**5 Implementation**

**5.1 Login and logout**

**5.1.1 Login**

A screenshot of a cell phone

Description automatically generatedThe first page in which a user is presented with is the sign in screen. Once the user has entered their details correctly (email and password), and clicks the ‘sign in’ button the *onLogin()* method located in the *signin.component.ts* is called, seen in*figure 29*. If the sign in form is not entered correctly, for example, an input is field is left blank, the *onLogin()* method returns and the relevant error message is shown to the user. If the form has been entered correctly this method passes the entered email and password to the *login()* method inside the *auth.service.ts* as shown in *figure 30.* The *login()* method makes use of Angular’s HttpClientModule to send the data from the form as the body of the request to the server URL endpoint as a POST method *(figure 30****:****31-33).*

A screenshot of a social media post

Description automatically generatedThe object from the front-end is passed onto the backend server in which uses the data to query the database, this can be view in **APPENDIX NUMBER.** The server *login()* method queries the database and finds an entry where the email is equal to the data sent from the body of the HTTP POST request. If there is a user with the email provided their password is compared against the password stored. As all the passwords that are found in the database are hashed to improve security a node library called Bycrpt is used to decode the stored password. This decrypted password is then compared against the password inputted by the user. If the email and password match the information stored in the database then the *login()* method returns a response containing the user’s ID number and the user’s status as a JSON object. However, if any information is incorrect or a user is not found a HTTP error (which will be discussed in more depth later), correlating to the corresponding error is returned. This response is then displayed to the user on the front-end in the form of an error message.

Figure 30 – login() method

Figure 29 – onLogin() method

Once the response is returned the *login()* methodinside the *auth.service.ts* stores the *userId* and *userStatus* in local storage in the browser (*figure 30:37)*. While using the browsers local storage is sometimes considered bad practice, it was used in this case as the userId and userStatus have been used multiple times throughout the application, so they needed to easily accessible. This ease of access to items is an advantage of using local storage. Another storage method is to use a session. This method was decided against because it removes all data from storage if the tab is closed. By using local storage, it means that if the user accidently clicks off the tab they will not be logged out of the system and can return to the page and continue where they left off. In terms of security, the system does not store any sensitive information, such as passwords and emails, in the local storage so any malicious attempts to get users data have been reduced slightly.

The *login()* methodalso returns an observable in which the *onLogin()* method subscribes to (*figure 29:29).* As the application has two user groups and each group has different functionality the status of the user is checked by using the data that has been subscribed to. If the user has the status of admin, they are navigated to the admin home page and the related functionality is loaded. However, if the user has the status of user the system navigates them to the user home page and the related user functionality is loaded.

**5.1.2 Logout**

In order to log out of the application the user clicks the log out button located in the right of the navigation bar. By clicking this button it calls the *logOut() (figure 31:78)* method located in the auth.service.ts. which in turn removes all the items from the local storage and navigates the user back to the sign in screen.

A screenshot of a cell phone

Description automatically generated

Figure 31 - Logout() method

**5.2 Authentication guard**

The created application has two types of users both with different functionality and permissions depending upon their status. If a user has the admin status, they have access to all the functionality, were as a general user does not. As this is the case some pages need to be protected so that only a user that has the authentication status of admin can gain access to these pages, while a general user is blocked from accessing these pages. In order to accomplish this Angular route guards were made use of and they are used to prevent users from navigating to parts of an app without authorization (Angular.io, 2020). This will explain how the admin authentication guard has been implemented, however, it should be remembered that a user guard has also been used within the system but as both guards follow the same process only one will be discussed in depth. *Figure 32* shows a typical flow of events with regards to the authentication guards within the system.

A screenshot of a cell phone

Description automatically generated

Figure 32 - Flow of events for authentication guards

A screenshot of a social media post

Description automatically generatedWhen a user logs into the system their details are passed to the admin authentication guard as seen in *figure 33:25* as an observable. The admin guard first checks to see if the user exists and if the user status is that of an admin. If this is the case the guard returns true. Inside the *app-routing.module.ts* file the admin guard is added to all the routes, for example, the rota creation page, that only a admin has access to by adding it to the *canActivate* property, as seen in figure 34:20. By adding the admin guard to the canActivate property it ensures that anytime someone tries to access a page in which they need admin status the application checks to see if they have the correct authentication status, (are logged in and have the admin status) through the admin guard before proceeding.

Figure 33 – Admin authentication guard

A screenshot of a social media post

Description automatically generated

Figure 34 - User and Admin routes with canActivate property

As mentioned early this is the same process for the user guard however, it has been used to allow access to general user pages by adding the created guard to all user routes. By using the authentication guards, it helps to protect routes and denies access to pages and features of the system if the user does not have the correct status. Furthermore, it also stops any user that is not a registered user from accessing and using the system. If an unregistered user enters the URL that is used for rota creation, “/admin/rota” they are denied access as first of all they are not logged in and secondly, they don’t not have the admin status. As a result, the system blocks their access as the authentication guard on that route returns false and returns them to sign in screen.

