Advanced Web Techniques - 6G6Z1011
Online Website – 1CWK50
December 2014
Keith Yates

David M^c^Queen – 10153465

1. Design

1.1 Database

Using the specification provided for the NHS Shift Management system, the key aspects were extracted to assist in the design on an appropriate system.
The system needs to store multiple users, in which each user belongs to a specific level. The nature of the system is to manage users shifts; as such a table is required to store which shifts each staff member is working.
With this in mind, a database structure was developed which fulfils all of these storage requirements, using 3 separate tables. Figure 1 shows a visual representation of the database structure, whilst figure 2 shows a use case for the system.

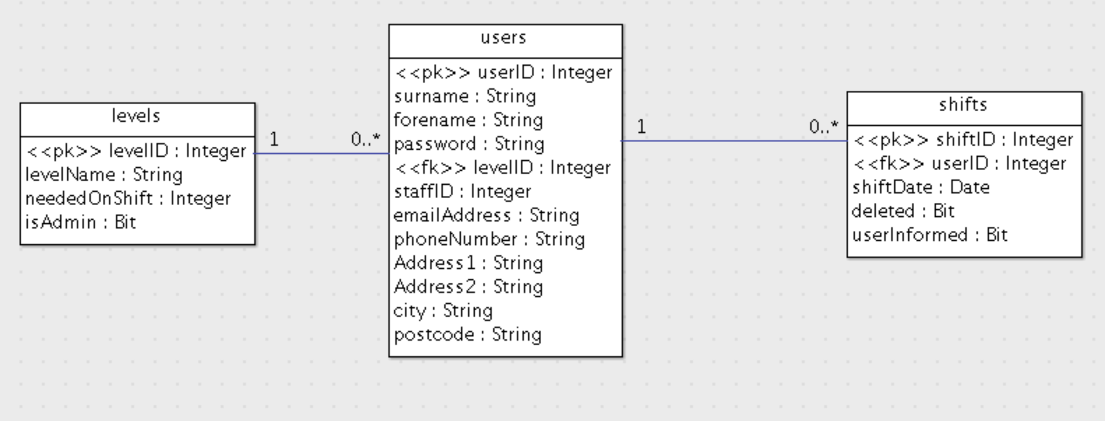

Figure 1 – ERD database design

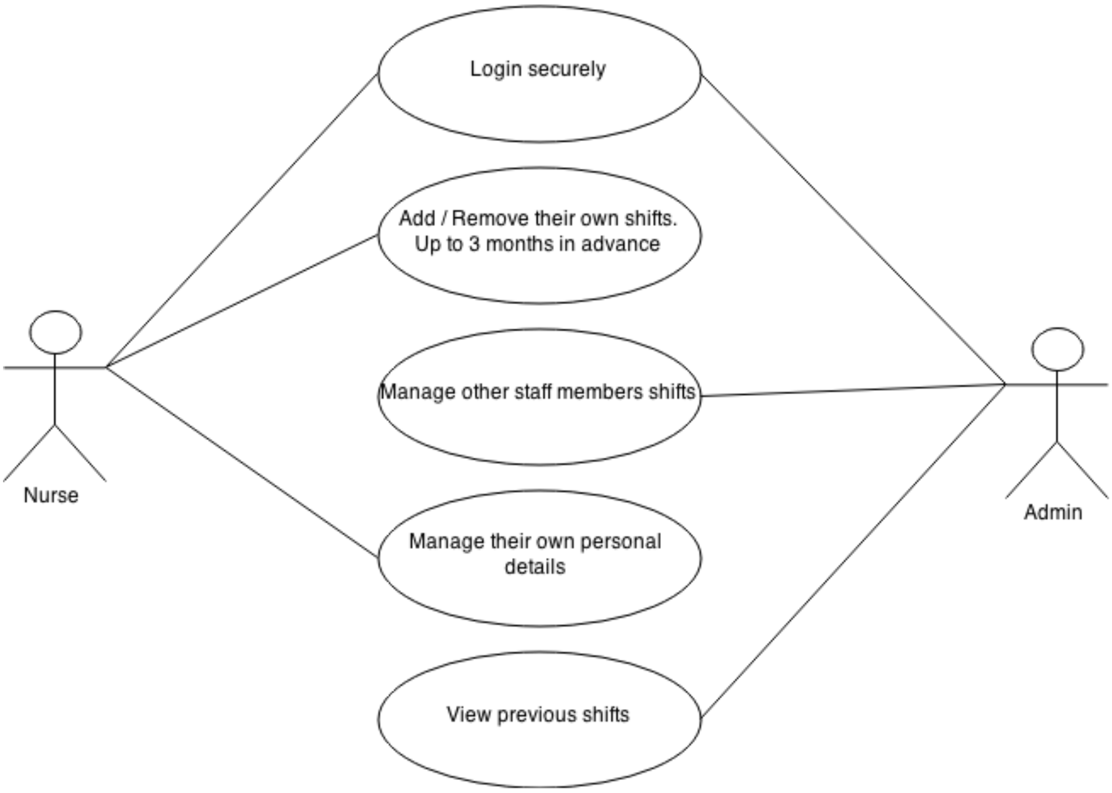
Login securely
Add / Remove their own shifts.
Up to 3 months in advance
Manage other staff members shifts
Manage their own personal details
View previous shifts
Nurse
Admin

