VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

Neil Rice

Prima Power Analytics Application

Using React.js

Information Technology
2018

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Degree Program in Information Technology

ABSTRACT

Author	Neil Rice
Title	Prima Power analytics application
Year	2018
Language	English
Pages	65
Name of Supervisor	Timo Kankaanpää

Over the last decade companies in the manufacturing industry have become very efficient at gathering large amounts of data from machines. The problem many companies face is how best to use this data for analysis in order to improve production and essentially save money. Reporting dashboards have become increasingly popular as a way to display data, however the analysis is still mainly left to the human operator of the dashboard. With the current advancement of machine learning and data visualization tools, future data analytic tools can rely more on technology, giving the human operator more time to do other tasks.

The aim of this thesis work was to design and develop a front end web application for the company Prima Power to demonstrate at the 2018 EuroBlech trade fair. The application, made with the javascript framework ReactJS was to be modern in design, fully functional and include a number of data analytic visualizations as well as a notification system demonstrating machine learning outcomes.

The thesis work was completed with the finished web application meeting all the set requirements and was presented at the 2018 EuroBlech trade fair in Hannover, Germany.

Keywords React, redux, analytics, web application

Table of Contents

List of figures and tables

List of abbreviations

JS	Javascript
MIT	Massachusetts Institute of Technology.
NPM	Node package manager
JSON	JavaScript Object Notation
JSX	An XML like syntax extension to ECMAScript
HTML	Hypertext Markup Language
UI	User interface
UX	User experience
DOM	Document Object Model
ES6	ECMAScript 6
CSS	Cascading Style Sheet
SVG	Scalable Vector Graphics
AI	Artificial Intelligence
UML	Unified Modeling Language
PC	Personal computer
API	Application programming interface
URL	Uniform resource locator

1. Introduction

In recent years it has become more and more common for companies manufacturing machines to offer ongoing digital services to their customers to help technicians fix problems and also to give performance information usually delivered through some kind of a reporting dashboard. Although these reporting dashboards can give a good overview of the machine data and give a general idea of where problem areas are, they do not usually provide detailed analytic tools which give the user the ability to prevent errors occurring without the need for time consuming manual analysis of data. As the quality of machine learning technologies are currently at a point where they can be utilized to analyze large sets of data to detect patterns and anomalies, companies are starting to look at what more can be done to get the most out of their data collections and provide their customers with tools to make their production more efficient. Prima Power, a manufacturer of laser and sheet metal machines is one such company and will serve as the client for this report. This thesis shall show the front-end development of an analytic React.js web application made to be presented by Prima Power at the 25th International Sheet Metal Working Technology Exhibition, EuroBLECH 2018. The main purpose of the application is to market current and upcoming tools to be offered as a service to customers.

1.1 Prima Power

Prima Power is one of the largest manufacturers of systems and machines for sheet metal working with a global customer base. Prima Power is the brand created by the merging of Finn-Power OY based in Finland and Prima Industrie group based in Italy. The product range covers all stages of the sheet metal working process, covering all applications: laser processing, punching, shearing, bending and automation. Prima Power offer extensive services and support to customers worldwide. As part of offered services, Prima Power have large data collection on each customer machine, storage of the data and reporting tools to help analyze performance and production. /1/

1.2 Euroblech

Euroblech is a trade fair for sheet metal working taking place once every two years in Hannover, Germany. Over fifty thousand visitors from around the world attended the 2018 Euroblech trade fair to view the latest innovations of over 1500 companies from 40 countries./2/

2 Technologies used

2.1 Node.js

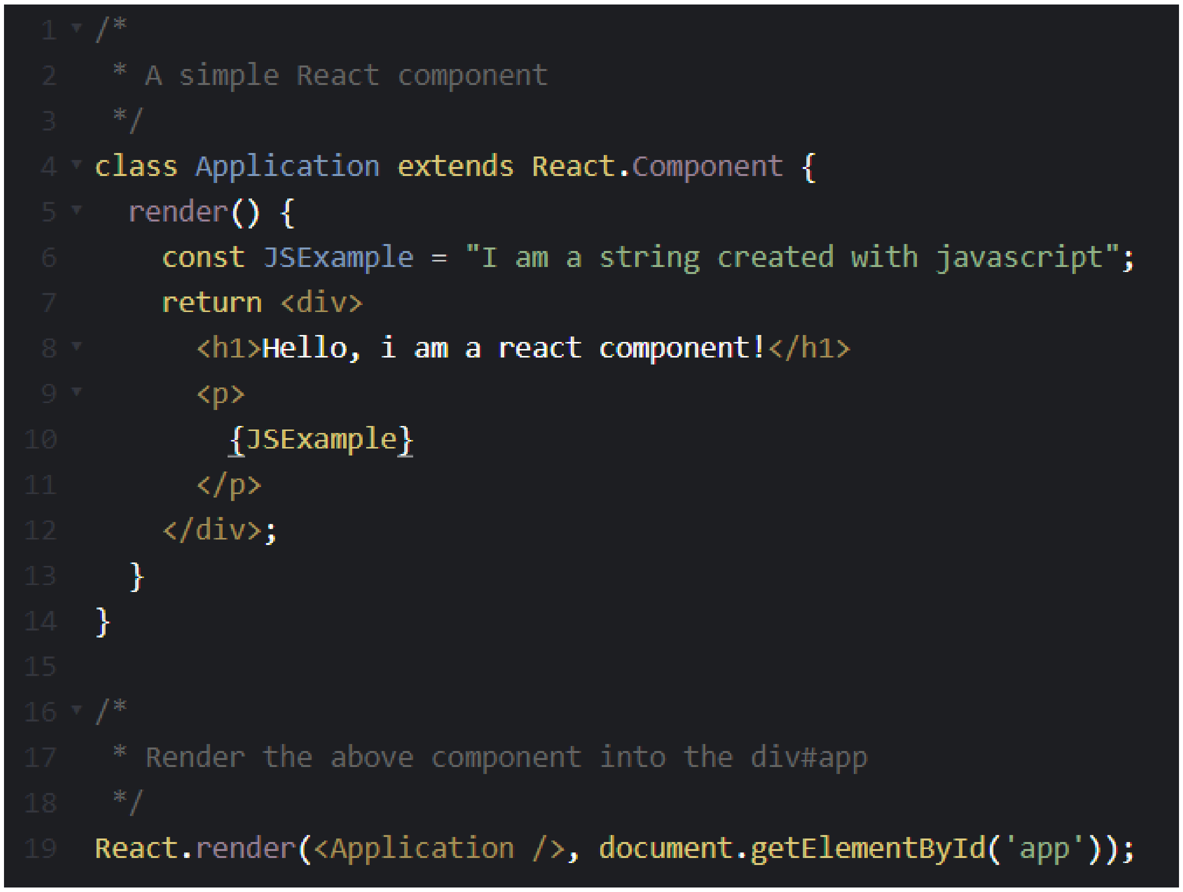
Built on the Google open sourced V8 javascript engine, Node.js is a run-time environment which executes javascript code without the need for a browser. It was first developed by creator Ryan Dahl in 2009, is open source under MIT license and works on all the widely used operating systems. /3/

Along with the fact that javascript is a very commonly used language in modern web development, node.js is very popular due to its high performance which can be attributed to the fact it uses event driven programming. Instead of using sequential programming the event driven programming model uses callbacks to report when a task has finished which eliminates the need for threading. Node.js can be used for command line tools, desktop applications and backend web applications but for front-end developers it mainly used locally to create a development server where development tools such as webpack and babel can be used. /3/

In modern web development modules are used as a simple way to share and re-use code. NPM standing for ‘node package manager’ has a registry with close to half a million packages available to use with a simple command using the command line. A package.json file keeps track of all installed packages, version numbers and any dependancies. There are other package managers also available to use with node.js, however NPM is the default package manager used by node.js. /3/

2.2 ReactJS

Originally created by a Facebook engineer named Jordan Walke, ReactJS, commonly known as simply React, is an open-source front-end javascript library used to create user interfaces for web applications. React applications are made up of components which are most commonly created using JSX syntax. Rather than keep the html and javascript separate the JSX brings them together to give the developer a more powerful and secure way to create a section of the UI. The separation of concerns is achieved by the fact that each part of the UI is its own component. Components can be placed anywhere in the application, any number of times and they can have any number of sub components. The JSX syntax looks like a HTML templating language but has all the capabilities of javascript, producing elements that are added to the DOM. Figure 1 is an example of using JSX syntax to create a simple component. /4/



```
/*
 * A simple React component
 */
class Application extends React.Component {
  render() {
    const JSExample = "I am a string created with javascript";
    return <div>
      <h1>Hello, i am a react component!</h1>
      <p>
        {JSExample}
      </p>
    </div>;
  }
}

/*
 * Render the above component into the div#app
 */
React.render(<Application />, document.getElementById('app'));
```

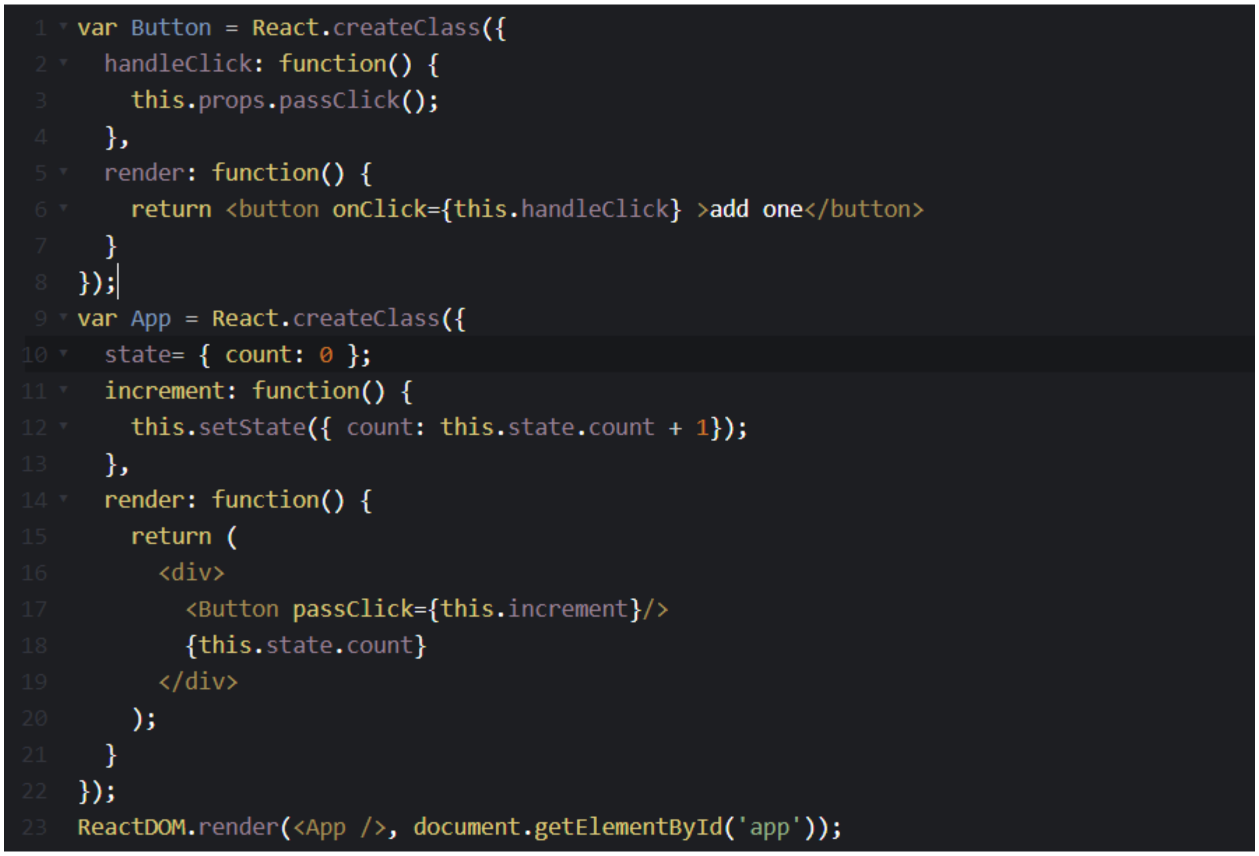
Figure 1. A React component example

2.2.1 React virtual DOM

React uses something very powerful called the virtual DOM which is a lightweight copy of the actual DOM. As each virtual DOM object is only a representation of a real DOM object it is very fast to update and so each time a JSX element is updated the whole virtual DOM is updated. Once the virtual DOM has updated React can compare it with a snapshot of the actual DOM and check for any differences. Only the elements with differences found are then redrawn on the actual DOM then which gives the user a very fast and smooth experience. /4/

2.2.2 One way data binding and flux

As React applications are built on a series of nested components, data is passed down through props (properties) from a parent component to its child component. This can be thought of as using javascript functions where an argument is passed in and this same logic gives the developer a way to pass data from the child back up to the parent using callbacks as seen in Figure 2. /5/



```
var Button = React.createClass({
  handleClick: function() {
    this.props.passClick();
  },
  render: function() {
    return <button onClick={this.handleClick} >add one</button>
  }
});
var App = React.createClass({
  state= { count: 0 };
  increment: function() {
    this.setState({ count: this.state.count + 1});
  },
  render: function() {
    return (
      <div>
        <Button passClick={this.increment}/>
        {this.state.count}
      </div>
    );
  }
});
ReactDOM.render(<App />, document.getElementById('app'));
```

Figure 2. An example using a callback in one data binding

Using callbacks works perfectly for simple applications like the example in Figure 2 but as applications increase in complexity the callback method becomes increasingly difficult to maintain and so a cleaner solution is needed which is achievable using Flux. The Flux pattern was designed to control how data is shared between components in an application by using a type of observer pattern. The idea of the Flux pattern is there is one place within the application that holds all state information which are called stores and these stores can only be updated by using actions. The actions call on a dispatcher and so then the stores can in turn subscribe these actions to update the stores state making it simple for the developer to know what, why and how the change in the data is happening. /5/

2.3 Redux and React-redux

The Flux design pattern gives the developer an efficient way to manage state in a React application, however it is not an out of the box library or framework but rather an architecture design pattern. To make the lives of developers easier a programmer named Dan Abranov created Redux which is a library using some of the main ideas of Flux with some added features improving the management of application state with the two most valuable of these features being reducers and middleware. Reducers are functions that determine the way in which the state is affected by actions. The reducer is called from the store each time an action is dispatched and whatever the reducer returns replaces the internal state. Middleware in React gives a point between the actions and reducers where third party code can be added. Redux can be used in React applications by installing and importing the react-redux module. /6/

2.4 Redux-thunk

When using react-redux action creators return an action object which is limiting as it results in the action creator running synchronously. Redux-thunk is a middleware which gives the action the possibility to return a function which has the dispatch function of the store. This means actions can be conditionally dispatched for example they can wait for an asynchronous task to finished before dispatching which makes redux asynchronous scripting a lot easier to achieve and manage. /7/

2.5 Create-react-app

During the last few years one of the biggest problems a frontend developer (especially a beginner) faces is the fact that technologies are changing at a rate that the developer cannot keep up. The tooling, for example, is constantly improving and whilst this can make applications easier to maintain (in some cases even improve performance), the tooling still needs to be installed and understanding how to do that is not always simple. To just begin a new React app many of these tools need to be installed and configured which can be tiresome and frustrating and so the React team created Create-react-app which is a command that creates a new React project with all the build tools installed and configured including webpack, babel, jest and yarn. Babel is the transpiler which allows the developer to include ES6 syntax which is not yet browser standard and webpack is a tool that bundles modules ready to be used in the browser. /8/

2.6 KendoUI-react

KendoUI- react is a framework of components which can be imported into any React app ready to use. The components can follow a default style theme or a new theme can be create on the KendoUI website. Examples of components include grids, date pickers, navigation and more advanced UI components such as charts and maps. /9/

2.7 ReCharts

ReCharts is a chart library built on React components. The components are decoupled and easily reusable so creating custom elements of the chart for example a custom tooltip is made simple. /10/

2.8 D3.js

A javascript data visualization library which uses SVG and other modern web standards such as HTML and CSS. Where other libraries and frameworks make a lot of the decisions for the developer, D3.js gives the developer the freedom to create truly unique interactive visualizations making it a very powerful tool. /11/

2.9 Chap-links timeline

Chap-links timeline is a javascript library which is used to visualize data over a period of time. The visualization can be zoomed and panned and the time/dates are automatically scaled. The timeline supports data which takes places on a single moment down to a millisecond and also supports data which can have a start and end time. /12/

2.10 Twitter bootstrap CSS

The twitter bootstrap framework provides responsive classes which can be combined to create professionally styled elements such as grids, panels and modals. The bootstrap column classes make creating responsive layouts simple. /13/

2.11 JustInMyind

JustInMind is a UI prototyping tool which gives the user the possibility to create interactive mockups for example clicking on a navigation item can actually change the mockup to navigate to a mockup up of another page. /14/

3 Analysis

3.1 Requirements

3.1.1 Table of requirements

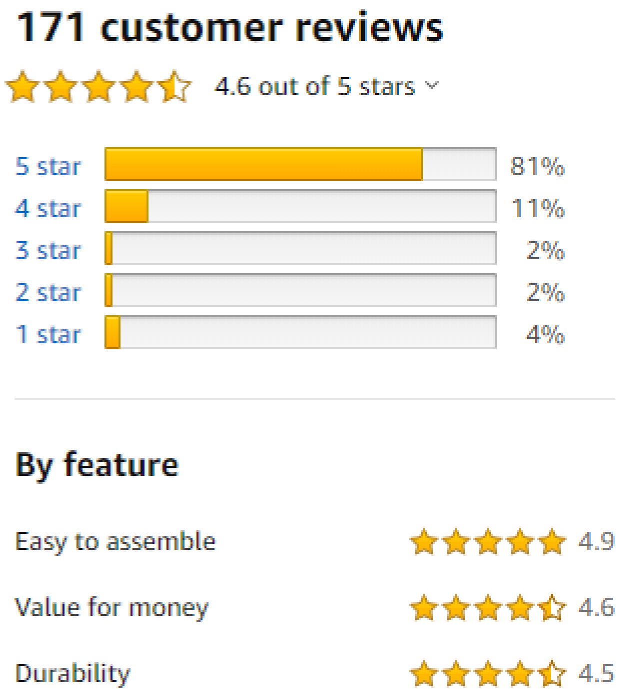
Table 1. Table of requirements

Requirement	**Importance (1 being the highest)**
Use React.js	R1
Professional, modern, responsive design	R1
Shift statistics view	R1
Notification system	R1
Root cause view	R1
Create realistic mock data	R1
Advisor view	R2
Calendar view	R2
Alarm view	R2
Messages component	R3
Timeline view	R3

Add note to timeline capability	R4
Maintenance view	R4
Scheduler popup	R5
Performance view	R6

3.1.2 Professional, modern, responsive design (R1)

Often in the beginning stages of developing an application, functionality is of higher importance than the design of the UI and UX. As the purpose of this application is to be a presentation to existing and potential future customers, it is important that the UI and UX have a finished look and feel. The UI should look clean and uncluttered on all screen sizes and devices as well as having a uniformity across views and individual elements. The UX should feel simple and natural to use and give a first-time user a sense of familiarity with the navigation and user inputs. The UI should include the Prima Power logo and consequently the color scheme should work well with the company logo colors. Data visualizations should use a consistent color scheme where possible. The online retail store Amazon is a good example of using a color scheme for the UI and data visualizations which incorporate the company brand colors. As seen in Figure 3, colors from the company logo (Figure 4) are used in the data visualizations which in this case is a rating system. The purpose of creating a uniformed color scheme is to increase readability and legibility of the content being displayed and to give a comfortable experience for the user.



171 customer reviews

4.6 out of 5 stars

5 star		81%
4 star		11%
3 star		2%
2 star		2%
1 star		4%

By feature

Easy to assemble	4.9
Value for money	4.6
Durability	4.5

Figure 3. Screenshot taken from the amazon.com webpage.

amazon

Figure 4. Amazon logo

3.1.3 Shift statistics view (R1)

A meeting was held with the Prima Power web development team and a group of employees of one of Prima Power's customers. The main points on the meeting agenda were to gather information from a customer point of view and to understand better what a typical customer would find useful in an application, why they would find it useful and how to implement the features. One of the outcomes of this meeting was that there should be a view that gives the statistics and important information of the previous shift. The main purpose of this view would be for the factory manager to see how the night shift had gone as soon as they open the application at the start of the day. The view should include machine utilization averages, a performance history chart, a message center and the option to drill down to machine level more detailed statistics and information. When drilling down to machine level, the view should show a timeline of the shift for each device, a table of stoppages, a chart showing top alarms and the last known status of the machine. The user should be given the option to change which date and which shift are being displayed. The default date should be the current date and the default shift should be the previous shift.

3.1.4 Notification system (R1)

As machine learning technologies are being developed at Prima Power, the customer will soon have the option to learn of any anomaly analysis and error prediction of their machines. This information will be able to be viewed in many ways such as email reports but will also be included in the analytics application. Rather than have the information displayed on a page and give the user the responsibility to check for updates, the application will include a notification system where the user will see a color change of a notification icon to notify them there is a new notification. Additionally the notification system should also notify the user of any new comments or messages.

3.1.5 Root cause view (R1)

As the machines themselves are quite complex with many stages of the manufacturing process it means that problems can occur in all different stages and within different sub stages of those stages. Reading the error data may be helpful for solving each individual problem but for trying to analyze large sets of data to understand why a certain problem is occurring, a different approach is needed. A data visualization is needed which can show the errors in the stages and sub stages in which the error occurred. The purpose of the visualization is that the user can look at large time spans of data and instantly see in what stages are occurring the most errors.

Additionally the alarm data which correlates to the error data should also be displayed. The user should have a number of options when selecting the data to be displayed including date range, shift name, line name and device. The default date range should be starting one week from current date and ending on current date. The default line will be the first in an alphabetical list. The default shift option will be ‘All shifts’ and the default device option will be ‘All devices’.