**5.3 Employee management**

As the one of the main functions of the system is to provide an employee management. The *employee* component does this by providing a list of all employees and allows the user to create a new employee as well as editing and deleting existing employees. This section aims to provide an overview of this was achieved using the created the system.

**5.3.1 Displaying all employees**

A screenshot of a cell phone

Description automatically generatedWhen the user navigates to the employees page it invokes Angular’s ngOnInit lifecycle hook that calls the *getEmployees() (figure 34:28)* method in the *employee service* that makes an HTTP GET request and returns all the registered employees as an JSON object. The *employee-list* component then subscribes to the results and the items are stored in an employee object. Using a structural directive, *\*ngIf,* in the employee-list HTML template it checks to see if any employees exist. If no employees exist a message of “No employees yet” is displayed. However, if employees exist another structure directive, *\*ngFor,* binds to the HTML pages and loops through all the employees displaying each individual employee as a new row in the created employee table. As a result, the user can view all the employees registered to use the system as a list. Furthermore, a total of all employees is displayed to the user. The total is declared as *total* inside the component by setting it equal to the length of the employee array. This calculated value is then embedded into the HTML template by using interpolation.

**5.3.2 Viewing an individual employee.**

In order to view an individual employee, Angular’s event bindings were made use of. Event bindings respond to any DOM events and are triggered by user input (Angular.io, 2020). Beside each employee a “view” button has been placed that enables to the user to view the selected individual employee. The following syntax on the button is used – *(click)=”viewEmployee(user)”.* The click event on the button calls the *viewEmployee()* method and the user is passed to it (*figure 35:42).* The *viewEmployee()* method then navigates to the view employee page in which the *viewEmployee* component loads all the selected employee’s information.

Figure 35 – Employee-list component with call to getEmployees() method and viewEmployee() method.

The *viewEmployee* component parses the user’s id number from the URL by using the paramMap interface provided by Angular (*figure 36:27)*. This component then subscribes to the *getEmployee()* method which sends a request to the server and returns all the users information based upon their id. This information is then used to display the employee’s profile, shifts, holidays and training.

By looking at the URL’s this process can be understood in more detail. When the user navigates to the employee-list component the URL is “/admin/employees”. If the user would like to view an individual employee, for example John McDonald, who has the user id of 32. They would click the view button and the system would navigate to “admin/employees/32”. Inside the view-employee component it would parse the users id from the URL, in this case 32, and load all the information associated with the user with the id number of 32.

A screenshot of a cell phone

Description automatically generated

Figure 37 – paramMap interface used in the viewEmployee component

One problem with passing the user id through the URL that was realised during user testing was that if the user manually entered the user id that didn’t exist the view employee page would not load. It was realised that this was because all the features on the page relied on the user id. If there was no user for the selected id, then the system could provide any information. This was changed and if a user manually enters a user id into the URL for a user that does not exist the page will load with a message stating “Sorry, a user for the selected id does not exist”.

**5.3.3 Deleting an employee.**

A picture containing knife

Description automatically generatedOnce the user has navigated to the view employee page, they have the option to remove an employee from the system. This feature was added as it allows for the removal of staff who no longer work for the company and therefore do not need to use the application. Event binding is again used but this time on the delete button. This event calls the *deleteUser() (Figure 38)* which makes a call to the employee service method of delete user that removes the user from the system. A window alert is used that shows the user that the employee has been successfully deleted and then navigates to the employee homepage. After the user has been deleted the employee list updates and the user is removed from the list and the employee total also decreases by one.

Figure 38 – deleteUser() method

**5.3.4 Creation of an employee**

To create an employee the user clicks the create employee button located in the employee list component. This in turn opens a modal which contains a reactive form in which the user inputs the new employee’s information, such as name, email and date of birth. Reactive forms use an explicit and immutable approach to managing the state of a form at a given point of time (Angular.io, 2020). This helps to maintain the integrity of the model between changes as each change returns a new state. The creation of the form involved three main steps.

The first was to create a property and set it to a new form group instance. In this case a property of *employeeForm* was created and set to a new FormGroup instance. Individual form controls were added to the created form group (*figure 39:29-36)* and the validators property was added to each control to ensure that they weren’t left blank by the user. If any input was left blank the system would display an error message below the respective input.

A screenshot of a cell phone

Description automatically generated

The second step involved was to associate the FormGroup with the model and the view. Each input in the view was given a FormControlName that binds each individual input to the form control defined in the created FormGroup. The form controls communicate with their respective elements and store any inputs from the user.

