Front-end development of a modern CRM web application using React.

Alan Wallace Ross

This dissertation was submitted in part fulfilment of requirements for the degree of MSc Software Development.

Dept. of Computer and Information Sciences

University of Strathclyde

August 2019

DECLARATION

This dissertation is submitted in part fulfilment of the requirements for the degree of MSc of the University of Strathclyde.

I declare that this dissertation embodies the results of my own work and that it has been composed by myself. Following normal academic conventions, I have made due acknowledgement to the work of others.

I declare that I have sought, and received, ethics approval via the Departmental Ethics Committee as appropriate to my research. I give permission to the University of Strathclyde, Department of Computer and Information Sciences, to provide copies of the dissertation, at cost, to those who may in the future request a copy of the dissertation for private study or research.

I give permission to the University of Strathclyde, Department of Computer and Information Sciences, to place a copy of the dissertation in a publicly available archive. (please tick) Yes [ ] No [ ]

I declare that the word count for this dissertation (excluding title page, declaration, abstract, acknowledgements, table of contents, list of illustrations, references and appendices is 16623.

I confirm that I wish this to be assessed as a Type 1 2 3 4 5

Dissertation (please circle)

Signature:

Date: 17/08/2019

Abstract

This project documents the front-end development of a customer relationship management (CRM) web application to replace the CRM system of a client that is currently using Microsoft Dynamics 2011. The aim of this new software is to help better integrate all of the client's business units, offer them flexibility of staff resources to cover other business areas (as the CRM and customer journey will be similar), and cost savings due to efficiencies made from the customer journey process. To achieve this, an initial training period was needed where Javascript and React tutorials were used for around 2-3 weeks before work finally began on an existing base of code that was taken from a previous project of the company I work for. This project will hence involve topics such as the development model of the application, the design phase of the application, the implementation of the application, the testing of the application. Due to the size of this application however, the project won't actually be completed before the deadline of this dissertation and so while there are user testing deployments, this dissertation won't be able to discuss the final deployment of the application.

Acknowledgement:

I would like to acknowledge my supervisor Konstantinos Liaskos for his support and guidance over not only the course of the project but over the course of the degree too. I would also like to thank my colleagues at work for their support and advice over the course of the project too.

Contents

List of figures:

1 INTRODUCTION

With businesses aiming to maximize the efficiency of their business processes, one studied way of doing this is by introducing CRM (customer relationship management) software according to Howard (2019). The role of this software is to be able to manage relationships with customers. As time has progressed, CRM's have progressed as well [2], they are capable of storing all of the customer's data in an organized and easily accessible database that allows staff to carry out their business functions.

CRM's aren't just databases though, they provide other uses as well. The business can ask for business feature's to be included in the CRM and a software company like Pulsion Technology can also help in guiding what is really needed as well as making sure the specifications of the software are detailed enough to ensure the client will be satisfied with what they are getting. This CRM is what has been worked on for the past two months and will continue to be worked on until November before it will be released. This project will document the development of the front-end of this CRM, specifically the Organisation's User story which in total had one person responsible for it while at the same time was part of a team of four.

This team of four was developing this CRM because their client's current system is based on old technology (Microsoft Dynamics)[1] and wasn't custom built for their business functions and so is very limited in what functions it can carry out as well as lacking in performance. The aim of this new software is to help them better integrate all of their business units, offer flexibility of staff resources to cover other business areas (as the CRM and customer journey will be similar), and create cost savings due to efficiencies made from the customer journey process.

1.1 OBJECTIVES:

For the purposes of this dissertation my main objectives were to learn languages, libraries as well as web services. Furthermore from a more practical perspective, the development of the

[1] https://dynamics.microsoft.com/en-gb/

"Organisations" section of the CRM project. This section which could also be called a User story is responsible for the CRM being used to record organisation details so that the clients can keep track with what organisation's they have engaged with and when.

As part of learning new languages, libraries and web services my objectives were as follows:

1. Learn JavaScript
2. Learn React
3. Learn more about AWS and how to use it.
4. Learn other web related languages such as HTML and CSS.

And as part of developing the Organisation's section of the CRM:

1. Develop a page where you can create an organisation.
2. Develop a page that lists these organisations.
3. Develop a page where a selected organisation's details can be viewed as well as edited.

1.2 Outcome:

Through my initial training as well as learning throughout applying the pages, I will be able to develop the pages mentioned. Through these pages, system administrators will be able to view what organisation's there are through a table form factor, create an organisation based on the form data that the CRM requires to create one and then have the ability to select an organisation to view that organisation's details, have the choice to edit that data as well being able to "de-activate" that organisation.

1.3 Report Structure:

The following sections in this dissertation will document the methods and processes followed for the development of this project. There will also be a discussion on what related work has been done in terms of history, motivation and technology. Following on from that, the project's problem description and specification will be discussed along with the system design of the software as well as the methodology. With the methodology covered, the training, the

technologies and the detailed design and implementation of the system will be documented.

Then the verification and validation of the system and finally the conclusion of the project.

2 Related Work

2.1 History:

CRM's back in the 1980's weren't known as CRM's back then. In fact, back then, what existed was only digital rolodexes and database marketing (applying statistical methods to analyse and gather customer data). The digital rolodex allowed for efficient storage and organization of customer contact information and was known as "contact management software" (History of CRM Software, 2019).

It wasn't actually until the 1990's before digital rolodexes and database marketing were merged together to offer "sales force automation" (SFA). SFA refers to software applications that aids in sales management by providing automated workflows that create a simple sales process to manage business leads, sales forecasts and team performance.

As SFA's progressed through the 1990's, SFA and contact management continued to offer more and more automation of business tasks like inventory control as well as being able to provide much more useful customer information, that by 1995, they closely resembled a modern CRM software and it was by the end of 1995 that the term CRM was given to this kind of software (History of CRM Software, 2019).

From there, CRM's continued to evolve and provide even a broader range of services as more and more competition was provided by emerging technology companies.

Then came the end of the 90's with the launch of Software as a Service (SaaS) which was more suited to smaller businesses.

After the bursting of the dot-com bubble in the 2000's though, a lot of technology companies suffered, and hence the CRM industry were one of the industries that were affected the most.

CRM's didn't stop evolving though, and the purpose of CRM's grew to also manage all business relationships as well and companies such as Microsoft entered the CRM market by introducing their Dynamics CRM software alongside other giants also clamouring for control over the CRM industry.

Since then, CRM software has also moved to being cloud-based and Social CRM's were also invented which embraced the use of social media to interact with their customer's instead (History of CRM Software, 2019).

With products like the Microsoft Dynamics and Salesforce CRM, they are an easy solution for businesses that are starting up and want to take advantage of the benefits that a CRM can provide but as a business grows and matures, the business can outgrow that one size fits all kind of CRM and needs a CRM that is fully customized specifically to the needs of the business.

The above only gives a brief history of Customer Relationship Management software though.

2.2 MOTIVATION:

The motivation behind Customer Relationship Management is that it "characterises a management philosophy that is a complete orientation of the company towards existing and potential customer relationships" as said by Raab (2008). This is important because according to Peppers and Rogers(2011), there seems to be a global trend in customer relationship management that points to a shift from a transactional model where it's based on each transaction between the business and a customer, to a relationship model where it is about a customer journey and not about transactions, this involves aspects of a business such as customer support which has a large impact on the customer's satisfaction with a company. What this means is that it can be argued that it's not enough for businesses to only have transactional relationships with their customers anymore to be able to ensure that the business will see long-term growth.

Pepper and Roggers(2011) also says that if long-term relationships with customers are not aimed for by businesses, that businesses are less likely to maintain loyalty from customers as well as flexibility of customers when it comes to customer's increasing expectations.

Mathur (2010) helps back up the notion of how important customer relationship management software is by providing a large range of customer relationship techniques that are used by multinational businesses.

Also according to Nucleus Research (2019) CRM's pay back $8.71 for every dollar spent, and this has actually increased from 2011 where it used to be only $5.60 for every dollar spent.

Another finding based on a survey by Ivey at Software Advice (2019) is that a large majority said that their CRM system offered improved access to customer data.

With all this motivation for a business having a CRM, what features can a CRM provide then?

Well it depends on what kind of CRM is needed. It can be an Operational CRM, one in which helps streamline processes, an Analytical CRM that helps source big data so to provide the business with insights on customers and then there is the Collaborative CRM, these are designed to improve communication and teamwork among the business staff. Of course, it doesn't need to be that only one can be chosen, functionality can be added from other parts to provide a hybrid CRM. According to Cleveroad Inc (2019), the main features of a CRM are:

1. Managing Contacts
2. Setting Reminders
3. Editing calendars
4. Managing tasks
5. Generating simple reports

Pulsion Technology has already did many of these type of customer relationship management systems in the past with older software using technologies such as C#, SQL and PHP, and only

recently they have been using new technologies to implement them. The reason for this was to break away from the previous monolithic architecture that was being used to one where everything was separated and could be re-used. It is actually these past projects which provided the wire-frame for the project that's being documented in this dissertation since it has the same basic architecture as what the other projects had. The technologies that have been used in the front-end are Javascript, React, CSS and HTML while in the backend, a microservice architecture has been adopted using Javascript (Node), Amazon Web Services (AWS) and Serverless.