3.1.6 Creating the mock data (R1)

The application must demonstrate the full capabilities of the Prima Power machine data collection and machine learning tools. For confidentiality reasons the machine data cannot be taken from an existing customer and so the data must be taken from the Prima Power test machines. As the data on the test machines comes from testing, it means often there can be days without any data input. To create the mock data, first data on the test machine which is similar to real scenario must be found and then replicated and modified to give a six month range of data for two assembly lines and both with two devices each. The machine data types required are triggers, alarms and machine state descriptions. As the Prima Power test machine which would be used for data samples is not used on a consistent basis, it means that the data is not sufficient to give optimum machine learning results. As there are no sufficient machine learning samples to use for this application then the machine learning results will have to be fabricated for demonstration purposes.

3.1.7 Advisor view (R2)

This view is to show the notification history in a grid format. The notifications shall be divided in to two sections, AI notifications and user generated notifications. AI notifications will show any anomaly detection and future error detection such as tool wear out. User generated notifications will show notifications such as ‘comment added to timeline’ and ‘new message’. The AI generated notifications will also include advice or a remedy for the user to act on. The grid cell background colors will show the severity of the notification for example red representing a critical problem.

3.1.8 Calendar view (R2)

The calendar view will be an interactive scheduler with day, week, month and agenda views to choose from. The calendar will display upcoming tasks needed to be done for example maintenance tasks and in addition show AI generated predictions such as tool wear out. The view should include filtering capability of different assembly lines as well as the option to filter different tasks and error predictions.

3.1.9 Alarm view (R2)

The alarm view should consist of bar charts of two different types. The first will show the top ten alarm number frequencies. A method will be needed to determine the alarm frequencies. The second chart type will be for top ten alarm duration. Each alarm has an alarm duration property and so for each alarm number, the duration of each alarm will be added. By default the view will display both the frequency and duration charts but there will also be an option to be able to compare charts of the night shift data with combined charts of that of the day and evening shifts. The user should have a number of options when selecting the data to be displayed including date range, shift name, production line name and device. The default date range should be starting one week from current date and ending on current date. The default line will be the first in an alphabetical list. The default shift option will be ‘All shifts’ and the default device option will be ‘All devices’.

3.1.10 Messages component (R3)

A component is needed to display messages sent from a local Prima Power application on the machine PC. The component will give messages background colors to show the importance of the message for example if the machine is currently not running the background color will be red.

3.1.11 Timeline view (R3)

The main purpose of the timeline view is to show data over a period of time with zooming and panning capabilities. The data to be displayed on the timeline is alarms as items and machine state descriptions as background colors. A red background to represent failure, yellow for idle and green for running. Upon clicking an alarm, a modal view will be displayed to give detailed information of each alarm. Below the timeline will be a grid showing alarms currently displayed in the timeline and show some details of alarm information as well as a background color to represent relevant machine state description. Beneath the timeline should also be space to show an alarm chart which by default should be alarm frequencies but the user should also have the option to change the chart to display alarm duration. The user should have a number of options when selecting the data to be displayed including date range, shift name, line name and device. The default date range should be starting one week from current date and ending on current date. The default line will be the first in an alphabetical list. The default shift option will be ‘All shifts’ and the default device option will be ‘All devices’.

3.1.12 Add note to timeline capability (R4)

By double clicking on the timeline a modal popup dialog should be displayed with user inputs which gives the user the option of recording a comment (a note). All notes recorded will be displayed as note icons on the timeline, when the user clicks on a note icon a modal will be shown displayed the note information. When a new note is recorded, all authorized users of the same customer will receive a notification.

3.1.13 Maintenance view (R4)

The maintenance view will show a history log of the maintenance schedule. A grid component will be used to show each log and on clicking the grid row a more detailed table of each maintenance task will open up along with any remarks the operator has recorded. Each log will have a background color to show if all tasks are complete or not, green for complete and red for incomplete. The grid shall be filterable by date, status (complete or incomplete), line, operator name and remark. The view should also give the user the option to log the maintenance task, a grid of upcoming tasks will be shown and upon clicking a row, a modal will be shown which lets the user log the tasks. If a remark is logged on consecutive weeks for the same task and same line then a notification should be created to bring this to attention.

3.1.14 Scheduler upcoming events component (R5)

An easily accessible modal or dropdown to display upcoming calendar items without navigating away from current view. The user should be able to filter which events are listed. When the user clicks on an event item, the main view will navigate to the calendar view.

3.1.15 Performance view (R6)

A view to show charts displaying machine utilization percentage for each day. There is to be a chart for each assembly line. The default view will show each device’s utilization on the chart but also an option to display each shift’s utilization. The charts will be filterable by date with the default start date to be one week from current date and the default end date to be current date.

3.2 Initial mockups

3.2.1 Navigation

Figure 5 shows main navigation is a left aligned text list with icons. The list collapses to only icons upon clicking a hamburger menu icon in the top navigation bar as well as when the screen width is smaller than a certain width. The Prima Power logo is included at the top left of the navigation and when the left menu is collapsed the logo's text is hidden.

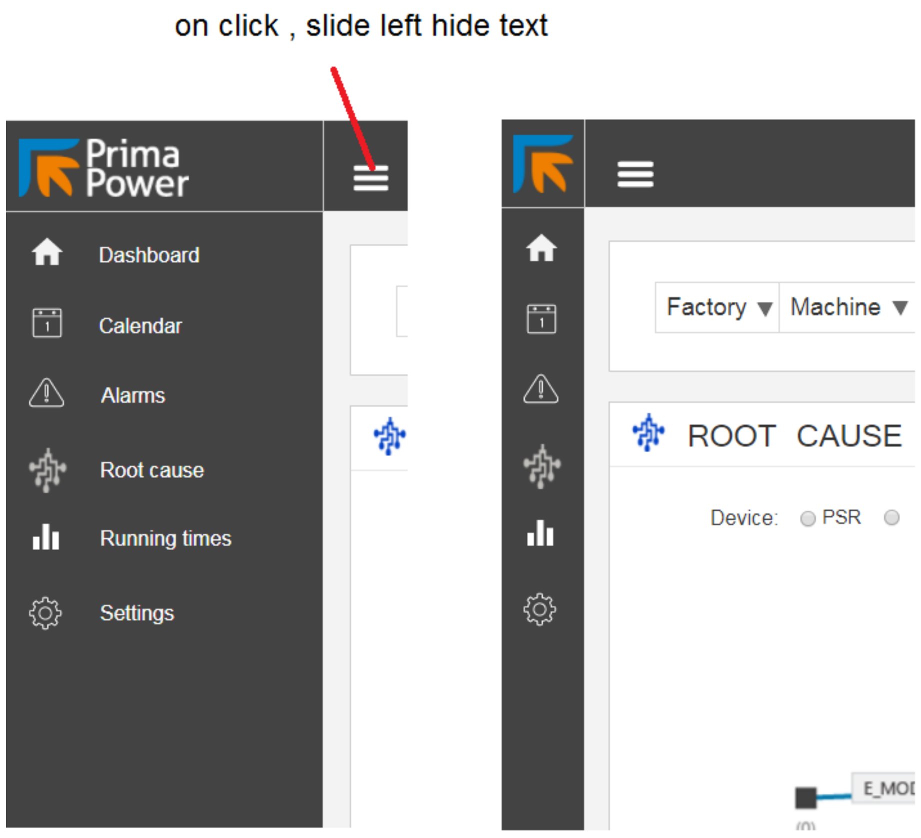


Figure 5. Mock design of left navigation

Figure 6 shows a top navigation bar with icons in the top right of the screen. This includes a user icon to show a dropdown menu of user options. The calendar icon will open a panel from the right with upcoming events as shown in Figure 7.

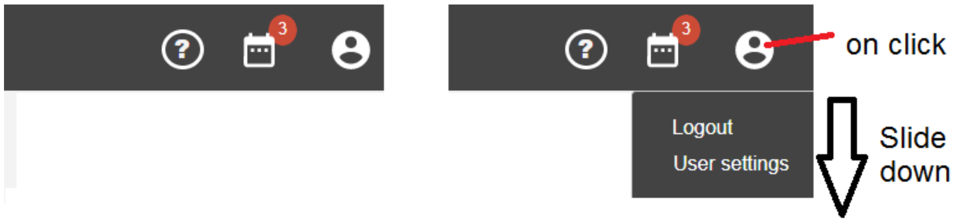


Figure 6. Mock design of right navigation

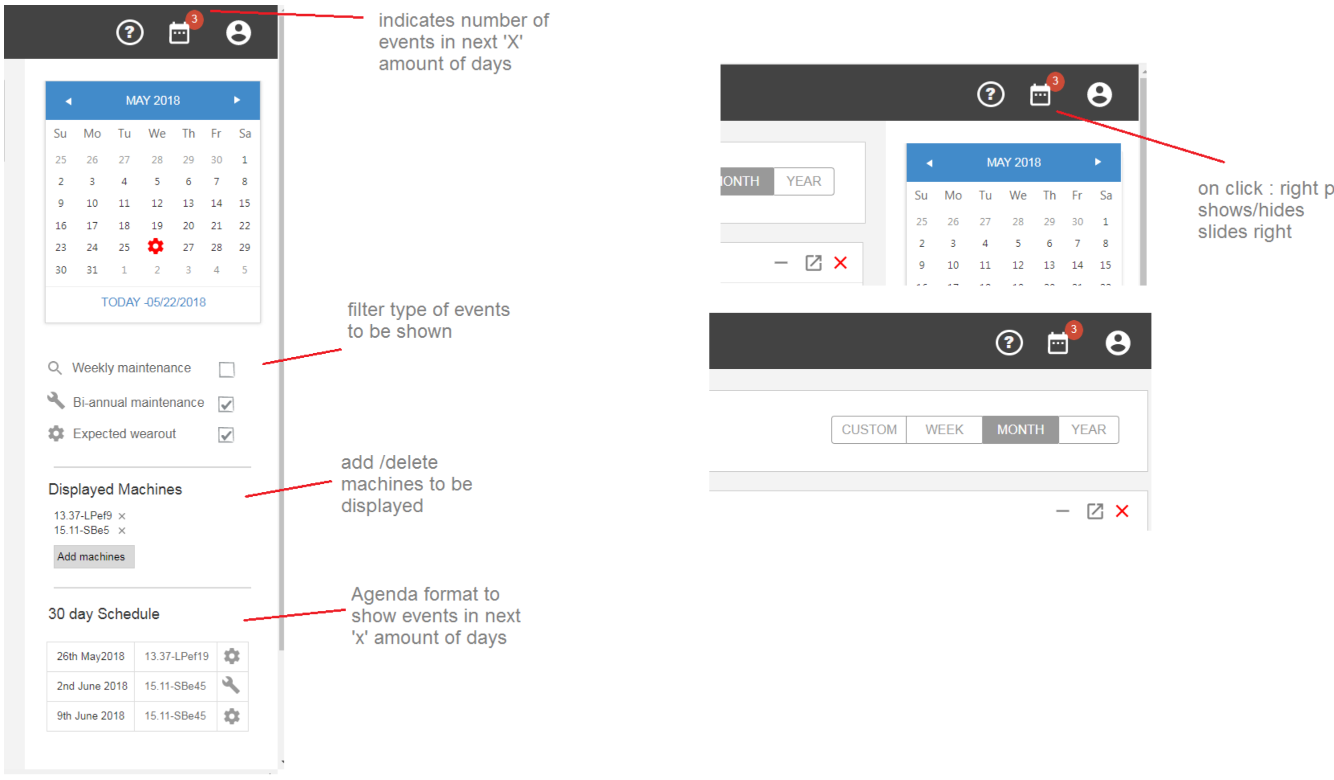
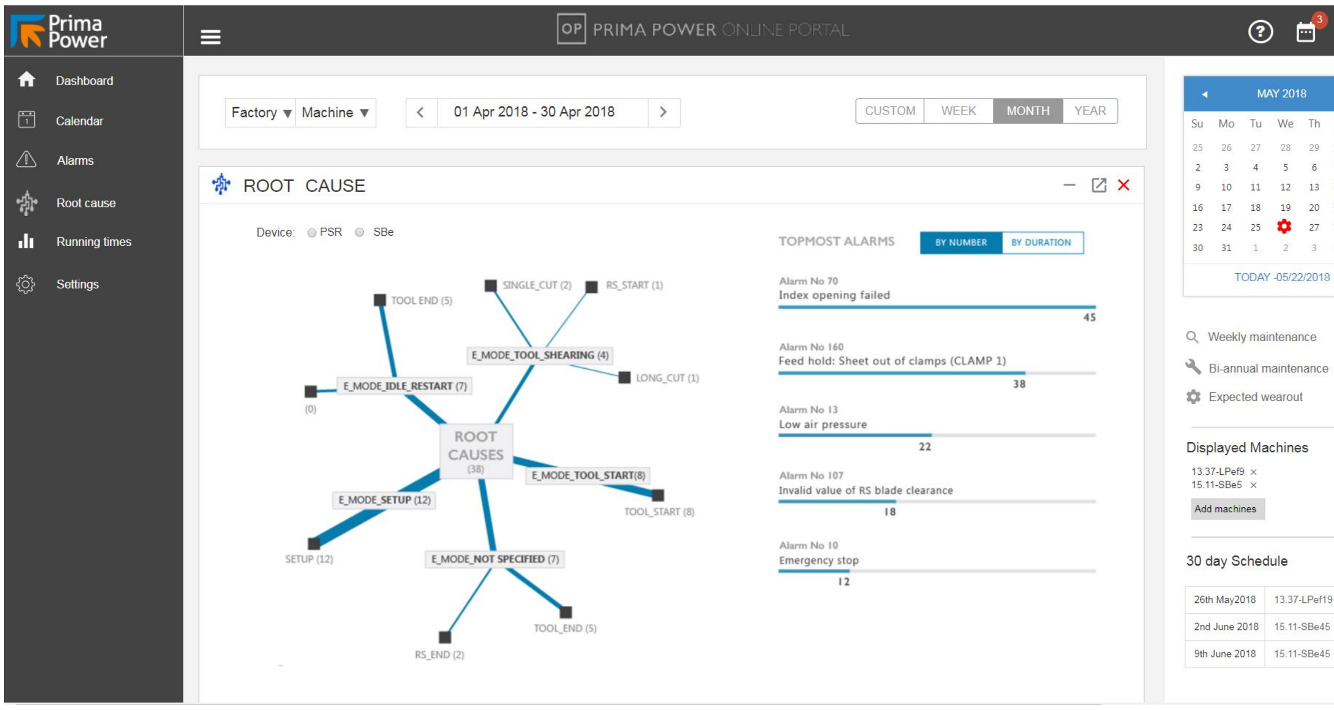

3
indicates number of events in next 'X' amount of days
MAY 2018
Su Mo Tu We Th Fr Sa
TODAY 05/22/2018
filter type of events to be shown
Weekly maintenance
Bi-annual maintenance
Expected wearout
Displayed Machines
13.37-LPef9
15.11-SBe5
Add machines
add /delete machines to be displayed
30 day Schedule
26th May2018 13.37-LPef19
2nd June 2018 15.11-SBe45
9th June 2018 15.11-SBe45
Agenda format to show events in next 'x' amount of days
ONTH YEAR
on click : right p shows/hides slides right
CUSTOM WEEK MONTH YEAR

Figure 7. Mock design of left navigation

3.2.2 Content

Figure 8 shows the default view for large screens which will show both left and right sidebars. On a medium sized screen the right side bar will be hidden by default.

default view for large screens



Alarm No 70
Index opening failed
45

Alarm No 160
Feed hold: Sheet out of clamps (CLAMP 1)
38

Alarm No 13
Low air pressure
22

Alarm No 107
Invalid value of RS blade clearance
18

Alarm No 10
Emergency stop
12

26th May2018	13.37-LPef19
2nd June 2018	15.11-SBe45
9th June 2018	15.11-SBe45

Figure 8. Mockup design for default view of large screen.

3.2.3 Root cause

Figure 9 shows the initial idea for the root cause visualization. The triggers (errors) are split up into separate nodes representing stages and sub stages. The center node represents all triggers. Each node is connected to a parent node by a blue line with the thickness of the line representing the amount of triggers in the child node. Looking at the visualization the user can quickly see which stages have the most errors.

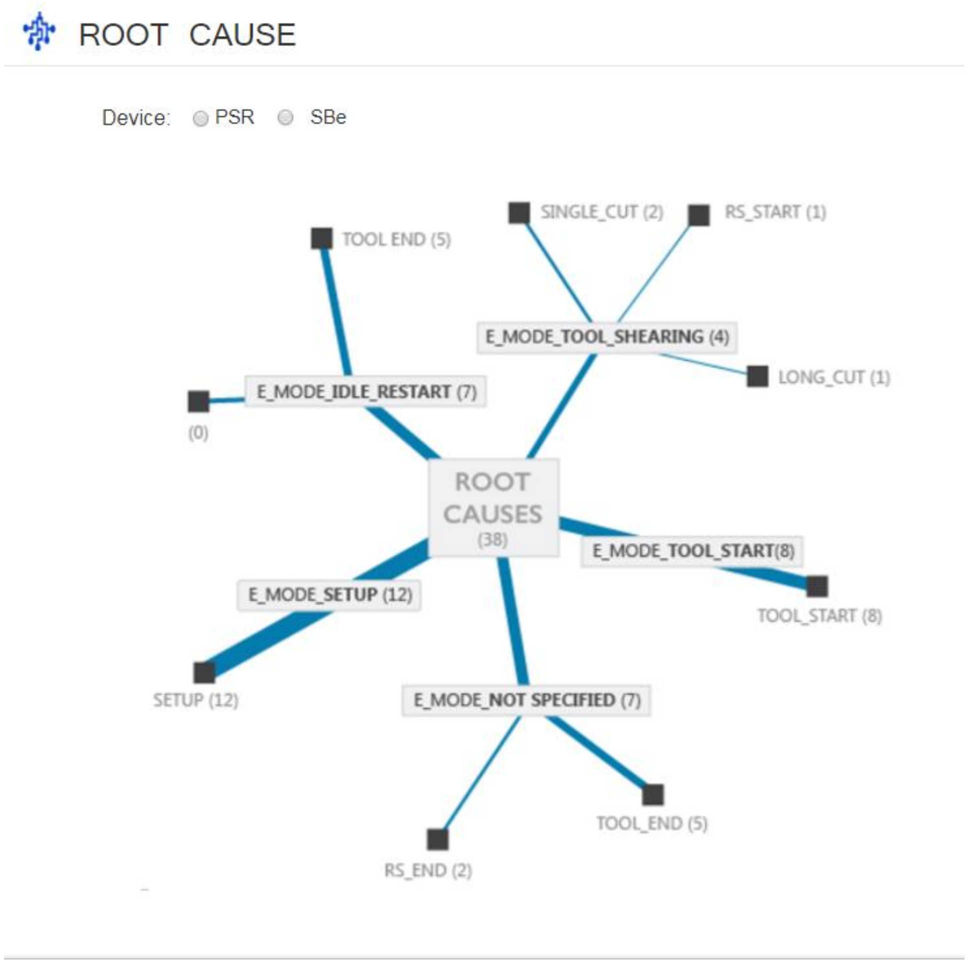
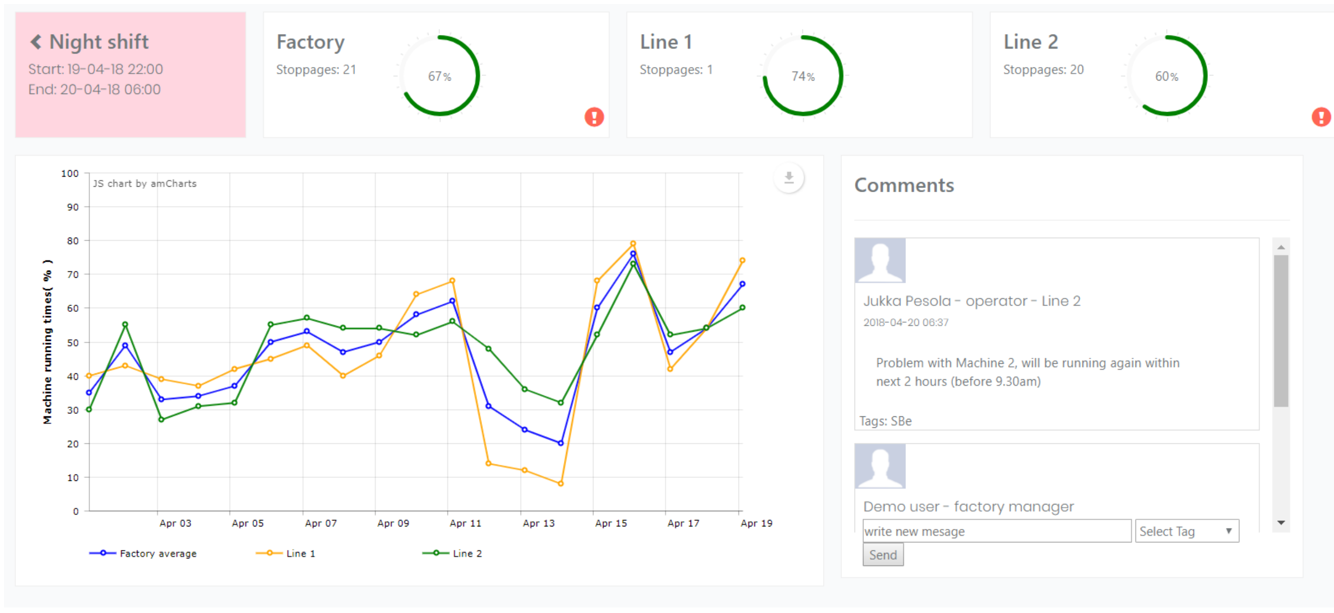


Figure 9. Mock up design for root cause visualization

3.2.4 Shift statistics

Figure 10 shows the mock up for the shift statistics page to show the machine utilization clearly for each line and the factory average. Each line/factory box contains a pie chart to show the shift utilization with a green bar if the utilization was above base line. Each box also shows how many stoppages occurred in the shift and if the shift ended in failure and explanation mark symbol is displayed. The shift selection box is displayed with a red background if there were problems during the shift. The machine utilization history is displayed as a line chart below the pie charts so the user can compare shift to previous weeks. A message box is also displayed so the user can see if any messages were wrote regarding the relevant shift.


