|  |
| --- |
| Burford School – 62225 |
| Room Booking System |
| Keith Collister – 9164 |



|  |
| --- |
|  |

Contents

[Analysis 6](#_Toc448432939)

[Client 6](#_Toc448432940)

[Background, Description of Current System 6](#_Toc448432941)

[Benefits of introducing a new system 6](#_Toc448432942)

[Fact Finding Methods 7](#_Toc448432943)

[Interview with Mr Wilsdon, Client 7](#_Toc448432944)

[Paperwork from current system 8](#_Toc448432945)

[Observation of current system 8](#_Toc448432946)

[Questionnaires 12](#_Toc448432947)

[Identification of Users 13](#_Toc448432948)

[Data Sources/Destinations – Existing System 14](#_Toc448432949)

[Data Dictionaries – Existing System 14](#_Toc448432950)

[Teacher 14](#_Toc448432951)

[Room 14](#_Toc448432952)

[Department 14](#_Toc448432953)

[Booking 15](#_Toc448432954)

[Entity-Relationship Diagram – Existing 15](#_Toc448432955)

[Data Flow Diagram – Current System 15](#_Toc448432956)

[Potential Solutions 16](#_Toc448432957)

[Appraisal of Potential Solutions 16](#_Toc448432958)

[Justification of Chosen Solution (Client-Server) 17](#_Toc448432959)

[Limitations 17](#_Toc448432960)

[Objectives 18](#_Toc448432961)

[Data Sources/Destinations – Proposed Solution 19](#_Toc448432962)

[Data Volumes 20](#_Toc448432963)

[Entity-Relationship Diagram – Proposed 20](#_Toc448432964)

[Data Flow Diagram – Proposed 21](#_Toc448432965)

[Design 22](#_Toc448432966)

[Overall System Design 22](#_Toc448432967)

[Description of Modular Structure 22](#_Toc448432968)

[Client 22](#_Toc448432969)

[Server 22](#_Toc448432970)

[NetCore 22](#_Toc448432971)

[Data 23](#_Toc448432972)

[Shared 23](#_Toc448432973)

[Data Requirements/Record Structure/Database Design 23](#_Toc448432974)

[Student 24](#_Toc448432975)

[Teacher 24](#_Toc448432976)

[Room 24](#_Toc448432977)

[Class 25](#_Toc448432978)

[Department 25](#_Toc448432979)

[Period 25](#_Toc448432980)

[Booking 25](#_Toc448432981)

[Subject 25](#_Toc448432982)

[Validation Required 26](#_Toc448432983)

[Processing and Algorithms 26](#_Toc448432984)

[Client 26](#_Toc448432985)

[Server 27](#_Toc448432986)

[NetCore 27](#_Toc448432987)

[Network Protocol Design 28](#_Toc448432988)

[Database Queries 28](#_Toc448432989)

[Storage Material and Format 29](#_Toc448432990)

[Human-Computer Interface 29](#_Toc448432991)

[Design Factors 29](#_Toc448432992)

[Timetable Window 30](#_Toc448432993)

[Admin Controls Window 30](#_Toc448432994)

[EditBooking Window 32](#_Toc448432995)

[Edit Student Window 32](#_Toc448432996)

[Edit Department Window 33](#_Toc448432997)

[Edit Teacher Window 33](#_Toc448432998)

[Connection Lost Error Dialog 34](#_Toc448432999)

[Sample of Data Entry 34](#_Toc448433000)

[Sample of Valid Output 34](#_Toc448433001)

[Security & Integrity 34](#_Toc448433002)

[System Security 34](#_Toc448433003)

[Test Plan 35](#_Toc448433004)

[Testing 36](#_Toc448433005)

[Test Plan 36](#_Toc448433006)

[Test Results - Evidence 49](#_Toc448433007)

[Minimum Test Data 62](#_Toc448433008)

[Departments 62](#_Toc448433009)

[Rooms 62](#_Toc448433010)

[Teachers 62](#_Toc448433011)

[Subjects 62](#_Toc448433012)

[Periods 63](#_Toc448433013)

[Students 63](#_Toc448433014)

[Classes 63](#_Toc448433015)

[Bookings 63](#_Toc448433016)

[System Maintenance 64](#_Toc448433017)

[Introduction 64](#_Toc448433018)

[Description of Modular System 64](#_Toc448433019)

[Annotated Source Code 64](#_Toc448433020)

[Shared 65](#_Toc448433021)

[Shared.Writer 66](#_Toc448433022)

[Shared.TextWriter 67](#_Toc448433023)

[Shared.NetWriter 68](#_Toc448433024)

[Shared.Reader 69](#_Toc448433025)

[Shared.TextReader 70](#_Toc448433026)

[Shared.NetReader 71](#_Toc448433027)

[Shared.Helpers 74](#_Toc448433028)

[Shared.ISerialisable 75](#_Toc448433029)

[Data 76](#_Toc448433030)

[Data.IDataRepository 78](#_Toc448433031)

[Data.DataSnapshot 79](#_Toc448433032)

[Data.Models.DataModel 81](#_Toc448433033)

[Data.Models.User 83](#_Toc448433034)

[Data.Models.Student 85](#_Toc448433035)

[Data.Models.Teacher 87](#_Toc448433036)

[Data.Models.Department 89](#_Toc448433037)

[Data.Models.Room 91](#_Toc448433038)

[Data.Models.Class 94](#_Toc448433039)

[Data.Models.Subject 96](#_Toc448433040)

[Data.Models.TimeSlot 98](#_Toc448433041)

[Data.Models.Booking 101](#_Toc448433042)

[NetCore 105](#_Toc448433043)

[NetCore.BlockingQueue 106](#_Toc448433044)

[NetCore.Messages.Message 107](#_Toc448433045)

[NetCore.Messages.TestMessage 109](#_Toc448433046)

[NetCore.Messages.ConnectMessage 110](#_Toc448433047)

[NetCore.Messages.DisconnectMessage 111](#_Toc448433048)

[NetCore.Messages.DataMessages.InitialiseMessage 112](#_Toc448433049)

[NetCore.Messages.DataMessages.UserInformationMessage 113](#_Toc448433050)

[NetCore.Messages.DataMessage.DataMessage 114](#_Toc448433051)

[NetCore.Server.Client 115](#_Toc448433052)

[NetCore.Server.Listener 118](#_Toc448433053)

[NetCore.Client.Connection 122](#_Toc448433054)

[Server 125](#_Toc448433055)

[Server.MailHelper 126](#_Toc448433056)

[Server.DataRepository 127](#_Toc448433057)

[Server.Settings 129](#_Toc448433058)

[Server.Program 130](#_Toc448433059)

[Client 134](#_Toc448433060)

[Client.DataRepository 136](#_Toc448433061)

[Client.Settings 143](#_Toc448433062)

[Client.Checkable 145](#_Toc448433063)

[Client.Converters.BookingTypeToStringConverter 146](#_Toc448433064)

[Client.Converters.BooleanToVisibilityConverter 147](#_Toc448433065)

[Client.Converters.IntToStringConverter 148](#_Toc448433066)

[Client.Converters.InverseBooleanConverter 149](#_Toc448433067)

[Client.Converters.InverseNullableBooleanConverter 150](#_Toc448433068)

[Client.Converters.InverseVisibilityConverter 151](#_Toc448433069)

[Client.Converters.NullableToBoolConverter 152](#_Toc448433070)

[Client.Converters.NullableToVisibilityConverter 153](#_Toc448433071)

[Client.Converters.StringToIntConverter 154](#_Toc448433072)

[Client.TimetableDisplay.TimetableTile (Design) 155](#_Toc448433073)

[Client.TimetableDisplay.TimetableTile (Code-behind) 156](#_Toc448433074)

[Client.TimetableDisplay.TimetableDisplay (Design) 159](#_Toc448433075)

[Client.TimetableDisplay.TimetableDisplay (Code-behind) 160](#_Toc448433076)

[Client.EditWindows.EditWindow 163](#_Toc448433077)

[Client.EditWindows.StudentSelector (Design) 164](#_Toc448433078)

[Client.EditWindows.StudentSelector (Code-behind) 165](#_Toc448433079)

[Client.EditWindows.EditBooking (Design) 167](#_Toc448433080)

[Client.EditWindows.EditBooking (Code-behind) 169](#_Toc448433081)

[Client.EditWindows.EditClass (Design) 172](#_Toc448433082)

[Client.EditWindows.EditClass (Code-behind) 173](#_Toc448433083)

[Client.EditWindows.EditDepartment (Design) 175](#_Toc448433084)

[Client.EditWindows.EditDepartment (Code-behind) 177](#_Toc448433085)

[Client.EditWindows.EditPeriod (Design) 179](#_Toc448433086)

[Client.EditWindows.EditPeriod (Code-behind) 180](#_Toc448433087)

[Client.EditWindows.EditRoom (Design) 182](#_Toc448433088)

[Client.EditWindows.EditRoom (Code-behind) 184](#_Toc448433089)

[Client.EditWindows.EditStudent (Design) 187](#_Toc448433090)

[Client.EditWindows.EditStudent (Code-behind) 189](#_Toc448433091)

[Client.EditWindows.EditTeacher (Design) 192](#_Toc448433092)

[Client.EditWindows.EditTeacher (Code-behind) 194](#_Toc448433093)

[Client.EditWindows.EditSubject (Design) 197](#_Toc448433094)

[Client.EditWindows.EditSubject 199](#_Toc448433095)

[Client.Admin.AdminWindow (Design) 202](#_Toc448433096)

[Client.Admin.AdminWindow (Code-behind) 208](#_Toc448433097)

[Client.MainWindow (Design) 213](#_Toc448433098)

[Client.MainWindow (Code-behind) 214](#_Toc448433099)

[Client.TrayIcon (Design) 217](#_Toc448433100)

[Client.TrayIcon (Code-behind) 218](#_Toc448433101)

[Client.App (Design) 222](#_Toc448433102)

[Client.App (Code-behind) 223](#_Toc448433103)

[**Explanation of complex algorithms** 226](#_Toc448433104)

[Sending data over the network 226](#_Toc448433105)

[Serialisation/Deserialisation (General) 229](#_Toc448433106)

[Serialisation/Deserialisation (Specifically DataModels) 232](#_Toc448433107)

[Establishing references on the client side – Expanding, Attaching, and Detaching 233](#_Toc448433108)

[User Manual 236](#_Toc448433109)

[Appraisal 237](#_Toc448433110)

[Objectives 237](#_Toc448433111)

[Client Feedback 239](#_Toc448433112)

[Questionnaires 239](#_Toc448433113)

[Student 239](#_Toc448433114)

[Teacher 240](#_Toc448433115)

[Admin 241](#_Toc448433116)

[Extensions 241](#_Toc448433117)

# Analysis

## Client

Client: Burford School

Client Liaisons:

* Mr Wilsdon (Network Manager), for technical information and access to existing system.
* Mrs Bogdiukiewicz (Head of Computer Science), supervisor.

## Background, Description of Current System

At our school we have a number of rooms that can be booked for classes – mostly computer rooms and engineering workshops, with a few special cases such as the Library, Gym and Main Hall. At the moment, this booking process is done through a web interface (accessible from any computer as it is hosted by an external company). This system is **only accessible by teachers** – students don’t have a logon.

This can produce some problems – students, in particular sixth-formers, often use such rooms for quiet work areas, and are frequently disturbed by booked classes arriving without warning and using the room. This can have large impacts on academic performance – for example, a student needing to use an engineering workshop to complete some urgent coursework may be forced to leave due to another class using the room, which could potentially damage the student’s grades.

In addition, as the website is hosted externally, were the connection disrupted/the website shutdown, the school’s booking system would be rendered non-functional. Backups are taken and dealt with by the external company. The school must pay a **subscription fee** for both external hosting and backup services.  
  
The intended solution is therefore to create a system which introduces **students** into the booking system – not as “bookers”, but as viewers of the bookings. This will allow students to plan their studies around which facilities are available, hopefully assisting their academic studies.

The solution will provide a **replacement** to the current (external) booking system – it would not be possible to provide student access without this. The act of replacement does present an opportunity to improve the system for teachers as well, however, so I intend to make full use of this by making improvements to their use of the booking system as well, hopefully by streamlining the booking process and making it easier to use. The replacement system will be hosted internally on the central school server, providing more reliable access compared to the external website.

## Benefits of introducing a new system

Many students (in particular sixth-formers) feel that being forced to leave bookable rooms is a major disruption to their work. This system will hopefully alleviate that problem, by making it easier to see which rooms are available when. Even if they don’t check on the booking system themselves, one of the most strongly requested features is an automated message on logon informing them that they’re in a booked room, ensuring they’re aware. The main benefit that’s intended to be introduced is the ability for **students** to be able to view the bookings on the system – this will resolve the issues that they’ve been raising.  
There are also a few other problems that have been identified in the current system, such as the inability of the website to support both block-based and time-based scheduling, and unclear error messages. These I hope to fix during the implementation the new system, as well as adding the new functionality.

## Fact Finding Methods

I’ve used 3 common research methods to find out information about the existing system so that I can get a clearer sense of the capabilities of it. This should enable me to create objectives that precisely target specific flaws in the current system, while keeping it at the same level of usability as the current one.

Interview with Mr Wilsdon, Client

Q. What useful features are there in the current system that must to be carried across?

Teachers are emailed when they successfully book a room, cancel a room, or have a pending booking accepted. Multiple users can be imported into the system via CSV file which saves a lot of time, but individual users can be added too without bothering with a CSV. Some important rooms can have bookings requested, which are then approved by a set member of staff. New rooms can be added easily and appear instantly when staff try to book a new room. They can view rooms they’ve booked in timetable layout or as a list.

Q. What features are there that should be carried across if possible?

The current system can show statistics about how many rooms each department or teacher booked over different months – that’s useful for our IT report to the headteacher. It would also be good if the system could handle bookings for all the different types of bookings we do – when students aren’t in school we offer caravan parking on the school fields, which the current system lets us treat as booking venues.

Q. And what features would you like added?

At the moment you can’t have both “period scheduling”, where pre-defined blocks of time are booked, and “time scheduling”, where specific times chunks are booked, at the same time – this is problematic for when we book rooms after school for meetings, as we have to effectively book the 17 hour time chunk of 15:00 – 8:00. Also, some teachers “share” rooms – for example rooms D6 and D12 are next door to each other, so if we’ve got a large IT class in one room, and a much smaller A-level class in the other, sometimes the IT group will use some computers in the other class’s room. I guess that would be hard to implement, but it would certainly be useful.

Q. How much overhead is there for running the system? What extra costs are incurred?

I pay a few hundred pounds a year subscription to the company who host this solution, and I have to buy an SSL certificate to make sure the data is transferred securely. I also need to back-up all the data into two locations to abide by school rules, which can be quite time-consuming.

Q. How frequently would you need to add new rooms/teachers/users?

Obviously I’d need to add new users every year for the year 7’s. Adding teachers would be far less frequent, perhaps once a month maximum. Adding a new room would be very infrequent, once a year perhaps. Although having said that, I’ll need to add all the data to this system at the start anyway, so I will need a method of adding bulk data.

Q. What different types of bookings do you have?

There are the standard bookings, like the computer rooms, and then there are the less used rooms like the conference room, library, gym. Finally we have some strange ones, like the mobile phone and the slots for caravan rallies during holiday times.

Paperwork from current system

As the current system is electronic, there is no paperwork – instead I have provided many print-screens in the subsequent section to show each main area of the system.

Observation of current system

1. The user logs on. They can choose to log on as an administrator using the shown hyperlink (figure 1.0).



Figure 1.0

1. The user can view all the days’ bookings, and edit their bookings (figure 2.0).
   1. Clicking an empty timeslot will bring up a small dialog that enables the user to set some standard settings (figure 2.1), transfer to a page showing advanced settings (see 3.0), and confirm the booking.
   2. Selecting a category will change the shown timetable to the newly selected one (figure 2.2).
   3. Clicking “Week View” or “Category View” at the top switch to different viewing styles (a list of booked rooms or booked rooms by category).



Figure 2.0

1. The “Bookings” tab links to a list of viewed rooms, similar to the “Week View”, but specific to the logged in teacher (figure 3.0).

Figure 2.2



Figure 2.1

* 1. The “Add Booking” tab takes you to the same page as the “Advanced Options” choice on the calendar page did (see 2.1), with a choice of tabs to add extra detail to a booking.
     1. “Booking Notes” contains a textbox labelled “Year/Subject”, but most teachers write notes to the site team instead (figure 3.1.1).
     2. “Recurrence” provides detailed options for the teacher to set up automatic bookings, as well as a button to verify that the bookings don’t clash (figure 3.1.2).
     3. “Attendees” allows other teachers to be notified of a room booking along with the booker themselves (figure 3.1.3(a)). The address book button displays a list of all teachers in the system for easy adding (figure 3.1.3(b)).

Figure 3.0



Figure 3.1.1



Figure 3.1.2

Figure 3.1.3(a)



Figure 3.1.3(b)

1. The “Statistics” page instantly displays the number of bookings the logged-in teacher has made between the shown Start and End dates (defaults to the most recent week). It also displays some extra details as well as a pie chart of actual rooms booked, and a graph showing the number of bookings per day (figure 4.0).



Figure 4.0

Questionnaires

I have distributed questionnaires to students in years 12, 13, along with teachers; this constitutes the primary intended users of the system – students lower down the school don’t need to see which rooms are booked as they don’t have frees to use them in. I’ve included scans of a small sample of the completed forms, and done an analysis of the results below.

|  |  |  |
| --- | --- | --- |
| H:\Burford\Year 13\Computing\Project\_Writeup\Resources\Analysis\Questionnaires\1.jpg | H:\Burford\Year 13\Computing\Project\_Writeup\Resources\Analysis\Questionnaires\2.jpg | H:\Burford\Year 13\Computing\Project\_Writeup\Resources\Analysis\Questionnaires\3.jpg |
| H:\Burford\Year 13\Computing\Project\_Writeup\Resources\Analysis\Questionnaires\12-5.jpg | H:\Burford\Year 13\Computing\Project\_Writeup\Resources\Analysis\Questionnaires\12-3.jpg | H:\Burford\Year 13\Computing\Project\_Writeup\Resources\Analysis\Questionnaires\12-1.jpg |

40% of students who completed questionnaires were Year 12 students, and 60% were Year 13. The results highlighted a few important points, and conclusions have been drawn from these:

* 53% of students claimed to have been interrupted in their studies by a booked class more than 11 times a month, and 20% between 6-10 times. This is a very high frequency and many students commented on how annoying it was while filling out the forms – evidence that some system is needed to reduce this occurrence.
* 100% of students thought it would be useful to be able to see which rooms have been booked by accessing a computer. This is the clearest statistic emphasising the demand for this system, as there is clearly significant student opinon on the matter.
* 80% of students frequently used computer rooms around the school, and 73% used the library regularly. By contrast, no sampled students used the workshops outside of booked lessons. This data has some important ramifications; the two most frequently used types of room both have computers nearby, which would make a computerised system preferable as it would be highly accessible to students.
* 93% of the sample found the timetable layout used on the sheets given to us at the start of the year easy to understand. Taking this on board, when designing the display of the timetable in the implementation, I will base it on the layout of our paper ones as there’s evidence that students find it easy to read, which will increase usability.
* 60% of students described their skill with computers as “Average”, with 40% describing themselves as “Skilled”. This is a positive result, as it suggests that there will be few problems with students being able to use the system, reducing the need for training (which for a sixth-form student body of around 200 people would be a difficult task).

## Identification of Users

At the moment there are 2 styles of access: one for teachers and one for administrators – teachers can use a slightly restricted version of CRUD: they can Read all bookings but only Create/Update /Delete their own bookings through the web interface. Administrators can view and edit all bookings. Students are unable to view any bookings, and in fact most are unaware that a booking system is currently used.

The current system doesn’t require much training, although this is recommended by the school so that mistakes aren’t made. Administrators do need training but it’s just so that they are aware of all the features – the system is clear enough to use without extensive training. The current administrators are the network managers who have a high degree of technical knowledge, while the teachers have varying knowledge from basic to advanced.

In the proposed solution, there will be 3 tiers of access: for teachers, administrators, and students. The abilities of teachers and administrators will be almost identical to the current system. Students will be able to view bookings for the current day and possibly the next few days. Administrators should also be able to revoke individual student’s room viewing rights, as a layer of control.

Ideally the solution will be intuitive and easy to understand, meaning most users won’t need training – this is almost necessary in a system with potentially over 1000 users. The students access to the system is very limited and simple, so likely won’t need any training. Teachers may need to be shown how to use the features, although they will be largely the same as those in the existing system. Administrators may need some extra instruction on the usage of more advanced features. Easily accessible help documentation should be included to assist all users.

## Data Sources/Destinations – Existing System

The following is a table containing the various data in the existing system, and the transfers taking place.

|  |  |  |
| --- | --- | --- |
| Data | Source | Destination |
| List of teachers | Admin input | Website -> Database |
| List of students | Admin input | Website -> Database |
| List of bookable rooms | Admin input | Website -> Database |
| List of periods | Admin input | Website -> Database |
| List of bookings | Database | Website -> Database |
| New/edited booking | Teacher input | Website -> Database |

## Data Dictionaries – Existing System

Teacher

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Field Name | Field Purpose | Field Type | Example Data | Validation |
| First Name | Stores first name of the teacher | String | “Peter”, “Bill” | Not empty |
| Last Name | Stores last name of the teacher | String | “Smith”, “Brown” | Not empty |
| Username | ID used by the teacher to log onto the system | String | “mb”, “cn” | Not empty |
| Password | Secure string used to verify the teacher’s logon | String | “password”, “010498”, “h#\_2” | Alphanumeric and symbols |
| Title | Stores the title of the teacher | String | “Mr”, “Ms”, “Mrs” | Not empty |
| Email | Stores the email address of the teacher | String | “admin.4040 @burford.oxon .sch.uk” | Not empty, valid email format |

Room

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Field Name | Field Purpose | Field Type | Example Data | Validation |
| Room Name | Stores the name of the room | String | “D6”, “Library” | Not empty |

Department

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Field Name | Field Purpose | Field Type | Example Data | Validation |
| Name | Stores the name of the department | String | “Technology”, “MFL” | Not empty |

Booking

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Field Name | Field Purpose | Field Type | Example Data | Validation |
| Start | Stores the start time of the period | Time | “8:50”, “13:10” | Valid time format |
| End | Stores the end time of the period | Time | “9:50”, “14:10” | Valid time format |

## Entity-Relationship Diagram – Existing

The existing system has a simple entity model – all relations are one-to-many, and there are only 4 entities.

E:\Burford\Year 13\Computing\Project\_Writeup\Resources\Analysis\ER Diagram.png

Room has many Bookings

Booking is in multiple rooms

Room belongs to a Department

Department has many rooms

Department has many teachers

Teacher belongs to one department

Booking is made by one teacher

Teachers have many Bookings

## Data Flow Diagram – Current System

This is a simplified model of the data processes and flow in the current system. It should provide a neat summary of the current system for me to refer to during the design stage. It should be noted that due to the tiered nature of access to the system for different users, colour coding has been used for clarity. Administrators retain access to the same features as Teachers. 

## Potential Solutions

The current system was implemented to replace a highly inefficient paper-based system, so it doesn’t make much sense to regress back to a manual system. As such, I have only considered electronic solutions.

1. A web-based system, which would allow teachers to book rooms from anywhere (not just school), and let students view the bookings without needing to log onto a computer by using their phones. Access control would be done via user accounts (username/password).
2. A Client-Server based system, where on logon an application starts that communicates with a server process running on the central school server and presents an interface to book/view the rooms. This would let the system know where people are logged on, but presents the problem of accessibility – users need to log onto a school computer to view bookings, which isn’t much use for checking where friends are.
3. A mobile app that provides the same level of interaction as the web-based solution, but is developed as an app (so a client-server system). This would provide the same amount of mobility as the website version, but would allow for a more complex UI.

## Appraisal of Potential Solutions

1. Web-based system: While this would provide an easily-accessible system, I have little experience in web-development, so don’t think I would be able to create a satisfactory solution using this approach. The existing system is also web-based, so I would likely be limited to the same design as the current one.
2. Client-Server system: I have the most experience developing solutions of this form, so I would feel comfortable using this design. However, this would need a lot of work as the communications between client-server would need to be designed as well. This has the most room for expansion though, as complex UI and logic can be added fairly easily.
3. Mobile App – I have some experience with mobile technology, but like the website solution not enough to create a satisfactory solution. The mobility factor is a positive feature, as people can browse the bookings without needing to log on, but if the overall solution is of low quality then the gained mobility doesn’t really matter.

## Justification of Chosen Solution (Client-Server)

I have chosen to go with the Client-Server solution, as I believe I can make a more robust and effective system using this design rather than the other two possible designs.  
It does mean that mobility is reduced, as students and teachers need to log on to a school computer to see the system, but the application’s GUI can be more responsive and dynamic.

## Limitations

The solution will be developed in C#, which I am comfortable with, but will have to use fairly advanced features of the language such as TCP/IP communications and multithreading, as well as producing a user-friendly interface using the Windows Presentation Foundation (WPF). I may also need to use multiple paradigms – primarily Imperative/Procedural, although the use of C# will naturally result in primarily Object Oriented code, and in producing a responsive user interface I will use Event-driven programming. Also, as I will likely need to use multiple threads, I will need to use a blocking paradigm to ensure operations perform in a thread-safe manner. As such I expect there will be lots of bugs to be fixed in the development versions, so need to run many tests to root them out (see the Test Plan at the end of the Design section).

The following are the *TELOS* considerations for the project:

* Technical
  + Hardware – the server will need to be able to handle connections to many computers around the school. As such it will need a large range of available ports for allocating as needed. High speed in this system isn’t required (although it is obviously desirable!), so the server doesn’t require any special hardware. The school logon server should be able to handle running the software process.  
    The client machines only need to run an application in the background, and it won’t be doing any significant processing, so they won’t need specialised hardware either.  
    Ideally, no money should be spent on buying new hardware specifically for this project.
  + Software – if I develop a computerised solution in a .NET language, all the machines running the resultant programs will require the .NET framework installed and updated to at least .NET version 4. This is installed on most computers already, but it would be need to be ensured to be present and updated as a prerequisite of this system.
* Economic – as the proposed systems don’t require any extra hardware the costs should be minimal. Similarly, none of the features of the system require funding. The server machine will need to do more processing so will use more power, but those costs are likely negligible.
* Legal – as my system will be handling data pertaining to living persons (teachers and students) such as names, email addresses, and their probable location at a given time, I need to make sure that all data is encrypted and stored securely (Data Protection Act, 1998).
* Operational – the current system has been used for many years and its uses are deeply ingrained in the minds of the staff; the new solution should therefore be easy to use and intuitive, reflecting the features of the current system, although without being averse to adding new features/streamlining existing ones.
* Scheduling – leading on from the previous limitation, with such a large application to develop I will no doubt be pushed for time. Changes to the user requirements will exacerbate this pressure, so I will need to be very careful in how frequently I check my progress with the client, in order to make sure I spend as little time as possible working “in the wrong direction”.

## Objectives

As well as observing the current system, I have released a number of surveys and questionnaires to students, teachers, and administrative staff. I feel that as they are the majority users, they will be far more able to identify problems in the current system than I will by observation. My research has yielded the following list of objectives, ranked very loosely from most important to least important, based off the results from the interviews and surveys:

| # | Objective |
| --- | --- |
|  | Create a program that will allow 3 layers of access to the system for the 3 different users. |
|  | Allow users to view bookings over different days/months etc. |
|  | Allow teachers to view and book specific rooms around the school site. |
|  | Allow students to view bookings for rooms around the school site. |
|  | Prevent unauthorised access to the system for non- students/teachers/admins. |
|  | Allow administrators to override bookings |
|  | Allow administrators to revoke student access to viewing bookings. |
|  | Provide an easy to use interface, so that students can use it without training, and teachers can use it with minimal training. |
|  | Provide clear and useful error messages that should give teachers an idea of what went wrong, not just that something did go wrong. |
|  | Allow teachers to set up recurring bookings on varying schemes (daily, weekly, monthly). |
|  | Provide checks to ensure users aren’t making mistakes when editing bookings. |
|  | Provide a warning to users when logging onto a computer in a room that’s been booked for that period. |
|  | Provide email notifications to teachers when they book/cancel a room. |
|  | Provide useful statistics to teachers about room bookings, including per-teacher and per-department information. |
|  | Allow data to be added in bulk, using CSV files or possibly Active Directory services. |
|  | Allow the clients to handle loss of connection to the server gracefully, and restore connection and services as soon as possible. |
|  | Securely hold records of all bookings. |
|  | Allow bookings to be backed up easily. |
|  | Allow the server to be turned on and off with minimal interruption to service (ie doesn’t “forget” bookings). |
|  | Allow new bookable rooms to be added (by administrators) with ease. Immediate availability of the new room for booking is not required, but is preferred. |

These objectives have been developed with the “SMART” acronym in mind: objectives are clear to prevent confusion later on in the design stage; most will either end up as complete or incomplete (and thus easily quantifiable), which makes it easy to measure their success; most of the objectives are strict requirements of the system in order for it to work well, so must be achievable; given my experience with programming, I feel quite confident that I can develop this solution, so I believe the goals to be realistic; in terms of time constraints, I would like to finish the solution with plenty of time to spare for testing/bugfixes/refinements, which gives me a few months to develop the system in.

## Data Sources/Destinations – Proposed Solution

This is the table of data transfers in the proposed solution. Note that an example of data flow would be “Database -> Server -> Client” meaning that data is sourced from the Database, transmitted through the server, and ends up at the Client.

|  |  |  |
| --- | --- | --- |
| Data | Source | Destination |
| List of teachers | Admin input | Client -> Server -> Database |
| List of students | Admin input | Client -> Server -> Database |
| List of bookable rooms | Admin input | Client -> Server -> Database |
| List of periods | Admin input | Client -> Server -> Database |
| List of classes | Admin input | Client -> Server -> Database |
| List of departments | Admin input | Client -> Server -> Database |
| List of bookings | Admin input | Client -> Server -> Database |
| New/edited booking | Teacher/Admin input | Client -> Server -> Database |
| List of teachers | Database -> Server -> Client | Client (admin) interface |
| List of students | Database -> Server -> Client | Client (admin) interface |
| List of bookable rooms | Database -> Server -> Client | Client (admin) interface |
| List of periods | Database -> Server -> Client | Client (admin) interface |
| List of classes | Database -> Server -> Client | Client (admin) interface |
| List of departments | Database -> Server -> Client | Client (admin) interface |
| List of bookings | Database -> Server -> Client | Client (all) interface |

## Data Volumes

I intend to store the data in a database – there are quite a few references between records, such as a Booking having a list of Students. A serial access file like a binary file would need loading entirely into memory, and it would be difficult to follow the references. If the server crashed unexpectedly while the data was in memory before being saved to the file, it would be lost. A hierarchical file such as XML wouldn’t be able to represent all the relationships between data, and would also need to load into memory.  
A database allows random access and doesn’t need loading into memory, while also natively supporting references between records.  
The use of the Entity Framework will also allow me to access the data in a C# object-based manner, rather than directly through queries.  
  
An empty SQL database file is around 3MBs in size, and my records are unlikely to use any more than 100bytes each. Using this as an upper bound, however, this means that for every 10,000 records the file will increase by 1MB; an acceptable scale factor. It’s safe to say that there will never be more than 100,000 records stored, but if there were the file size would still only be around 13MB.

## Entity-Relationship Diagram – Proposed

The model for the proposed system is more complex – there are 8 entities and the relations include one-to-many and many-to-many.  
For example, a Booking can include multiple Students and take place in many Rooms, but can only use one Period and have one Teacher. At the same time, a Teacher or Student may have multiple Bookings, and so on.  
Most entities have a link to Booking, as the system is mostly centred around the Bookings. Other entities provide information that’s relevant and useful to know for the structure of the system.

E:\Burford\Year 13\Computing\Project\_Writeup\Resources\Design\ER Diagram.png

Period has many Bookings

Room has many Bookings

Booking is in multiple rooms

Room belongs to a Department

Booking is during a Period

Department has many rooms

Department has many teachers

Teacher belongs to one department

Teacher teachers multiple classes

Class only has one teacher

Class has many students

Students are in many classes

Booking involves many students

Students are involved in many bookings

Booking is made by one teacher

Teachers have many Bookings

## Data Flow Diagram – Proposed

This is a general flow chart to show the rough logic of the system’s usage. The different colours represent different levels of access, using the key displayed on the diagram. Admins have access to all the functionality available to Teachers, who can in turn access all features available to students.



# Design

## Overall System Design

In the analysis section I decided on using the Client-Server system – as such, access to the system will be through an application available to computer users, which connect to a central server process running on the either an external server or more likely the internal school server. Specifically, upon login on to the computer, an application will start in the background which will present the current bookings. In the background, it will inform the server that the user has logged on, automatically sending relevant information such as the username and computer name. This will cause the server to update its internal model and distribute the change to all logged on clients, and send a copy of the current system state to the Client.

All types of users will use the system through the same application, but different “areas” of it will be accessible only to certain access levels of user. This access is cumulative, so a Teacher has access to all areas a Student does plus a few more, and an admin has access to all those a Teacher does plus more.

## H:\Burford\Year 13\Computing\Project\_Writeup\Resources\Maintenance\Modular Structure of Solution.pngDescription of Modular Structure

The planned system has quite a lot of room for code re-use; the Client and Server will both need access to both common networking code and descriptions of the data stored in the database. As such, I’ve attempted to structure the solution in a modular layout: there are 5 components which are connected by code references like a tree (diagram on the right). Descriptions of each of these components are given below.

Client

A process running on each school computer; it runs on user logon and continues in the background. Presents a GUI to the logged in user that allows them to perform actions as allowed by their access rights (determined by a search of the database for the windows username). Also communicates with the server to send/receive updates to the database. No special hardware is needed to run this component – it isn’t particularly demanding.

Server

A process running on the school server; it provides information about the bookings to clients as well as updating the database. This is the only component with direct access to the database – all other access is through the server, allowing a central mechanism of ensuring validity of inputs. The only special hardware required for this module is a machine that’s online all the time that the clients are. The Server load won’t be massive as there will be fewer than 500 clients (machines) running, and if more are ever added, load will only increase linearly.

NetCore

This assembly should be referenced by both the Client and the Server and will provide access to common networking code. This will increase code reuse, saving time during development and making refactoring easier.

The networking framework will be developed as an event-oriented system, as it best matches the logic of the client-server relationship: A client will send a booking message to the server, which *upon receiving it* will update the database and distribute the change to the clients. *Upon receiving* these messages, the clients will update their UI.

Data

This assembly holds the classes that represent items in the database. The Server must reference this, obviously, as it deals with the database, but the Client will too – messages between the Server and Client will most likely pass these objects around to indicate the changes that have been made to the database, so the Client requires access to these classes.

I intend to use the Entity Framework Code First library (<http://www.asp.net/entity-framework>) to provide access to the database. This will let me design the model using C# code, and the database schema will then be generated and applied independently. Because the model is developed using C#, I can interact with it easily from the rest of my program logic without needing to perform literal queries within code – they will be abstracted to properties within the model classes.

Shared

This assembly is the result of the need to abstract out even further from the NetCore and Data assemblies – the NetCore assembly must have access to the classes defined in the Data assembly in order to create messages that carry the database models, but at the same time the Data models need to specify how they should be serialised to the network which requires knowledge of classes defined in the NetCore assembly. This would create a circular dependency, so I’ll been forced to create a separate assembly to provide some highly abstract classes/interfaces (descriptions of the behaviour of a class, rather than the specific logic it performs) that can be used to resolve the conflict.

## Data Requirements/Record Structure/Database Design

The Data module described above is referenced by many of the other components and provides descriptions of the entities stored in the database. The fields and information associated with these entities are described in the Data Dictionaries below, while the relations between them are shown in the following Entity Relationship diagram (repeated in the Analysis section).

E:\Burford\Year 13\Computing\Project\_Writeup\Resources\Design\ER Diagram.png

A Subject has many Bookings

A Booking has a Subject

Period has many Bookings

Room has many Bookings

Booking is in multiple rooms

Room belongs to a Department

Booking is during a Period

Department has many rooms

Department has many teachers

Teacher belongs to one department

Teacher teachers multiple classes

Class only has one teacher

Class has many students

Students are in many classes

Booking involves many students

Students are involved in many bookings

Booking is made by one teacher

Teachers have many Bookings

Student

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Field Name | Field Purpose | Field Type | ≈Field Size (bytes/chars) | Example Data | Validation |
| First Name | Stores first name of the student | String | 50 | “Peter”, “Bill” | Not empty |
| Last Name | Stores last name of the student | String | 50 | “Smith”, “Brown” | Not empty |
| Logon Name | Stores the username of the student | String | 50 | “09135”, “caict10” | Not empty |
| Year | Stores the year group of the student | Integer (unsigned) | 4 | 7, 8, 9, 10, 11, 12, 13 | Nonnegative, maybe 7 ≤ Year ≤ 13? |
| Form | Stores the form group of the student | String | 20 | “WT”, “BR”, “CEB” | Not empty |

Teacher

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Field Name | Field Purpose | Field Type | ≈Field Size (bytes/chars) | Example Data | Validation |
| First Name | Stores first name of the teacher | String | 100 | “Peter”, “Bill” | Not empty |
| Last Name | Stores last name of the teacher | String | 100 | “Smith”, “Brown” | Not empty |
| Title | Stores the title of the teacher | String | 25 | “Mr”, “Ms”, “Mrs” | Not empty |
| Logon Name | Stores the username of the teacher | String | 50 | “ceb”, “mb” | Not empty |
| Email | Stores the email address of the teacher | String | 100 | “admin.4040 @burford.oxon .sch.uk” | Not empty, valid email format |

Room

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Field Name | Field Purpose | Field Type | ≈Field Size (bytes/chars) | Example Data | Validation |
| Room Name | Stores the name of the room | String | 50 | “D6”, “Library” | Not empty |
| Standard Seats | Stores the number of normal “desk-spaces” | Integer | 4 | 50, 20 | Nonnegative |
| Special Seats | Stores the number of non-desk spaces | Integer | 4 | 50, 20 | Nonnegative |
| Special Seats Type | A description the non-desk spaces | String | 50 | “Computer”, “Workbench” | Not empty if Special Seats ≠ 0 |

Class

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Field Name | Field Purpose | Field Type | ≈Field Size (bytes/chars) | Example Data | Validation |
| Class Name | Stores the name of the class | String | 100 | “Maths”, “Computing” | Not empty |

Department

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Field Name | Field Purpose | Field Type | ≈Field Size (bytes/chars) | Example Data | Validation |
| Name | Stores the name of the department | String | 100 | “Technology”, “MFL” | Not empty |

Period

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Field Name | Field Purpose | Field Type | ≈Field Size (bytes/chars) | Example Data | Validation |
| Name | Stores the name of the period | String | 100 | “Period 1”, “Lunchtime” | Not empty |
| Start | Stores the start time of the period | Time | 8 | “8:50”, “13:10” | Valid time format |
| End | Stores the end time of the period | Time | 8 | “9:50”, “14:10” | Valid time format |

Booking

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Field Name | Field Purpose | Field Type | ≈Field Size (bytes/chars) | Example Data | Validation |
| Date | Stores the day that the booking occurs | Date | 8 | “01/01/2016”, “28/02/2016” | Valid date format |
| BookingType | Stores how often the booking recurs | Enumeration | 1 | “Single”, “Weekly” | Value defined by the enumeration |

Subject

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Field Name | Field Purpose | Field Type | ≈Field Size (bytes/chars) | Example Data | Validation |
| Name | The name of the subject | String | 100 | “Computing”, “Maths” | Not empty |
| Colour | The colour used to represent the subject. | Colour | 16 | “FF00FF00”, “FF888888” | Valid hex/colour |

## Validation Required

Specific validation rules have been covered in the tables above. There are some more general ones, however, which are outlined here:

* Any text input into the system (usually) permits any characters and any length – in most cases the only “validation” required is a simple presence check.
* Any numerical input must obviously only make use of the digits between 0-9, and will accept leading zeros (but will ignore them for the actual value). In all cases the value of the number must be either greater than 0, or greater than or equal to 0 (>0 / ≥0), but this does depend on the context of the input.

An additional note (although not a rule per-se, more of a design guideline for during the implementation), is that there isn’t much “raw” data entered into the system by the user – Admins are the only users who should ever actually enter text/numbers, with teachers simply selecting pre-set options. As such, heavy use of combo-boxes and lists for selection should be used, as this will reduce the validation required as data can only be selected from a fixed set of valid values.

## Processing and Algorithms

The following are high-level (pseudocode) descriptions of some of the algorithms I intend to use – a lot is taken for granted, such as the specifics of sending a message, which would require a significant amount of development in the background to work in an object-oriented and scalable fashion.  
I use the keyword “Upon” to indicate a function that is executed upon an event occurring, to distinguish it from a normal procedure that executes when called (synchronously).

Client

// Called when the user tries to add a booking  
Upon BookingAdded(NewBooking) Do  
 If Invalid(NewBooking) Then  
 ShowErrorMessage()  
 EndIf  
 SendNewBookingMessage(Server, NewBooking)  
End  
  
// Logic for when the server acknowledges the new booking  
Upon BookingMessageAcknowledgementReceived(Message) Do  
 If IsBookingFailedMessage(Message) Then  
 ShowErrorMessage()  
 EndIf  
End  
  
// Logic for when a message containing a new booking layout is received  
Upon BookingChangeMessageReceived(Bookings) Do  
 If *User is in the process of making a new booking* Then  
 *Wait until user finishes*  
 EndIf  
  
 UpdateBookingUI(Bookings)  
End  
  
// Fired when the application starts up. Connects and displays a message if the  
// room the user’s in has been booked for this period  
Upon ApplicationStart() Do  
 Connect(Server)  
 Message = ReceiveUserInformationMessage(Server)  
 Database = ReceiveInitialDatabaseState(Server)  
  
 Foreach Booking b in Database Do  
 If GetPeriod(b) == CurrentPeriod Do  
 Foreach Room r in GetRooms(b) Do  
 If r == CurrentRoom Do  
 DisplayNotification()  
 EndIf  
 EndForeach  
 EndIf  
 EndForeach  
End

Server

// Called when a new booking message (Booking) is received from a client (Sender)  
Upon NewBookingMessageReceived(Sender, Booking) Do  
 If Invalid(Booking) Then  
 SendAcknowledgeBookingFailed(Sender)  
 Return  
 EndIf  
 ThreadLock(Database) Then  
 If *Booking clashes with existing database* Then  
 SendAcknowledgeBookingFailed(Sender)  
 Return  
 EndIf  
 EndLock  
 SendAcknowledgeBookingSuccess(Sender)  
  
 AddBookingToDatabase(Booking)  
 ForEach Client in ConnectedClients Do  
 SendBookingChangeMessage(Booking)  
 End  
End  
  
// Called when a new client (Sender) connects with given username and computer name  
Upon ClientConnectMessageReceived(Sender, Username, ComputerName) Do  
 ThreadLock(Database) Then  
 AccessLevel = GetAccessLevel(Database, Username)  
 Room = GetRoom(Database, ComputerName)  
  
 SendUserInformationMessage(Sender, AccessLevel, Room)  
 SendInitialDatabaseState(Sender, Database)  
 EndLock  
End

NetCore

// Sends message to the target computer by serialising message then making sure all are sent  
Proc Send(Message, Target) Do  
 Bytes = Serialise(Message)  
 While *not sent all bytes* Do  
 SendBytes(*Remaining bytes*, Target)  
 End  
End  
  
// Converts an object to an array of bytes recursively  
Func Convert(Object) Do  
 Bytes = *Empty byte array* ForEach Member m in MembersOf(Object) Do  
 If *m is primitive type* Then  
 Append(GetBytes(m), Bytes) // Get the equivalent bytes using a built-in function  
 Else  
 Append(Convert(m), Bytes) // Get the equivalent bytes using a recursive call  
 EndIf  
 End  
 Return Bytes  
End

## Network Protocol Design

In this section I will attempt to describe the network protocol used in transferring data across the networks between client and server.

In order to encode an object to the network, they first need to be serialised. As I intend to use a stream-based communication method (TCP/IP), I will likely implement the following algorithm for serialising data:

Serialise(Object *where object is serialisable*)  
 ForEach *Field to be serialised in Object* Do  
 If *Current field is serialisable* Do  
 Serialise(*Current field)*  
 Else  
 Send(*Current field*)  
 End  
 End  
End

H:\Burford\Year 13\Computing\Project\_Writeup\Resources\Design\Serialisation Demo.pngAs can be seen, this algorithm for serialising data is recursive – it sends each field on an object to be sent by either sending the field plainly if it’s a primitive type (eg string, int), or if the type of field is itself serialisable, it lets that field serialise itself using the same function.

The diagram to the right demonstrates this recursive approach. Serialisable data structures can be modelled as trees, and serialising the structure is effectively just a depth-first/postorder traversal of the tree.

Deserialisation can be performed in a very similar manner, but deserialising rather than serialising. It is very important, however, that the order fields are serialised in is the same order they are deserialised in, otherwise data will be jumbled up between fields.

Deserialise(Object *where object is deserialisable*)  
 ForEach *Field to be deserialised in Object* Do  
 If *Current field is deserialisable* Do  
 Deserialise(*Current field)*  
 Else  
 Receive(*Current field*)  
 End  
 End  
End

This is all an abstraction of the actual implementation details, so there are likely going to be adjustments made to this plan during implementation to deal with pitfalls I haven’t thought of yet.

## Database Queries

As I will be using C# and Entity Framework to access my database, it would be pointless to use direct SQL queries. Instead, Entity Framework hides the actual querying behind C#-esque syntax, using LINQ extension methods and often lambda functions from native C#.

An example “query” using this system might look like the following:

Data.Students.Where(s => s.Year == 13)

which would return all students (from some database repository named Data that are in year 13). A more complex example could be:

Data.Bookings.Where(b => b.Date.Date == DateTime.Now.Date).SelectMany(b => b.Students).Distinct().OrderBy(s => s.FirstName)

This query would first collect all the Booking objects whose Date is set to the same Date as the code being run, then use SelectMany to “flatten” the many lists of lists of students into a single list. Distinct is then called to remove any items that are repeated, before the list being sorted using OrderBy on the first name of the students. The resulting list thus contains all students that are involved in a booking today, sorted alphabetically by first name (and without any repeats).

Obviously this isn’t SQL – however it is very similar, in the order of operations and terminology used.

## Storage Material and Format

The data will be stored in an SQL database on the server’s HDD. As covered in the Data Volumes section in the analysis, the maximum file size of the database is under 15MB, so the existing hard drives can be used to save hardware expenses. An SSD could be used, but fast read speeds aren’t necessary.

Due to the small size of the database, backups would ideally be taken daily, with maybe up to a month’s old files stored – storing 30 copies of the database files from subsequent days would still only use up to 450MB. An offsite backup is recommended, and could be done as part of the school’s pre-existing weekly offsite backup system.

## Human-Computer Interface

Design Factors

User

Users ideally won’t need training with the system (although for admins/teachers it would make sense to provide it nonetheless). Users have a variety of technical ability, from novice (mostly younger students) to highly skilled (administrators and teachers), so the UI should be tailored to these ability sets.

Colour

The school does have an existing colour scheme (which happens to be the same as the scheme used in this writeup), but I decided not to use the same scheme in the solution: instead, using the standard Windows colour schemes helps make windows seem more familiar, less abnormal. Hopefully this will in turn make my application more intuitive – or rather, not make it any less intuitive. As a result, my UI will use grey background and black text in the “Segoe UI” font, the default windows scheme.

The only window that uses any colour besides this is the Timetable window, which will use bright colours to represent bookings, in order to draw attention to them and make it clearer to the user where the important information on the form is.

Presentation

Most windows are static, with no animations or changes except for those triggered directly in the UI by the user (eg hovering over a button changes its colour slightly). The exception is the timetable window, which ideally would have some method of emphasising certain lessons that the current user is involved in, so they can easily spot which bookings are most relevant to them on the timetable. This emphasis will likely be a temporary change in colour.