One of the reasons for this shift is that Javascript can be used for both the front-end and the back-end and so it is much easier for people to switch between developing the front-end and back-end since they don't need to learn another language or have to remember the subtle differences between how two languages deal with syntax, or semantics. This in turn also saves on costs of time spent training. Another reason would be the fact that JavaScript is the most popular language now and so not only is it easier for recruitment purposes but at the same time there it is seeing a lot of work and support going into it. It's also the standard programming language of the web too so it only makes sense for a company building web application's to deviate towards it.

And to further elaborate on why the client asked for a custom CRM as opposed to their current Microsoft Dynamics solution was that Microsoft Dynamics couldn't be set up to automate the complicated processes that the client had, for example for this application, an "entity" has to be created based on the creation of another "entity" however there is various conditional behaviour around this functionality as well which Microsoft Dynamics just didn't support.

To give a better example of what an "entity" is, it can be considered a Referral or an Organisation, or a Customer or something else. There is a process for a Referral that must be followed however it is a complex process that can lead to a customer but might not and then data has to be handled properly so that if the referral doesn't become a customer, then only a limited amount of data on the referral should be stored.

3 PROBLEM DESCRIPTION AND SPECIFICATION

3.1 PROBLEM DESCRIPTION:

As stated before in the description, the problem that this project aims to solve is primarily the issue of keeping the different departments of a business connected so that they can organize all their different departmental information into one cohesive system that the users (the users of the CRM will be the staff associated with the client) of that business can easily access and take advantage of at any time. This then makes it easier than ever for departments to

coordinate with one another and at the same time make it possible to offer each customer a personalized customer journey.

3.2 ORGANISATION REQUIREMENTS:

When first starting this project, the User Stories document had nearly 160 pages, and over the course of development so far, the number of pages has increased to 177. In these 177 pages, there are 59 different sections, each containing their own story as well as a priority level of High, High/Medium, Medium/Low and Low. This dissertation is mostly concerned with User story 14 which focusses on Organisations and is a high priority requirement. The description for this story is as follows: "As an administrator I want to be able to record details of an organisation so that specialists can link the customer records to them when they have been involved in the journey so that we can track when we have engaged with different organisations."

The acceptance criteria is as follows:

"

- Fields to record
    - Organisation Name
    - Address 1
    - Address 2
    - Local authority area
    - Postcode
    - Phone
    - Email (global)
    - Contact Name (multiple)
    - Contact Phone (linked to contact)
    - Contact email (linked to contact)
    - Type of organisation
    - If certain type of organisation, record SIC code (look up)
    - Last contact (date)
    - Last contact (programme)
    - Last contact by whom (name)
- Ability to display last contact information over the last two months
- Ability to search on local authority area
- Ability to migrate data from spreadsheet into organisations record

Notes:

Going forward with other programmes, there will be additional 'type' options and likely a secondary 'type'.

Requirements here may change as other programmes are added to the CRM."

The way Organisation's was done was through first replicating the content on the User's page on the Organisation's page and then replacing User variables with Organisation variables as well as using other endpoints for the use of different microservice instances. As well as that, organisation's included a more complex form that required the use of an external date-picker package that was installed from NPM[2]. The multiple contacts part of the Organisation form was the most complex piece of the Organisation form though and while it wasn't mentioned in the Organisation's user story, implementing the form validation was also required and was also not simple.

3.3 OTHER REQUIREMENTS

There was also times where work was done on other User Stories too including the User page and the Customers page that will be discussed in the implementation chapter. To give better context to the CRM here are the requirements for them too:

3.3.1 Customers:

"Description

As an organisation I want to create a global contact record accessed across the organisation so that customers can be supported across all areas of the business (i.e referred to other areas of the business for support)

Acceptance Criteria

- Fields will be Name>Address>Phone 1>Phone 2>Email>NI Number>Consent to store data>Latest risk rating (overall)
- Basic customer record will have various programme records associated with it (it is likely the programme record will be created first to generate this information, this is to be confirmed with Pulsion.
- Ability to attach a customer photograph to the record
- Referral mechanism between programmes from the contact record (as more projects are built into CRM this may be an email to a shared inbox or an in system alert)
- Fields:
    - WG Programme Referred to (as more CRM systems are built there will be a list of contacts supplied who will deal with referrals by project – assume in system at this point but potentially via email)
    - Referral from (WG programme list – tbd as they are built)

[2] https://www.npmjs.com/

- Referral to (WG programme list – tbd as they are built)
- Date of referral
- Referral information (currently free text, as other programmes are built on CRM there will be more defined requirements here)
- Outcome of referral: accepted/unsuitable/follow up

Notes

The global contact record should contain a high level outline of the customer's time with the WG from point of first contact (or referral in), to programmes engaged with and start/end dates.

There should be a mechanism that allows cross referral on programmes (e.g. New Routes can refer to an employability programme (yet to be developed on the new CRM) through the basic customer record.

Governance Manager: confirmed they will ensure privacy notice reflects this requirement and include the photo request"

3.3.2 Users:

"Create Users

Description

As an ICT Co-ordinator I want the system to integrate with the Wise Group active directory active directory so that assigning a user to a specific AD group will automatically assign them a corresponding security role.

Acceptance Criteria

- Ideally, there is no need to separately provision a user account within the application; the creation of a user in Active Directory (and it's assignment to an AD group) and the synchronisation between AD and the application is sufficient for that user to access the application.

Within a short, pre-defined time period of assigning a user to an AD group (ideally no more than 1 hour; preferably 15 mins. or less), the user is able to authenticate into the application and has the expected access rights of the role assigned.

Notes

Ideally, user access rights should be assignable by ICT whilst not allowing staff access to customer data.

Partners will not as a matter of course be added to the active directory, although there may be some instances, in the future, where we wish partners to be part of the AD and access single sign on.

3.3.3 Single Sign On

Description

As a user I want to use my windows password to sign in to the CRM so that I don't have to sign in multiple times (and remember multiple passwords)

Acceptance Criteria

- Logging in to the application redirects the user to the active directory sign-in page for authentication

The authentication process honours / fully works with 2-factor authentication

Notes

The login process should be simple for the user whilst allowing the enforcement of IT security requirements.

After 15 minutes of inactivity the user will be logged out (timeout user story)

Single sign-on will not be a function available to partners (they should access through a link and create username and password) at present, but this may change in the future so please be aware of this potential future requirement.

Timeout

Description

As the ICT role I want CRM access to timeout after 15 minutes of inactivity so that access to large volumes of sensitive data is minimised

Acceptance Criteria

- Inactivity by user for 15 minutes = system timeout
- Autosave function included (to ensure system does not timeout when users are typing if they have not saved within 15 mins)

Notes

- Must not clear content of CRM (i.e. users do not lose work entered onto CRM even if not saved), but timeout means you must enter password again"

4 System Design

In this chapter, the design and development methodology of the system is documented as well as the system's structure. Since this dissertation is mostly focussed on the front-end development of a web application that already had an architecture in place as well as UI that was already decided between the graphic designer and the client, there is a limited discussion on why the system was designed the way it was.

4.1 Design and Development Methodology:

For this project, the basic building blocks were taken from the most recent previous project that had been completed. The decision behind this is that the company had invested money into developing systems that would be more made of separate components rather than monolithic systems and to see a return on this, you need to reuse what has already been built. The issue with this is that the first time the system was developed in React as well as AWS and Serverless, people were not very experienced with the technology and so there are some flaws in the design of this system but the methodology behind this is that with each project there will be incremental improvements in the separate components so that as time goes by, systems will become more and more refined and also be quicker to build as people build more experience with the technology.

As for why React was chosen over Vue or Angular was mostly because it's not too difficult a learning curve compared to Angular but at the same time it also boasts a much more larger and mature community than the likes of Vue. React is also just a library for creating UI's (View) rather than a fully-fledged framework that handles everything such as the Model, View and Controller. As for why AWS was picked over Azure, is the fact that at this time AWS is the dominant web service provider out of all the cloud service providers with almost twice as much marketshare as Azure which comes in 2nd place (Canalys.com 2019). Due to the dominance some of the services AWS provides are much cheaper than some of the services Azure provides, for example, Amazon's elastic search service might cost £30 a month compared to a different service on Azure that provides the same functionality but in a different form factor. Amazon's elastic search provides search functionality for our DynamoDB tables, something that DynamoDB lacks by itself.

For the development of such a large project, a combination between a traditional methodology and an agile methodology was followed.

The explanation behind this is that a large bulk of the requirements were there from the beginning and then the analysis was done as well as the design. For the most part, these three aspects of the software development life cycle have only seen minor change. There still has to be a user acceptance test though so the above could change in the future. So far, there has only been a cycle between coding and testing. Overall though, there has been a roadmap from since the start for the project and so there has been a certain amount of structure which is associated with the waterfall method.

