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



|  |
| --- |
|  |

Contents

[Analysis 4](#_Toc443324054)

[Client 4](#_Toc443324055)

[Background, Description of Current System 4](#_Toc443324056)

[Identification of Users 4](#_Toc443324057)

[Limitations 5](#_Toc443324058)

[Data Sources/Destinations – Existing System 5](#_Toc443324059)

[Data Sources/Destinations – Proposed Solution 6](#_Toc443324060)

[Data Volumes 6](#_Toc443324061)

[Data Dictionaries 7](#_Toc443324062)

[Student 7](#_Toc443324063)

[Teacher 7](#_Toc443324064)

[Room 7](#_Toc443324065)

[Class 8](#_Toc443324066)

[Department 8](#_Toc443324067)

[Data Flow Diagram – Current System 8](#_Toc443324068)

[Objectives 9](#_Toc443324069)

[Potential Solutions 10](#_Toc443324070)

[Appraisal of Potential Solutions 10](#_Toc443324071)

[Justification of Chosen Solution (Client-Server) 11](#_Toc443324072)

[Entity-Relationship Diagrams 11](#_Toc443324073)

[System Flow Chart 11](#_Toc443324074)

[Fact Finding Methods 12](#_Toc443324075)

[Interview with Mr Wilsdon, Client 12](#_Toc443324076)

[Paperwork from current system 13](#_Toc443324077)

[Observation of current system 13](#_Toc443324078)

[Questionnaires 18](#_Toc443324079)

[Benefits of the proposed system 18](#_Toc443324080)

[Design 19](#_Toc443324081)

[Overall System Design 19](#_Toc443324082)

[Solution Limitations 19](#_Toc443324083)

[Description of Modular Structure 19](#_Toc443324084)

[Client 19](#_Toc443324085)

[Server 19](#_Toc443324086)

[NetCore 19](#_Toc443324087)

[Data 20](#_Toc443324088)

[Shared 20](#_Toc443324089)

[Processing and Algorithms 20](#_Toc443324090)

[Client 20](#_Toc443324091)

[Server 21](#_Toc443324092)

[Storage Material and Format 21](#_Toc443324093)

[Human-Computer Interface 22](#_Toc443324094)

[Timetable Window 22](#_Toc443324095)

[EditBooking Window 23](#_Toc443324096)

[Admin Controls Window 23](#_Toc443324097)

[Sample of Data Entry 24](#_Toc443324098)

[Sample of Valid Output 24](#_Toc443324099)

[Security & Integrity 24](#_Toc443324100)

[System Security 24](#_Toc443324101)

[Test Plan 24](#_Toc443324102)

[Testing 25](#_Toc443324103)

[Test Plan 25](#_Toc443324104)

[Test Results - Evidence 32](#_Toc443324105)

[Minimum Test Data 43](#_Toc443324106)

[System Maintenance 44](#_Toc443324107)

[Introduction 44](#_Toc443324108)

[Description of Modular System 44](#_Toc443324109)

[Shared 45](#_Toc443324110)

[Shared.Writer 46](#_Toc443324111)

[Shared.TextWriter 47](#_Toc443324112)

[Shared.NetWriter 48](#_Toc443324113)

[Shared.Reader 49](#_Toc443324114)

[Shared.TextReader 50](#_Toc443324115)

[Shared.NetReader 51](#_Toc443324116)

[Shared.Helpers 54](#_Toc443324117)

[Shared.ISerialisable 55](#_Toc443324118)

[Data 56](#_Toc443324119)

[Data.IDataRepository 58](#_Toc443324120)

[Data.DataSnapshot 59](#_Toc443324121)

[Data.Models.DataModel 61](#_Toc443324122)

[Data.Models.User 63](#_Toc443324123)

[Data.Models.Student 65](#_Toc443324124)

[Data.Models.Teacher 67](#_Toc443324125)

[Data.Models.Department 69](#_Toc443324126)

[Data.Models.Room 71](#_Toc443324127)

[Data.Models.Class 74](#_Toc443324128)

[Data.Models.Subject 76](#_Toc443324129)