Figure 2 – Use case for the system

1.1.2 Database Tables

The users table (table 1) contains all of the user information, including the personal information, in one central location. There is a NHS staffID column that connects the staff member to their main NHS account. The levelID links the staff member to a specific staffing level.
The password field will store passwords that have been hashed, to prevent plain text passwords getting stored in the database.
The email field is the users personal email address, and bears no relation to the email that is used in order to log into the system.

Table 1 – Users Table Data Dictionary

Description	Column Name	Data Type	Length	Nullable	Unique	Key
The unique, auto incrementing userID	userID	int		False	True	Primary
Staff members surname	surname	varchar	200	False	False	
Staff members forename	forename	varchar	200	False	False	
Encrypted Password	password	varchar		False	False	
Staff levelID	levelID	int		False	False	Foreign (levels)
Staff NHS ID	staffID	int		False	True	
Personal email address	emailAddress	varchar	100	True	False	
Personal phone number	phoneNumber	varchar	14	True	False	
Personal address	Address1	varchar	100	True	False	
Personal address	Address2	varchar	100	True	False	
Personal address	City	varchar	100	True	False	
Personal address	postcode	varchar	9	True	False	

The levels table (table 2) is to contain information regarding each staff level. It is structured in a method that makes further development of the system easy, with

little to no modification of the existing system needed. One such development would be allowing for new staffing levels to be created (such as junior nurse). If a new level is created, this will be reflected in the relevant areas of the shift management tool, such as the minimum amount of staff needed per shift, per level (neededOnShift column).
The 'isAdmin' column determines if the staff level should be granted admin permission within the system, this allows for the possibility of multiple staffing levels having admin permission that could be of use if the system needs to be expanded, or linked to another system.
The levels table links to the users table on levelID.

Table 2 – Levels Table Data Dictionary

Description	Column Name	Data Type	Length	Nullable	Unique	Key
The unique, auto incrementing levelID	levelID	int		False	True	Primary
The level name	levelName	varchar	100	False	False	
How many staff of this level are needed as a minimum for a shift	neededOnShift	int		False	False	
Determine if the level is granted Admin privileges.	isAdmin	bit		False	False	

The shifts table (table 3) contains each and every shift, for all staff members. If a shift is deleted, then it is kept in the tables but is marked as deleted. This allows for deleted shifts to be reviewed, and it also allows for the deleted shift to be communicated to the staff member. The date format used to store the shift date is the international standard, ISO 8601 (ISO, 2014).
If any shift is created or deleted by admin, the userInformed column will allow the system to find these shifts, and then communicate the relevant message to the user.

Table 3 – Shifts Table Data Dictionary

Description	Column Name	Data type	length	Nullable	Unique	Key
The unique, auto incrementing	shiftID	bigint		False	True	Primary

shiftID						
The user which is working the shift	userID	Int		False	False	Foreign (users)
The date of the shift	shiftDate	Date (ISO 8601)		False	False	
If the shift has been deleted	Deleted	Bit		False	False	
If the user is aware of the shift (Admin created)	userInformed	Bit		False	False	

1.1.3 Stored Procedures

"*A stored procedure in SQL is a group of one or more SQL statements*" (Microsoft, 2014). These group functionality together, carrying out pre determined tasks, optionally accepting inputs and returning record sets. They are similar to functions in Object Orientated languages, and are useful to abstract database functionality away from the web server and keep it local to the database.

For the shift management system, all interaction with the database will be carried out using Stored Procedures. This drastically reduces the level of SQL code that is present on the server (section 1.4), and keeps all database functionality together. The only SQL code that is present on the server is the call to the relevant stored procedure.