Night shift
Start: 19-04-18 22:00
End: 20-04-18 06:00

Factory
Stoppages: 21
67%

Line 1
Stoppages: 1
74%

Line 2
Stoppages: 20
60%

Comments

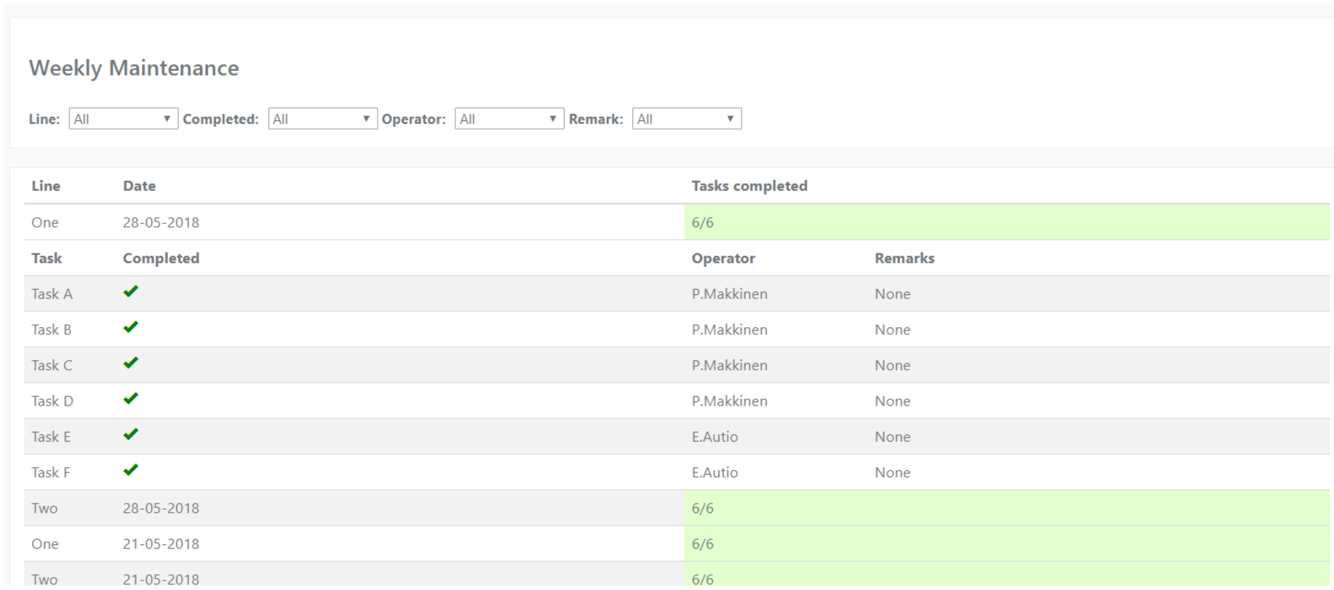
Jukka Pesola - operator - Line 2
2018-04-20 06:37
Problem with Machine 2, will be running again within next 2 hours (before 9.30am)
Tags: SBe

Demo user - factory manager
write new mesage
Select Tag
Send

Figure 10. Mock up design for shift statistics layout

3.2.5 Maintenance

Figure 11 shows the weekly maintenance view mockup showing an example of what is displayed when a user clicks on a log item.



Weekly Maintenance

Line: All ▾ Completed: All ▾ Operator: All ▾ Remark: All ▾

Line	Date	Tasks completed	
One	28-05-2018	6/6	
Task	**Completed**	**Operator**	**Remarks**
Task A	✓	P.Makkinen	None
Task B	✓	P.Makkinen	None
Task C	✓	P.Makkinen	None
Task D	✓	P.Makkinen	None
Task E	✓	E.Autio	None
Task F	✓	E.Autio	None
Two	28-05-2018	6/6	
One	21-05-2018	6/6	
Two	21-05-2018	6/6	

Figure 11. The mockup design for the maintenance view

4 Design

4.1 Library and component selection

4.1.1 Root cause view

After researching different open sourced data visualization libraries, originally a component from a library named vis.js was selected to work with. The resulting demos were successful in implementing the mockup design idea, however after some user testing, it was decided the visualization was not easy to understand and for presentational purposes it was not sufficient. A second solution was designed as shown below.

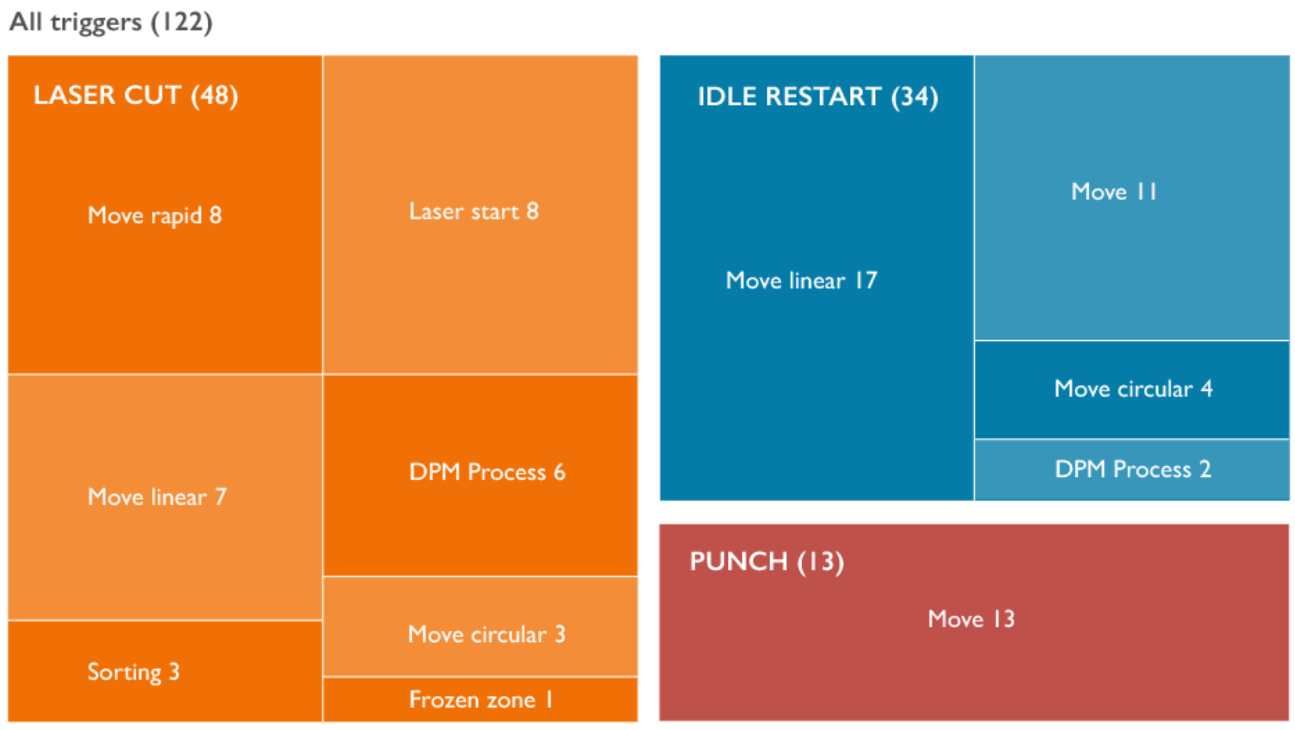


Figure 12. Treemap mockup design for rootcause visualization

The type of design in Figure 12 is known as a treemap of which there are many open sourced libraries available to create such a tool. After researching many libraries it became apparent that there would need to be some customization done to achieve the desired outcome. The most customizable data visualization library is D3.js which already has a feature to build treemaps. D3.js allows users to create something called a block which is a URL to a block of code that can be shared with other developers. Rather than developers starting from scratch on a visualization it is common to fork a block. Deenar Toraskar’s Block 336efdeb97f1c79d4a6b was selected to fork. /15/

4.1.2 Line and bar charts

The React re-charts component was to be used to create the charts. A custom tooltip will be created so that the tooltip colors will match the site color theme.

4.1.3 Timeline

Chap-links timeline was to be used for timeline component. The library is not available as a React component and so a wrapper is needed to be created to link the plain javascript with the React library. The library does not include a feature to display background colors and a custom implementation of the source code is needed.

4.1.4 Form inputs and calendar

The React version of the KendoUI framework will provide React components for all user inputs and grids as well as the calendar component.

4.1.5 Modal

React-bootstrap is to be used for the modal components. The modal components will be shown and hidden using the setState() method.

4.2 Architecture

4.2.1 State management

The React-redux library will be used to manage state between views. This makes the design and maintenance of the state management much more simple and efficient. It could be argued that in an application of this size that all state-management could be handled through call backs, however it is important that this application is scalable as it will grow in size and requirements in the future and therefor react-redux was considered the best choice. Each view will have four directories: containers, components, reducers and actions, an example of this can be seen in the root cause branch of the file structure diagram in Figure 13. The separate reducers and actions files will be imported into their respective global files. Only containers will contain logic handling global state. Component files will only contain logic for local state.

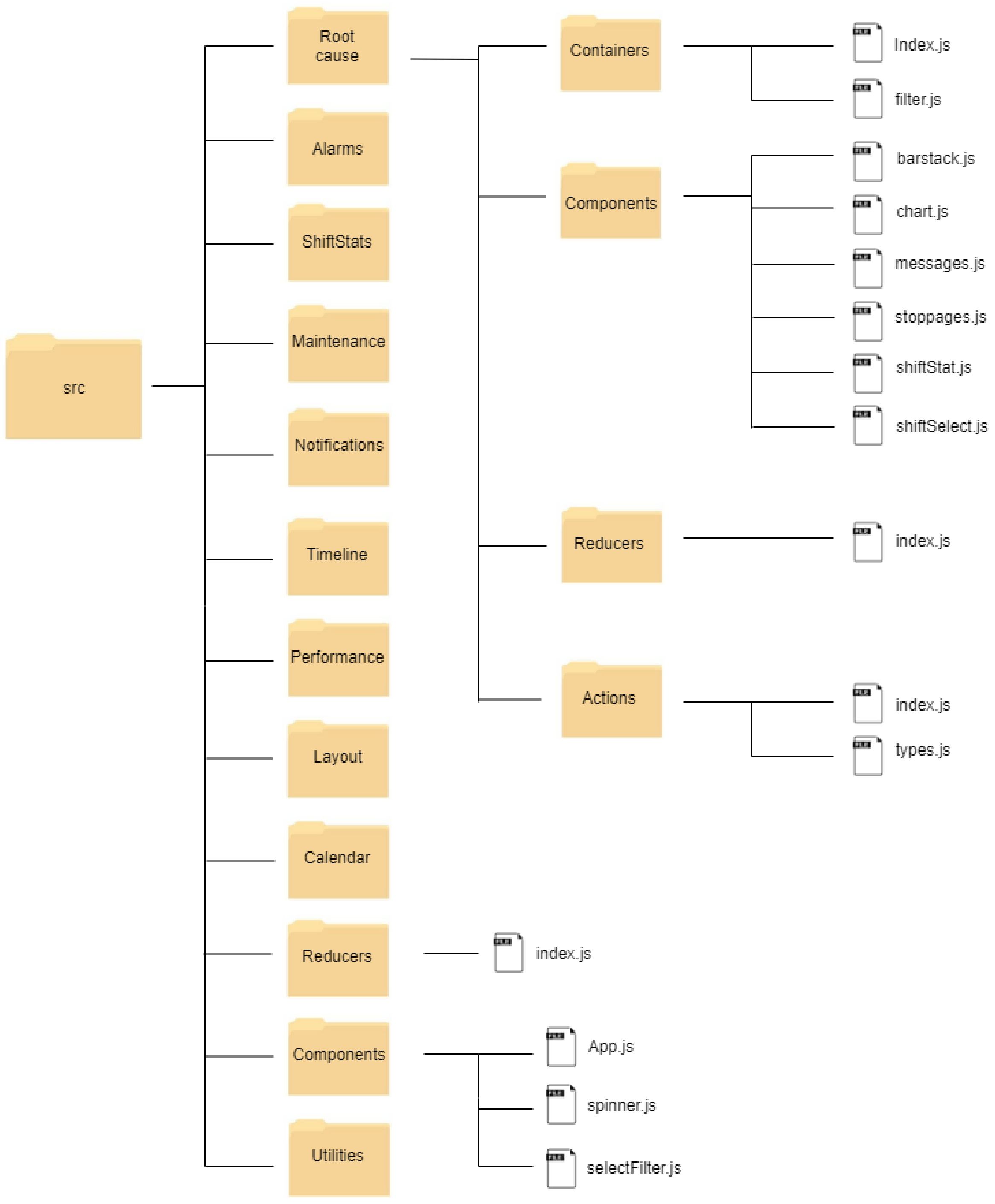


Figure 13. src container file structure

4.2.2 Asynchronous data request

When using react-redux, action creators return an action object which is limiting as it results in the action creator running synchronously. Redux-thunk is a middleware which gives the action the possibility to return a function which has the dispatch function of the store. This means actions can be conditionally dispatched for example they can wait for an asynchronous task to finish before dispatching which makes redux asynchronous scripting a lot easier to achieve and manage. Figure 14 displaying a diagram of how a middleware such as thunk is placed between the redux store and the reducers which gives possibilities such as animating a loading bar.

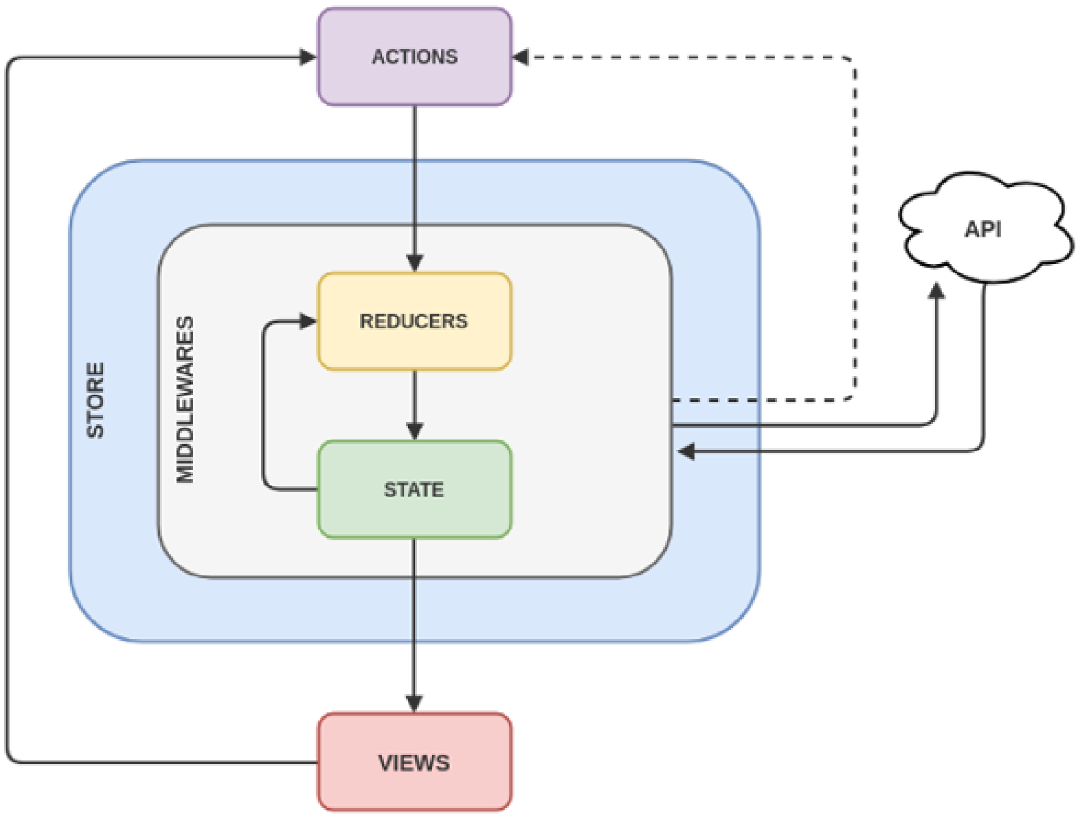


Figure 14. The react-redux design with middleware

4.2.3 Views

The architecture of each view will be divided into two containers connected to the redux store. The first container is used to set the filter options from user inputs. The second container will contain all visualizations and other content as components. The diagrams shown from Figure 15 through to 20 show how the React components of each view are organized. The diagram design is taken from developer Ashley Kitson with the core principles as follows: /16/

- Dashed two headed arrow indicates that the linked components share their stores. A change initiated in any component is reflected wherever that store is surfaced.
- Parent to child component connected with line and the parent is denoted with an arrowhead.
- A callback is represented by a dashed arrow pointing towards the parent component with a label showing the child component method name. The parent method is added as a label on the parent/ child connecting arrowhead line.
- A user interaction can be shown by using a standard UML inheritance arrow with an ‘instance’ stereotype.

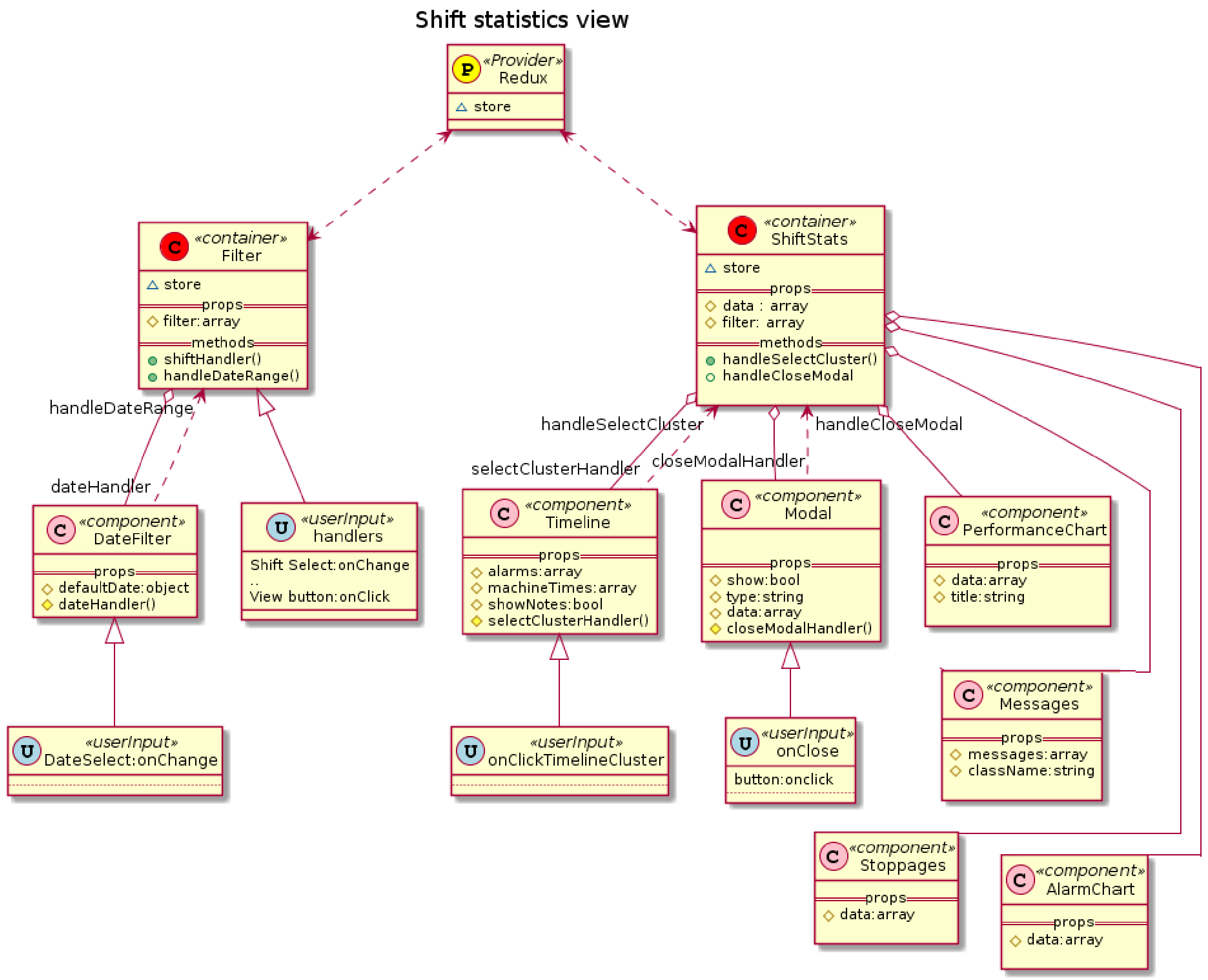


Figure 15. UML diagram to show component structure for the shift statistics view.