Feedback

All user actions should have some clear feedback associated with it – pressing a button shouldn’t silently make a change, it should show a dialogue box verifying an action has been taken or make some noticeable change in the UI to indicate the action. The choice of feedback varies on the context – attempting to delete a booking should show a modal confirmation dialogue as it demands the attention of the user, whereas navigating to a new day on the timetable would simply update the timetable itself as well as the displayed date being viewed.

Exits

The standard “red X” at the top right of each window is the standard way of exiting a window. This is available on all windows used in the system. There aren’t any “back” buttons, each subsequent layer of windowing is built on top of ones below (ie, opening windows 2 from window 1 doesn’t close window 1, but simply prevents input to it whilst window 2 is open). This creates a very natural system where users can see which windows they’ve come from and return to them by closing the current one.

Shortcuts

No keyboard shortcuts have been provided, as there are no tasks which are done particularly frequently and which would benefit from the utility of a keyboard shortcut. All navigation is done using buttons which lead to either data-entry or data-view windows.

Error Messages

All error messages are displayed using dialogue boxes containing text about the error. Phrasing should be chosen to reduce jargon and make it clearer what’s actually gone wrong, so users have a better understanding of how to fix the error.

Timetable Window

Current date, for easy identification of viewed day.



First table column is like the table heading – different colour to show distinct area, and shows name of Rooms as the most important bit of information.

Easily identifiable (blank) cells that can be booked, provide a backdrop to the brightly coloured bookings. Colour contrast increases speed of recognition and draws attention to important items.

Timetable area is by far the largest control on the window – draws attention to the main and most important part first.

Table header shows Period name and range of times, which is the most important information. Different colour to the table cells, so it’s a distinct separate part and not confusing.

Next/Previous buttons to move between days. Large, so easy to click.

Admin Controls Window



Add/Edit/Delete buttons (CRUD). At bottom of window and positioned below the densest area of information to keep focus in one region.

Information on items of the type specified by the tab, in an easy-to-read table.

Tabs for different entities editable by the Admins. All have a consistent design within them to increase usability.

EditBooking Window



Generic info about the booking, such as teacher, subject, period etc. Inside a labelled GroupBox to show the logical collection of items.

Multiselect rooms – information on each room is displayed, along with checkboxes to select each one.

Multiselect Students along with intuitive filter functionality to speed up searching. Information on students is displayed along with a checkbox to select them. Same style of control as the room multiselect, to reduce confusion.

Delete/Submit buttons at bottom of form as they’ll be used last. Submit at bottom right as “Continue” style buttons are customarily bottom right.

Edit Student Window



Back/Save as only two options, both positioned in the standard location for submit buttons.

Combox of Access modes, populated at runtime with the available values.

Multiple text fields to accept the different editable values. Validation/conversion between text and the actual value can be performed later.

Consistent size of input controls improves aesthetics and uniformity.

Edit Department Window

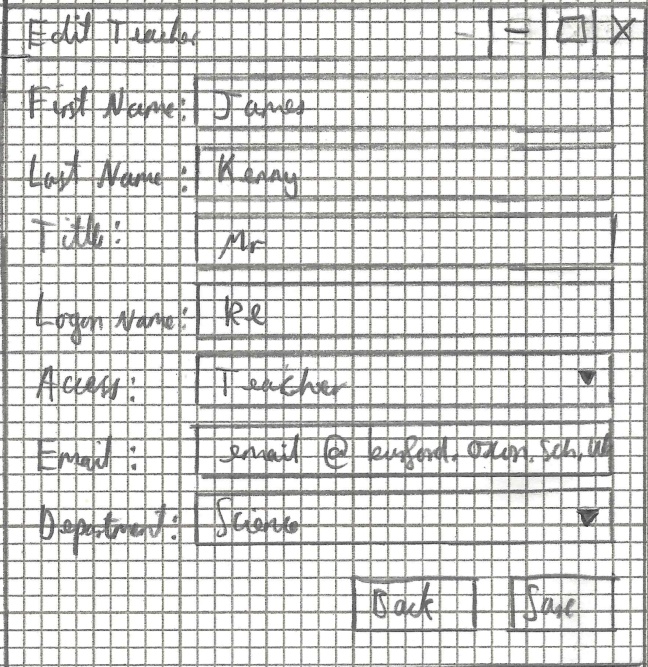


Same design theory as the above – easy to read, type of data is marked by a group box.

Selectable values displayed in an easy-to-interpret tabular layout, inside a groupbox that identifies the type of data being selected.

“Simple” fields positioned at top for clean layout.

Edit Teacher Window

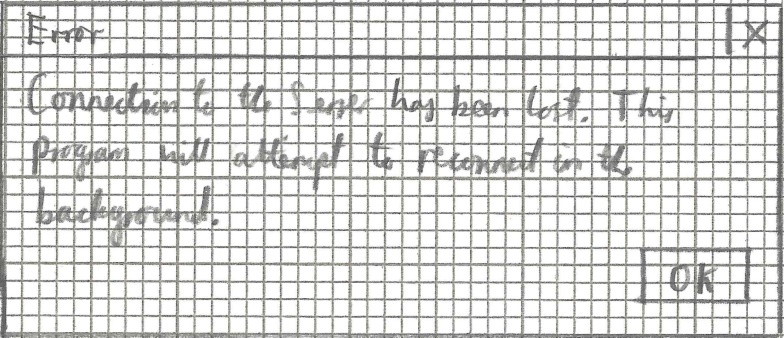


Email field should be validated with a regular expression for mail addresses to prevent typos and reduce errors.

Comboboxes for fields with only discrete values to reduce user mistakes.

Free-entry text fields with no validation as any name/title/logon is feasible.

Connection Lost Error Dialog



Simple but clear informative message: tells the user what went wrong and what’s being done to fix it.

Use of modal dialogues means all windows other than this are inaccessible until this window is closed, ensures the user is aware of the error.

## Sample of Data Entry

Data entry to the system is only through the windows called “Edit \_\_\_ Window” displayed in the Human-Computer Interface section above, so I’ve included on those diagrams examples of possible inputs to the system – all data shown would pass validation for those fields. Combo-boxes (drop down menus) have been used where possible to restrict user input to a set of decirete values in order to reduce mistakes/validation needed.  
Forms tend to be used simultaneously for input and output – data can be inputted, or can simply be viewed all from the same form. As a result, I’ve grouped the screenshots into the one sction above rather than duplicate them for the Valid Output section below.

## Sample of Valid Output

The main visible “output” from the system is the visual representation of the timetable, a diagram of which has already been included in the HCI section. Other forms of output include the emails that teachers will receive when a booking by them is edited or created, but that output is simply text and can’t really be represented as a diagram. Error dialogue messages are the final form of output, which again I’ve included in the HCI section above.   
Forms tend to be used simultaneously for input and output – data can be inputted, or can simply be viewed all from the same form. As a result, I’ve grouped the screenshots into the one sction above rather than duplicate them for the Data Entry section above.

## Security & Integrity

In order to access the application that provides access to the server, the user needs to log on – the application then sends the server the logged on username, and only if the username is registered in the database as an allowed username does the server provide the information about the bookings etc. This provides a fairly safe system. I intend to write a user manual to help people, and integrated help documentation in the form of UI tooltips and explanatory UI messages.  
  
All input will be validated to ensure it is correct. Ideally, the design of the UI will in fact prevent users from entering invalid data in the first place – for example, using a “ComboBox” to restrict user input to a selection of discrete values. Where such restrictions aren’t possible, such as when entering a teacher’s email address, standard validation will be used to ensure that reasonable input is made.

## System Security

As my system is dealing with personal information (names, ages of students, etc.), I do need to ensure that the security is up to scratch. I have the significant advantage that the whole system will be hosted inside the school’s network, which is already secured in accordance with regulations – it has firewalls and password-protected access. As such, many security requirements are already fulfilled by the school – the network is kept secure, and physical access to the servers is restricted. SQL provides integrated security as well, requiring authentication to access the database. The school server is in a dedicated room that has a smoke detector and burglar alarm installed.  
To prevent data loss from fires/burglary etc, backups of the database need to be taken frequently (this can happen along with the school server’s automated backups). Ideally a copy of the system data would also be stored off-site for a greater strength of fall-back backups.

## Test Plan

Due to the size of the solution I intend to use multiple test strategies on different parts of the system. For example, because the Data and NetCore assemblies are independent modules, I can test them using one technique, and then test the Client UI with another technique.

| Module | Test Method(s) | Description |
| --- | --- | --- |
| NetCore | Black Box | I intend to make a simple project to test this module, which sends various messages to the server and reports if any responses are incorrect. This will also allow me to stress test the server to see if it can handle high load, and check for memory leaks which are a major issue in servers. |
| Data | Black Box | As this module doesn’t have much built-in logic, it simply provides the classes for other projects to use, this won’t take much testing. |
| Client | Mixed-level, Integration | Some parts of this module will be fairly free of bugs, while others may be heavily affected. Using mixed-level testing I can dedicate more time to the high risk areas. This module integrates heavily with NetCore, so needs to be tested for bugs related to the interfacing. |
| Server | White Box, Integration | As perhaps the most important module, I will test this assembly thoroughly using White Box testing to make sure I can catch any obvious bugs. I will also use integration testing as this module references the NetCore and Data modules, so is the most likely to suffer from bugs caused by the interfacing. |
| Shared | Black Box | This assembly provides only very basic algorithms – the majority of the code is providing interfaces, which don’t specify any implementation and are used by other assemblies to increase code re-use. As such, this module will not need extensive testing, and I can infer that it works by seeing if the function calls by other modules return the correct value. |

# Testing

## Test Plan

I will test my application with a combination of typical, erroneous, and boundary data. For operations such as testing that pressing a button opens the correct window, it’s impossible to enter invalid data, so I won’t mark these inputs with their type. For tests where actual variable user input is entered, I will annotate the input with the type of data being entered, for clarity. The types of data are as follows:

* Typical: This is usually acceptable input, tests should be successful when run with this as input.
* Erroneous: This is invalid data for some reason, most often due to incorrect format or content.
* Boundary: This is data just outside the range of valid inputs, to test that bounds checking is correct.
* Duplicate: This is data which should be unique but already exists elsewhere in the system (see “Minimal Test Plan” for a list of data already in the system – each test is run with only the minimal data present, a “clean slate”).

The following table details all the tests I’ve recorded, with their number, description, expected input, expected output and real output. Thick black lines between rows separate the tests into rough “topics” of test, such as testing Client responses to messages, or tests related to opening windows etc.

| Test No. | Description | Input | Expected Output | Actual Output |
| --- | --- | --- | --- | --- |
|  | As a student or teacher, the taskbar context menu has the correct items. | Right click on application’s taskbar icon. | One option, “View Bookings”. | As expected (see evidence 1). |
|  | As an admin, the taskbar context menu has the correct items. | Right click on application’s taskbar icon. | Three options, “View Bookings”, “Customise system”, “Exit”. | As expected (see evidence 2). |
|  | Clicking the taskbar icon opens the Timetable window. | Left click on the application’s taskbar icon | Timetable window opens. | As expected (see evidence 3). |
|  | Clicking the menu item opens the Timetable window. | Right click on the taskbar icon, left click on the “Timetable” option. | Timetable window opens. | As expected. |
|  | Clicking the taskbar icon with the Timetable window already open brings it to the front. | Left click on the application’s taskbar icon | Timetable window gains focus. | As expected. |
|  | Clicking the menu item with the Timetable window already open brings it to the front. | Right click on the taskbar icon, left click on the “Timetable” option. | Timetable window gains focus. | As expected. |
|  | When the application starts in a room that’s been booked for the current period, a balloon is shown with a message. | Open the client on a matching room/time to a booking (edited the booking to match the time for ease of testing) | Balloon message appears displaying booking details. | As expected (see evidence 7). |
|  | When the application is open and a booking is made/is entered in the current room and time, a message is shown. | Wait for the time to enter a period. | Balloon message appears displaying booking details. | As expected. |
|  | Timetable window displays bookings with the correct Subject colour and details. | Opening the Timetable window on a day with Bookings set. | Bookings have correct colour and details. | As expected (see evidence 9). |
|  | Bookings are displayed correctly if they recur (eg weekly) | Observing the timetable a long way in the future to check the recurrence still works. | Recurring bookings show up on the timetable. | As expected (see evidence 10). |
|  | Hovering over a booking tile in the Timetable Window darkens the colour. | Hover mouse over a booking on the timetable. | Tile colour darkens. | As expected (see evidence 11). |
|  | Hovering over a room on the timetable displays information | Hover mouse over a room on the timetable. | Tooltip shows the number of seats and special seats. | As expected (see evidence 12). |
|  | Booking Tiles play an animation if the current user is involved in them. | Opening the Timetable window when a set booking involves the current user, either as a student or teacher. | Tile text fades then reappears slowly. | As expected (see evidence 13). |
|  | Closing the Timetable window doesn’t close the application, keeping the icon in the taskbar. | Closing the timetable window. | Window closes, taskbar icon remains. | As expected. |
|  | The “Next Day/Previous Day” buttons work correctly. | Pressing the “Next Day/Previous Day” buttons on the Timetable window. | Timetable shows correct bookings, day and date of timetable are displayed. | As expected. |
|  | Teachers can open the “EditBooking” window by clicking an empty tile. | Clicking a booking tile (when logged in as a teacher or admin). | “EditBooking” window appears (“Teacher” drop-down menu uneditable). | As expected. |
|  | Teachers cannot open other teacher's bookings by clicking their tiles. | When logged in on a teacher account, click on a tile representing a Booking made by a different teacher. | No window opens. | As expected. |
|  | Admins can open the "EditBooking" window by clicking an empty tile. | When logged in on an admin account, click on an empty tile. | Edit Booking window opens (“Teacher” drop-down menu editable). | As expected. |
|  | Admins can open other user's bookings by clicking their tiles. | When logged in on an admin account, click on a tile representing a booking not made by the current user. | Edit Booking window opens (“Teacher” drop-down menu editable). | As expected. |
|  | Students cannot open the “EditBooking” window from the Timetable window. | Clicking a booking tile (when logged in as a student). | Nothing happens | As expected. |
|  | Creating a new Booking with the “EditBooking” window has initial data set (Logged in teacher, selected room, selected period). | Opening the “EditBooking” window by clicking on an empty tile. | “EditBooking” window appears with some data fields already filled in. | As expected (see evidence 21). |
|  | Editing an existing Booking with the “EditBooking” window has the correct data from the Booking filled in. | Opening the “EditBooking” window by clicking on a tile with an existing booking. | “EditBooking” window appears with all data fields filled in correctly. | As expected (see evidence 22). |
|  | Cancelling creating/editing a booking doesn’t submit it by mistake. | Pressing the “Back” or “Close” button on the “EditBooking” window. | Window closes, no other action takes place. | As expected. |
|  | Omitting details while trying to submit a Booking displays an informative error message. | Pressing the “Submit” button with incomplete data on the “EditBooking” window. | Message box with a useful message appears, Booking isn’t submitted. | As expected (see evidence 24). |
|  | The filter on the Students section of the Edit Booking window works correctly. (See evidence for a print screen of initial test data). The filter text has no restrictions on it, so no validation tests are needed. | {“”, No Filter, “All Students”}  {“”, Checked, “All Students”} {“”, Unchecked, “All Students”} {“e”, First Name, “All Students”} {“or”, Last Name, “All Students”} {“m”, Form, “All Students”} {“11”, Year, “All Students”} {“a”, First Name, “Computing”} | {List of all students}  {No students} {All students} {All students with an e in their name} {Max Norman}  {Dan, Mia, Isobel} {Kaleb, Sam,Isobel} {Max, Dan} | All as expected (see evidence 25 for screenshots of test data 1, 6, and 8). |
|  | Pressing Delete on the EditBooking window causes the Server to delete the booking from the internal model and distribute it to all clients. | Pressing the “Delete” button on the EditWindow of an existing booking. | Window closes, server receives a delete booking message, clients remove the booking from their timetable. | As expected ( see evidence 26). |
|  | Submitting a booking that conflicts with other bookings (eg overlapping rooms), results in an error message being displayed. | Pressing the “Submit” button with some of the rooms selected clashing with other bookings in the same period. | Message box displayed showing an error message, Booking isn’t submitted. | As expected (see evidence 27). |
|  | Submitting a booking causes the server to receive the booking, add it to the database, and distribute it to all clients. | Pressing the “Submit” button on the “EditBooking” window. | Window closes, server receives booking message, clients update their timetable view to reflect the new Booking. | As expected (see evidence 28). |
|  | Making a change to a Booking sends an automated email to the teacher who made the Booking. | Pressing the Submit/Delete button on the “EditBooking” window, either creating, editing, or deleting a booking. | Email dispatched to teacher with useful information. | As expected (see output 29). |
|  | Clicking the “Customise System” option on the taskbar context menu opens the Admin Control window. | Click the “Customise System” option on the taskbar context menu. | Admin Control window appears. | As expected (see evidence 30). |
|  | Clicking the “Customise System” option on the taskbar context menu with the Admin Control window open brings it to the front. | Right click on the taskbar icon, left click on the “Customise System” option. | Admin window gains focus. | As expected. |
|  | Clicking “Add Room” on the Rooms tab of the Admin Control window displays the right window. | Click “Add Room” on the Rooms tab on the Admin Control. | Edit Room window is displayed, fields blank. | As expected (see evidence 32). |
|  | Clicking “Edit Room” on the Rooms tab of the Admin Control displays the right window. | Click “Edit Room” on the Rooms tab on the Admin Control. | Edit Room window appears, fields prefilled. | As expected. |
|  | Clicking “Delete Room” on the Rooms tab of the Admin Control checks for conflicts and deletes necessary Entities. | Click “Delete Room” on the Rooms tab on the Admin Control. | Confirmation dialog appears, selecting yes removes all dependent entities from the server. | As expected. |
|  | Entering a name that already exists in the Name field on the Edit Room window is handled with an error message. | “Library” [Duplicate] “D12” [Duplicate] “SF6” [Typical] | Error Error Accepted | As expected (see evidence 35). |
|  | Leaving a field blank on the Edit Room window is handled with an error message. | Leaving any field on the form blank (except for the “Special Seat Type” field if the “Special Seats” field is set to 0. | Error dialogue displayed . | As expected (see evidence 36). |
|  | Entering invalid input to the Standard/Special Seats fields on the Edit Room window is handled with an error message. | -100 [Boundary] ab172H#1¬ [Erroneous] 1.5 [Erroneous] 20 [Typical] 0 [Typical] | Error Error Error Accepted Accepted | As expected (see evidence 37). |
|  | Clicking “Add Period” on the Periods tab of the Admin Control window displays the right window. | Click “Add Period” on the Periods tab on the Admin Control. | Edit Period window appears, fields blank. | As expected (see evidence 38) |
|  | Clicking “Edit Period” on the Periods tab of the Admin Control displays the right window. | Click “Edit Period” on the Periods tab on the Admin Control. | Edit Period window appears, fields prefilled. | As expected. |
|  | Clicking “Delete Period” on the Periods tab of the Admin Control checks for conflicts and deletes necessary Entities. | Click “Delete Period” on the Periods tab on the Admin Control. | Confirmation dialog appears, selecting yes removes all dependent entities from the server. | As expected. |
|  | Entering a name that already exists in the Name field on the Edit Period window is handled with an error message. | “Period 1” [Duplicate] “Period 5” [Duplicate] “Lunchtime” [Typical] | Error Error Accepted | As expected (see evidence 41). |
|  | Leaving a field blank on the Edit Period window is handled with an error message. | Leaving any of the fields on the form empty. | Error dialogue displayed. | As expected. |
|  | Entering invalid input to the Start/End time fields on the Edit Period window is handled with an error message. | “12:10” [Typical] “25:59” [Boundary] “1230” [Erroneous] “12:104” [Erroneous] “ab:13” [Erroneous] | Accepted  Error  Error  Error  Error | As expected (see evidence 43). |
|  | Clicking “Add Teacher” on the Teachers tab of the Admin Control window displays the right window. | Click “Add Teacher” on the Teachers tab on the Admin Control. | Edit Teacher window appears, fields blank. | As expected (see evidence 44) |
|  | Clicking “Edit Teacher” on the Teachers tab of the Admin Control displays the right window. | Click “Edit Teacher” on the Teachers tab on the Admin Control. | Edit Teacher window appears, fields prefilled. | As expected. |
|  | Clicking “Delete Teacher” on the Teachers tab of the Admin Control checks for conflicts and deletes necessary Entities. | Click “Delete Teacher” on the Teachers tab on the Admin Control. | Confirmation dialog appears, selecting yes removes all dependent entities from the server. | As expected. |
|  | Leaving a field blank on the Edit Teacher window is handled with an error message. | Leaving any of the fields on the form blank. | Error dialogue displayed. | As expected. |
|  | Entering invalid input to the Email field on the Edit Teacher window is handled with an error message. | “test5@email.com” [Typical] “@email.com” [Erroneous] “mail@.com” [Erroneous] | Accepted Error Error | As expected (see evidence 48). |
|  | Clicking “Add Student” on the Students tab of the Admin Control window displays the right window. | Click “Add Student” on the Students tab on the Admin Control. | Edit Student window appears, fields blank. | As expected (see evidence 49). |
|  | Clicking “Edit Student” on the Students tab of the Admin Control displays the right window. | Click “Edit Student” on the Students tab on the Admin Control. | Edit Student window appears, fields prefilled. | As expected. |
|  | Clicking “Delete Student” on the Students tab of the Admin Control checks for conflicts and deletes necessary Entities. | Click “Delete Student” on the Students tab on the Admin Control. | Confirmation dialog appears, selecting yes removes all dependent entities from the server. | As expected. |
|  | Leaving a field blank on the Edit Student window is handled with an error message. | Leaving any field blank on the form blank. | Error dialogue displayed. | As expected. |
|  | Clicking “Add Department” on the Departments tab of the Admin Control window displays the right window. | Click “Add Department” on the Departments tab on the Admin Control. | Edit Department window appears, fields blank. | As expected (see evidence 53). |
|  | Clicking “Edit Department” on the Departments tab of the Admin Control displays the right window. | Click “Edit Department” on the Departments tab on the Admin Control. | Edit Department window appears, fields prefilled. | As expected. |
|  | Clicking “Delete Department” on the Departments tab of the Admin Control checks for conflicts and deletes necessary Entities. | Click “Delete Department” on the Students tab on the Admin Control. | Confirmation dialog appears, selecting yes removes all dependent entities from the server. | As expected. |
|  | Entering a name that already exists in the Name field on the Edit Department window is handled with an error message. | “Science” [Duplicate] “Maths” [Duplicate] “DT” [Typical] “PE” [Typical] | Error Error Accepted Accepted | As expected. |
|  | Leaving a field blank on the Edit Department window is handled with an error message. | Leaving any field on the form empty. | Error dialogue displayed. | As expected. |
|  | Clicking “Add Class” on the Classes tab of the Admin Control window displays the right window. | Click “Add Class” on the Classes tab on the Admin Control. | Edit Class window appears, fields blank. | As expected (see evidence 58). |
|  | Clicking “Edit Class” on the Classes tab of the Admin Control displays the right window. | Click “Edit Class” on the Classes tab on the Admin Control. | Edit Class window appears, fields prefilled. | As expected. |
|  | Clicking “Delete Class” on the Classes tab of the Admin Control checks for conflicts and deletes necessary Entities. | Click “Delete Class” on the Classes tab on the Admin Control. | Confirmation dialog appears, selecting yes removes all dependent entities from the server. | As expected. |
|  | Entering a name that already exists in the Name field on the Edit Class window is handled with an error message. | “Computing Yr13” [Duplicate] “Maths Yr12” [Duplicate] “Maths Yr13” [Typical] “Physics AS” [Typical] | Error Error Accepted Accepted | As expected. |
|  | Leaving a field blank on the Edit Class window is handled with an error message. | Leaving any field on the window blank. | Error dialogue displayed. | As expected. |
|  | Clicking “Add Subject” on the Subjects tab of the Admin Control window displays the right window. | Click “Add Subject” on the Subjects tab on the Admin Control. | Edit Subject window appears, fields blank. | As expected (see evidence 63). |
|  | Clicking “Edit Subject” on the Subjects tab of the Admin Control displays the right window. | Click “Edit Subject” on the Classes tab on the Admin Control. | Edit Subject window appears, fields prefilled. | As expected. |
|  | Clicking “Delete Subject” on the Subjects tab of the Admin Control checks for conflicts and deletes necessary Entities. | Click “Delete Subject” on the Classes tab on the Admin Control. | Confirmation dialog appears, selecting yes removes all dependent entities from the server. | As expected. |
|  | Entering a name that already exists in the Name field on the Edit Subject window is handled with an error message. | “Physics” [Duplicate] “Maths” [Duplicate] “Geography” [Typical] “Games” [Typical] | Error Error Accepted Accepted | As expected. |
|  | Leaving a field blank on the Edit Subject window is handled with an error message. | Leaving any field on the window empty. | Error dialogue displayed. | As expected. |
|  | Entering invalid input to the Red/Green/Blue component fields on the Edit Subject window is handled with an error message. | “-20” [Erroneous] “256” [Boundary] “abhy6~=” [Erroneous] “asdf” [Erroneous] “128” [Typical] “255” [Typical] | Error  Error  Error  Error  Accepted Accepted | As expected. |
|  | The colour demonstrator on the Edit Subject Window displays the indicated colour in real-time. | Change the values in the component fields digit by digit. | As each digit is typed the demonstration box should change colour. | As expected (see evidence 69). |
|  | Upon connecting to a Server, the Client sends a message containing relevant information about the logged on user. | Start up a Client and allow it to connect to a running Server (ensure correct target IP and Port etc). | Client sends message, Server receives message. | As expected (see evidence 70). |
|  | Upon receiving a log-on message from a Client, the Server responds with a message containing user information from the database. | After the previous test, the Server will automatically respond with a message. | Server sends message, Client receives message. | As expected. |
|  | Clicking "Submit" on the Edit Room Window sends a message to the server with the new item. | Create/Edit a Room in the Edit Room window, then hit “Submit”. | Message is sent from Client containing the Room and whether to delete or not. | As expected. |
|  | Clicking "Submit" on the Edit Period Window sends a message to the server with the new item. | Create/Edit a Period in the Edit Period window, then hit “Submit”. | Message is sent from Client containing the Period and whether to delete or not. | As expected. |
|  | Clicking "Submit" on the Edit Teacher Window sends a message to the server with the new item. | Create/Edit a Teacher in the Edit Teacher window, then hit “Submit”. | Message is sent from Client containing the Teacher and whether to delete or not. | As expected. |
|  | Clicking "Submit" on the Edit Student Window sends a message to the server with the new item. | Create/Edit a Student in the Edit Student window, then hit “Submit”. | Message is sent from Client containing the Student and whether to delete or not. | As expected. |
|  | Clicking "Submit" on the Edit Department Window sends a message to the server with the new item. | Create/Edit a Department in the Edit Department window, then hit “Submit”. | Message is sent from Client containing the Department and whether to delete or not. | As expected. |
|  | Clicking "Submit" on the Edit Class Window sends a message to the server with the new item. | Create/Edit a Class in the Edit Class window, then hit “Submit”. | Message is sent from Client containing the Class and whether to delete or not. | As expected. |
|  | Clicking "Submit" on the Edit Subject Window sends a message to the server with the new item. | Create/Edit a Subject in the Edit Subject window, then hit “Submit”. | Message is sent from Client containing the Subject and whether to delete or not. | As expected. |
|  | Upon receiving a Room object, the Server updates the database appropriately. | Message sent from the Client to the Server containing details of a change of a Room object. | Server receives message, updates database. | As expected (see evidence 79). |
|  | Upon receiving a Period object, the Server updates the database appropriately. | Message sent from the Client to the Server containing details of a change of a Period object. | Server receives message, updates database. | As expected. |
|  | Upon receiving a Teacher object, the Server updates the database appropriately. | Message sent from the Client to the Server containing details of a change of a Teacher object. | Server receives message, updates database. | As expected. |
|  | Upon receiving a Student object, the Server updates the database appropriately. | Message sent from the Client to the Server containing details of a change of a Student object. | Server receives message, updates database. | As expected. |
|  | Upon receiving a Department object, the Server updates the database appropriately. | Message sent from the Client to the Server containing details of a change of a Department object. | Server receives message, updates database. | As expected. |
|  | Upon receiving a Class object, the Server updates the database appropriately. | Message sent from the Client to the Server containing details of a change of a Class object. | Server receives message, updates database. | As expected. |
|  | Upon receiving a Subject object, the Server updates the database appropriately. | Message sent from the Client to the Server containing details of a change of a Subject object. | Server receives message, updates database. | As expected. |
|  | Upon updating the database with a Room object, the Server distributes a message to all clients containing the changed item. | Server updates the database with a received item. | Server sends message to Clients containing details of change, Clients receive message. | As expected. |
|  | Upon updating the database with a Period object, the Server distributes a message to all clients containing the changed item. | Server updates the database with a received item. | Server sends message to Clients containing details of change, Clients receive message. | As expected. |
|  | Upon updating the database with a Teacher object, the Server distributes a message to all clients containing the changed item. | Server updates the database with a received item. | Server sends message to Clients containing details of change, Clients receive message. | As expected. |
|  | Upon updating the database with a Student object, the Server distributes a message to all clients containing the changed item. | Server updates the database with a received item. | Server sends message to Clients containing details of change, Clients receive message. | As expected. |
|  | Upon updating the database with a Department object, the Server distributes a message to all clients containing the changed item. | Server updates the database with a received item. | Server sends message to Clients containing details of change, Clients receive message. | As expected. |
|  | Upon updating the database with a Class object, the Server distributes a message to all clients containing the changed item. | Server updates the database with a received item. | Server sends message to Clients containing details of change, Clients receive message. | As expected. |
|  | Upon updating the database with a Subject object, the Server distributes a message to all clients containing the changed item. | Server updates the database with a received item. | Server sends message to Clients containing details of change, Clients receive message. | As expected. |
|  | Upon receiving a message from the Server with a new Room, the Client updates its internal model. | Message received from Server containing the details of a change in the model. | Client updates internal model of the databse to reflect the changes made. | As expected. |
|  | Upon receiving a message from the Server with a new Period, the Client updates its internal model. | Message received from Server containing the details of a change in the model. | Client updates internal model of the databse to reflect the changes made. | As expected. |
|  | Upon receiving a message from the Server with a new Teacher, the Client updates its internal model. | Message received from Server containing the details of a change in the model. | Client updates internal model of the databse to reflect the changes made. | As expected. |
|  | Upon receiving a message from the Server with a new Student, the Client updates its internal model. | Message received from Server containing the details of a change in the model. | Client updates internal model of the databse to reflect the changes made. | As expected. |
|  | Upon receiving a message from the Server with a new Department, the Client updates its internal model. | Message received from Server containing the details of a change in the model. | Client updates internal model of the databse to reflect the changes made. | As expected. |
|  | Upon receiving a message from the Server with a new Class, the Client updates its internal model. | Message received from Server containing the details of a change in the model. | Client updates internal model of the databse to reflect the changes made. | As expected. |
|  | Testing server capabilities under extreme load. | Open numerous clients on multiple computers and connect them all to the server, perform standard operations such as editing bookings/teachers. | System runs as normal, no crashes or exceptions thrown. | As expected. |

## Test Results - Evidence



1. As a student or teacher, the taskbar context menu displays the correct items when the icon is right clicked.
2. As an admin, the taskbar context menu displays the correct items when the icon is clicked.
3. Clicking the icon on the taskbar opens the Timetable window on the current date (also evidence of clicking the “Timetable” item on the right-click context menu shown before).



1. When the application starts in a room that’s been booked for the current period, a balloon is shown with a warning message.



1. The Timetable window displays bookings with the correct Subject colour and details, and in the correct Period/Room location.
2. Bookings are displayed correctly if they recur over multiple weeks/fortnights/months.

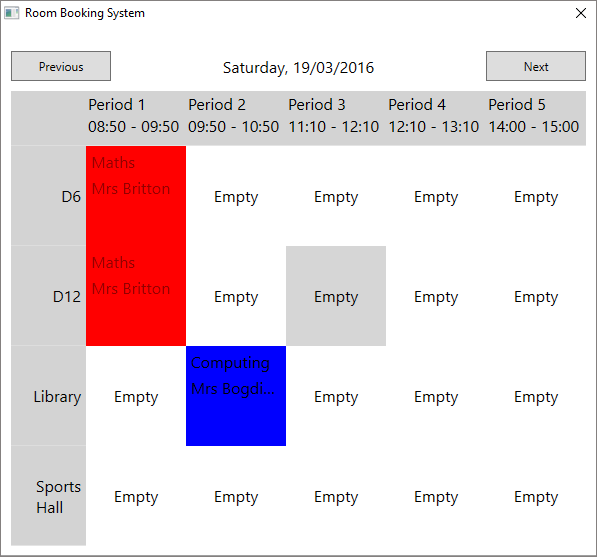
 



1. Hovering over a tile on the Timetable window darkens the colour.



1. Hovering over a room tile on the timetable displays relevant information about the booking.



1. Booking Tiles play an animation if the current user is involved in them.

1. As a Teacher/Admin, creating a new Booking with the EditBooking window initialises the window with information that can be inferred from how the window was opened (eg. Room, period, teacher etc).

1. Editing an existing Booking with the EditBooking window initialises the window with all the information of the existing booking.
2. Omitting details while trying to submit a Booking displays an informative error message in a modal dialog box.
3. The filter on the Students section of the EditBooking window works correctly (a variety and combination of filters all function appropriately).



The initial set of students, with no filters applied.



Filtering all students by form – using the letter “M” will perform a case-insensitive filter for any forms containing the letter “M”.



Filtering only the “Computing Yr 13” class (subset of all students), by “First Name” containing “A”. This shows that the aggregation of filters still works appropriately.

1. Pressing delete on the EditBooking window displays a confirmation dialog, and causes the server to delete the Booking from the internal model, update the database, and distribute the changes to all connected clients. Clients connecting subsequent to the deletion are also given the correctly changed model.





1. Submitting a booking that conflicts with other bookings (eg overlapping rooms/timeslots), results in an error message being displayed.
2. Submitting a Booking causes the Server to receive and process the Booking, adding/editing the database records, and distributing it to all the connected clients. Again, due to the database updates, new clients connecting will receive the correct model as well.

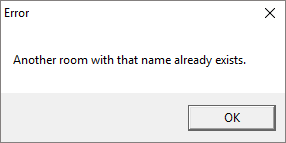




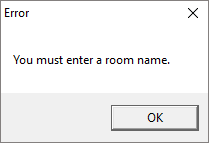
1. Making a change to a Booking sends an informative notification email to the relevant teacher.
2. Clicking the “Customise System” option on the taskbar context menu opens the Admin Control window.



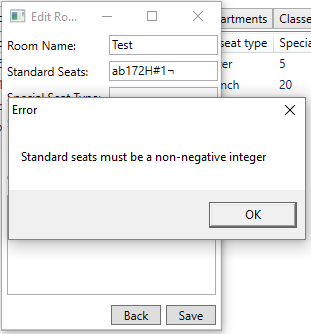
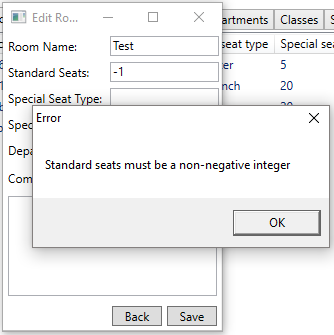
1. Clicking “Add Room” on the Rooms tab of the Admin Control window displays the correct window.



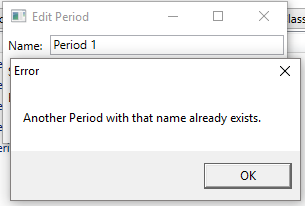
1. Entering a name that already exists in the Name field on the Edit Room window is handled with an error message.



1. Leaving a field blank on the Edit Room window is handled with an error message.



1. Entering invalid input to the Standard/Special seats fields on the Edit Room window is handled with an informative error message.
2. Clicking “Add Period” on the Periods tab of the Admin Control window displays the correct window.



1. Entering a name that already exists in the Name field on the Edit Period window is handled with an error message.



1. Entering invalid input to the Start/End time fields on the Edit Period window is handled with an informative error message.



1. Clicking “Add Teacher” on the Teachers tab of the Admin Control window displays the correct window.



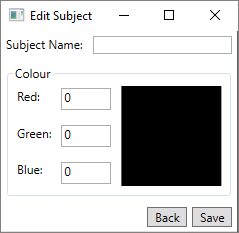
1. Entering invalid input to the Email field on the Edit Teacher window is handled with an informative error message.



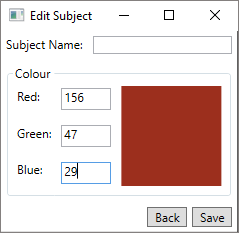
1. Clicking “Add Student” on the Students tab of the Admin Control window displays the correct window.



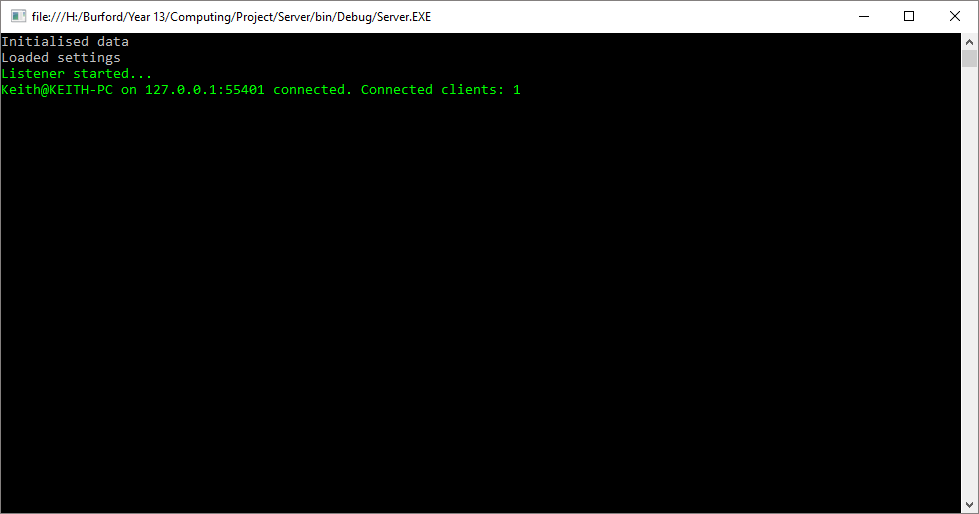
1. Clicking “Add Department” on the Departments tab of the Admin Control window displays the correct window.
2. Clicking “Add Class” on the classes tab of the Admin Control window displays the correct window.

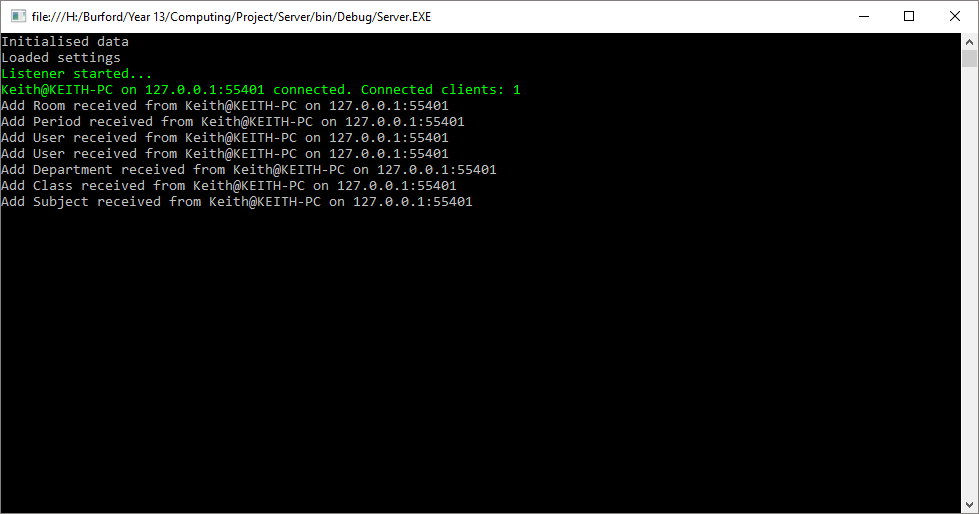


1. Clicking “Add Subject” on the Subjects tab of the Admin Control window displays the right window.



1. The colour demonstrator on the Edit Subject Window displays the indicated colour in real-time.
2. Upon connecting to a Server, the Client sends a message containing relevant information about the logged on user.



1. Upon receiving a Room object, the Server updates the database appropriately.

## Minimum Test Data

In order to test my application works in general usage, as opposed to under specific testing conditions, while running these tests I’ve loaded the system with number of sample data entries. For example, I have added “dummy” users, classes, rooms, etc. so that when testing the functionality of my system I can ensure that it scales correctly. Wherever a test involves data being sent across the network, I have set up multiple clients connected to the server at the same time, to ensure the server can deal with the load and be able to distribute data correctly.

The initial test data used is listed below in tables relating to the tables stored in the database. Columns with titles in *italics* imply the field is a foreign key, and provide the Id of the record in the related table.

Departments

|  |  |  |  |
| --- | --- | --- | --- |
| Id | Name | *Teachers* | *Rooms* |
| 1 | “Maths” | Id: 4 | Id: 4 |
| 2 | “Science” | Id: 3 | Id: 3 |
| 3 | “Computing/IT” | Id: 1, 2 | Id: 1, 2 |
| 4 | “History” | Id: 5 | *Empty* |

Rooms

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Id | RoomName | StandardSeats | SpecialSeats | SpecialSeatType | *Department* | ComputerNames | *Bookings* |
| 1 | “D6” | 10 | 5 | “Computer” | Id: 3 | “D6D00”,…,”D6D30” | Id: 1 |
| 2 | “D12” | 5 | 20 | “Workbench” | Id: 3 | “D12D00”,…,”D12D30” | Id: 1 |
| 3 | “Library” | 30 | 20 | “Computer” | Id: 2 | *Empty* | Id: 2 |
| 4 | “Sports Hall” | 100 | 0 | *Empty* | Id: 1 | *Empty* | *Empty* |

Teachers

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Id | Title | FirstName | LastName | Logon- Name | Email | AccessMode | *Depart-ment* | Bookings | Classes |
| 1 | “Mrs” | “Mary” | “Bogdiukiewicz” | “mb” | “hnefatl@gmail.com” | Teacher | Id: 3 | Id: 2 | Id: 1 |
| 2 | “Mr” | “Patrick” | “Count” | “pc” | “hnefatl@gmail.com” | Teacher | Id: 3 | *Empty* | *Empty* |
| 3 | “Mr” | “James” | “Kenny” | “jk” | “hnefatl@gmail.com” | Teacher | Id: 2 | *Empty* | *Empty* |
| 4 | “Mrs” | “Rosemary” | “Britton” | “rb” | “hnefatl@gmail.com” | Teacher | Id: 1 | Id: 1 | Id: 2,3 |
| 5 | “Mrs” | “Emma” | “Denny” | “ed” | “hnefatl@gmail.com” | Teacher | Id: 4 | *Empty* | *Empty* |

All email addresses are set to mine for testing purpose, so I’ll receive any emails sent by the system.

Subjects

|  |  |  |  |
| --- | --- | --- | --- |
| Id | Name | Colour | *Bookings* |
| 1 | “Maths” | FFFF000016 | Id: 1  Colours are stored as a 32-bit integer, so I’ve represented the colours here using hex notation and in the colour they represent. |
| 2 | “Physics” | FFFFA50016 | *Empty* |
| 3 | “Computing” | FF0000FF16 | Id: 2 |
| 4 | “History” | FF00800016 | *Empty* |

Periods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Id | Name | Start | End | *Bookings* |
| 1 | “Period 1” | 08:50 | 09:50 | Id: 1 |
| 2 | “Period 2” | 09:50 | 10:50 | Id: 2 |
| 3 | “Period 3” | 11:10 | 12:10 | *Empty* |
| 4 | “Period 4” | 12:10 | 13:10 | *Empty* |
| 5 | “Period 5” | 14:00 | 15:00 | *Empty* |

Students

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Id | FirstName | LastName | Form | Year | LogonName | AccessMode | *Bookings* | *Classes* |
| 1 | “Keith” | “Collister” | “WT” | 13 | 09135 | Admin | Id: 1 | Id: 1,2 |
| 2 | “Dan” | “Wrenn” | “MB” | 13 | 09154 | Teacher | Id: 2 | Id: 1 |
| 3 | “Euan” | “Rossie” | “WT” | 13 | 09185 | Student | Id: 1 | Id: 2 |
| 4 | “Max” | “Norman” | “WT” | 13 | 09001 | Student | Id: 1 | Id: 1,2 |
| 5 | “Peter” | “Champion” | “WT” | 13 | 09002 | Student | Id: 1 | Id: 1,2 |
| 6 | “Mia” | “West” | “MB” | 13 | 09003 | Student | Id: 2 | *Empty* |
| 7 | “Matthew” | “Pilkington” | “WT” | 13 | 09004 | Student | Id: 1 | Id: 2 |
| 8 | “Kaleb” | “Poole” | “BI” | 11 | 11001 | Student | *Empty* | Id: 3 |
| 9 | “Sam” | “Kitto” | “BI” | 11 | 11002 | Student | *Empty* | Id: 3 |
| 10 | “Isobel” | “Stephens” | “CR” | 11 | 11003 | Student | *Empty* | Id: 3 |

Classes

|  |  |  |  |
| --- | --- | --- | --- |
| Id | ClassName | *Owner* | *Students* |
| 1 | “Computing Yr13” | Id: 1 | Id: 1, 2, 4, 5 |
| 2 | “Maths Yr13” | Id: 4 | Id: 1, 3, 4, 5, 7 |
| 3 | “History Yr11” | Id: 4 | Id: 8, 9, 10 |

Bookings

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Id | Date | BookingType | *TimeSlot* | *Rooms* | *Subject* | *Teacher* | *Students* |
| 1 | *Current* | Single | Id: 1 | Id: 1, 2 | Id: 1 | Id: 4 | Id: 1, 3, 4, 5, 7 |
| 2 | *Current* | Weekly | Id: 2 | Id: 3 | Id: 3 | Id: 1 | Id: 2, 6 |

The *Date* field of each booking is manually set to the date of the test before testing is begun, so that the system’s faster to use.

# System Maintenance

## Introduction

The objective of the project, as stated in the Design section, was to provide a robust, scalable internal system enabling teachers to book rooms easily while giving students a degree of access to the system, providing notifications and information on bookings. It was decided to achieve this using a networked Client-Server system running inside the school network. The Client executables run on computer log-on, and connect to the Server running on a central host machine using a custom protocol on top of TCP/IP.

The Server keeps a database of Bookings (along with all other components of the system to allow customisability, such as Rooms and Periods) and updates it with necessary changes, and is responsible for receiving, updating, and distributing the changes. This is the workhorse of the system.

The Client is the front-end displayed to all users (Admin, Teacher, and Student roles). It sends data to the Server concerning changes to bookings etc, and displays the Timetable with bookings for the users. This is the “pretty front-end”.

There are 3 separate assemblies referenced by the two main applications – *Shared*, *Data*, and *NetCore*. These each provide some shared components, and are more or less independent logical structures. They’re described in more detail below.

Database interaction is done using the .NET Entity Framework (<http://www.asp.net/entity-framework>). The Server is the only application that interacts with the database – Clients receive a model of the internal model from the Server.

## Description of Modular System

The system has been designed to reuse as much code as possible, which coupled with the common tasks of networking from both the Server and the Client, has allowed for the creation of shared libraries of code referenced by the various other assemblies. The diagram on the right displays this structure, the arrows representing the references between different assemblies of code in the solution. These match up with the links as proposed in the Design section.

## Annotated Source Code

The following is the Annotated Source Code for the solution – it is divided into 5 subsections, which represent the 5 assemblies in the project (Shared, Data, NetCore, Server, Client). Each section has a class diagram showing the general structure and cursory information about each module, followed by the actual source code. This has been commented fully, but there are also textboxes containing more detailed explanations of certain sections. There is also a small textbox describing the purpose of each class at the top of each new source code block.