As a minimum, stored procedures will be needed to carry out the following tasks:

- Create a user
- Read users details
- Update a users details
- Delete a user
- Create Shift
- Read Shifts
- Delete Shift
- Login

1.2 Server

The server section, which connects the database to the frontend display, will be created using the Model View Controller (MVC) pattern. This pattern separates each of the different sections out, so that each area is atomic, only being responsible for one specific task. The Database section (model) doesn't handle any logic or display; it is purely responsible for retrieving and updating the data, using stored procedures (section 1.2). The display section (View) simply displays information to the user, using data passed from the controller. The logic section (controller) connects the Model and View sections together. It performs all of the logic, calling the Model to perform any database functionality, and calling the View relevant for the information that needs to be displayed.

In order to implement the MVC pattern, the CodeIgniter framework will be utilised. "*CodeIgniter is a community-developed open source project*" (CodeIgniter 2014) "*with a very small footprint*". It allows for MVC to be implemented in an easy manor, allowing for security to be added preventing malicious attack.

1.3 Frontend

To achieve all of the functionality set out in the specification, a total of 3 pages will be available for the user.

- Login
- Calendar View
- Settings

1.3.1 Login

The login page will display a simple form, allowing the user to enter an email address and password. This will be the first page the user sees, and the page in which they are directed to if they are not logged in. Figure 3 shows the design of the login page, 2 input fields with a 'login' button.

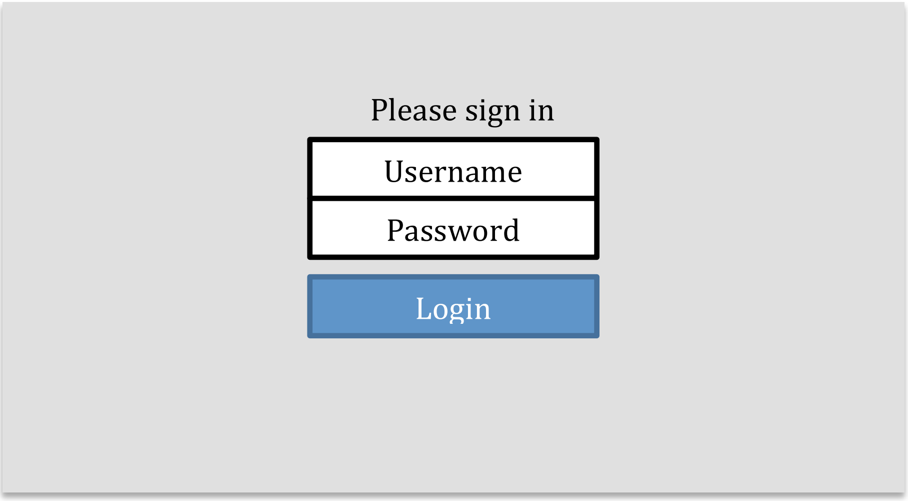

Figure 3 – Design of the login page

1.3.2 Calendar

The calendar view (Figure 4) will be the main view in which the user will use. This will display the calendar, which will allow the user to carry out all the shift management described in the specification.
Alongside displaying the calendar, messages will be displayed to the user (Figure 4, section 4.2). The messages will be; any modifications made by admin to the users shift; Instructions on how to use the calendar; a list of weeks which the user is below the shift level specified; Error messages and warning messages.

The calendar (Figure 4, section 4.3) aspect will be created using the fullcalendar.io library. This is a JQuery library released under the MIT license, meaning it can be used in almost any way so long as credit to the creator is left in tact.
There is one main benefit to using FullCalendar, over creating a new calendar system; there is no need to reinvent the wheel. Multiple aspects are modifiable enabling FullCalendar to fulfil all of the functionality set out in the specification., without the needed to create a new calendar. Through modifying different

parameters in the <script> section of the page, all functionality can be achieved, from changing the background colour of the events to handling the process of when a user clicks on a specific day.
As Fullcalender doesn't provide a 4-week view by default, it will be necessary to create a new view, in the core code, which fulfils the requirements of the spec; to show the next 4 weeks, with the first week being the next week.

At the top of the calendar view will be a navigation bar (Figure 4, section 4.1), this will display the users name along with navigation controls, allowing the user to access the settings page and logout. The Navigation Bar is present at the top of every page in which a logged in user sees.