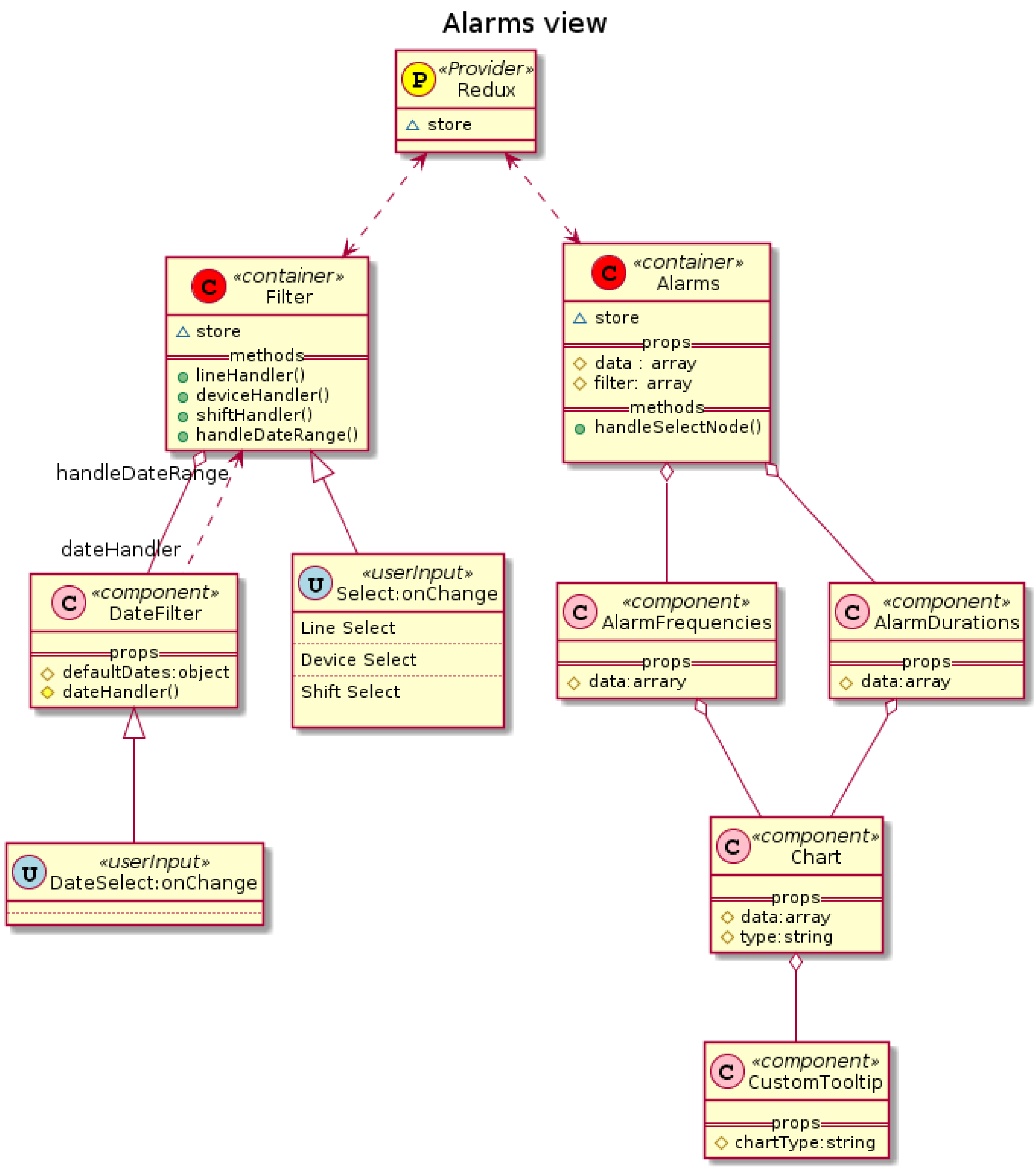


Figure 16. UML diagram to show component structure for the alarms view.

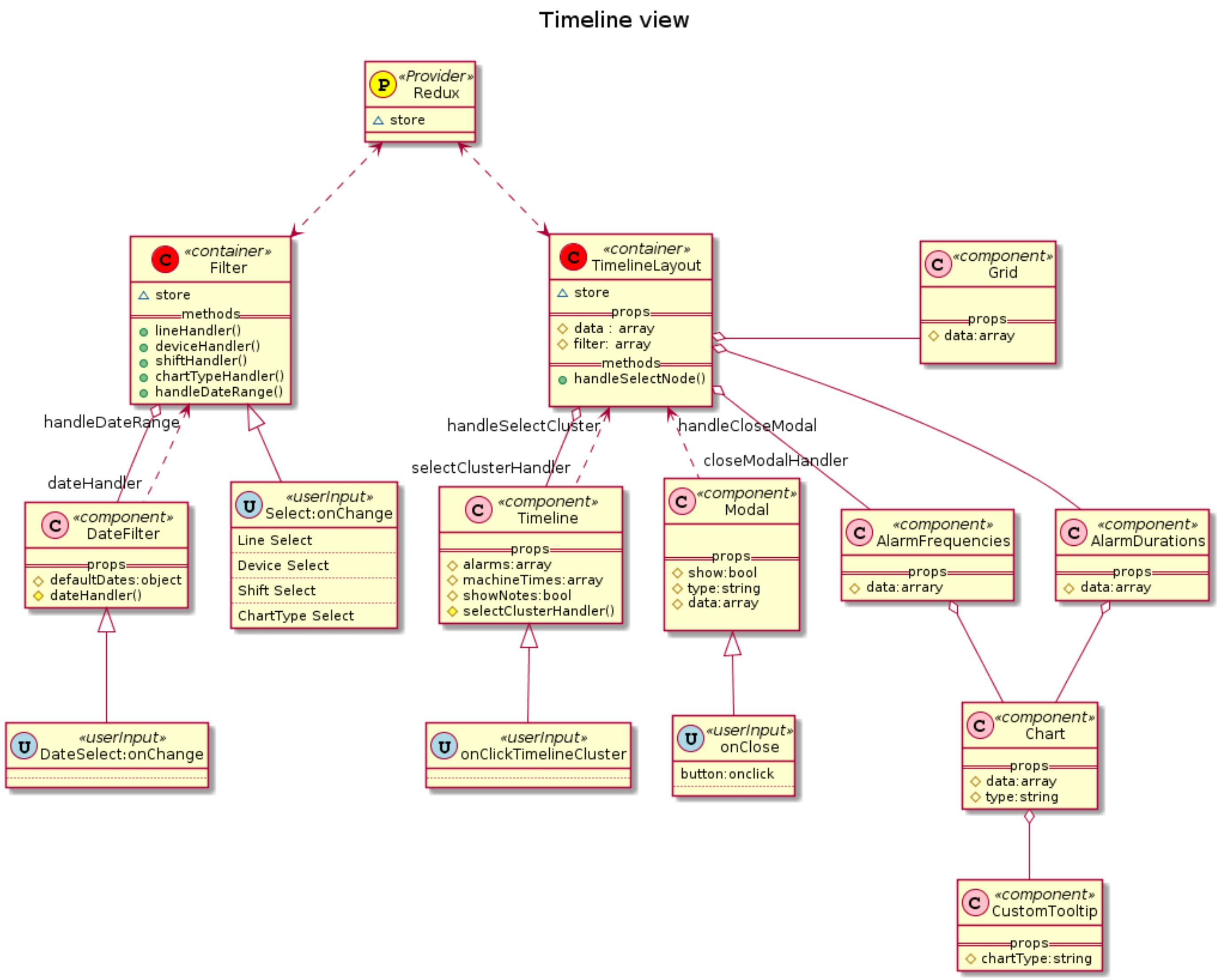


Figure 17. UML diagram to show component structure for the timeline view.

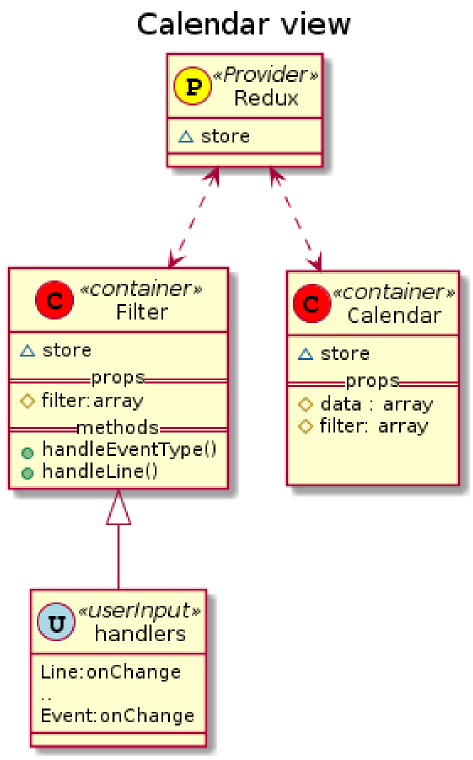


Figure 18. UML diagram to show component structure for the calendar view.

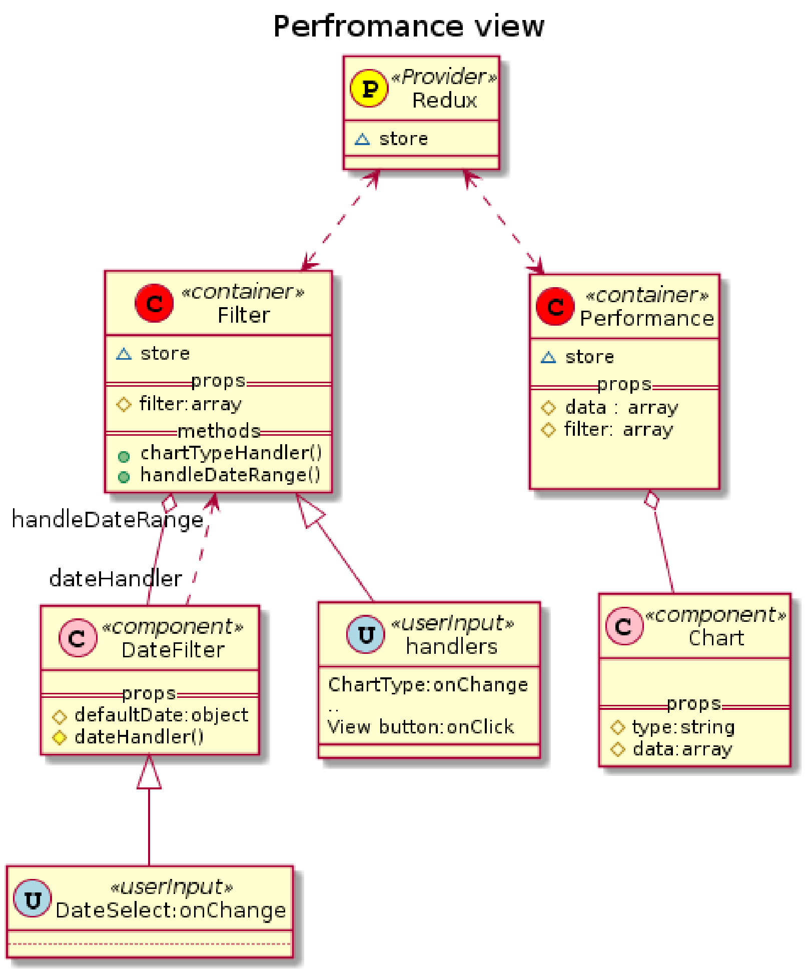


Figure 19. UML diagram to show component structure for the performance view.

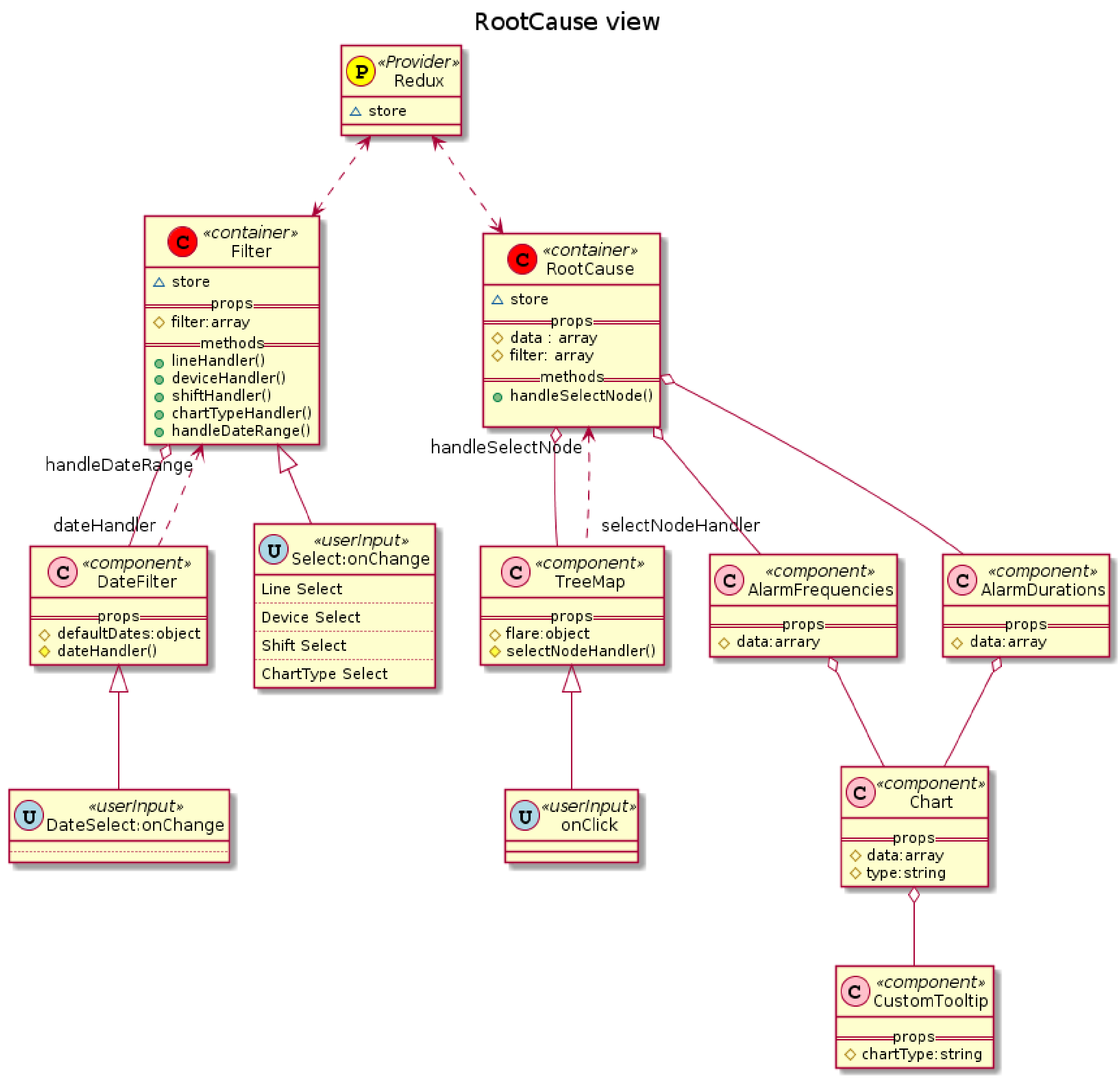


Figure 20. UML diagram to show component structure for the root cause view.

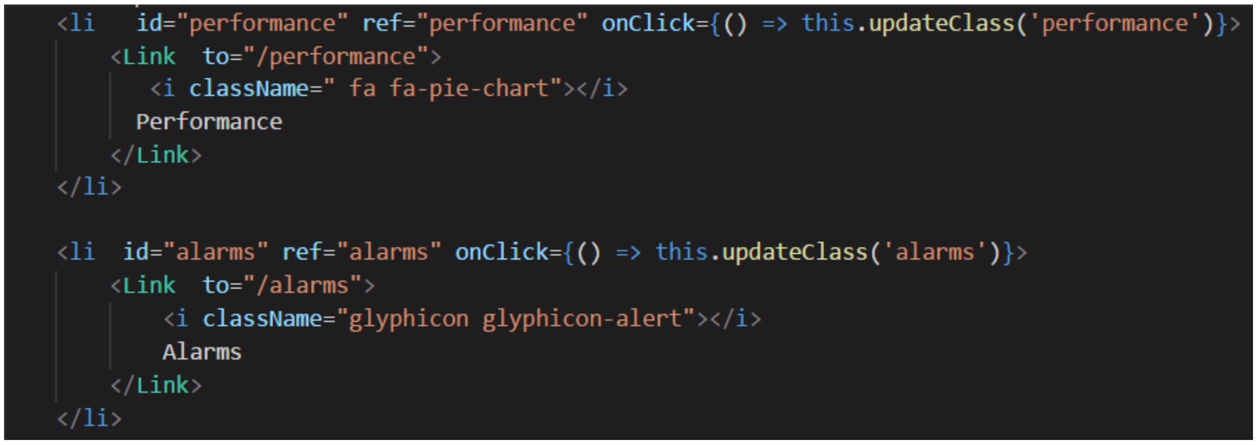
5 Implementation

5.1 Setup

Before development on the application could be started, first the development environment and build tools had to be installed. Node.js and npm were installed before the create-react-app command was used to set up the build tools and the basic directory structure of the application. Next using npm all dependencies were installed and any none npm dependencies were manually added to the public directory and referenced in index.html. The npm start command starts the application using nodejs in a local environment.

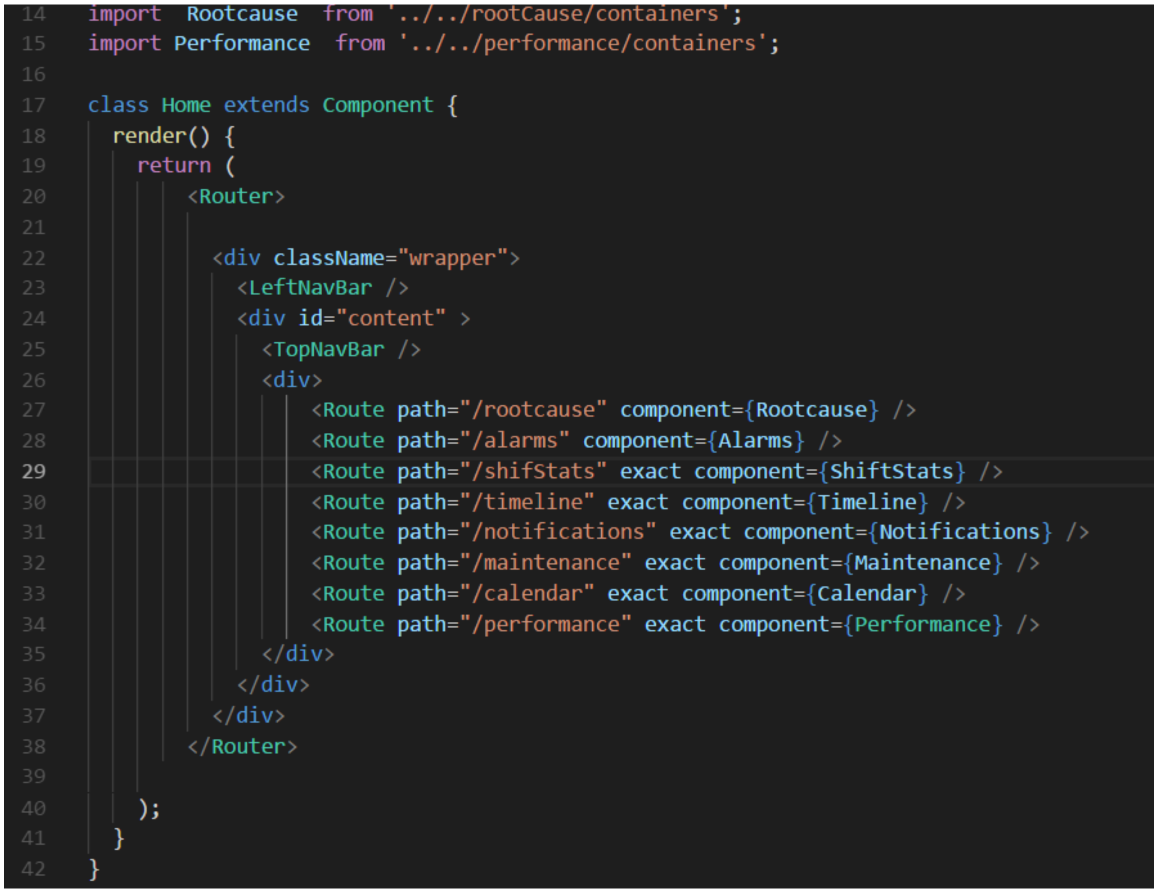
5.2 Layout / Navigation

The layout of the application is split into 3 sections: the left navigation sidebar, the top navigation bar and an area to display the selected view with the shiftStatistics view as the default view. The views are selected and displayed using the Link and BrowserRouter components from the react-router-dom package. The Link components are used in the navigation components whilst BrowserRouter is used to display the routes as shown in Figure 21 and Figure 22 respectively.



```
<li   id="performance" ref="performance" onClick={() => this.updateClass('performance')}>
    <Link  to="/performance">
      <i className=" fa fa-pie-chart"></i>
     Performance
    </Link>
</li>

<li  id="alarms" ref="alarms" onClick={() => this.updateClass('alarms')}>
    <Link  to="/alarms">
       <i className="glyphicon glyphicon-alert"></i>
       Alarms
    </Link>
</li>
```

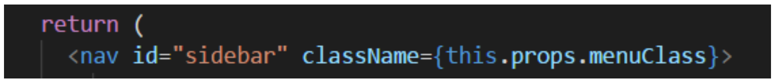
Figure 21. Code snippet to show use of react-router-dom Link component



```
import  Rootcause  from '../../rootCause/containers';
import Performance  from '../../performance/containers';

class Home extends Component {
  render() {
    return (
        <Router>

          <div className="wrapper">
            <LeftNavBar />
            <div id="content" >
              <TopNavBar />
              <div>
                  <Route path="/rootcause" component={Rootcause} />
                  <Route path="/alarms" component={Alarms} />
                  <Route path="/shifStats" exact component={ShiftStats} />
                  <Route path="/timeline" exact component={Timeline} />
                  <Route path="/notifications" exact component={Notifications} />
                  <Route path="/maintenance" exact component={Maintenance} />
                  <Route path="/calendar" exact component={Calendar} />
                  <Route path="/performance" exact component={Performance} />
              </div>
            </div>
          </div>
        </Router>

    );
  }
}
```

Figure 22. Code snippet to show use of BrowserRouter component

The left sidebar changes to a smaller size when either the user clicks on the hamburger menu icon or the screen width is detected to be below 768px. When the user clicks the hamburger icon the state is updated which in turn changes the CSS class of the sidebar using the JSX className attribute as seen in Figure 23. CSS media queries are used to change the style of the sidebar for different screen widths.