There are over 150 pages of source code listed here, so at the end of this section I’ve included a subsection titled “**Explanation of complex algorithms**”, which aims to provide an easier-to-read, more in depth look at some of the most important algorithms in the project, with full explanations. I urge readers to look at this section, as it more clearly showcases the important parts of the project.

## E:\Burford\Year 13\Computing\Project\_Writeup\Resources\Maintenance\Shared.pngShared

The Shared library holds some classes which are usually the highest form of abstraction – for example, the *Writer* abstract class provides an abstraction for any classes that can write to a Stream. An initial implementation, *TextWriter* is given within the same assembly, and provides simple text output to a Stream. This is used in writing to the Settings file. The *Reader* class is the input analogue to the *Writer* class.

Another key class provided by this assembly is the *ISerialisable* interface – this is used by the Data and NetCore assemblies to provide a neatly abstracted way of serialising various objects (network messages, primarily).

The *Helpers* static class is used only for converting from a .NET *Color* object to an *Int32*, as databases cannot store most complex CLR types, only primitive types such as *Int32* etc.

Shared.Writer

The *Writer* class is an abstract superclass for the Stream IO derived classes.  
It provides the interface that all the subclasses must implement.

using System;

using System.IO;

namespace Shared

{

// Provides an abstract hierarchy for writing .NET objects to a Stream.

public abstract class Writer

: IDisposable

{

// The output stream to write to

The member property *Base* (type *Stream*) inherits *IDisposable*, so must be disposed of appropriately – good practice dictates this class should thus be disposable as well, which makes it easier to deal with disposing of the member.

protected Stream Base { get; set; }

public Writer(Stream Base)

{

this.Base = Base;

}

// Disposes of the output stream

public virtual void Dispose()

{

Base.Flush();

Base.Dispose();

}

These abstract functions dictate that all concrete subclasses of Writer must provide implementations for these standard outputs.

public abstract void Write(byte b);

public abstract void Write(bool b);

public abstract void Write(short s);

public abstract void Write(int i);

public abstract void Write(long l);

public abstract void Write(string s);

// Writes a loosely typed Object if it's of a supported type.

public virtual void Write(object Item)

{

Type t = Item.GetType();

if (t == typeof(byte))

This function allows an *Object* to be written without needing to know the type in advance, by testing it against different types to find a match.

Write((byte)Convert.ChangeType(Item, typeof(byte)));

else if (t == typeof(bool))

Write((bool)Convert.ChangeType(Item, typeof(bool)));

else if (t == typeof(short))

Write((short)Convert.ChangeType(Item, typeof(short)));

else if (t == typeof(int))

Write((int)Convert.ChangeType(Item, typeof(int)));

else if (t == typeof(long))

Write((long)Convert.ChangeType(Item, typeof(long)));

else if (t == typeof(string))

Write((string)Convert.ChangeType(Item, typeof(string)));

else

throw new ArgumentException("Cannot write values of the type specified.");

}

}

}

Shared.TextWriter

using System;

using System.IO;

namespace Shared

{

// Writes data to the output Stream using standard text encoding

public class TextWriter

This is a concrete implementation of the abstract *Writer* class designed to write using plain text. In essence, this is simply a wrapper over a *StreamWriter*, but inherits *Writer*.

: Writer

{

// Internal wrapper object

protected StreamWriter Writer { get; set; }

public TextWriter(Stream Base)

: base(Base)

{

Writer = new StreamWriter(Base);

}

public override void Dispose()

{

// Dispose the Writer, then dispose the base class

Writer.Dispose();

base.Dispose();

}

Overrides of all of *Writer*’s abstract functions to provide plain-text output. Data is written newline-delimited, so no metadata about the data itself needs to be sent.

public override void Write(byte b)

{

Writer.WriteLine(b);

}

public override void Write(bool b)

{

Writer.WriteLine(b);

}

public override void Write(short s)

{

Writer.WriteLine(s);

}

public override void Write(int i)

{

Writer.WriteLine(i);

}

public override void Write(long l)

{

Writer.WriteLine(l);

}

public override void Write(string s)

{

Writer.WriteLine(s);

}

}

}

Shared.NetWriter

using System;

This is a concrete implementation of the abstract *Writer* class designed to write using binary. *BinaryWriter* couldn’t be used as it outputs in host rather than network byte order (so data transferred between different endian architectures may be unreadable).

using System.Text;

using System.IO;

using System.Net;

namespace Shared

{

// Write data to a stream using bytes rather than text.

public class NetWriter

: Writer

{

public NetWriter(Stream Base)

:base(Base)

{

}

public virtual void Write(byte[] Data)

These overridden functions simply convert the input into bytes, ensuring they’re in network order (Big Endian), and writes them to the stream.

{

Base.Write(Data, 0, Data.Length);

}

public override void Write(byte b)

{

Write(new byte[] { b });

}

public override void Write(bool b)

{

Write(new byte[] { Convert.ToByte(b) });

}

public override void Write(short s)

{

// Writing to a network, so use Network order conversion

Write(BitConverter.GetBytes(IPAddress.HostToNetworkOrder(s)));

}

public override void Write(int i)

{

Write(BitConverter.GetBytes(IPAddress.HostToNetworkOrder(i)));

}

public override void Write(long l)

{

Write(BitConverter.GetBytes(IPAddress.HostToNetworkOrder(l)));

}

public override void Write(string s)

{

// Write the length of the string, then the actual string data

Write(Encoding.BigEndianUnicode.GetByteCount(s));

Write(Encoding.BigEndianUnicode.GetBytes(s));

To write a variable-length *String*, we first need to send its length so the receiver knows how many subsequent bytes to receive for the actual string contents.

}

}

}

Shared.Reader

using System;

Again, the member property *Base* inherits *IDisposable*, so this class inherits *IDisposable* in order to clean up.

This is a generic function to read an object of known type from the Stream. Provides strongly typed access to the Stream while maintaining a generic interface.

The abstract *Reader* class provides an input analogue to the *Writer* class, creating a wrapper for *Stream* IO.

using System.IO;

namespace Shared

{

// Abstract class providing a hierarchy for reading data from a stream.

public abstract class Reader

: IDisposable

{

// The stream to read from

protected Stream Base { get; set; }

public Reader(Stream Base)

{

this.Base = Base;

}

public virtual void Dispose()

{

Base.Dispose();

}

public abstract byte ReadByte();

public abstract bool ReadBool();

public abstract short ReadInt16();

public abstract int ReadInt32();

public abstract long ReadInt64();

public abstract string ReadString();

// Writes a generic type by checking the type against supported ones.

public virtual T Read<T>()

This generic function provides strongly typed runtime access to the stream by checking the desired type against the allowed types and calling the desired function.

{

Type t = typeof(T);

if (t == typeof(byte))

return (T)Convert.ChangeType(ReadByte(), t);

else if (t == typeof(bool))

return (T)Convert.ChangeType(ReadBool(), t);

else if (t == typeof(short))

return (T)Convert.ChangeType(ReadInt16(), t);

else if (t == typeof(int))

return (T)Convert.ChangeType(ReadInt32(), t);

else if (t == typeof(long))

return (T)Convert.ChangeType(ReadInt64(), t);

else if (t == typeof(string))

return (T)Convert.ChangeType(ReadString(), t);

else

throw new ArgumentException("Cannot read values of the type specified.");

}

}

}

Shared.TextReader

using System;

using System.Text;

*TextReader* is a concrete implementation of *Reader*, designed to be used with a *TextWriter*. Similarly, it’s a wrapper around *StreamReader*.

using System.IO;

namespace Shared

{

// Reads data from a stream using standard encoding.

public class TextReader

: Reader

{

// Internal wrapper object

protected StreamReader Reader { get; set; }

public TextReader(Stream Base)

:base(Base)

{

Reader = new StreamReader(Base);

}

public override void Dispose()

{

// Dispose of the wrapper first, then the base class.

Reader.Dispose();

base.Dispose();

}

public override byte ReadByte()

{

Reading non-string types is achieved by simply parsing each line as the type.

return (byte)Reader.Read();

}

public override bool ReadBool()

{

return bool.Parse(Reader.ReadLine());

}

public override short ReadInt16()

{

return short.Parse(Reader.ReadLine());

}

public override int ReadInt32()

{

return int.Parse(Reader.ReadLine());

}

public override long ReadInt64()

{

return long.Parse(Reader.ReadLine());

}

public override string ReadString()

{

return Reader.ReadLine();

}

}

}

Shared.NetReader

*NetReader* is the largest class in this module. It provides the same style of access to the stream as *TextReader* does, but using binary. This class is also unique amongst the *Reader*/*Writer* hierarchy in that it provides asynchronous methods. These are extremely useful when implementing the network protocols in the *NetCore* assembly.

using System;

using System.Text;

using System.IO;

using System.Net;

namespace Shared

{

// Reads data from a stream using bytes rather than text.

public class NetReader

: Reader

{

public NetReader(Stream Base)

: base(Base)

{

ReadBytes is a relatively complicated function compared to most in this part of the solution – it reads a specified number of bytes into a buffer, returning the array of bytes or throwing an exception if it can’t read as many as are specified.

}

// Read a specified number of bytes

public virtual byte[] ReadBytes(int Count)

{

if (Count <= 0)

return new byte[0];

byte[] Buffer = new byte[Count];

int Remaining = Count;

// Not guaranteed to read all bytes on first try - retry until all read

while (Remaining > 0)

Remaining -= Base.Read(Buffer, Buffer.Length - Remaining, Remaining);

if (Remaining != 0)

throw new Exception("Bad read (read " + (Buffer.Length - Remaining) + " of " + Buffer.Length + " bytes).");

return Buffer;

Reading multi-byte values from the network can cause errors due to the endianness of data being sent/received. Converting such values to Network order (Big endian) eliminates such concerns.

}

public override byte ReadByte()

{

return ReadBytes(1)[0];

}

public override bool ReadBool()

{

return Convert.ToBoolean(ReadByte());

}

public override short ReadInt16()

{

return IPAddress.NetworkToHostOrder(BitConverter.ToInt16(ReadBytes(sizeof(short)), 0));

}

public override int ReadInt32()

{

return IPAddress.NetworkToHostOrder(BitConverter.ToInt32(ReadBytes(sizeof(int)), 0));

}

public override long ReadInt64()

{

return IPAddress.NetworkToHostOrder(BitConverter.ToInt64(ReadBytes(sizeof(long)), 0));

}

public override string ReadString()

{

int Length = ReadInt32(); // Read the length of the data first

return Encoding.BigEndianUnicode.GetString(ReadBytes(Length));

}

// Asynchronous methods using Begin-End paradigm

public virtual IAsyncResult BeginReadBytes(int Count, AsyncCallback Callback)

{

byte[] Buffer = new byte[Count]; // Allocate the memory to be used

// Delegate reading to the stream's async methods, passing the buffer as the state

return Base.BeginRead(Buffer, 0, Buffer.Length, Callback, Buffer);

}

public virtual byte[] EndReadBytes(IAsyncResult Handle)

Reading data asynchronously is useful in some parts of the Client code – wrappers are provided over the *Stream*’s underlying asynchronous functions.

{

// Retrieve the buffer as the state

byte[] Buffer = (byte[])Handle.AsyncState;

// Retrieve the number of bytes read

int Read = Base.EndRead(Handle);

// Check for a valid read (correct number of bytes)

if (Read != Buffer.Length)

throw new Exception("Bad read (read " + Read + " of " + Buffer.Length + " bytes).");

Handle.AsyncWaitHandle.Dispose(); // Cleanup

return Buffer;

}

public virtual IAsyncResult BeginReadByte(AsyncCallback Callback)

{

Asynchronous functions for reading most primitive types are provided, for utility.

return BeginReadBytes(sizeof(byte), Callback);

}

public virtual byte EndReadByte(IAsyncResult Handle)

{

return EndReadBytes(Handle)[0];

}

public virtual IAsyncResult BeginReadBool(AsyncCallback Callback)

{

return BeginReadByte(Callback);

}

public virtual bool EndReadBool(IAsyncResult Handle)

{

return Convert.ToBoolean(EndReadByte(Handle));

}

public virtual IAsyncResult BeginReadInt16(AsyncCallback Callback)

{

return BeginReadBytes(sizeof(short), Callback);

}

public virtual short EndReadInt16(IAsyncResult Handle)

{

return IPAddress.NetworkToHostOrder(BitConverter.ToInt16(EndReadBytes(Handle), 0));

}

public virtual IAsyncResult BeginReadInt32(AsyncCallback Callback)

{

return BeginReadBytes(sizeof(int), Callback);

}

public virtual int EndReadInt32(IAsyncResult Handle)

{

return IPAddress.NetworkToHostOrder(BitConverter.ToInt32(EndReadBytes(Handle), 0));

}

public virtual IAsyncResult BeginReadInt64(AsyncCallback Callback)

{

return BeginReadBytes(sizeof(long), Callback);

}

public virtual long EndReadInt64(IAsyncResult Handle)

{

return IPAddress.NetworkToHostOrder(BitConverter.ToInt64(EndReadBytes(Handle), 0));

}

public virtual IAsyncResult BeginReadString(AsyncCallback Callback)

{

return BeginReadInt32(Callback);

}

public virtual string EndReadString(IAsyncResult Handle)

{

int Length = EndReadInt32(Handle);

return Encoding.BigEndianUnicode.GetString(ReadBytes(Length));

}

// Utility function to determine the number of bytes required to send a particular object.

public static int NetworkLength(object o)

{

This function is used in some Serialisation/Deserialisation methods later on – it calculates the exact number of bytes required to send an object over the *Stream*.

if (o is short)

return sizeof(short);

else if (o is int)

return sizeof(int);

else if (o is long)

return sizeof(long);

else if (o is byte)

return sizeof(byte);

else if (o is string) // Sends size of contents + actual contents

return sizeof(int) + Encoding.BigEndianUnicode.GetByteCount((string)o);

else

throw new NotSupportedException();

}

}

}

Shared.Helpers

The *Helpers* class provides some utility functions that are used by various projects and so need to be accessible from different parts of the solution.

using System;

using System.Windows.Media;

namespace Shared

{

public static class Helpers

{

public static Color IntToColour(int i)

{

byte[] Bytes = BitConverter.GetBytes(i);

return Color.FromArgb(Bytes[3], Bytes[2], Bytes[1], Bytes[0]);

}

public static int ColorToInt(Color c)

{

return BitConverter.ToInt32(new byte[] { c.B, c.G, c.R, c.A }, 0);

}

These two functions provide quick conversions from a CLR *Color* object to an *int* primitive – this is used primarily in the Data assembly in order to store *Color* objects in the database.

}

}

Shared.ISerialisable

The *ISerialisable* interface is used to enforce that a class can be Serialised/Deserialised to/from a *Writer*/*Reader* object from this assembly. This provides a common base interface for classes that is used to enforce polymorphism.

namespace Shared

{

public interface ISerialisable

{

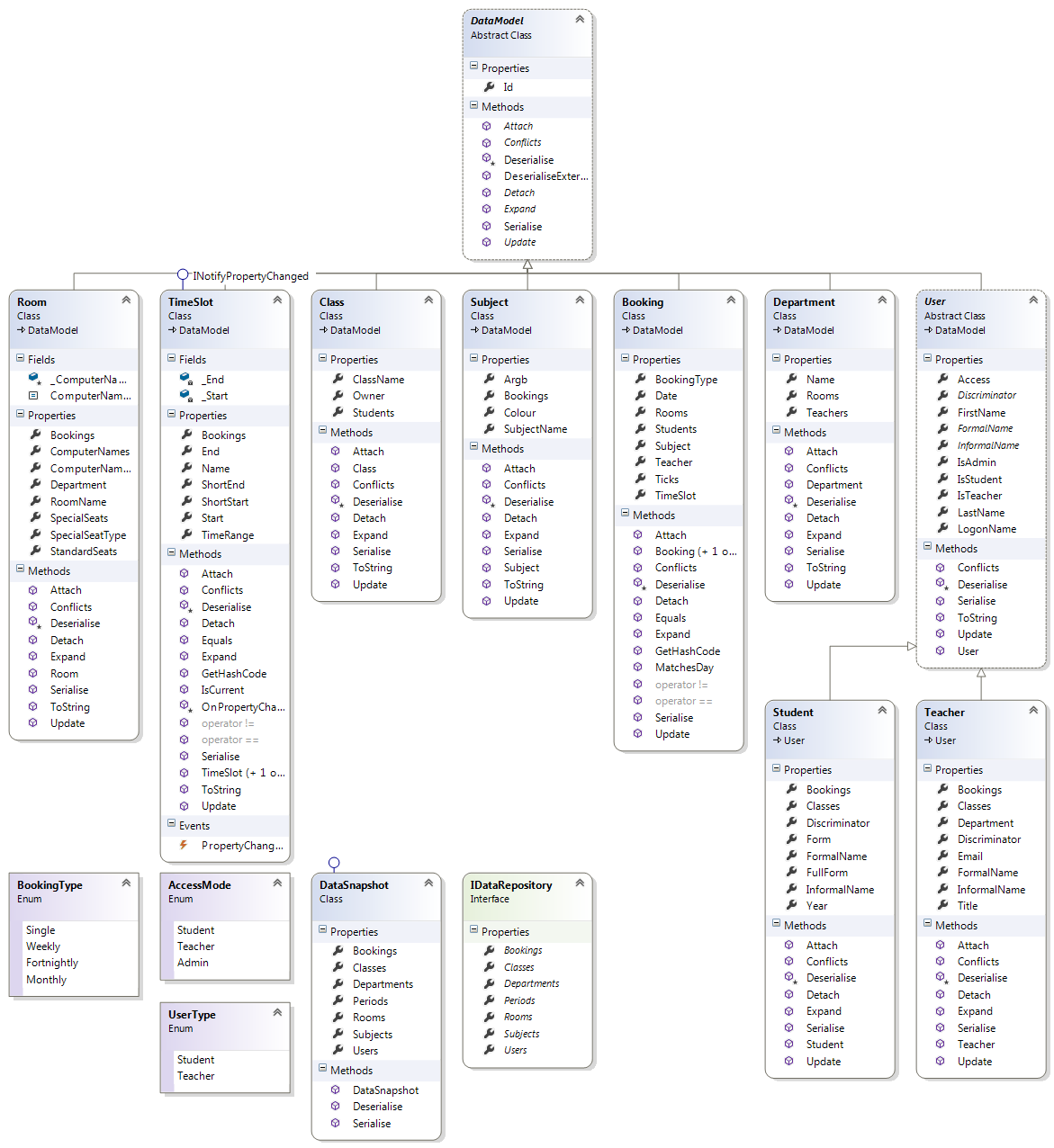
void Serialise(Writer Out);

void Deserialise(Reader In);

}

}

## Data



The Data assembly is one of the largest; it specifies the database objects that we want to work with using plain C# code, which Entity Framework then processes at compile-time to structure the database.

The most core class in this hierarchy is the *DataModel* class. It provides common subroutines for all the other database entities. The entities themselves constitute the bulk of the code for this module, describing the structure of the database, as well as the operations that can be performed on them.

A few other classes exist in this module – *DataSnapshot* provides a copy of all the information stored in the database – this is useful when an extended connection to the database isn’t required, but all the data needs to be retrieved.

*IDataRepository* is an interface describing common properties of classes that provide access to the items in a Database. The Server uses this class in its DataRepository class, which provides the true (Entity Framework) access to the database, while the Client also uses it for the class that provides access to its local copy of the data which is synced with the server.

Data.IDataRepository

The *IDataRepository* interface enforces that all its subclasses provide access to a *List* of certain objects – the objects found in the database. Note it only enforce the *get* property, not the *set*, as it isn’t required that external classes can reassign the data.

using System;

using System.Collections.Generic;

using Data.Models;

namespace Data

{

public interface IDataRepository

{

List<Booking> Bookings { get; }

List<Department> Departments { get; }

List<Room> Rooms { get; }

List<User> Users { get; }

List<Subject> Subjects { get; }

List<TimeSlot> Periods { get; }

List<Class> Classes { get; }

}

}

Data.DataSnapshot

The *DataSnapshot* class is the first large class relating to the database. It is used to store a copy of all the data in the database, for use in operations that may take a long time and so don’t want a continued connection to the database itself.

using System;

using System.Collections.Generic;

using Data.Models;

using Shared;

namespace Data

{

public class DataSnapshot

: ISerialisable, IDataRepository

{

public List<Booking> Bookings { get; set; }

public List<Department> Departments { get; set; }

public List<Room> Rooms { get; set; }

public List<User> Users { get; set; }

public List<Subject> Subjects { get; set; }

public List<TimeSlot> Periods { get; set; }

public List<Class> Classes { get; set; }

public DataSnapshot()

{

Bookings = new List<Booking>();

Departments = new List<Department>();

Rooms = new List<Room>();

Users = new List<User>();

Subjects = new List<Subject>();

Periods = new List<TimeSlot>();

Classes = new List<Class>();

}

A *DataSnapshot* can be serialised to a *Writer* for easy transfer across networks.  
The serialisation process is simple – for each type of *DataModel* in the database, write the number of that item, and then call the object-specific serialise method on each of the objects. LINQ is used for simplicity.

public void Serialise(Writer Out)

{

Out.Write(Bookings.Count);

Bookings.ForEach(b => b.Serialise(Out));

Out.Write(Departments.Count);

Departments.ForEach(d => d.Serialise(Out));

Out.Write(Periods.Count);

Periods.ForEach(p => p.Serialise(Out));

Out.Write(Rooms.Count);

Rooms.ForEach(r => r.Serialise(Out));

Out.Write(Users.Count);

Users.ForEach(t => t.Serialise(Out));

Out.Write(Subjects.Count);

Deserialisation is slightly more complicated – the number of each item is read, then *DeserialiseExternal* is called to read a loosely-typed object from the stream, before casting it to the desired type and storing it.

Subjects.ForEach(s => s.Serialise(Out));

Out.Write(Classes.Count);

Classes.ForEach(c => c.Serialise(Out));

}

public void Deserialise(Reader In)

{

Bookings = new List<Booking>(In.ReadInt32());

for (int x = 0; x < Bookings.Capacity; x++)

Bookings.Add((Booking)DataModel.DeserialiseExternal(In));

Departments = new List<Department>(In.ReadInt32());

for (int x = 0; x < Departments.Capacity; x++)

Departments.Add((Department)DataModel.DeserialiseExternal(In));

Periods = new List<TimeSlot>(In.ReadInt32());

for (int x = 0; x < Periods.Capacity; x++)

Periods.Add((TimeSlot)DataModel.DeserialiseExternal(In));

Rooms = new List<Room>(In.ReadInt32());

for (int x = 0; x < Rooms.Capacity; x++)

Rooms.Add((Room)DataModel.DeserialiseExternal(In));

Users = new List<User>(In.ReadInt32());

for (int x = 0; x < Users.Capacity; x++)

Users.Add((User)DataModel.DeserialiseExternal(In));

Subjects = new List<Subject>(In.ReadInt32());

for (int x = 0; x < Subjects.Capacity; x++)

Subjects.Add((Subject)DataModel.DeserialiseExternal(In));

Classes = new List<Class>(In.ReadInt32());

for (int x = 0; x < Classes.Capacity; x++)

Classes.Add((Class)DataModel.DeserialiseExternal(In));

}

}

}

Data.Models.DataModel

*DataModel* is the base class of all items stored in the database. It fulfils all the requirements for its subclasses to be stored in the database, allowing them to only specify the minimum. It provides some abstract function to be implemented by derived classes, some virtual functions that perform actions that need overriding by derived classes, as well as a utility function that can be used to deserialise any derived class of *DataModel*.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Reflection;

using System.ComponentModel.DataAnnotations;

using System.ComponentModel.DataAnnotations.Schema;

using Shared;

namespace Data.Models

{

public abstract class DataModel

{

[Key]

[DatabaseGenerated(DatabaseGeneratedOption.Identity)]

public int Id { get; set; }

// Returns true if the current object is already contained in the given list

public abstract bool Conflicts(List<DataModel> Others);

// Serialises the object to a Writer

public virtual void Serialise(Writer Out)

{

Out.Write(GetType().FullName);

Out.Write(Id);

}

// Deserialises the object from a Reader

protected virtual void Deserialise(Reader In)

{

Id = In.ReadInt32();

}

// Edits this object's properties to reflect the argument's

public abstract void Update(DataModel Other);

// Acquires references to all related entities in the database from a repository

public abstract bool Expand(IDataRepository Repo);

// Adds references to this item to all related entities in the database (flipside to Expand)

public abstract void Attach();

// Removes references to this item from related entities in the database (before deletion)

public abstract void Detach();

// Deserialise an unknown type of DataModel from a Reader

public static DataModel DeserialiseExternal(Reader In)

{

string TypeName = In.ReadString(); // Read the full type name of the sent object

// Test all subclasses of DataModel against the type name

foreach(Type t in Assembly.GetExecutingAssembly().GetTypes().Where(t => t.IsSubclassOf(typeof(DataModel))))

{

if(t.FullName == TypeName)

{

// Create the object, let it deserialise from the stream, then return it

DataModel m = (DataModel)Activator.CreateInstance(t);

m.Deserialise(In);

return m;

}

}

// If no matching type found, it's possible a client is using an out-of-date assembly

throw new Exception("Failed to find type name - suggests out of date Data.dll");

}

// Deserialise a compile-time-known type of DataModel

public static T DeserialiseExternal<T>(Reader In) where T : DataModel

{

try

{

// If T is abstract, we can’t use this method, so use the first one

if (typeof(T).IsAbstract)

return (T)DeserialiseExternal(In);

string TypeName = In.ReadString(); // Read the type name but ignore it

// Create the object, deserialise and return

T Result = Activator.CreateInstance<T>();

Result.Deserialise(In);

return Result;

}

catch

{

// If something went wrong, then deserialisation failed

throw new Exception("Received type isn't of type expected.");

}

*DeserialiseExternal* is used to read a type known at runtime from the input Reader – there are two overloads. The first returns an unknown type of object (ie, at compile time all that’s known is that it’s of a type that inherits *DataModel*, which can be unwieldy in some cases.  
The second returns a type specified by the Generic parameter *T*, which means at compile time the specific type is known, which offers more control.

}

}

}

Data.Models.User

*User* is the base class of *Student* and *Teacher*, and provides some standard base properties and methods used by both of these subclasses. It’s abstract, so can’t be instantiated on its own.  
*AccessMode* and *UserType* are enumerations used to identify the access to the system, and the table to store the classes in, respectively.

using System;

using System.Collections.Generic;

using System.Linq;

using System.ComponentModel.DataAnnotations.Schema;

using Shared;

namespace Data.Models

{

public enum AccessMode // Actual access mode in the system

{

Student,

Teacher,

Admin,

}

public enum UserType // Internal discriminator for the database between Students and Teachers

{

Student,

Teacher,

}

public abstract class User

: DataModel

{

public string FirstName { get; set; }

public string LastName { get; set; }

public string LogonName { get; set; } // User's school username

[NotMapped] // InformalName isn't stored in the DB, constructed from First and Last names

public abstract string InformalName { get; }

[NotMapped] // Again, not stored in the DB but constructed

public abstract string FormalName { get; }

// Access level in the system (used for all authentication checks)

public virtual AccessMode Access { get; set; }

// Used to differentiate between Students and Teachers while storing both in the same table.

public abstract UserType Discriminator { get; }

[NotMapped] // Helpful unmapped properties, good for UI bindings

public virtual bool IsStudent { get { return Access == AccessMode.Student; } }

[NotMapped]

public virtual bool IsTeacher { get { return Access == AccessMode.Teacher; } }

[NotMapped]

public virtual bool IsAdmin { get { return Access == AccessMode.Admin; } }

public User()

{

FirstName = string.Empty;

LastName = string.Empty;

}

public override bool Conflicts(List<DataModel> Others)

{

// Conflicts occur if the same ID and same LogonName are used elsewhere

return Others.Cast<User>().Any(u => u.Id != Id && u.LogonName == LogonName);

}

// Update this entry's data to match those provided

public override void Update(DataModel Other)

{

User u = (User)Other;

FirstName = u.FirstName;

*Update*, *Serialise*, *Deserialise*, and *Expand* can all be overridden from *DataModel*, which deals with these operations only for the properties provided by this class – deriving classes must provide their own implementations as well, and call these base methods as well in order to work properly.

LastName = u.LastName;

LogonName = u.LogonName;

Access = u.Access;

}

// Serialise all required properties

public override void Serialise(Writer Out)

{

base.Serialise(Out);

Out.Write((int)Access);

Out.Write(FirstName);

Out.Write(LastName);

Out.Write(LogonName);

}

// Deserialise all properties

protected override void Deserialise(Reader In)

{

base.Deserialise(In);

Access = (AccessMode)In.ReadInt32();

FirstName = In.ReadString();

LastName = In.ReadString();

LogonName = In.ReadString();

}

public override string ToString()

{

return InformalName;

}

}

}

Data.Models.Student

A *Student* represents a school pupil – they have lessons to attend and classes they belong to.  
*Student* is a subclass of *User*, and so override some functions/properties of the base class.

using System;

using System.Collections.Generic;

using System.Linq;

using System.ComponentModel.DataAnnotations.Schema;

using Shared;

namespace Data.Models

{

// Contains all the students and relevant info

[Table("Students")]

public class Student

: User

{

public int Year { get; set; } // Eg. 13

public string Form { get; set; } // Eg. WT

[NotMapped] // Utility property to get the full form name

public string FullForm { get { return Year + Form; } }

public override string InformalName { get { return FirstName + " " + LastName; } }

public override string FormalName { get { return InformalName; } }

public override UserType Discriminator { get { return UserType.Student; } }

// All bookings the user's involved in

public virtual List<Booking> Bookings { get; set; }

// All classes the user's involved in

public virtual List<Class> Classes { get; set; }

public Student()

{

Access = AccessMode.Student;

Form = string.Empty;

Bookings = new List<Booking>();

Classes = new List<Class>();

}

public override bool Conflicts(List<DataModel> Others)

{

return base.Conflicts(Others);

}

public override void Update(DataModel Other)

{

base.Update(Other);

Student s = (Student)Other;

Year = s.Year;

Form = s.Form;

Classes.Clear();

Classes.AddRange(s.Classes);

Bookings.Clear();

Bookings.AddRange(s.Bookings);

}

// Serialise all properties to the stream

public override void Serialise(Writer Out)

{

Calling the base class’s functions as part of the override function lets this class make use of existing code written into the *User* class, such as conflict checking and property serialisation.

// Serialise the base class' properties

base.Serialise(Out);

Out.Write(Year);

Out.Write(Form);

Out.Write(Classes.Count);

Classes.ForEach(c => Out.Write(c.Id));

Out.Write(Bookings.Count);

Bookings.ForEach(b => Out.Write(b.Id));

}

// Deserialise from the stream

protected override void Deserialise(Reader In)

{

base.Deserialise(In);

Year = In.ReadInt32();

Form = In.ReadString();

Classes = Enumerable.Repeat(new Class(), In.ReadInt32()).ToList();

Classes.ForEach(c => c.Id = In.ReadInt32());

Bookings = Enumerable.Repeat(new Booking(), In.ReadInt32()).ToList();

Bookings.ForEach(b => b.Id = In.ReadInt32());

}

// Obtain references to related entities

public override bool Expand(IDataRepository Repo)

{

try

{

for (int x = 0; x < Classes.Count; x++)

Classes[x] = Repo.Classes.SingleOrDefault(c => c.Id == Classes[x].Id);

for (int x = 0; x < Bookings.Count; x++)

Bookings[x] = Repo.Bookings.SingleOrDefault(b => b.Id == Bookings[x].Id);

}

catch

{

return false;

}

return true;

}

// Set references to this from other related objects

public override void Attach()

{

Bookings.ForEach(b => b.Students.Add(this));

Classes.ForEach(c => c.Students.Add(this));

}

// Remove references before deletion

public override void Detach()

{

Bookings.ForEach(b => { if (b != null) b.Students.RemoveAll(i => i.Id == Id); });

Classes.ForEach(c => { if (c != null) c.Students.RemoveAll(i => i.Id == Id); });

}

}

}

Data.Models.Teacher

A *Teacher* is a subclass of *User* that has more access and can create/edit bookings etc. An Admin user is just a *Teacher* with *AccessRights* set to *AccessRights.Admin*.   
This class is very similar to *Student*, it just has slightly different properties.

using System;

using System.Collections.Generic;

using System.Linq;

using System.ComponentModel.DataAnnotations.Schema;

using Shared;

namespace Data.Models

{

// Contains all teachers and properties

[Table("Teachers")]

public class Teacher

: User

{

public string Title { get; set; } // Eg. Mr, Mrs

public string Email { get; set; } // Eg. email@gmail.com

public override string InformalName { get { return FirstName + " " + LastName; } }

public override string FormalName { get { return Title + " " + LastName; } }

public override UserType Discriminator { get { return UserType.Teacher; } }

public virtual Department Department { get; set; }

// All bookings the user's involved in

public virtual List<Booking> Bookings { get; set; }

// All classes the user's involved in

public virtual List<Class> Classes { get; set; }

public Teacher()

{

Access = AccessMode.Teacher;

Title = string.Empty;

Email = string.Empty;

Bookings = new List<Booking>();

Classes = new List<Class>();

}

public override bool Conflicts(List<DataModel> Others)

{

// Return true if the base conflicts or the properties introduced in this class conflict

return base.Conflicts(Others) || Others.OfType<Teacher>().Any(t => t.Id != Id && t.Title == Title && t.Email == Email);

Conflict if another *Teacher* has matching Title and Email properties.

}

public override void Update(DataModel Other)

{

base.Update(Other);

Teacher t = (Teacher)Other;

Title = t.Title;

Email = t.Email;

Department = t.Department;

Classes.Clear();

Classes.AddRange(t.Classes);

Bookings.Clear();

Bookings.AddRange(t.Bookings);

}

public override void Serialise(Writer Out)

{

// Output base class properties followed by this class's properties

base.Serialise(Out);

Out.Write(Title);

Out.Write(Department.Id);

Out.Write(Classes.Count);

Classes.ForEach(c => Out.Write(c.Id));

Out.Write(Email);

Out.Write(Bookings.Count);

Bookings.ForEach(b => Out.Write(b.Id));

}

protected override void Deserialise(Reader In)

{

// Deserialise base class then this class

base.Deserialise(In);

Title = In.ReadString();

Department = new Department() { Id = In.ReadInt32() };

Classes = Enumerable.Repeat(new Class(), In.ReadInt32()).ToList();

Classes.ForEach(c => c.Id = In.ReadInt32());

Email = In.ReadString();

Bookings = Enumerable.Repeat(new Booking(), In.ReadInt32()).ToList();

Bookings.ForEach(b => b.Id = In.ReadInt32());

}

// Obtain IDs of related objects

public override bool Expand(IDataRepository Repo)

{

try

{

Department = Repo.Departments.SingleOrDefault(d => d.Id == Department.Id);

for (int x = 0; x < Classes.Count; x++)

Classes[x] = Repo.Classes.SingleOrDefault(c => c.Id == Classes[x].Id);

for (int x = 0; x < Bookings.Count; x++)

Bookings[x] = Repo.Bookings.SingleOrDefault(b => b.Id == Bookings[x].Id);

}

catch

{

return false;

}

return true;

}

// Branch out references

public override void Attach()

{

Bookings.ForEach(b => b.Teacher = this);

if (Department != null)

Department.Teachers.Add(this);

Classes.ForEach(c => c.Owner = this);

}

// Remove references

public override void Detach()

{

Bookings.ForEach(b => { if (b != null) b.Teacher = null; });

if (Department != null)

Department.Teachers.RemoveAll(i => i.Id == Id);

Classes.ForEach(c => { if (c != null) c.Owner = null; });

}

}

}

Data.Models.Department

A *Department* is a collection of *Room*s and *Teacher*s.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Data;

using System.ComponentModel.DataAnnotations.Schema;

using Shared;

namespace Data.Models

{

[Table("Departments")]

public class Department

: DataModel

{

// Name of the department (eg Computing, Maths)

public string Name { get; set; }

// Teachers in the department

public virtual List<Teacher> Teachers { get; set; }

// Rooms owned by the department

public virtual List<Room> Rooms { get; set; }

public Department()

{

Name = string.Empty;

Teachers = new List<Teacher>();

Rooms = new List<Room>();

}

public override bool Conflicts(List<DataModel> Others)

{

// Conflicts occur on matching names and ID

return Others.Cast<Department>().Any(d => d.Id != Id && d.Name == Name);

}

Conflict if another *Department* has matching Name.

public override void Update(DataModel Other)

{

Department d = (Department)Other;

Name = d.Name;

Teachers.Clear();

Teachers.AddRange(d.Teachers);

Rooms.Clear();

Rooms.AddRange(d.Rooms);

}

// Serialise properties and IDs

Standard serialisation/deserialisation method.

public override void Serialise(Writer Out)

{

base.Serialise(Out);

Out.Write(Name);

Out.Write(Teachers.Count);

Teachers.ForEach(t => Out.Write(t.Id));

Out.Write(Rooms.Count);

Rooms.ForEach(r => Out.Write(r.Id));

}

// Deserialise properties and IDs

protected override void Deserialise(Reader In)

{

base.Deserialise(In);

Name = In.ReadString();

Teachers = Enumerable.Repeat(new Teacher(), In.ReadInt32()).ToList();

Teachers.ForEach(t => t.Id = In.ReadInt32());

Rooms = Enumerable.Repeat(new Room(), In.ReadInt32()).ToList();

Rooms.ForEach(r => r.Id = In.ReadInt32());

}

// Obtain references to related items

public override bool Expand(IDataRepository Repo)

{

try

{

for (int x = 0; x < Teachers.Count; x++)

Teachers[x] = Repo.Users.OfType<Teacher>().SingleOrDefault(t => t.Id == Teachers[x].Id);

for (int x = 0; x < Rooms.Count; x++)

Rooms[x] = Repo.Rooms.SingleOrDefault(r => r.Id == Rooms[x].Id);

}

catch

{

return false;

}

return true;

}

// Set references to this item

public override void Attach()

{

Teachers.ForEach(t => t.Department = this);

Rooms.ForEach(r => r.Department = this);

}

// Remove references to this item

public override void Detach()

{

Teachers.ForEach(t => { if (t != null) t.Department = null; });

Rooms.ForEach(r => { if (r != null) r.Department = null; });

}

public override string ToString()

{

return Name;

}

}

}

Data.Models.Room

A *Room* is an area that can be booked – the most standard type would be a computer room, but for example workshops or sports areas can be included as well. The *SpecialSeats* properties allow customisation of the apparent “type” of room, from computer to workbench etc.

using System;

using System.Collections.Generic;

using System.Linq;

using System.ComponentModel.DataAnnotations.Schema;

using Shared;

namespace Data.Models

{

// Rooms are the area that can be booked

[Table("Rooms")]

public class Room

: DataModel

{

// Recognisable name of the room (eg D6, D12, Library)

public string RoomName { get; set; }

// Number of "standard seats". Usually just number of available desks

public int StandardSeats { get; set; }

// Number of "special seats", for example a computer, workbench etc

public int SpecialSeats { get; set; }

// Type of "Special Seat", so eg "Computer", "Workbench"

public string SpecialSeatType { get; set; }

// Bookings using this room

public virtual List<Booking> Bookings { get; set; }

public virtual Department Department { get; set; }

protected List<string> \_ComputerNames = new List<string>();

[NotMapped] // The list of names of computers in a room (eg D12C2)

public List<string> ComputerNames

{

get { return \_ComputerNames; }

set

{

// Can't have a room containing the character used to delimit computer names

if (value.Any(s => s.Contains(ComputerNameSeperator)))

throw new ArgumentException("Computer name cannot contain '" + ComputerNameSeperator + "'.");

\_ComputerNames = value;

}

}

// Can't store a List<string> in the DB, so store a string formed by joining all

// the names, delimiting with a seperator. Gettings/Setting this property creates

// the string by working on the list. This is the actual property stored in the DB

public string ComputerNamesJoined

{

get { return string.Join("" + ComputerNameSeperator, ComputerNames); }

set { ComputerNames = value.Split(ComputerNameSeperator).ToList(); }

}

// Character used to delimit computer names in the joined string

public const char ComputerNameSeperator = '|';

The database can’t store *List<string>* objects, so an unmapped property *ComputerNames* and its backing field *\_ComputerNames* are used to store the names in a useful *List* object, while a mapped property *ComputerNamesJoined* is the one actually written to the DB. The mapped property is constructed when needed by joining/splitting the list.

public Room()

{

Bookings = new List<Booking>();

ComputerNames = new List<string>();

RoomName = string.Empty;

SpecialSeatType = string.Empty;

}

public override bool Conflicts(List<DataModel> Others)

{

return Others.Cast<Room>().Any(r => r.Id != Id && r.RoomName == RoomName);

}

Conflict on duplicate Name and ID.

public override void Update(DataModel Other)

{

Room r = (Room)Other;

RoomName = r.RoomName;

StandardSeats = r.StandardSeats;

SpecialSeatType = r.SpecialSeatType;

SpecialSeats = r.SpecialSeats;

Bookings.Clear();

Bookings.AddRange(r.Bookings);

Department = r.Department;

}

// Serialise properties and IDs

public override void Serialise(Writer Out)

{

base.Serialise(Out);

Out.Write(RoomName);

Out.Write(StandardSeats);

Out.Write(SpecialSeats);

Out.Write(SpecialSeatType);

Out.Write(ComputerNamesJoined);

Out.Write(Bookings.Count);

Bookings.ForEach(b => Out.Write(b.Id));

Out.Write(Department.Id);

}

// Deserialise properties and IDs

protected override void Deserialise(Reader In)

{

base.Deserialise(In);

RoomName = In.ReadString();

StandardSeats = In.ReadInt32();

SpecialSeats = In.ReadInt32();

SpecialSeatType = In.ReadString();

ComputerNamesJoined = In.ReadString();

Bookings = Enumerable.Repeat(new Booking(), In.ReadInt32()).ToList();

Bookings.ForEach(b => b.Id = In.ReadInt32());

Department = new Department() { Id = In.ReadInt32() };

}

// Obtain references to related items

public override bool Expand(IDataRepository Repo)

{

try

{

for (int x = 0; x < Bookings.Count; x++)

Bookings[x] = Repo.Bookings.SingleOrDefault(b => b.Id == Bookings[x].Id);

Department = Repo.Departments.SingleOrDefault(d => d.Id == Department.Id);

}

catch

{

return false;

}

return true;

}

// Set references to this item

public override void Attach()

{

Bookings.ForEach(b => b.Rooms.Add(this));

if (Department != null)

Department.Rooms.Add(this);

}

// Remove references to this item

public override void Detach()

{

Bookings.ForEach(b => { if (b != null) b.Rooms.RemoveAll(i => i.Id == Id); });

if (Department != null)

Department.Rooms.RemoveAll(i => i.Id == Id);

}

public override string ToString()

{

return RoomName;

}

}

}

Data.Models.Class

A *Class* is effectively a group of students identified by a name. They’re used to easily select students without having to search through all students.

using System;

using System.Collections.Generic;

using System.Linq;

using System.ComponentModel.DataAnnotations.Schema;

using Shared;

namespace Data.Models

{

// Used to easily select multiple students from a pre-set list

// Links multiple students to a class name, with an owning teacher

[Table("Classes")]

public class Class

: DataModel

{

// Name of the class, used for selections

public string ClassName { get; set; }

// Teacher that's responsible for the class

public virtual Teacher Owner { get; set; }

// Students included in the class

public virtual List<Student> Students { get; set; }

public Class()

{

ClassName = string.Empty;

Students = new List<Student>();

}

public override bool Conflicts(List<DataModel> Others)

{

return Others.Cast<Class>().Any(c => c.Id != Id && c.ClassName == ClassName);

}

Conflict on matching class name.

public override void Update(DataModel Other)

{

Class c = (Class)Other;

ClassName = c.ClassName;

Owner = c.Owner;

Students.Clear();

Students.AddRange(c.Students);

}

// Output properties and IDs

public override void Serialise(Writer Out)

{

base.Serialise(Out);

Out.Write(ClassName);

Out.Write(Owner.Id);

Out.Write(Students.Count);

Students.ForEach(s => Out.Write(s.Id));

}

// Input properties and IDs

protected override void Deserialise(Reader In)

{

base.Deserialise(In);

ClassName = In.ReadString();

Owner = new Teacher() { Id = In.ReadInt32() };

Students = new List<Student>(In.ReadInt32());

for (int x = 0; x < Students.Capacity; x++)

Students.Add(new Student() { Id = In.ReadInt32() });

}

// Acquire references

public override bool Expand(IDataRepository Repo)

{

try

{

Owner = (Teacher)Repo.Users.SingleOrDefault(t => t.Id == Owner.Id);

for (int x = 0; x < Students.Count; x++)

Students[x] = Repo.Users.OfType<Student>().SingleOrDefault(s => s.Id == Students[x].Id);

}

catch

{

return false;

}

return true;

}

// Set references to this

public override void Attach()

{

if (Owner != null)

Owner.Classes.Add(this);

Students.ForEach(s => s.Classes.Add(this));

}

// Remove references to this

public override void Detach()

{

if (Owner != null)

Owner.Classes.RemoveAll(i => i.Id == Id);

Students.ForEach(s => { if (s != null) s.Classes.RemoveAll(i => i.Id == Id); });

}

public override string ToString()

{

return ClassName;

}

}

}

Data.Models.Subject

A *Subject* is used by *Booking* objects to denote what lesson Is being taught (maths, PE etc). Objects contain the name of the subject, as well as the colour used to represent the subject on the main Timetable window.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Data;

using System.ComponentModel.DataAnnotations.Schema;

using System.Windows.Media;

using Shared;

namespace Data.Models

{

// Contains all subjects (eg Maths, Computing)

[Table("Subjects")]

public class Subject

The database can’t store CLR *Color* objects, but it can store primitive *int* types, and a *Color* can be converted to an *int*. We store an unmapped property which holds the CLR *Color* object for use in code, and a property of type *int* that is actually stored in the database and just provides a wrapper around the true *Color* object.

: DataModel

{

// Friendly name of the subject (Maths, Computing)

public string SubjectName { get; set; }

// Store the integer equivalent of the colour

public int Argb

{

get { return Helpers.ColorToInt(Colour); }

set { Colour = Helpers.IntToColour(value); }

}

// Colour used to display bookings of this subject on the timetable

[NotMapped]

public Color Colour { get; set; }

// Bookings of this subject

public virtual List<Booking> Bookings { get; set; }

public Subject()

{

Bookings = new List<Booking>();

SubjectName = string.Empty;

}

public override bool Conflicts(List<DataModel> Others)

{

return Others.Cast<Subject>().Any(s => s.Id != Id && s.SubjectName == SubjectName);

}

public override void Update(DataModel Other)

{

Subject s = (Subject)Other;

SubjectName = s.SubjectName;

Argb = s.Argb;

Bookings.Clear();

Bookings.AddRange(s.Bookings);

}

// Write properties and IDs

public override void Serialise(Writer Out)

{

base.Serialise(Out);

Out.Write(SubjectName);

Out.Write(Argb);

Out.Write(Bookings.Count);

Bookings.ForEach(b => Out.Write(b.Id));

}

// Read properties and IDs

protected override void Deserialise(Reader In)

{

base.Deserialise(In);

SubjectName = In.ReadString();

Argb = In.ReadInt32();

Bookings = Enumerable.Repeat(new Booking(), In.ReadInt32()).ToList();

Bookings.ForEach(b => b.Id = In.ReadInt32());

}

// Obtain references to related entities

public override bool Expand(IDataRepository Repo)

{

try

{

for (int x = 0; x < Bookings.Count; x++)

Bookings[x] = Repo.Bookings.SingleOrDefault(b => b.Id == Bookings[x].Id);

}

catch

{

return false;

}

return true;

}

// Set references to this object

public override void Attach()

{

Bookings.ForEach(b => b.Subject = this);

}

// Remove references to this object

public override void Detach()

{

Bookings.ForEach(b => { if (b != null) b.Subject = null; });

}

public override string ToString()

{

return SubjectName;

}

}

}

Data.Models.TimeSlot

A *TimeSlot* is used to represent a school period – a start and end time. As this class is used quite frequently in the UI, it implements *INotifyPropertyChanged*, providing a neat way for the UI to respond to changes of properties on this object. It also provides overloads of the equality operators for the same reasons.

using System;

using System.Collections.Generic;

using System.Linq;

using System.ComponentModel;

using System.ComponentModel.DataAnnotations.Schema;

using Shared;

namespace Data.Models

{

// A timeslot is a school period - a start and end time of a lesson

[Table("Periods")]

public class TimeSlot

: DataModel, INotifyPropertyChanged

{

private TimeSpan \_Start;

// The Start time of the period

public TimeSpan Start

{

get { return \_Start; }

set { \_Start = value; OnPropertyChanged("Start"); }

}

[NotMapped] // Start time in short time format (12:10), used for UI

public string ShortStart { get { return new DateTime(Start.Ticks).ToShortTimeString(); } }

private TimeSpan \_End;

// The End time of the period

public TimeSpan End

{

get { return \_End; }

set { \_End = value; OnPropertyChanged("End"); }

}

[NotMapped] // End time in short time format(13:10), used in UI

public string ShortEnd { get { return new DateTime(End.Ticks).ToShortTimeString(); } }

// Name of the period

public string Name { get; set; }

[NotMapped] // Utility property for UI - range of times

public string TimeRange

{

get { return ShortStart + " - " + ShortEnd; }

}

// Bookings using this period

public virtual List<Booking> Bookings { get; set; }

public TimeSlot()

: this(new TimeSpan(0, 0, 0), new TimeSpan(0, 0, 0))

{

}

public TimeSlot(TimeSpan Start, TimeSpan End)

{

PropertyChanged = delegate { };

this.Start = Start;

this.End = End;

Bookings = new List<Booking>();

Name = string.Empty;

}

// Returns if the provided Time's time is in this period

public bool IsCurrent(DateTime Time)

{

TimeSpan Mod = Time - Time.Date;

return Start <= Mod && End >= Mod;

}

public override bool Conflicts(List<DataModel> Others)

{

return Others.Cast<TimeSlot>().Any(t => t.Id != Id && t.Name == Name || (t.Start == Start && t.End == End));

Conflict on matching Name, Start and End times.