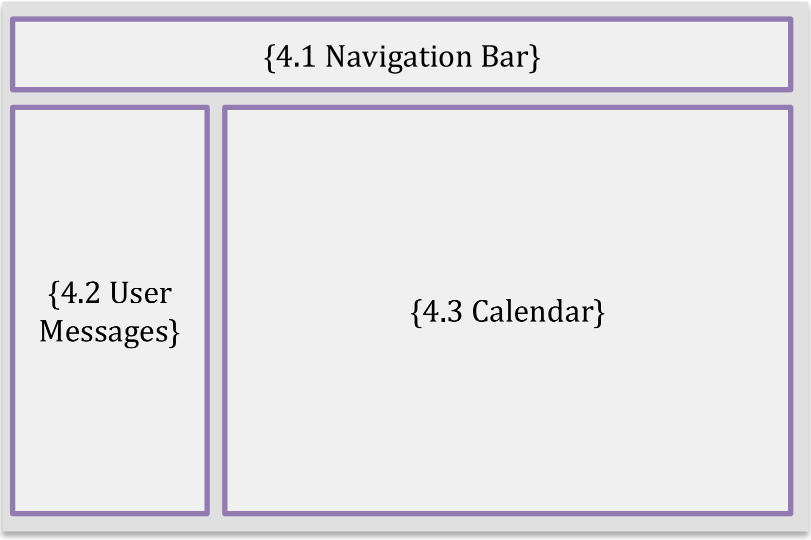

Figure 4 – Basic layout of the calendar view

1.3.2.1 Improvements

The standard user will only have one view available to them, as covered in section 1.4.2, however Admin will have 2 views available. These will be a month view, and a week view. The month view will allow admin users to view the entire month, previous months, and future months. Thus they will be able to review all previous months, and all staff shifts, providing a record of which shifts have been worked in the chance that this information is needed for administrative purposes.
The week view will provide more screen real estate in which to view the staff members that are working. This is essential due to the fact that each staff member is displayed, as opposed to the staff categories the standard user sees. Whilst this isn't a major problem with the current staff levels and staffing rules, if these were to change the system is able to accommodate for this increase and easily display all the staff that are working on specific days.

1.3.3 Settings

The settings page, figure 5, will display all of the current users personal details, which are in the system. This will allow the user to update their personal details as circumstances change. Further to being able to update persona details, users will be able to create a new password, which must meet the password requirements.

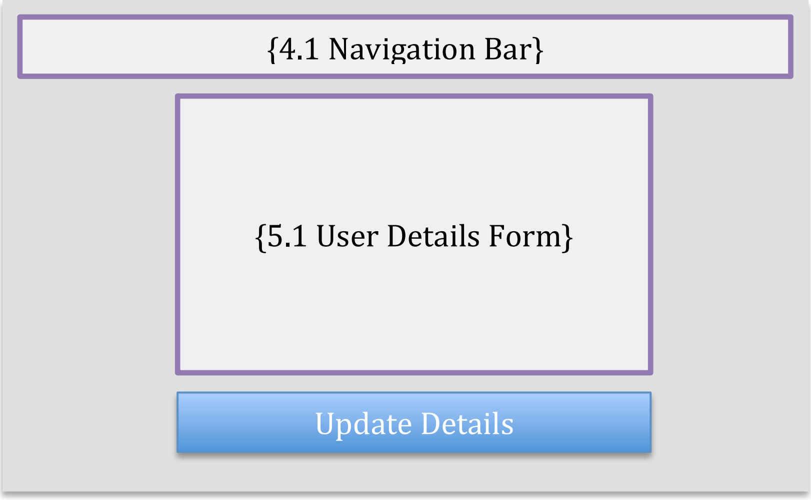

Figure 5 – Basic layout of the user settings page

1.3.4 Bootstrap

To assist in the presentation and style of the system, the Bootstrap framework will be utilised. “*Bootstrap is the most popular HTML, CSS, and JS framework for developing responsive, mobile first projects on the web*” (Bootstrap, 2014), it is released under the MIT license, making it freely available to use.

Bootstrap provides an easy to use framework that can be applied to existing HTML in order to create beautiful, clean displays. As such it will be used to style display the system, being applied to each page.

2. Implementation

Throughout the development of the system, all of the code is checked into source control, using GitHub. Source control is an incredibly valuable tool, allowing for every stage of development to be reverted to, creating a checkpoint in case of any catastrophic failures in future development.

2.1 Database

The database is the initial section of the system to be created; it is the central hub for all of the data and the structure of the data can have a great impact on the rest of the system, affecting how the server requests the data, and the classes in which the data is based on.

