**ERSWebApp Documentation**

**Installation**

The ERSWebApp depends upon the following being installed before it can be used:

* Visual Studio 2017 – with Node.js module installed
* ASP NET Core 2.1/2.2 SDK
* SQL Server Management Studio (SSMS)
* Node.js (additional setup notes before)

There are a few additional steps required to run Node.js on the current work environment. Firstly, Node.js’s installed directory should be within your “C:\local\_it” directory along with the “Add to PATH” option checked during installation (this can be added manually if missed during installation).

After this, the Node config file has to be set up to make sure it has the correct WBS proxy settings allowed long with ensuring all packages are installed within the “C:\local\_it” directory rather than the user profile’s AppData folder by default as this causes issues with the network profile space. To achieve this, create a file within BOTH your normal AND admin accounts “Documents” folder called “.npmrc” and input the following lines:

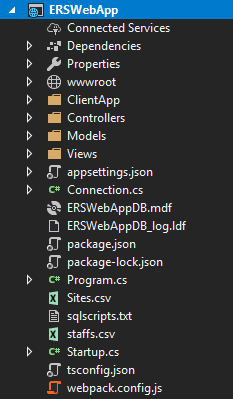


The easiest method of creating a file writing is to open a text document using Notepad, copy the lines above in, go to “Save As”, set “Save as type” to “All Files” and set the name to “.npmrc”. As mentioned, it’s important to do this for both of your normal AND admin accounts as the building of npm will be done through your normal account but the downloading of packages will be through your admin account. To test this is set up correctly, use the command “npm --version” in PowerShell for both normal and Admin versions and you should retrieve the version number (6.4.1 at the time of writing).

Once set up, we need to restore the NuGet and npm packages the web app depends upon. To restore the NuGet packages, cd to “C:\Source\ERSWebApp\ERSWebApp” within Admin Powershell the “npm install” command which will download all of the required node packages. The easiest method of restoring NuGet packages with the current work environment is to open the solution file “ERSWebApp.sln” in Visual Studio, go to “Project -> Manage NuGet Packages…” and check if any packages have to be restored. The NuGet packages the web app requires are Dapper, Microsoft.AspNetCore.App, Microsoft.NETCore.App and Newtonsoft.Json.

With all the packages restored, it’s time to run the app for the first time. The “IIS Express” method of launching is preferred and any browser OTHER than IE as a more modern browser such as Edge/Firefox/Chrome is required to run the newer JavaScript Vue depends upon. Run the app with the IIS Express method and wait for it to build. It’s worth nothing the app MIGHT time out during the first time it’s built which will result in a timeout error of the browser. This is normal as the first build can take some time to complete. If this does occur, simply stop and rerun the app which will fix this.

**ERSWebApp Structure**



Overall, the structure and design pattern for the ERSWebApp is exactly like that as the TemplateWebApp. It is recommended to read through that documentation first as this documentation will be concentrating on ERSWebApp specifics. The differences here are that the MSSQL database files are included and two csv files (Sites and staffs) included to easily input in the Site and testing Employee data if required. A text file “sqlscripts” is also included which contains the required SQL scripts to run in order to build the relevant database if required.

**ClientApp – Components**

**App Component**

Overall, the App Component for the ERSWebApp is the same as that found within the TemplateWebApp except a “SelectedDate” object is used to act as a Vue Prop (<https://vuejs.org/v2/guide/components-props.html>) which allows simple data to be passed between components using the router-view. The “date” property of the SelectedDate is converted to match a “YYYY-MM-DD” pattern:





**FetchSession Component**

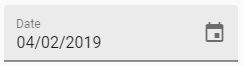
The SelectedDate Prop is accessed within the component’s script and can then be used within the v-date-picker (Vuetify Calender):

****



The text of the date picker is formatted so that it follows a “DD/MM/YYYY” pattern so that the date is more easily readable to the user but the actual date is still in the “YYYY-MM-DD” pattern preferred by the backend:





This date is then used within the “loadSessions” method to return all Sessions on that date by calling the GetSessions method of the Session Controller:





This method must ensure that the List<Employee> property of each Session has to be populated. This is currently done by using the Session Id to retrieve all of the Employees from the SessionEmployee Table for each of the Sessions. This could probably be improved upon using a Foreign Key relationship but this works for now.