[Data.Models.TimeSlot 78](#_Toc443324130)

[NetCore 82](#_Toc443324131)

[NetCore.BlockingQueue 83](#_Toc443324132)

[NetCore.Messages.Message 84](#_Toc443324133)

[NetCore.Messages.TestMessage 86](#_Toc443324134)

[NetCore.Messages.ConnectMessage 87](#_Toc443324135)

[NetCore.Messages.DisconnectMessage 88](#_Toc443324136)

[NetCore.Messages.DataMessages.InitialiseMessage 89](#_Toc443324137)

[NetCore.Messages.DataMessages.UserInformationMessage 90](#_Toc443324138)

[NetCore.Messages.DataMessages.DataMessage 91](#_Toc443324139)

[NetCore.Server.Client 92](#_Toc443324140)

[NetCore.Server.Listener 95](#_Toc443324141)

[NetCore.Client.Connection 99](#_Toc443324142)

# Analysis

## Client

Client: Burford School

Client Liaisons:

* Mr Wilsdon (Network Manager), for technical information and access to existing system.

## Background, Description of Current System

At our school we have quite a few 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 connection to be disrupted/the website to be shutdown, the school’s booking system would be rendered non-functional. Backups are taken and dealt with by the external company, but the school must pay a subscription fee.

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 lives.

The solution will also 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.

## 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.

## Limitations

* 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”.

## Data Sources/Destinations – Existing System

|  |  |  |
| --- | --- | --- |
| 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 Sources/Destinations – Proposed Solution

|  |  |  |
| --- | --- | --- |
| 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 more than 100bytes each. This means that for every 10,000 records the file will increase by 1MB, which is 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.

## Data Dictionaries

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 |

## 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.



## 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:

1. Create a client that will allow 3 layers of access to the system for the 3 different users.
2. Allow users to view bookings over different days/months etc. (and possibly limit the scope of viewing for students).
3. Allow teachers to view and book specific rooms around the school site.
4. Allow students to view bookings for rooms around the school site.
5. Prevent unauthorised access to the system for non- students/teachers/admins.
6. Allow administrators to override bookings
7. Allow administrators to revoke student access to viewing bookings.
8. Provide an easy to use interface, so that students can use it without training, and teachers can use it with minimal training.
9. Provide clear and useful error messages that should give teachers an idea of what went wrong, not just that something did go wrong.
10. Allow teachers to set up recurring bookings on varying schemes (daily, weekly, monthly).
11. Provide checks to ensure users aren’t making mistakes when editing bookings.
12. Provide a warning to users when logging onto a computer in a room that’s been booked for that period.
13. Provide email notifications to teacher when they book/cancel a room and when a booking is authorised.
14. Provide useful statistics to teachers about room bookings, including per-teacher and per-department information.
15. Allow the clients to handle loss of connection to the server gracefully, and restore connection and services as soon as possible.
16. Allow students to view where their friends are logged on, provided their friends have marked that they want to be “visible” to that student.
17. Securely hold records of all bookings.
18. Allow the server to be turned on and off with minimal interruption to service (ie doesn’t “forget” bookings).
19. 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.

## 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 only consider 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. However, a website has no way of viewing where people are logged in, so this wouldn’t solve the main problems of students being “disconnected” from the system.
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

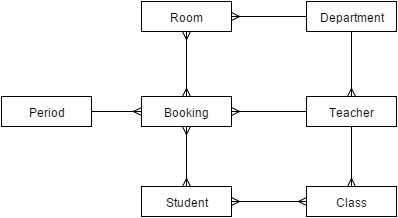
1. Web-based system: While this would provide an easily-accessible system, I have little experience in web-development, so don’t think I would be able to create a satisfactory solution using this approach. The existing system is also web-based, so I would likely be limited to the same design as the current 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 lacking, then the mobility doesn’t 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.

## Entity-Relationship Diagrams

The entities and their relationships are the same in both the existing and the proposed system – the differences are in how these entities are processed by the systems.  
The relationships between entities are fairly simple – there’s a roughly even split of one-to-many and many-to-many relationships.  
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.