Using a text editor (sublime), a database script was created which contained all the tables and stored procedures, along with dummy data inserted into the database. This allowed for easy transfer between database instances, such as the local development environment and the production environment. Creating the SQL code in such a way enables it to be stored in source control along with the rest of the development code.

MySQLWorkbench was used in order to connect to the database instances, further to this, it was used to write and execute additional SQL code throughout the development of the system. MySQLWorkbench proved to be a better portal for accessing the database instance than that of phpMyAdmin that ships with Xampp servers.

The SQL code that has been created for the system makes extensive use of various database operations, Including: Inner Joins, Left Joins, Aggregate functions, Group By, an Order By Statements. Making use of these operations allows for the creating of efficient SQL that performs all the necessary data modifications in one batch.

Alongside the stored procedures described in chapter 1.2, additional stored procedures were created to perform basic functionality that, although not necessary for the specified system, would be desirable for future development and ease of modification.

These stored procedures perform tasks such as:

- Create staff members
- Delete staff members
- Add new levels
- Delete levels
- Edit levels

If necessary, further development could be easily made to the system, allowing for these stored procedures to be utilised, thus providing extended functionality to the Admin level users.

The database script, created for all database operations, can be found on the server where the web application is hosted (CW1/Database/DatabaseSchema.sql).

2.1.1 Passwords

The Admin user that was defined in the specification had an incorrect password, and as such the password was changed to 'Organ1sed', the rest of the staff details have remained the same.

The passwords for each user are hashed to prevent plain text passwords being readable. In order to enhance the security of the passwords, each is 'salted' on either side of the user-defined password. This comprises of a Pre an Post salt. Essentially sandwiching the password between these salts, ensuring that no hashed password is the same as another hashed password, meaning the possibility f using hashing tables to attack the system is drastically reduced. The 'pre-salt' is a random string comprising of letters, numbers and symbols ('zhjbfvh56^%&'). The 'post-salt' is the users staff ID. Doing so ensures that the salt changes for each user. Using the admin timetable account as an example, which has the staff ID of '6189' and the password 'Organ1sed'. When this is concatenated with both of the salts results in a pre-hash password of 'zhjbfvh56^%&Organ1sed6189', which is then hashed with the SHA256 algorithm. If another user uses the same password, the resulting hash would be different as the post-salt is different.
In order to ensure that the password hashing worked fully, it was necessary to explicitly make the predicate use Binary comparison and explicitly set the collation, as shown in Code Example 1. Doing so ensures that when the database is deployed to a different database instance, the comparison would work as expected.

"BINARY u.password COLLATE utf8_general_ci = BINARY SHA2(CONCAT('zhjbfvh56^%&' , CONCAT(password, u.staffID)), 256) COLLATE utf8_general_ci;"

Code Example 1 – Password matching using Binary comparisons, and explicitly setting the collation.

2.2 Server

In order to develop the system locally, Xampp was installed on the development machine. This allowed for server side scripts (such as index.php) to run. Xampp was chosen as the specification stated the system needed to be developed in php and MySQL, both of which can run on a Xampp server.

As discussed in chapter 1.2, the MVC framework CodeIgniter was used for the development of the server side code. After configuring the framework to connect to the database instance, work commenced on the database access functions, which are present in the Model section. Comprising of 2 models, 1 for the shift functionality and a 2nd for the user functionality, each of the functions call the relevant stored procedure in the database, and then pass the result set back to the controller as a result array.

To perform the server side logic, 3 Controllers were created; One each for shift functionality and user functionality, and a 3rd specifically for verifying the users login credentials. The controller for verifying user credentials was kept separate from the other 2 controllers because although it was handling user functions, the user is not logged into the system at this point so it was prudent to keep logged in functionality and non-logged in functionality separately. Alongside the controller deciding which Model to call, it was also responsible for deciding which View to show to the user. This is relatively straight forward, with a single controller generally having a single associated view (other than the common header and footer files). With the main calendar, there could be 2 different level of user; Admin or Standard. As such different functionality is needed for each user. In this instance, the controller makes a decision on which view to load depending on if the user is marked as admin or not.

2.2.1 Server Side Validation

The controller also performs server side validation, which is essential to work as support for client side validation ensuring that the data input by the user is a) valid data for what has been requested, and b) non malicious, such as SQL injection. Server side validation is essential as client side validation can't always be trusted; JavaScript might be displayed on the clients machine, or the client might intentionally bypass the validation, both of which could cause problems in the logic and database sections of the system. It is not possible to bypass validation on the server, due to the fact that clients don't have access to the server itself, thus they don't have access to the validation code.