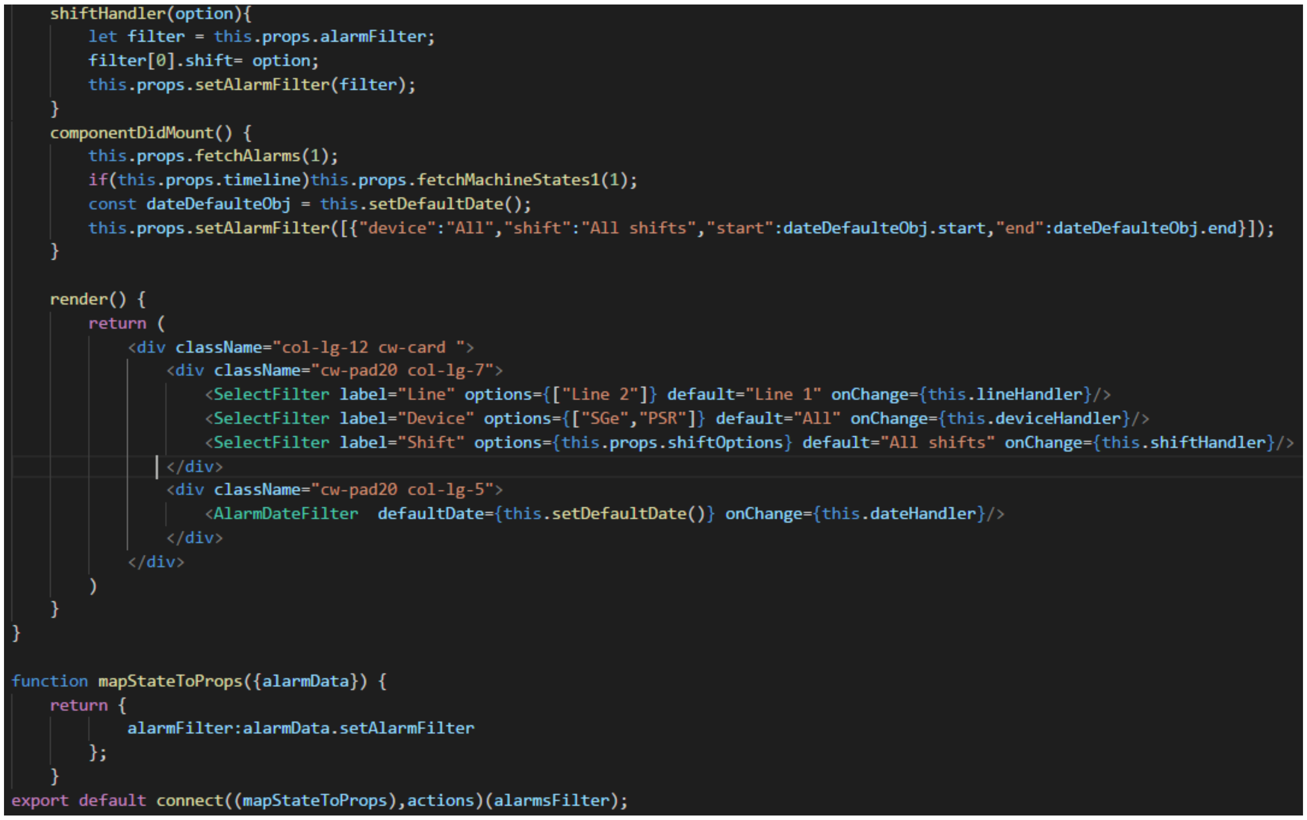
```
return (
  <nav id="sidebar" className={this.props.menuClass}>
```

Figure 23. JSX dynamic className example

5.3 Example of the general architecture of each view

5.3.1 Data fetching and data filtering

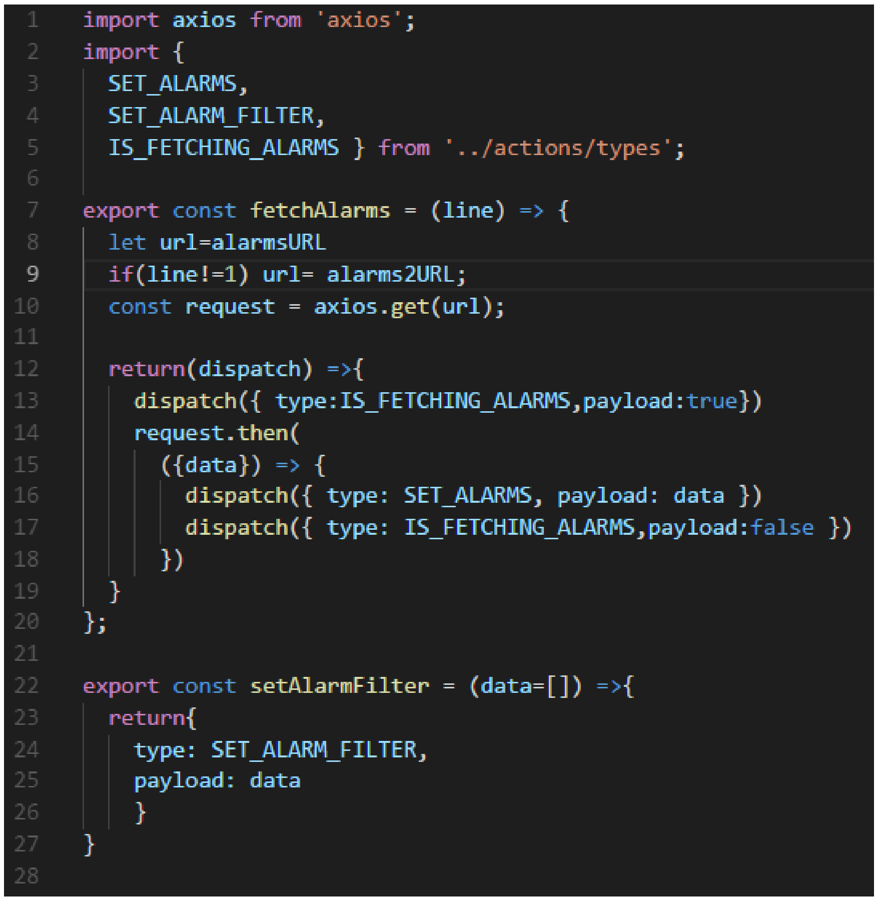
React-redux and the middleware thunk are used to manage the application state and enable the container files to share information. Each view contains two containers: a filter container and a content container. When the filter container mounts an action creator is called to fetch the relevant data and the default filter object is set. The content container uses conditional rendering to display an animation of a loading spinner until data is received from the reducer at which point it will read the filter object to decide which content to render. An example of this is the alarm view module of which the React code can be viewed in figures 24 through to 29. Figure 24 shows the alarmFilter container which uses the React componentDidMount method to call action creators to fetch the data and set the default filter properties. The setAlarmFilter object is the single element of an array, the reason being that redux reducers do not always recognize changes in an object but always update arrays when the es6 spread operator is used in the reducer. The shiftHandler method seen in Figure 24 is an example of how a user input calls an action creator. The redux mapStateToProps method returns the filter in its latest state which can be accessed using this.props.



```
    shiftHandler(option){
        let filter = this.props.alarmFilter;
        filter[0].shift= option;
        this.props.setAlarmFilter(filter);
    }
    componentDidMount() {
        this.props.fetchAlarms(1);
        if(this.props.timeline)this.props.fetchMachineStates1(1);
        const dateDefaulteObj = this.setDefaultDate();
        this.props.setAlarmFilter([{"device":"All","shift":"All shifts","start":dateDefaulteObj.start,"end":dateDefaulteObj.end}]);
    }

    render() {
        return (
            <div className="col-lg-12 cw-card ">
                <div className="cw-pad20 col-lg-7">
                    <SelectFilter label="Line" options={["Line 2"]} default="Line 1" onChange={this.lineHandler}/>
                    <SelectFilter label="Device" options={["SGe","PSR"]} default="All" onChange={this.deviceHandler}/>
                    <SelectFilter label="Shift" options={this.props.shiftOptions} default="All shifts" onChange={this.shiftHandler}/>
                </div>
                <div className="cw-pad20 col-lg-5">
                    <AlarmDateFilter  defaultDate={this.setDefaultDate()} onChange={this.dateHandler}/>
                </div>
            </div>
        )
    }
}

function mapStateToProps({alarmData}) {
    return {
            alarmFilter:alarmData.setAlarmFilter
        };
    }
export default connect((mapStateToProps),actions)(alarmsFilter);
```

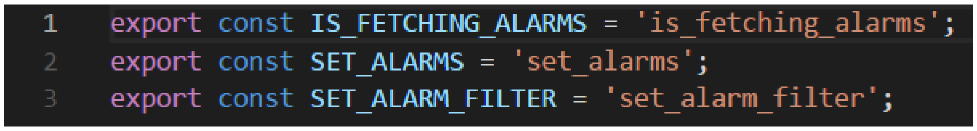
Figure 24. A code snippet of the alarmFilter container

The code as seen in Figure 25 shows the action creators used in the alarm module. The fetchAlarms action creator uses thunk middleware to control when the actions are dispatched which allows the use of the axios library for requesting data from the server and waiting for the response before dispatching the relevant action. This control over when an action is dispatched also gives the possibility to dispatch the IS_FECTHING_ALARMS action which when set to true will result in the UI displaying a loading animation. The setAlarmFilter does not use the thunk middleware and is instead returning the default redux action-creator object which has two properties the type and the payload. All types are imported from a separate file (Figure 26) as they are also used in the reducers file and so having the types coming from a single source ensures easier maintenance of the code and less start up bugs.



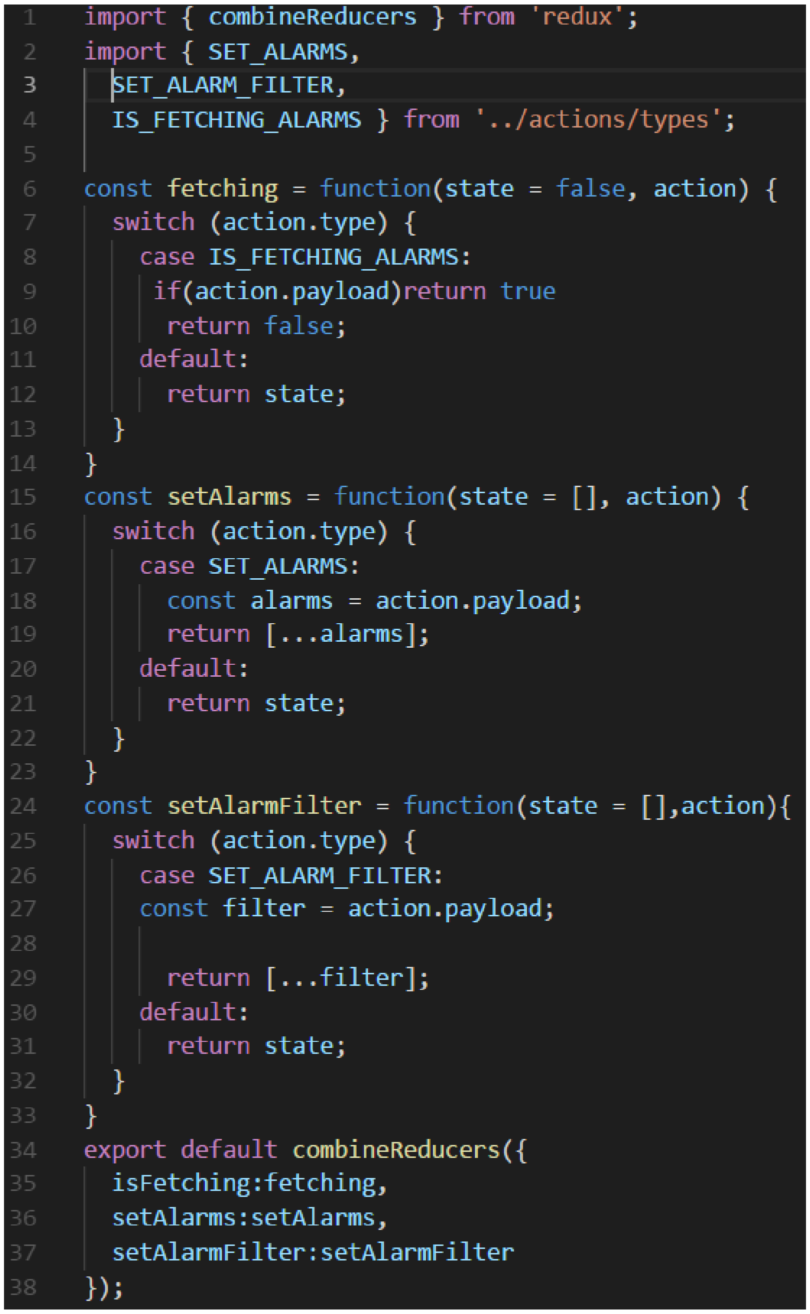
```
import axios from 'axios';
import {
  SET_ALARMS,
  SET_ALARM_FILTER,
  IS_FETCHING_ALARMS } from '../actions/types';

export const fetchAlarms = (line) => {
  let url=alarmsURL
  if(line!=1) url= alarms2URL;
  const request = axios.get(url);

  return(dispatch) =>{
    dispatch({ type:IS_FETCHING_ALARMS,payload:true})
    request.then(
      ({data}) => {
        dispatch({ type: SET_ALARMS, payload: data })
        dispatch({ type: IS_FETCHING_ALARMS,payload:false })
      })
  }
};

export const setAlarmFilter = (data=[]) =>{
  return{
    type: SET_ALARM_FILTER,
    payload: data
    }
}

```

Figure 25. The full code of the alarm module actions file.



```
export const IS_FETCHING_ALARMS = 'is_fetching_alarms';
export const SET_ALARMS = 'set_alarms';
export const SET_ALARM_FILTER = 'set_alarm_filter';
```

Figure 26. The alarms module types file



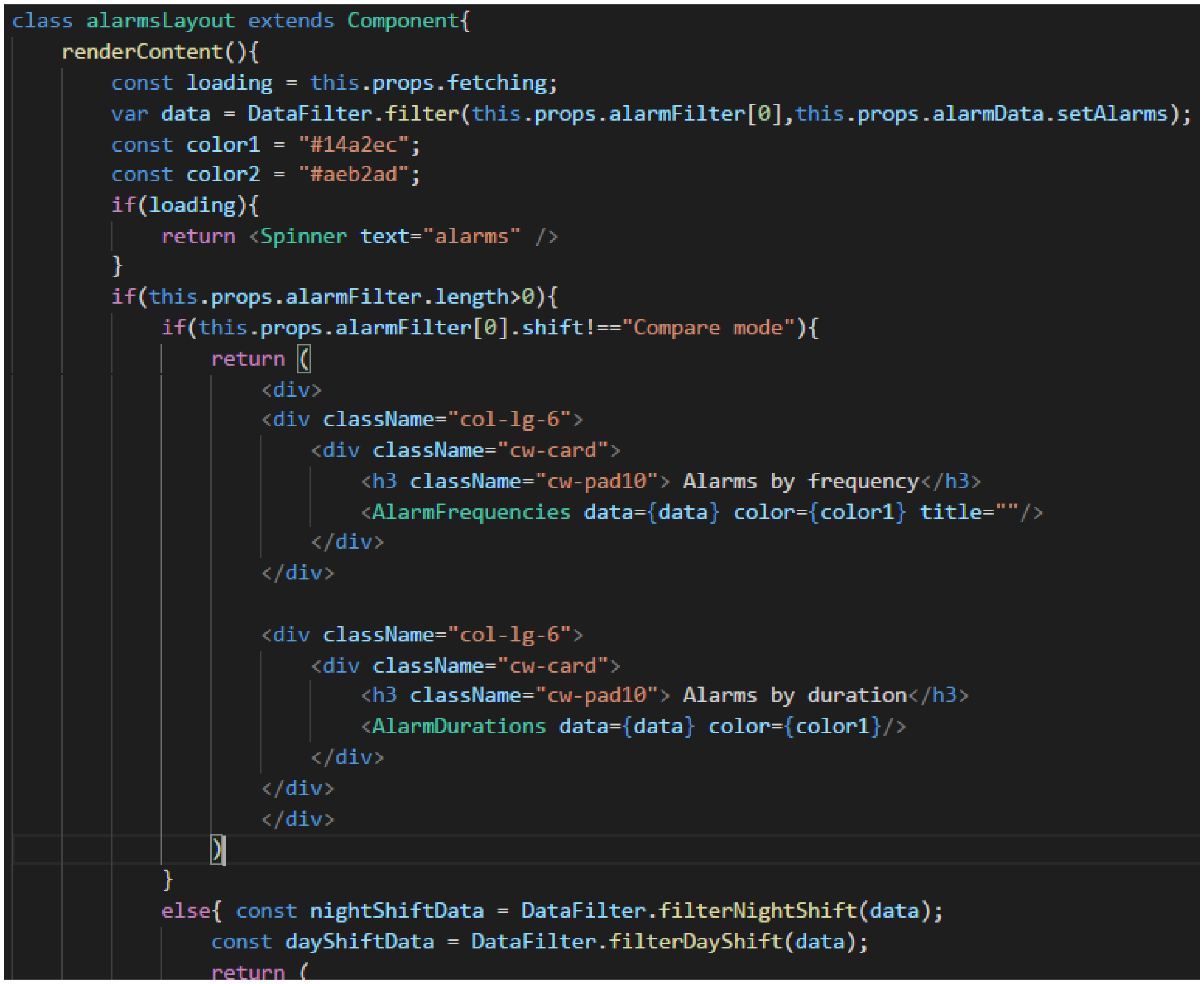
```javascript
import { combineReducers } from 'redux';
import { SET_ALARMS,
  SET_ALARM_FILTER,
  IS_FETCHING_ALARMS } from '../actions/types';

const fetching = function(state = false, action) {
  switch (action.type) {
    case IS_FETCHING_ALARMS:
     if(action.payload)return true
      return false;
    default:
      return state;
  }
}
const setAlarms = function(state = [], action) {
  switch (action.type) {
    case SET_ALARMS:
      const alarms = action.payload;
      return [...alarms];
    default:
      return state;
  }
}
const setAlarmFilter = function(state = [],action){
  switch (action.type) {
    case SET_ALARM_FILTER:
    const filter = action.payload;

      return [...filter];
    default:
      return state;
  }
}
export default combineReducers({
  isFetching:fetching,
  setAlarms:setAlarms,
  setAlarmFilter:setAlarmFilter
});
```

Figure 27. The alarms module reducer file

Figure 27 shows the alarm module reducer file. In each reducer function a switch statement is used to determine if the dispatched action is relevant and if so the new state is returned using the ES6 spread operator.

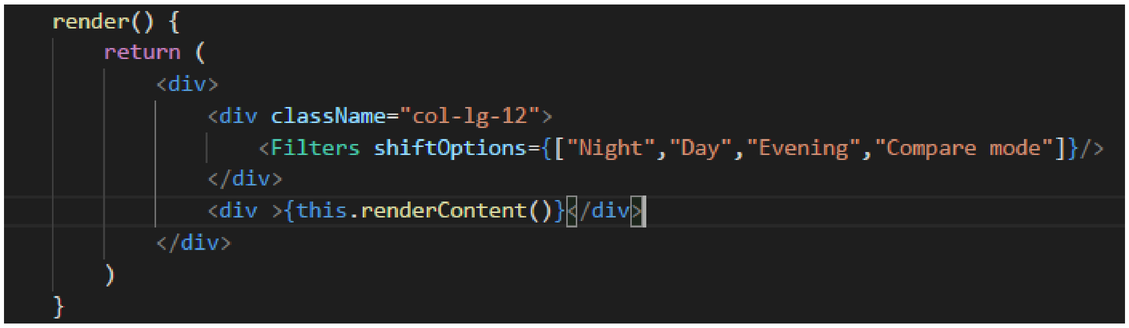
5.3.2 Conditional rendering

Using the mapStateToProps method the alarm content container receives the state of the setAlarms array, the alarmFilter array and the fetching Boolean which are set as props to be accessed by the JSX. Conditional rendering is used in the renderContent method to return the appropriate JSX as shown in Figure 28. The renderContent method is called from the component render method (see Figure 29) which updates every time state is updated. The finished alarm view can be viewed in Figure 30 and Figure 31.