## System Flow Chart



## Fact Finding Methods

Need to talk about observation, questionnaires, interview, and paperwork even if none was collected.

Interview with Mr Wilsdon, Client

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.

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.

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.

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.

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.

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

Naturally as the current system is computerised there is no paperwork – in lieu of this I have taken screenshots of multiple views of the current system. The current solution has the following steps:

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



Figure 4.0



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)).

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 3.0



Figure 3.1.1



Figure 3.1.2

Figure 3.1.3(a)



Figure 3.1.3(b)

Questionnaires

I have distributed questionnaires to students in years 13, 12, along with teachers; this constitutes the primary intended users of the system.  
  
TODO: Questionnaires

## Benefits of the proposed 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 supported features is an automated message on logon informing them that they’re in a booked room, ensuring they’re aware.

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.

# Design

## Overall System Design

A hybrid solution the web-based and the client-based proposed solutions – the main system will be accessible 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. Upon logon, 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.

## Solution 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 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 synchronously. As such I expect there will be lots of bugs in the development versions.

Also, students need to log on to a computer in order to see where other students are logged on – it’s not quite as easy to use as the website solution would be. However, I have little experience developing websites, and this solution allows for a far more powerful and responsive interface than that of a web UI.

## Description of Modular Structure

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 booking model. No special hardware is needed for this constituent – 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 constituent with 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 constituent is a machine that’s online all the time. 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 roughly linearly.

NetCore

This assembly should be referenced by both the Client and the Server and will provide access to the networking framework. 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 will be referenced only by the Server – I still intend to create it as a separate assembly however, as I feel that its logic is sufficiently different to that of the server to warrant the separation.

I intend to use the Entity Framework Code First 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 queries within code – they are 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. This would create a circular dependency, so I’ve been forced to create a separate assembly to provide interfaces (descriptions of the behaviour of a class, rather than the class itself) that remove the conflict.

## Processing and Algorithms

The following are high-level descriptions of some of the algorithms I intend to use – a lot is taken for granted, such as sending a message, which would require a significant amount of development in the background to work.  
  
I use the keyword “Upon” to introduce a function that is called upon an event occurring, to distinguish it from a normal procedure.

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 BookingAcknowledgementReceived(Message) Do  
 If BookingFailed(Message) Then  
 ShowErrorMessage(Message)  
 EndIf  
End.

// This is the logic for when a message containing a new booking layout is received  
Upon BookingChangeMessageReceived(Bookings) Do  
 If *User isn’t actively making a new booking* Then  
 UpdateBookingUI(Bookings)  
 EndIf  
End.

Server

// Called when a new booking message is received from a client  
Upon NewBookingMessageReceived(Sender, Booking) Do  
 If Invalid(Booking) Then  
 SendAcknowledgeMessageFailed(Sender)  
 Return  
 EndIf  
 ThreadLock(Database) Then  
 If *Booking clashes with existing database* Then  
 SendAcknowledgeMessageFailed(Sender)  
 Return  
 EndIf  
 EndLock  
 SendAcknowledgeMessageSuccess(Sender)  
  
 AddBookingToDatabase(Booking)  
 ForEach Client in ConnectedClients Do  
 SendBookingChangeMessage(*Bookings from database*)  
 End  
End.  
  
// Called when a new client connects  
Upon ClientConnectMessageReceived(Sender, Username, ComputerName) Do  
 ThreadLock(Database) Then  
 SendAccessLevelMessage(Sender, GetAccessLevel(Database, Username))  
 SetCurrentUser(Database, ComputerName, Username)  
 EndLock  
End.

## 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 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.

The data stored isn’t entirely critical, so backups can be taken once a week, likely as part of the school server backup.

## Human-Computer Interface

Current date, for easy identification of viewed day.

Timetable Window



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

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.

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.

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.

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

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.

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.