From a day to day standpoint, there is a daily stand up at 11:30am where each member of the team tell the rest of the team three things. 1. What they did yesterday 2. What they are doing today, and 3. If anything is blocking their progress. These standups can be finished in 5 minutes or they can last for 15 minutes depending on what is being worked on, as sometimes discussions about the design of a certain feature are brought up. The team also has JIRA tickets who are responsible for creating and moving along on our Kanban board. The flow the team has for the board is as follows: The initial tasks/stories are put in the backlog and then tasks/stories are chosen from that back-log into the to-do column. The next column is the selected or development column and to the right of that again is the “in progress” column. Then it is the “awaiting client feedback” column and then then “ready for test” column. After that, is ofcourse the “in test” column and then once it has been tested, it moves onto the “ready for user acceptance test (UAT)” column and once it is time for the UAT, they are moved to the “UAT” column. Finally, if there are no problems with that task or story, it would move to “done”. This won’t always be the case though and some of these tasks/stories may go back to the start in the form of bugs or even due to a change in the client’s mind on what behaviour they want. In this case, the process of the task moving to the right of the board would start all over again. The idea behind using a board like this is to make it easier to see what tasks are making progress and just as importantly, what tasks aren’t seeing progress:

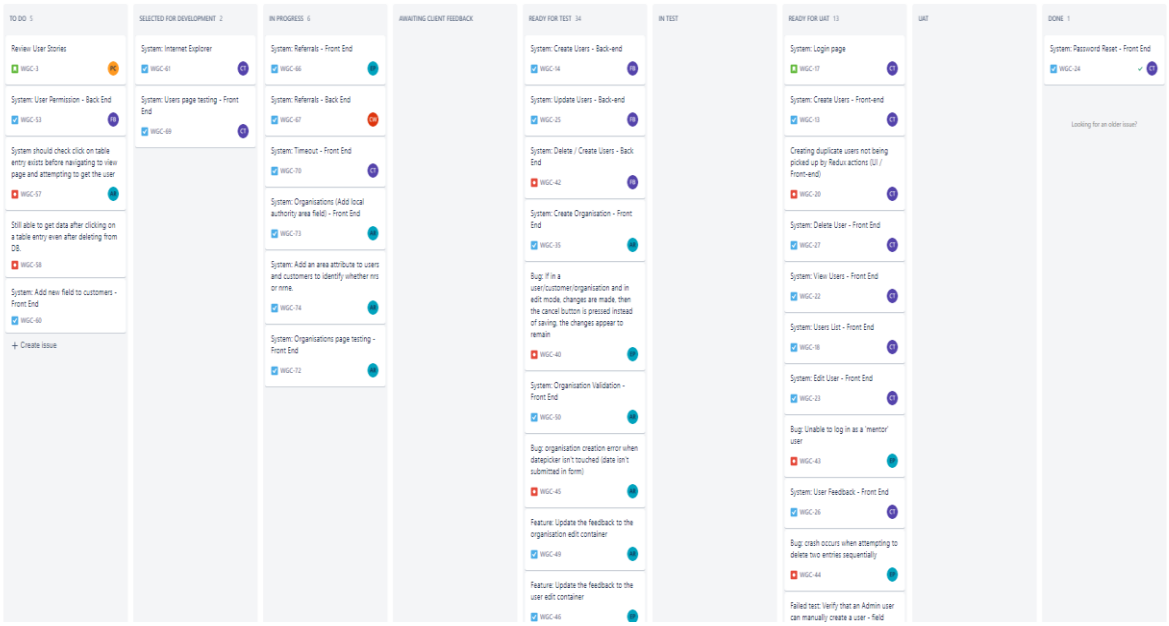

Figure 1: JIRA's kanban board

In Jira it was also common practice to make child tickets within the original tickets due to the fact that original tickets would be associated with a User Story and user stories tend to have more than just one requirement for the user story to be fulfilled. So different engineers could make children tickets out of that original ticket and once all the children tickets were completed, then the original ticket could move to the next stage.

From a week to week standpoint, there is now also a Weekly Planning Session each Monday as an extension to the daily standups where the team discusses what is hoped to be implemented by the end of the week. These meetings tend to take up an hour now due to

the initial team of four increasing to seven due to the start of another project and a new approach the head of project delivery proposed. This new approach would separate the entire team in to two teams: front-end and back-end developers. The front-end and back-end developers would go between projects depending on what tasks on each project took priority. This dissertation will only focus on the work carried out in the initial project though.

Once there are enough features that have been implemented over a period of time (decided by the project manager and the senior software engineer, the system will be then deployed for testing purposes and at that stage, our QA tester thoroughly tests all the features of the application and any bugs that are caught are posted as a ticket onto Jira.

Moving on to GitLab, is the version control repository. As part of this project the workflow was to branch off from master and if it was a bug the branch was called "bug/nameOfBugHere-JiraTicketNumber" or if it was a feature then it was called "feature/nameOfFeature-JiraTicketNumber". That bug was then fixed or that feature was then implemented.

While that was being done, changes are staged, committed and then pushed onto Gitlab. Any changes were always pushed at the end of the day to ensure that team members could always access the most recent changes made to that in case the person who implemented the changes were off the next day.

Once the engineer was satisfied that their code does what it said, a merge request was then created where a description of what the branch implemented or fixed was included in the merge request and a maintainer was assigned to be responsible for checking it over and merging it. The person who opened the merge request was also responsible for tagging other team members in the description so they were made aware of the merge request.

Once the merge request was open, it was time for the other team members to look over the changes in the code and also check it out on their local system to test that it worked as expected and to catch any bugs. If any bugs were caught, a discussion was brought up on the merge request which would be used to highlight a piece of code they wanted to discuss or point out an issue with it.

It was then up to the team member that opened that discussion to have resolved that discussion once the person responsible for the code resolved the issue that was brought up. Once discussions had been resolved and everyone was satisfied with the code, the merge request could then be merged with the master branch.

After the merge request had been merged, everyone then needed to pull from their master branch to ensure their master branch code was up to date. They also checked out the branches they were working on and pulled the new updates from the recently updated master too. Sometimes conflicts occurred and these needed to be carefully resolved.

Besides this, my supervisor and I tried to skype once a week, during my lunch where I let him know how I was progressing with the application as well as discussing matters such as copyright and privacy as since the company owns the software and the clients have reasons to remain anonymous, I am not at liberty to share details of the business the CRM is for.

4.2 SYSTEM STRUCTURE:

When the client came to Pulsion Technology looking for a CRM system, they already had decided they wanted the medium for the system to be a web application so that while they could access it through a desktop, the application would also be mobile friendly so if they were working out in the field, they could still access it through an Ipad.

All data records such as Organisation records, Customer Records and User records were stored in a dynamoDB database hosted by Amazon Web Service where each developer had set up their own development environment with their own databases so that our systems were completely seperate. The way the CRM reached the database as well as the other services such as Cognito was through an API that went through AWS's API gateway and then through AWS's lambda functions which executes code written by the back-end team which then makes use of microservices such as DynamoDB or Cognito.

The way the application works is that a user needs to be created on the system by an admin user, that user can then login with their given username and password (this will change once it leaves development). Once the user logins, they land on the dashboard called "My work" which is currently an empty page still to be developed.

However on the left of a screen is a collapsible sidebar that has various routes that the user can go in. If Organisation's is clicked, the user is then directed to the Organisation's list page which contains a table with all the organisation's the user has created along with columns showing details of that Organisation.

Depending on what user role the user had been given, the user may not actually have permission to see the Organisation's page though and if the user did have permission to see the Organisation's page, the user may not be able to create a new organisation or modify an existing organisation (the buttons are hidden). The way a user can create an organisation though is through clicking the "New organisation" button which presents the user with a form that the user needs to fill out, and if the user forgets to fill in a field or doesn't fill it in correctly, validation messages attached to the bottom of the field inputs would appear.

Once a form has been successfully filled in, a modal appears to confirm the creation of a new organisation with them details. In the case that the user does confirm the creation, the user is then led back to the list page with the new organisation now being present in the table. The user can then click this organisation entry in the table and this will lead the user to the view page of that organisation.

On this page, if the user has the permission to do so, they can click the edit button and this allows the user to edit any of the fields If they user didn't have permission though, based on their user role, the edit button would be hidden. In the future, users will need to send approval requests to modify certain fields.

Once the user has edited fields though (given they are valid changes) the user can then submit the changes which will again open a modal up for them to save or cancel the changes. If the

user saves the changes, the user stays on the organisation page, however if the user wanted to go back to the list page, there is a button at the top that the user can click to direct them back.

4.3 Final Design Outcome:

As Organisation's progressed, insight into the Organisation's user story grew, when the form was initially completed, there was no field to record a "Standard Industrial Classification" (SIC) code, there was no field to record the local authority area as well as the only being able to have one contact for each Organisation with their own name, email address and phone number, however the Organisation's form now has the ability to record a SIC code, record a local authority area as well as having the ability to record multiple contacts for each Organisation.

4.4 UI Design:

Through weeks of communication between the clients and the graphic designer, this is the general layout and appearance the clients wanted. The general layout of the Organisation's was very much similar to all the other pages, the only difference is that the Organisation's form was more complex as it had the ability to add multiple contacts.

In the first picture, the Organisation's dashboard is shown.

