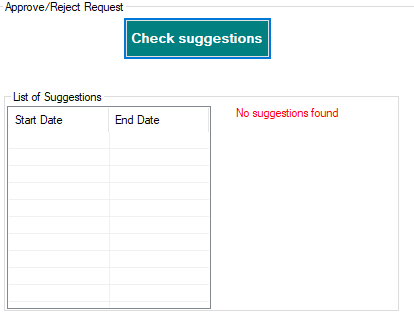


Figure 47 If no suggestions available

### Functionality F

#### Custom Holiday Bookings Visualization Component

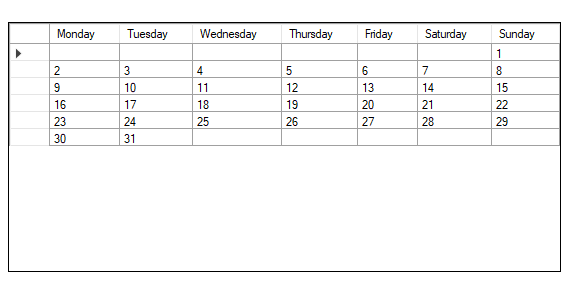


Figure 48 Calendar Visualization Component

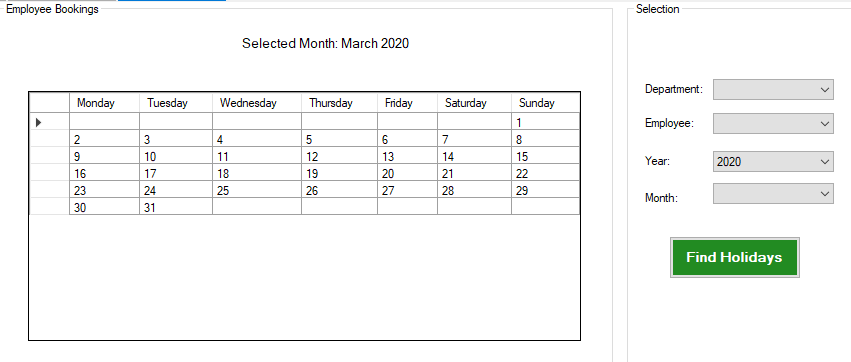


Figure 49 User Control integrating the component

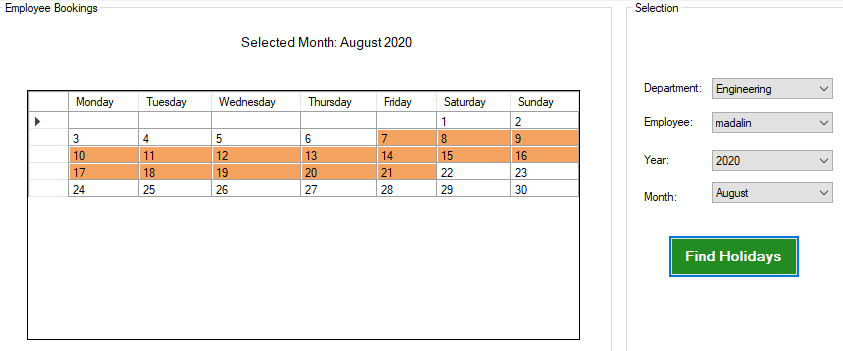


Figure 50 Displaying Days employee is on holiday

# Evaluation

Before starting working on the coursework it seemed a nearly impossible task to finish all the requirements because we had to develop various applications (Web, Desktop and Mobile Applications) that do not entirely behave the same, especially when it comes to User Interface (UI) design and components control. Although the tools that come together with ASP.NET looked similarly on the surface, such as Visual components like TextBox, Button and others, they would have few differences when discussing matters such as properties and exposed events, this has been observed by comparing those elements from Windows Forms to Web Forms to Windows Presentation Foundation pages. Such difficulties required more time to be spend on understanding the way things work within each of the app building formats mentioned previously so that implementations can benefit from similar quality of User Interface, Input checks and validation, and error handling.