Although it is possible to perform validation server side, it is not a replacement for client side validation. Client side validation is essential for providing the user with a better experience using the system. They can be alerted of any issues with their inputted data without leaving the page. This has multiple benefits: If the data is invalid then it will need to be re-entered, if validation solely takes place on the server then when the page reloads the user will need to re-enter all of the data they have previously entered. Further to this, the process is creating unnecessary communication between the client and the server, which could have adverse affects if the user is on a slow connection, such as a mobile device.
Code Example 2 shows the server side validation that is applied to the user updating their personal details. In this example an array is being created with the new details, before the array is passed to the model that will perform the update in the database. Multiple operations are taking place here, for each of the values. The first is a ternary operator, checking the length of the new value. If it is less than 0, an empty sting is inserted into the database. However if there is a length greater than 0, the new value is used. In using this new value, it is limited to the max data size that the table can hold, to ensure that trying to insert a string that is too long generates no database errors. Using ternary operators in this situation removes the need for IF blocks for each new value, that would take up extra room and duplicated code, as the ternary operators are simple to understand and can be condensed onto a single line.

```
$userSettings = array(
‘userID' => $userID,
'password' => $newSettings['password'],
'forename' => (strlen($newSettings['forename']) > 0 ? substr($newSettings['forename'], 0, 200) : ''),
‘surname' => (strlen($newSettings['surname']) > 0 ? substr($newSettings['surname'], 0, 200) : ''),
'email' => (strlen($newSettings['emailAddress']) > 0 ? substr($newSettings['emailAddress'], 0, 100) :
''),
'phone' => (strlen($newSettings['phoneNumber']) > 0 ? substr($newSettings['phoneNumber'], 0, 14)
: ''),
'address1' => (strlen($newSettings['address1']) > 0 ? substr($newSettings['address1'], 0, 100) : ''),
‘address2' => (strlen($newSettings['address2']) > 0 ? substr($newSettings['address2'], 0, 100) : ''),
'city' => (strlen($newSettings['city']) > 0 ? substr($newSettings['city'], 0, 100) : ''),
'postcode' => (strlen($newSettings['postcode']) > 0 ? substr($newSettings['postcode'], 0, 9) : '')
);
```

Code Example 2 – Server side validation on updating the users settings

The security that was implemented on the system comprises of storing the users details in a session, which is destroyed when the user logs out. When the user logs in, by providing valid credentials, the server retrieves specific user information from the database, including their unique user ID, and their name. With this information stored in a session, a check is performed at the beginning of each function performing sensitive tasks. If the user is logged in, then they are allowed to proceed. However if they are not logged in, then the user is redirected to the login page and prompted to login with valid credentials.

Performing security in this method has similar benefits to those discussed for server side validation; the client is unable to bypass the security checks, as they don’t have access to the server.

2.3 Frontend

The front end, visual aspect, of the system was developed using the 3 main web development technologies; HTML, CSS, and JavaScript. All of these languages are essential in creating a website that is required to display content to the user.

Alongside the main languages, several libraries and frameworks were used. Bootstrap (chapter 1.4.4) was used to style the system, making use of both the display elements and the JavaScript functionality to enhance the display elements by using transition elements.

FullCalendar (chapter 1.4.2) was used to display and handle all the frontend calendar functionality. FullCalendar is a library that is specifically designed for displaying and handling calendar events, the platform it provides for creating custom functionality was utilised heavily for the purpose of this system. Event handlers, such as ‘DayClick’, ‘EventClick’, ‘ViewRender’, and ‘eventAfterAllRender’ were all utilised in the development of the front end of the system. With the different event handlers, it was easy to separate the different segments of code, keeping functionality separate and increasing code maintainability and readability. The specific code for each section was placed inside the relevant handlers, with some code being abstracted out into a separate function, which is then called from inside the handler, to increase code reusability.

As different functionality is needed for both the standard user and the admin user, 2 separate scripts were created, with one or the other being loaded depending on the user logged in. The decision on which script to load is made inside the controller, server side (as described in chapter 2.2). Once the relevant script has been loaded, the user is able to use the functionality in the calendar, whilst only having access to the functionality in which they are meant to have access to. This increases security, as non-admin users are unable to use the admin functionality, however this would also be prevented server side, and increases code maintainability as there is a different file for the different functionality. When making modifications that only apply to admin users, then changes only need to be made to the admin script file.