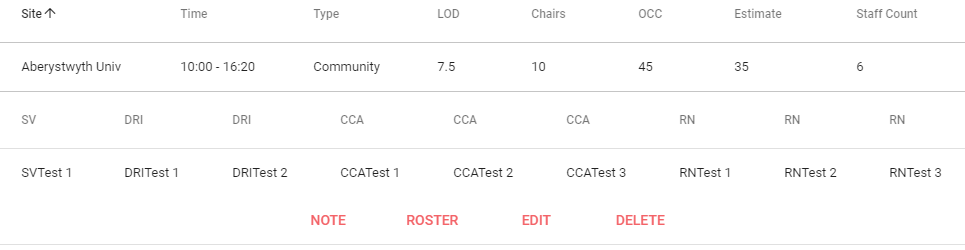
The general design of this component is similar to that of the FetchEmployee component in the TemplateWebApp. One additional is how the colour of the text for the Session’s Site property is based upon the Session’s State with it displaying as ‘gray’ if 0 (incomplete), ‘black’ if 1 (default) or ‘red’ if 2 (used as error in testing and left in if needed).



Additionally, the information within the expand slot demonstrates how to produce a table using a v-for loop only if the length is over 0:

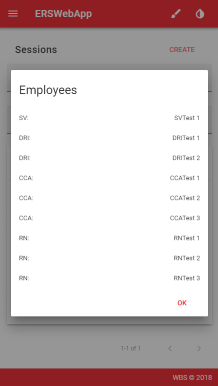


This enables the table presented within the expansion slot to reflect only data that Session object contains:



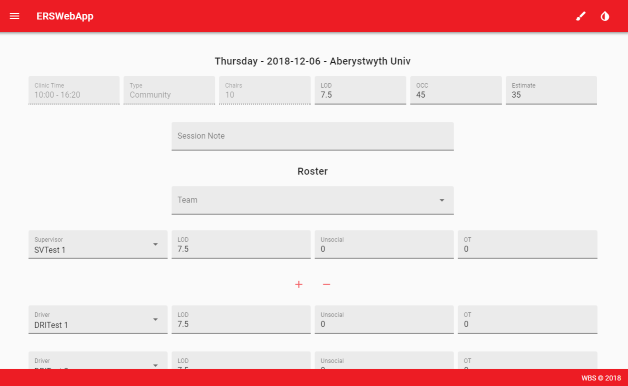
As this wouldn’t be possible when viewing the web app on a mobile device, a Dialog is used using the a v-for with a v-list-tile instead to present this information:





**RosterSession Component**

Upon selecting the Roster option within the FetchSession component, you will be directed towards the RosterSession component. This is the component in which the main function of the ERSWebApp is contained: rostering employees to sessions.



First, let’s look at what happens within the component’s “mounted” method:



First, the loading bool is set to true which displays a progress circle to indicate that the data for the component is loading to the user:



The current session selected from the FetchSession component is then “fetched” using the Session Id from the route parameters. The response is then processed with a “before” and “after” copy saved as a before and after are later sent back to the SessionController in order to easily determine any changes made. The “JSON.parse(JSON.stringify(data))” method is used to create a deep copy of the data as otherwise only a shallow copy could be created as everything is a reference in Java/TypeScript.

Following this, the “Holiday” property of the Session is then checked as a different layout for this component is used if true. During the writing of this documentation, there Unsocial Hours fields are not necessary when the Session is deemed as on a Holiday (weekend/special date such as a bank holiday) as all Appointed Hours on those Sessions are already deemed as Unsocial. For example, here is the first text field for each layout to highlight how the Unsocial field (shortened to “UNS”) is not present when the “v-if=holiday”. Below is when the holiday is not true:



And then if holiday is true:



As shown, the Unsocial Hours v-text-field is not present when “v-if=’holiday’”. The v-flex responsive classes are also changed from “md3 sm6 xs12” are changed to “sm4 xs12” in order to ensure the three text fields are presented appropriately compared to four text fields.