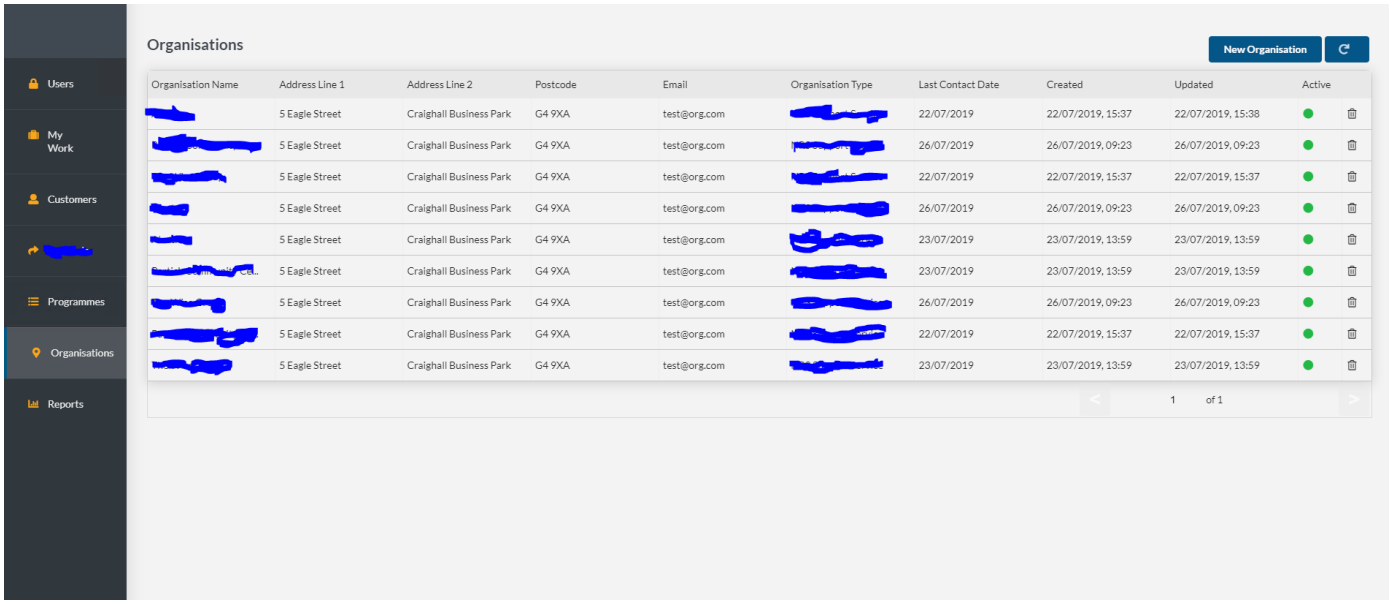

Figure 2 Organisations Dashboard

When the “New Organisation” button is clicked in the right-hand corner, then the Create page is shown in figure 2.

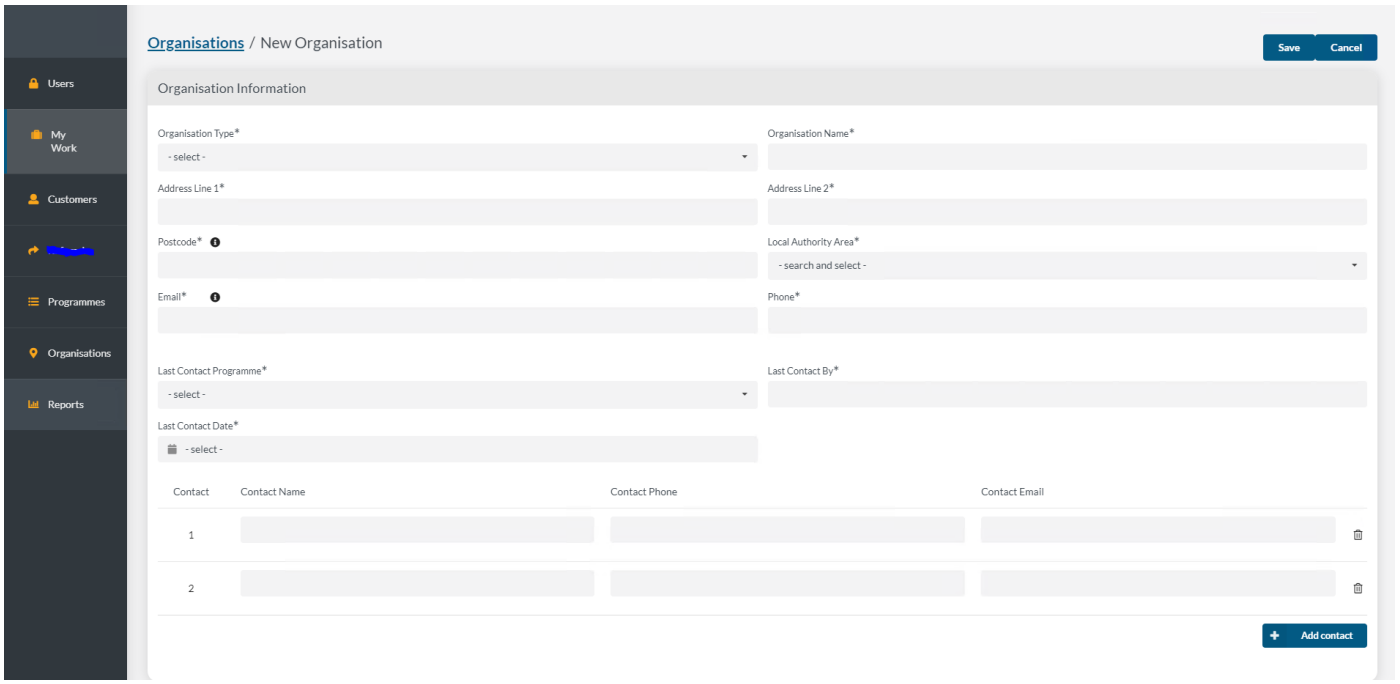

Figure 3 Organisations Create Page

If instead, one of the organisations are clicked inside the table, then the View page is shown like In figure 3.

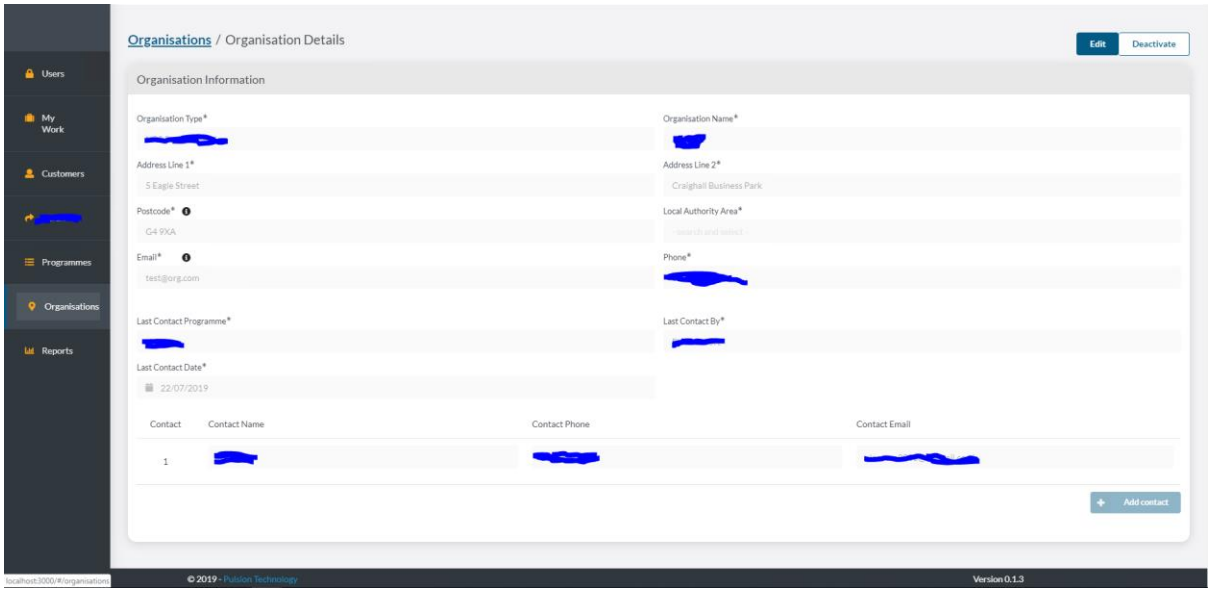

Figure 4 Organisation's View Page

Then if the edit button is clicked, the form fields now allow you to edit them.

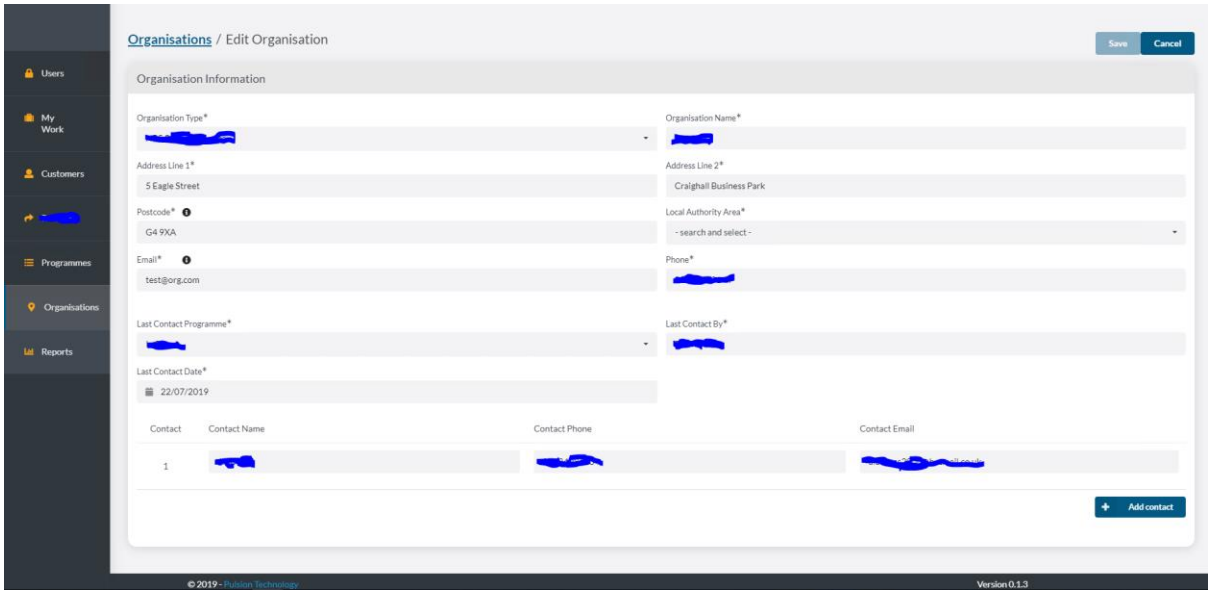

Figure 5 Organisation's Edit Page

If the user goes to submit the filling in all the fields, you will get validation error messages as shown below.