}

public override void Update(DataModel Other)

{

TimeSlot t = (TimeSlot)Other;

Start = t.Start;

End = t.End;

Name = t.Name;

Bookings.Clear();

Bookings.AddRange(t.Bookings);

}

// Serialise properties and IDs

public override void Serialise(Writer Out)

{

base.Serialise(Out);

Out.Write(\_Start.Ticks);

Out.Write(\_End.Ticks);

Out.Write(Name);

Out.Write(Bookings.Count);

Bookings.ForEach(b => Out.Write(b.Id));

}

// Deserialise properties and IDs

protected override void Deserialise(Reader In)

{

base.Deserialise(In);

\_Start = new TimeSpan(In.ReadInt64());

\_End = new TimeSpan(In.ReadInt64());

Name = In.ReadString();

Bookings = Enumerable.Repeat(new Booking(), In.ReadInt32()).ToList();

Bookings.ForEach(b => b.Id = In.ReadInt32());

}

// Obtain references to other items

public override bool Expand(IDataRepository Repo)

{

try

{

for (int x = 0; x < Bookings.Count; x++)

Bookings[x] = Repo.Bookings.SingleOrDefault(b => b.Id == Bookings[x].Id);

}

catch

{

return false;

}

return true;

}

// Add references to this to the related items

public override void Attach()

{

Bookings.ForEach(b => b.TimeSlot = this);

}

// Remove references to this from the related items

public override void Detach()

{

Bookings.ForEach(b => { if (b != null) b.TimeSlot = null; });

}

public override string ToString()

{

if (!string.IsNullOrWhiteSpace(Name))

return Name;

else

return TimeRange;

}

public static bool operator ==(TimeSlot One, TimeSlot Two)

{

// If both object references are actually the same object, return true

if (ReferenceEquals(One, Two))

return true;

// Equal if the name, start and end times all match for two non-null objects

return (object)One != null && (object)Two != null && One.Start == Two.Start && One.End == Two.End && One.Name == Two.Name;

}

public static bool operator !=(TimeSlot One, TimeSlot Two)

{

// Required overload of !=, just invert the already overriden == operator

return !(One == Two);

}

public override bool Equals(object obj)

{

// Required overload, check for null then do a standard equality check

TimeSlot Obj = obj as TimeSlot;

The *Equals* and *GetHashCode* functions must be overridden after overriding the == operator. The *Equals* function simply delegates the check to the == operator, while *GetHashCode* just calls the base function.

if (Obj == null)

return false;

return this == Obj;

}

public override int GetHashCode()

{

// Required overload, just perform the base function

return base.GetHashCode();

}

// Utility function to fire the PropertyChanged event using less code

protected void OnPropertyChanged(string PropertyName)

{

if (PropertyChanged != null)

PropertyChanged(this, new PropertyChangedEventArgs(PropertyName));

}

// Event to be fired on a property changing - used for UI responsiveness

public event PropertyChangedEventHandler PropertyChanged;

}

The *INotifyPropertyChanged* interface is really useful in UI systems – when the *PropertyChanged* event is fired, the UI can automatically update itself to reflect the change. The *OnPropertyChanged* function just provides an easier-to-use wrapper around the event.

}

Data.Models.Booking

A *Booking* object ties together the majority of other data items in this module – it associates a *Teacher* teaching a particular S*ubject* with a list of *Student*s during a specified *Timeslot* in a set of *Room*s.  
Although it initially appears more complex than the other data items, in reality it just has a greater quantity of items rather than a greater complexity.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Data;

using System.ComponentModel.DataAnnotations.Schema;

using Shared;

namespace Data.Models

{

// Recurrence type of the booking

public enum BookingType

{

// A one off booking

Single,

*BookingType* is a handy enumeration of values used to identify how many days there are between recurring bookings.

// Occurs every 7 days

Weekly,

// Occurs every 14 days

Fortnightly,

// Occurs every 4 weeks

Monthly,

}

// A Booking is a effectively a single lesson.

[Table("Bookings")]

public class Booking

: DataModel

{

// The Date of the booking (the time part is irrelevant)

[NotMapped]

public DateTime Date

{

get { return DateTime.FromBinary(Ticks).Date; }

set { Ticks = value.Date.ToBinary(); }

}

// Representation of the Date as a primitive datatype for storage in the database

public long Ticks { get; set; }

public BookingType BookingType { get; set; }

// The duration of the booking (used to work out which period for display purposes)

public virtual TimeSlot TimeSlot { get; set; }

// Rooms used by this booking

public virtual List<Room> Rooms { get; set; }

// The subject of this booking

public virtual Subject Subject { get; set; }

// Students attending this booking

public virtual List<Student> Students { get; set; }

// Teacher who made the booking

public virtual Teacher Teacher { get; set; }

public Booking()

: this(null, new List<Room>(), null, new List<Student>(), null, BookingType.Single)

{

}

public Booking(TimeSlot Time, List<Room> Rooms, Subject Subject, List<Student> Students, Teacher Teacher, BookingType BookingType)

{

TimeSlot = Time;

this.Rooms = Rooms;

this.Subject = Subject;

this.Students = Students;

this.Teacher = Teacher;

this.BookingType = BookingType;

}

// Determines if this booking will happen on a given day

public bool MatchesDay(DateTime Day)

{

return (BookingType == BookingType.Single && Day.Date == Date) ||

(BookingType == BookingType.Weekly && (Day.Date - Date).Days % 7 == 0) ||

(BookingType == BookingType.Fortnightly && (Day.Date - Date).Days % 14 == 0) ||

(BookingType == BookingType.Monthly && (Day.Date - Date).Days % 28 == 0);

}

public override bool Conflicts(List<DataModel> AllBookings)

{

// Checks for conflicts between booking objects

return AllBookings.Cast<Booking>().Any(b => b.Id != Id && b.Date == Date && b.TimeSlot == TimeSlot && b.Rooms.Intersect(Rooms).Count() != 0);

}

To check if this *Booking* will occur on a given date we compare the type of booking we’re dealing with and the number of days between the given date and the original date of the booking.

public override void Update(DataModel Other)

{

// Copy data from other item into this

Booking b = (Booking)Other;

Ticks = b.Ticks;

BookingType = b.BookingType;

TimeSlot = b.TimeSlot;

Rooms = b.Rooms;

Subject = b.Subject;

Students.Clear();

Students.AddRange(b.Students);

Teacher = b.Teacher;

}

// Serialise to an output stream

public override void Serialise(Writer Out)

{

base.Serialise(Out);

Out.Write(Ticks);

Out.Write((int)BookingType);

Out.Write(TimeSlot.Id);

Out.Write(Rooms.Count);

foreach (Room r in Rooms)

Out.Write(r.Id);

Out.Write(Subject.Id);

Out.Write(Students.Count);

foreach (Student s in Students)

Out.Write(s.Id);

Out.Write(Teacher.Id);

}

// Deserialise from an input stream

protected override void Deserialise(Reader In)

{

base.Deserialise(In);

Ticks = In.ReadInt64();

BookingType = (BookingType)In.ReadInt32();

TimeSlot = new TimeSlot() { Id = In.ReadInt32() };

Rooms = new List<Room>(In.ReadInt32());

for (int x = 0; x < Rooms.Capacity; x++)

Rooms.Add(new Room() { Id = In.ReadInt32() });

Subject = new Subject() { Id = In.ReadInt32() };

Students = new List<Student>(In.ReadInt32());

for (int x = 0; x < Students.Capacity; x++)

Students.Add(new Student() { Id = In.ReadInt32() });

Teacher = new Teacher() { Id = In.ReadInt32() };

}

// Obtain references to related objects

public override bool Expand(IDataRepository Repo)

{

try

{

TimeSlot = Repo.Periods.SingleOrDefault(t => t.Id == TimeSlot.Id);

for (int x = 0; x < Rooms.Count; x++)

Rooms[x] = Repo.Rooms.SingleOrDefault(r => Rooms[x].Id == r.Id);

Subject = Repo.Subjects.SingleOrDefault(s => s.Id == Subject.Id);

for (int x = 0; x < Students.Count; x++)

Students[x] = (Student)Repo.Users.SingleOrDefault(s => Students[x].Id == s.Id);

Teacher = (Teacher)Repo.Users.SingleOrDefault(t => t.Id == Teacher.Id);

}

catch

{

return false;

}

return true;

}

// Set references to this object

public override void Attach()

{

if (TimeSlot != null)

TimeSlot.Bookings.Add(this);

Rooms.ForEach(r => r.Bookings.Add(this));

if (Subject != null)

Subject.Bookings.Add(this);

Students.ForEach(s => s.Bookings.Add(this));

if (Teacher != null)

Teacher.Bookings.Add(this);

}

// Remove references to this object

public override void Detach()

{

if (TimeSlot != null)

TimeSlot.Bookings.RemoveAll(i => i.Id == Id);

Rooms.ForEach(r => { if (r != null) r.Bookings.RemoveAll(b => b.Id == Id); });

if (Subject != null)

Subject.Bookings.RemoveAll(i => i.Id == Id);

Students.ForEach(s => { if (s != null) s.Bookings.RemoveAll(b => b.Id == Id); });

if (Teacher != null)

Teacher.Bookings.RemoveAll(i => i.Id == Id);

}

// Equality operators

public override bool Equals(object obj)

{

if (!(obj is Booking)) // Type check

return false;

return this == (obj as Booking);

}

public static bool operator ==(Booking One, Booking Two)

{

// Check for references

if (ReferenceEquals(One, null))

Equality comparisons are surprisingly complicated – we need to check for references first, but after that it’s simply a matter of comparing fields.

{

if (ReferenceEquals(Two, null))

return true;

return false;

}

if (ReferenceEquals(Two, null))

return false;

// Check for references then compare fields

return ReferenceEquals(One, Two) || (One.BookingType == Two.BookingType && One.Date == Two.Date && One.Id == Two.Id &&

One.Rooms == Two.Rooms && One.Students == Two.Students && One.Subject == Two.Subject && One.Teacher == Two.Teacher && One.Ticks == Two.Ticks &&

One.TimeSlot == Two.TimeSlot);

}

public static bool operator !=(Booking One, Booking Two)

{

return !(One == Two);

}

public override int GetHashCode()

{

return base.GetHashCode();

}

}

}}

## NetCore



NetCore is possibly the most complicated assembly of the solution, despite being roughly half the amount of lines of code as the Data assembly. The purpose of this library is to provide the framework for network communications to the assemblies referencing it – this assembly references both Shared and Data.

The main bulk of code is in describing the various Messages that can be transmitted around the system. These inherit from a single abstract base class *Message*. Much of the Client/Server model networking code is written here as well, in the *Listener* and *Client* classes (Server), and *Connection* class (Client).  
Numerous delegate function signatures are also defined throughout this assembly to describe the signature of event handlers for events exposed by the various networking classes.

NetCore.BlockingQueue

This class is really just for convenience – a *BlockingQueue<T>* is just a *BlockingCollection<T>* with an underlying container of a *ConcurrentQueue<T>*. Creating a new class for this composition makes code later on easier to read.  
The purpose of this class is to provide a queue that is thread-safe while also temporarily blocking the thread if an item is requested when one isn’t available, returning control when a new item is available. Using this class makes multithreaded code in the Listener class far easier to implement.

using System;

using System.Collections.Generic;

using System.Collections.Concurrent;

namespace NetCore

{

public class BlockingQueue<T>

: BlockingCollection<T>

{

public BlockingQueue()

: base(new ConcurrentQueue<T>())

{

}

public BlockingQueue(IEnumerable<T> Collection)

: base(new ConcurrentQueue<T>(Collection))

{

}

}

}

NetCore.Messages.Message

*Message* is the superclass of all the types of message that can be sent – it provides the base functionality that can then be extended/altered by its subclasses. It also provides static methods for reading Messages from a stream, even without knowing the type of the message in advance. This allows the Server and Client to deal with messages in any order. It inherits *ISerialisable* from the Shared library, and subclasses provide specific implementations.

using System;

using System.Linq;

using System.Reflection;

using Shared;

namespace NetCore.Messages

{

// Superclass for all types of message

public abstract class Message

: ISerialisable

{

private static Type[] MessageTypes { get; set; }

public virtual void Serialise(Writer Writer)

{

// Send a single byte as a notification

Writer.Write((byte)0);

Writer.Write(GetType().Name);

}

public abstract void Deserialise(Reader Reader);

public static void RegenMessageTypes() // Only call if a new assembly has been loaded

{

// Load all the types that inherit from this class in the executing assembly

MessageTypes = Assembly.GetExecutingAssembly().GetTypes().Where(t => t.IsSubclassOf(typeof(Message))).ToArray();

}

public static Message ReadMessage(NetReader Reader)

{

try

{

if (MessageTypes == null)

RegenMessageTypes(); // Load message classes if necessary

string Id = Reader.ReadString(); // Read the type

foreach (Type t in MessageTypes)

{

if (t.Name == Id) // Found the right class

{

// Create an object of the class, deserialise to it and return

Message m = (Message)Activator.CreateInstance(t);

m.Deserialise(Reader);

return m;

}

}

throw new Exception("Invalid Type received"); // No matching class found

}

catch (MissingMethodException e)

{

// MissingMethodException is thrown if a subclass of Message can't be constructed

throw new Exception("All Message subclasses must define a public parameterless constructor.", e);

}

}

// Read a generic Message

public static T ReadMessage<T>(NetReader Reader) where T : Message

{

try

{

// Read the Type but ignore it

Reader.ReadString();

// Create and deserialise

T Msg = Activator.CreateInstance<T>();

Msg.Deserialise(Reader);

return Msg;

}

catch

{

throw new Exception("Invalid message received.");

}

}

}

}

NetCore.Messages.TestMessage

*TestMessage* is a simple class that was used to test the client/server connections all work – while now not used anywhere in the main code, it’s a nice example of how behaviour can be built into messages very simply, only overriding two functions.

using System;

using Shared;

namespace NetCore.Messages

{

// Test message, used in debugging

public class TestMessage

: Message

{

// Internal message

public string Message { get; protected set; }

public TestMessage()

:this(null)

{

}

public TestMessage(string Message)

{

this.Message = Message;

}

public override void Serialise(Writer Writer)

{

base.Serialise(Writer);

Writer.Write(Message);

}

public override void Deserialise(Reader Reader)

{

Message = Reader.ReadString();

}

}

}

NetCore.Messages.ConnectMessage

*ConnectMessage* is another simple class – it is sent by the Client to the Server upon connection, and provides simple initial information: the logged on user’s username, as well as the computer name of the machine the Client is running on. This is used to display connection information, determine the access level of the user, and later provide relevant information to the user about Bookings occurring in the room they’re currently in.

using System;

using Shared;

namespace NetCore.Messages

{

// Message used when a client connects to the server

public class ConnectMessage

: Message

{

// Logged on user's username

public string Username { get; protected set; }

// Computer name of the client

public string ComputerName { get; protected set; }

public ConnectMessage()

:this(null, null)

{

}

public ConnectMessage(string Username, string ComputerName)

{

this.Username = Username;

this.ComputerName = ComputerName;

}

public override void Serialise(Writer Writer)

{

base.Serialise(Writer);

Writer.Write(Username);

Writer.Write(ComputerName);

}

public override void Deserialise(Reader Reader)

{

Username = Reader.ReadString();

ComputerName = Reader.ReadString();

}

}

}

NetCore.Messages.DisconnectMessage

*DisconnectMessage* is sent by a Client to a Server if the Client disconnects, and sent by the Server to all connected Clients if the Server is stopped. It contains minimal information on the error that occurred – whether it was expected or unexpected.  
This is used to detect a “soft” disconnect, when a program’s logic tells it to disconnect for whatever reason. A “hard” disconnect is when a program is forced to terminate before it can send any message, and is dealt with as an assumed unexpected disconnect.

using System;

using Shared;

namespace NetCore.Messages

{

// Records the type of disconnection

public enum DisconnectType

: byte

{

Expected, // eg. Logoff, application closed

Unexpected, // eg. Process crashed

}

//Message sent to the Server when a Client closes

public class DisconnectMessage

: Message

{

// Type of disconnection

public DisconnectType Reason { get; protected set; }

public DisconnectMessage()

:this(DisconnectType.Unexpected)

{

}

public DisconnectMessage(DisconnectType Reason)

{

this.Reason = Reason;

}

public override void Serialise(Writer Writer)

{

base.Serialise(Writer);

Writer.Write((byte)Reason);

}

public override void Deserialise(Reader Reader)

{

Reason = (DisconnectType)Reader.ReadByte();

}

}

}

NetCore.Messages.DataMessages.InitialiseMessage

*InitialiseMessage* is sent by the Server to a Client upon it connecting. It holds a snapshot of the database’s state at the time of sending, and is used to bulk-import all the data from the DB to the client. Subsequent messages may be sent creating/editing/deleting records, but this message transfers the majority of the data in one go.  
The frame of data is the only information transferred by this message.

using System;

using Shared;

using Data;

namespace NetCore.Messages.DataMessages

{

// Sent by Server on Client connection

public class InitialiseMessage

: Message

{

// Frame of all data in the repository

public DataSnapshot Snapshot { get; set; }

public InitialiseMessage()

:this(new DataSnapshot())

{

}

public InitialiseMessage(DataSnapshot Frame)

{

Snapshot = Frame;

}

public override void Serialise(Writer Out)

{

base.Serialise(Out);

Snapshot.Serialise(Out);

}

public override void Deserialise(Reader In)

{

Snapshot.Deserialise(In);

}

}

}

NetCore.Messages.DataMessages.UserInformationMessage

*UserInformationMessage* is sent by the Server to the Client upon connection, and provides information about the logged on user’s state – the actual User entity in the database that the logged on user corresponds to, which is used for access control in the UI, and the current Room they’re logged in from, which allows for room-specific notifications to be shown.

using System;

using Data.Models;

using Shared;

namespace NetCore.Messages.DataMessages

{

// Sent by the Server to the Client on connection

public class UserInformationMessage

: Message

{

// The User that the logged on user corresponds to

public User User { get; set; }

// Identifies the Room the user is currently in

public Room Room { get; set; }

public UserInformationMessage()

: this(null, null)

{

}

public UserInformationMessage(User User, Room Room)

{

this.User = User;

this.Room = Room;

}

public override void Serialise(Writer Out)

{

base.Serialise(Out);

Out.Write(User != null);

if (User != null)

User.Serialise(Out);

Out.Write(Room != null);

if (Room != null)

Room.Serialise(Out);

}

public override void Deserialise(Reader In)

{

if (In.ReadBool())

User = DataModel.DeserialiseExternal<User>(In);

if (In.ReadBool())

Room = DataModel.DeserialiseExternal<Room>(In);

}

}

}

NetCore.Messages.DataMessage.DataMessage

*DataMessage*s are sent between the Server and Client in order to synchronise records of the internal database model. Whenever an item is edited, added or deleted from the database, a message should be sent recording the change from where the change occurred to all other machines.  
The flag *Delete* records whether the change in question was actually the deletion operation, as deleting an item requires different processing to simply editing one.

using System;

using Data.Models;

using Shared;

namespace NetCore.Messages.DataMessages

{

// Sent when an item in the database is changed somehow

public class DataMessage

: Message

{

// The item that was changed

public DataModel Item { get; set; }

// Whether or not the item was deleted

public bool Delete { get; set; }

public DataMessage()

: this(null, false)

{

}

public DataMessage(DataModel Item, bool Delete)

{

this.Item = Item;

this.Delete = Delete;

}

public override void Serialise(Writer Out)

{

base.Serialise(Out);

Out.Write(Delete);

Item.Serialise(Out);

}

public override void Deserialise(Reader In)

{

Delete = In.ReadBool();

Item = DataModel.DeserialiseExternal(In);

}

}

}

NetCore.Server.Client

The Client class is actually part of the Server’s code, confusingly – it represents a logical Client of the Listener class, and provides functions to communicate with this remote client. This class is extremely important, it details how the Listener interacts with the remote clients.

using System;

using System.Net;

using System.Net.Sockets;

using System.IO;

using NetCore.Messages;

using Shared;

namespace NetCore.Server

{

// Delegate functions for the events defined below

public delegate void MessageReceivedHandler(Client Sender, Message Message);

public delegate void DisconnectHandler(Client Sender, DisconnectMessage Message);

// Used by Listener to represent a virtual "client" (wrapper around a socket, effectively).

public class Client

: IDisposable

{

// Local username of the user logged on with the client

public string Username { get; protected set; }

// Name of the machine the user's logged in on

public string ComputerName { get; protected set; }

// Remote network endpoint, hold IP and Port number

public IPEndPoint Remote { get { return (IPEndPoint)Connection.RemoteEndPoint; } }

// Internal Socket used for IO

protected Socket Connection { get; set; }

public bool Connected { get { return Connection.Connected; } }

// Internal Stream used to read data from the network

protected NetworkStream Stream { get; set; }

// Readers/Writers from the stream (defined in Shared library)

protected NetReader In { get; set; }

protected NetWriter Out { get; set; }

// Events for important actions

public event MessageReceivedHandler MessageReceived;

public event DisconnectHandler Disconnect;

// Internal buffer object, used to receive messages into

private byte[] Buffer { get; set; }

// Private constructor - Clients can only be created by calling

// the Create static method provided way below

private Client(ConnectMessage m, Socket Connection)

{

The constructor is private – in order to create a Client, the static method *Create* needs to be called. This is because the creation is actually a fairly intensive process, so should be kept separate from the lightweight initialisation done by a constructor.

Username = m.Username;

ComputerName = m.ComputerName;

this.Connection = Connection;

Stream = new NetworkStream(Connection);

In = new NetReader(Stream);

Out = new NetWriter(Stream);

MessageReceived = delegate { };

Disconnect = delegate { };

}

public void Dispose()

{

// On disposal, send a disconnect message

Send(new DisconnectMessage(DisconnectType.Expected));

Stream.Dispose();

In.Dispose();

Out.Dispose();

Connection.Close();

}

public void Start()

{

// Start reading from the stream

if (Connected)

StartRead();

}

public void Stop()

{

// Stopping is just disposing, effectively

Dispose();

}

private void StartRead()

When reading, receive a single notification byte asynchronously as a marker, before synchronously reading the rest of the message. This makes code simpler, and the network tasks far easier. Receiving a *DisconnectMessage* isn’t handled the same as other messages – it’s consumed internally to trigger the disconnect event, while other messages are passed up to the calling code.

{

// Asynchronously read the notification byte

In.BeginReadByte(Stream\_ReadComplete);

}

// Called when an asynchronous read completes

private void Stream\_ReadComplete(IAsyncResult Result)

{

try

{

// Retrieve the notification byte

In.EndReadByte(Result);

Result.AsyncWaitHandle.Dispose();

// Read a unknown type of message

Message New = Message.ReadMessage(In);

// If it's a disconnection, handle it internally

if (New is DisconnectMessage)

{

// Signal the disconnection

Disconnect(this, (DisconnectMessage)New);

return;

}

else // Not a disconnection, allow controlling code to handle it

MessageReceived(this, New);

StartRead(); // Go for another message read

return;

}

catch (ObjectDisposedException)

{ } // Stream was closed

catch (IOException)

{ } // Stream was closed

// If something went wrong, disconnect unexpectedly

Disconnect(this, new DisconnectMessage(DisconnectType.Unexpected));

}

// Send a message to a single client

public void Send(Message m)

{

try

{

lock (Connection) // Thread safe

{

if (Connected)

m.Serialise(Out); // Serialise the message out

}

}

catch

{

// If the send failed, treat it as a disconnect

Disconnect(this, new DisconnectMessage(DisconnectType.Unexpected));

}

}

// Provide nice description string

public override string ToString()

{

if (!Connected)

return "Disconnected client";

else

return Username + "@" + ComputerName + " on " + Remote.Address.ToString() + ":" + Remote.Port;

}

// Statically accept a client

public static Client Create(Socket Connection)

{

// Create a new reader around the provided socket

NetReader Reader = new NetReader(new NetworkStream(Connection));

Reader.ReadByte(); // Read the single notification byte

// Return a new client, initialising it using a received ConnectMessage

return new Client(Message.ReadMessage<ConnectMessage>(Reader), Connection);

}

}

This static *Create* function returns a new *Client* by reading the notification byte of a message followed by a *ConnectMessage*, passing the message to the private constructor of this class. This is good practice as it removes long-running/blocking code (receiving the data) from the constructor of the class, leaving only light initialisation work to be done there.

}

NetCore.Server.Listener

The *Listener* class is handles most of the internal workings of a server – accepting clients, sending and receiving messages, handling disconnections. It’s a wrapper around a Socket, and can provide both event-based message handling or buffer-based, where messages are stored in a buffer until they are consumed.

using System;

using System.Collections.Generic;

using System.Net;

using System.Net.Sockets;

using System.Threading.Tasks;

using NetCore.Messages;

namespace NetCore.Server

{

// Delegate functions for events created by the Listener class

public delegate void ClientConnectHandler(Listener Sender, Client Client);

public delegate void ClientDisconnectHandler(Listener Sender, Client Client, DisconnectMessage Message);

public delegate void ClientMessageReceivedHandler(Listener Sender, Client Client, Message Message);

// Used to listen for, accept, and control connections to Clients

public class Listener

: IDisposable

{

// The internal socket to listen on

protected Socket Inner { get; set; }

// The local network endpoint the server runs on

public IPEndPoint Endpoint { get { return (IPEndPoint)Inner.LocalEndPoint; } }

// A list of all clients that have connected to the Listener

public IList<Client> Clients { get; protected set; }

// A thread-safe, blocking queue of messages received

public BlockingQueue<ClientMessagePair> Messages { get; protected set; }

// If set to true, messages will be stored in the Messages list, for consumption later.

// If set to false, messages will be handled by firing the message received event

public bool BufferMessages { get; protected set; }

// Events using the delegates defined above which are fired when something important happens

public event ClientConnectHandler ClientConnect;

public event ClientDisconnectHandler ClientDisconnect;

public event ClientMessageReceivedHandler ClientMessageReceived;

// Internal thread that accepts clients asynchronously

protected Task AcceptingTask { get; set; }

// Internal flag representing whether the Listener is currenly running

protected bool Run { get; set; }

These constructors take fewer arguments that the one below, and effectively just fill in default values/change the format of the parameters to match the final one, so as to reduce code duplication.

public Listener(int Port)

: this(new IPEndPoint(IPAddress.Any, Port))

{

}

public Listener(string IP, int Port)

: this(new IPEndPoint(IPAddress.Parse(IP), Port))

{

}

public Listener(IPEndPoint Endpoint)

: this(Endpoint, new List<Client>())

{

// By default, use a List as the underlying collection

}

public Listener(IPEndPoint Endpoint, IList<Client> ClientListType)

{

Clients = new List<Client>();

// Create a new TCP/IP socket

Inner = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);

Inner.Bind(Endpoint); // Bind to the local endpoint

Inner.Listen(50); // Start listening with a backlog of 50 connections

Using an *IList<Client>* allows collections other than a *List* to be used – such as an *ObservableCollection*, which provides more features than the default one. This allows for more customisation and control.

Clients = new List<Client>();

Messages = new BlockingQueue<ClientMessagePair>();

ClientConnect = delegate { };

ClientDisconnect = delegate { };

ClientMessageReceived = delegate { };

Run = false;

}

public void Dispose()

{

// If disposed, just call stop, which performs correct cleanup

Stop();

}

// Start running the listener on the endpoint provided earlier, handling messages as specified

public void Start(bool BufferMessages)

{

// If told to start while already running, ignore the call

if (AcceptingTask != null && AcceptingTask.Status == TaskStatus.Running)

return;

this.BufferMessages = BufferMessages;

Run = true;

// Start asynchronously accepting connections

AcceptingTask = Task.Factory.StartNew(Accept);

}

public void Stop()

{

Run = false;

lock (Clients)

{

foreach (Client c in Clients)

{

try

{

// Send a disconnect message, then dispose the connection

c.Send(new DisconnectMessage(DisconnectType.Expected));

}

catch { }

finally { c.Dispose(); }

}

}

Inner.Dispose();

// Wait for the accepting task to end, just to ensure the

// listener's completely shut down before control is released

AcceptingTask.Wait();

}

protected void Accept()

{

while (Run)

{

Client New = null;

try

{

// Acept a client and exchange a ConnectionMessage

New = Client.Create(Inner.Accept());

}

catch (Exception e)

{

if (e is SocketException) // Assume Listener's been told to stop

break; // Stop listening

}

// Assign event handlers to when a client's message is received

New.MessageReceived += Client\_MessageReceived;

New.Disconnect += Client\_Disconnect;

lock (Clients) // Ensure thread safe access

Clients.Add(New);

// Fire the ClientConnect event

ClientConnect(this, New);

New.Start(); // Set the client listening for new messages

}

}

// Send a message to all connected clients

public void Send(Message Msg)

{

lock (Clients) // Thread safe

{

foreach (Client c in Clients)

{

try

{

c.Send(Msg);

}

catch { }

}

}

}

// Called when a message is received from a client

protected void Client\_MessageReceived(Client Sender, Message Msg)

{

if (BufferMessages) // Store message in message queue

Messages.Add(new ClientMessagePair(Sender, Msg));

else // Fire message received event

ClientMessageReceived(this, Sender, Msg);

}

protected void Client\_Disconnect(Client Sender, DisconnectMessage Msg)

{

Event handlers for receiving a message and for a client disconnecting. These are invoked on events fired from the clients, then usually fire the corresponding event on this listener – allowing the controlling code to handle individual client disconnections etc.

// Detach event handlers

Sender.Disconnect -= Client\_Disconnect;

Sender.MessageReceived -= Client\_MessageReceived;

Sender.Dispose();

lock (Clients) // Thread safe

Clients.Remove(Sender);

// Fire disconnection event

ClientDisconnect(this, Sender, Msg);

}

}

// Groups together a Client with a Message, for utility

This small class is effectively just a pairing of a received *Message* object with the *Client* that sent it. As it is used solely within the *Listener* class and has no notable features, I’ve included it within this section.

public class ClientMessagePair

{

public Client Client { get; set; }

public Message Message { get; set; }

public ClientMessagePair()

: this(null, null)

{

}

public ClientMessagePair(Client Client, Message Message)

{

this.Client = Client;

this.Message = Message;

}

}

}

NetCore.Client.Connection

The *Connection* class is the Client’s part of the networking code. It provides the ability to connect to a Server (that uses the *Listener* class for communications), and provides functionality based around this role.

using System;

using System.Net;

using System.Net.Sockets;

using NetCore.Messages;

using Shared;

namespace NetCore.Client

{

// Delegate function signatures for the events defined below

public delegate void DisconnectHandler(Connection Sender, DisconnectMessage Message);

public delegate void MessageReceivedHandler(Connection Sender, Message Message);

// Used by the actual network client

public class Connection

: IDisposable

{

// Inner socket to wrap around, and use for network IO

public Socket Inner { get; protected set; }

// Flag representing connection status. Really just a proxy for the internal socket's flag

public bool Connected { get { return Inner.Connected; } }

// The stream being read from

protected NetworkStream Stream { get; set; }

protected NetReader In { get; set; }

protected NetWriter Out { get; set; }

// Events to signal disconection and messages being received

public event DisconnectHandler Disconnect;

public event MessageReceivedHandler MessageReceived;

The internal data buffer is used during the asynchronous receive call – it’s where data is written to during the call.

// Internal buffer of data received

private byte[] Buffer { get; set; }

public Connection()

{

Disconnect = delegate { };

MessageReceived = delegate { };

}

public void Dispose()

{

try

{

// On Dispose, send an unexpected disconnect signal.

// If we've already disconnected cleanly, this will just fail

Send(new DisconnectMessage(DisconnectType.Unexpected));

}

catch { }

Stream.Dispose();

Out.Dispose();

}

// Alternative overloaded signature for next function

public bool Connect(string ServerAddress, ushort Port, ConnectMessage ConnectionMessage)

{

IPAddress Target;

if (!IPAddress.TryParse(ServerAddress, out Target))

return false;

return Connect(new IPEndPoint(Target, Port), ConnectionMessage);

}

// Attempts to connect to a specified server with a given connection message

public bool Connect(IPEndPoint Server, ConnectMessage ConnectionMessage)

{

// Initialise socket for TCP/IP communications

Inner = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);

try

{

Connecting involves connecting the basic TCP/IP socket, initialising the IO devices used to read from the socket, then sending a connection message containing information about the current client. After this is done, the asynchronous read cycle is started.

// Try to connect

Inner.Connect(Server);

// Set up IO streams around the socket

Stream = new NetworkStream(Inner);

In = new NetReader(Stream);

Out = new NetWriter(Stream);

// Send off the initial connection message

Send(ConnectionMessage);

// Begin reading from the Server

StartRead();

return true;

}

catch

{

return false;

}

}

public void Close(DisconnectType Reason)

{

Send(new DisconnectMessage(Reason));

Dispose();

}

protected void StartRead()

{

*StartRead* tells the input stream to read a byte asynchronously, and call a function on completion. This function then reads in the actual message that was sent, handles it appropriately, then starts reading another byte, repeating the cycle.

// Read the notification byte

In.BeginReadByte(Stream\_ReadComplete);

}

protected void Stream\_ReadComplete(IAsyncResult Result)

{

try

{

// Read and ignore the notification byte

In.EndReadByte(Result);

Result.AsyncWaitHandle.Dispose();

// Read in the message preceded by the notification byte

Message New = Message.ReadMessage(In);

if (New is DisconnectMessage) // D/C if neccessary

{

Disconnect(this, (DisconnectMessage)New);

return;

}

else // Otherwise fire the message received event

MessageReceived(this, New);

StartRead(); // Go for another read

}

catch

{

Disconnect(this, new DisconnectMessage(DisconnectType.Unexpected));

}

}

// Try to send a message to the Server

public bool Send(Message Msg)

{

try

{

lock (Inner) // Thread safe

Msg.Serialise(Out);

}

catch

{

return false;

}

return true;

}

}

}

## E:\Burford\Year 13\Computing\Project\_Writeup\Resources\Maintenance\Server.pngServer

The Server executable’s codebase is surprisingly small – the majority of the code is already written in the NetCore library, and only handlers for the events etc need to be defined and hooked up.  
There are 4 classes defined here: *MailHelper*, which is a utility class for sending emails to teachers with information about changes made to Bookings; *Settings*, another utility class which makes loading and using settings defined in the “Settings.txt” file easier; *DataRepository*, which is the first concrete implementation of the *IDataRepository* interface defined in the Data library – this class uses Entity Framework to actually store the data in a database, and handles actions associated with this (this is perhaps the most complicated class in this module); *Program*, which holds the entry point for the program (the function *Main*)as well as handlers for events fired by the *Listener* class defined in the NetCore assembly, which actually handle the Client/Server communication.

The following table describes the settings that can be stored in the “Settings.txt” file, as these serve as application settings.

|  |  |  |
| --- | --- | --- |
| Settings Name | Description | Default value |
| DatabasePath | The absolute path to the database file. | <current directory>\Data.mdf |
| Port | The network port to listen for client connections on. | 34652 |

Server.MailHelper

*MailHelper* is a utility class that makes it far easier to send emails to teachers when their bookings are edited. It contains 4 constants which denote various internal settings for email transfer (the password has been blacked out for security reasons), and a single static function which sends an email and reports its success/failure by its return value.

using System.Linq;

using System.Net;

using System.Net.Mail;

using Data.Models;

namespace Server

{

public class MailHelper

{

// Email Address to send from

public const string SenderAddress = "mail@burford.oxon.sch.uk";

// Password of above email address

public const string SenderPassword = " ";

// Mail server to use

public const string SMTPServer = "smtp.gmail.com";

// Port on mail server

public const int SMTPPort = 587;

// Tries to send an email to the Teacher specified by the given Booking,

// containing information on the booking itself, and whether it was edited/created

public static bool Send(Booking Booking, bool Edited)

{

try

{

// Make a new Email socket

using (SmtpClient Client = new SmtpClient(SMTPServer, SMTPPort))

{

// Secure, specific credentials, over network delivery

Client.EnableSsl = true;

Client.UseDefaultCredentials = false;

Client.Credentials = new NetworkCredential(SenderAddress, SenderPassword);

Client.DeliveryMethod = SmtpDeliveryMethod.Network;

// Create the message and fill out the fields

MailMessage Message = new MailMessage();

Message.From = new MailAddress(SenderAddress);

Message.To.Add(new MailAddress(Booking.Teacher.Email));

// Pick an appropriate subject based on what happened to the booking Message.Subject = Edited ? "One of your bookings has been edited" : "You've made a new booking";

// Fill out the body using information from the Booking object

Message.Body = (Edited ? "One of your bookings has been edited." : "You've made a new booking.") + "\r\n\r\n" +

"Date: " + Booking.Date.ToShortDateString() + "\r\n" +

"Period: " + Booking.TimeSlot + "\r\n" +

"Rooms: " + Booking.Rooms.Aggregate("", (a, r) => { return a + r.RoomName + ", "; }).TrimEnd(',', ' ') + "\r\n" +

"Recurrence: " + Booking.BookingType + "\r\n" +

"Subject: " + Booking.Subject.SubjectName + "\r\n" +

"Students: " + Booking.Students.Count + "\r\n";

// Send off the message

Client.SendAsync(Message, null);

The code above sets the message body to a nicely formatted list of the properties of the booking. Some tricks are used to format it, such as using the extension function *Aggregate* on the list of Rooms to construct a string containing all the room names separated by commas.

}

}

catch

{

return false;

}

return true;

}

}

}

Server.DataRepository

The Server’s *DataRepository* class provides the Entity Framework link to the actual database, by inheriting from *DbContext*. However, the definitions are slightly confusing: A *virtual DbSet<T>* is a single “table” in the database, while for example the *IDataRepository.Bookings* member is actually just satisfying the interface *IDataRepository*’s contract. The *.Include* expressions are used to make sure the variables are loaded correctly, as a side effect of how EF deals with variables.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Data.Entity;

using System.Threading;

using Data.Models;

using Data;

namespace Server

{

// The DataRepository interfaces with Entity Framework

// to store data in a database file. Inheriting from DbContext

// and listing properties in DbSet<T> makes the tables.

public class DataRepository

: DbContext, IDataRepository

{

List<Booking> IDataRepository.Bookings { get { return Bookings.Include(b => b.Rooms).Include(b => b.Students).Include(b => b.Subject).Include(b => b.Teacher).Include(b => b.TimeSlot).ToList(); } }

public virtual DbSet<Booking> Bookings { get; set; }

List<Department> IDataRepository.Departments { get { return Departments.Include(d => d.Teachers).Include(d => d.Rooms).ToList(); } }

public virtual DbSet<Department> Departments { get; set; }

List<Room> IDataRepository.Rooms { get { return Rooms.Include(r => r.Bookings).Include(r => r.Department).ToList(); } }

public virtual DbSet<Room> Rooms { get; set; }

public virtual DbSet<Student> Students { get; set; }

public virtual DbSet<Teacher> Teachers { get; set; }

List<User> IDataRepository.Users { get { return Users.ToList(); } }

public virtual DbSet<User> Users { get; set; }

List<Subject> IDataRepository.Subjects { get { return Subjects.Include(s => s.Bookings).ToList(); } }

public virtual DbSet<Subject> Subjects { get; set; }

List<TimeSlot> IDataRepository.Periods { get { return Periods.Include(p => p.Bookings).ToList(); } }

public virtual DbSet<TimeSlot> Periods { get; set; }

List<Class> IDataRepository.Classes { get { return Classes.Include(c => c.Owner).Include(c => c.Students).ToList(); } }

public virtual DbSet<Class> Classes { get; set; }

// Handy flag for testing this when I'm at home vs in school based off the user name

public static readonly bool Home = Environment.UserName == "Keith";

// Different versions of Entity Framework at home/school

private static readonly string ServerProvider = Home ? "MSSQLLocalDb" : "v11.0";

// Default path to the database

private static string \_Path = "Data.mdf";

public static string Path { get { return \_Path; } set { \_Path = value; } }

// Static object used for thread safety between multiple instantiations of this class

protected static object Lock = new object();

The *Home* readonly variable is used to denote whether the executable is running on my home machine or on a school machine, as the school has a different version of the development tools. The *ServerProvider* variable then uses this value to format the connection string correctly below.

// Initialises the class and passes in a connection string to Entity Framework

public DataRepository()

: base(@"data source=(LocalDb)\" + ServerProvider + @";AttachDbFilename=" + Settings.DatabasePath + ";Database=Data;integrated security=True;MultipleActiveResultSets=True;App=EntityFramework")

{

// Acquire the lock for this instance of the class

Monitor.Enter(Lock);

Many instances of this class can be instantiated at once, but they must only access the database one at a time. To this end, a static locking object is used – newly constructed objects must wait until they can acquire the lock before the construction finishes, then release the lock when they’re disposed of.

SetProxies(false);

}

protected override void Dispose(bool Disposing)

{

base.Dispose(Disposing);

// Release the lock for this instance

Monitor.Exit(Lock);

}

protected override void OnModelCreating(DbModelBuilder modelBuilder)

{

// Allows Students and Teachers to be stored in the same table for ease of access

modelBuilder.Entity<User>().Map<Student>(c => c.Requires("Discriminator").HasValue((int)UserType.Student));

modelBuilder.Entity<User>().Map<Teacher>(c => c.Requires("Discriminator").HasValue((int)UserType.Teacher));

base.OnModelCreating(modelBuilder);

}

// Returns a snapshot of the data in the database at the moment

public static DataSnapshot TakeSnapshot()

{

DataSnapshot Frame = new DataSnapshot();

using (DataRepository Repo = new DataRepository())

{

Repo.SetProxies(false);

// Extract all the tables' data

Frame.Bookings = Repo.Bookings.Include(b => b.Subject).Include(b => b.Teacher).Include(b => b.Rooms).Include(b => b.Students).ToList();

Frame.Departments = Repo.Departments.Include(d => d.Teachers).ToList();

Frame.Periods = Repo.Periods.Include(p => p.Bookings).ToList();

Frame.Rooms = Repo.Rooms.Include(r => r.Department).Include(r => r.Bookings).ToList();

Frame.Users = Repo.Users.ToList();

Frame.Subjects = Repo.Subjects.Include(s => s.Bookings).ToList();

Frame.Classes = Repo.Classes.Include(c => c.Students).ToList();

Repo.SetProxies(true);

}

return Frame;

}

// Minor tweaks at runtime - Entity Framework uses proxies

// to objects rather than the objects themselves, which can

// cause problems when editing them. Use of this function

// alleviates the problem

public void SetProxies(bool Enabled)

{

Entity Framework uses proxies to access variables, which can cause some problems when reading the values held by entities representing records in the tables. This function allows code to temporarily switch off proxies for sensitive pieces of code.

Configuration.ProxyCreationEnabled = Enabled;

Configuration.LazyLoadingEnabled = Enabled;

}

}

}

Server.Settings

The *Settings* class is nice and simple, it simply holds some static fields that represent settings in the “Settings.txt” file for the server, and provides functions to save to/load from the file.

using System;

using System.IO;

namespace Server

{

public sealed class Settings

{

// Default path to the database file

public static string DatabasePath { get { return \_DatabasePath; } private set { \_DatabasePath = value; } }

private static string \_DatabasePath = Environment.CurrentDirectory + "\\Data.mdf";

// Default Port to listen on

public static int Port { get { return \_Port; } private set { \_Port = value; } }

private static int \_Port = 34652;

// Default path to the Settings file

public static string Path { get { return \_Path; } set { \_Path = value; } }

private static string \_Path = "Settings.txt";

// Loads in the settings from the file

public static void Load()

{

try

{

using (Shared.TextReader In = new Shared.TextReader(File.OpenRead(Path)))

{

// Read a key on one line

string Key = In.ReadString();

// Set the appropriate variable depending on the key

if (Key == "DatabasePath")

DatabasePath = In.ReadString();

else if (Key == "Port")

Port = In.ReadInt32();

}

}

catch { }

}

// Save the existing settings to the file

public static bool Save()

{

try

{

using (Shared.TextWriter Out = new Shared.TextWriter(File.OpenWrite(Path)))

{

// Output all the variables

Out.Write("DatabasePath");

Out.Write(DatabasePath);

Out.Write("Port");

Out.Write(Port);

}

}

catch

{

return false;

}

return true;

}

}

}

Server.Program

The *Program* class holds the entry point for the executable, as well the object of the *Listener* class which runs all the server code. The remaining functions are either handlers for the events fired by the *Listener* object, or utility functions to reduce code duplication.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Windows.Media;

using System.Data.Entity;

using NetCore.Server;

using NetCore.Messages;

using NetCore.Messages.DataMessages;

using Data;

using Data.Models;

namespace Server

{

class Program

{

// This is the actual "Server" - runs all the networking code.

// All this class does is specify handlers for events provided by the listener

static Listener Listener { get; set; }

static void Main(string[] args)

{

Print("Initialised data", ConsoleColor.Gray);

// Load the settings from the Settings.txt file

Settings.Load();

Print("Loaded settings", ConsoleColor.Gray);

// Flag to represent whether the server is in the middle of shutting down

bool Closing = false;

// Initialise the Listener with the port defined in the settings file

Listener = new Listener(Settings.Port);

try

{

// Hook up the event handlers - these define the actual action taken by the server

Listener.ClientConnect += ClientConnected;

Listener.ClientDisconnect += ClientDisconnect;

Listener.ClientMessageReceived += ClientMessageReceived;

// Start and don't buffer messages - use events instead

Listener.Start(false);

Print("Listener started...", ConsoleColor.Green);

Event handlers are hooked up, server is started, and then the program waits for a keypress, before shutting down the server. The central logic is very simple, but the handlers themselves are somewhat complex.