Following this, the “filterSessionRoles” method is run in order to filter any Session Employees into their respective text fields:



The Sessions List of Employees is first looped through with each of the employees being placed within their respective role array. The length of these are then checked to see if any are under 1 with a blank employee being pushed into that array so the text field is still displayed. The LOD property of the blank SessionEmployee objects are set to match the Session’s LOD:



The “loadAvailable” method is then run to retrieve all of the employees which are available to be rostered for that appointment by passing the Session’s date and day properties to the GetAvailable EmployeeController method:





This method first creates an empty List<Employee> followed by a loop which goes through each Employee returned by the “GetEmployees” method with the date parameter passed through. This is then checked with the Employee’s WorkPattern to see if they are able to work that day:



This method retrieves all of the Employees from the EmployeeTable and then runs the GetStatuses method along with the GetAbsences static method from the AbsenceController using the date as a parameter:



The static version of the GetAbsences method retrieves all of the Absences whereby the date is in between the Absence StartDate and EndDate which can then be used for the GetStatuses method:



This method simply loops through both Employee and Absence lists, sets the Status to the Employee if there are any matches and also appends “ – Part” if the Absence is a Part one meaning only part of the day is an absence resulting in them still being able to be rostered for that day.

After this, the employees array within the RosterSession component is then filtered in a similar fashion to if there were already employees rostered for that session but these are places within arrays that the user chooses the roles from:



Finally, the Teams are fetched in which calls the GetTeams method of the TeamController:

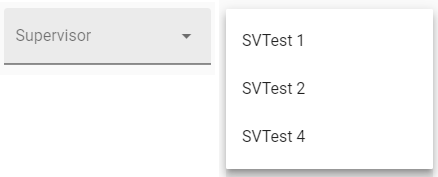




Similar to the GetSessions method, the Team is first retrieved from the TeamTable followed by each TeamMember being retrieved to fill the Team’s List<TeamMember> property.

The RosterSession component allows allocation of employees to the session either manually from the v-select or they can assign them through from the Team v-select. As displayed previously, the user is able to select a user from a dropdown style component called a “v-select” with the Supervisor below as an example:





As shown, the v-select is bound to the “employeeId” of the “sv” item within the “v-for=’sv in sessionsvs’” (as in this Employee with the role of SV within the sessionsvs array), the “:items” bound to the “svs” array (which are all of the Employees available on that day with the role of SV”, the “item-text” set to “name” so that the Name of the Employee is presented but the “item-value” set to “id” as only the Employee Id is requried. Note, SVText3 is not present as they did not return from the available array either because their WorkPattern doesn’t match the day given or they have an Absence. If the user wishes to add or remove an employee from their respective role array, they can use the + and – Icon buttons below each role section:





Clicking on these will trigger their respective method which simply either adds or removes an employee from their chosen role array:



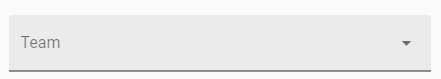
This is also where the RosterSession Component is able to apply a limit to how many of each role is able to be rostered to a session which is currently:

* Supervisor – 2
* Driver – 2
* CCA – 5
* RN – 5

This will need to be updated and can be used alongside the Session’s Chair property to apply a “Chair Model” to the sessions whereby the amount of each role is relevant to the amount of chairs that session has.

As mentioned, the other method the user can roster employees to the Session is to use the Team v-select:





It’s important to note that his v-select differs in that it uses the “return-object” attribute to ensure that the entire Team object is returned during the “v-on:input” attribute rather than just the name. When a user selects a Team, the “setTeam” method is triggered as indicated by the “v-on:input”. For this method, the session role arrays have to first be cleared so that the team be replace any employees already rostered. Each member of the selected team is then looped over to see if their Id matches any within the available employees array which means that team member is available for that date:



If a match is found, this team member is then put through a switch statement to determine its role and also converted from a TeamMember object to a SessionEmployee object so that the employee can be assigned the various Session related hours. If there aren’t any matches, the session arrays are then populated with a blank employee to ensure that the user is able assign an employee for that role.