## Sample of Data Entry

## Sample of Valid Output

## 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 sensible input is taken.

## 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 (Typical, Erroneous, Boundary) being entered, for clarity.

| 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) |
|  | 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 a tile. | Clicking a booking tile (when logged in as a teacher or admin). | “EditBooking” window appears. | 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 17) |
|  | 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 18) |
|  | 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 20) |
|  | The filter on the Students section of the Edit Booking window works correctly. (See evidence for a print screen of initial test data) | {“”, 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 21 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 22) |
|  | 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 23) |
|  | 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 24) |
|  | 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 25) |
|  | 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 26) |
|  | 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 28) |
|  | 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 |
|  | 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 31) |
|  | 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 |
|  | 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 34) |
|  | 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 |
|  | 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 37) |
|  | 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 |
|  | 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 40) |
|  | 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 |
|  | Clicking “Add Class” on the Classes tab of the Admin Control window displays the right window. | Click “Add Class” on the Classs tab on the Admin Control. | Edit Class window appears, fields blank. | As expected (see evidence 43) |
|  | 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 invalid input to the Standard/Special Seats fields on the Edit Room window is handled with an error message. | “20” [Typical] “-1” [Boundary] “5Ps#,” [Erroneous] | Accepted Error Error | All as expected (see evidence 46) |
|  | 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 | All as expected (see evidence 47) |
|  | 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 | All as expected (see evidence 48) |
|  | 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 (see evidence 49). |

## 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. 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).
2. Editing an existing Booking with the EditBooking window initialises the window with all the information of the existing booking.



1. Omitting details while trying to submit a Booking displays an informative error message in a modal dialog box.
2. 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. Clicking “Add Period” on the Periods tab of the Admin Control window displays the correct window.
2. Clicking “Add Teacher” on the Teachers tab of the Admin Control window displays the correct window.



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.



1. Clicking “Add Class” on the classes tab of the Admin Control window displays the correct window.
2. Entering invalid input to the Standard/Special seats fields on the Edit Room window is handled with an informative 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. Entering invalid input to the Email field on the Edit Teacher window is handled with an informative error message.
2. Open numerous clients on multiple computers and connect them all to the server, perform standard operations such as editing Bookings/teachers

## 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.

# 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.

## 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, but constructed from the Name properties

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; }

// All bookings the user's involved in

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

// All classes the user's involved in

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

[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;

Bookings = new List<Booking>();

Classes = new List<Class>();

}

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;

LastName = u.LastName;

LogonName = u.LogonName;

Access = u.Access;

Bookings.Clear();

Bookings.AddRange(u.Bookings);

}

// Serialise all required properties

public override void Serialise(Writer Out)

{

*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.

base.Serialise(Out);

Out.Write((int)Access);

Out.Write(FirstName);

Out.Write(LastName);

Out.Write(LogonName);

Out.Write(Bookings.Count);

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

}

// 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();

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

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

}

// Obtain references to all related items by ID

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;

}

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; } }

public Student()

{

Access = AccessMode.Student;

Form = string.Empty;

}

public override bool Conflicts(List<DataModel> Others)

{

return base.Conflicts(Others);

}

public override void Update(DataModel Other)

{

base.Update(Other);

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.

Student s = (Student)Other;

Year = s.Year;

Form = s.Form;

Classes.Clear();

Classes.AddRange(s.Classes);

}

// Serialise all properties to the stream

public override void Serialise(Writer Out)

{

// 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));

}

// 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());

}

// Obtain references to related entities

public override bool Expand(IDataRepository Repo)

{

base.Expand(Repo);

try

{

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

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

public Teacher()

{

Access = AccessMode.Teacher;

Title = string.Empty;

Email = string.Empty;

}

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);

}

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);

}

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();

}

// Obtain IDs of related objects

public override bool Expand(IDataRepository Repo)

{

base.Expand(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);

}

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;

if (Obj == null)

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.

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.

}

## 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;

foreach (Client c in Clients)

{

try

{

// Send a disconnect message, then dispose the connection

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

c.Dispose();

}

catch { }

}

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 System.IO;

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;

}

}

}

## Server

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.

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.Send(Message);

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)

{

// 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>();

// If we're not deleting, then we want references to related items

if (!Delete)

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.

Client.DataRepository