Figure 39 – Employee form group

*A screenshot of a cell phone

Description automatically generated*Finally the *create-employee* component in which the form group is located listens for a submit event by the form and emits an *ngSubmit* event that is bound to the *onSubmit()* method which can be seen in *figure 40:40.* This method then sends the user inputted form, values to the server via an HTTP POST request that is called inside the *createEmployee* method located in the employee service. The new employee’s details are then stored in the database and if successful and alert is shown, the window reloads, and the new employee is added to the employee list. However, if unsuccessful, or if a user with the provided email is already registered to the system an error message corresponding to the error that has occurred.

Figure 40 - onSubmit() method used in the create-employee component

A screenshot of a social media post

Description automatically generatedUpon creation of a user an email is sent to the created user’s email address. Within this email it provides the new created employee with their password that they can use to log into the application. The process of generating a secure password and sending it to the new employee via email is done within the server-side code, more specificity within the *createUser* method located in the user-controller. The createUser method receives the inputted form values from the user as a JSON object. Using an Generate-password, an external password generator it produces a random password for the user. The user’s email is taken from the JSON object and using Nodemailer, an imported module that allows the application to send emails. By using this it allows the generated password is sent to the user’s email. Upon receiving the email containing the password, the user can use it to log in and use application. *Figure 41* shows the use of Nodemailer within the system.

Figure 41 – The use of Nodemailer to send an email to the created user containing their generated password.

While the functionality of creating a user was created in one of the first sprints, the extra functionality of the system to send an email containing a generated user password was implemented at a later stage. Before it was implemented the created user had to be verbally told their password. It was decided that this was not be the most secure or easy way to inform new users of their password to gain access to application. As the password that is generated by the system produces a password that is 20 characters in length and contains a mix of letters in upper and lower case as well as numbers, verbally telling a new user a generated password would not be ideal. One incorrect character would mean that the user would not be able to gain access to the system. Furthermore, if one user had access to all the created user passwords it would mean that they could gain access to each individual employee’s information and could lead to possible unwanted changes to be made to their profile, shifts and holidays.

**5.4 Rota creation**

The rota creation component can be divided into 3 main functions that all combine together to provide the designed functionality needed in order to create rotas. These functions are the date controls, the drag and drop of users into a selected area and inserting, edit and deleting individual employee’s shifts. This section will discuss each of these functions and how they combined in order to create a weekly rota for employees. Upon loading of the page, the user is presented with the list of employees on the left-hand side, the date controls that show the current at the top and the different areas located in the middle of the screen. Any shifts that have already been created for the current day are displayed in their respective areas. This is done by making an HTTP get request to the API that returns any shifts for the current date.

**5.4.1 Date controls**

As stated before, upon loading of the rota creation component, the user is presented with the current date. A date pipe provided by Angular is used to make the date more readable for the user. For example, if the date pipe wasn’t applied the current date would be formatted as *“Thu Sep 03 2020 12:00:00 GMT+0100 (BST*)”. However, using the date pipe it is a displayed as *‘Thursday, September 3, 2020” and makes it much easier for the user to understand.* As discussed in chapter 4, there is controls that the user can interact with to change the week or day. By clicking the weekly controls an *onChange* event is emitted that will either increase the displayed date by a week or decrease it by a week depending upon which button the user clicks*.* The date is bound to each area component as the property *selectedDate*. Within each area component a *selectedDate* property is declared and decorated with the @Input(). By decorating the property with input, it ensures that when a change is made, such as clicking the button to change the week the new selected date is updated within each area component.

#### Upon changing the day any created shifts for the selected day are shown to the user. By using the ngOnChanges lifecycle hook within each area component it detects that the data bound property (selectedDate) has changed. When a change has been detected, each area component calls the getShifts method (figure 43) in the schedule-service that returns a shift object as an observable. Each area component then subscribes to the observable and any shifts for the new selected date are displayed via the view. Figure 42 shows the ngOnChanges lifecycle hook within the manager-area component and shows it calling the getShifts method by passing the new updated date to it. It also shows the component subscribing to the observable that the getShifts method returns.

#### A screenshot of a cell phone Description automatically generated

As a JavaScript date is formatted as “Wed Mar 25 2015 00:00:00 GMT+0000 (Greenwich Mean Time)” it needed to be converted to the SQL date format (YYYY-MM-DD) to retrieve the shifts for the selected date. In order to convert the selected date a function was created (*figure number)* that converts the JavaScript formatted date into an SQL formatted date. The selected date was passed into the function and a variable of convertDate was set to the result (*figure 42:67)*. This convertDate variable is then used in the getShifts method and allows for the database dates to be queried using the correct date format.