When the SUBIT button is pressed, the “rosterSession” is called which uses a method called “checkDuplicates”. This method simply goes through each of the selected employees and alerts to the user if there are any duplicates chosen with the use of the error Snackbar with the text being bound to the “errorMessage” string script variable:





This is the script for the “checkDuplicates” method for the SV role which is performed for the other roles too. Below is the “rosterSession” script:



As mentioned, first the employees are checked for any duplicates. The “after” Session’s Employee array is then populated by looping through the session role arrays such as for the SV role:



A Session array is then created with the “before” and “after” session put into so that they can be sent to the RosterController Update method. As usual, the response is then returned to determine if the update was successful or not. The Update method of the RosterController will be broken down next as quite a lot happens within this.



The Session array from the RosterSession Component is first deserialized so that differences made between the “before” and “after” Sessions can be checked. The Holiday property of the Session is then checked to determine which UpdateRoster method is to be performed. During the writing of this documentation, Sessions set on a Saturday as classed as “LowRate” meaning that all appointed hours for that session are classed as “LowRateUHours” (as in low rate unsocial hours). Sessions set on a Sunday or “Special Date” (bank holiday or date which is input into the SpecialDateTable) are classes as “HighRate” and so appointed hours for that are classes as “HighRateUHours”. This is done as requested due to the amount of money given per hour is different depending upon the type of “holiday” (as in weekend of special date) the session is on. The UpdateRoster first retrieves which week of the year the Session is set on as a “Week.Year” double is used to identify which week the rostered hours are on within the RosterTable. The method for achieve this uses a combination of the “DateTimeFormatInfo” and “Calender” classes:





First, the Week double is retrieved along with the Session’s StaffCount property reset to 0 so that it can be recalculated (this property was requested in that the count of the non RN role employees are displayed). Following this, each of the Employees within the Before Session are looped through to see if they match any of the Employees within the After Session. If a match is found, the LOD (length of day/appointed), UNS (unsocial) and OT (overtime) hours are then checked to see if any differences are found. If so, the difference is calculated and updated using the appropriate method e.g. UpdateAppointed if there is a difference in appointed hours. This Employee is then removed from the Before Session Employees List as they are no longer needed to be looped through during the After Employees loop. If no match is found then that Employee has all of their hours assigned to the RosterTable using the UpdateHours method:



This method appends the RosterTable with the appropriate hours of the Employee if there is a match. If there is no record of this employee within the table, the CreateRoster method is used to create a record.

After this, the Session’s StaffCount is then incremented unless the Employee’s Role equals to RN as requested. Following this, the Before Employees List is then searched through again as and any Employees’ hours present are updated with their hours as negatives from the RosterTable as this indicates that they are no longer being rostered to that Session resulting in their hours being removed.

The UpdateRosterLowRate and UpdateRosterHighRate methods are very similar except that their appointed (LOD) hours are also used for the unsocial hours of the RosterTable such as seen with the UpdateHoursLowRate method:



After the hours have be rostered accordingly, the State of the Session is set depending upon the Session Type and StaffCount. This is currently done so that the Session State can be used to determine which colour to set the Text in the FetchSession Component as described here. This is also where any additional States need to be implemented to the Session such as under or overstaffed.