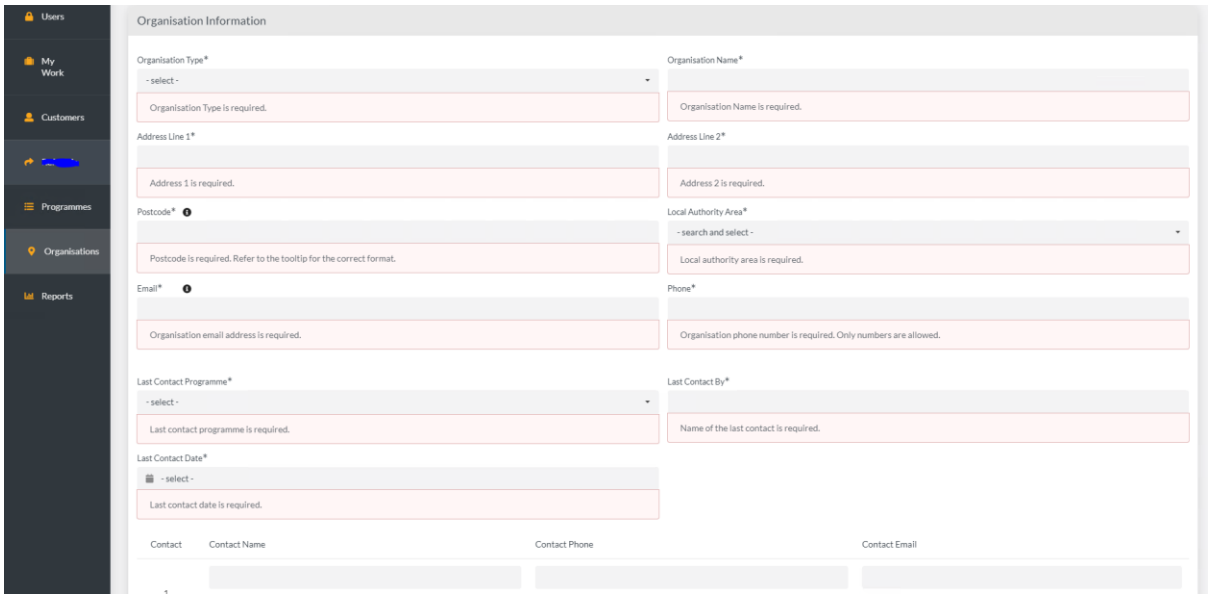

Figure 6 Validation messages when all fields are empty

And even if fields are filled in, for the likes of the email and postcode, there is regex checks to ensure it is a valid email address or a valid UK postcode.

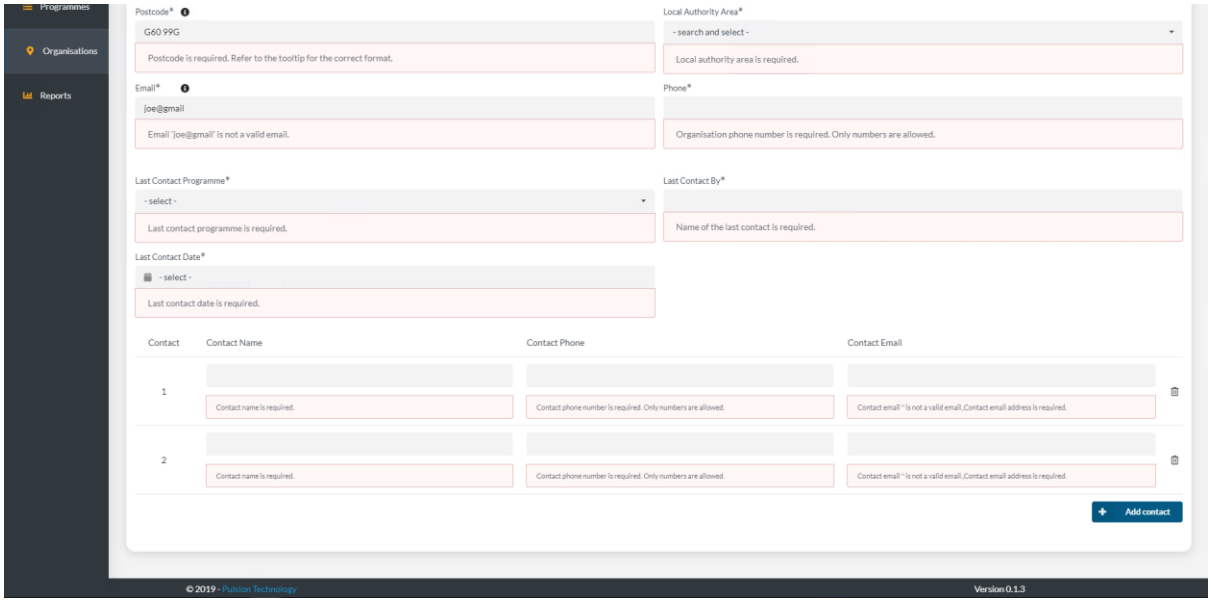

Figure 7 Postcode and Email validation messages when value is invalid.

5 Detailed Design and Implementation

This chapter will focus on the detailed design and implementation of the development of the Organisation page along with any other work that was carried out relating to the application. This will include the documentation of all the technologies utilized and learned in as well as the main features of each page of the Organisation's and how they were implemented along with the issues that were encountered in developing them.

5.1 Learning:

To get started, these courses were used to learn about AWS, Serverless, HTML, CSS, Javascript and React:

1. Codecademy[3] Introduction to HTML: This was a short course that introduced the basics of HyperText Markup Language (HTML) and how to start building and editing web pages.
2. Codecademy Introduction to CSS: This was a short course that introduced how HTML could be organized and styled with Cascading Style Sheets (CSS).
3. Codecademy Introduction to Javascript: This was a longer course that taught the fundamentals of Javascript which also included the newest features of Javascript ES6.
4. Codecademy Learn ReactJS: Part 1: This course introduced JSX which is a syntax extension for Javascript. It was written to be used with React and it looks a lot like HTML. The course also introduced the idea of components as well as props which is a

[3] https://www.codecademy.com/learn

way of passing information from one component to another and state which is dynamic information stored by a component.

5. Codecademy Learn ReactJS Part 2: This course introduced the relationship between stateless and stateful components, styling of the components and component's lifecycle methods.
6. Official Reactjs tutorial[4]: In this tutorial, the game of tic-tac-toe was programmed.
7. Udemy.com: React – The Complete Guide (incl Hooks, React Router, Redux)[5]: This course/tutorial also went over the fundamentals in a much thorough way, it went through different ways of doing things and at the same time walked through the development of a "Burger Building App".
8. TylerMcginnis.com[6]: This website focusses on modern Javascript and React, where things are updated so that nothing is ever 6 months out of date. This course really helped explain a lot of the fundamentals of React and how React apps should be built.
9. AWS WildRydes[7]: This course introduced AWS's microservices such as AWS Lambda, Amazon API Gateway, Amazon S3, Amazon DynamoDB and Amazon Cognito. It introduced these services by showing how to create a simple serverless web application that enables users to request unicorn rides from the "Wild Rydes" fleet. The application presents users with an HTML based user interface (Amazon S3) for indicating the location where they would like to be picked up. The application interfaces on the backend with a RESTful web service (Amazon API Gateway, AWS Lambda, DynamoDB) to submit the request and dispatch a nearby unicorn. The application also handles users registering with the service and requiring users to log in (Amazon Cognito) before requesting unicorn rides.

5.2 Development Technologies Used:

5.2.1 Javascript:

Javascript[8] is a high-level, interpreted language programming language that conforms to the ECMAScript specification[9]. Javascript has curly-bracket syntax, dynamic typing, prototype-based object-orientation and first-class functions and was used throughout the entire front-end with the help of React to provide the interactivity of the CRM web application as well as the back-end in the form of Node.js.

Serverless:

[4] https://reactjs.org/tutorial/tutorial.html
[5] https://www.udemy.com/react-the-complete-guide-incl-redux/
[6] https://tylermcginnis.com/
[7] https://aws.amazon.com/getting-started/projects/build-serverless-web-app-lambda-apigateway-s3-dynamodb-cognito/
[8] https://www.javascript.com/
[9] https://www.ecma-international.org/publications/standards/Ecma-262.htm

The Serverless Framework[10] is a free and open-source web framework written using Node.js. It was used in building the web application due to the fact that our web applications back-end runs on Amazon Web Services (AWS). This was used to deploy back-end code to AWS Lambda.