Additional problems emerged from the use of Entity Framework and integrating the Web Service with the mobile application. Generating an ADO.NET Data model from a database it is quite a simple task when using Entity Framework to take care of it. However, when it comes to changes that take place within the database and the model is required to update, it will suffer at times, in particular when a table is deleted or a column is changed, in such case scenarios there are times when errors occur and is hard to discover how to fix those problems, few times the model was deleted and completely scaffolded again, and this used to fix everything. The SOAP (Simple Object Access Protocol) Web Service is another example of struggles that I encountered, first time I created a separated project and added it there, but there seemed to be an issue when reopening the solution, such that the mobile app should run a command to update the service reference before actually running the app. Moreover, when the Web Service was moved to reside inside the Web application, the session used with the Web Service started malfunctioning so that once user logged in the session ID changes when making the second request from the mobile app. Nevertheless, a temporary solution was found so that user credentials are stored in the mobile app and sent together with other data for submitting a holiday request, even though SSL (Secure Sockets Layer) is enabled on the Web Service, it still represents a risk. Preventing the exposure mentioned before can be done by using a token (JSON Web Token is quite popular) and passed with data on every request, that could have been accomplished if there was more time to work on it.

The solution is meant to provide high quality code as subsequent components were created to have high cohesion and low coupling as it is quite important in software design for object-oriented programming. Although, some classes might depend a bit more on others, in some cases polymorphism was used to reduce coupling, so that some dependencies can be at times avoided, for example the case of using classes like HolidayRequests when only StartDate and EndDate are needed can be avoided, by using the example of DateRange which can store those two values and avoid carrying Navigation Properties from Entity Framework and other irrelevant data. However, it was not always the case that this has been attempted, thus some refactoring might decrease coupling between components.

There were some additions that were not required, but I decided they might be useful such that both, web application and mobile app allow users to log out. Additionally, the mobile app was designed with Windows Presentation Foundation pages which come in XAML (Extensible Application Markup Language) format which describe user interface elements for software applications designed for Windows Phone apps, meaning that they could potentially be used for actual phones already, the coursework was merely asking for a desktop app prototype. Moreover, the implementation provides individual helper class Desktop app, which is called DesktopAppUtils, and is mainly used for seeding the database if no data records are found, so that testing can be done with some fictional data, the later only allows generating alerts. Furthermore, a SolutionUtils project was created and contains some of the components required as part of functionalities for PART B, however there is an additional class called GeneralUtils which is a static type, designed to provide typical functions that need to be performed across the different projects created and provide with Constants/Variables so that very little hardcoded values are used within the solution, thus when in need of changing a role name, department name or boundaries for some conditions it can be done from within this class.

The naming conventions used were mainly similar to JavaScript camelCase notation for methods and variables, although C# seems to have use a different naming convention for those, nevertheless classes and namespaces were always named starting with uppercase letter. Sensible naming standards were used throughout and comments added were things might be confusing. Adding code on top the existing one should be quite easy, and changing the data source should not represent a problem as all the projects are bound to the same object meaning that a change it there should be reflected in all three, Desktop app, Web app and Web service.

One of the downsides of my solution is that it lacks of unit testing, although a lot of white box testing was performed, this would never be enough, therefore unit testing should be a must. Another disadvantage might be that it technically works only for the current year to make holiday requests, thus this is clearly a limitation, if more time was available the system would have been designed so that it would overcome this impediment, thus moving to a new year will mean resetting holiday request allowance manually for now. Also, for making suggestions the assumption was that they will only be available for users breaking constraints during peak times, and once a suggestion seems feasible to the administrator, it is reflected for the specific employee on the Web application side, however he cannot accept, nor decline the suggestion for now, it could be implemented if more time was given. Furthermore, constraint checking should also be done every time when outstanding holidays are viewed by the administrator in the desktop app because if a request is accepted, then the next one selected my break a constraint if it happened that the two interfered by having intercalating dates span.

# Algorithm

The constraint checking implementation has been done by creating a class called ConstraintChecking which resides inside SolutionUtils project since it is needed to be used in three places, Desktop Application, Web Service and Web forms. The component exposes few methods, amongst which one is a static method that allows checking if a ConstraintsBroken class contains any broken constraints. The constructor created expects two parameters which are User and HolidayRequest types such that when the object is created, all the users from the given department of the user passed as an argument are fetched from database. The component allows changing the holiday request that is being evaluated and there are two public methods available for checking if any constraints are broken, getBrokenCosntraints() returning an object containing constraints and if there are rules broken, the other called isItBreakingConstraints() which returns whether any constraint is broken or not.