// Wait for a keypress (to signal exit)

Console.ReadKey(true);

// Shut down the server, unhook the handlers

Closing = true;

Listener.Stop();

Print("Listener stopped...", ConsoleColor.Red);

Listener.ClientConnect -= ClientConnected;

Listener.ClientDisconnect -= ClientDisconnect;

Listener.ClientMessageReceived -= ClientMessageReceived;

}

catch (Exception e)

{

// If an exception is fired while the server's shutting down,

// it's usually a send error and is safe to ignore

if (!Closing)

{

Print("Error: " + e.ToString(), ConsoleColor.Red);

Console.ReadKey(true);

}

}

try

{

Listener.Dispose();

}

catch { }

}

// Called when client connects to the server

static void ClientConnected(Listener Sender, Client c)

{

// Notification message

Print(c.ToString() + " connected. Connected clients: " + Sender.Clients.Count, ConsoleColor.Green);

// Take an frame of the current database

DataSnapshot Frame = DataRepository.TakeSnapshot();

// Send the data initialisation message

c.Send(new InitialiseMessage(Frame));

// Send info on the user that's logged in

c.Send(new UserInformationMessage(Frame.Users.Where(u => u.LogonName == c.Username).SingleOrDefault(), Frame.Rooms.Where(r => r.ComputerNames.Contains(c.ComputerName)).FirstOrDefault()));

}

// Called when a client disconnects

static void ClientDisconnect(Listener Sender, Client c, DisconnectMessage Message)

{

// Just print a message notifying of the disconnection

Print(c.ToString() + " disconnected. Reason: " + Message.Reason.ToString() + ". Connected clients: " + Sender.Clients.Count, ConsoleColor.DarkGreen);

}

// Called when a client sends a message

static void ClientMessageReceived(Listener Sender, Client c, Message Message)

{

// Output holds the text to be displayed at the end

string Output = null;

// Special case for TestMessages - just print their contents

if (Message is TestMessage)

These following blocks of code involves editing the right table for each sort of data item that can be received. Editing a *Booking* takes slightly more code, as an email is sent.

Output = "Message received from " + c.ToString();

else if (Message is DataMessage)

{

// We're dealing with a DataMessage

DataMessage Data = (DataMessage)Message;

if (Data.Item is Booking)

{

// Get references from the received object to the existing ones in the database

using (DataRepository Repo = new DataRepository())

Data.Item.Expand(Repo);

// Store whether the data was edited or created

bool Edited = EditDataEntry((Booking)Data.Item, Data.Delete);

// Form the output based on whether the item was deleted, edited, or created

Output = (Data.Delete ? "Delete" : Edited ? "Edit" : "Add") + " Booking received from " + c.ToString();

// Send an email and append an appropriate message to the end

if (MailHelper.Send((Booking)Data.Item, Edited))

Output += " Email sent to teacher.";

else

Output += " Email failed to send.";

}

else if (Data.Item is Class)

{

// Edit the data and form a suitable output

bool Edited = EditDataEntry((Class)Data.Item, Data.Delete);

Output = (Data.Delete ? "Delete" : Edited ? "Edit" : "Add") + " Class received from " + c.ToString();

}

else if (Data.Item is Department)

{

bool Edited = EditDataEntry((Department)Data.Item, Data.Delete);

Output = (Data.Delete ? "Delete" : Edited ? "Edit" : "Add") + " Department received from " + c.ToString();

}

else if (Data.Item is Room)

{

bool Edited = EditDataEntry((Room)Data.Item, Data.Delete);

Output = (Data.Delete ? "Delete" : Edited ? "Edit" : "Add") + " Room received from " + c.ToString();

}

else if (Data.Item is Subject)

{

bool Edited = EditDataEntry((Subject)Data.Item, Data.Delete);

Output = (Data.Delete ? "Delete" : Edited ? "Edit" : "Add") + " Subject received from " + c.ToString();

}

else if (Data.Item is User)

{

bool Edited = EditDataEntry((User)Data.Item, Data.Delete);

Output = (Data.Delete ? "Delete" : Edited ? "Edit" : "Add") + " User received from " + c.ToString();

}

else if (Data.Item is TimeSlot)

{

bool Edited = EditDataEntry((TimeSlot)Data.Item, Data.Delete);

Output = (Data.Delete ? "Delete" : Edited ? "Edit" : "Add") + " Period received from " + c.ToString();

}

}

// If the output's been set, print it

if (Output != null)

Print(Output, ConsoleColor.Gray);

}

// This function alters the database with a specific item, returning

// true if the item was edited and false if it was created

static bool EditDataEntry<T>(T Entry, bool Delete) where T : DataModel

{

This function is called by all the above if-statements to edit the right data. It loads in what’s needed, and makes changes as necessary, returning whether it created/edited and item.

bool Edited = false;

using (DataRepository Repo = new DataRepository())

{

Repo.SetProxies(true);

// Get the relevant table

DbSet<T> Set = Repo.Set<T>();

// We want references to related items

Entry.Expand(Repo);

if (Delete) // Remove if deleting

Set.Remove(Set.Single(e => e.Id == Entry.Id));

else

{

// Check for conflicts if necessary

if (!Entry.Conflicts(Set.ToList().Cast<DataModel>().ToList()))

{

if (Set.Any(m => m.Id == Entry.Id)) // Updating existing item

{

// Call Update on the object to preserve references

Set.ToList().Single(m => m.Id == Entry.Id).Update(Entry);

Edited = true;

}

else // Add new item

{

Set.Add(Entry);

Edited = false;

}

}

}

// Flush changes

Repo.SaveChanges();

// Send the update to all clients

Listener.Send(new DataMessage(Entry, Delete));

}

return Edited;

}

// This simple function prints the desired text in the given colour

static void Print(string Text, ConsoleColor Colour)

This function simply reduces code bloat by creating a way to print in colour using one line (function call) rather than many.

{

// Thread safe

lock (Console.Out)

{

Console.ForegroundColor = Colour;

Console.WriteLine(Text);

Console.ForegroundColor = ConsoleColor.Gray;

}

}

}

}

## Client

The Client assembly is the largest codebase in the project – most of this code is in the appearance and logic for the UI, which is written using Windows Presentation Foundation.  
This executable is the main one seen by the end user – the server is only run once in an installation, and only admins should really see it. This executable, on the other hand, is used by every user of the system.

There are 4 main areas of the code for this project:

1. There are two “main” (displayed) windows – the Timetable view (*MainWindow*) and the Admin control panel (*AdminWindow*). Both of these windows have associated custom controls that have been written for them (*StudentSelector*, *TimetableTile*, *TimetableDisplay*). There is also a special class (*TrayIcon*) which is technically a window but is never actually shown. This handles the clickable icon in the taskbar tray. Also included in this informal group is the App class, which houses the entry point of the executable, as well as controlling all the basic networking tasks like connection etc.
2. The Edit windows are a collection of windows which all inherit from the *EditWindow* class. These provide windows to edit individual subclasses of *DataModel*, eg *Booking* and *Room*. Whilst there are many of these windows, they are all fairly simple from a purely code-based perspective – the design of some are quite complex.
3. The Converters are a collection of classes used by the windows/controls to perform conversions between the values of variables in the code-behind with (usually) strings which are displayed in the UI. These are all fairly simple, each having only two functions, which are very similar in each one.
4. The remaining classes, which are just the *Settings* and *DataRepository* classes. These provide useful services to code throughout the project.

The order of the classes described in the following sections is slightly oddly ordered – it’s ordered with respect to which classes are used in which other classes, so that classes are analysed before they’re then used in later classes. This does mean that they don’t follow the conventional order of depth of namespace (ie list all classes in namespace *Client* followed by those in namespace *Client.TimetableDisplay*), as it’ll be harder to understand the code that way.

The following table describes the settings that can be stored in the “Settings.txt” file, as these serve as application settings.

|  |  |  |
| --- | --- | --- |
| Settings Name | Description | Default value |
| ServerAddress | The IPv4 address of the computer running the Server executable. | *No default value* |
| ServerPort | The network port to try to connect to the server on (should be the same value as the Server’s Port setting) | 34652 |

Client.DataRepository

The Client’s *DataRepository* is a rather interesting class – it mimics the behaviour of the Server’s class with the same name, but without actually storing the data in a database. Instead, changes to the collections of items it provides are noted, undone, and then the Server is notified of the change. Simultaneously, any messages received by the Server relating to *DataModel* items are consumed and used to update the collections. Such changes can be detected by other Client code through an exposed event.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading;

using System.Collections;

using System.Collections.ObjectModel;

using System.Collections.Specialized;

using NetCore.Client;

using NetCore.Messages;

using NetCore.Messages.DataMessages;

using Data;

using Data.Models;

namespace Client

{

// Delegate function signatures for the events described below

public delegate void DataChangedHandler(Type ChangedType);

// Holds a copy of the data in the database that's synchronised with the server

public class DataRepository

: IDisposable, IDataRepository

{

// Reference to the connection to the server used

private static Connection Server { get; set; }

// List of all the Bookings

List<Booking> IDataRepository.Bookings { get { return Bookings.ToList(); } }

private static ObservableCollection<Booking> \_Bookings = new ObservableCollection<Booking>();

public ObservableCollection<Booking> Bookings

{

The Client provides *ObservableCollection*s of the *DataModel*s rather than *List*s as the interface specifies, so an explicit property is used to cast them.

get { return \_Bookings; }

set { \_Bookings = value; }

}

// All Departments

List<Department> IDataRepository.Departments { get { return Departments.ToList(); } }

private static ObservableCollection<Department> \_Departments = new ObservableCollection<Department>();

public ObservableCollection<Department> Departments

{

get { return \_Departments; }

set { \_Departments = value; }

}

List<Room> IDataRepository.Rooms { get { return Rooms.ToList(); } }

private static ObservableCollection<Room> \_Rooms = new ObservableCollection<Room>();

public ObservableCollection<Room> Rooms

{

get { return \_Rooms; }

set { \_Rooms = value; }

}

List<User> IDataRepository.Users { get { return Users.ToList(); } }

private static ObservableCollection<User> \_Users = new ObservableCollection<User>();

public ObservableCollection<User> Users

{

get { return \_Users; }

set { \_Users = value; }

}

List<Subject> IDataRepository.Subjects { get { return Subjects.ToList(); } }

private static ObservableCollection<Subject> \_Subjects = new ObservableCollection<Subject>();

public ObservableCollection<Subject> Subjects

{

get { return \_Subjects; }

set { \_Subjects = value; }

}

List<TimeSlot> IDataRepository.Periods { get { return Periods.ToList(); } }

private static ObservableCollection<TimeSlot> \_Periods = new ObservableCollection<TimeSlot>();

public ObservableCollection<TimeSlot> Periods

{

get { return \_Periods; }

set { \_Periods = value; }

}

List<Class> IDataRepository.Classes { get { return Classes.ToList(); } }

private static ObservableCollection<Class> \_Classes = new ObservableCollection<Class>();

public ObservableCollection<Class> Classes

{

get { return \_Classes; }

set { \_Classes = value; }

}

// Holds references to the tables by loosely typed IList collections.

// Reduces code bloat later on

private static readonly Dictionary<Type, IList> Tables = new Dictionary<Type, IList>()

{

This *Dictionary* is used later to acquire a reference to a table of entities by virtue of the type stored in it. This makes the code neater, less complicated.

{ typeof(Booking), \_Bookings },

{ typeof(Department), \_Departments },

{ typeof(Room), \_Rooms },

{ typeof(User), \_Users },

{ typeof(Subject), \_Subjects },

{ typeof(TimeSlot), \_Periods },

{ typeof(Class), \_Classes },

};

// Internal locking object for thread safety

private static object Lock = new object();

// Whether to inform the server of model changes

private static bool ReportModelChanges { get; set; }

// Signals between threads indicating the completion of certain tasks

private static ManualResetEvent InitialisedEvent { get; set; }

private static ManualResetEvent UserEvent { get; set; }

// Reference to the current User and Room, set when an approriate message is received

private static User CurrentUser { get; set; }

private static Room CurrentRoom { get; set; }

// Fired when any colelction of items is changed

public static event DataChangedHandler DataChanged = delegate { };

// Whether this instance will check for thread safety before accessing the data

private bool LockData;

public DataRepository(bool LockData = true)

{

// Optionally allow one DataRepository to be instantiated at a time.

// Block until all other ones are Disposed.

this.LockData = LockData;

if (LockData)

Monitor.Enter(Lock);

}

// Initialise the database model from the server

public static Tuple<User, Room> Initialise(Connection Server, ConnectMessage Msg)

{

Initialising the *DataRepository* requires a message from the Server informing it of the initial model and the current user information.

try

{

// Thread safe

Monitor.Enter(Lock);

// Reset the signal that's set when the data's received

if (InitialisedEvent == null)

InitialisedEvent = new ManualResetEvent(false);

InitialisedEvent.Reset();

// Reset the signal that's set when the user information's received

if (UserEvent == null)

UserEvent = new ManualResetEvent(false);

UserEvent.Reset();

// Set the current server

DataRepository.Server = Server;

// Hook up network events

Server.MessageReceived += MessageReceived;

Server.Disconnect += Disconnected;

// Hook up data changed events

\_Bookings.CollectionChanged += Data\_CollectionChanged;

\_Departments.CollectionChanged += Data\_CollectionChanged;

\_Rooms.CollectionChanged += Data\_CollectionChanged;

\_Users.CollectionChanged += Data\_CollectionChanged;

\_Subjects.CollectionChanged += Data\_CollectionChanged;

\_Periods.CollectionChanged += Data\_CollectionChanged;

\_Classes.CollectionChanged += Data\_CollectionChanged;

// Send the connection message

Server.Send(Msg);

}

catch

{

return null;

}

finally

{

// Release the lock

Monitor.Exit(Lock);

}

try

{

// Wait for both signals to fire, signalling completion

InitialisedEvent.WaitOne();

UserEvent.WaitOne();

}

catch

{

// Disconnected during initialise

return null;

}

// Return the User and their Room (grouped together for easy return value)

return new Tuple<User, Room>(CurrentUser, CurrentRoom);

}

// Unlock the object on disposal

public void Dispose()

{

if (LockData)

Monitor.Exit(Lock);

}

// Take a frame of the information in the database model

public static DataSnapshot TakeSnapshot(bool Lock = true)

{

DataSnapshot Frame = new DataSnapshot();

using (DataRepository Repo = new DataRepository(Lock))

{

Being able to acquire a *DataSnapshot* is quite useful, but more useful for the Client’s repository is being able to load one. One of the two initialisation messages sent by the server contains a DataSnapshot containing the initial state of the server’s database.

Frame.Bookings = Repo.Bookings.ToList();

Frame.Departments = Repo.Departments.ToList();

Frame.Periods = Repo.Periods.ToList();

Frame.Rooms = Repo.Rooms.ToList();

Frame.Users = Repo.Users.ToList();

Frame.Subjects = Repo.Subjects.ToList();

Frame.Classes = Repo.Classes.ToList();

}

return Frame;

}

// Load in a snapshot to the database model

public static void LoadSnapshot(DataSnapshot Frame, bool Lock)

{

using (DataRepository Repo = new DataRepository(Lock))

{

Repo.Bookings.Clear();

Frame.Bookings.ForEach(b => Repo.Bookings.Add(b));

Repo.Departments.Clear();

Frame.Departments.ForEach(d => Repo.Departments.Add(d));

Repo.Periods.Clear();

Frame.Periods.ForEach(p => Repo.Periods.Add(p));

Repo.Rooms.Clear();

Frame.Rooms.ForEach(t => Repo.Rooms.Add(t));

Repo.Users.Clear();

Frame.Users.ForEach(u => Repo.Users.Add(u));

Repo.Subjects.Clear();

Frame.Subjects.ForEach(s => Repo.Subjects.Add(s));

Repo.Classes.Clear();

Frame.Classes.ForEach(c => Repo.Classes.Add(c));

// Run through all lists and expand the items within them

foreach (IList Table in Tables.Values)

After all the entities are added, we expand each *DataModel* within each table to ensure they all have the correct references.

{

foreach (DataModel d in Table)

{

d.Expand(Repo);

}

}

}

}

// Handler for when the server receives a message

private static void MessageReceived(Connection Sender, Message Msg)

{

This function handles receiving messages about changes to the database as well as initialisation messages. *InitialiseMessage* and *UserInformationMessage* are both initialisation messages, while any received *DataMessages* are inserted/edited/deleted as necessary. This is the main workhorse function of this class.

bool Locked = true;

Monitor.Enter(Lock);

// Don't echo chages back to the server - changes made in this

// function have been sent to us by the server

ReportModelChanges = false;

if (Msg is InitialiseMessage)

{

// Initialisation of the client - load in the data

LoadSnapshot((Msg as InitialiseMessage).Snapshot, false);

// Signal that we've received the initial data

InitialisedEvent.Set();

}

else if (Msg is UserInformationMessage)

{

// Information on the User and their Room

UserInformationMessage m = (Msg as UserInformationMessage);

User User = m.User;

Room Room = m.Room;

if (User == null)

throw new ArgumentNullException("Received a null user.");

if (Room == null)

throw new ArgumentNullException("Received a null room.");

// Acquire references to the actual user/room

DataSnapshot Frame = TakeSnapshot(false);

CurrentUser = Frame.Users.SingleOrDefault(u => u.Id == User.Id);

CurrentRoom = Frame.Rooms.SingleOrDefault(r => r.Id == Room.Id);

// Signal that we've received the user data

UserEvent.Set();

}

else if (Msg is DataMessage)

{

DataMessage Data = (DataMessage)Msg;

// Get references to linked objects

using (DataRepository Repo = new DataRepository(false))

Data.Item.Expand(Repo);

// If we're not deleting it, set references to this item

if (!Data.Delete)

Data.Item.Attach();

else // Otherwise remove references

Data.Item.Detach();

// Get the right table from the dictionary

IList Table = Tables[Data.Item.GetType()];

// Find the index of the item in the table

int Index = -1;

for (int x = 0; x < Table.Count; x++)

{

if (((DataModel)Table[x]).Id == Data.Item.Id)

{

Index = x;

break;

}

}

if (!Data.Delete)

{

if (Index < 0) // Doesn't already exist, add it

Table.Add(Data.Item);

else // Already exists, update it

((DataModel)Table[Index]).Update(Data.Item);

}

else // Delete it

Table.RemoveAt(Index);

// Release the lock

Monitor.Exit(Lock);

// Note that we've already released it

Locked = false;

// Fire the change of data handler

DataChanged(Data.Item.GetType());

}

ReportModelChanges = true; // Continue reporting changes

if (Locked) // Release the lock if we haven't already

Monitor.Exit(Lock);

}

// On the server disconnecting

private static void Disconnected(Connection Sender, DisconnectMessage Message)

{

If the Server disconnects, reset all the handlers/signals used, as when we connect they’ll be re-registered.

// Unhook handlers

Server.MessageReceived -= MessageReceived;

Server.Disconnect -= Disconnected;

// Reset the signals

InitialisedEvent.Dispose();

InitialisedEvent = null;

UserEvent.Dispose();

UserEvent = null;

}

// On a collection changing somehow

private static void Data\_CollectionChanged(object sender, NotifyCollectionChangedEventArgs e)

If a table is modified, **undo** the change and notify the server of it. This way all clients are updated at the same time, when the server declares it.

{

// Only report if we're meant to

if (ReportModelChanges)

{

// Thread safe

lock (Lock)

{

// Turn off reporting so we don't end up calling this function again

ReportModelChanges = false;

if (e.NewItems != null)

{

foreach (DataModel d in e.NewItems)

{

// Remove the item that was just added, we wait for the

// server to send it back to us

((IList)sender).Remove(d);

// Send the new item marked for insertion

Server.Send(new DataMessage(d, false));

}

}

if (e.OldItems != null)

{

foreach (DataModel d in e.OldItems)

{

// Return the item to the list, wait for the server to

// tell us to remove it

((IList)sender).Add(d);

// Send the new item marked for deletion

Server.Send(new DataMessage(d, true));

}

}

// Turn reporting back on

ReportModelChanges = true;

}

}

}

}

}

Client.Settings

This class is similar to the Server’s *Settings* class, but slightly more flexible as the client can have different formats of settings. Data is stored in a dictionary and access is provided through static functions.

using System;

using System.Collections.Generic;

using System.IO;

namespace Client

{

public sealed class Settings

{

// Default path to the Settings.txt file

private static string \_Path = "Settings.txt";

public static string Path { get { return \_Path; } set { \_Path = value; } }

private static Dictionary<string, object> Inner { get; set; }

// Returns the value of a setting given the key

public static object Get(string Setting)

{

return Inner[Setting];

}

// Returns a strongly typed value of a setting given the key

public static T Get<T>(string Setting)

{

return (T)Convert.ChangeType(Get(Setting), typeof(T));

}

// Sets the value mapped to by a key

public static void Set(string Setting, object Value)

{

These functions all basically provide a static interface to the member functions of the internal dictionary.

Inner[Setting] = Value;

}

// Adds a new key-value mapping

public static void Add(string Setting, object Value)

{

Inner.Add(Setting, Value);

}

// Remove an existing key-value mapping

public static void Remove(string Setting)

{

Inner.Remove(Setting);

}

// Remove all mappings

public static void Clear()

{

Inner.Clear();

}

// Checks if the key exists already

public static bool Contains(string Setting)

{

return Inner.ContainsKey(Setting);

}

// Loads in the settings from the file, returning success/failure

public static bool Load()

{

try

{

Inner = new Dictionary<string, object>();

using (Shared.TextReader In = new Shared.TextReader(File.OpenRead(Path)))

{

while (true)

{

string Key = In.ReadString();

string Value = In.ReadString();

if (Key == null || Value == null)

break;

These functions handle saving/loading settings from a file. Nothing special, they use text encoding and simply read/write the key and value. Return true/false if they succeed/fail.

Inner.Add(Key, Value);

}

}

}

catch

{

return false;

}

return true;

}

// Save the settings to the file, returning success/failure

public static bool Save()

{

try

{

using (Shared.TextWriter Out = new Shared.TextWriter(File.OpenWrite(Path)))

{

foreach (KeyValuePair<string, object> Setting in Inner)

{

Out.Write(Setting.Key);

Out.Write(Setting.Value);

}

}

}

catch

{

return false;

}

return true;

}

}

}

Client.Checkable

The *Checkable* generic class is very useful in the UI – it links a Boolean value representing whether a value has been selected (such as in a textbox) with a strongly typed object. This is extremely similar to the *Nullable* generic class in the .NET framework, but represents selection rather than nullity.  
Inheriting from *INotifyPropertyChanged* allows the UI to update itself to reflect changes on this object, which is extremely useful when binding to values.

using System;

using System.ComponentModel;

namespace Client

{

public class Checkable<T>

: INotifyPropertyChanged

{

// Whether the object is selected

protected bool \_Checked;

public bool Checked

{

get { return \_Checked; }

set { \_Checked = value; OnPropertyChanged("Checked"); }

}

// The object that can be selected

protected T \_Value;

public T Value

{

get { return \_Value; }

set { \_Value = value; OnPropertyChanged("Value"); }

}

public Checkable()

: this(Activator.CreateInstance<T>(), false)

{

}

public Checkable(T Value)

: this(Value, false)

{

}

public Checkable(T Value, bool Checked)

{

this.Value = Value;

this.Checked = Checked;

PropertyChanged = delegate { };

}

protected void OnPropertyChanged(string PropertyName)

{

if (PropertyChanged != null)

PropertyChanged(this, new PropertyChangedEventArgs(PropertyName));

}

public event PropertyChangedEventHandler PropertyChanged;

}

}

Client.Converters.BookingTypeToStringConverter

This class is an *IValueConverter* used in the UI that converts from a value specified by the *BookingType* enumeration to the string representing the booking type.

using System;

using System.Globalization;

using System.Windows.Data;

using Data.Models;

namespace Client.Converters

{

public class BookingTypeToStringConverter

: IValueConverter

{

public object Convert(object value, Type targetType, object parameter, CultureInfo culture)

{

return Enum.GetName(typeof(BookingType), value);

}

public object ConvertBack(object value, Type targetType, object parameter, CultureInfo culture)

{

return Enum.Parse(typeof(BookingType), (string)value);

}

}

}

Client.Converters.BooleanToVisibilityConverter

This class is an *IValueConverter* used in the UI that converts from a *bool* to a value specified by the *Visibility* enumerable, for showing/hiding UI controls.

using System;

using System.Globalization;

using System.Windows;

using System.Windows.Data;

namespace Client.Converters

{

public class BooleanToVisibilityConverter

: IValueConverter

{

public object Convert(object value, Type targetType, object parameter, CultureInfo culture)

{

if(value is bool)

{

return ((bool)value) ? Visibility.Visible : Visibility.Collapsed;

}

throw new ArgumentException("value was not of type bool.");

}

public object ConvertBack(object value, Type targetType, object parameter, CultureInfo culture)

{

if (value is Visibility)

{

return ((Visibility)value) == Visibility.Visible;

}

throw new ArgumentException("value was not of type Visibility.");

}

}

}

Client.Converters.IntToStringConverter

This class is an *IValueConverter* used in the UI that converts from an *int* to a *string*.

using System;

using System.Globalization;

using System.Windows.Data;

using System.Linq;

namespace Client.Converters

{

class IntToStringConverter

: IValueConverter

{

public object Convert(object value, Type targetType, object parameter, CultureInfo culture)

{

if (value is int)

return System.Convert.ToString((int)value);

return string.Empty;

}

public object ConvertBack(object value, Type targetType, object parameter, CultureInfo culture)

{

if (value is string)

if (!string.IsNullOrWhiteSpace(value as string))

return System.Convert.ToInt32(new string((value as string).Where(c => char.IsDigit(c)).ToArray()).PadLeft(1, '0'));

return 0;

}

}

}

Client.Converters.InverseBooleanConverter

This class is an *IValueConverter* used in the UI that inverts a *bool* passed in – the input/output types are both *bool*, but the conversion just “flips” the value of the input.

using System;

using System.Globalization;

using System.Windows.Data;

namespace Client.Converters

{

class InverseBooleanConverter

: IValueConverter

{

public object Convert(object value, Type targetType, object parameter, CultureInfo culture)

{

if (targetType != typeof(bool))

throw new ArgumentException("Target must be a boolean.");

return !(bool)value;

}

public object ConvertBack(object value, Type targetType, object parameter, CultureInfo culture)

{

return Convert(value, targetType, parameter, culture);

}

}

}

Client.Converters.InverseNullableBooleanConverter

This class is an *IValueConverter* used in the UI that inverts a *Nullable<bool>* passed in – the input/output types are the both *Nullable<bool>*, but the conversion just “flips” the value of the input.

using System;

using System.Globalization;

using System.Windows.Data;

namespace Client.Converters

{

class InverseNullableBooleanConverter

: IValueConverter

{

public object Convert(object value, Type targetType, object parameter, CultureInfo culture)

{

if (targetType != typeof(bool?) && targetType != typeof(bool))

throw new ArgumentException("Target must be a boolean.");

return !(bool?)value;

}

public object ConvertBack(object value, Type targetType, object parameter, CultureInfo culture)

{

return Convert(value, targetType, parameter, culture);

}

}

}

Client.Converters.InverseVisibilityConverter

This class is an *IValueConverter* used in the UI that inverts a value specified by the *Visibility* enumeration – it converts from *Visible* to *Collapsed* and back again.

using System;

using System.Globalization;

using System.Windows.Data;

using System.Windows;

namespace Client.Converters

{

class InverseVisibilityConverter

: IValueConverter

{

public object Convert(object value, Type targetType, object parameter, CultureInfo culture)

{

if (targetType != typeof(Visibility) && targetType != typeof(Visibility))

throw new ArgumentException("Target must be a Visibility.");

return ((Visibility)value) == Visibility.Collapsed ? Visibility.Visible : Visibility.Collapsed;

}

public object ConvertBack(object value, Type targetType, object parameter, CultureInfo culture)

{

return Convert(value, targetType, parameter, culture);

}

}  
}

Client.Converters.NullableToBoolConverter

This class is an *IValueConverter* used in the UI that converts any reference type into a *bool* representing whether the reference was *null*.

using System;

using System.Globalization;

using System.Windows.Data;

namespace Client.Converters

{

class NullableToBoolConverter

: IValueConverter

{

public object Convert(object value, Type targetType, object parameter, CultureInfo culture)

{

return value != null;

}

public object ConvertBack(object value, Type targetType, object parameter, CultureInfo culture)

{

throw new NotImplementedException();

}

}

}

Client.Converters.NullableToVisibilityConverter

using System;

This class is an *IValueConverter* used in the UI that converts any reference type into a value specified by the *Visibility* enumerable representing whether the reference was *null* – if it was null, *Collapsed* is returned, otherwise *Visible*.

using System.Globalization;

using System.Windows;

using System.Windows.Data;

namespace Client.Converters

{

class NullableToVisibilityConverter

: IValueConverter

{

public object Convert(object value, Type targetType, object parameter, CultureInfo culture)

{

return value == null ? Visibility.Collapsed : Visibility.Visible;

}

public object ConvertBack(object value, Type targetType, object parameter, CultureInfo culture)

{

throw new NotSupportedException("Back-conversion not supported");

}

}

}

Client.Converters.StringToIntConverter

This class is an *IValueConverter* used in the UI that converts a *string* to an *int*.

using System;

using System.Globalization;

using System.Windows.Data;

namespace Client.Converters

{

class StringToIntConverter

: IValueConverter

{

public object Convert(object value, Type targetType, object parameter, CultureInfo culture)

{

if (value is string)

return System.Convert.ToInt32(value as string);

return 0;

}

public object ConvertBack(object value, Type targetType, object parameter, CultureInfo culture)

{

if (value is int)

return System.Convert.ToString((int)value);

return string.Empty;

}

}

}

Client.TimetableDisplay.TimetableTile (Design)

<UserControl x:Class="Client.TimetableDisplay.TimetableTile"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

mc:Ignorable="d"

xmlns:converters="clr-namespace:Client.Converters"

HorizontalAlignment="Stretch" VerticalAlignment="Stretch"

d:DesignHeight="100" d:DesignWidth="100">

<UserControl.Resources>

<converters:NullableToVisibilityConverter x:Key="NullableToVisibilityConverter"/>

<converters:InverseVisibilityConverter x:Key="InverseVisibilityConverter"/>

<Storyboard x:Key="PulseEffect">

<DoubleAnimation Storyboard.TargetName="Outer" Storyboard.TargetProperty="Opacity" From="1.0" To="0.1" AutoReverse="True" Duration="0:0:1"/>

This storyboard is used in code to “pulse” a tile if the current user is involved in that booking.

</Storyboard>

</UserControl.Resources>

<Grid Name="Outer">

<Grid Name="Panel\_Booked" Visibility="{Binding Booking, Converter={StaticResource NullableToVisibilityConverter}}">

<Grid.RowDefinitions>

<RowDefinition Height="Auto"/>

<RowDefinition Height="\*"/>

</Grid.RowDefinitions>

<TextBlock Text="{Binding Booking.Subject.SubjectName}" FontSize="16" Margin="5,5,5,0" TextWrapping="NoWrap" TextTrimming="CharacterEllipsis" Grid.Row="0"/>

<TextBlock Text="{Binding Booking.Teacher.FormalName}" FontSize="16" Margin="5,5,5,0" TextWrapping="NoWrap" TextTrimming="CharacterEllipsis" Grid.Row="1"/>

</Grid>

<Grid Name="Panel\_Empty" Visibility="{Binding ElementName=Panel\_Booked, Path=Visibility, Converter={StaticResource InverseVisibilityConverter}}">

<TextBlock Text="Empty" FontSize="16" VerticalAlignment="Center" HorizontalAlignment="Center"/>

</Grid>

</Grid>

</UserControl>



A *TimetableTile* that’s been allocated a Booking



A *TimetableTile* that’s not been allocated a Booking

A *TimetableTile* being hovered over by the user



Client.TimetableDisplay.TimetableTile (Code-behind)

using System;

A TimetableTile is a small control that represents a single *Booking* object on the timetable display. Much of the work is done using bindings on the UI, the code for which is above, while the code in this file mostly deals with changing the background colour of the tile and providing properties for the UI to bind onto.

using System.Linq;

using System.Windows;

using System.Windows.Controls;

using System.Windows.Input;

using System.Windows.Media;

using Data.Models;

using System.Windows.Media.Animation;

namespace Client.TimetableDisplay

{

// Represents a single Tile on the timetable display

public partial class TimetableTile

: UserControl

{

// The time this tile represents

public TimeSlot Time { get; protected set; }

// The room this tile represents

public Room Room { get; protected set; }

// The Booking (or null) for the Booking in this slot

public Booking Booking { get; protected set; }

// The brush used to paint the background colour

public SolidColorBrush Brush { get; protected set; }

// Function object that determines how dark a tile gets when hovered over

public Func<float, float> BrightnessCurve { get; set; }

public TimetableTile(Booking Booking, TimeSlot Time, Room Room, User CurrentUser)

{

// Sets UI bindings to reference this object

DataContext = this;

this.Booking = Booking;

this.Time = Time;

this.Room = Room;

// Set the default brightness function to the default

BrightnessCurve = DefaultBrightnessCurve;

// Default to the background window colour

Brush = SystemColors.WindowBrush;

if (Booking != null) // If there's actually a booking in this slot, use its colour

Brush = new SolidColorBrush(Booking.Subject.Colour);

Background = Brush;

// Initialise the UI

InitializeComponent();

// Hook up mouse events

MouseEnter += TimetableTile\_MouseEnter;

MouseLeave += TimetableTile\_MouseLeave;

// If there's a booking in this slot

if (Booking != null && CurrentUser != null)

{

// If the current user is somehow involved in the booking (either

// as a student or teacher)

if (((CurrentUser is Student) && Booking.Students.Any(s => s.Id == CurrentUser.Id))

|| ((CurrentUser is Teacher) && Booking.Teacher.Id == CurrentUser.Id))

{

// Do a simple animation on the tiles to draw attention

Storyboard PulseEffect = (Storyboard)Resources["PulseEffect"];

PulseEffect.Begin(Outer);

Find the storyboard from the resources defined in the design file, and play it.

}

}

}

// When the mouse hovers over the tile

protected void TimetableTile\_MouseEnter(object sender, MouseEventArgs e)

{

// Change the background colour

Background = new SolidColorBrush(ScaleLuminosity(Brush.Color));

}

// When the mouse stops hovering over the tile

protected void TimetableTile\_MouseLeave(object sender, MouseEventArgs e)

{

// Reset the background colour

Background = new SolidColorBrush(Brush.Color);

}

The brightness curve is a function that takes a value between 0-1 and returns a shifted value that represents the brightness while being hovered over.The default is to simply cube the initial brightness.

// Initial brightness curve

public float DefaultBrightnessCurve(float Y)

{

return (float)Math.Pow(Y, 3);

}

// Changes the brightness of the given colour using the brightness curve

private Color ScaleLuminosity(Color c)

{

byte[] YUV = RGBToYUV(new byte[] { c.R, c.G, c.B });

// Calculate the new brightness using the provided colour curve

float NewBrightness = BrightnessCurve(YUV[0] / 255f);

YUV[0] = Clamp((int)Math.Round(NewBrightness \* 255f));

byte[] ScaledRGB = YUVToRGB(YUV);

return new Color() { A = c.A, R = ScaledRGB[0], G = ScaledRGB[1], B = ScaledRGB[2] };

}

// Converts an RGB array into a YUV array

private byte[] RGBToYUV(byte[] RGB)

{

if (RGB.Length != 3)

throw new ArgumentException("Invalid number of bytes provided.");

Calculations done here are from <https://msdn.microsoft.com/en-us/library/aa917087.aspx?f=255&MSPPError=-2147217396>

int R = RGB[0];

int G = RGB[1];

int B = RGB[2];

int Y = ((66 \* R + 129 \* G + 25 \* B + 128) >> 8) + 16;

int U = ((-38 \* R - 74 \* G + 112 \* B + 128) >> 8) + 128;

int V = ((112 \* R - 94 \* G - 18 \* B + 128) >> 8) + 128;

return new byte[] { Clamp(Y), Clamp(U), Clamp(V) };

}

// Converts a YUV array into an RGB array

private byte[] YUVToRGB(byte[] YUV)

{

if (YUV.Length != 3)

throw new ArgumentException("Invalid number of bytes provided.");

Calculations done here are from <https://msdn.microsoft.com/en-us/library/aa917087.aspx?f=255&MSPPError=-2147217396>

int C = YUV[0] - 16;

int D = YUV[1] - 128;

int E = YUV[2] - 128;

int R = (298 \* C + 409 \* E + 128) >> 8;

int G = (298 \* C - 100 \* D - 208 \* E + 128) >> 8;

int B = (298 \* C + 516 \* D + 128) >> 8;

return new byte[] { Clamp(R), Clamp(G), Clamp(B) };

}

// Limits the values taken by the input

private byte Clamp(int x)

{

if (x < byte.MinValue)

return byte.MinValue;

else if (x > byte.MaxValue)

return byte.MaxValue;

else

return (byte)x;

}

}

}

Client.TimetableDisplay.TimetableDisplay (Design)

<UserControl x:Class="Client.TimetableDisplay.TimetableDisplay"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

mc:Ignorable="d"

d:DesignHeight="300" d:DesignWidth="500"

Margin="5"

DataContext="{Binding RelativeSource={RelativeSource Self}}">

<Grid>

<Grid Name="Container"/>

</Grid>

</UserControl>



This is the *TimetableDisplay* control with some test data being used for the timeslots/rooms/bookings.

Client.TimetableDisplay.TimetableDisplay (Code-behind)

The *TimetableDisplay* is the most iconic control in the application – it’s the actual view of the timetable. The design code is obviously extremely simple, as the construction of the table is too complex to do in UI. Instead, the function *SetTimetable* within this code-behind handles the initialisation and layout of all the tiles that make up the grid.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Windows;

using System.Windows.Controls;

using System.Windows.Media;

using System.ComponentModel;

using Data;

using Data.Models;

namespace Client.TimetableDisplay

{

// Delegate function signature for the event indicating

// a tile has been clicked

public delegate void TileClickHandler(TimetableTile Tile);

// Shows a table of Rooms and Timeslots where each cell represents a Booking

public partial class TimetableDisplay

: UserControl, INotifyPropertyChanged

{

// The internal array of Tiles used

protected TimetableTile[,] Tiles { get; set; }

// The dimensions of a single tile

protected float TileWidth = 100;

protected float TileHeight = 100;

// The dimensions of the top/left headings of the table

protected float LeftWidth = 75;

protected float TopHeight = 75;

// Brush used to colour the background of the headings

protected Brush MarginBrush = Brushes.LightGray;

// Event indicating a tile has been clicked

public event TileClickHandler TileClicked;

public TimetableDisplay()

{

InitializeComponent();

PropertyChanged = delegate { };

TileClicked = delegate { };

DataContext = this;

3 main tasks – add the row headers of the table, add the column headers of the table, and add the contents of the table. This initial section is generating the row/columns but not filling them in yet.

}

// Organises the table for a given user on a particular day

public void SetTimetable(User CurrentUser, DateTime Day)

{

// Get the current database state

DataSnapshot Frame = DataRepository.TakeSnapshot();

// Generate the correct number of rows

Container.RowDefinitions.Clear();

// Top header row

Container.RowDefinitions.Add(new RowDefinition() { Height = new GridLength(1, GridUnitType.Auto) });

// Actual rows of the table

for (int y = 0; y < Frame.Rooms.Count; y++)

Container.RowDefinitions.Add(new RowDefinition() { Height = new GridLength(TileHeight) });

// Generate the columns

Container.ColumnDefinitions.Clear();

// Left-hand side-heading column

Container.ColumnDefinitions.Add(new ColumnDefinition() { Width = new GridLength(LeftWidth) });

// Actual columns in the table

for (int x = 0; x < Frame.Periods.Count; x++)

Container.ColumnDefinitions.Add(new ColumnDefinition() { Width = new GridLength(TileWidth) });

// Add the left-hand bar, contains room names and a tooltip

Container.Children.Clear();

for (int y = 0; y < Frame.Rooms.Count; y++)

{

// Create a textblock displaying the room name

Now we start adding actual content, starting with the left-hand bar containing the room information.

TextBlock Child = new TextBlock();

Child.Text = Frame.Rooms[y].RoomName;

// Set standard font, margin, wrapping style etc

Child.FontSize = 16;

Child.Margin = new Thickness(0, 0, 5, 0);

Child.TextWrapping = TextWrapping.Wrap;

Child.VerticalAlignment = VerticalAlignment.Center;

Child.HorizontalAlignment = HorizontalAlignment.Right;

// Create the alignment control for nice layout

Border LeftTile = new Border();

// Set tooltip to useful information

LeftTile.ToolTip = "Standard Seats: " + Frame.Rooms[y].StandardSeats + (Frame.Rooms[y].SpecialSeats == 0 ? "" : "\n" + Frame.Rooms[y].SpecialSeatType + ": " + Frame.Rooms[y].SpecialSeats);

// Set the UI child of this control to be the texblock above

LeftTile.Child = Child;

// Background colour

LeftTile.Background = MarginBrush;

// Positioning on the layout grid

LeftTile.SetValue(Grid.RowProperty, y + 1);

LeftTile.SetValue(Grid.ColumnProperty, 0);

Next section is the upper heading, with the timeslot names and time ranges.

// Add the controls to the grid

Container.Children.Add(LeftTile);

}

// Add the top heading, contains timeslot name and time interval

for (int x = -1; x < Frame.Periods.Count; x++)

{

// Use a grid for ease of layout

Grid TopTile = new Grid();

TopTile.RowDefinitions.Add(new RowDefinition() { Height = new GridLength(1, GridUnitType.Auto) });

TopTile.RowDefinitions.Add(new RowDefinition() { Height = new GridLength(1, GridUnitType.Auto) });

// Set background and child alignments

TopTile.Background = MarginBrush;

TopTile.VerticalAlignment = VerticalAlignment.Bottom;

TopTile.HorizontalAlignment = HorizontalAlignment.Left;

// If we're not filling out the top-left corner cell, store the timeslot name

string Text = "";

if (x >= 0 && !string.IsNullOrWhiteSpace(Frame.Periods[x].Name))

Text = Frame.Periods[x].Name;

// First textblock - timeslot name

TopTile.Children.Add(new TextBlock() { Text = Text, FontSize = 16, Margin = new Thickness(2, 2, 2, 0), TextWrapping = TextWrapping.Wrap, Width = TileWidth });

// Second textblock - timeslot duration

TopTile.Children.Add(new TextBlock() { Text = x >= 0 ? Frame.Periods[x].TimeRange : "", FontSize = 16, Margin = new Thickness(2, 0, 2, 10), TextWrapping = TextWrapping.Wrap, Width = TileWidth });

// Set the position of each textblock within the local grid

for (int y = 0; y < TopTile.Children.Count; y++)

TopTile.Children[y].SetValue(Grid.RowProperty, y);

// Algin the local grid within the table's grid

TopTile.SetValue(Grid.RowProperty, 0);

TopTile.SetValue(Grid.ColumnProperty, x + 1);

Container.Children.Add(TopTile);

}

// Add the main content

// Find bookings on this day

List<Booking> RelevantBookings = Frame.Bookings.Where(b => b.MatchesDay(Day)).ToList();

// Initialise the internal array of tiles

Tiles = new TimetableTile[Frame.Rooms.Count, Frame.Periods.Count];

for (int y = 0; y < Frame.Rooms.Count; y++)

{

for (int x = 0; x < Frame.Periods.Count; x++)

{

// Get the booking (or null) for this combination of room and timeslot

Booking Current = RelevantBookings.Where(b => b.TimeSlot == Frame.Periods[x] && b.Rooms.Contains(Frame.Rooms[y])).SingleOrDefault();

// Create the timetable tile

Tiles[y, x] = new TimetableTile(Current, Frame.Periods[x], Frame.Rooms[y], CurrentUser);

// Layout

This is all adding the main content – the actual Tiles.

Container.Children.Add(Tiles[y, x]);

Tiles[y, x].SetValue(Grid.RowProperty, y + 1);

Tiles[y, x].SetValue(Grid.ColumnProperty, x + 1);

// Hook up the tile clicked handler

Tiles[y, x].MouseLeftButtonDown += (o, e) => TileClicked((TimetableTile)o);

}

}

}

// Standard INotifyPropertyChanged interface implementation, for UI bindings

protected void OnPropertyChanged(string PropertyName)

{

if (PropertyChanged != null)

PropertyChanged(this, new PropertyChangedEventArgs(PropertyName));

}

public event PropertyChangedEventHandler PropertyChanged;

Classes inheriting INotifyPropertyChanged must implement this event – using this interface is really useful in the UI, as it allows text to be bound to values of properties in the code-behind responsively.

}

}

Client.EditWindows.EditWindow

The *EditWindow* generic base class is used to reduce the amount of code needed to describe the other Edit windows. It provides two main features, the first being an implementation of *INotifyPropertyChanged*, which is used in all the Edit windows for UI responsiveness. The second is to provide the abstract *GetItem* function, which returns the new item made by the Edit window for adding/removing from the database.

using System;

using System.Windows;

using System.ComponentModel;

using Data.Models;

namespace Client.EditWindows

{

// Provides a generic base class for all the edit windows, reduces their code

public abstract class EditWindow<T>

: Window, INotifyPropertyChanged where T : DataModel

{

public EditWindow()

{

PropertyChanged = delegate { };

}

// Gets the item that's been constructed by the EditWindow

public abstract T GetItem();

public void OnPropertyChanged(string PropertyName)

{

if (PropertyChanged != null)

PropertyChanged(this, new PropertyChangedEventArgs(PropertyName));

}

public event PropertyChangedEventHandler PropertyChanged;

}

This class is purely logical – despite inheriting from *Window*, it provides no design code. This is left to the subclasses, as is the remainder of the logic needed for their specific windows.

}

Client.EditWindows.StudentSelector (Design)

<UserControl x:Class="Client.EditWindows.StudentSelector"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:local="clr-namespace:Client.EditWindows"

mc:Ignorable="d"

d:DesignHeight="300" d:DesignWidth="400"

DataContext="{Binding RelativeSource={RelativeSource Self}}">

<Grid VerticalAlignment="Stretch">

<Grid.RowDefinitions>

<RowDefinition Height="Auto"/>

<RowDefinition Height="\*"/>

</Grid.RowDefinitions>

<Grid Grid.Row="0" Margin="5" VerticalAlignment="Top">

<Grid.ColumnDefinitions>

<ColumnDefinition Width="Auto"/>

<ColumnDefinition Width="Auto"/>

<ColumnDefinition Width="Auto"/>

<ColumnDefinition Width="Auto"/>

</Grid.ColumnDefinitions>

<TextBlock Text="Filter:" Margin="0,7,5,0" Grid.Column="0"/>

<TextBox Name="Text\_StudentFilter" Width="125" Margin="0,5,5,0" Grid.Column="1" TextChanged="Text\_StudentFilter\_TextChanged"/>

<ComboBox Name="Combo\_FilterType" ItemsSource="{Binding Path=FilterValues, Mode=OneTime}" SelectedIndex="0" Width="100" Margin="0,5,5,0" Grid.Column="2" SelectionChanged="Combo\_FilterType\_SelectionChanged"/>

<ComboBox Name="Combo\_Classes" ItemsSource="{Binding ClassNames, Mode=OneWay}" SelectedIndex="0" Width="100" Margin="0,5,5,0" Grid.Column="3" SelectionChanged="Combo\_Classes\_SelectionChanged"/>

</Grid>

<ListView Name="List\_Students" ItemsSource="{Binding FilteredStudents}" VerticalAlignment="Stretch" Margin="5" Grid.Row="1">

<ListView.View>

<GridView>

<GridView.Columns>

<GridViewColumn Width="30">

<GridViewColumn.CellTemplate>

<DataTemplate>

<CheckBox IsChecked="{Binding Checked}"/>

</DataTemplate>

</GridViewColumn.CellTemplate>

</GridViewColumn>

<GridViewColumn Header="First Name" Width="125" DisplayMemberBinding="{Binding Value.FirstName}"/>

<GridViewColumn Header="Last Name" Width="125" DisplayMemberBinding="{Binding Value.LastName}"/>

<GridViewColumn Header="Form" Width="40" DisplayMemberBinding="{Binding Value.Form}"/>

 <GridViewColumn Header="Year" Width="40" DisplayMemberBinding="{Binding Value.Year}"/>

</GridView.Columns>

</GridView>

</ListView.View>

Example of the *StudentSelector* control being used on the UI.