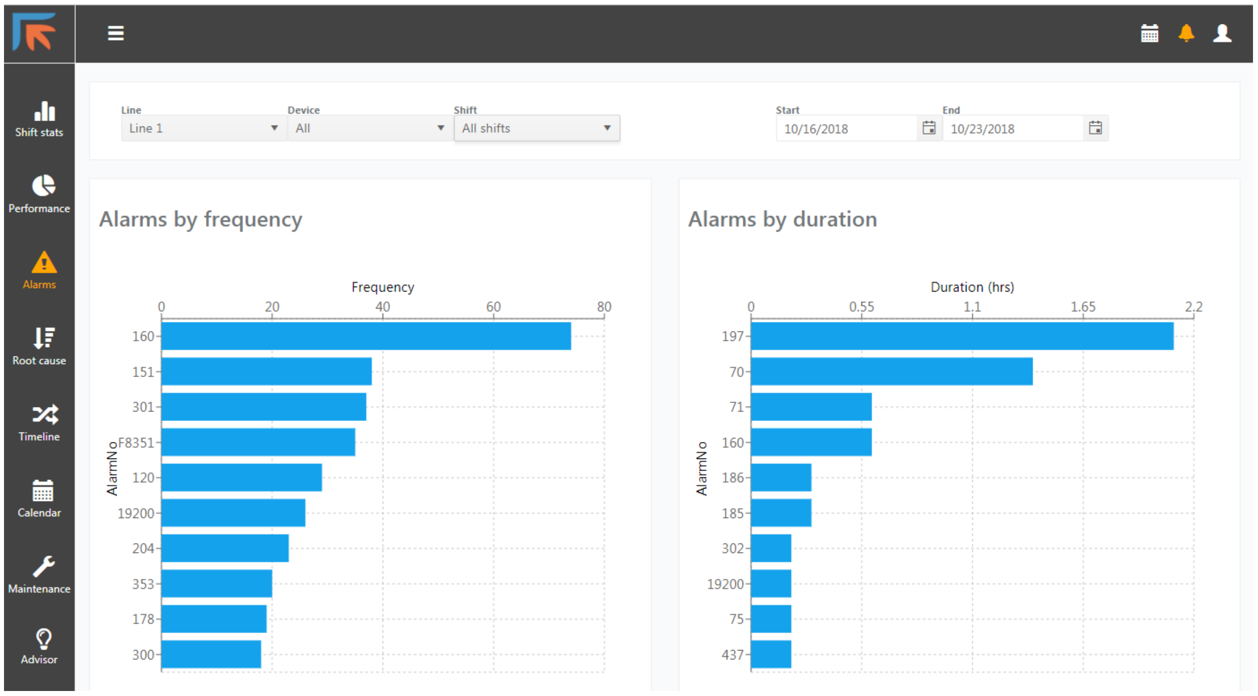
```
class alarmsLayout extends Component{
    renderContent(){
        const loading = this.props.fetching;
        var data = DataFilter.filter(this.props.alarmFilter[0],this.props.alarmData.setAlarms);
        const color1 = "#14a2ec";
        const color2 = "#aeb2ad";
        if(loading){
            return <Spinner text="alarms" />
        }
        if(this.props.alarmFilter.length>0){
            if(this.props.alarmFilter[0].shift!=="Compare mode"){
                return (
                    <div>
                    <div className="col-lg-6">
                        <div className="cw-card">
                            <h3 className="cw-pad10"> Alarms by frequency</h3>
                            <AlarmFrequencies data={data} color={color1} title=""/>
                        </div>
                    </div>

                    <div className="col-lg-6">
                        <div className="cw-card">
                            <h3 className="cw-pad10"> Alarms by duration</h3>
                            <AlarmDurations data={data} color={color1}/>
                        </div>
                    </div>
                    </div>
                )
            }
            else{ const nightShiftData = DataFilter.filterNightShift(data);
                const dayShiftData = DataFilter.filterDayShift(data);
                return (
```

Figure 28. Code snippet of the alarm module content container showing example of conditional rendering.



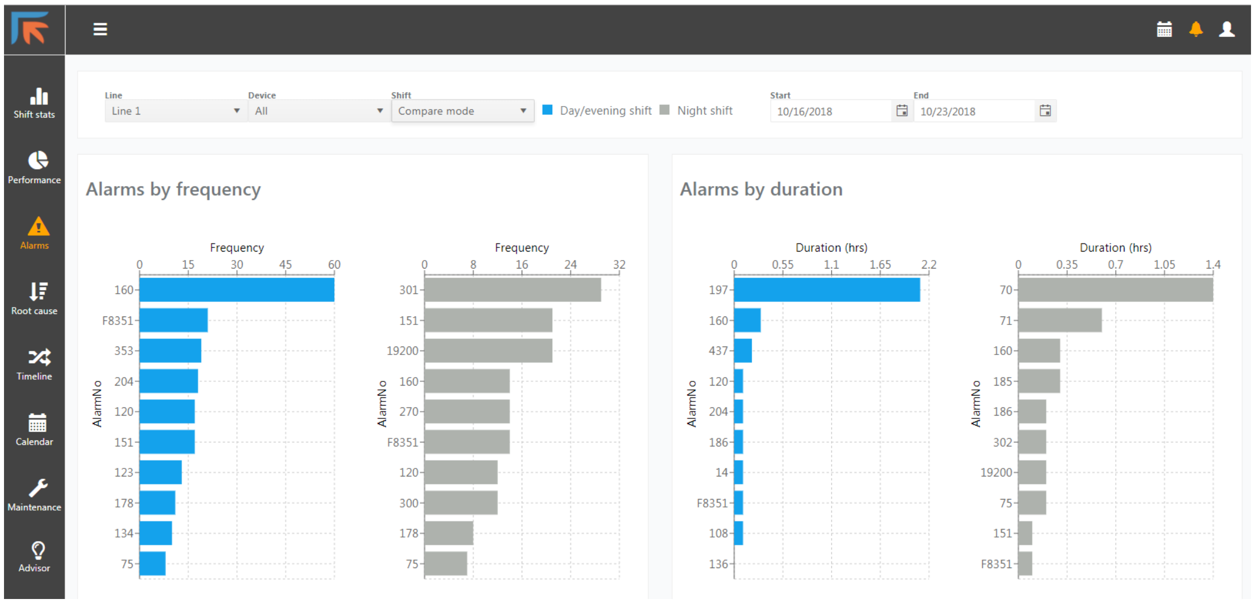
```
render() {
    return (
        <div>
            <div className="col-lg-12">
                <Filters shiftOptions={["Night","Day","Evening","Compare mode"]}/>
            </div>
            <div >{this.renderContent()}</div>
        </div>
    )
}
```

Figure 29. Code snippet from the alarm content container showing the render method.



Line: Line 1
Device: All
Shift: All shifts
Start: 10/16/2018
End: 10/23/2018

Figure 30. Alarm view with default settings.

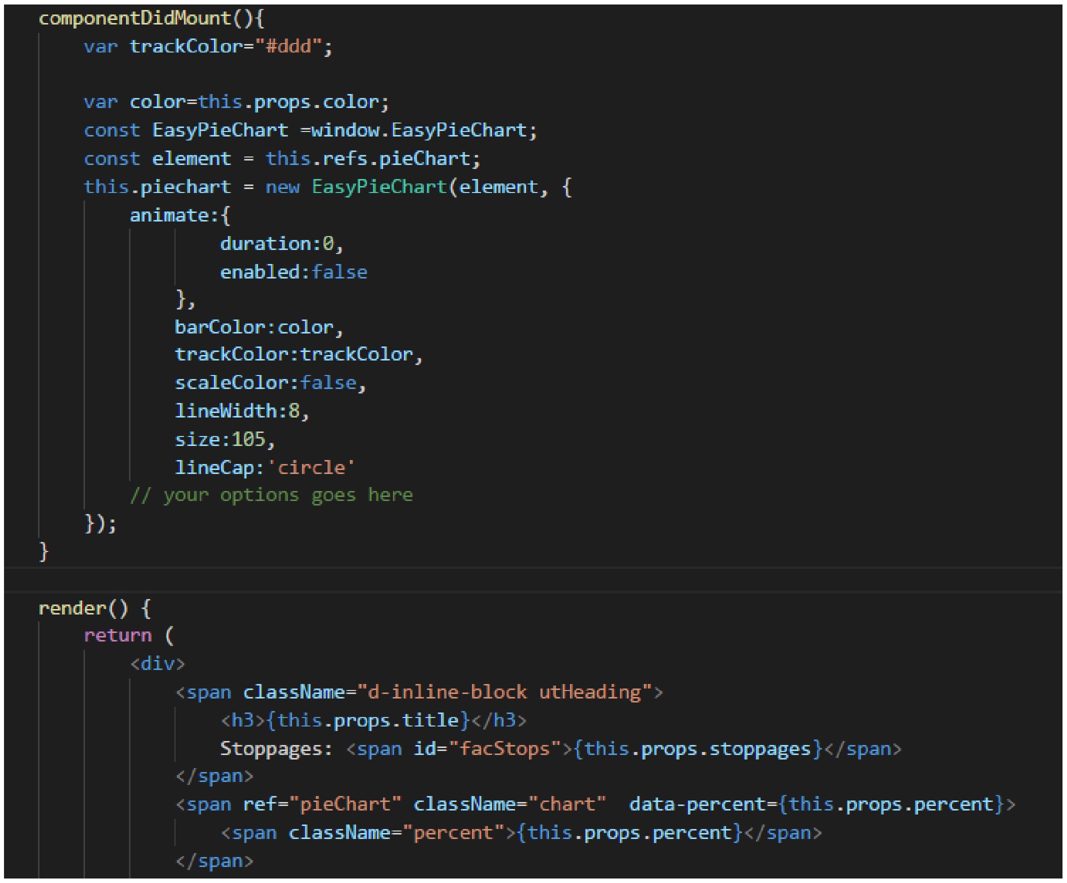


Line: Line 1
Device: All
Shift: Compare mode
Start: 10/16/2018
End: 10/23/2018

Figure 31. Alarm view with compare mode option selected.

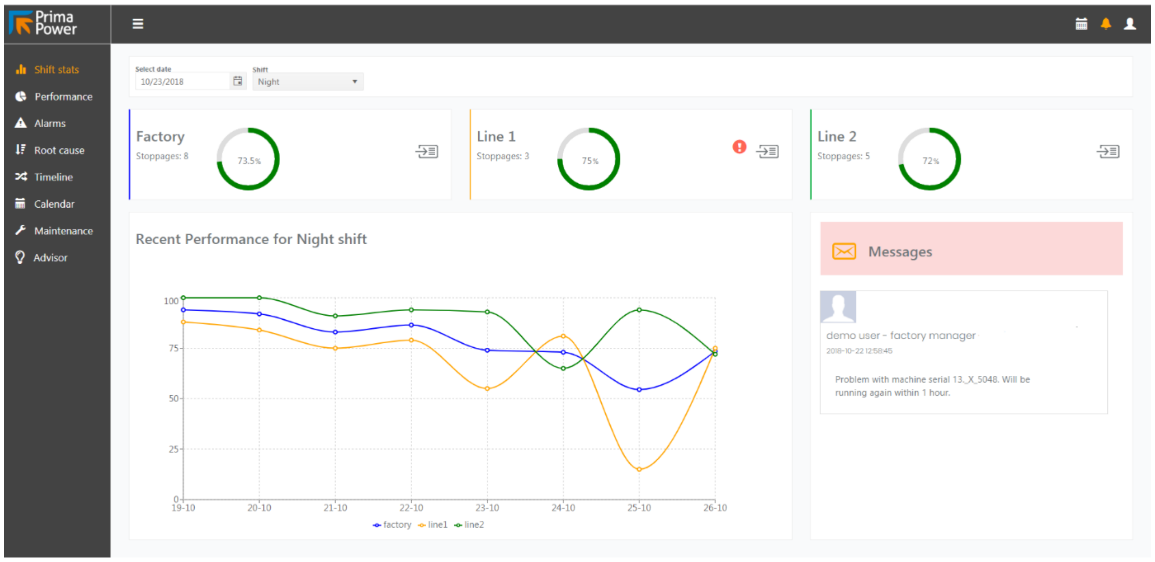
5.4 Using an external library with React

Some libraries were chosen to use in this application that did not have a React package which meant those libraries could not be imported as React components and instead had to be added to the global window object. An example of this was the jquery easy-pie-chart library of which the script was included in the index.html head. To use the library, the library's global variable was referenced using window.EasyPieChart and assigned to the const variable EasyPieChart. To pass in the relevant HTML element to the library, a ref is assigned to the element and accessed using this.refs. Figure 32 shows a code snippet of how the easy-pie-chart was implemented in the shiftStats component.



```
componentDidMount(){
    var trackColor="#ddd";

    var color=this.props.color;
    const EasyPieChart =window.EasyPieChart;
    const element = this.refs.pieChart;
    this.piechart = new EasyPieChart(element, {
        animate:{
            duration:0,
            enabled:false
        },
        barColor:color,
        trackColor:trackColor,
        scaleColor:false,
        lineWidth:8,
        size:105,
        lineCap:'circle'
        // your options goes here
    });
}

render() {
    return (
        <div>
            <span className="d-inline-block utHeading">
                <h3>{this.props.title}</h3>
                Stoppages: <span id="facStops">{this.props.stoppages}</span>
            </span>
            <span ref="pieChart" className="chart"  data-percent={this.props.percent}>
                <span className="percent">{this.props.percent}</span>
            </span>
```

Figure 32. A code snippet of the shiftStats component showing example of external library use.



Select date: 10/23/2018
Shift: Night

demo user - factory manager
2018-10-22 12:58:45

Problem with machine serial 13._X_5048. Will be running again within 1 hour.

Figure 33. The shift stats view including pie charts created with external library

5.5 Passing data from the child component to the parent component

To pass data from a child component to the parent container, callback functions were used. An example of this technique is in the root cause view. Figure 34 shows the root cause view which includes a tree map visualization and an alarm chart. When the user clicks on a treemap node, the alarm chart is updated to show relevant alarm data of selected node. Figure 35 shows a code snippet of the root cause content container (parent) passing a callback function to treemap component (child), this callback is then called from within the child component using props as seen in Figure 36. Calling the callback is done so from a function which is passed as a callback into the external rootCause script. When the original callback is called the parent container filters the relevant alarms and renders the alarm chart with the updated data as seen in Figure 37.

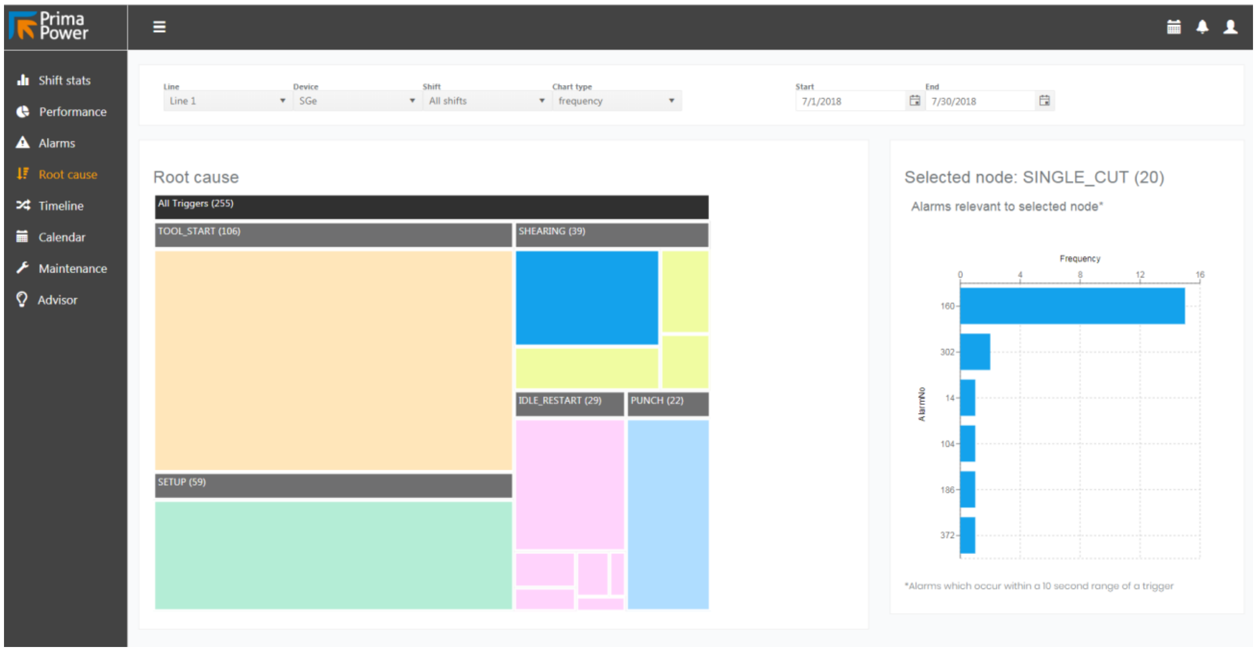
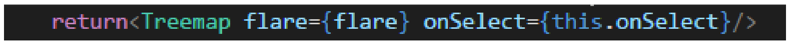
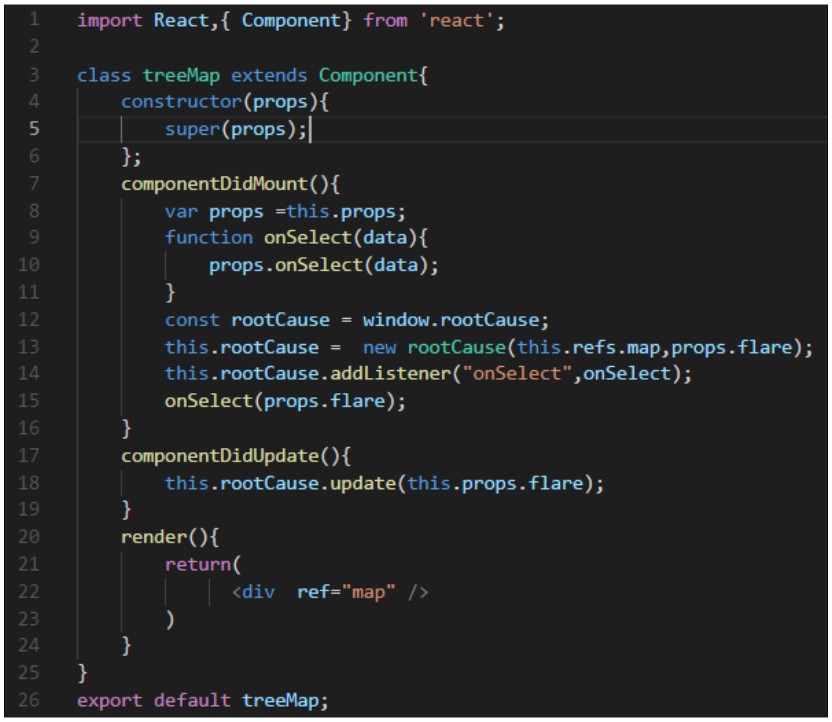


Figure 34. The root cause view



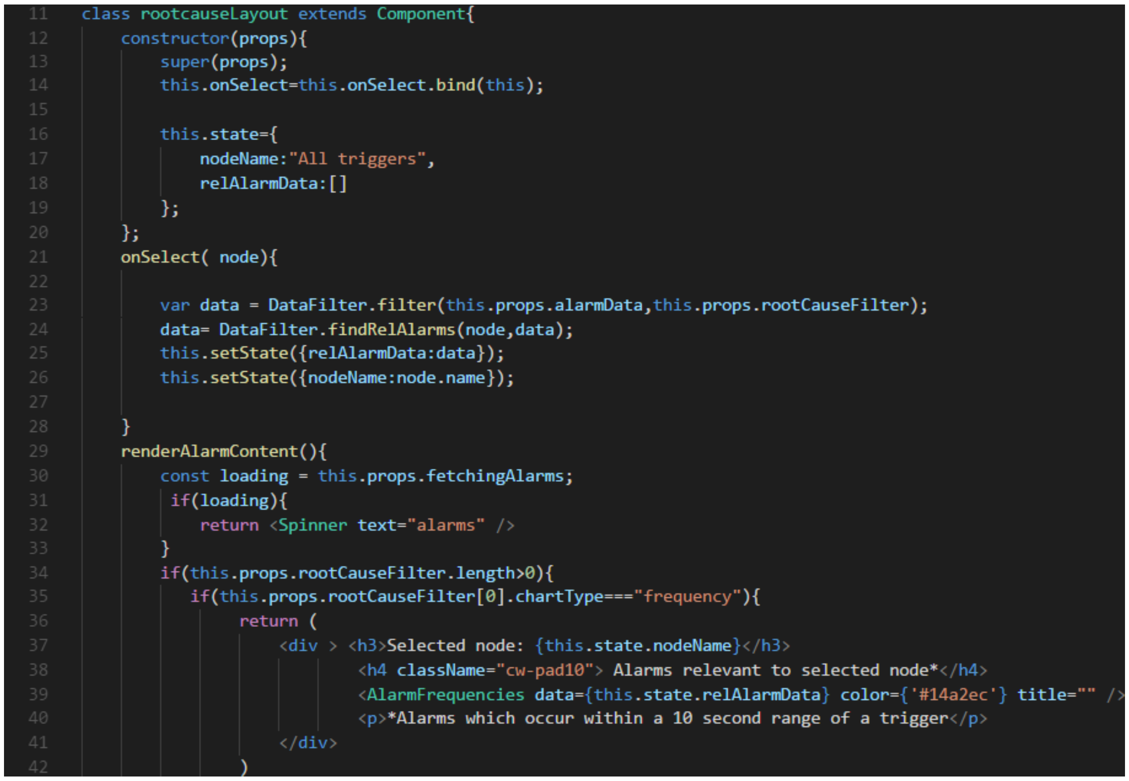
```
    return<Treemap flare={flare} onSelect={this.onSelect}/>
```

Figure 35. Code snippet of the root cause content container passing a callback function to child component.



```
import React,{ Component} from 'react';

class treeMap extends Component{
    constructor(props){
        super(props);
    };
    componentDidMount(){
        var props =this.props;
        function onSelect(data){
            props.onSelect(data);
        }
        const rootCause = window.rootCause;
        this.rootCause =  new rootCause(this.refs.map,props.flare);
        this.rootCause.addListener("onSelect",onSelect);
        onSelect(props.flare);
    }
    componentDidUpdate(){
        this.rootCause.update(this.props.flare);
    }
    render(){
        return(
              <div  ref="map" />
        )
    }
}
export default treeMap;
```