Figure 42 – ngOnChanges lifecycle hook within the manager-area component

**A picture containing knife

Description automatically generated**A screenshot of a cell phone

Description automatically generated

**5.4.2 Drag and drop of employees**

In order to create a shift for an employee their card which holds their name must be dragged and dropped into one of the 4 created ‘areas’. The ‘@angular/cdk/drag-drop’ module was used as allows for easy creation of drag and drop interfaces. The users and area components are wrapped within the cdkDropListGroup attribute that allows for items to dragged between multiple lists. A list of employees is located on the left-hand side of the window and are set as the cdkDropListData containing all the data for each employee. The area components listen to the drag event and when an employee is dragged and dropped into the selected area the employee’s data is copied and inserted into the selected area list. For example, the user selects the user John King and drops it into the manager area. That information is then copied from the employee data list and inserted into the array, managerList, with in the manager-area component.

A screenshot of a cell phone

Description automatically generatedUpon dropping the user into the selected area, the system checks to see if the user isn’t already allocated to work that day or that they haven’t requested the day off as a holiday. If either of these conditions are true an error will show at the top of the screen and remove the employee from the area. Two Rxjs behavioural subjects have been created with the schedule service. One behavioural subject has been created to show an error if the user is already allocated to work and the other is to show an error if the user has requested the day off. Within the schedule service both subjects have been set to false. However, when an employee has been dragged into an area and either of the conditions are true then the area component will change the value of the behavioural subject to true. As the area-error-handler component subscribes to both behavioural subjects any changes in the value will cause the corresponding error to display to the user. For example, if a user drags a user that has already been allocated to work the area component will change the userExists variable to true and as a result the area-error-handler component will display the message “User is already allocated to work this day”.

Figure 1

A screenshot of text

Description automatically generatedIf one of the errors do occur the employee needs to be removed from the area so that they are not allocated to work. To remove the employee from the area the JavaScript array filter method is used. First of a forEach loop is used to go through each of the shifts and the holiday for the employee. If the employee is already in the created area array the filter method removes the employee from the area. The view then is updated and the employee is removed from the area.

Figure 2

**5.4.3 Inserting, deleting and editing individual shifts**

When an employee is dragged into an area a card with the employee’s name, a start time and finish time is presented. The user must insert a start time and finish time in order for the shift to be allocated. If the start and finish time are entered an HTTP POST request is sent to the API in which the shift is inserted, allocated to the selected user and added to the corresponding shift area array.

To delete a shift for an employee the user clicks the delete button and a HTTP DELETE request is sent to the API and as a result the shift is removed from the database. To remove the shift from the view the filter array method is made use of again. Upon the click of the delete button the shift id is passed to the *deleteShift* method. The filter method finds and removes the shift based upon its id from the shift array and returns a new array that is displayed to the user in the view.

Editing an individual is done by clicking the edit button that allows for the start and finish times to become editable. The pervious values are bound to the ngModel on the start and finish time inputs. This allows the user to see what the pervious values were and therefore can update the times correctly. Upon clicking the save changes button the new form values are sent to the API as a PUT request and changes the values based upon the shift ID.

**5.5 Routes and controllers**

Routing refers to how an application’s endpoints respond to client-side requests (Express, 2020). Routing is defined within the application using methods of the Express routing object that correspond to specific HTTP methods; for example *router.get()* to handle all GET requests. These routing methods specify a call back function and are called when a request is made to the specified end point. The application listens to requests made by the front-end and when a route is matched it calls the function associated with that route.

Within the application separate routes have been set for different end points. For example, all routes that are associated with user have been included in a user-route file. Furthermore, call back functions, or handler function have been separated into controllers and placed into separate controller files. These methods have been exported and are called within the associated route. When the client side sends a request to the specified route the handler function is executed and returns the results of the request back to the client as a JSON object, in which the front-end manipulates this data and displays it to the user.

By walking through one example of how this process takes place within the application it will explain how routes and controllers have been used. It should be noted that while this is only one example, routes and controllers have been set up throughout the application and allow the client to get, update and delete information.

In order to allow the user to view all employees the *employee-list* component makes a call to the *getEmployees* method. The *getEmployees,* inside the employee service makes use of Angular’s HTTP client to send a GET request to the URL http://localhost:3000/api/users/. Express listens to this this request and a get route within the user-routes file has been created to respond to this request. The handler-function, in this case *getUsers*, is executed. The *getUsers* method uses Sequelize to query the database and returns the results from the database query as a JSON object. However, if no users were found or there was an error with the server it would respond with the corresponding message. If successful, the results are returned to the client and the process described in section *5.3.1* is carried out and all the employees are displayed to the user. This process of sending and responding to requests has been used throughout the created application and is the main driving force in how the user can interact and use functionality of the system.

A screenshot of a cell phone

Description automatically generated

Figure 47 – Route to return all users inside the user-route file.

A screenshot of a cell phone

Description automatically generated

Figure 48 – getUsers controller.