</ListView>

</Grid>

</UserControl>

Client.EditWindows.StudentSelector (Code-behind)

The *StudentSelector* control is used to select students from the *Student* objects in the database. It allows filtering of students by various criteria, as well as by their classes.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Windows.Controls;

using System.Collections.ObjectModel;

using System.ComponentModel;

using Data.Models;

This dictionary below is used to map a filter name to its function. This provides a very concise, neat way of writing filters.

namespace Client.EditWindows

{

// Used to select students from a list

public partial class StudentSelector

: UserControl, INotifyPropertyChanged

{

// Dictionary containing the name of the filter mapped to the actual filter function.

// Each function takes a Checkable<Student>, the filter text, and returns a bool indicating

// whether the Student should be included in the displayed list

// Functions are specified using lambda functions for simplicity

public readonly Dictionary<string, Func<Checkable<Student>, string, bool>> Filters = new Dictionary<string, Func<Checkable<Student>, string, bool>>()

{

{ "No Filter", (s, f) => true },

{ "Checked", (s, f) => s.Checked },

{ "Unchecked", (s, f) => !s.Checked },

{ "First Name", (s, f) => s.Value.FirstName.ToLower().Contains(f.ToLower()) },

{ "Last Name", (s, f) => s.Value.LastName.ToLower().Contains(f.ToLower()) },

{ "Form", (s, f) => s.Value.Form.ToLower().Contains(f.ToLower()) },

{ "Year", (s, f) => Convert.ToString(s.Value.Year).ToLower().Contains(f.ToLower()) }

};

// The Keys from the dictionary, used for displaying in UI

public List<string> FilterValues { get { return Filters.Keys.ToList(); } }

// The internal list of all students

public List<Checkable<Student>> Students { get; set; }

// The list of students displayed after filtering

public ObservableCollection<Checkable<Student>> FilteredStudents { get; set; }

// The list of students actually selected

public List<Student> SelectedStudents { get { return Students.Where(s => s.Checked).Select(s => s.Value).ToList(); } }

// The list of classnames, and the respective Class objects

public List<string> ClassNames { get; set; }

public List<Class> Classes { get; set; }

public StudentSelector()

{

using (DataRepository Repo = new DataRepository())

{

// Get the classes

Classes = Repo.Classes.ToList();

// Get the names of the classes

ClassNames = Repo.Classes.Select(c => c.ClassName).ToList();

// Insert a "dummy" class which ignores the selection

ClassNames.Insert(0, "All students");

// Get the students

Students = Repo.Users.OfType<Student>().Select(s => new Checkable<Student>(s)).ToList();

// Initialise the filtered student list

FilteredStudents = new ObservableCollection<Checkable<Student>>(Students);

}

InitializeComponent();

}

// Call when the list of filtered students needs updating

public void UpdateFilter()

{

// Prevents calls before the UI is up and running

if (!IsInitialized)

return;

// Check we're running on the UI thread

if (!Dispatcher.CheckAccess())

Dispatcher.Invoke((Action)UpdateFilter);

else

{

// Grab the filter text from the UI

string Filter = Text\_StudentFilter.Text;

// Grab the type of filter from the UI

string FilterType = FilterValues[Combo\_FilterType.SelectedIndex];

// Filter by the filter text

IEnumerable<Checkable<Student>> Filtered = Students.Where(s => Filters[FilterType](s, Filter));

// If we're filtering by class, run the secondary filetr

if (Combo\_Classes.SelectedIndex != 0)

{

Class Class = Classes.Single(c => c.ClassName == (string)Combo\_Classes.SelectedItem);

Filtered = Filtered.Where(s => Class.Students.Contains(s.Value));

}

// Update the list of filtered students that the UI sees

FilteredStudents.Clear();

foreach (Checkable<Student> s in Filtered)

FilteredStudents.Add(s);

}

}

// These next 3 event handlers just update the filter if any relevant control was changed

private void Combo\_FilterType\_SelectionChanged(object sender, SelectionChangedEventArgs e)

{

UpdateFilter();

}

private void Text\_StudentFilter\_TextChanged(object sender, TextChangedEventArgs e)

{

UpdateFilter();

}

private void Combo\_Classes\_SelectionChanged(object sender, SelectionChangedEventArgs e)

{

UpdateFilter();

}

public void OnPropertyChanged(string PropertyName)

{

if (PropertyChanged != null)

PropertyChanged(this, new PropertyChangedEventArgs(PropertyName));

}

public event PropertyChangedEventHandler PropertyChanged;

}

}

Client.EditWindows.EditBooking (Design)

<local:EditWindow x:Class="Client.EditWindows.EditBooking"

x:TypeArguments="data:Booking"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:converters="clr-namespace:Client.Converters"

xmlns:local="clr-namespace:Client.EditWindows"

xmlns:data="clr-namespace:Data.Models;assembly=Data"

mc:Ignorable="d"

Title="Edit Booking" Height="800" Width="430"

SizeToContent="Height"

DataContext="{Binding RelativeSource={RelativeSource Self}}">

<Window.Resources>

<converters:InverseNullableBooleanConverter x:Key="InverseNullableBooleanConverter"/>

<converters:BooleanToVisibilityConverter x:Key="BooleanToVisibilityConverter"/>

<converters:BookingTypeToStringConverter x:Key="BookingTypeToStringConverter"/>

</Window.Resources>

<Grid Name="Container">

<Grid.RowDefinitions>

Lots of rows for the different fields – between each row is a small gap for the grid-splitter control to allow resizing.

<RowDefinition Height="Auto" MinHeight="129"/>

<RowDefinition Height="5"/>

<RowDefinition Height="\*" MinHeight="200"/>

<RowDefinition Height="5"/>

<RowDefinition Height="\*" MinHeight="200"/>

<RowDefinition Height="5"/>

<RowDefinition Height="Auto" MinHeight="35"/>

First groupbox holds 4 rows with the Subject, Recurrence, Teacher, and Period fields.

</Grid.RowDefinitions>

<GroupBox Header="Info" Margin="5" Grid.Row="0">

<Grid>

<Grid.RowDefinitions>

<RowDefinition Height="Auto" Name="Row\_Subject"/>

<RowDefinition Height="Auto" Name="Row\_Recurrence"/>

<RowDefinition Height="Auto" Name="Row\_Teacher"/>

<RowDefinition Height="Auto" Name="Row\_Period"/>

</Grid.RowDefinitions>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="Auto"/>

<ColumnDefinition Width="\*"/>

</Grid.ColumnDefinitions>

<TextBlock Text="Subject:" Margin="5,7,5,5" Grid.Row="0" Grid.Column="0"/>

<ComboBox Name="Combo\_Subject" ItemsSource="{Binding Subjects}" SelectedItem="{Binding SelectedSubject, UpdateSourceTrigger=PropertyChanged}" Width="150" Margin="5" HorizontalAlignment="Left" Grid.Row="0" Grid.Column="1"/>

<TextBlock Text="Recurrence: " Margin="5,7,5,5" Grid.Row="1" Grid.Column="0"/>

<ComboBox Name="Combo\_BookingType" ItemsSource="{Binding BookingTypes}" SelectedItem="{Binding SelectedBookingType, UpdateSourceTrigger=PropertyChanged, Converter={StaticResource BookingTypeToStringConverter}}" Width="150" Margin="5" HorizontalAlignment="Left" Grid.Row="1" Grid.Column="1"/>

<TextBlock Text="Teacher:" Margin="5,7,5,5" Grid.Row="2" Grid.Column="0"/>

<ComboBox Name="Combo\_Teacher" ItemsSource="{Binding Teachers}" SelectedItem="{Binding SelectedTeacher, UpdateSourceTrigger=PropertyChanged}" Width="150" Margin="5" HorizontalAlignment="Left" Grid.Row="2" Grid.Column="1"/>

<TextBlock Text="Period: " Margin="5,8,0,0" Grid.Column="0" Grid.Row="3"/>

<ComboBox Name="Time\_Existing\_Combo\_Time" ItemsSource="{Binding Periods}" SelectedItem="{Binding SelectedTimeslot}" Margin="5,5,5,0" Width="150" HorizontalAlignment="Left" Grid.Column="1" Grid.Row="3"/>

</Grid>

</GroupBox>

<GridSplitter Grid.Row="1" Height="5" HorizontalAlignment="Stretch"/>

<GroupBox Header="Rooms" Margin="5" Grid.Row="2">

<ListView Name="List\_Rooms" ItemsSource="{Binding Rooms}" VerticalAlignment="Stretch" Margin="5" Grid.Column="0" Grid.ColumnSpan="2">

Second groupbox holds the list of Rooms that can be selected, using Bindings to display the right info.

<ListView.View>

<GridView>

<GridView.Columns>

<GridViewColumn Width="30">

<GridViewColumn.CellTemplate>

<DataTemplate>

<CheckBox IsChecked="{Binding Checked}"/>

</DataTemplate>

</GridViewColumn.CellTemplate>

</GridViewColumn>

<GridViewColumn DisplayMemberBinding="{Binding Value.RoomName}" Width="80" Header="Room Name"/>

<GridViewColumn DisplayMemberBinding="{Binding Value.StandardSeats}" Width="90" Header="Standard Seats"/>

<GridViewColumn DisplayMemberBinding="{Binding Value.SpecialSeatType}" Width="100" Header="Special seat type"/>

<GridViewColumn DisplayMemberBinding="{Binding Value.SpecialSeats}" Width="80" Header="Special seats"/>

</GridView.Columns>

Third groupbox simply holds a *StudentSelector* control to allow selection of the attending students.

</GridView>

</ListView.View>

</ListView>

</GroupBox>

<GridSplitter Grid.Row="3" Height="5" HorizontalAlignment="Stretch"/>

<GroupBox Header="Students" Margin="5" Grid.Row="4" Height="270">

<local:StudentSelector x:Name="StudentSelector"/>

</GroupBox>

<GridSplitter Grid.Row="5" Height="5" HorizontalAlignment="Stretch"/>

<Button Name="Button\_Delete" Content="Delete" Click="Button\_Delete\_Click" Margin="5,5,60,5" Width="50" Height="25" HorizontalAlignment="Right" VerticalAlignment="Bottom" Grid.Row="8"/>

<Button Name="Button\_Submit" Content="Submit" Click="Button\_Submit\_Click" Margin ="5" Width="50" Height="25" HorizontalAlignment="Right" VerticalAlignment="Bottom" Grid.Row="8"/>

</Grid>

Final row is just delete/submit buttons. To cancel, the window is closed using the default top-right close button.

</local:EditWindow>

Example of an EditBooking window being used in the UI.

Client.EditWindows.EditBooking (Code-behind)

The *EditBooking* window is perhaps the most complicated window in the system. It allows for selecting rooms, students, timeslots, and a few other settings to represent a Booking in the system. With the amount of information handled by this single window, a lot of WPF Binding to the UI is used.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Windows;

using System.Collections.ObjectModel;

using Data.Models;

namespace Client.EditWindows

{

public partial class EditBooking

: EditWindow<Booking>

{

// The user that's currently logged on

public User CurrentUser { get; private set; }

// All the Rooms that can be booked

public ObservableCollection<Checkable<Room>> Rooms { get; set; }

// Utility property to collect the selected rooms

public List<Checkable<Room>> SelectedRooms { get { return Rooms.Where(r => r.Checked).ToList(); } }

These next few properties all follow the same pattern – allocate the backing variable, then the public property that the UI binds to.

// Periods that can be booked in

public ObservableCollection<TimeSlot> Periods { get; set; }

// The single timeslot that's been booked

private TimeSlot \_SelectedTimeslot;

public TimeSlot SelectedTimeslot

{

get { return \_SelectedTimeslot; }

set { \_SelectedTimeslot = value; OnPropertyChanged("SelectedTimeslot"); }

}

// Types of booking (protected array with a public accessor property)

protected readonly string[] \_BookingTypes = Enum.GetNames(typeof(BookingType));

public string[] BookingTypes { get { return \_BookingTypes; } }

// The currently selected recurrence mode

protected BookingType \_SelectedBookingType = BookingType.Single;

public BookingType SelectedBookingType

{

get { return \_SelectedBookingType; }

set { \_SelectedBookingType = value; OnPropertyChanged("SelectedBookingType"); }

}

// The subjects that can be selected

public List<Subject> Subjects { get; private set; }

// The subject that's been selected

private Subject \_SelectedSubject;

public Subject SelectedSubject

{

get { return \_SelectedSubject; }

set { \_SelectedSubject = value; OnPropertyChanged("SelectedSubject"); }

}

// The Teachers that can be selected

public List<Teacher> Teachers { get; private set; }

// The teacher that's currently selected

private Teacher \_SelectedTeacher;

public Teacher SelectedTeacher

{

get { return \_SelectedTeacher; }

set { \_SelectedTeacher = value; OnPropertyChanged("SelectedTeacher"); }

}

// The ID of the booking returned by this window. Set to either 0 to

// represent a new booking or the ID of the booking being edited.

protected int BookingId { get; set; }

// Whether or not a deletion ws requested by the user

public bool DeleteBooking { get; private set; }

// The current date displayed on the Timetable

public DateTime CurrentDate { get; set; }

// For making a new booking given the time and room

public EditBooking(User CurrentUser, bool NewBooking, TimeSlot StartTime, Room StartRoom)

: this(CurrentUser, NewBooking, 0, new List<Room>(), StartRoom, StartTime, new List<Student>(), null, null, BookingType.Single)

{

}

// For editing an existing booking given the booking itself

public EditBooking(User CurrentUser, bool NewBooking, Booking Booking)

: this(CurrentUser, NewBooking, Booking.Id, Booking.Rooms, null, Booking.TimeSlot, Booking.Students, Booking.Subject, Booking.Teacher, Booking.BookingType)

{

}

// Derived constructor to handle the common tasks of the above two constructors

public EditBooking(User CurrentUser, bool NewBooking, int Id, List<Room> SelectedRooms, Room StartRoom, TimeSlot TimeSlot, List<Student> SelectedStudents, Subject Subject, Teacher Teacher, BookingType BookingType) // For editing an existing booking

There are 3 constructors defined – the first two take fairly simple parameters, the third takes quite a complex assortment. The 3rd constructor does all the work for the first two, they simply break down their arguments and call the 3rd. This approach reduces repeated code.

{

// Store necessary data

this.CurrentUser = CurrentUser;

SelectedBookingType = BookingType;

BookingId = Id;

DeleteBooking = false;

SelectedTimeslot = TimeSlot;

using (DataRepository Repo = new DataRepository())

{

// Store the rooms

Rooms = new ObservableCollection<Checkable<Room>>(Repo.Rooms.ToList().Select(r1 => new Checkable<Room>(r1, (StartRoom != null && r1.Id == StartRoom.Id) || SelectedRooms.Any(r2 => r1.Id == r2.Id))));

// Store the timeslots

Periods = new ObservableCollection<TimeSlot>(Repo.Periods);

// Store the subjects

Subjects = Repo.Subjects.ToList();

if (Subject != null) // Initialise if needed

SelectedSubject = Subject;

// Store teacher list

Teachers = Repo.Users.OfType<Teacher>().ToList();

if (Teacher != null) // Initialise if needed

SelectedTeacher = Teacher;

else if (CurrentUser is Teacher) // Else give default value

SelectedTeacher = (Teacher)CurrentUser;

}

// Initialise the UI

InitializeComponent();

// Perform initial selection of students

StudentSelector.Students.Where(s => SelectedStudents.Contains(s.Value)).ToList().ForEach(s => s.Checked = true);

if (!CurrentUser.IsAdmin) // Only let the teacher be changed if the user is an admin

Combo\_Teacher.IsEnabled = false;

if (NewBooking) // Must be editing a booking to delete it

Button\_Delete.IsEnabled = false;

}

// Return the Booking object created by this window

public override Booking GetItem()

{

// Validate

if (SelectedTimeslot == null || SelectedRooms == null || SelectedRooms.Count == 0 || SelectedSubject == null || StudentSelector.SelectedStudents == null || SelectedTeacher == null)

return null;

return new Booking(SelectedTimeslot, SelectedRooms.Select(c => c.Value).ToList(), SelectedSubject,

StudentSelector.SelectedStudents.ToList(), SelectedTeacher, SelectedBookingType)

{ Id = BookingId, Date = CurrentDate };

Overrides the *EditWindow* function *GetItem* to return the *Booking* object that can be created from the information entered in this window.

}

// When told to submit the changes

private void Button\_Submit\_Click(object sender, RoutedEventArgs e)

{

// Perform validation

string Error = string.Empty;

if (SelectedRooms.Count == 0)

Error = "You must select at least one room.";

else if (SelectedSubject == null)

When the user tries to submit the data, we validate that it’s all valid, and then call G*etItem* to retrieve the newly made item and check for conflicts, before finally either showing an error message or closing the window.

Error = "You must select a subject.";

else if (SelectedTeacher == null)

Error = "You must select a teacher";

else

{

// Check for conflicts

Booking b = GetItem();

if (b == null)

Error = "Invalid booking.";

else

{

using (DataRepository Repo = new DataRepository())

if (b.Conflicts(Repo.Bookings.Cast<DataModel>().ToList()))

Error = "Booking conflicts with another booking.";

}

}

// Display error or close the window

if (!string.IsNullOrWhiteSpace(Error))

MessageBox.Show(Error, "Error");

else

{

DialogResult = true;

Close();

}

}

// When told to delete the item

private void Button\_Delete\_Click(object sender, RoutedEventArgs e)

{

// Confirm then signal deletion

MessageBoxResult r = MessageBox.Show("Are you sure you want to delete this booking?", "Delete Booking", MessageBoxButton.YesNo);

When trying to delete data, do a quick double-check with the user then close the window flagging the booking for deletion.

if (r == MessageBoxResult.Yes)

{

DeleteBooking = true;

DialogResult = true;

Close();

}

}

}

}

Client.EditWindows.EditClass (Design)

<local:EditWindow x:Class="Client.EditWindows.EditClass"

x:TypeArguments="data:Class"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:Client.EditWindows"

xmlns:data="clr-namespace:Data.Models;assembly=Data"

mc:Ignorable="d"

Title="Edit Class" Height="300" Width="390"

DataContext="{Binding RelativeSource={RelativeSource Self}}">

<Grid>

<Grid.RowDefinitions>

<RowDefinition Height="Auto"/>

<RowDefinition Height="Auto"/>

<RowDefinition Height="\*"/>

<RowDefinition Height="Auto"/>

</Grid.RowDefinitions>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="Auto"/>

<ColumnDefinition Width="\*"/>

</Grid.ColumnDefinitions>

<TextBlock Text="Class Name:" Margin="5" Grid.Row="0"/>

<TextBox Text="{Binding ClassName, UpdateSourceTrigger=PropertyChanged}" Margin="3" Width="150" HorizontalAlignment="Left" Grid.Row="0" Grid.Column="1" TabIndex="0"/>

<TextBlock Text="Teacher:" Margin="5" Grid.Row="1"/>

<ComboBox Name="Combo\_Teacher" ItemsSource="{Binding Teachers}" SelectedItem="{Binding Teacher, UpdateSourceTrigger=PropertyChanged}" Width="150" HorizontalAlignment="Left" Margin="3" Grid.Row="1" Grid.Column="1"/>

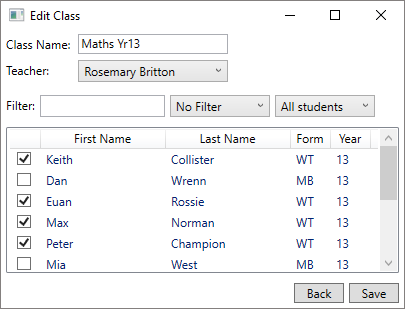
<local:StudentSelector x:Name="Students" Grid.Row="2" Grid.ColumnSpan="2"/>

<Button Name="Button\_Back" Content="Back" Width="50" Height="20" Margin="5,5,60,5" HorizontalAlignment="Right" VerticalAlignment="Bottom" Grid.Row="3" Grid.Column="1" Click="Button\_Back\_Click"/>

<Button Name="Button\_Save" Content="Save" Width="50" Height="20" Margin="5" HorizontalAlignment="Right" VerticalAlignment="Bottom" Grid.Row="3" Grid.Column="1" Click="Button\_Save\_Click"/>

</Grid>

</local:EditWindow>



Example of the *EditClass* window being used.

Client.EditWindows.EditClass (Code-behind)

*EditClass* is simpler than EditBooking (the same applies for the rest of the Edit Windows). There are two simple input fields and a multi-select field for the Students in the class.   
All the subsequent windows in this namespace follow the same pattern.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Windows;

using Data.Models;

namespace Client.EditWindows

{

// Window for editing a a Class object

public partial class EditClass

: EditWindow<Class>

{

// Name of the class

protected string \_ClassName;

public string ClassName

{

get { return \_ClassName; }

set { \_ClassName = value; OnPropertyChanged("ClassName"); }

}

// Name of the teacher teaching the class

protected string \_Teacher;

public string Teacher

{

get { return \_Teacher; }

set { \_Teacher = value; OnPropertyChanged("Teacher"); }

}

// Internal array of all teacher names

public string[] Teachers { get; set; }

// Parallel array to the above holding the IDs of the teachers

public int[] TeacherIds { get; set; }

// ID of the Class being edited/created

public int ClassId { get; set; }

public EditClass(Class Existing)

{

// Store the values that can be selected

using (DataRepository Repo = new DataRepository())

{

IEnumerable<Teacher> ts = Repo.Users.OfType<Teacher>();

Teachers = ts.Select(t => t.InformalName).ToArray();

TeacherIds = ts.Select(t => t.Id).ToArray();

}

InitializeComponent();

// If new Class, enter default values

if (Existing == null)

{

ClassName = string.Empty;

Teacher = string.Empty;

ClassId = 0;

}

else // Existing class, load the values from it

{

ClassName = Existing.ClassName;

Teacher = Existing.Owner.InformalName;

ClassId = Existing.Id;

// Select already involved students in advance

Students.Students.Where(s => Existing.Students.Contains(s.Value)).ToList().ForEach(s => s.Checked = true);

}

}

public override Class GetItem()

{

Class New = new Class();

try

{

// Fill in values

New.Id = ClassId;

New.ClassName = ClassName;

New.Students = Students.SelectedStudents;

using (DataRepository Repo = new DataRepository())

{

// Find the selected teacher

New.Owner = Repo.Users.OfType<Teacher>().Single(t => t.Id == TeacherIds[Combo\_Teacher.SelectedIndex]);

}

}

catch

{

return null;

}

return New;

}

private void Button\_Back\_Click(object sender, RoutedEventArgs e)

{

// Set the "cancel" flag and close

DialogResult = false;

Close();

}

private void Button\_Save\_Click(object sender, RoutedEventArgs e)

{

string Error = null;

// Perform validation

using (DataRepository Repo = new DataRepository())

{

if (string.IsNullOrWhiteSpace(ClassName))

Error = "You must enter a class name.";

else if (string.IsNullOrWhiteSpace(Teacher))

Error = "You must enter a Teacher.";

else if (Repo.Classes.Any(c => c.Id != ClassId && c.ClassName == ClassName))

Error = "Another class with that name already exists.";

}

// Error or close window

if (Error != null)

MessageBox.Show(Error, "Error", MessageBoxButton.OK);

else

{

DialogResult = true;

Close();

}

}

}

}

Client.EditWindows.EditDepartment (Design)

<local:EditWindow x:Class="Client.EditWindows.EditDepartment"

x:TypeArguments="data:Department"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:Client.EditWindows"

xmlns:data="clr-namespace:Data.Models;assembly=Data"

mc:Ignorable="d"

Title="Edit Department" Height="500" Width="500"

DataContext="{Binding RelativeSource={RelativeSource Self}}">

<Grid>

<Grid.RowDefinitions>

<RowDefinition Height="Auto"/>

<RowDefinition Height="\*"/>

<RowDefinition Height="\*"/>

<RowDefinition Height="Auto"/>

</Grid.RowDefinitions>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="Auto"/>

<ColumnDefinition Width="\*"/>

</Grid.ColumnDefinitions>

<TextBlock Text="Name:" Margin="5" Grid.Row="0"/>

<TextBox Text="{Binding DepartmentName, UpdateSourceTrigger=PropertyChanged}" Margin="3" Grid.Row="0" Grid.Column="1" TabIndex="0"/>

<GroupBox Header="Teachers" Margin="5" Grid.Row="1" Grid.ColumnSpan="2" >

<ListView Name="List\_Teachers" ItemsSource="{Binding Teachers}" Margin="5">

<ListView.View>

<GridView>

<GridView.Columns>

<GridViewColumn Width="30">

<GridViewColumn.CellTemplate>

<DataTemplate>

<CheckBox IsChecked="{Binding Checked}"/>

</DataTemplate>

</GridViewColumn.CellTemplate>

</GridViewColumn>

<GridViewColumn DisplayMemberBinding="{Binding Value.FirstName}" Width="80" Header="First Name"/>

<GridViewColumn DisplayMemberBinding="{Binding Value.LastName}" Width="110" Header="Last Name"/>

<GridViewColumn DisplayMemberBinding="{Binding Value.Title}" Width="50" Header="Title"/>

<GridViewColumn DisplayMemberBinding="{Binding Value.LogonName}" Width="80" Header="Logon Name"/>

<GridViewColumn DisplayMemberBinding="{Binding Value.Access}" Width="80" Header="Access"/>

<GridViewColumn DisplayMemberBinding="{Binding Value.Email}" Width="150" Header="Email"/>

</GridView.Columns>

</GridView>

</ListView.View>

</ListView>

</GroupBox>

<GroupBox Header="Rooms" Margin="5" Grid.Row="2" Grid.ColumnSpan="2">

<ListView Name="List\_Rooms" SelectionMode="Single" ItemsSource="{Binding Rooms}" Margin="5">

<ListView.View>

<GridView>

<GridView.Columns>

<GridViewColumn Width="30">

<GridViewColumn.CellTemplate>

<DataTemplate>

<CheckBox IsChecked="{Binding Checked}"/>

</DataTemplate>

</GridViewColumn.CellTemplate>

</GridViewColumn>

<GridViewColumn DisplayMemberBinding="{Binding Value.RoomName}" Width="80" Header="Room Name"/>

<GridViewColumn DisplayMemberBinding="{Binding Value.StandardSeats}" Width="110" Header="Standard Seats"/>

<GridViewColumn DisplayMemberBinding="{Binding Value.SpecialSeatType}" Width="100" Header="Special seat type"/>

<GridViewColumn DisplayMemberBinding="{Binding Value.SpecialSeats}" Width="80" Header="Special seats"/>

</GridView.Columns>

</GridView>

</ListView.View>

</ListView>

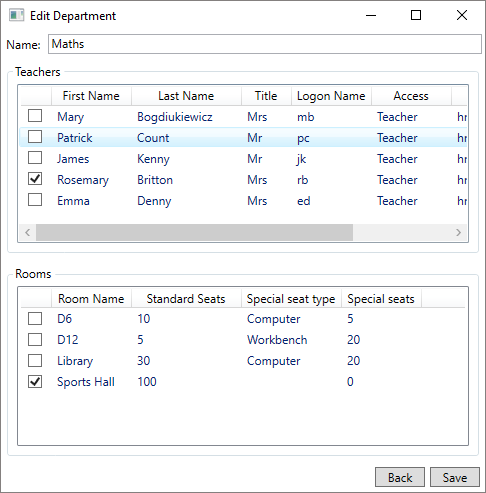
</GroupBox>

<Button Name="Button\_Back" Content="Back" Width="50" Height="20" Margin="5,5,60,5" HorizontalAlignment="Right" VerticalAlignment="Bottom" Grid.Row="3" Grid.Column="1" Click="Button\_Back\_Click"/>

<Button Name="Button\_Save" Content="Save" Width="50" Height="20" Margin="5" HorizontalAlignment="Right" VerticalAlignment="Bottom" Grid.Row="3" Grid.Column="1" Click="Button\_Save\_Click"/>

</Grid>

</local:EditWindow>



Example of an EditDepartment window being used in the UI.

Client.EditWindows.EditDepartment (Code-behind)

using System;

using System.Linq;

using System.Windows;

using System.Collections.ObjectModel;

using Data.Models;

namespace Client.EditWindows

{

// Window to edit a Department object

public partial class EditDepartment

: EditWindow<Department>

{

// Name of the department

protected string \_DepartmentName;

public string DepartmentName

{

get { return \_DepartmentName; }

set { \_DepartmentName = value; OnPropertyChanged("DepartmentName"); }

}

// List of all teachers that can be selected

protected ObservableCollection<Checkable<Teacher>> \_Teachers;

public ObservableCollection<Checkable<Teacher>> Teachers

{

get { return \_Teachers; }

set { \_Teachers = value; OnPropertyChanged("Teachers"); }

}

// List of all rooms that can be selected

protected ObservableCollection<Checkable<Room>> \_Rooms;

public ObservableCollection<Checkable<Room>> Rooms

{

get { return \_Rooms; }

set { \_Rooms = value; OnPropertyChanged("Rooms"); }

}

// Id of the object being edited

protected int DepartmentId { get; set; }

public EditDepartment(Department Existing)

{

// Store the lists of entities that can possibly be selected

using (DataRepository Repo = new DataRepository())

{

Teachers = new ObservableCollection<Checkable<Teacher>>(Repo.Users.OfType<Teacher>().Select(t => new Checkable<Teacher>(t, Existing != null && Existing.Teachers.Contains(t))));

Rooms = new ObservableCollection<Checkable<Room>>(Repo.Rooms.Select(r => new Checkable<Room>(r, Existing != null && Existing.Rooms.Contains(r))));

}

InitializeComponent();

// Initialse fields with empty/existing data

if (Existing == null)

{

DepartmentId = 0;

DepartmentName = string.Empty;

}

else

{

DepartmentId = Existing.Id;

DepartmentName = Existing.Name;

}

}

public override Department GetItem()

{

Department New = new Department();

try

{

// Fill out the properties on the object

New.Id = DepartmentId;

New.Name = DepartmentName;

New.Teachers = Teachers.Where(t => t.Checked).Select(t => t.Value).ToList();

New.Rooms = Rooms.Where(r => r.Checked).Select(r => r.Value).ToList();

}

catch

{

return null;

}

return New;

}

private void Button\_Back\_Click(object sender, RoutedEventArgs e)

{

DialogResult = false;

Close();

}

private void Button\_Save\_Click(object sender, RoutedEventArgs e)

{

string Error = null;

// Perform validation

using (DataRepository Repo = new DataRepository())

{

if (string.IsNullOrWhiteSpace(DepartmentName))

Error = "You must enter a department name.";

else if (Repo.Departments.Any(d => d.Id != DepartmentId && d.Name == DepartmentName))

Error = "Another department with that name already exists.";

}

// Show error or close the window with a signalled success

if (Error != null)

MessageBox.Show(Error, "Error", MessageBoxButton.OK);

else

{

DialogResult = true;

Close();

}

}

}

}

Client.EditWindows.EditPeriod (Design)

<local:EditWindow x:Class="Client.EditWindows.EditPeriod"

x:TypeArguments="data:TimeSlot"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:Client.EditWindows"

xmlns:data="clr-namespace:Data.Models;assembly=Data"

mc:Ignorable="d"

Title="Edit Period" Height="150" Width="300"

DataContext="{Binding RelativeSource={RelativeSource Self}}">

<Grid>

<Grid.RowDefinitions>

<RowDefinition Height="Auto"/>

<RowDefinition Height="Auto"/>

<RowDefinition Height="Auto"/>

<RowDefinition Height="\*"/>

</Grid.RowDefinitions>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="Auto"/>

<ColumnDefinition Width="\*"/>

</Grid.ColumnDefinitions>

<TextBlock Text="Name:" Margin="5" Grid.Row="0"/>

<TextBox Text="{Binding PeriodName, UpdateSourceTrigger=PropertyChanged}" Margin="3" Grid.Row="0" Grid.Column="1" TabIndex="0"/>

<TextBlock Text="Start:" Margin="5" Grid.Row="1"/>

<TextBox Text="{Binding Start, UpdateSourceTrigger=PropertyChanged}" Margin="3" Grid.Row="1" Grid.Column="1"/>

<TextBlock Text="End:" Margin="5" Grid.Row="2"/>

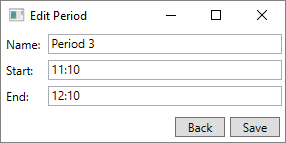
<TextBox Text="{Binding End, UpdateSourceTrigger=PropertyChanged}" Margin="3" Grid.Row="2" Grid.Column="1"/>

<Button Name="Button\_Back" Content="Back" Margin="5,5,60,5" Width="50" Height="20" HorizontalAlignment="Right" VerticalAlignment="Bottom" Grid.Row="3" Grid.Column="1" Click="Button\_Back\_Click"/>

<Button Name="Button\_Save" Content="Save" Margin="5,5,5,5" Width="50" Height="20" HorizontalAlignment="Right" VerticalAlignment="Bottom" Grid.Row="3" Grid.Column="1" Click="Button\_Save\_Click"/>

</Grid>

</local:EditWindow>



Example of an EditPeriod window being used in the UI.

Client.EditWindows.EditPeriod (Code-behind)

using System;

using System.Linq;

using System.Windows;

using Data.Models;

namespace Client.EditWindows

{

public partial class EditPeriod

: EditWindow<TimeSlot>

{

// Name of the period being edited

protected string \_PeriodName;

public string PeriodName

{

get { return \_PeriodName; }

set { \_PeriodName = value; OnPropertyChanged("PeriodName"); }

}

// String representation of the start time

protected string \_Start;

public string Start

{

get { return \_Start; }

set { \_Start = value; OnPropertyChanged("Start"); }

}

// String representation of the end time

protected string \_End;

public string End

{

get { return \_End; }

set { \_End = value; OnPropertyChanged("End"); }

}

// ID of the period being edited

public int PeriodId { get; set; }

public EditPeriod(TimeSlot Existing)

{

InitializeComponent();

// Initialise with new/existing data

if (Existing == null)

{

PeriodName = string.Empty;

Start = string.Empty;

End = string.Empty;

PeriodId = 0;

}

else

{

PeriodName = Existing.Name;

// Convert the TimeSpan objects into hh:mm format by padding and concatenating srings

Start = Convert.ToString(Existing.Start.Hours).PadLeft(2, '0') + ":" + Convert.ToString(Existing.Start.Minutes).PadLeft(2, '0');

End = Convert.ToString(Existing.End.Hours).PadLeft(2, '0') + ":" + Convert.ToString(Existing.End.Minutes).PadLeft(2, '0');

PeriodId = Existing.Id;

}

}

public override TimeSlot GetItem()

{

TimeSlot New = new TimeSlot();

try

{

// Fill out the fields, parsing data

New.Name = PeriodName;

New.Start = TimeSpan.Parse(Start);

New.End = TimeSpan.Parse(End);

New.Id = PeriodId;

}

catch

{

return null;

}

return New;

}

private void Button\_Back\_Click(object sender, RoutedEventArgs e)

{

DialogResult = false;

Close();

}

private void Button\_Save\_Click(object sender, RoutedEventArgs e)

{

string Error = null;

using (DataRepository Repo = new DataRepository())

{

// Validation

TimeSpan StartOut;

TimeSpan EndOut;

if (string.IsNullOrWhiteSpace(PeriodName))

Error = "You must enter a period name.";

else if (string.IsNullOrWhiteSpace(Start) || !TimeSpan.TryParse(Start, out StartOut) || !CompatibleTime(StartOut))

Error = "Invalid start time format. Must be in the format \"hh:mm\"";

else if (string.IsNullOrWhiteSpace(End) || !TimeSpan.TryParse(End, out EndOut) || !CompatibleTime(EndOut))

Error = "Invalid end time format. Must be in the format \"hh:mm\"";

else if (StartOut > EndOut)

Error = "Start time must be before End time.";

else if (Repo.Periods.Any(p => p.Name == PeriodName))

Error = "Another Period with that name already exists.";

}

// Error message or close window

if (!string.IsNullOrWhiteSpace(Error))

MessageBox.Show(Error, "Error", MessageBoxButton.OK);

else

{

DialogResult = true;

Close();

}

}

// Simple validation function that checks the timespan is represents a valid time

private bool CompatibleTime(TimeSpan t)

{

return t.Hours < 24 && t.Hours >= 0 && t.Minutes >= 0 && t.Seconds >= 0;

}

}

}

Client.EditWindows.EditRoom (Design)

<local:EditWindow x:Class="Client.EditWindows.EditRoom"

x:TypeArguments="data:Room"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:data="clr-namespace:Data.Models;assembly=Data"

xmlns:local="clr-namespace:Client.EditWindows"

mc:Ignorable="d"

Title="Edit Room" Width="300"

SizeToContent="WidthAndHeight"

DataContext="{Binding RelativeSource={RelativeSource Self}}">

<Grid>

<Grid.RowDefinitions>

<RowDefinition Height="Auto"/>

<RowDefinition Height="Auto"/>

<RowDefinition Height="Auto"/>

<RowDefinition Height="Auto"/>

<RowDefinition Height="Auto"/>

<RowDefinition Height="\*"/>

<RowDefinition Height="Auto"/>

</Grid.RowDefinitions>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="Auto"/>

<ColumnDefinition Width="\*"/>

</Grid.ColumnDefinitions>

<TextBlock Text="Room Name:" Margin="5" Grid.Row="0"/>

<TextBox Text="{Binding RoomName, UpdateSourceTrigger=PropertyChanged}" Margin="3" Grid.Row="0" Grid.Column="1" TabIndex="0"/>

<TextBlock Text="Standard Seats:" Margin="5" Grid.Row="1"/>

<TextBox Text="{Binding StandardSeats, UpdateSourceTrigger=PropertyChanged}" Margin="3" Grid.Row="1" Grid.Column="1"/>

<TextBlock Text="Special Seat Type:" Margin="5" Grid.Row="2"/>

<TextBox Text="{Binding SpecialSeatType, UpdateSourceTrigger=PropertyChanged}" Margin="3" Grid.Row="2" Grid.Column="1"/>

<TextBlock Text="Special Seats:" Margin="5" Grid.Row="3"/>

<TextBox Text="{Binding SpecialSeats, UpdateSourceTrigger=PropertyChanged}" Margin="3" Grid.Row="3" Grid.Column="1"/>

<TextBlock Text="Department:" Margin="5" Grid.Row="4"/>

<ComboBox ItemsSource="{Binding Departments}" SelectedItem="{Binding Department, UpdateSourceTrigger=PropertyChanged}" Margin="3" Grid.Row="4" Grid.Column="1"/>

<Grid Grid.Row="5" Grid.ColumnSpan="2">

<Grid.RowDefinitions>

<RowDefinition Height="Auto"/>

<RowDefinition Height="\*"/>

</Grid.RowDefinitions>

<TextBlock Text="Computer Names (one per row): " Margin="5"/>

<TextBox Text="{Binding Computers, UpdateSourceTrigger=PropertyChanged}" Height="100" VerticalScrollBarVisibility="Auto" Margin="5" Grid.Row="1"/>

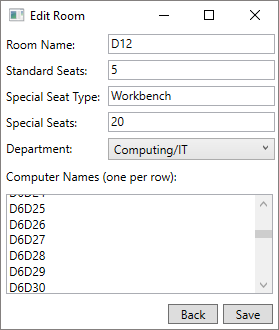
</Grid>

<Button Name="Button\_Back" Content="Back" Width="50" Height="20" Margin="5,5,60,5" HorizontalAlignment="Right" VerticalAlignment="Bottom" Grid.Row="6" Grid.Column="1" Click="Button\_Back\_Click"/>

<Button Name="Button\_Save" Content="Save" Width="50" Height="20" Margin="5" HorizontalAlignment="Right" VerticalAlignment="Bottom" Grid.Row="6" Grid.Column="1" Click="Button\_Save\_Click"/>

</Grid>

</local:EditWindow>



Example of an EditRoom window being used in the UI.

Client.EditWindows.EditRoom (Code-behind)

using System;

using System.Collections.Generic;

using System.Linq;

using System.Windows;

using Data.Models;

namespace Client.EditWindows

{

public partial class EditRoom

: EditWindow<Room>

{

// Name of the room being edited

protected string \_RoomName;

public string RoomName

{

get { return \_RoomName; }

set { \_RoomName = value; OnPropertyChanged("RoomName"); }

}

// String representation of the number of standard seats in the room

protected string \_StandardSeats;

public string StandardSeats

{

get { return \_StandardSeats; }

set { \_StandardSeats = value; OnPropertyChanged("StandardSeats"); }

}

// String representation of the number of special seats in the room

protected string \_SpecialSeats;

public string SpecialSeats

{

get { return \_SpecialSeats; }

set { \_SpecialSeats = value; OnPropertyChanged("SpecialSeats"); }

}

// The type of special seat (eg Computer, Workbench)

protected string \_SpecialSeatType;

public string SpecialSeatType

{

get { return \_SpecialSeatType; }

set { \_SpecialSeatType = value; OnPropertyChanged("SpecialSeatType"); }

}

// Name of the Department this room belnogs to

protected string \_Department;

public string Department

{

get { return \_Department; }

set { \_Department = value; OnPropertyChanged("Department"); }

}

// The names of the departments that this room can belong to

public string[] Departments { get; set; }

// The (long) string holding all the computers in the room, line delimited

protected string \_Computers;

public string Computers

{

get { return \_Computers; }

set { \_Computers = value; OnPropertyChanged("Computers"); }

}

// Utility property to get the individual computer names from the long string

public string[] ComputerLines { get { return Computers.Split('\n'); } }

// The ID of this room

protected int RoomId { get; set; }

// The list of bookings using this room (can't be edited)

protected List<Booking> Bookings { get; set; }

public EditRoom(Room Current)

{

// Store the names of all available departments

using (DataRepository Repo = new DataRepository())

Departments = Repo.Departments.Select(d => d.Name).ToArray();

InitializeComponent();

// Initialise with empty/existing details

if (Current == null)

{

RoomName = string.Empty;

StandardSeats = string.Empty;

SpecialSeatType = string.Empty;

SpecialSeats = string.Empty;

Bookings = new List<Booking>();

Computers = string.Empty;

RoomId = 0;

}

else

{

RoomName = Current.RoomName;

StandardSeats = Convert.ToString(Current.StandardSeats);

SpecialSeatType = Current.SpecialSeatType;

SpecialSeats = Convert.ToString(Current.SpecialSeats);

Department = Current.Department.Name;

Bookings = Current.Bookings;

// Get the list of computer names in the room and line-separate the computer names

Computers = Current.ComputerNamesJoined.Replace(Room.ComputerNameSeperator, '\n');

RoomId = Current.Id;

}

}

public override Room GetItem()

{

Room New = new Room();

try

{

// Fill out all the details

New.RoomName = RoomName;

New.StandardSeats = Convert.ToInt32(StandardSeats);

New.SpecialSeatType = SpecialSeatType;

New.SpecialSeats = Convert.ToInt32(SpecialSeats);

New.Id = RoomId;

New.Bookings = Bookings;

New.ComputerNames = ComputerLines.ToList();

// Select the correct department

using (DataRepository Repo = new DataRepository())

New.Department = Repo.Departments.Single(d => d.Name == Department);

}

catch

{

return null;

}

return New;

}

private void Button\_Back\_Click(object sender, RoutedEventArgs e)

{

DialogResult = false;

Close();

}

private void Button\_Save\_Click(object sender, RoutedEventArgs e)

{

string Error = null;

// Validate

int Temp;

if (string.IsNullOrWhiteSpace(RoomName))

Error = "You must enter a room name.";

else if (!int.TryParse(StandardSeats, out Temp) || Temp <= 0)

Error = "Standard seats must be a non-negative integer";

else if (!int.TryParse(SpecialSeats, out Temp) || Temp < 0)

Error = "Special seats must be a non-negative integer";

else if (Temp != 0 && string.IsNullOrWhiteSpace(SpecialSeatType))

Error = "You must enter a special seat type (eg Workbench, Computer)";

else if (ComputerLines.Any(s => s.Contains(Room.ComputerNameSeperator)))

Error = "A computer name cannot contain '" + Room.ComputerNameSeperator + "'.";

else if (string.IsNullOrWhiteSpace(Department))

Error = "You must select a department.";

else

{

// Check for naming conflicts

using (DataRepository Repo = new DataRepository())

if (Repo.Rooms.Any(r => r.Id != RoomId && r.RoomName == RoomName))

Error = "Another room with that name already exists.";

}

// Show an error message or close the window

if (Error != null)

MessageBox.Show(Error, "Error", MessageBoxButton.OK);

else

{

DialogResult = true;

Close();

}

}

}

}

Client.EditWindows.EditStudent (Design)

<local:EditWindow x:Class="Client.EditWindows.EditStudent"

x:TypeArguments="data:Student"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:Client.EditWindows"

xmlns:data="clr-namespace:Data.Models;assembly=Data"

mc:Ignorable="d"

Title="EditStudent" Height="250" Width="300"

DataContext="{Binding RelativeSource={RelativeSource Self}}">

<Grid>

<Grid.RowDefinitions>

<RowDefinition Height="Auto"/>

<RowDefinition Height="Auto"/>

<RowDefinition Height="Auto"/>

<RowDefinition Height="Auto"/>

<RowDefinition Height="Auto"/>

<RowDefinition Height="Auto"/>

<RowDefinition Height="\*"/>

</Grid.RowDefinitions>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="Auto"/>

<ColumnDefinition Width="\*"/>

</Grid.ColumnDefinitions>

<TextBlock Text="First Name:" Margin="5" Grid.Row="0"/>

<TextBox Text="{Binding FirstName, UpdateSourceTrigger=PropertyChanged}" Margin="3" Grid.Row="0" Grid.Column="1" TabIndex="0"/>

<TextBlock Text="Last Name:" Margin="5" Grid.Row="1"/>

<TextBox Text="{Binding LastName, UpdateSourceTrigger=PropertyChanged}" Margin="3" Grid.Row="1" Grid.Column="1"/>

<TextBlock Text="Logon Name:" Margin="5" Grid.Row="2"/>

<TextBox Text="{Binding LogonName, UpdateSourceTrigger=PropertyChanged}" Margin="3" Grid.Row="2" Grid.Column="1"/>

<TextBlock Text="Year:" Margin="5" Grid.Row="3"/>

<TextBox Text="{Binding Year, UpdateSourceTrigger=PropertyChanged}" Margin="3" Grid.Row="3" Grid.Column="1"/>

<TextBlock Text="Form:" Margin="5" Grid.Row="4"/>

<TextBox Text="{Binding Form, UpdateSourceTrigger=PropertyChanged}" Margin="3" Grid.Row="4" Grid.Column="1"/>

<TextBlock Text="Access:" Margin="5" Grid.Row="5"/>

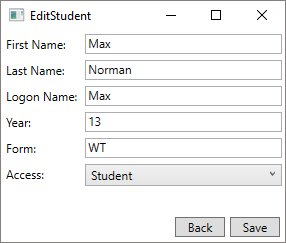
<ComboBox ItemsSource="{Binding AccessModes}" SelectedItem="{Binding Access, UpdateSourceTrigger=PropertyChanged}" Margin="3" Grid.Row="5" Grid.Column="1"/>

<Button Name="Button\_Back" Content="Back" Width="50" Height="20" Margin="5,5,60,5" HorizontalAlignment="Right" VerticalAlignment="Bottom" Grid.Row="6" Grid.Column="1" Click="Button\_Back\_Click"/>

<Button Name="Button\_Save" Content="Save" Width="50" Height="20" Margin="5" HorizontalAlignment="Right" VerticalAlignment="Bottom" Grid.Row="6" Grid.Column="1" Click="Button\_Save\_Click"/>

</Grid>

</local:EditWindow>



Example of an EditStudent window being used in the UI.