Figure 36. Full code of treeMap component

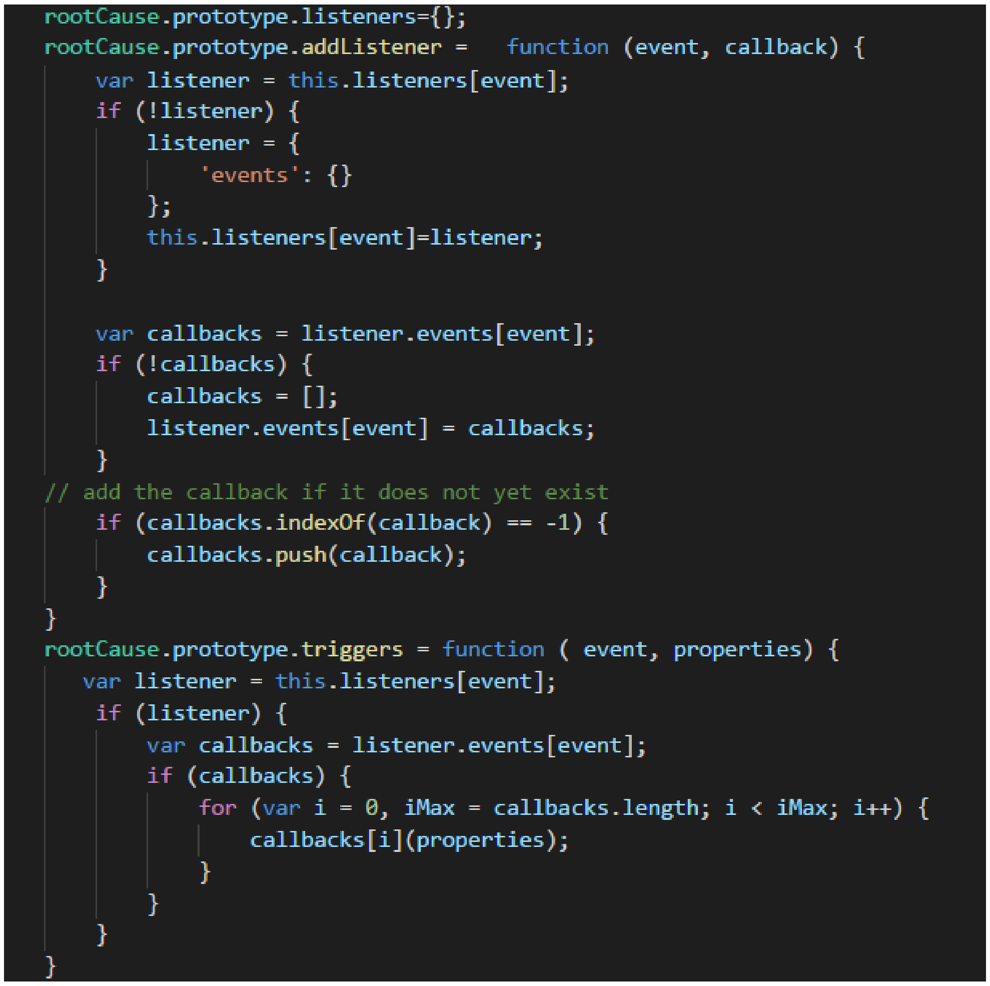


```
class rootcauseLayout extends Component{
    constructor(props){
        super(props);
        this.onSelect=this.onSelect.bind(this);

        this.state={
            nodeName:"All triggers",
            relAlarmData:[]
        };
    };
    onSelect( node){

        var data = DataFilter.filter(this.props.alarmData,this.props.rootCauseFilter);
        data= DataFilter.findRelAlarms(node,data);
        this.setState({relAlarmData:data});
        this.setState({nodeName:node.name});

    }
    renderAlarmContent(){
        const loading = this.props.fetchingAlarms;
         if(loading){
            return <Spinner text="alarms" />
        }
        if(this.props.rootCauseFilter.length>0){
           if(this.props.rootCauseFilter[0].chartType==="frequency"){
                return (
                    <div > <h3>Selected node: {this.state.nodeName}</h3>
                            <h4 className="cw-pad10"> Alarms relevant to selected node*</h4>
                            <AlarmFrequencies data={this.state.relAlarmData} color={'#14a2ec'} title="" />
                            <p>*Alarms which occur within a 10 second range of a trigger</p>
                    </div>
                )
```

Figure 37. Code snippet of root cause content container

5.6 Passing callbacks from a React component to external scripts

Each external script of code is wrapped in a function expression that can be accessed by the React component. The code shown in Figure 38 demonstrates how callbacks can be added and triggered on a functional expression.

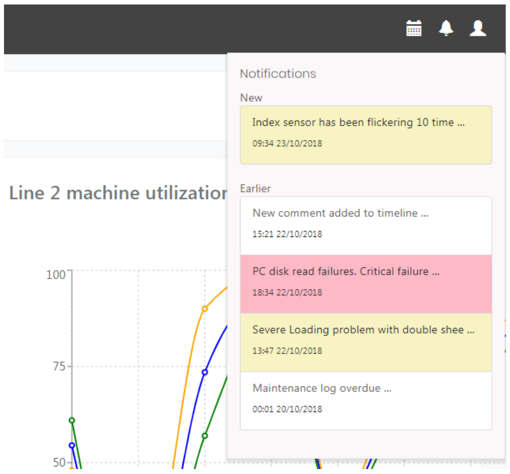


```
rootCause.prototype.listeners={};
rootCause.prototype.addListener =   function (event, callback) {
    var listener = this.listeners[event];
    if (!listener) {
        listener = {
            'events': {}
        };
        this.listeners[event]=listener;
    }

    var callbacks = listener.events[event];
    if (!callbacks) {
        callbacks = [];
        listener.events[event] = callbacks;
    }
// add the callback if it does not yet exist
    if (callbacks.indexOf(callback) == -1) {
        callbacks.push(callback);
    }
}
rootCause.prototype.triggers = function ( event, properties) {
   var listener = this.listeners[event];
    if (listener) {
        var callbacks = listener.events[event];
        if (callbacks) {
            for (var i = 0, iMax = callbacks.length; i < iMax; i++) {
                callbacks[i](properties);
            }
        }
    }
}
```

Figure 38. Code snippet to demonstrate how callbacks can be added and triggered on a functional expression.

5.7 Notification system

As the server side of the application was not yet implemented, the requirement of a notification system was only to demonstrate to the customer how it would look and work so the concept could be easily understood. Figure 39 shows how the drop down list displays an array of notification objects whilst Figure 40 shows the notifications array displayed in the advisor view along with the advice/remedy for each notification. A new notification can be made for example when a user makes a note on the timeline. When a new notification is made, the notification object is pushed to the notification array and an action creator is called to update a new notification Boolean. The top navbar container updates the color of the notification icon whenever the new notification state is updated.


Notifications

New

Index sensor has been flickering 10 time ...
09:34 23/10/2018

Earlier

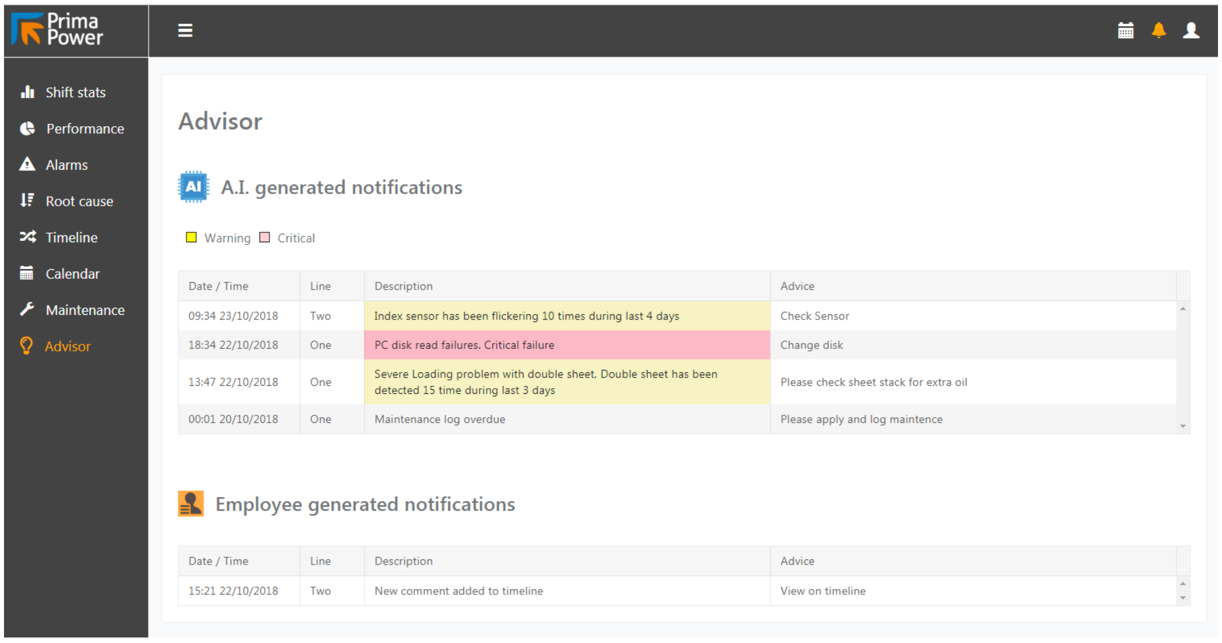
New comment added to timeline ...
15:21 22/10/2018

PC disk read failures. Critical failure ...
18:34 22/10/2018

Severe Loading problem with double shee ...
13:47 22/10/2018

Maintenance log overdue ...
00:01 20/10/2018

Figure 39. The notification drop down list



Advisor

A.I. generated notifications

🟨 Warning 🟥 Critical

Date / Time	Line	Description	Advice
09:34 23/10/2018	Two	Index sensor has been flickering 10 times during last 4 days	Check Sensor
18:34 22/10/2018	One	PC disk read failures. Critical failure	Change disk
13:47 22/10/2018	One	Severe Loading problem with double sheet. Double sheet has been detected 15 time during last 3 days	Please check sheet stack for extra oil
00:01 20/10/2018	One	Maintenance log overdue	Please apply and log maintence

Employee generated notifications

Date / Time	Line	Description	Advice
15:21 22/10/2018	Two	New comment added to timeline	View on timeline

Figure 40. The advisor view

5.8 Implementation of the timeline

5.8.1 Adding the background color feature

The chap-links-timeline javascript library was used to visualize alarm data across a timespan. One of the requirements of the timeline was to display machine times as backgrounds colors with start and end times, however the chap-links-timeline does not have such a feature. The chap-links-timeline has an event named onRangeChange which returns the start and end times of the timeline each time the timeline is zoomed or panned by the user. The chap-links timeline has a method to set the start and end times of the timeline named setVisibleChartRange. The solution was to create two timeline instances, the first to display alarm items and the second to display machine times as blocks of color. All axis and labels were removed from the second timeline instance which was then placed directly behind the transparent first timeline. Using the onRangeChange event and the setVisibleChartRange method the two timeline instances could be synced when the user panned or zoomed the first timeline instance which gives the illusion of one complete timeline. The result of syncing the two timeline instances can be seen in Figure 41.

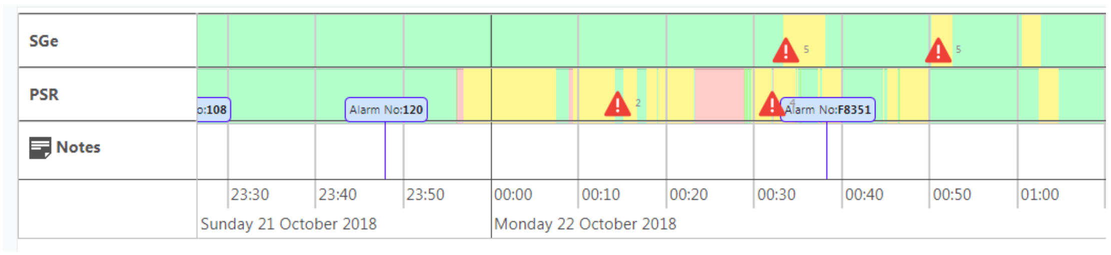
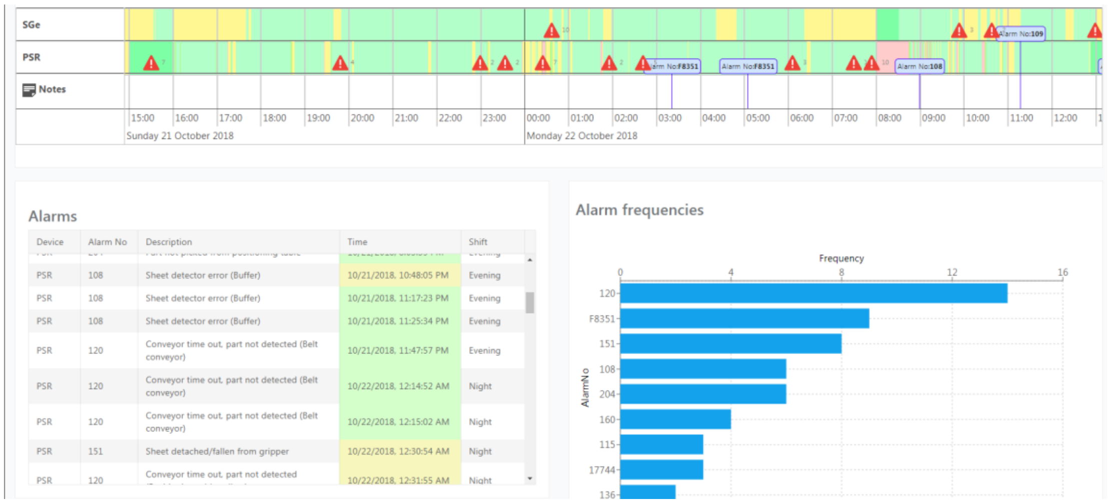


Figure 41. The timeline component with background colors.

5.8.2 Optimization

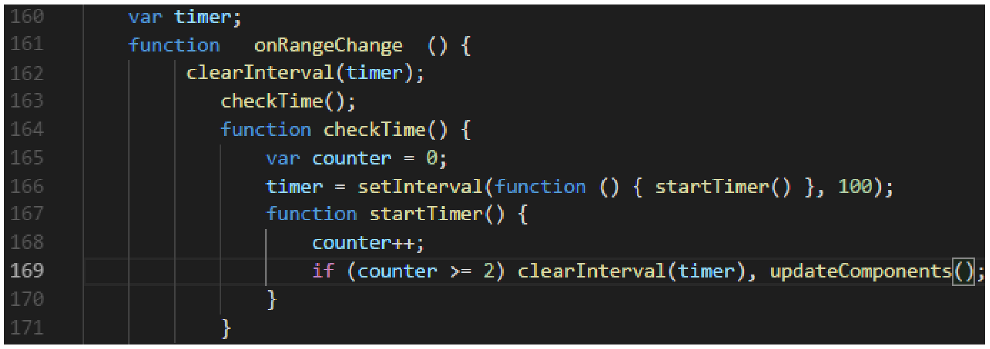
The timeline view contains a grid component and a chart component displaying data from the timeline component as seen in Figure 42. Each time the timeline is zoomed or panned by the user the data visible on the timeline can change and so the grid and chart components have to be updated. This is achieved through callbacks, however the way in which chap-links-timeline works is that the update would occur every single time the timeline is changed which on a mouse scroll zoom would be faster than once every millisecond. Attempting to render the grid and chart components every millisecond results in the event loop being overloaded and crashing. To stop the script from crashing it was decided that the callback should be only fired every 300 milliseconds. This was achieved using javascript setInterval function.The setInterval was cleared each time the onRangeChanged event was triggered and the components only updated if the setInterval of 100 milliseconds occurred three times. Figure 43 is a code snippet demonstrating this method.



Alarms

Device	Alarm No	Description	Time	Shift
PSR	108	Sheet detector error (Buffer)	10/21/2018, 10:48:05 PM	Evening
PSR	108	Sheet detector error (Buffer)	10/21/2018, 11:17:23 PM	Evening
PSR	108	Sheet detector error (Buffer)	10/21/2018, 11:25:34 PM	Evening
PSR	120	Conveyor time out, part not detected (Belt conveyor)	10/21/2018, 11:47:57 PM	Evening
PSR	120	Conveyor time out, part not detected (Belt conveyor)	10/22/2018, 12:14:52 AM	Night
PSR	120	Conveyor time out, part not detected (Belt conveyor)	10/22/2018, 12:15:02 AM	Night
PSR	151	Sheet detached/fallen from gripper	10/22/2018, 12:30:54 AM	Night
PSR	120	Conveyor time out, part not detected	10/22/2018, 12:31:55 AM	Night

Figure 42. The timeline view.

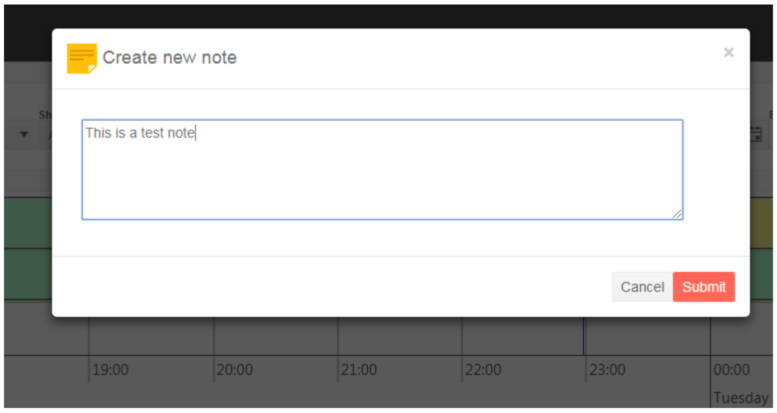


```
    var timer;
    function   onRangeChange  () {
        clearInterval(timer);
          checkTime();
          function checkTime() {
              var counter = 0;
              timer = setInterval(function () { startTimer() }, 100);
              function startTimer() {
                  counter++;
                  if (counter >= 2) clearInterval(timer), updateComponents();
              }
          }
```

Figure 43. Code snippet demonstrating how amount of user updates is controlled

5.8.3 Adding a note to the timeline

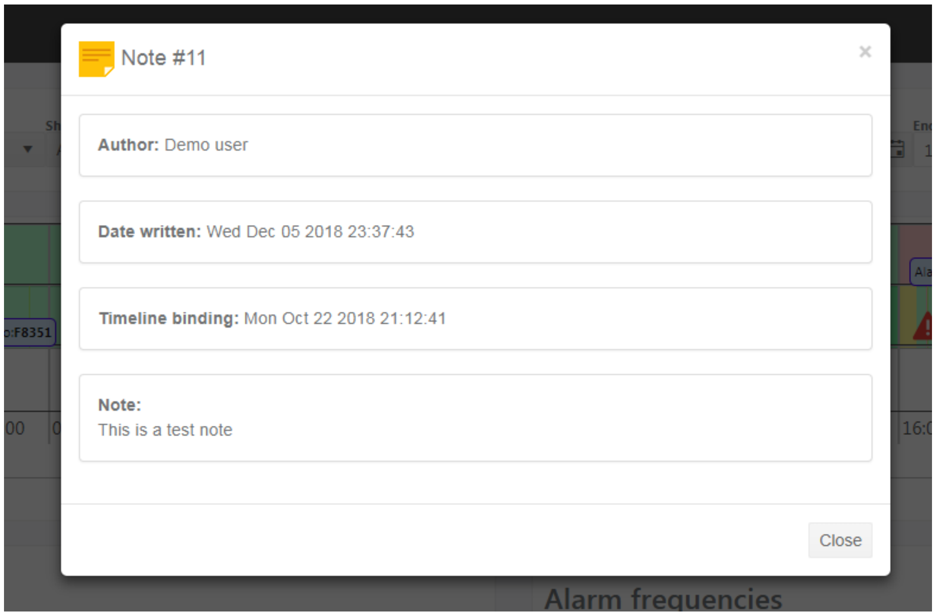
The chap-links-timeline library contains an event named select. When a timeline object is clicked the select event returns the object and when an empty space on the timeline is clicked, an object is returned containing the time of the point clicked. Callbacks are used to the send the select event up to the parent container where a popup modal component can be displayed. When a timeline item is clicked the modal renders the item information (Figure 45). When an empty space on the timeline is clicked the modal displays a text area to add a new note (Figure 44). When the user submits a new note, the note object is pushed into note array and the timeline is updated to show the note.