5.2.2 HTML/JSX:

Hypertext Markup Language[11] is the standard markup language that dictates how a web page should be displayed in a browser as well as telling what elements should be loaded. Apart from being used in the index page, HTML wasn't directly used. Javascript has syntax extension called JSX which looks like HTML but isn't HTML. It can be seen from the screenshot below in figure 10 in subchapter 5.3.1 where there is html looking syntax: <Form.Dropdown attributes=…/>.

5.2.3 CSS:

Cascading Style Sheets (CSS)[12] is the standard way of styling HTML. It modifies the way in which HTML is displayed on a page. For example, it can decide the location of where HTML is displayed, whether it is hidden or not, the colour of it, the size and much more. A lot of CSS was done in the forms of classes, where a class of CSS styles were associated with a certain component and its sub components. There was a template that most engineers took from though and if needed, it would be tweaked to suit a page but most of it was kept the same to keep the look of the pages consistent.

5.2.4 SCSS:

Sass[13] (SCSS) is an extension to CSS which increases the features and abilities of CSS and is approved by industry. Sass was what was used when it came to making the styles like mentioned for CSS.

5.2.5 React:

React[14] is a declarative, efficient and flexible JavaScript library that's used for building user interfaces. The main philosophy of it is that complex UIs can be created from using small and isolated blocks of code called "Components" that can interact with one another by passing information.

5.2.6 React Router:

React Router[15] is the standard routing library that is used for React. It was used to help keep the UI in sync with the URL's that were used to ensure the correct components were being rendered on the correct URL pages as well as ensuring that if a link button was clicked to another page, that the other page would be landed on.

[10] https://serverless.com/
[11] https://html.com/
[12] https://en.wikipedia.org/wiki/Cascading_Style_Sheets
[13] https://sass-lang.com/
[14] https://reactjs.org/
[15] https://reacttraining.com/react-router/

5.2.7 React Redux:

Redux[16] is mostly used for complex web applications, where passing state information in the form of props, from one component to another is not very elegant. For example, if a component needs to access some information based on another component but that other component is maybe 8 components away, it is exhaustive passing props through all of them components that are inbetween. What Redux does is create a container for this state that can accessed directly from any component. Since the CRM was a large web app, it made sense to make use of Redux however, it is unnecessary when it comes to simple actions like UI changes which the local state of a component can handle fine.

5.2.8 Axios:

Axios[17] is yet another Javascript library that was used in the development of the web application. This library helps make HTTP requests from node.js which is used in the back-end code as well as supporting the Promise API that's part of Javascript ES6. In the context of this project, Redux actions would call functions from the API class, and these functions would use axios to make requests to the AWS API Gateway which would then send a response back which axios would handle. There was four different requests, "post" which is used for when something is to be created, "put" can also be used to create but also update, "get" is used when information is needed to be retrieved, and "delete" is for when it's desired to delete information. Depending on how things are set up though, Post could even be used to retrieve information as shown in figure 27 in the verification and validation chapter.

5.2.9 Semantic UI:

A design decision made from the start was to use a UI package called Semantic UI[18] that has its own Elements (button, icon, input etc.), Collections (Form, Grid and Message etc.), Views (Comments, Feed and Card), Modules (Dropdown, Modal, Search etc.) and more. This saved a lot of time by not having to create UI components from scratch. The dropdown shown in a later screenshot is an example of this.

5.2.10 Validate.js package:

Validate.js[19] is a javascript package used for validating Javascript objects. The main benefit of this package is that it provides a cross framework and cross language way of validating data and so the validation constraints by which objects are validated on, can be shared between clients and the server. It was used to validate all of the fields of an Organisation as well as for Users and Customers etc.

5.2.11 Node:

Node.js[20] is an open source server environment that allows javascript to run outside a browser environment. Node.js was needed from the beginning to use NPM and set up the

[16] https://react-redux.js.org/
[17] https://github.com/axios/axios
[18] https://semantic-ui.com/
[19] https://validatejs.org/
[20] https://nodejs.org/en/

application on the system. This was used in the back-end to create functions to handle requests from axios.

5.2.12 NPM:

NPM[21] is an online package manager for JavaScript and is the default package manager for Node.js hence why Node.js was required. NPM is what allowed for libraries like React and packages like React-Router, Axios and Semantic UI to be installed and used as part of the web application.

5.2.13 Webpack:

Webpack[22] is a module bundler. Its purpose is to bundle JavaScript files for usage in a browser but it also supports many other assets such as CSS and images. This was used to help build the application using the package.json file which let's webpack know which packages/dependencies the application requires to be installed and what version. A small part of it is shown below:

[21] https://www.npmjs.com/
[22] https://webpack.js.org/

Figure 8:Excerpt from package.json file

Each line in the above figure represents a package that webpack will read and then fetch from the web. Not only that but with “scripts” as shown, webpack can enable a project to be ran, built, tested and ejected. If a project is ran, it just means it runs on a browser, this happens during development. Once it is production ready, it is then built, when built, not all packages are needed, when tested,

test packages will run and when the project has been ejected, all of these packages can be seen, which has already been done.

5.2.14 Git:

Git[23] is a distributed version-control system for tracking changes in source code during software development. It is designed for coordinating work among engineers, but it can only be used to track any kinds of documents. This was used extensively throughout development.

5.2.15 Gitlab:

Gitlab[24] is a web-based DevOps lifecycle tool that provides a Git-repository manager providing wiki, issue-tracking and continuous integration/continuous delivery pipeline features. The project's source code was stored on a repository on this website.

5.2.16 Postman:

Postman[25] allowed for sending API requests to AWS and receiving responses without having to use the front-end of the application. This came in useful when testing the back-end and couldn't rely on the front-end.

5.2.17 Visual Studio Code:

Visual Studio Code[26] is a source-code editor developed by Microsoft. It includes support for debugging, embedded Git controls and GitLab, syntax highlighting, intelligent code completion, snippets and code refactoring. It also features excellent extensions such as Prettier which formats your code so that it is more readable as well as an extension called GitLens which shows who has coded the selected line as well as when.

5.2.18 Microsoft Teams:

Microsoft Teams[27] is a collaboration tool used for office communication and this is used throughout the development of the project so that developers can post questions, see that merge requests have been opened as well as merged and also see updates others have posted on projects.

5.2.19 Jira:

Jira[28] helps the team plan by allowing developers and others to create user stories and issues, as well as plan sprints and distribute tasks across the software development team. Not only that, but it helps developers and others track what is going on as well as prioritize and discuss the team's work in full context with everything else visible. As mentioned in the methodology, Jira was used extensively to track the progress of the project.

[23] https://git-scm.com/
[24] https://about.gitlab.com/
[25] https://www.getpostman.com/
[26] https://code.visualstudio.com/
[27] https://products.office.com/en-us/microsoft-teams/group-chat-software
[28] https://www.atlassian.com/software/jira

5.2.20 Aceproject:

Aceproject[29] is a web-based application that Pulsion Technology uses to track time spent on projects and User Stories within projects and at the end of the day, the time spent on a user story was recorded on to the corresponding task on AceProject.

5.2.21 Google Chrome Developer Tools:

The developer tools[30] found in Google Chrome are very powerful inspection tools for observing many parts of a web application. It provides a very useful way for identifying the document object model (DOM), the firing of events, console logs, monitoring of network activity as well as having the ability to be extended so that it is also possible to analyse the web application's Redux store as well as looking at the local state and props of components in React. Not only that but it also provides the ability to inspect the CSS of elements as well as the ability to temporarily modify the CSS so that it doesn't need to be changed in the code editor and redeployed.

5.2.22 Amazon Web Service:

Amazon Web Service[31] (AWS) is a cloud platform that can provide 165 different services [ref] from data centres across the glove. Cloud computing provides on-demand delivery of computing power, databases, storage, applications as well as many other features. AWS is what is used to provide the storage, databases, back-end functionality and security for the CRM.

5.2.23 Amazon API Gateway:

Amazon API Gateway[32] is a fully managed service that made it easy for API's to be created, published, maintained, monitored and secured. It allowed for the application to access data as well as business logic which was code running on AWS Lambda. This was what axios interacted with.

5.2.24 Amazon DynamoDB:

Amazon DynamoDB[33] is a fully managed proprietary NoSQL database service that supported all the data stored in the CRM web application. It stores Organisation's as well as other entities in a document format. For example each Organisation is stored in the form of an object data-structure as part of a table of Organisations.

[29] https://www.aceproject.com/
[30] https://developers.google.com/web/tools/chrome-devtools/
[31] https://aws.amazon.com/
[32] https://aws.amazon.com/api-gateway/
[33] https://aws.amazon.com/dynamodb/

5.2.25 Amazon Cognito:

Amazon Cognito[34] is a service that handles user authentication and access for mobile applications on internet-connected devices. It allowed for the web application to easily have user sign-up, sign-in and access control and was what handled users for the CRM.

5.2.26 AWS Lambda:

AWS Lambda[35] let the back-end of the web application to be ran without provisioning or managing servers. It is very cost effective way of hosting a web application as only the compute time consumed needs to be paid for and can be scaled without having to upgrade an existing server. For this project, there was back-end code that was stored on the computer however, to get that code running on AWS Lambda, Serverless would deploy the code on to AWS Lambda for AWS Lambda to then be able to execute that code when the application was running.

5.3 Organisations User Story Features:

This chapter will delve into each area of what was done within the Organisation's user story where each part will be discussed as to how it was developed, how it functions and explains the approach taken.