Client.EditWindows.EditStudent (Code-behind)

using System;

using System.Collections.Generic;

using System.Windows;

using Data.Models;

namespace Client.EditWindows

{

public partial class EditStudent

: EditWindow<Student>

{

// First name of the student being edited

protected string \_FirstName;

public string FirstName

{

get { return \_FirstName; }

set { \_FirstName = value; OnPropertyChanged("FirstName"); }

}

// Last name of the student being edited

protected string \_LastName;

public string LastName

{

get { return \_LastName; }

set { \_LastName = value; OnPropertyChanged("LastName"); }

}

// Username of the student being edited

protected string \_LogonName;

public string LogonName

{

get { return \_LogonName; }

set { \_LogonName = value; OnPropertyChanged("LogonName"); }

}

// Year the student's in

protected string \_Year;

public string Year

{

get { return \_Year; }

set { \_Year = value; OnPropertyChanged("Year"); }

}

// Form the student's in

protected string \_Form;

public string Form

{

get { return \_Form; }

set { \_Form = value; OnPropertyChanged("Form"); }

}

// Access level of the student

protected string \_Access;

public string Access

{

get { return \_Access; }

set { \_Access = value; OnPropertyChanged("Access"); }

}

// List of possible access modes

public string[] AccessModes { get; set; }

// Bookings and classes can't be edited from this window but need storing

protected List<Booking> Bookings { get; set; }

protected List<Class> Classes { get; set; }

// ID of the student being edited

public int StudentId { get; set; }

public EditStudent(Student Existing)

{

// Fill out the types of AccessMode

AccessModes = Enum.GetNames(typeof(AccessMode));

InitializeComponent();

// Initialise with empty/existing information

if (Existing != null)

{

FirstName = Existing.FirstName;

LastName = Existing.LastName;

LogonName = Existing.LogonName;

Year = Convert.ToString(Existing.Year);

Form = Existing.Form;

Access = Enum.GetName(typeof(AccessMode), Existing.Access);

Bookings = Existing.Bookings;

Classes = Existing.Classes;

StudentId = Existing.Id;

}

else

{

FirstName = string.Empty;

LastName = string.Empty;

LogonName = string.Empty;

Year = string.Empty;

Form = string.Empty;

Access = Enum.GetName(typeof(AccessMode), AccessMode.Student);

Bookings = new List<Booking>();

Classes = new List<Class>();

StudentId = 0;

}

}

public override Student GetItem()

{

Student New = new Student();

try

{

// Fill out the new details, parsing where necessary

New.FirstName = FirstName;

New.LastName = LastName;

New.LogonName = LogonName;

New.Year = Convert.ToInt32(Year);

New.Form = Form;

New.Access = (AccessMode)Enum.Parse(typeof(AccessMode), Access);

New.Bookings = Bookings;

New.Classes = Classes;

New.Id = StudentId;

}

catch

{

return null;

}

return New;

}

private void Button\_Back\_Click(object sender, RoutedEventArgs e)

{

DialogResult = false;

Close();

}

private void Button\_Save\_Click(object sender, RoutedEventArgs e)

{

string Error = null;

// Perform validation

AccessMode OutAccess;

int OutInt;

if (string.IsNullOrWhiteSpace(FirstName))

Error = "You must enter a first name.";

else if (string.IsNullOrWhiteSpace(LastName))

Error = "You must enter a last name.";

else if (string.IsNullOrWhiteSpace(LogonName))

Error = "You must enter a logon name.";

else if (!int.TryParse(Year, out OutInt) || OutInt < 0)

Error = "Year must be a non-negative integer.";

else if (string.IsNullOrWhiteSpace(Form))

Error = "You must enter a Form.";

else if (!Enum.TryParse(Access, out OutAccess)) // Should never happen, we're using a combobox

Error = "Invalid access mode.";

// Print the error message or close

if (!string.IsNullOrWhiteSpace(Error))

MessageBox.Show(Error, "Error", MessageBoxButton.OK);

else

{

DialogResult = true;

Close();

}

}

}

}

Client.EditWindows.EditTeacher (Design)

<local:EditWindow x:Class="Client.EditWindows.EditTeacher"

x:TypeArguments="data:Teacher"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:Client.EditWindows"

xmlns:data="clr-namespace:Data.Models;assembly=Data"

mc:Ignorable="d"

Title="Edit Teacher" Height="270" Width="300"

DataContext="{Binding RelativeSource={RelativeSource Self}}">

<Grid>

<Grid.RowDefinitions>

<RowDefinition Height="Auto"/>

<RowDefinition Height="Auto"/>

<RowDefinition Height="Auto"/>

<RowDefinition Height="Auto"/>

<RowDefinition Height="Auto"/>

<RowDefinition Height="Auto"/>

<RowDefinition Height="Auto"/>

<RowDefinition Height="\*"/>

</Grid.RowDefinitions>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="Auto"/>

<ColumnDefinition Width="\*"/>

</Grid.ColumnDefinitions>

<TextBlock Text="First Name:" Margin="5" Grid.Row="0"/>

<TextBox Name="Text\_FirstName" Text="{Binding FirstName, UpdateSourceTrigger=PropertyChanged}" Margin="3" Grid.Row="0" Grid.Column="1" TabIndex="0"/>

<TextBlock Text="Last Name:" Margin="5" Grid.Row="1"/>

<TextBox Name="Text\_LastName" Text="{Binding LastName, UpdateSourceTrigger=PropertyChanged}" Margin="3" Grid.Row="1" Grid.Column="1"/>

<TextBlock Text="Title:" Margin="5" Grid.Row="2"/>

<TextBox Name="Text\_Title" Text="{Binding TeacherTitle, UpdateSourceTrigger=PropertyChanged}" Margin="3" Grid.Row="2" Grid.Column="1"/>

<TextBlock Text="Logon Name:" Margin="5" Grid.Row="3"/>

<TextBox Name="Text\_LogonName" Text="{Binding LogonName, UpdateSourceTrigger=PropertyChanged}" Margin="3" Grid.Row="3" Grid.Column="1"/>

<TextBlock Text="Access:" Margin="5" Grid.Row="4"/>

<ComboBox Name="Combo\_Access" ItemsSource="{Binding AccessModes}" SelectedItem="{Binding Access, UpdateSourceTrigger=PropertyChanged}" Margin="3" Grid.Row="4" Grid.Column="1"/>

<TextBlock Text="Email:" Margin="5" Grid.Row="5"/>

<TextBox Name="Text\_Email" Text="{Binding Email, UpdateSourceTrigger=PropertyChanged}" Margin="3" Grid.Row="5" Grid.Column="1"/>

<TextBlock Text="Department:" Margin="5" Grid.Row="6"/>

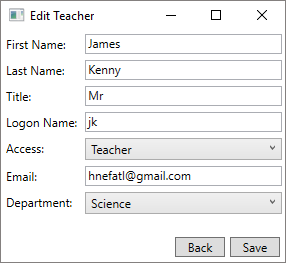
<ComboBox Name="Combo\_Department" ItemsSource="{Binding Departments}" SelectedItem="{Binding Department, UpdateSourceTrigger=PropertyChanged}" Margin="3" Grid.Row="6" Grid.Column="1"/>

<Button Name="Button\_Back" Content="Back" Width="50" Height="20" Margin="5,5,60,5" HorizontalAlignment="Right" VerticalAlignment="Bottom" Grid.Row="7" Grid.Column="1" Click="Button\_Back\_Click"/>

<Button Name="Button\_Save" Content="Save" Width="50" Height="20" Margin="5" HorizontalAlignment="Right" VerticalAlignment="Bottom" Grid.Row="7" Grid.Column="1" Click="Button\_Save\_Click"/>

</Grid>

</local:EditWindow>



Example of an EditTeacher window being used in the UI.

Client.EditWindows.EditTeacher (Code-behind)

using System;

using System.Collections.Generic;

using System.Linq;

using System.Windows;

using System.Text.RegularExpressions;

using Data.Models;

namespace Client.EditWindows

{

public partial class EditTeacher

: EditWindow<Teacher>

{

// The first name of the teacher being edited

protected string \_FirstName;

public string FirstName

{

get { return \_FirstName; }

set { \_FirstName = value; OnPropertyChanged("FirstName"); }

}

// The last name of the teacher being edited

protected string \_LastName;

public string LastName

{

get { return \_LastName; }

set { \_LastName = value; OnPropertyChanged("LastName"); }

}

// The title of the teacher being edited (eg Mr, Mrs)

protected string \_TeacherTitle;

public string TeacherTitle

{

get { return \_TeacherTitle; }

set { \_TeacherTitle = value; OnPropertyChanged("TeacherTitle"); }

}

// The username of the teacher being edited

protected string \_LogonName;

public string LogonName

{ get

{ return \_LogonName; }

set { \_LogonName = value; OnPropertyChanged("LogonName"); }

}

// The access level of the teacher

protected string \_Access;

public string Access

{ get

{ return \_Access; }

set { \_Access = value; OnPropertyChanged("Access"); }

}

public string[] AccessModes { get { return Enum.GetNames(typeof(AccessMode)); } }

// The email address of the teacher

protected string \_Email;

public string Email

{

get { return \_Email; }

set { \_Email = value; OnPropertyChanged("Email"); }

}

// Name of the department the teacher belongs to

protected string \_Department;

public string Department

{

get { return \_Department; }

set { \_Department = value; OnPropertyChanged("Department"); }

}

// Names of departments that can be selected

public string[] Departments { get; set; }

// Classes and Bookings can't be edited but need to be stored

protected List<Class> Classes { get; set; }

protected List<Booking> Bookings { get; set; }

// The ID of the teacher being edited

public int TeacherId { get; set; }

public EditTeacher(Teacher Existing)

{

// Store the names of departments that can be chosen

using (DataRepository Repo = new DataRepository())

Departments = Repo.Departments.Select(d => d.Name).ToArray();

InitializeComponent();

// Initialise with empty values/existing values

if (Existing == null)

{

FirstName = string.Empty;

LastName = string.Empty;

TeacherTitle = string.Empty;

LogonName = string.Empty;

Access = string.Empty;

Email = string.Empty;

Department = string.Empty;

Classes = new List<Class>();

Bookings = new List<Booking>();

TeacherId = 0;

}

else

{

FirstName = Existing.FirstName;

LastName = Existing.LastName;

TeacherTitle = Existing.Title;

LogonName = Existing.LogonName;

Access = Enum.GetName(typeof(AccessMode), Existing.Access);

Email = Existing.Email;

Department = Existing.Department.Name;

Classes = Existing.Classes;

Bookings = Existing.Bookings;

TeacherId = Existing.Id;

}

}

public override Teacher GetItem()

{

Teacher New = new Teacher();

try

{

// Fill out the details

New.FirstName = FirstName;

New.LastName = LastName;

New.Title = TeacherTitle;

New.LogonName = LogonName;

New.Access = (AccessMode)Enum.Parse(typeof(AccessMode), Access);

New.Email = Email;

// Get a reference to the actual department

using (DataRepository Repo = new DataRepository())

New.Department = Repo.Departments.Single(d => d.Name == Department);

New.Classes = Classes;

New.Bookings = Bookings;

New.Id = TeacherId;

}

catch

{

return null;

}

return New;

}

private void Button\_Back\_Click(object sender, RoutedEventArgs e)

{

DialogResult = false;

Close();

}

private void Button\_Save\_Click(object sender, RoutedEventArgs e)

{

string Error = null;

// Validate

AccessMode Out;

if (string.IsNullOrWhiteSpace(FirstName))

Error = "You must enter a first name.";

else if (string.IsNullOrWhiteSpace(LastName))

Error = "You must enter a last name.";

else if (string.IsNullOrWhiteSpace(TeacherTitle))

Error = "You must enter a title.";

else if (string.IsNullOrWhiteSpace(LogonName))

Error = "You must enter a logon name.";

else if (!Enum.TryParse(Access, out Out)) // Should never happen, we're using a combobox

Error = "Invalid access mode.";

else if (string.IsNullOrWhiteSpace(Department))

Error = "You must enter a Department.";

else if (!string.IsNullOrEmpty(Email) && !Regex.IsMatch(Email, @"[\w.-]+@[\w]+\.[.\w]+"))

Error = "Invalid email address.";

// Show an error or close

if (!string.IsNullOrWhiteSpace(Error))

MessageBox.Show(Error, "Error", MessageBoxButton.OK);

else

{

DialogResult = true;

Close();

}

}

}

}

Client.EditWindows.EditSubject (Design)

<local:EditWindow x:Class="Client.EditWindows.EditSubject"

x:TypeArguments="data:Subject"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:Client.EditWindows"

xmlns:data="clr-namespace:Data.Models;assembly=Data"

mc:Ignorable="d"

Title="Edit Subject" Height="Auto" Width="Auto"

SizeToContent="WidthAndHeight"

DataContext="{Binding RelativeSource={RelativeSource Self}}">

<Grid>

<Grid.RowDefinitions>

<RowDefinition Height="Auto"/>

<RowDefinition Height="Auto"/>

<RowDefinition Height="Auto"/>

</Grid.RowDefinitions>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="Auto"/>

<ColumnDefinition Width="\*"/>

</Grid.ColumnDefinitions>

<TextBlock Text="Subject Name:" Margin="5" Grid.Row="0" Grid.Column="0"/>

<TextBox Text="{Binding SubjectName, UpdateSourceTrigger=PropertyChanged}" Margin="5" Grid.Row="0" Grid.Column="1" TabIndex="0"/>

<GroupBox Header="Colour" Margin="5" Grid.Row="1" Grid.Column="0" Grid.ColumnSpan="2">

<Grid>

<Grid.RowDefinitions>

<RowDefinition Height="Auto"/>

<RowDefinition Height="Auto"/>

<RowDefinition Height="Auto"/>

</Grid.RowDefinitions>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="Auto"/>

<ColumnDefinition Width="Auto"/>

<ColumnDefinition Width="\*"/>

</Grid.ColumnDefinitions>

<TextBlock Text="Red:" Margin="5,7,5,5" Grid.Row="0" Grid.Column="0"/>

<TextBox Text="{Binding ColourRed, UpdateSourceTrigger=PropertyChanged}" PreviewTextInput="ComponentTextBox\_TextChanged" Margin="5" Width="50" Height="22" Grid.Row="0" Grid.Column="1"/>

<TextBlock Text="Green:" Margin="5,7,5,5" Grid.Row="1" Grid.Column="0"/>

<TextBox Text="{Binding ColourGreen, UpdateSourceTrigger=PropertyChanged}" PreviewTextInput="ComponentTextBox\_TextChanged" Margin="5" Width="50" Height="22" Grid.Row="1" Grid.Column="1"/>

<TextBlock Text="Blue:" Margin="5,7,5,5" Grid.Row="2" Grid.Column="0"/>

<TextBox Text="{Binding ColourBlue, UpdateSourceTrigger=PropertyChanged}" PreviewTextInput="ComponentTextBox\_TextChanged" Margin="5" Width="50" Height="22" Grid.Row="2" Grid.Column="1"/>

<Rectangle Name="Rect\_Demo" Fill="{Binding Colour, UpdateSourceTrigger=PropertyChanged}" Margin="5" HorizontalAlignment="Right" Width="100" Height="100" Grid.Row="0" Grid.RowSpan="3" Grid.Column="2"/>

</Grid>

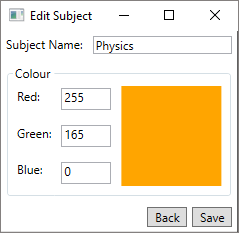
</GroupBox>

<Button Name="Button\_Back" Content="Back" Margin="5,5,50,5" Grid.Row="2" Grid.Column="1" Width="40" HorizontalAlignment="Right" Click="Button\_Back\_Click"/>

<Button Name="Button\_Save" Content="Save" Margin="5" Grid.Row="2" Grid.Column="1" Width="40" HorizontalAlignment="Right" Click="Button\_Save\_Click"/>

</Grid>

</local:EditWindow>



Example of an EditSubject window being used in the UI.

Client.EditWindows.EditSubject

using System;

using System.Linq;

using System.Collections.Generic;

using System.Windows;

using System.Windows.Media;

using System.Windows.Input;

using Data.Models;

using System.Windows.Controls;

namespace Client.EditWindows

{

public partial class EditSubject

: EditWindow<Subject>

{

// The Subject Name, as used in the UI

protected string \_SubjectName;

public string SubjectName

{

get { return \_SubjectName; }

set { \_SubjectName = value; OnPropertyChanged("SubjectName"); }

}

// Gets a Brush for the demonstration square by converting the components

public SolidColorBrush Colour

{

get

The UI rectangle must bind to a *SolidColorBrush*, so this property is supplied to convert from the individual components which are used to form a *Color* object for storage in the database, into the required type.

{

return new SolidColorBrush(new Color()

{

R = Convert.ToByte(ColourRed),

G = Convert.ToByte(ColourGreen),

B = Convert.ToByte(ColourBlue),

A = byte.MaxValue,

});

}

set

{

ColourRed = Convert.ToString(value.Color.R);

ColourGreen = Convert.ToString(value.Color.G);

ColourBlue = Convert.ToString(value.Color.B);

OnPropertyChanged("Colour");

}

}

// The red component as used in the UI

protected string \_ColourRed;

public string ColourRed

{

get { return \_ColourRed; }

set { \_ColourRed = string.IsNullOrWhiteSpace(value) ? "0" : value; OnPropertyChanged("ColourRed"); OnPropertyChanged("Colour"); }

}

// The green component as used in the UI

protected string \_ColourGreen;

public string ColourGreen

{

get { return \_ColourGreen; }

set { \_ColourGreen = string.IsNullOrWhiteSpace(value) ? "0" : value; OnPropertyChanged("ColourGreen"); OnPropertyChanged("Colour"); }

}

// The blue component as used in the UI

protected string \_ColourBlue;

public string ColourBlue

{

get { return \_ColourBlue; }

set { \_ColourBlue = string.IsNullOrWhiteSpace(value) ? "0" : value; OnPropertyChanged("ColourBlue"); OnPropertyChanged("Colour"); }

}

// Stores the uneditable settings until the item's recreated

protected int SubjectId { get; set; }

protected List<Booking> Bookings { get; set; }

public EditSubject(Subject Existing)

{

if (Existing == null)

{

SubjectId = 0;

SubjectName = string.Empty;

Colour = Brushes.Black;

Bookings = new List<Booking>();

}

else

{

If text is entered into one of the textboxes, and the resulting text isn’t valid, then prevent the input from being entered by settings the event “handled” property to true.

SubjectId = Existing.Id;

SubjectName = Existing.SubjectName;

Colour = new SolidColorBrush(Existing.Colour);

Bookings = Existing.Bookings;

}

InitializeComponent();

}

protected void ComponentTextBox\_TextChanged(object sender, TextCompositionEventArgs e)

{

byte Out;

// Don't allow the user to enter something that's not a valid byte

if (!byte.TryParse((sender as TextBox).Text + e.Text, out Out))

e.Handled = true;

}

public override Subject GetItem()

{

// Fill out the details as necessary

return new Subject() { Id = SubjectId, SubjectName = SubjectName, Colour = Colour.Color, Bookings = Bookings };

}

protected void Button\_Back\_Click(object sender, RoutedEventArgs e)

{

// Close with a negative flag

DialogResult = false;

Close();

}

protected void Button\_Save\_Click(object sender, RoutedEventArgs e)

{

string Error = null;

// Validate

byte Temp;

if (string.IsNullOrWhiteSpace(SubjectName))

Error = "You must enter a subject name.";

else if (!byte.TryParse(ColourRed, out Temp))

Error = "Invalid value for Red component.";

else if (!byte.TryParse(ColourGreen, out Temp))

Error = "Invalid value for Green component.";

else if (!byte.TryParse(ColourBlue, out Temp))

Error = "Invalid value for Blue component.";

else

{

using (DataRepository Repo = new DataRepository())

if (Repo.Subjects.Any(s => s.SubjectName == SubjectName))

Error = "Another Subject with that name already exists.";

}

// Show an error message or close the window

if (Error != null)

MessageBox.Show(Error, "Error", MessageBoxButton.OK);

else

{

DialogResult = true;

Close();

}

}

}

}

Client.Admin.AdminWindow (Design)

<Window x:Class="Client.Admin.AdminWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:Client.Admin"

xmlns:c="clr-namespace:Client.Converters"

mc:Ignorable="d"

Title="Admin Controls" Height="500" Width="600"

DataContext="{Binding RelativeSource={RelativeSource Self}}">

The AdminWindow design section is very large – it describes the layout of 7 tabs, each of which displays the details and allows for customisation of one of the entities in the system (Rooms, Periods, Teachers, Students, Departments, Classes , and Subjects). Bookings are excluded as these are edited from the Timetable window.  
All tabs have a very similar layout – a central listview showing the details of all the entities, then 3 buttons for the creation, editing, and deletion of records.

<Window.Resources>

<c:BooleanToVisibilityConverter x:Key="BoolToVis"/>

<c:NullableToVisibilityConverter x:Key="NullToVis"/>

<c:NullableToBoolConverter x:Key="NullToBool"/>

<c:StringToIntConverter x:Key="StringToInt"/>

<c:IntToStringConverter x:Key="IntToString"/>

</Window.Resources>

<Grid>

<TabControl Grid.Row="0">

<TabItem Header="Rooms">

<Grid>

<Grid.RowDefinitions>

<RowDefinition Height="\*"/>

<RowDefinition Height="Auto"/>

</Grid.RowDefinitions>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="\*"/>

<ColumnDefinition Width="Auto"/>

</Grid.ColumnDefinitions>

<ListView Name="List\_Rooms" SelectionMode="Single" ItemsSource="{Binding Rooms, UpdateSourceTrigger=PropertyChanged}" Grid.Row="0" Grid.Column="0">

<ListView.View>

<GridView>

<GridView.Columns>

<GridViewColumn DisplayMemberBinding="{Binding RoomName}" Width="80" Header="Room Name"/>

<GridViewColumn DisplayMemberBinding="{Binding StandardSeats}" Width="110" Header="Standard Seats"/>

<GridViewColumn DisplayMemberBinding="{Binding SpecialSeatType}" Width="100" Header="Special seat type"/>

<GridViewColumn DisplayMemberBinding="{Binding SpecialSeats}" Width="80" Header="Special seats"/>

</GridView.Columns>

</GridView>

</ListView.View>

</ListView>

<Grid Grid.Row="1" Grid.ColumnSpan="2">

<Grid.ColumnDefinitions>

<ColumnDefinition Width="Auto"/>

<ColumnDefinition Width="Auto"/>

<ColumnDefinition Width="Auto"/>

</Grid.ColumnDefinitions>

<Button Name="Button\_AddRoom" Content="Add Room" Height="25" Width="110" Margin="5" Grid.Column="0" Click="Button\_AddRoom\_Click"/>

<Button Name="Button\_EditRoom" Content="Edit Room" IsEnabled="{Binding ElementName=List\_Rooms, Path=SelectedItem, Converter={StaticResource NullToBool}}" Height="25" Width="110" Margin="5" Grid.Column="1" Click="Button\_EditRoom\_Click"/>

<Button Name="Button\_DeleteRoom" Content="Delete Room" IsEnabled="{Binding ElementName=List\_Rooms, Path=SelectedItem, Converter={StaticResource NullToBool}}" Height="25" Width="110" Margin="5" Grid.Column="2" Click="Button\_DeleteRoom\_Click"/>

</Grid>

</Grid>

</TabItem>

<TabItem Header="Periods">

<Grid>

<Grid.RowDefinitions>

<RowDefinition Height="\*"/>

<RowDefinition Height="Auto"/>

</Grid.RowDefinitions>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="\*"/>

<ColumnDefinition Width="Auto"/>

</Grid.ColumnDefinitions>

<ListView Name="List\_Periods" SelectionMode="Single" ItemsSource="{Binding Periods, UpdateSourceTrigger=PropertyChanged}" Grid.Row="0" Grid.Column="0">

<ListView.View>

<GridView>

<GridView.Columns>

<GridViewColumn DisplayMemberBinding="{Binding Name}" Width="80" Header="Name"/>

<GridViewColumn DisplayMemberBinding="{Binding ShortStart}" Width="110" Header="Start Time"/>

<GridViewColumn DisplayMemberBinding="{Binding ShortEnd}" Width="100" Header="End Time"/>

</GridView.Columns>

</GridView>

</ListView.View>

</ListView>

<Grid Grid.Row="1" Grid.ColumnSpan="2">

<Grid.ColumnDefinitions>

<ColumnDefinition Width="Auto"/>

<ColumnDefinition Width="Auto"/>

<ColumnDefinition Width="Auto"/>

</Grid.ColumnDefinitions>

<Button Name="Button\_AddPeriod" Content="Add Period" Height="25" Width="110" Margin="5" Grid.Column="0" Click="Button\_AddPeriod\_Click"/>

<Button Name="Button\_EditPeriod" Content="Edit Period" IsEnabled="{Binding ElementName=List\_Periods, Path=SelectedItem, Converter={StaticResource NullToBool}}" Height="25" Width="110" Margin="5" Grid.Column="1" Click="Button\_EditPeriod\_Click"/>

<Button Name="Button\_DeletePeriod" Content="Delete Period" IsEnabled="{Binding ElementName=List\_Periods, Path=SelectedItem, Converter={StaticResource NullToBool}}" Height="25" Width="110" Margin="5" Grid.Column="2" Click="Button\_DeletePeriod\_Click"/>

</Grid>

</Grid>

</TabItem>

<TabItem Header="Teachers">

<Grid>

<Grid.RowDefinitions>

<RowDefinition Height="\*"/>

<RowDefinition Height="Auto"/>

</Grid.RowDefinitions>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="\*"/>

<ColumnDefinition Width="Auto"/>

</Grid.ColumnDefinitions>

<ListView Name="List\_Teachers" SelectionMode="Single" ItemsSource="{Binding Teachers, UpdateSourceTrigger=PropertyChanged}" Grid.Row="0" Grid.Column="0">

<ListView.View>

<GridView>

<GridView.Columns>

<GridViewColumn DisplayMemberBinding="{Binding FirstName}" Width="80" Header="First Name"/>

<GridViewColumn DisplayMemberBinding="{Binding LastName}" Width="110" Header="Last Name"/>

<GridViewColumn DisplayMemberBinding="{Binding Title}" Width="50" Header="Title"/>

<GridViewColumn DisplayMemberBinding="{Binding LogonName}" Width="80" Header="Logon Name"/>

<GridViewColumn DisplayMemberBinding="{Binding Access}" Width="80" Header="Access"/>

<GridViewColumn DisplayMemberBinding="{Binding Email}" Width="150" Header="Email"/>

</GridView.Columns>

</GridView>

</ListView.View>

</ListView>

<Grid Grid.Row="1" Grid.ColumnSpan="2">

<Grid.ColumnDefinitions>

<ColumnDefinition Width="Auto"/>

<ColumnDefinition Width="Auto"/>

<ColumnDefinition Width="Auto"/>

</Grid.ColumnDefinitions>

<Button Name="Button\_AddTeacher" Content="Add Teacher" Height="25" Width="110" Margin="5" Grid.Column="0" Click="Button\_AddTeacher\_Click"/>

<Button Name="Button\_EditTeacher" Content="Edit Teacher" IsEnabled="{Binding ElementName=List\_Teachers, Path=SelectedItem, Converter={StaticResource NullToBool}}" Height="25" Width="110" Margin="5" Grid.Column="1" Click="Button\_EditTeacher\_Click"/>

<Button Name="Button\_DeleteTeacher" Content="Delete Teacher" IsEnabled="{Binding ElementName=List\_Teachers, Path=SelectedItem, Converter={StaticResource NullToBool}}" Height="25" Width="110" Margin="5" Grid.Column="2" Click="Button\_DeleteTeacher\_Click"/>

</Grid>

</Grid>

</TabItem>

<TabItem Header="Students">

<Grid>

<Grid.RowDefinitions>

<RowDefinition Height="\*"/>

<RowDefinition Height="Auto"/>

</Grid.RowDefinitions>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="\*"/>

<ColumnDefinition Width="Auto"/>

</Grid.ColumnDefinitions>

<ListView Name="List\_Students" SelectionMode="Single" ItemsSource="{Binding Students, UpdateSourceTrigger=PropertyChanged}" Grid.Row="0" Grid.Column="0">

<ListView.View>

<GridView>

<GridView.Columns>

<GridViewColumn DisplayMemberBinding="{Binding FirstName}" Width="150" Header="First Name"/>

<GridViewColumn DisplayMemberBinding="{Binding LastName}" Width="150" Header="Last Name"/>

<GridViewColumn DisplayMemberBinding="{Binding LogonName}" Width="100" Header="Logon Name"/>

<GridViewColumn DisplayMemberBinding="{Binding Year}" Width="50" Header="Year"/>

<GridViewColumn DisplayMemberBinding="{Binding Form}" Width="50" Header="Form"/>

</GridView.Columns>

</GridView>

</ListView.View>

</ListView>

<Grid Grid.Row="1" Grid.ColumnSpan="2">

<Grid.ColumnDefinitions>

<ColumnDefinition Width="Auto"/>

<ColumnDefinition Width="Auto"/>

<ColumnDefinition Width="Auto"/>

</Grid.ColumnDefinitions>

<Button Name="Button\_AddStudent" Content="Add Student" Height="25" Width="110" Margin="5" Grid.Column="0" Click="Button\_AddStudent\_Click"/>

<Button Name="Button\_EditStudent" Content="Edit Student" IsEnabled="{Binding ElementName=List\_Students, Path=SelectedItem, Converter={StaticResource NullToBool}}" Height="25" Width="110" Margin="5" Grid.Column="1" Click="Button\_EditStudent\_Click"/>

<Button Name="Button\_DeleteStudent" Content="Delete Student" IsEnabled="{Binding ElementName=List\_Students, Path=SelectedItem, Converter={StaticResource NullToBool}}" Height="25" Width="110" Margin="5" Grid.Column="2" Click="Button\_DeleteStudent\_Click"/>

</Grid>

</Grid>

</TabItem>

<TabItem Header="Departments">

<Grid>

<Grid.RowDefinitions>

<RowDefinition Height="\*"/>

<RowDefinition Height="Auto"/>

</Grid.RowDefinitions>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="\*"/>

<ColumnDefinition Width="Auto"/>

</Grid.ColumnDefinitions>

<ListView Name="List\_Departments" SelectionMode="Single" ItemsSource="{Binding Departments, UpdateSourceTrigger=PropertyChanged}" Grid.Row="0" Grid.Column="0">

<ListView.View>

<GridView>

<GridView.Columns>

<GridViewColumn DisplayMemberBinding="{Binding Name}" Width="300" Header="Name"/>

</GridView.Columns>

</GridView>

</ListView.View>

</ListView>

<Grid Grid.Row="1" Grid.ColumnSpan="2">

<Grid.ColumnDefinitions>

<ColumnDefinition Width="Auto"/>

<ColumnDefinition Width="Auto"/>

<ColumnDefinition Width="Auto"/>

</Grid.ColumnDefinitions>

<Button Name="Button\_AddDepartment" Content="Add Department" Height="25" Width="110" Margin="5" Grid.Column="0" Click="Button\_AddDepartment\_Click"/>

<Button Name="Button\_EditDepartment" Content="Edit Department" IsEnabled="{Binding ElementName=List\_Departments, Path=SelectedItem, Converter={StaticResource NullToBool}}" Height="25" Width="110" Margin="5" Grid.Column="1" Click="Button\_EditDepartment\_Click"/>

<Button Name="Button\_DeleteDepartment" Content="Delete Department" IsEnabled="{Binding ElementName=List\_Departments, Path=SelectedItem, Converter={StaticResource NullToBool}}" Height="25" Width="110" Margin="5" Grid.Column="2" Click="Button\_DeleteDepartment\_Click"/>

</Grid>

</Grid>

</TabItem>

<TabItem Header="Classes">

<Grid>

<Grid.RowDefinitions>

<RowDefinition Height="\*"/>

<RowDefinition Height="Auto"/>

</Grid.RowDefinitions>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="\*"/>

<ColumnDefinition Width="Auto"/>

</Grid.ColumnDefinitions>

<ListView Name="List\_Classes" SelectionMode="Single" ItemsSource="{Binding Classes, UpdateSourceTrigger=PropertyChanged}" Grid.Row="0" Grid.Column="0">

<ListView.View>

<GridView>

<GridView.Columns>

<GridViewColumn DisplayMemberBinding="{Binding ClassName}" Width="200" Header="Name"/>

<GridViewColumn DisplayMemberBinding="{Binding Owner.InformalName}" Width="200" Header="Owner"/>

</GridView.Columns>

</GridView>

</ListView.View>

</ListView>

<Grid Grid.Row="1" Grid.ColumnSpan="2">

<Grid.ColumnDefinitions>

<ColumnDefinition Width="Auto"/>

<ColumnDefinition Width="Auto"/>

<ColumnDefinition Width="Auto"/>

</Grid.ColumnDefinitions>

<Button Name="Button\_AddClass" Content="Add Class" Height="25" Width="110" Margin="5" Grid.Column="0" Click="Button\_AddClass\_Click"/>

<Button Name="Button\_EditClass" Content="Edit Class" IsEnabled="{Binding ElementName=List\_Classes, Path=SelectedItem, Converter={StaticResource NullToBool}}" Height="25" Width="110" Margin="5" Grid.Column="1" Click="Button\_EditClass\_Click"/>

<Button Name="Button\_DeleteClass" Content="Delete Class" IsEnabled="{Binding ElementName=List\_Classes, Path=SelectedItem, Converter={StaticResource NullToBool}}" Height="25" Width="110" Margin="5" Grid.Column="2" Click="Button\_DeleteClass\_Click"/>

</Grid>

</Grid>

</TabItem>

<TabItem Header="Subjects">

<Grid>

<Grid.RowDefinitions>

<RowDefinition Height="\*"/>

<RowDefinition Height="Auto"/>

</Grid.RowDefinitions>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="\*"/>

<ColumnDefinition Width="Auto"/>

</Grid.ColumnDefinitions>

<ListView Name="List\_Subjects" SelectionMode="Single" ItemsSource="{Binding Subjects, UpdateSourceTrigger=PropertyChanged}" Grid.Row="0" Grid.Column="0">

<ListView.View>

<GridView>

<GridView.Columns>

<GridViewColumn DisplayMemberBinding="{Binding SubjectName}" Width="200" Header="Subject Name"/>

</GridView.Columns>

</GridView>

</ListView.View>

</ListView>

<Grid Grid.Row="1" Grid.ColumnSpan="2">

<Grid.ColumnDefinitions>

<ColumnDefinition Width="Auto"/>

<ColumnDefinition Width="Auto"/>

<ColumnDefinition Width="Auto"/>

</Grid.ColumnDefinitions>

<Button Name="Button\_AddSubject" Content="Add Subject" Height="25" Width="110" Margin="5" Grid.Column="0" Click="Button\_AddSubject\_Click"/>

<Button Name="Button\_EditSubject" Content="Edit Subject" IsEnabled="{Binding ElementName=List\_Subjects, Path=SelectedItem, Converter={StaticResource NullToBool}}" Height="25" Width="110" Margin="5" Grid.Column="1" Click="Button\_EditSubject\_Click"/>

<Button Name="Button\_DeleteSubject" Content="Delete Subject" IsEnabled="{Binding ElementName=List\_Subjects, Path=SelectedItem, Converter={StaticResource NullToBool}}" Height="25" Width="110" Margin="5" Grid.Column="2" Click="Button\_DeleteSubject\_Click"/>

</Grid>

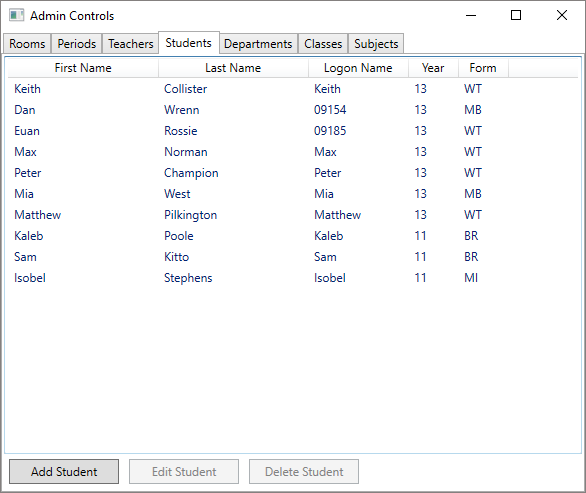
</Grid>

</TabItem>

</TabControl>

</Grid>

</Window>



The AdminWindow on the “Students” tab.

Client.Admin.AdminWindow (Code-behind)

The code-behind is fairly simple – there are a lot of *ObservableCollection* declarations to hold the lists of entities that the UI listviews then bind to, and the rest of the code is mostly just event handlers for the various buttons on the window (add, edit, delete for each entity).  
The *EditData* generic function is really useful, it takes advantage of the common base class of the Edit Windows to generically store the item that an Edit Window creates.

using System;

using System.Linq;

using System.Windows;

using System.ComponentModel;

using System.Collections.ObjectModel;

using Data.Models;

using NetCore.Client;

using Client.EditWindows;

namespace Client.Admin

{

public partial class AdminWindow

: Window, INotifyPropertyChanged

{

// Reference to the connection to the server

public Connection Connection { get; set; }

// Reference to the current user

public User CurrentUser { get; set; }

// List of all rooms in the system

protected ObservableCollection<Room> \_Rooms = new ObservableCollection<Room>();

public ObservableCollection<Room> Rooms

{

get { return \_Rooms; }

set { \_Rooms = value; OnPropertyChanged("Rooms"); }

}

// List of all periods in the system

protected ObservableCollection<TimeSlot> \_Periods = new ObservableCollection<TimeSlot>();

public ObservableCollection<TimeSlot> Periods

{

get { return \_Periods; }

set { \_Periods = value; OnPropertyChanged("Periods"); }

}

// List of all teachers in the system

protected ObservableCollection<Teacher> \_Teachers = new ObservableCollection<Teacher>();

public ObservableCollection<Teacher> Teachers

{

get { return \_Teachers; }

set { \_Teachers = value; OnPropertyChanged("Teachers"); }

}

// List of all students in the system

protected ObservableCollection<Student> \_Students = new ObservableCollection<Student>();

public ObservableCollection<Student> Students

{

get { return \_Students; }

set { \_Students = value; OnPropertyChanged("Students"); }

}

// List of all departments in the system

protected ObservableCollection<Department> \_Departments = new ObservableCollection<Department>();

public ObservableCollection<Department> Departments

{

get { return \_Departments; }

set { \_Departments = value; OnPropertyChanged("Departments"); }

}

// List of all classes in the system

protected ObservableCollection<Class> \_Classes = new ObservableCollection<Class>();

public ObservableCollection<Class> Classes

{

get { return \_Classes; }

set { \_Classes = value; OnPropertyChanged("Classes"); }

}

// List of all subjects in the system

protected ObservableCollection<Subject> \_Subjects = new ObservableCollection<Subject>();

public ObservableCollection<Subject> Subjects

{

get { return \_Subjects; }

set { \_Subjects = value; OnPropertyChanged("Subjects"); }

}

public AdminWindow(Connection Connection, User CurrentUser)

{

InitializeComponent();

this.Connection = Connection;

this.CurrentUser = CurrentUser;

// Take a snapshot of the contents of the database

using (DataRepository Repo = new DataRepository())

{

Rooms = new ObservableCollection<Room>(Repo.Rooms);

Periods = new ObservableCollection<TimeSlot>(Repo.Periods);

Teachers = new ObservableCollection<Teacher>(Repo.Users.OfType<Teacher>());

Students = new ObservableCollection<Student>(Repo.Users.OfType<Student>());

Departments = new ObservableCollection<Department>(Repo.Departments);

Classes = new ObservableCollection<Class>(Repo.Classes);

Subjects = new ObservableCollection<Subject>(Repo.Subjects);

}

}

// Utility function that customises a generic item

// Importantly, makes use of generics to reduce code bloat

// Takes the abstract EditWindow to actually use (determines the type)

private void EditData<T>(EditWindow<T> Wnd) where T : DataModel

{

// Show the window and store the close result

bool? Result = Wnd.ShowDialog();

// If closed succesfully, process the item

if (Result.HasValue && Result.Value)

{

// Acquire the (generic) item

T New = Wnd.GetItem();

if (New != null)

{

// Add the item to the relevant table

Type t = typeof(T);

using (DataRepository Repo = new DataRepository())

{

if (t == typeof(Booking))

Repo.Bookings.Add((Booking)(object)New);

else if (t == typeof(Class))

Repo.Classes.Add((Class)(object)New);

else if (t == typeof(Department))

Repo.Departments.Add((Department)(object)New);

else if (t == typeof(Room))

Repo.Rooms.Add((Room)(object)New);

else if (t == typeof(Student))

Repo.Users.Add((Student)(object)New);

else if (t == typeof(Subject))

Repo.Subjects.Add((Subject)(object)New);

else if (t == typeof(Teacher))

Repo.Users.Add((Teacher)(object)New);

else if (t == typeof(TimeSlot))

Repo.Periods.Add((TimeSlot)(object)New);

}

}

}

}

// Button handlers for the Room section

private void Button\_AddRoom\_Click(object sender, RoutedEventArgs e)

{

EditData(new EditRoom(null));

}

private void Button\_EditRoom\_Click(object sender, RoutedEventArgs e)

{

EditData(new EditRoom((Room)List\_Rooms.SelectedItem));

}

private void Button\_DeleteRoom\_Click(object sender, RoutedEventArgs e)

{

Room r = (Room)List\_Rooms.SelectedItem;

if (MessageBox.Show("Deleting this Room will force the deletion of " + r.Bookings.Count + " bookings.\n" +

"Please confirm this action.", "Confirm", MessageBoxButton.YesNo) == MessageBoxResult.No)

return;

using (DataRepository Repo = new DataRepository())

{

r.Bookings.ForEach(b => Repo.Bookings.Remove(b));

Repo.Rooms.Remove(r);

}

}

// Button handlers for the Period section

private void Button\_AddPeriod\_Click(object sender, RoutedEventArgs e)

{

EditData(new EditPeriod(null));;

}

private void Button\_EditPeriod\_Click(object sender, RoutedEventArgs e)

{

EditData(new EditPeriod((TimeSlot)List\_Periods.SelectedItem));;

}

private void Button\_DeletePeriod\_Click(object sender, RoutedEventArgs e)

{

TimeSlot t = (TimeSlot)List\_Periods.SelectedItem;

if (MessageBox.Show("Deleting this Period will force the deletion of " + t.Bookings.Count + " bookings.\n" +

"Please confirm this action.", "Confirm", MessageBoxButton.YesNo) == MessageBoxResult.No)

return;

using (DataRepository Repo = new DataRepository())

{

t.Bookings.ForEach(b => Repo.Bookings.Remove(b));

Repo.Periods.Remove(t);

}

}

// Button handlers for the Teacher section

private void Button\_AddTeacher\_Click(object sender, RoutedEventArgs e)

{

EditData(new EditTeacher(null));;

}

private void Button\_EditTeacher\_Click(object sender, RoutedEventArgs e)

{

EditData(new EditTeacher((Teacher)List\_Teachers.SelectedItem));;

}

private void Button\_DeleteTeacher\_Click(object sender, RoutedEventArgs e)

{

Teacher t = (Teacher)List\_Teachers.SelectedItem;

if (MessageBox.Show("Deleting this Teacher will force the deletion of " + t.Bookings.Count + " bookings and " + t.Classes.Count + " classes.\n" +

"Please confirm this action.", "Confirm", MessageBoxButton.YesNo) == MessageBoxResult.No)

return;

using (DataRepository Repo = new DataRepository())

{

t.Bookings.ForEach(b => Repo.Bookings.Remove(b));

t.Classes.ForEach(c => Repo.Classes.Remove(c));

Repo.Users.Remove(t);

}

}

// Button handlers for the Student section

private void Button\_AddStudent\_Click(object sender, RoutedEventArgs e)

{

EditData(new EditStudent(null));;

}

private void Button\_EditStudent\_Click(object sender, RoutedEventArgs e)

{

EditData(new EditStudent((Student)List\_Students.SelectedItem));;

}

private void Button\_DeleteStudent\_Click(object sender, RoutedEventArgs e)

{

Student s = (Student)List\_Students.SelectedItem;

if (MessageBox.Show("Deleting this Student will remove it from " + s.Bookings.Count + " bookings and " + s.Classes.Count + " classes.\n" +

"Please confirm this action.", "Confirm", MessageBoxButton.YesNo) == MessageBoxResult.No)

return;

using (DataRepository Repo = new DataRepository())

Repo.Users.Remove(s);

}

// Button handlers for the Department section

private void Button\_AddDepartment\_Click(object sender, RoutedEventArgs e)

{

EditData(new EditDepartment(null));;

}

private void Button\_EditDepartment\_Click(object sender, RoutedEventArgs e)

{

EditData(new EditDepartment((Department)List\_Departments.SelectedItem));;

}

private void Button\_DeleteDepartment\_Click(object sender, RoutedEventArgs e)

{

Department d = (Department)List\_Departments.SelectedItem;

if (MessageBox.Show("Deleting this Department will force the deletion of " + d.Teachers.Count + " teachers, and " + d.Rooms.Count + " rooms.\n" +

"Please confirm this action.", "Confirm", MessageBoxButton.YesNo) == MessageBoxResult.No)

return;

using (DataRepository Repo = new DataRepository())

{

d.Teachers.ForEach(t => Repo.Users.Remove(t));

d.Rooms.ForEach(r => Repo.Rooms.Remove(r));

Repo.Departments.Remove(d);

}

}

// Button handlers for the Class section

private void Button\_AddClass\_Click(object sender, RoutedEventArgs e)

{

EditData(new EditClass(null));

}

private void Button\_EditClass\_Click(object sender, RoutedEventArgs e)

{

EditData(new EditClass((Class)List\_Classes.SelectedItem));

}

private void Button\_DeleteClass\_Click(object sender, RoutedEventArgs e)

{

Class c = (Class)List\_Classes.SelectedItem;

if (MessageBox.Show("Deleting this Class will remove it from 1 teacher, and " + c.Students.Count + " students.\n" +

"Please confirm this action.", "Confirm", MessageBoxButton.YesNo) == MessageBoxResult.No)

return;

using (DataRepository Repo = new DataRepository())

Repo.Classes.Remove(c);

}

// Button handlers for the Subject section

private void Button\_AddSubject\_Click(object sender, RoutedEventArgs e)

{

EditData(new EditSubject(null));

}

private void Button\_EditSubject\_Click(object sender, RoutedEventArgs e)

{

EditData(new EditSubject((Subject)List\_Subjects.SelectedItem));

}

private void Button\_DeleteSubject\_Click(object sender, RoutedEventArgs e)

{

Subject c = (Subject)List\_Subjects.SelectedItem;

if (MessageBox.Show("Deleting this Subject will remove it from " + c.Bookings.Count + " bookings.\n" +

"Please confirm this action.", "Confirm", MessageBoxButton.YesNo) == MessageBoxResult.No)

return;

using (DataRepository Repo = new DataRepository())

Repo.Subjects.Remove(c);

}

public event PropertyChangedEventHandler PropertyChanged;

public void OnPropertyChanged(string PropertyName)

{

if (PropertyChanged != null)

PropertyChanged(this, new PropertyChangedEventArgs(PropertyName));

}

}

}

Client.MainWindow (Design)

<Window x:Class="Client.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:TimetableDisplay="clr-namespace:Client.TimetableDisplay"

Title="Room Booking System" SizeToContent="WidthAndHeight" ResizeMode="NoResize"

DataContext="{Binding RelativeSource={RelativeSource Self}}">

<Grid Margin="5,20,5,5">

<Grid.RowDefinitions>

<RowDefinition Height="Auto"/>

<RowDefinition Height="\*"/>

</Grid.RowDefinitions>

<Grid Grid.Row="0">

<Grid.ColumnDefinitions>

<ColumnDefinition Width="Auto"/>

<ColumnDefinition Width="\*"/>

<ColumnDefinition Width="Auto"/>

</Grid.ColumnDefinitions>

<Button Name="Button\_PreviousDay" Content="Previous" Width="100" Height="30" Margin="5" Grid.Column="0" Click="Button\_PreviousDay\_Click"/>

<TextBlock Text="{Binding CurrentDayString, UpdateSourceTrigger=PropertyChanged, Mode=OneWay}" Name="Text\_Day" FontSize="16" MinWidth="100" HorizontalAlignment="Center" VerticalAlignment="Center" Grid.Column="1"/>

<Button Name="Button\_NextDay" Content="Next" Width="100" Height="30" Margin="5" Grid.Column="2" Click="Button\_NextDay\_Click"/>

</Grid>

<TimetableDisplay:TimetableDisplay x:Name="Timetable" TileClicked="Timetable\_TileClicked" Grid.Row="1" />

</Grid>

</Window>



The *MainWindow*. Note that the central timetable control is the *TimetableDisplay* control explained previously – this window consists of that control, the two buttons, and the date label.