Create new note

This is a test note

Figure 44. The modal to create new note



Note #11

Author: Demo user

Date written: Wed Dec 05 2018 23:37:43

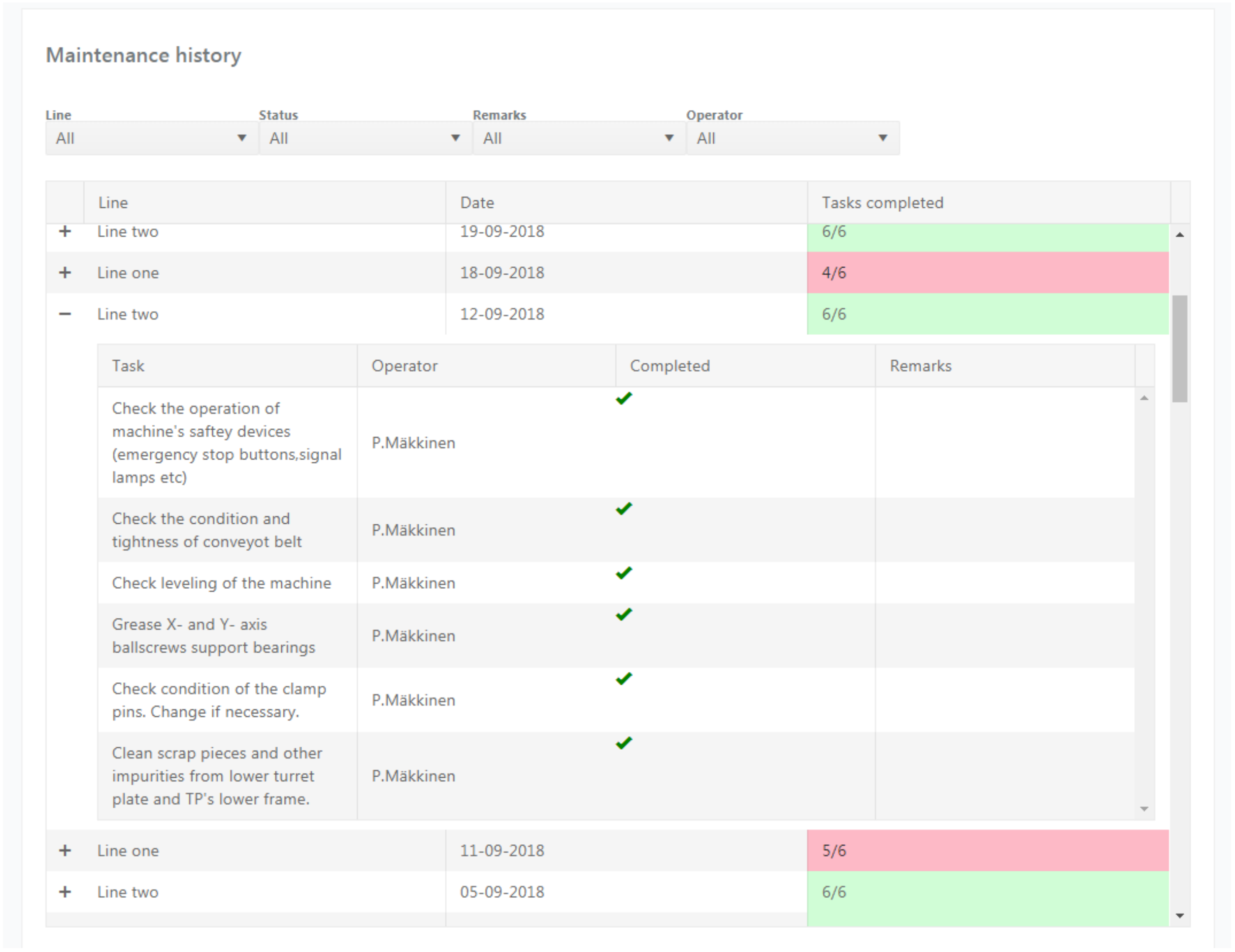
Timeline binding: Mon Oct 22 2018 21:12:41

Note:
This is a test note

Figure 45. Modal to display note

5.9 Creating custom cells in kendoUI grid component

The kendoUI React grid component is used throughout the application. To give grid cells custom colors depending on a condition, custom components were made. Figure 46 shows the Maintenance view which has a nested grid with custom background colors and content. Figure 47 shows the grid component and Figure 48 shows a code snippet to demonstrate how custom cells and nested grid components for the kendoUI grid component are implemented.


Maintenance history

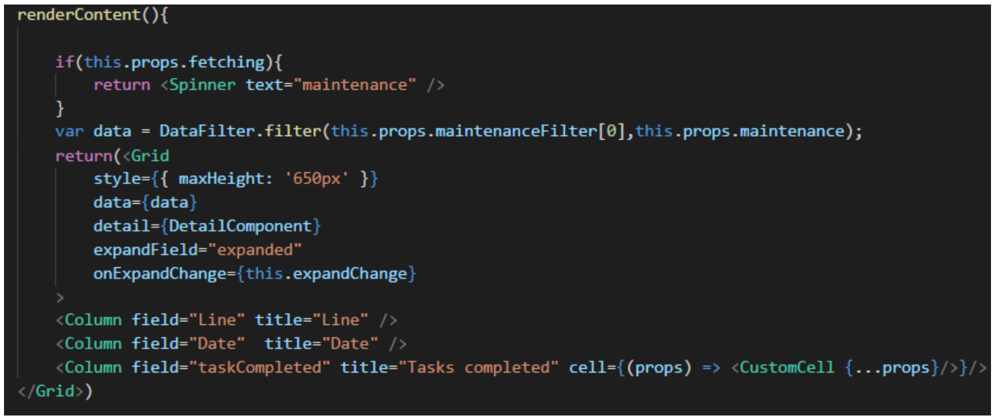
Line: All | Status: All | Remarks: All | Operator: All

	Line	Date	Tasks completed
+	Line two	19-09-2018	6/6
+	Line one	18-09-2018	4/6
−	Line two	12-09-2018	6/6

Task	Operator	Completed	Remarks
Check the operation of machine's saftey devices (emergency stop buttons,signal lamps etc)	P.Mäkkinen	✓	
Check the condition and tightness of conveyot belt	P.Mäkkinen	✓	
Check leveling of the machine	P.Mäkkinen	✓	
Grease X- and Y- axis ballscrews support bearings	P.Mäkkinen	✓	
Check condition of the clamp pins. Change if necessary.	P.Mäkkinen	✓	
Clean scrap pieces and other impurities from lower turret plate and TP's lower frame.	P.Mäkkinen	✓	

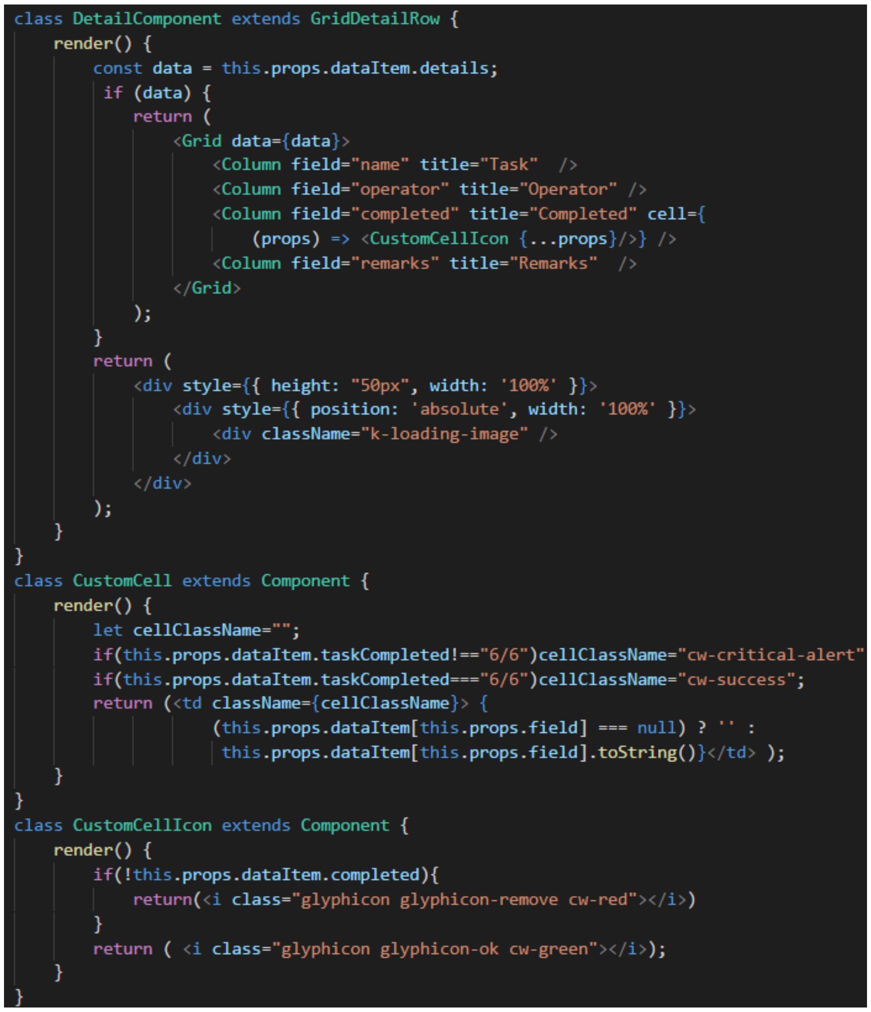
	Line	Date	Tasks completed
+	Line one	11-09-2018	5/6
+	Line two	05-09-2018	6/6

Figure 46. Maintenance history grid view



```
renderContent(){

    if(this.props.fetching){
        return <Spinner text="maintenance" />
    }
    var data = DataFilter.filter(this.props.maintenanceFilter[0],this.props.maintenance);
    return(<Grid
        style={{ maxHeight: '650px' }}
        data={data}
        detail={DetailComponent}
        expandField="expanded"
        onExpandChange={this.expandChange}
    >
    <Column field="Line" title="Line" />
    <Column field="Date"  title="Date" />
    <Column field="taskCompleted" title="Tasks completed" cell={(props) => <CustomCell {...props}/>}/>
</Grid>)
```

Figure 47. KendoUI grid code snippet



```
class DetailComponent extends GridDetailRow {
    render() {
        const data = this.props.dataItem.details;
         if (data) {
            return (
                <Grid data={data}>
                    <Column field="name" title="Task"  />
                    <Column field="operator" title="Operator" />
                    <Column field="completed" title="Completed" cell={
                        (props) => <CustomCellIcon {...props}/>} />
                    <Column field="remarks" title="Remarks"  />
                </Grid>
            );
        }
        return (
            <div style={{ height: "50px", width: '100%' }}>
                <div style={{ position: 'absolute', width: '100%' }}>
                    <div className="k-loading-image" />
                </div>
            </div>
        );
    }
}
class CustomCell extends Component {
    render() {
        let cellClassName="";
        if(this.props.dataItem.taskCompleted!=="6/6")cellClassName="cw-critical-alert"
        if(this.props.dataItem.taskCompleted==="6/6")cellClassName="cw-success";
        return (<td className={cellClassName}> {
                    (this.props.dataItem[this.props.field] === null) ? '' :
                     this.props.dataItem[this.props.field].toString()}</td> );
    }
}
class CustomCellIcon extends Component {
    render() {
        if(!this.props.dataItem.completed){
            return(<i class="glyphicon glyphicon-remove cw-red"></i>)
        }
        return ( <i class="glyphicon glyphicon-ok cw-green"></i>);
    }
}
```

Figure 48. Code snippet to demonstrate custom cells and nesting component for KendoUi grid

6 Testing

On completion of each module, manual testing was used where the developer acted as a user of the application to verify all the features work correctly and any bugs were found and recorded to be fixed. During the development of each module, unit testing and integration testing techniques were used. The testing framework used was Jest and the library Enzyme was used to give more options in creating tests. Jest is installed and run by using npm and the results of the test can be viewed in the command line. When a test fails, information is given to as why the test failed, the test then runs again each time there are then changes made to the code until the test passes. The process of Jest testing can be seen in Figure 49.

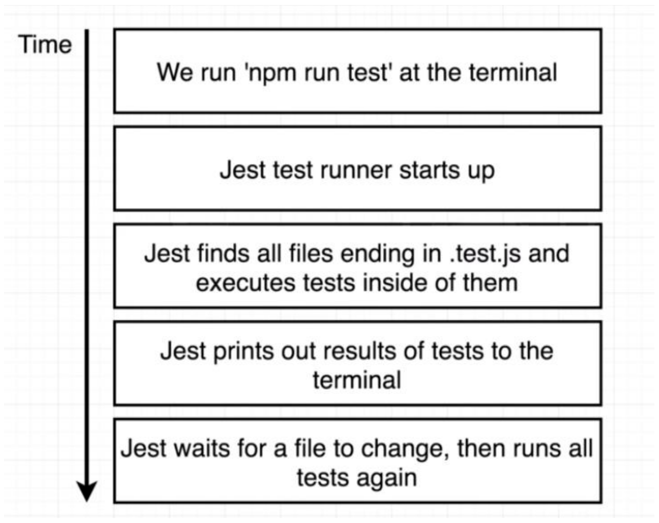
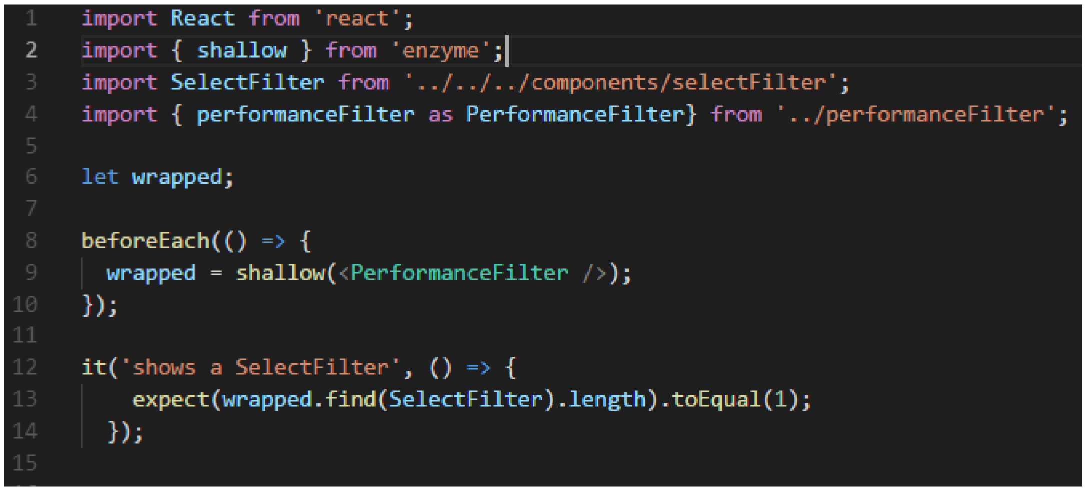

Time
We run 'npm run test' at the terminal
Jest test runner starts up
Jest finds all files ending in .test.js and executes tests inside of them
Jest prints out results of tests to the terminal
Jest waits for a file to change, then runs all tests again

Figure 49. A diagram of the Jest testing process

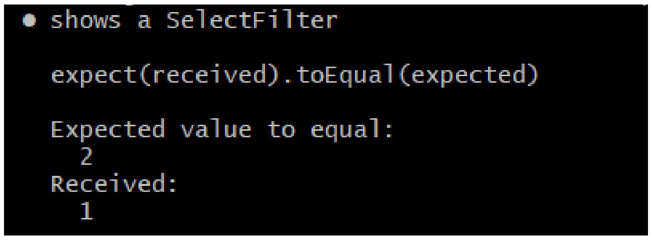
Unit testing is done by only testing individual units of code, an example of a unit test can be seen in Figure 50. The unit test shown in Figure 50 is a test to see if the performanceFilter component contains one SelectFilter component using the enzyme shallow feature. Before passing a unit of code as tested, the test should first be failed.

To fail the test shown in Figure 50, line 13 of the code snippet was changed so that the toEqual method on the expect method had a parameter of 2 instead of 1. The test fail message shown in the command line can be viewed in Figure 51.



```
 1  import React from 'react';
 2  import { shallow } from 'enzyme';
 3  import SelectFilter from '../../../components/selectFilter';
 4  import { performanceFilter as PerformanceFilter} from '../performanceFilter';
 5
 6  let wrapped;
 7
 8  beforeEach(() => {
 9    wrapped = shallow(<PerformanceFilter />);
10  });
11
12  it('shows a SelectFilter', () => {
13      expect(wrapped.find(SelectFilter).length).toEqual(1);
14    });
15
```

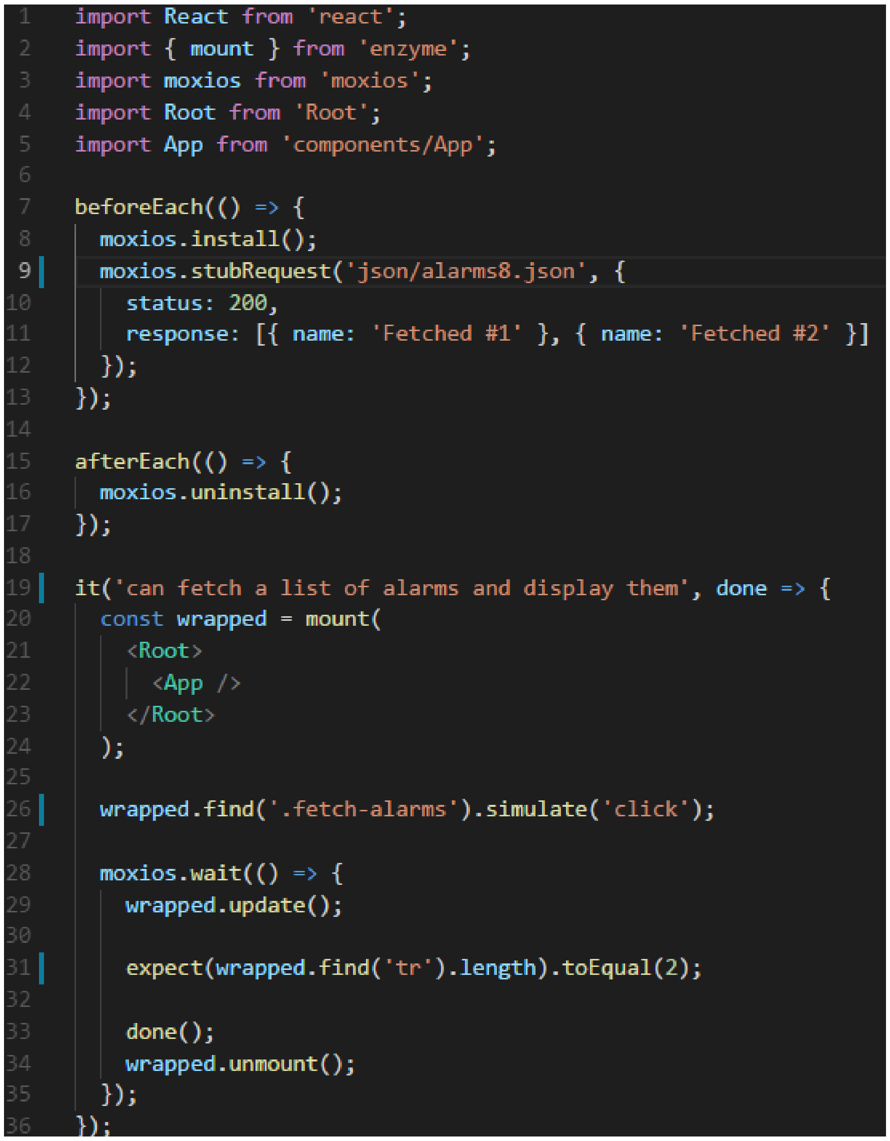
Figure 50. An example of a unit test



```
● shows a SelectFilter

  expect(received).toEqual(expected)

  Expected value to equal:
    2
  Received:
    1
```

Figure 51 Command line displaying failed test

Whilst unit testing can be suitable for testing individual units of code, it can be unnecessary with applications with a lot of boilerplate code such as the application of this thesis work. As this application uses the redux library, action creators and reducer functions are required which creates a lot of boilerplate. Reducers are very simple and writing a unit test for each reducer function takes up time that could be used more productively. Integration testing is when many units of code are combined and tested together which makes it suitable to use for redux code as it means the action creators, reducers and components can be combined in one test. The code snippet in Figure 52 shows an example of an integration test using the Jest, enzyme and moxios libraries. The test is begun by simulating the initialization of the code which in the case of the example is a button click, the test can find this button by using the HTML class attribute. The moxios library enables the possibility to give a mock API response. The moxios stubRequest method takes in a URL and whenever that URL is requested in the test, moxios gives the mock response which in the case of the example is a JSON array with 2 objects with the attribute ‘name’. This mock request and response even though very fast, taking just microseconds to complete still needs to be waited on before the test can continue and to achieve this, the moxios.wait method is used. To complete the test, the component displaying the response is checked to see if the expected output is correct which in this example case is two table row elements.



```
import React from 'react';
import { mount } from 'enzyme';
import moxios from 'moxios';
import Root from 'Root';
import App from 'components/App';

beforeEach(() => {
  moxios.install();
  moxios.stubRequest('json/alarms8.json', {
    status: 200,
    response: [{ name: 'Fetched #1' }, { name: 'Fetched #2' }]
  });
});

afterEach(() => {
  moxios.uninstall();
});

it('can fetch a list of alarms and display them', done => {
  const wrapped = mount(
    <Root>
      <App />
    </Root>
  );

  wrapped.find('.fetch-alarms').simulate('click');

  moxios.wait(() => {
    wrapped.update();

    expect(wrapped.find('tr').length).toEqual(2);

    done();
    wrapped.unmount();
  });
});
```

Figure 52. An example of an integration test

7 Conclusion

The main goal of the application described in this thesis was to give a demonstration of an online analytics tool to customers at the euroBlech trade fair as part of the Prima Power exhibition. All the requirements of the Prima Power analytics application were met and Prima Power were able to give a demonstration of a professional, working application. Challenges in developing this application included learning how to make an application with the React javascript framework and deciding on what were the most appropriate libraries to use. Throughout the development of this application, the technologies used were continually studied and the techniques used were continually revised. The application is designed to be scalable so that Prima Power can continue to add new features to the existing application in the future.

8 References

/1/ Prima Power – Last access 06/12/2018

https://www.primapower.com/company/

/2/ EuroBlech – Last access 06/12/2018

https://www.euroblech.com/2018/english/event/about-euroblech/

/3/ node.js – Last access 06/12/2018

https://nodejs.org/en/about/

/4/ ReactJS – Last access 06/12/2018

https://reactjs.org/

/5/ Flux – Last access 06/12/2018

https://scotch.io/tutorials/getting-to-know-flux-the-react-js-architecture

/6/ Redux – Last access 06/12/2018

https://redux.js.org/

/7/ Thunk – Last access 06/12/2018

https://redux.js.org/api/applymiddleware

/8/ Create-react-app – Last access 06/12/2018

https://reactjs.org/docs/create-a-new-react-app.html

/9/ KendoUI – Last access 06/12/2018

https://www.telerik.com/kendo-ui

/10/ Recharts – Last access 06/12/2018

http://recharts.org/en-US/

/11/ D3.js – Last access 06/12/2018

https://d3js.org/

/12/ Chap-links-timeline – Last access 06/12/2018

https://almende.github.io/chap-links-library/timeline.html

/13/ Twitter bootstrap – Last access 06/12/2018

https://getbootstrap.com/

/14/ Just in mind – Last access 06/12/2018

https://www.justinmind.com/

/15/ Deenar Toraskar d3 block – Last access 06/12/2018
https://bl.ocks.org/deenar/336efdeb97f1c79d4a6b

/16/ Ashley Kitson UML React design – Last access 06/12/2018

http://zf4.biz/blog/visualizing-react