The most important method, that resides at the base of most algorithms that requires checks of whether two date spans (example: 03/11/2011-07/12/2011 and 06/11/2011-10/11/2011) have any intersecting dates in between, is called isOverlappingDateRanges() (also, isOverlappingHolidayRequests() for HolidayRequests arguments type) and since the checks are always done for possible holiday bookings, meaning that they should be situated somewhere in the future, this is one of the first checks done within the function. The algorithm is shown down below (Figure 51).

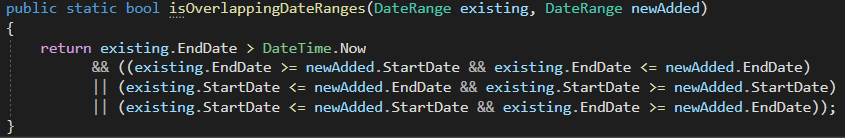


Figure 51 Overlapping date ranges

The algorithm written for the purpose of prioritizing holiday requests makes use of a custom class called PriorityRequest which stores data related to a HoldayRequests, Users and ConstraintsBroken tables from the database. This custom class implements the IComparable interfaces so that a custom sorting can be implemented for this class, thus some conditions were written to meet the prioritisation criteria required within the coursework specification. PriorityRequest makes use of another class called BreakingConstraints which contains same properties as ConstraintsBroken class which is part of the Database Model generated by Entity Framework, but I decided not to use it so as to reduce the dependencies. The component that takes care of requests prioritisation (PrioritiseRequests class) exposes two methods, one that returns the list of holiday requests in a prioritized order and a second method, daysFallPeakTimesCount, which returns the number of days that fall during peak time periods for a given range of dates. The conditions for prioritisation can be seen in Figure 52.

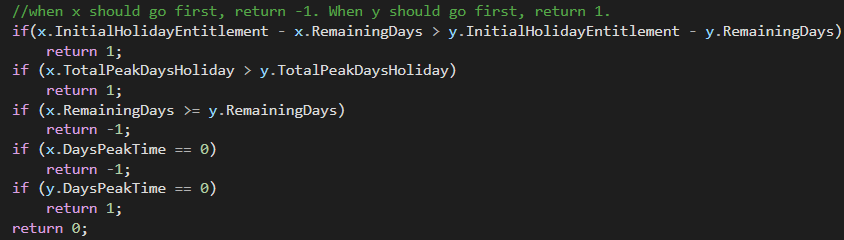


Figure 52 Conditions for prioritization

The component used to find suggestions for a holiday request that breaks constraints exposes only one method which return a list of DateRange objects. The algorithm is designed so as to make suggestion checking within a given range of the currently analyzed holiday request, however the limit can be changed for both ends by editing the value of the constant SUGGESTIONS\_MAX\_DAYS\_BOUNDARY that can be found in GeneralUtils class. Before starting the calculus for possible suggestions, the algorithm checks if the broken constraint is the one regarding the Holiday Entitlement Exceeded, if that is the case then the holiday length is reduced to the number of days left from holiday entitlement and suggestions are made based on a new holiday end date calculated from the difference between the holiday entitlement left and the number of days requested initially. There are two methods that make suggestions by keeping the same length of the initial holiday request, calculateSuggestionsBeforeStartDate and calculateSuggestionsBeforeEndDate, which attempt to make suggestions by equally moving the given number of days before the selected start date, respectively after the selected end date. Moreover, there are two other methods that perform calculus for suggestions by gradually reducing the number of days to half, calculateSuggestionsByReducingDaysFromStart and calculateSuggestionsByReducingDaysFromEnd, by using the same start date, respectively the same end date. All the suggestions are done by checking whether any constraints are broken before adding it to the list of suggestions, therefore any suggestion will not break any constraint. Also, a limit can be set to the number of returned suggestions by editing the value of the constant field MAX\_SUGGESTIONS\_COUNT which can be found inside GeneralUtils class.