5.3.1 Initial Setup of Organisation's List, Edit and Create Pages:

At this point, the main application had already been set up with Redux, Router and the User's page had already had enough functionality to create, edit, view and delete Users. There was also a sidebar that showed a list of routes the user could click on such as "Customers" and "Organisations". These pages could be clicked on however they only led to an empty page. What was done then for Organisation's was all the code that was currently in User's was copied and pasted over into the Organisation's folder within the project's folder directory.

The first time this process was carried out, it was done in a way that each subfolder was copied and pasted one at a time and then refactored in such a way that all occurrences of the word "User" was replaced with the word "Organisation" or "users" with "organisations" as well as other variations. Once this was completed, the application was still not working and it was decided that it would be easier to start again and so this time the whole Users folder was copied as a whole and pasted inside the Organisation's folder and then Visual Studio Code's powerful search functionality was used to scan the files for occurrences and this time, the refactoring had been done properly and the Organisation page now had the same functionality as the User's page had. What had to be done now was adapt the User's page to what was required of Organisation's as dictated in the User Stories.

[34] https://aws.amazon.com/cognito/
[35] https://aws.amazon.com/lambda/

One mistake that was made near the start of the project was that the Active property was removed from the Organisation's because it was assumed that an Organisation doesn't need an active property like a User does. However this was a wrong assumption to make and the Active property was brought back for Organisation's.

At this stage in the development of the web application, despite having went through various tutorials, it was still very hard to navigate and understand what was doing what. The structure of the application was very large and complex compared to all the projects that had been seen before and at the same time, there was also a lot of code repeated throughout the source folder too.

The application also handled Router and Redux very differently from what had been seen before too and it was very difficult to make sense of it all. To help explain why this is, the project structure has been displayed in the appendix. At this point though, what had to be done was to change the form within the Create Organisation and Edit Organisation page so that it would record Organisation data rather than User data and that's what was focussed on to start with.

So at the start, instead of the form having a dropdown component for Roles, there was now a dropdown component for Organisation Types which required creating a dropdown options file within Resources and then creating an array like shown below:

```
export const organisationTypeOptions = [
  { key: 1, text: "S SS", value: "sSS" },
  { key: 2, text: "NE SS", value: "neSS" },
  { key: 3, text: "S Employer", value: "sE" },
  { key: 4, text: "NE Employer", value: "neE" },
  { key: 5, text: "TtG SS", value: "ttgSS" }
];
```

Figure 9: An array of options of what type of an organisation, an organisation can be

It's helpful to note that the curly brace "{" is the beginning of an object and "}" is the end of an object. It's also helpful to note that "[" and "]" is also the beginning and the end of an array, so it is then clear that the above array, has multiple objects in it.

The key in each object would be used as a way to uniquely identify each object in the array, the "value" would be what value would be recorded in the database if that row was selected and the text is for display purposes in the front-end as will be shown in figure 11 below where a dropdown is shown, displaying the text options displayed in figure 9. If "S SS" was selected, then the value "sSS" would be stored as what type of Organisation it is.

This array would then be imported into the ModifyOrganisation file and then referenced as the options for the Form.Dropdown component that is part of Semantic UI. The reason for the check at the start to check whether the mode is create or edit is because if it's in view mode, the dropdown should be disabled however for some reason the dropdown component doesn't disable so when view mode is on, the component is actually rendered as a Form.Input, another component from Semantic UI which is just an ordinary text input.

```
{this.props.mode === "create" || this.props.mode === "edit" ? (
              <Form.Dropdown
                label={strings.form.label.mainType}
                name="mainType"
                onChange={this.props.handleChange}
                placeholder={strings.form.placeholder.mainType}
                selection
                required
                options={organisationTypeOptions}
                value={this.props.selectedOrganisationData.mainType || ""}
                width={8}
              />
            ) : (
              <Form.Input
                label={strings.form.label.mainType}
                name="mainType"
                placeholder={strings.form.placeholder.mainType}
                required
                value={
                  this.props.getOrganisationTypeNameFromValue(
                    this.props.selectedOrganisationData.mainType
                  ) || ""
                }
                width={8}
              />
            )}
```

Figure 10: Form Dropdown component being rendered if in create or edit mode and a Form Input being rendered if not in them modes.

In the above screenshot, there are properties given to both the <Form.Dropdown /> and <Form.Input /> component. These properties are “label”, “name”, “onChange”, “placeholder”, “selection”, “required”, “options”, “value” and “width”.

The “label” property is simply the label given to the Dropdown. In this case the text that has been passed into the “label” property is “Organisation Type”, as shown in the label in figure 11. Next is “name” and that helps identify the component. The “onChange” property passes in a function that allows data to be changed in the dropdown. The “placeholder” property simply displays text in the dropdown when no value has been selected. The “required” property enforces that the form can’t be submitted unless it has been filled. The “value” property is the value that is stored in the dropdown. The “width” property simply tells the Dropdown or Form.Input component how wide it should be.

With this code and “options” in dropdown being set to equal the array from the previous screenshot, there is now a dropdown field displaying in the form as shown below:

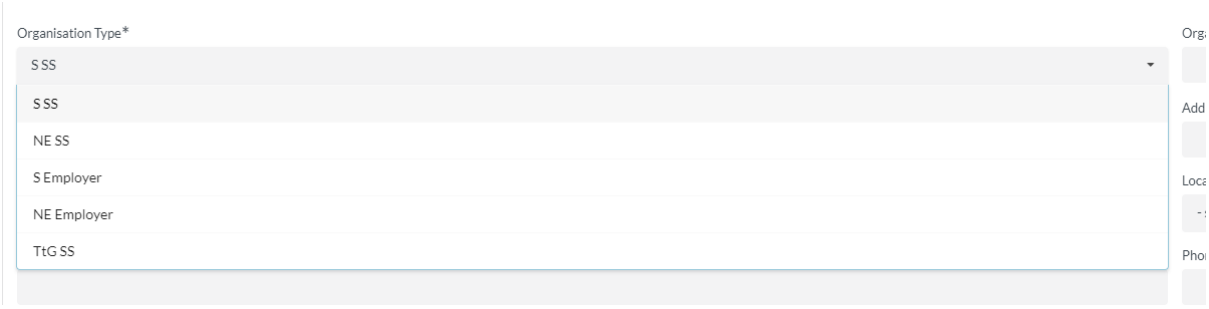

Figure 11: Dropdown shown in front-end

This part of the project was simple enough to do without having to understand too much of the larger picture.

Another simple part of the project was opening up the strings folder where strings were imported into other folders from and changing the strings to suit the Organisations page. To give a better idea of what this is, here is a screenshot of a part of it:

```
import LocalizedStrings from "react-localization";

export const strings = new LocalizedStrings({
  en: {
    header: {
      login: "Login",
      organisationList: "Organisations",
      createOrganisation: "New Organisation",
      editOrganisation: "Edit Organisation",
      viewOrganisation: "Organisation Details",
      loading: "Loading Information"
    },
    form: {
      header: {
        organisationInfo: "Organisation Information",
        contactInfo: "Contact Information",
        organisationsLink: "Organisations",
        clientsLink: "Clients",
        supportLink: "Support",
        settingsLink: "Settings"
      },
      label: {
        orgName: "Organisation Name",
        address1: "Address Line 1",
        address2: "Address Line 2",
        postcode: "Postcode",
        phone: "Phone",
        email: "Email",
        contactName: "Contact Name",
        contactPhone: "Contact Phone",
        contactEmail: "Contact Email",
        mainType: "Organisation Type",
        lastContactDate: "Last Contact Date",
        lastContactProgramme: "Last Contact Programme",
        lastContactBy: "Last Contact By",
        sicCode: "SIC Code",
        localAuthorityArea: "Local Authority Area"
```

Figure 12: File with list of strings to be used by other files.

Another change that was made that had to be discarded was a change in how the form was structured. An assumption was made that the form should be more spread out so that certain pieces of information such as address information was grouped together while other

information was separate however this change was not liked due to the fact it made the screen scrollable and so some fields were not visible unless the page scrolled down.

5.3.2 DatePicker:

One headache that was had was the DatePicker[36]. It was assumed that implementing a simple component that let a date be selected would be relatively easy but it wasn't. One issue was that when selecting a day of the calendar, only 12 days were available to click as for some reason it was taking that as the month. The datepicker documentation was scrutinised and various solutions were tried in the front-end to solve this issue as another problem the datepicker had when trying to correct it was that it would randomly break the application. Due to it being random, the datepicker was thought to be working until later on.