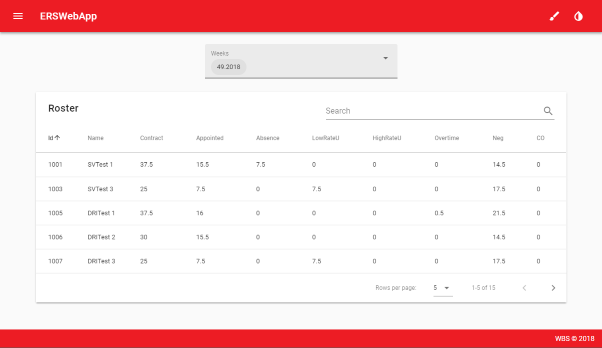
Following this, the SessionEmployeeTable has to be updated so that the correct Employees are linked to the respective Session. It’s worth noting that this current method could be improved upon using a MERGE statement. For instance, the method simply removes all previous records within the SessionEmployeeTable from that Session and then adding in the Employees again to ensure all changes are set. Although this works, this can cause fragmentation within the database which a MERGE statement could solve:



The rows of these are then returned in order to inform the front end if the UpdateRoster was performed successfully or not.

**FetchRoster Component**

The FetchRoster Component is where the user is able to view all of the rostered hours from the employees either by single or multiple weeks.



During the start-up of this component, the GetRosterWeeks method of the RosterController is fetched which simply returns all of the unique Week values in that column using the “Distinct()” Linq method and then places into the weeks array in the component’s scripts:





The v-select for this component uses the “multiple” attribute to allow multiple weeks to be selected and the “chips” attribute to show these weeks within the v-select in the “chip” style.



As shown, acting an input onto the v-select calls the “loadRoster” method which calls the GetRoster method of the RosterController. This method will be broken down so each part can be detailed:



The “selectedWeeks” array from the FetchRoster Component is first deserialized into a List<Double> which is then loops through in order to allow the user to retrieve the rosters for multiple weeks. An empty List<Employee> called “employees” is then created. The Employees within the RosterTable where the week matches are then retrieved and placed into a separate List<Employee> called “temp” which is first looped through. The “temp” List is temporarily used for each week within the selectedWeeks with and is first looped over to calculate if the employee has and NegHours (as in a negative difference when comparing the Contract and Appointed Hours indicated they worked under their ContractHours), and COHours (as in a positive difference when comparing indicated they worked more than their ContractHours):



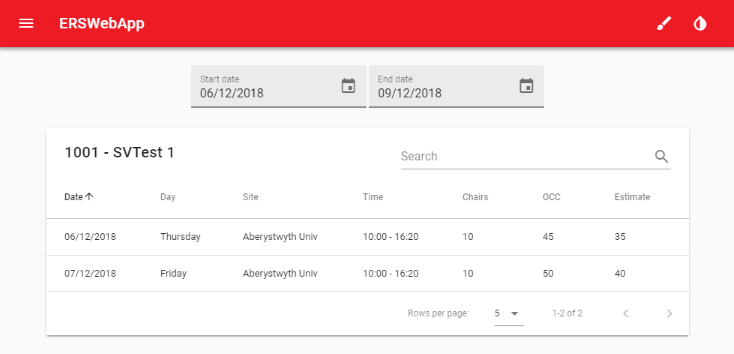
After this, the “employees” List is checked through to see if there are any matching employees within the “temp” List as their hours would have to be added together. If not, the “temp” employee is added to the “employees” List:



The “employees” List is then returned to the FetchRoster Component so that it can be used to populate the v-data-table. Note, the method for calculating the NegHours and COHours may be changed as discussion during development of the ERSWebApp indicated this.

**ViewEmployee Component**

This Component’s function is to allow the user to view which Sessions an Employee is between two given dates. Below is an example when viewing the “SVTest 1” employee which was selected by going to the Employees route in the navigation menu, clicking on the top employee “SVTest 1” causing the row to expand and then clicking on the VIEW text button presented:



As shown, the ViewEmployee Component uses two v-date-picker menus to allow the user to input the Start and End date range they wish to view. The “loadSessions” method is called when input:





First, the dates are checked to be not empty (“”) and then if the End date is later then the Start date with an error Snackbar being used to display if the dates are invalid:



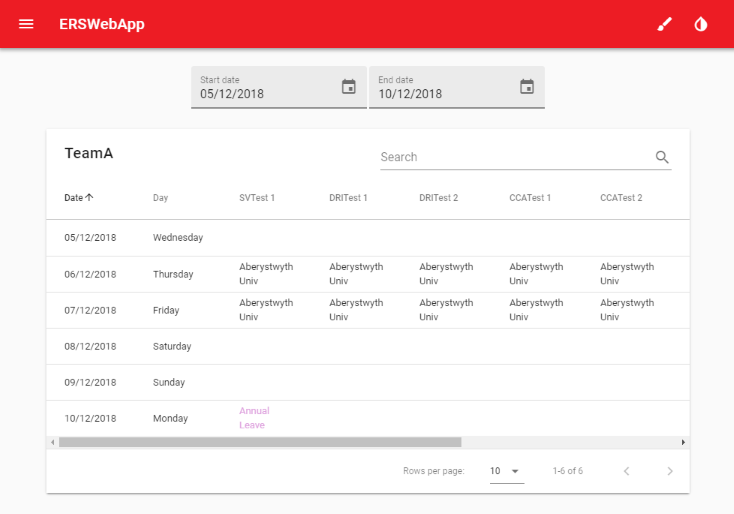
After this, these dates along with the selected Employee’s Id are sent to the GetEmployeeSessions method of the Employee Controller:



First, a List is created of all the matching Session Ids within the two dates where the Employee Id is matched from the SessionEmployeeTable meaning that the employee is rostered to those sessions. Each of these Session Ids are then looped through to retrieve that Session to create a List<Session> which is then returned to the ViewEmployee Component and populates the v-data-table.

**ViewTeam Component**

A similar method is used for viewing where each member of a Team is between two given dates:



As the TeamMember’s names are used for the columns in this table, the headers and columns were programmed to be dependent on the TeamMember List. The first step towards achieving this was to use a v-for within the v-data-table td row for each of the members:



The style of the text used is also passed through a method which returns a matching Absence colour which was a requirement for the ERSWebApp:



In order for the table’s headers to display the correct name, the headers array has these names pushed onto it during the “mounted” method along with its value being set to the “employeeSite” so that it will display the site/absence type for that TeamMember:



The selected dates are sent to the GetTeamSites method of the TeamController in the same method done for the ViewEmployee Component. The GetTeamSites method will be broken down:



The relevant Team from the “id” parameter is retrieved using the GetByIdStatic method of the Team Controller which simply retrieves the Team object based on the Id. Two Lists are then created: “dates” used for storing which dates to search through for each of those dates and “teamsites” which will be used to input the TeamSite (class which stores the date, day and SessionEmployee List of the Session which the ViewTeam Component’s v-data-table uses). Each of these dates are then looped through:



A TeamSite object is first created using the date to populate its Date and Day properties along with a List of SessionEmployees. Each TeamMember of the team’s Members property is then looped over whereby a SessionEmployee object is created with its EmployeeName set to that TeamMember. A list of the SessionSites are then retrieved using that EmployeeId from the SessionEmployeeTable (as an employee could be rostered to multiple sessions). If this list contains items then these are simply concatenated with a “, “ using the “string.Join” method. If not, the Status of the member is then retrieved using the GetEmployeeStatus method of the AbsenceController. In this method, the member’s Id along with that date is used to see if that member has an Absence for then. If so, this is returned also with “– Part” if the Absence was a PartDay. If not, a blank string is returned (“”) which is then used to indicate to the user that that member of the Team is neither rostered to a session or has an absence on that day:



**FetchAdmin Component**

The FetchAdmin Component allows the user to control various classes to which the rest of the components rely on such as sites, special dates and employee skills. For the most part, this component is very similar to other Fetch ones except a single delete v-dialog is used along with a switch statement to determine which section of the component is chosen to be deleted. This is handled with the respective “open…Delete” method for each class. Below is the code for the single v-dialog along with the “openSiteDelete” and “deleteSwitch” methods are an example:







The switch will then trigger the corresponding delete method which will trigger the matching Delete method without the AdminController.