Client.MainWindow (Code-behind)

The *MainWindow* is the most commonly used window in the solution – it displays the timetable, and lets users navigate through days. The actual timetable grid is handled by the *TimetableDisplay* control, so this window handles telling the timetable to update itself and keeping track of the current day being viewed.

using System;

using System.Linq;

using System.Windows;

using System.Windows.Controls;

using System.Threading.Tasks;

using System.ComponentModel;

using Client.TimetableDisplay;

using Client.EditWindows;

using NetCore.Client;

using NetCore.Messages;

using Data.Models;

namespace Client

{

// This is the timetable window

public partial class MainWindow

: Window, INotifyPropertyChanged

{

// Connection to the server

public Connection Connection { get; set; }

// Current day being displayed

protected DateTime \_CurrentDay = DateTime.Now.Date;

public DateTime CurrentDay

{

get { return \_CurrentDay; }

set { \_CurrentDay = value; OnPropertyChanged("CurrentDay"); OnPropertyChanged("CurrentDayString"); Text\_Day.GetBindingExpression(TextBlock.TextProperty).UpdateTarget(); }

}

// Nicely formatted date

public string CurrentDayString { get { return CurrentDay.DayOfWeek + ", " + CurrentDay.ToShortDateString(); } }

// The user currently logged in

public User CurrentUser { get; private set; }

public MainWindow(Connection Connection, User CurrentUser)

{

InitializeComponent();

PropertyChanged = delegate { };

// Listen for changes to data and disconnections

DataRepository.DataChanged += Data\_DataChanged;

Connection.Disconnect += Connection\_Disconnect;

this.Connection = Connection;

this.CurrentUser = CurrentUser;

// Initialise the timetable control

Timetable.SetTimetable(CurrentUser, CurrentDay);

}

private void Connection\_Disconnect(Connection Sender, DisconnectMessage Message)

{

// If the server disconnects, close the window

Dispatcher.Invoke((Action)Close);

}

// Run when a tile is pressed on the timetable

private void Timetable\_TileClicked(TimetableTile Tile)

{

// If the current user isn't a student and either there's no booking or the teacher owns the booking

if (!CurrentUser.IsStudent && (CurrentUser.IsAdmin || Tile.Booking == null || Tile.Booking.Teacher.Id == CurrentUser.Id))

{

EditBooking Window = null;

// Whether this is a new booking or an edited one

bool NewBooking = Tile.Booking == null;

if (NewBooking) // New booking

Window = new EditBooking(CurrentUser, true, Tile.Time, Tile.Room);

else // Editing booking

Window = new EditBooking(CurrentUser, false, Tile.Booking);

Window.CurrentDate = CurrentDay;

If a tile is clicked, determine if we need to show an edit booking window – if we do, show it passing in the necessary information (the existing booking if editing one). Once the window’s closed, check to see if it wasn’t closed by cancelling, then make changes to the data repository to reflect the results of the window being shown.

// Display the window, store the result

bool? Result = Window.ShowDialog();

// If the window closed successfully

if (Result.HasValue && Result.Value)

{

// Retrieve the new item

Booking b = Window.GetItem();

if (b == null)

return;

// Are we deleting?

bool Delete = Window.DeleteBooking;

b.Id = Tile.Booking == null ? 0 : Tile.Booking.Id;

// Add or remove as appropriate

using (DataRepository Repo = new DataRepository())

{

if (Delete)

Repo.Bookings.Remove(Repo.Bookings.Where(b2 => b2.Id == b.Id).Single());

else

Repo.Bookings.Add(b);

}

}

}

}

// If data in the database changes

protected void Data\_DataChanged(Type ChangedType)

{

if (!Timetable.Dispatcher.CheckAccess()) // Wrong thread, send it to the right one

Timetable.Dispatcher.Invoke((Action<Type>)Data\_DataChanged, ChangedType);

else // Right thread, update the timetable

Timetable.SetTimetable(CurrentUser, CurrentDay);

}

protected void Button\_PreviousDay\_Click(object sender, RoutedEventArgs e)

{

// Go back a day, reload the timetable

CurrentDay = CurrentDay.AddDays(-1);

Timetable.Dispatcher.Invoke((Action<User, DateTime>)Timetable.SetTimetable, CurrentUser, CurrentDay);

}

protected void Button\_NextDay\_Click(object sender, RoutedEventArgs e)

{

// Go forward a day, reload the timetable

CurrentDay = CurrentDay.AddDays(1);

Timetable.Dispatcher.Invoke((Action<User, DateTime>)Timetable.SetTimetable, CurrentUser, CurrentDay);

}

protected void OnPropertyChanged(string PropertyName)

{

PropertyChanged(this, new PropertyChangedEventArgs(PropertyName));

}

public event PropertyChangedEventHandler PropertyChanged;

}

}

Client.TrayIcon (Design)

<Window x:Class="Client.TrayIcon"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:Client"

mc:Ignorable="d"

Title="TrayIcon" Height="0" Width="0"

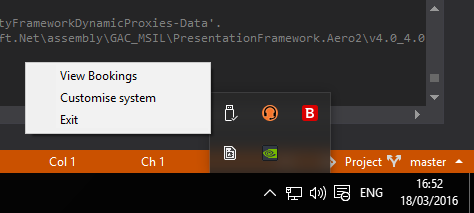
Visibility="Collapsed" WindowStyle="None"

Opacity="0" ResizeMode="NoResize" ShowInTaskbar="False" WindowState="Minimized">

<Grid>

</Grid>

</Window>



The *TrayIcon*, represented simply as a small icon in the icon tray on the taskbar. The context menu shown here is from an Admin’s logon, as it contains all the options available to users.

Client.TrayIcon (Code-behind)

The TrayIcon’s job is actually the root of the window hierarchy – the Timetable window and Admin window both open and are controlled from here, and they in turn control all their child windows.  
In essence, this is a wrapper around the Winforms implementation of a *NotifyIcon*, but provides code for ensuring correct windows are opened when clicked etc.

using System;

using System.Linq;

using System.Windows;

using System.Windows.Controls;

using System.Timers;

using NetCore.Client;

using NetCore.Messages;

using Data.Models;

using Data;

using Client.Admin;

namespace Client

{

public partial class TrayIcon

: Window

{

// Reference to the connection to the Server

public Connection Connection { get; protected set; }

// Reference to the current user logged on

public User CurrentUser { get; protected set; }

// Reference to the current room

public Room CurrentRoom { get; protected set; }

// The Winforms icon, this class is a wrapper around it

protected System.Windows.Forms.NotifyIcon ToolbarIcon { get; set; }

// The Winforms context menu attached to the icon

protected System.Windows.Forms.ContextMenu Menu { get; set; }

// The timetable window, only one allowed to be opened

protected MainWindow MainWindow { get; set; }

protected bool MainWindowShown { get; set; }

// The Admin window, only one allowed to be opened

protected AdminWindow AdminWindow { get; set; }

protected bool AdminWindowShown { get; set; }

// Timer used for booking checks

protected Timer Timer { get; set; }

// Timeslot that the program was running during in the last timer update

protected TimeSlot LastSlot { get; set; }

// How long the balloon message will linger for

protected const int MessageDuration = 5000;

public TrayIcon(Connection Connection, User CurrentUser, Room CurrentRoom)

{

InitializeComponent();

this.Connection = Connection;

this.CurrentUser = CurrentUser;

this.CurrentRoom = CurrentRoom;

Connection.Disconnect += Connection\_Disconnect;

MainWindowShown = false;

AdminWindowShown = false;

ToolbarIcon = new System.Windows.Forms.NotifyIcon();

ToolbarIcon.MouseClick += ToolbarIcon\_Click;

ToolbarIcon.Icon = Properties.Resources.ToolbarIcon;

ToolbarIcon.Visible = true;

// Construct the context menu so that it contains relevant options for the user

Menu = new System.Windows.Forms.ContextMenu();

Menu.MenuItems.Add(new System.Windows.Forms.MenuItem("View Bookings", (s, e) => ToolbarIcon\_Click(s, null)));

// Admins can customise

if (CurrentUser.Access == AccessMode.Admin)

Menu.MenuItems.Add(new System.Windows.Forms.MenuItem("Customise system", (s, e) => ShowAdminWindow()));

// Admins and teachers can exit

if (CurrentUser.Access == AccessMode.Admin || CurrentUser.Access == AccessMode.Teacher)

Menu.MenuItems.Add(new System.Windows.Forms.MenuItem("Exit", ExitClick));

ToolbarIcon.ContextMenu = Menu;

// Every 30 seconds, fire an event

Timer = new Timer(TimeSpan.FromSeconds(30).TotalMilliseconds);

Timer.Elapsed += Timer\_Elapsed;

Timer.Start();

Timer\_Elapsed(null, null); // Fire the timer event immediately

}

protected override void OnClosed(EventArgs e)

{

Timer.Stop();

Timer.Dispose();

ToolbarIcon.Visible = false;

ToolbarIcon.Icon.Dispose();

ToolbarIcon.Icon = null; // Actually hides the icon, otherwise it lingers for a bit

Menu.Dispose();

base.OnClosed(e);

}

// Displays a balloon message

public void ShowBalloon(string Title, string Message, System.Windows.Forms.ToolTipIcon Icon)

{

ToolbarIcon.ShowBalloonTip(MessageDuration, Title, Message, Icon);

}

// Every 30 seconds evaluate the bookings to see if a message needs displaying

private void Timer\_Elapsed(object sender, ElapsedEventArgs e)

{

DataSnapshot Frame = DataRepository.TakeSnapshot();

// Get the current timeslot

TimeSlot CurrentSlot = Frame.Periods.SingleOrDefault(t => t.IsCurrent(DateTime.Now));

if (CurrentSlot == null)

return;

// Get any matching bookings

Booking Booking = Frame.Bookings.SingleOrDefault(b => b.MatchesDay(DateTime.Now.Date) && b.TimeSlot == CurrentSlot && b.Rooms.Contains(CurrentRoom));

if (Booking != null)

{

// If we haven't already shown this message

if (LastSlot == null || LastSlot != CurrentSlot)

{

// Display the message

LastSlot = CurrentSlot;

ToolbarIcon.Visible = true;

ToolbarIcon.ShowBalloonTip(MessageDuration, "Scheduled booking", "A lesson is taking place in this room this period (" + CurrentSlot.Name + ").\n" +

"Teacher: " + Booking.Teacher.FormalName + "\n" +

"Subject: " + Booking.Subject.SubjectName, System.Windows.Forms.ToolTipIcon.Info);

}

}

}

// Open the window if clicked

private void ToolbarIcon\_Click(object sender, System.Windows.Forms.MouseEventArgs e)

{

if (e == null || (e != null && e.Button != System.Windows.Forms.MouseButtons.Right))

ShowMainWindow();

}

// Handle opening the window if it's closed, or bring it to the front if it's hidden

private void ShowMainWindow()

{

if (MainWindow != null && !MainWindow.Dispatcher.CheckAccess())

MainWindow.Dispatcher.BeginInvoke((Action)ShowMainWindow);

else

{

if (!MainWindowShown)

{

MainWindowShown = true;

MainWindow = new MainWindow(Connection, CurrentUser);

// Something really weird happens here - calling the MainWindow constructor causes the click event to be run again.

// Stack Trace shows it comes direct from the NotifyIcon itself, not from any accidental callbacks :/

// Some logic and flags avoids the issue, as the root cause seems to be threads updating when the new window is shown and therefore unavoidable.

MainWindow.Closed += (s, o) => MainWindowShown = false;

MainWindow.Show();

}

else

MainWindow.Activate();

}

}

// Same as ShowMainWindow but for the AdminWindow

private void ShowAdminWindow()

{

if (AdminWindow != null && !AdminWindow.Dispatcher.CheckAccess())

AdminWindow.Dispatcher.BeginInvoke((Action)ShowAdminWindow);

else

{

if (!AdminWindowShown)

{

AdminWindow = new AdminWindow(Connection, CurrentUser);

AdminWindow.Closed += (s, o) => AdminWindowShown = false;

AdminWindow.Show();

AdminWindowShown = true;

}

else

AdminWindow.Activate();

}

}

// Send a D/C message on exiting

private void ExitClick(object sender, EventArgs e)

{

Connection.Close(DisconnectType.Expected);

Environment.Exit(0);

}

// Close all windows if the server disconnects

private void Connection\_Disconnect(Connection Sender, NetCore.Messages.DisconnectMessage Message)

{

if (MainWindowShown)

MainWindow.Dispatcher.Invoke((Action)Close);

if (AdminWindowShown)

AdminWindow.Dispatcher.Invoke((Action)Close);

}

// Replace the existing show method with one that shows the icon

public new void Show()

{

if (!Dispatcher.CheckAccess())

Dispatcher.Invoke((Action)Show);

else

{

ToolbarIcon.Visible = true;

}

}

// Hides the icon, opposite of Show

public new void Hide()

{

if (!Dispatcher.CheckAccess())

Dispatcher.Invoke((Action)Show);

else

{

ToolbarIcon.Visible = false;

}

}

}

}

Client.App (Design)

<Application x:Class="Client.App"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

Startup="Application\_Startup" Exit="Application\_Exit">

<Application.Resources>

</Application.Resources>

</Application>

This is the class that defines the application resources and startup information for the program. All the .xaml code provides is the Startup and Exit handlers, which are then defined in the following code-behind.

Client.App (Code-behind)

The App class’s code behind performs a number of vital tasks. Firstly, it handles all the client-server connection code for the system, and delegates some responsibility for that to the *DataRepository* class. Secondly, it handles disconnection of the server by closing windows, displaying a message, and triggering attempts to reconnect.

using System;

using System.Windows;

using System.Threading;

using System.Threading.Tasks;

using Data.Models;

using NetCore.Client;

using NetCore.Messages;

namespace Client

{

public partial class App

: Application

{

// Reference to the conenction to the server

public Connection Connection { get; set; }

// Reference to the current user

public User CurrentUser { get; set; }

// Reference to the current room

public Room CurrentRoom { get; set; }

// Internal thread to handle connections to the server

protected Task NetTask { get; set; }

// Hides the "MainWindow" property of the base class with a TrayIcon

protected new TrayIcon MainWindow

{

get

{

// If running on a non-UI thread, invoke on the UI thread

if (!Dispatcher.CheckAccess())

return (TrayIcon)Dispatcher.Invoke((Func<TrayIcon>)(() => { return this.MainWindow; }));

else

return (TrayIcon)base.MainWindow; // Get the actual mainwindow cast to a TrayIcon

}

set

{

if (!Dispatcher.CheckAccess())

Dispatcher.Invoke((Action<TrayIcon>)(v => MainWindow = v), value);

else

base.MainWindow = value;

}

}

// On startup, load settings and start connecting

private void Application\_Startup(object sender, StartupEventArgs e)

{

if (!Settings.Load())

{

// Failed to load settings, print error

MessageBox.Show("Failed to load settings. Please contact an administrator.");

Environment.Exit(-1); // Fatal error

}

Connection = new Connection();

// Synchronously connect so that nothing happens until we're connected

NetHandler();

// Initialise the tray icon

MainWindow = new TrayIcon(Connection, CurrentUser, CurrentRoom);

// Show the logon message

MainWindow.ShowBalloon("Room Booking System started", "The Room Booking System client has started.",

System.Windows.Forms.ToolTipIcon.Info);

}

protected void NetHandler()

{

Connection.Disconnect += Connection\_Disconnect;

// Grab the address and port

string Address = Settings.Get<string>("ServerAddress");

ushort Port = Settings.Get<ushort>("ServerPort");

// Keep trying to connect

bool Connected = false;

while (!Connected)

{

// Connect with given info, store success/failure

Connected = Connection.Connect(Address, Port, new ConnectMessage(Environment.UserName, Environment.MachineName));

if (Connected)

{

// Initialise the database, store the retrieved user/room combo

Tuple<User, Room> Result = DataRepository.Initialise(Connection, new ConnectMessage(Environment.UserName, Environment.MachineName));

CurrentUser = Result.Item1;

CurrentRoom = Result.Item2;

if (CurrentUser == null) // Failed to initialise

continue; // Resume trying to connect

if (MainWindow != null)

{

// Show the tray icon again if necessary

MainWindow.Show();

MainWindow.ShowBalloon("Room Booking System started", "The Room Booking System client has started.",

System.Windows.Forms.ToolTipIcon.Info);

}

}

else

Thread.Sleep(1000); // Wait for an interval then try again

}

}

protected void Connection\_Disconnect(Connection Sender, DisconnectMessage Message)

{

// On disconnect, make sure we're invoked on the UI thread

if (!Dispatcher.CheckAccess())

Dispatcher.Invoke((Action<Connection, DisconnectMessage>)Connection\_Disconnect, Sender, Message);

else

{

Connection.Disconnect -= Connection\_Disconnect;

// Close all windows unless they're the tray icon

foreach (Window w in Windows)

{

if (w != MainWindow)

{

if (!w.Dispatcher.CheckAccess())

w.Dispatcher.Invoke((Action)w.Close);

else

w.Close();

}

}

// Only hide the tray icon, not close

MainWindow.Hide();

// Show the disconnection message

MessageBox.Show("Lost connection to the server. Will continue trying to connect in the background.");

// Restart the connection task

NetTask = Task.Factory.StartNew(NetHandler);

}

}

// Upon the client exiting, send a disconnect message

private void Application\_Exit(object sender, ExitEventArgs e)

{

try

{

Connection.Close(DisconnectType.Expected);

NetTask.Dispose();

}

catch { }

}

}

}

## Explanation of complex algorithms

Sending data over the network

Perhaps the most important part of the system is the ability for the Client and Server to communicate over the network, exchanging information about the bookings etc. Despite being extremely important, it’s surprisingly fairly simple – the more complex task is Serialisation/Deserialisation of data, which is covered in the next few sections.

Data exchange is performed over TCP/IP using the standard .NET Sockets implementation. However, the project manages to abstract away as much as possible of the underlying transport medium from the code used to actually transport data. The *Reader* and *Writer* abstract classes in the Shared assembly provide the abstract methods of reading/writing to a Stream. Code has been omitted from the following snippets, symbolised by “...”.

public abstract class Writer

: IDisposable  
{

...

public abstract void Write(byte b);

public abstract void Write(bool b);

public abstract void Write(short s);

public abstract void Write(int i);

public abstract void Write(long l);

public abstract void Write(string s);

...  
}

public abstract class Reader

: IDisposable

{

...

public abstract byte ReadByte();

public abstract bool ReadBool();

public abstract short ReadInt16();

public abstract int ReadInt32();

public abstract long ReadInt64();

public abstract string ReadString();

...

}

Both abstract classes effectively define methods to read/write each of the 6 most commonly used primitive datatypes. Subclasses then provide the specific implementations – two styles have been provided:

* NetReader and NetWriter, which write data to a Stream using nothing but the most basic capabilities of it – reading and writing sequences of bytes. These are intended primarily for use over the network, but due to the use of a Stream object, they could be used over windows Named Pipes for interprocess communication for example.
* TextReader and TextWriter, which are effectively wrappers around StreamReader and StreamWriter in the core .NET framework, and are used in read/writing to the settings files on both Server and Client applications. These work using “lines” (value are delimited using the newline character), whereas the previous implementation uses byte lengths to determine values.

TextWriter and TextReader are the simplest of the implementations. Of the items that can be written, there are only 4 “styles” of writing data – writing a *byte*, a *bool*, a *string*, and an *int*. *int*s, *short*s and *long*s are all written using the same methods, so only the *int* implementation has been included for brevity:

public class TextWriter

: Writer

{

...  
 protected StreamWriter Writer { get; set; }  
  
 public override void Write(byte b)

{

Writer.WriteLine(b);

}

public override void Write(bool b)

{

Writer.WriteLine(b);

}  
 public override void Write(int i)

{

Writer.WriteLine(i);

}

public override void Write(string s)

{

Writer.WriteLine(s);

}

...

}

public class TextReader

: Reader

{

...  
 protected StreamReader Reader { get; set; }

public override byte ReadByte()

{

return (byte)Reader.Read();

}

public override bool ReadBool()

{

return bool.Parse(Reader.ReadLine());

}

public override int ReadInt32()

{

return int.Parse(Reader.ReadLine());

}

public override string ReadString()

{

return Reader.ReadLine();

}

...

}

As can be seen, this implementation is rather trivial – StreamWriter and StreamReader provide basically the same abilities as this class, the only reason this implementation is provided is so that the Settings files can be manipulated using the same inheritance hierarchy as other IO operations. This would allow using binary files instead of text files to store settings very easily, as the output operations being performed are independent of which of the two implementations is actually used.

The NetWriter and NetReader implementations are slightly more complex. One major difference is that we need to take into account endianness – all multi-byte values need to be converted into network-byte-order, then when received converted back into host order:

public class NetWriter

: Writer

{

...

public virtual void Write(byte[] Data)

{

Base.Write(Data, 0, Data.Length);

}

public override void Write(byte b)

{

Write(new byte[] { b });

}

public override void Write(bool b)

{

Write(new byte[] { Convert.ToByte(b) });

}

public override void Write(int i)

{

Write(BitConverter.GetBytes(IPAddress.HostToNetworkOrder(i)));

}

public override void Write(string s)

{

Write(Encoding.BigEndianUnicode.GetByteCount(s));

Write(Encoding.BigEndianUnicode.GetBytes(s));

}

...

}

public class NetReader

: Reader

{

...

public virtual byte[] ReadBytes(int Count)

{

...

byte[] Buffer = new byte[Count];

int Remaining = Count;

// Not guaranteed to read all bytes on first try - retry until all read

while (Remaining > 0)

Remaining -= Base.Read(Buffer, Buffer.Length - Remaining, Remaining);

...

return Buffer;

}

public override byte ReadByte()

{

return ReadBytes(1)[0];

}

public override bool ReadBool()

{

return Convert.ToBoolean(ReadByte());

}

public override int ReadInt32()

{

return IPAddress.NetworkToHostOrder(BitConverter.ToInt32(ReadBytes(sizeof(int)), 0));

}

public override string ReadString()

{

int Length = ReadInt32(); // Read the length of the data first

return Encoding.BigEndianUnicode.GetString(ReadBytes(Length));

}

...

}

Writing data is more complicated now – a new function is definined that takes an array of bytes and writes them to the stream, which is used as a utility function by all the others. This reduces the tasks of the other functions to simply converting their parameter into an array of bytes.

* To send a single byte we simply create a new byte array with a single element – the byte to be sent.
* To send a Boolean, we convert the bool into a byte, then create the array containing that item.
* To send an int, we first convert it into network order (big-endian), then use BitConverter to get the bytes representing it, and send those.
* To send a string, we send the length of the string first (using the overload of write that takes an integer, and making sure to convert to network order), and then use a big-endian encoding format to get the bytes representing the string itself, which is then sent.

Reading is more complex again – for a variety of reasons, a call to Read on a stream may not retireve the required number of bytes of data (delayed packet, IO socket was closed etc). As a result, we create a buffer (byte array) of the desired size, and read data into it until it’s full. Reading individual types of data is then reduced to reading an array of the desired size, then converting it from a byte array into the output type:

* To read a single byte, we read an array of size 1 and return the 0th element.
* To read a bool, we read a byte then convert it into a Boolean.
* To read an int, we read *sizeof(int)* (equal to 4 on most systems) bytes into an array, then use BitConverter to convert the array into an integer, before finally switching it back to host order.
* To read a string, we first read an integer (using the previously defined function) that denotes the string length, then read the specified number of bytes and use a big-endian encoding format to convert the array into a string again.

NetReader also provides an assortment of asynchronous methods for reading data concurrently, but I won’t cover those here –they use almost exactly the same methods as the synchronous ones covered above, but modified slightly.

Using this abstract framework for IO to a stream, all that needs to be provided to the serialise/deserialise functions (covered below) on objects that need to be sent over the network is an abstract Reader/Writer object, which was previously given a NetStream or FileStream to work with, and data can be transmitted. This is a very tidy implementation, keeping things abstracted and using plenty of code re-use.

Serialisation/Deserialisation (General)

Serialisation/deserialisation is one of the more important parts of the system, as it lets the messages and entities be sent from the Client to the Server and vice versa. The idea behind this was briefly covered in the “Processing and Algorithms” subsection of the design section, but samples from the code will be provided here.  
The Serialisable system is a set of classes and methods that aid the development of other components that require serialisation. These are grouped into two hierarchies or trees: the Messages hierarchy, which is all under the NetCore assembly and is concerned with sending message between client and server, and the DataItem branch, which is under the Data assembly and is used to represent items in the database. These two hierarchies work together to allow sending both “normal” messages, containing useful information such as connection/disconnection information, and “data” messages, containing changes to the database.

The starting point of the entire serialisable system is the *Shared.ISerialisable* interface:

public interface ISerialisable

{

void Serialise(Writer Out);

void Deserialise(Reader In);

}

This defines two simple methods which must be implemented by all classes inheriting this interface. Using this means that *ISerialisable* objects can be passed around (in parameters and such) and can be Serialised/Deserialised without actually needing to know the type of the object, which can be useful.

The next class in the hierarchy is the *Message*:

public abstract class Message

: ISerialisable

{

...

public virtual void Serialise(Writer Writer)

{

// Send a single byte as a notification

Writer.Write((byte)0);

Writer.Write(GetType().Name);

}

public abstract void Deserialise(Reader Reader);

...  
}

This class is abstract, and uses this property to implement Deserialise as required by the interface, but leaves it abstract. This means that subclasses must provide their own implementation of it. Serialise is also implemented, but virtually rather than abstractly. This allows subclasses the ability to override it, but also provides some functionality that they’ll want to use.  
The Deserialise function obviously needs no explaining as there’s no code associated with it. Serialise however is very important – it sends a single notification byte followed by the full type name of the class. It should be noted that the type name isn’t “*NetCore.Messages.Message*”, as might first be assumed, rather the type name of the class that provides the implementation that then uses this function (so the subclass “TestMessage”, when calling this function, would have a type name of “*NetCore.Messages.TestMessage*”. These two values are essential – the **notification byte** signals the receiver that a message is about to be serialised, whilst the **type name** allows the receiver to determine which type of message is being sent, and thus deserialise it correctly.

To demonstrate why these two bits of data are needed, the following is the main snippet from the code used to read a message. This is called after the receiver’s received the notification byte:

public static Message ReadMessage(NetReader Reader)

{

...

string Id = Reader.ReadString(); // Read the type

foreach (Type t in MessageTypes)

{

if (t.Name == Id) // Found the right class

{

// Create an object of the class, deserialise to it and return

Message m = (Message)Activator.CreateInstance(t);

m.Deserialise(Reader);

return m;

}  
 }

...

}

The *Id* (full type name) is read from the stream, then a loop is run over every *Type* of message in the assembly, comparing their full type name to the type name received. If there’s a match, an instance of the class denoted by the type name is created, then the sent message is deserialised from the stream by it, and the message returned. This is an elegant solution to being able to deal with multiple types of message without needing explicit if/else if statements to determine the type.

The missing piece now is how the messages are actually serialised to the stream – the advantage of the object-oriented design is that adding new messages is very easy. Here’s some example code from the *ConnectMessage* class:

public class ConnectMessage

: Message

{

public string Username { get; protected set; }

public string ComputerName { get; protected set; }

...

public override void Serialise(Writer Writer)

{

base.Serialise(Writer);

Writer.Write(Username);

Writer.Write(ComputerName);

}

public override void Deserialise(Reader Reader)

{

Username = Reader.ReadString();

ComputerName = Reader.ReadString();

}

}

There are two properties of this class that need serialising/deserialising – *Username*, and *ComputerName*. All that needs to be done in each function is to either write the object or read the object from the provided streams. Note that *Serialise* calls its base classes’ implementation of Serialise as well. This is the Serialise function shown near the top of this section, which is necessary as it allows for the message to be deserialised at all.

A more complex example of a message is the *DataMessage*, responsible for sending any type of *DataModel* (so a *Booking*, *Room*, *Student* etc). Despite being required to send a variety of quite different objects, it is very easy to do so with this design:

public class DataMessage

: Message

{

public DataModel Item { get; set; }

public bool Delete { get; set; }

...

public override void Serialise(Writer Out)

{

base.Serialise(Out);

Out.Write(Delete);

Item.Serialise(Out);

}

public override void Deserialise(Reader In)

{

Delete = In.ReadBool();

Item = DataModel.DeserialiseExternal(In);

}

}

Because *DataModel* implements *ISerialisable*, it is trivial to encapsulate an object that inherits from *DataModel* within this *ISerialisable* object. When we need to serialise the message, we serialise all the fields normally except for the *DataModel*, which we call Serialise on in turn to encode it to the stream as well. When Deserialising, the process is very similar bar the call to DeserialiseExternal, which performs much the same task as the *Message* classes’ *ReadMessage* – it returns a *DataModel* that was sent, working out from sent data what specific type of object was meant (Booking, Room etc). This is a good example of the recursive nature of the serialisation process.

The following diagram provides another way of looking at how messages are constructed and sent:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Signal | Full Type Name | Message Fields | | Sub-fields that are serialisable |
| 0 | NetCore.Messages.ConnectMessage | Username (string) | ComputerName (string) |  |
| 0 | NetCore.Messages.DataMessages.DataMessage | Delete (Bool) | | Item (DataModel) |

Exactly the same design pattern has been applied to the *DataModel* hierarchy as to the *Message* one – the use of abstracted components (such as the *ISerialisable* interface) have helped reduce code bloat and made refactoring easier. The design of the system itself has made the code using it very easy to write, as the ability to reuse existing functionality defined by superclasses is used to full effect. And finally, satisfyingly, the two hierarchies of serialisable items work together perfectly, as Messages can be used to transport DataItems.

Another handy design result is that each function overriding Serialise/Deserialise only performs the operation on the data in the class it’s defined in – each Message subclass handles only the fields specified in the class. Even the root classes of the trees, Message and DataItem, both only actually manipulate variables that are defined in their respective classes. This creates a nicely encapsulated system without cross-dependencies in code.

All that’s required to send data over the network now is to use a NetworkStream object wrapped around a Socket that’s connected to the client/server. Creating any of the above implementations with this stream as a parameter will allow for easy communication over the network using either format.

Serialisation/Deserialisation (Specifically DataModels)

H:\Burford\Year 13\Computing\Project\_Writeup\Resources\Maintenance\Explanation of Algorithms\Circular Serialisation Demo.pngWhen Serialising/Deserialising DataModels, there’s a catch – each one has a reference to multiple other items, which in turn have references to more, and so on. In fact, if one datamodel has a reference to another, then due to the way references are implemented in Entity Framework, the second object has a reference to the first as well. If Datamodels were serialised in the same way as messages, by telling all referenced datamodels to serialise themselves as well, due to the cyclical dependencies, a *StackOverflowException* would be thrown as each object tries to serialise.

Attempting to Serialise a Booking object causes the Subject reference within it to try to Serialise the Booking object again.

In order to fix this issue, when Serialising a datamodel only the Id of its references to other datamodels are actually sent – on the receiving end, the Id of the referenced item is read, and from this a reference to the appropriate item in the database acquired. Also, when dealing with many-to-(many/one) relationships, where there are a number of references to send, the number of references are sent first, followed by each Id. The following code snippets from the *Class* and *Student* classes demonstrate this approach:

public class Class

: DataModel  
{  
 ...

public override void Serialise(Writer Out)

{

base.Serialise(Out);

Out.Write(ClassName);

Out.Write(Owner.Id);

Out.Write(Students.Count);

Students.ForEach(s => Out.Write(s.Id));

}

protected override void Deserialise(Reader In)

{

base.Deserialise(In);

ClassName = In.ReadString();

Owner = new Teacher() { Id = In.ReadInt32() };

Students = new List<Student>(In.ReadInt32());

for (int x = 0; x < Students.Capacity; x++)

Students.Add(new Student() { Id = In.ReadInt32() });

}  
 ...  
}

public class Student

: User

{  
 ...  
 public override void Serialise(Writer Out)

{

base.Serialise(Out);

Out.Write(Year);

Out.Write(Form);

Out.Write(Classes.Count);

Classes.ForEach(c => Out.Write(c.Id));

Out.Write(Bookings.Count);

Bookings.ForEach(b => Out.Write(b.Id));

}

protected override void Deserialise(Reader In)

{

base.Deserialise(In);

Year = In.ReadInt32();

Form = In.ReadString();

Classes = Enumerable.Repeat(new Class(), In.ReadInt32()).ToList();

Classes.ForEach(c => c.Id = In.ReadInt32());

Bookings = Enumerable.Repeat(new Booking(), In.ReadInt32()).ToList();

Bookings.ForEach(b => b.Id = In.ReadInt32());

}  
 ...  
}

As can be seen, primitive fields are simply written to the stream, while the collections of references are serialised by writing the number of items, followed by iterating over each item and outputting the Id.  
Deserialisation uses slightly more complex code, using Enumerable.Repeat to initialise the collection to the correct size, then using a LINQ statement to assign the Id of each created object with an Id received over the network.

This section has effectively only covered how to send the bare bones of the datamodels across – once they’ve been received, additional work must be done to establish the references from the supplied IDs. This is the subject of the next section.

Establishing references on the client side – Expanding, Attaching, and Detaching

Once a datamodel has been deserialised, all that’s known about it on the client side is the Id. The data about the item is stored in the DataRepository, as the *InitialiseMessage* sent when the client first connects holds all the data about item in the system, but the received item needs to be matched up to the actual full item in the database.

In order to accomplish this, some more abstract methods are defined in the *DataModel* class:

public abstract class DataModel

{

...

public abstract bool Expand(IDataRepository Repo);

public abstract void Attach();

public abstract void Detach();

...

}

These functions have very specific tasks:

* *Expand* uses the data repository provided to reach out to the related items (other DataModels) and obtain references to **them**. This allows other objects to access this object’s related entities. A way of imagining this function’s purpose is to see it expanding into the network of entities in the repository like a web, and linking to the relevant ones.  
  Note that this functions takes an *IDataRepository* – as the client and server both provide different implementations of a DataRepository but both need to be able to use this function, this interface has been created to allow access to the necessary fields while not depending on either implementation.
* Attach is called after Expand, and sets the **other** related objects references to **this** object. This allows other objects to access this object through the related entities. This is the mirror image of Expand, and ensures that the two-way references exist.
* Detach is called when the entity is being removed from the system, and removes references to **this** object from the **other** objects. This effectively removes the two-way connection, and cleans up the system so that there aren’t any “bad” references around (no references to objects already removed).

As these functions are abstract, an implementation must be provided by each subclass. An example is provided by the *Student* class:

public class Student

: User

{  
 ...

public override bool Expand(IDataRepository Repo)

{

try

{

for (int x = 0; x < Classes.Count; x++)

Classes[x] = Repo.Classes.SingleOrDefault(c => c.Id == Classes[x].Id);

for (int x = 0; x < Bookings.Count; x++)

Bookings[x] = Repo.Bookings.SingleOrDefault(b => b.Id == Bookings[x].Id);

}

catch { return false; }

return true;

}

public override void Attach()

{

Bookings.ForEach(b => b.Students.Add(this));

Classes.ForEach(c => c.Students.Add(this));

}

public override void Detach()

{

Bookings.ForEach(b => { if (b != null) b.Students.RemoveAll(i => i.Id == Id); });

Classes.ForEach(c => { if (c != null) c.Students.RemoveAll(i => i.Id == Id); });

}

...

}

As can be seen, even though the *Student* class has multiple fields such as *FirstName*, *LogonName* etc, these functions only deal with the other datamodel fields – effectively the relationships in the database.

* *Expand* loops over each item in the collections of relationships and re-assigns each entry to the object in the database with a matching Id. This establishes the reference to the other items. If anything goes wrong (item isn’t in the database), an exception is thrown and the function returns false.
* *Attach* loops over each item (using LINQ this time as the collection itself isn’t modified unlike the previous function), and adds a reference to this item to their collections of relations. This is the second step of establishing the two-way relations between items.
* *Detach* loops through all the items in the collections of relations and if the item isn’t null, removes the students from their colllections of references that point back to this student. This removes the two-way linx between items.

When a *DataModel* is received (on either the Client or Server), these functions are used to link the object into the database seamlessly, or remove it when necessary.

# User Manual

# Appraisal

## Objectives

This is the list of original numbered objectives, with colour codes for status (complete, incomplete, cancelled, added) and descriptions of the final result.

| # | Original Objective | Result |
| --- | --- | --- |
|  | Create a program that will allow 3 layers of access to the system for the 3 different users. | Complete – there are 3 levels of access, namely Student, Teacher and Admin, and they are only able to access restricted areas of the system. |
|  |
|  | Allow users to view bookings over different days/months etc. | Complete – on the main timetable window, users can cycle through days. |
|  |
|  | Allow teachers to view and book specific rooms around the school site. | Complete – administrators can add/edit/remove Rooms which are then available for booking by teachers/admins. |
|  |
|  | Allow students to view bookings for rooms around the school site. | Complete – students can see all bookings using the main timetable window. |
|  |
|  | Prevent unauthorised access to the system for non- students/teachers/admins. | Complete – if a user with a logon that’s not registered in the system is detected, the program simply won’t start. |
|  |
|  | Allow administrators to override bookings | Complete – administrators have complete control over bookings and can edit them entirely. |
|  |
|  | Allow administrators to revoke student access to viewing bookings. | Not complete – there would be no need to restrict a student’s ability to see bookings. If this is desired, removing the student from the system has the same effect. |
|  |
|  | Provide an easy to use interface, so that students can use it without training, and teachers can use it with minimal training. | Complete - user feedback has been taken and shows positive results – questionnaires and an analysis of the results is included below this section. |
|  |
|  | Provide clear and useful error messages that should give teachers an idea of what went wrong, not just that something did go wrong. | Complete – all error messages describe the problem and most give brief instructions on how to fix it. |
|  |
|  | Allow teachers to set up recurring bookings on varying schemes (daily, weekly, monthly). | Complete – teachers can set bookings to recur daily, weekly, fortnightly, or 4-weekly. |
|  |
|  | Provide checks to ensure users aren’t making mistakes when editing bookings. | Complete – full validation is in place on all input fields to restrict input to only valid data. There is no verification in place. |
|  |
|  | Provide a warning to users when logging onto a computer in a room that’s been booked for that period. | Complete – logging on in a room where a Booking’s scheduled to occur causes a balloon-style alert from the taskbar, with information on the booking. |
|  |
|  | Provide email notifications to teachers when they book/cancel a room. | Complete – when a booking is changed, an email is sent to the email address of the teacher who “owns” the Booking. |
|  |
|  | Provide useful statistics to teachers about room bookings, including per-teacher and per-department information. | Incomplete – not difficult to do, but testing all the functionality as well as designing new user interfaces and testing that would have taken too long. |
|  |
|  | Allow data to be added in bulk, using CSV files or possibly Active Directory services. | Incomplete – performing bulk addition of data proved more complicated than originally thought due to data having links that aren’t easily represented in a CSV. |
|  |
|  | Allow the clients to handle loss of connection to the server gracefully, and restore connection and services as soon as possible. | Complete – if the server disconnects, clients immediately hide all open windows, show an error message, then prevent any input until connection can be re-established. |
|  |
|  | Securely hold records of all bookings. | Complete – the data is held in a single database file on the school server, which is already secured. Access to the system can only be gained using a school user logon, which is a layer of access control. |
|  |
|  | Allow bookings to be backed up easily. | Complete – all system data is held in a single database file, and application settings are held in individual settings files in the same directory as the executables. This makes it very easy to backup data. |
|  |
|  | Allow the server to be turned on and off with minimal interruption to service (ie doesn’t “forget” bookings). | Complete – server reloads all data from the database on startup and flushes all data to the database when shut down, so no data is lost. |
|  |
|  | Allow new bookable rooms to be added (by administrators) with ease. Immediate availability of the new room for booking is not required, but is preferred. | Complete – all aspects of the system can be customised, adding new Rooms, Students, Teachers, Departments etc. |
|  |
|  | Allow teachers to sort students by various criteria when searching for them to add to the list of students in a Booking. | Complete – the filter allows sorting by name, year, form, and a variety of other fields. The filtering process is fast, responsive, and easy to use. |
|  |
|  | Allow for the creation of “Classes”, groups of students, that can be used to quickly and easily find students to add to a Booking. | Complete – classes can be created to group students together, and is integrated into the filter functionality to allow a filter to be applied to a class. |
|  |
|  | Allow for the creation of Subjects, to let users easily identify the which type of Booking has been made using just the colour. | Complete – subjects can be created by admins and Bookings can be tagged with a subject which changes the colour of the tile on the display. |
|  |
|  | Provide per-application settings for both Clients and the Server, allowing customisation of core features. | Complete – “Settings.txt” files in the directory of each executable allows for adjustments to the basic functionality of programs. |
|  |
|  | Allow a Booking to span multiple rooms, to reduce the number of separate bookings that need to be made by teachers. | Complete – a list of rooms is provided when creating a booking, in the same style as the students list. Common UI design means it’s easier to use. |
|  |

As can be seen, the vast majority of features have been realised, with quite a few additional ones being implemented as well (shown at the bottom of the list). Of the three that haven’t been met, the first (revoking student access) was discounted shortly after production was started as we realised that the feature it added was unnecessary, and if it truly was needed, removing the student from the system would server exactly the same purpose. The second, providing useful statistics, is incomplete as I ran out of time during development to implement it. The feature isn’t particularly complex, but it would take too long to develop the user interface and perform tests. Similarly the third, adding bulk data using CSV files or similar proved more complex than initially thought, and had to be cut in order to focus development on more essential features.

## Client Feedback

After the system was developed and tested, I did an interactive demonstration of the system with the client, Mr Wilsdon (network manager). This is his feedback:  
  
Hi Keith,  
Many thanks for the development of your Room Booking System. It looks impressive and meets most of the objectives set out.

One aspect I would like to see changed is the calendar navigation. Currently the system uses day to day navigation, by using day to day navigation users will find it hard to make advanced bookings. If you could change the date navigation to a date picker control this would greatly improve the date navigation.

It would also be nice if you could add a rapid import/export function for the booking data. This would speed up the input of fixed timetable bookings.

M. Wilsdon

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

26/2/2016

## Questionnaires

Student

|  |  |  |
| --- | --- | --- |
| H:\Burford\Year 13\Computing\Project\_Writeup\Resources\Appraisal\Questionnaires\Student-1.jpg | H:\Burford\Year 13\Computing\Project\_Writeup\Resources\Appraisal\Questionnaires\Student-2.jpg | H:\Burford\Year 13\Computing\Project\_Writeup\Resources\Appraisal\Questionnaires\Student-6.jpg |
| H:\Burford\Year 13\Computing\Project\_Writeup\Resources\Appraisal\Questionnaires\Student-9.jpg | H:\Burford\Year 13\Computing\Project\_Writeup\Resources\Appraisal\Questionnaires\Student-10.jpg | H:\Burford\Year 13\Computing\Project\_Writeup\Resources\Appraisal\Questionnaires\Student-7.jpg |

Student feedback was pleasingly positive – 100% of students who tested the system found it easy to use and understand, as well as being responsive and fast. The only complaint lodged was that it was hard to find the program in the icon-tray on the taskbar, as the balloon tip wasn’t enough of a clue as to the program’s location.  
This is a good overall result, as it shows the research done during the analysis stage yielded useful results – on the preliminary questionnaire in the Analysis stage, students found the existing school timetable layout easy to use, which lead to me designing the timetable UI in the same layout, which has finally resulted in students liking the UI design. In additition, the one complaint is a fairly easy-to-fix feature.

Teacher

|  |  |  |
| --- | --- | --- |
| H:\Burford\Year 13\Computing\Project\_Writeup\Resources\Appraisal\Questionnaires\Teacher-5.jpg | H:\Burford\Year 13\Computing\Project\_Writeup\Resources\Appraisal\Questionnaires\Teacher-1.jpg | H:\Burford\Year 13\Computing\Project\_Writeup\Resources\Appraisal\Questionnaires\Teacher-4.jpg |

Again, feedback from teachers was positive – all teachers found it easy to make bookings and, interestingly, all located the client running in the icon tray a lot faster than students did. This is likely due to the fact that there are a few other programs teachers use which have an icon in the same place, so it’s more natural for teachers to look there than it is for students.  
Regardless, all teachers found the system easy to use and fast, and understood all error messages/notifications that were presented to them. Some specific features which were praised included the use of “Empty” to denote a tile without a Booking, as it made it obvious that something *could* be booked. The use of drop-down menus were also popular, as it made it simpler to edit fields as all the data was clearly displayed for selection.

Admin

|  |
| --- |
| H:\Burford\Year 13\Computing\Project\_Writeup\Resources\Appraisal\Questionnaires\Admin-1.jpg |

Administrator feedback emphasised the points raised in the Client Feedback section, and fed into the list of Extensions below. Overall feedback was positive – the program itself was deemed well made.

## Extensions

I have drawn up a list of improvements that should be made as well as initial ideas about how to go about implementing the changes. These have been inspired by various sources, mostly by the Client feedback and user feedback during user testing, but also by features which I’ve thought of while developing the system which would have taken too long/be too far outside the scope of the objectives I’d already set to justify developing them.

1. Adding a “longer-range” date selection feature. The existing method only allows for single day increment/decrements to the date, whereas it would be more useful to be able to skip to a day at some arbitrary point in the future.  
   This would be very easy to implement, all it requires is adding a Datepicker control to the “MainWindow” design/codebehind rather than the next/previous day buttons, and adding event handlers so that when a new date is selected on it, the timetable’s updated. Setting the timetable to a specified day (*TimetableDisplay.SetTimetable*) already takes a given date to display as a parameter, so no extra code needs to be written to allow for jumping to a random day instead of changing by one.
2. Fully implementing the statistics feature as was originally planned. This would again be easy to implement, all that would be required to calculate the statistics are a few database queries. The majority of implementation time would be taken up developing a User Interface that allows for a range of stats to be viewed in a natural manner, as well as testing this window. Ideally, further research would be done to discover what types of statistce are most useful, in order to further improve the system. For example, the IT technicians may find it really useful to know which rooms are most often booked, so they can anticipate which computers are most likely to break frequently, while heads of departments may find it useful to see which teachers in their department booked the most rooms etc.
3. Adding more obvious instructions on how to open the application – a few students complained during the user feedback sessions that it wasn’t obvious how to open the application, despite a balloon tooltip showing up on the taskbar leading them to the icon in the icon tray.  
   This could be very simple to fix, all it will require is changing the initial balloon message to give further instructions on how to open the application.  
   Interestingly, no teachers complained of this – this is likely due to the fact that teachers already use programs which have icons in the icon tray, whereas students don’t.
4. Providing a method of bulk adding/removing data from the system. At the moment data has to be entered on a per-entity basis. While this is appropriate for most entity types (eg a Booking has many relations to other entities so it would be very difficult to add multiple Bookings without individual customisation), for others such as Students and Teachers, this is perhaps the greatest shortcoming of the solution.  
   Ideally, an option to load in data about students or teachers from a variety of different mediums would be supported, from CSV files to perhaps even Active Directory information on the network. This import option would make it far easier for admins to make large adjustments to the system.
5. Use the school’s mail server rather than a public one. At the moment, the Google SMTP server is used to send emails, as it was easier to test that than develop it to use the school’s internal one; as I was developing and testing this solution at home as well as in school, using a third party server was the only way I could implement this feature so that it worked everywhere.  
   Settings would likely be provided to customise the connection, such as the SMTP server’s IP address, port number, security settings etc., and could easily be added as another part of the Admin Control Window.
6. Providing facilities to customise the notification emails being sent, akin to “mailmerge”. At the moment, emails are sent using a hardcoded set of strings with variables concatenated in, but it would be a very useful feature to be able to create templates for emails.  
   It would be a fairly substantial task to set up, but once the window and associated logic has been completed it could easily be added to the Admin Control Window for easy access.
7. General UI improvements. More tooltips/help messages can always be added, and in a system like this where it would be highly advantageous for users to be able to solve their own problems, easy access to comprehensive help documentation is very desirable. In addition, the layouts of some windows could likely be improved, such as the Edit Windows which have a very uniform field layout.