5.3.3 RequestFeedback:

There was also plenty of times during the project where if something was changed in the Users or Customers pages, the changes had to be reflected in Organisation's as well. One of these changes that occurred was the feedback messages that would be shown. Initially the application had feedback messages that were only based on a Boolean value of false or true however due to an issue with the wrong feedback message being sent in some edge cases, the value of "undefined" was also required to be used for the variable "requestStatus" when the CRM was processing a request. In addition to these changes, variable names and function names were made more consistent and meaningful as well as styling changes and

[36] https://www.npmjs.com/package/semantic-ui-calendar-react

making RequestFeedback its own component as shown here:

```
import React from "react";
import { Grid, Message } from "semantic-ui-react";
import PropTypes from "prop-types";

class RequestFeedback extends React.Component {
  render = () => {
    return (
      <Grid className="requestFeedback" textAlign="right">
        {this.props.requestMade &&
          this.props.requestStatus === undefined &&
          !this.props.unknownRequestStatus && (
            <Message
              className="processingRequest"
              content={this.props.processingFeedbackMessage}
              header={this.props.statusFeedbackMessage}
            />
          )}
        {this.props.requestMade &&
          this.props.requestStatus &&
          !this.props.unknownRequestStatus && (
            <Message
              className="requestSuccess"
              success
              header={this.props.successFeedbackMessage}
              content={
                this.props.successMessage +
                " " +
                (this.props.optionalRefreshMessage || "")
              }
            />
          )}
        {this.props.requestMade &&
          this.props.requestStatus === false &&
          !this.props.unknownRequestStatus && (
            <Message
              className="requestFailure"
              error
              header={this.props.failureMessage}
              list={this.props.errorDetails
                ? this.props.errorDetails.map(error => {
                  return error;
                })
                : []
              }
            />
          )}
        {this.props.unknownRequestStatus &&
          this.props.requestStatus === undefined && (
            <Message
              className="requestUnknown"
              header={this.props.statusFeedbackMessage}
              content={this.props.unknownFeedbackMessage}
            />
          )}
      </Grid>
    );
  };
}
```

Figure 13: RequestFeedback component

These were reasonably simple changes to make however with the complexity of the application itself, it was easy to make mistakes and then it would require some amount of time to try and find and fix the issue.

One way of helping find mistakes was through the use of PropTypes:

```
RequestFeedback.propTypes = {
  requestStatus: PropTypes.bool,
  requestMade: PropTypes.bool.isRequired,
  unknownRequestStatus: PropTypes.bool.isRequired,
  successMessage: PropTypes.string.isRequired,
  failureMessage: PropTypes.string.isRequired,
  processingFeedbackMessage: PropTypes.string.isRequired,
  unknownFeedbackMessage: PropTypes.string.isRequired,
  statusFeedbackMessage: PropTypes.string.isRequired,
  successFeedbackMessage: PropTypes.string.isRequired,
  optionalRefreshMessage: PropTypes.string
};
```

Figure 14: Proptypes

PropTypes declares what props, the RequestFeedback component is to expect and so if for example no prop called “requestMade” was passed in to the RequestFeedback component, a console log warning would appear on chrome dev tools like this:

Figure 15: Proptypes console warning

When initially adding validation, the validation was done entirely within the OrganisationCreateContainer and OrganisationEditContainer, so this involved having a list of requiredFields as well as adding objects called policies that would be used for regex checking against values to ensure they were in the correct format.

5.3.4 Validation and DatePicker:

With the introduction of validation to the forms came another issue with the datepicker. Each form field was to raise a validation message if left empty when the save button was clicked and until these fields were no longer empty, it wouldn’t be possible to save. This was because there was an additional on-click function that validated the form based on logic that had been added to validate fields. For some reason though, if the datepicker was never clicked on, it wasn’t being seen as part of the form so when the form was sent, the API rejected the form due to the lack of that piece of information.

One solution was to revert to using Form.Input and setting it to type “date” which made use of HTML’s own datepicker however by looking at the website “https://caniuse.com/” it was found that Safari IOS as well as IE had bad support for it and so the original datepicker was reverted back to. It was eventually found that the value attribute should contain the string representation of the date instead of the date object that was being returned by moment.[37]

[37] https://www.npmjs.com/package/moment

Overall it took a while for this issue to be resolved however it had been put on standby while other tasks were carried out.

Another thing that was required was the adding of an html novalidate attribute that was needed for the organisation form itself due to the validation being carried out by javascript rather than html. This was done through the ability to read the DOM by carrying out document.getElementById(“organisationForm”) where “organisationForm” was the value given to the id attribute of the form.

5.3.5 Validate.js:

It was around this time that a meeting was had with the head of project delivery that a package that could be shared between the back-end and the front-end was to be desired for carrying out validation as having it in two separate places increased complexity and caused issues. This resulted in looking for a third party library that had a standardised approach to validation and a way to do this was through a package that the owner of Pulsion Technology had done some research into and this package was called validate.js.

This package allowed for a “constraints” JSON object that could then be shared by both the back-end and the front-end if wrapped inside an NPM package that was published by Pulsion Technology and it meant as future projects were done, more constraint files could be added and some could be reused. This required the validation that had already been done to either be taken out or refactored to make use of this new package that was now being used. It took some time to understand the documentation of the package and how it worked so that it could be implemented but it was deemed an investment for the future. The constraints file for Organisations looked like this:

```
{
  "constraints": {
    "mainType": {
      "presence": { "allowEmpty": false }
    },
    "orgName": {
      "presence": { "allowEmpty": false }
    },
    "address1": {
      "presence": { "allowEmpty": false }
    },
    "address2": {
      "presence": { "allowEmpty": false }
    },
    "phone": {
      "presence": { "allowEmpty": false },
      "regex": {
        "pattern": "^[0-9]*$",
        "message": "Wrong phone number format, please just enter numbers."
      }
    },
    "email": {
      "email": true,
      "presence": { "allowEmpty": false }
    },
    "postcode": {
      "presence": { "allowEmpty": false },
      "regex": {
        "pattern": "^([A-Za-z][A-Ha-hJ-Yj-y]?[0-9][A-Za-z0-9]? ?[0-9][A-Za-z]{2}|[Gg][Ii][Rr] ?0[Aa]{2})$",
        "message": "Wrong postcode format, use the same format as 'G70 4DW'."
      }
    },
    "contactName": {
      "presence": { "allowEmpty": false }
    },
    "contactPhone": {
      "presence": { "allowEmpty": false },
      "regex": {
        "pattern": "^[0-9]*$",
        "message": "Wrong phone number format, please just enter numbers."
      }
    },
    "contactEmail": {
      "email": true,
      "presence": { "allowEmpty": false }
    },
    "lastContactProgramme": {
      "presence": { "allowEmpty": false }
    },
    "lastContactBy": {
      "presence": { "allowEmpty": false }
    },
    "lastContactDate": {
      "presence": { "allowEmpty": false }
    }
  }
}
```

Figure 16: Incomplete constraints object

Where within the constraints object, there was an object for each form field property and then within that object there was either a “presence” object which checked for whether a field was empty and if it was it would return a message or a “regex” object which checked whether a string matched a particular regular expression where a regular expression is a pattern that can be matched with. If a string didn’t match the regular expression (pattern) then a validation message would be shown.

With these changes for validation, Form.Input, Form.Dropdown and Form.DateInput were extracted out into standalone components called ValidatedFormInput, ValidatedDropdown and ValidatedDateInput so future objects could also use these components in the future.

After this some further refactoring was required in various places where the form had to be set to valid in the case that the changes made were cancelled. Furthermore, autocomplete was also set to off for the validatedDateInput as an autocomplete popup was covering the popup of the validatedDateInput, for some reason that can’t be recalled, this was done by

using getElementsByName which had be fixed to getElementById which made sense as it suited the way how the form had been read before to add the “novalidate” attribute:

```
componentDidMount = async () => {
  await this.props.getOrganisationById(
    this.props.match.params.id,
    this.props.headers
  );

  let form = document.getElementById("organisationForm");
  form.setAttribute("novalidate", true);
  let datepicker = document.getElementById("lastContactDate");
  datepicker.setAttribute("autocomplete", "off");
};
```

Figure 17: When page renders, form is read by "getElementById" so that attributes could be set directly to the DOM.

Another refactoring was also carried out within the reducer so that the feedback messages being displayed were given by an attribute called “Reason”:

```
  case organisationActionTypes.DELETE_ORGANISATION_SUCCESS:
    return {
      ...state,
      result: action.response.data.Reason,
      organisationRequestStatus: true,
      loadingPage: false
    };
```

Figure 18: A case statement inside the reducer file showing how a Reason would be returned as a result.

This was again to help make the code more consistent and understandable. These types of changes did not often take long to carry out and it didn’t take long to open a merge request and get it merged into the master branch.

Throughout these merge requests sometimes redundant code was spotted and commits were made to remove or improve on it. It helped to be a bit more careful before opening a merge request so that less of these discussions were made on the merge request so that less time was spent on focussing on getting rid of maybe a console.log or comment that was accidently left in rather than true problems with the code.

By this point of the implementation, a better understanding of the code with regards to the Containers and ModifyOrganisation component had been built up however it still wasn’t understood how Redux was working.

It was then discovered that to prevent a user from accessing an edit container that had already been deleted, the API had to be used in the container. The reason for this was that it would check whether the api call fails or not, if the call fails, a message would be shown on the list page and if otherwise you would navigate to the view page as normal.

Moving on from that, more work was assigned for finding out how to do validation messages properly because despite using the validate.js package, its ability to create messages were not being used and instead, the messages from the strings file was still being used. It wasn't too hard to make this change and it also allowed for dynamic messages to be shown where it could show the current value that was invalid. For example if the email "test@gmail" was submitted, the user would then be faced with a validation message which would say along the lines of "'test@gmail' is not a valid email. Please enter a valid email". These changes were made throughout not only Organisations, but also Customers and Users. All these changes were made in around 2 days. Here is the complete version of constraints:

```
{
  "constraints": {
    "mainType": {
      "presence": {
        "message": "^Organisation Type is required.",
        "allowEmpty": false
      }
    },
    "orgName": {
      "presence": {
        "message": "^Organisation Name is required.",
        "allowEmpty": false
      }
    },
    "address1": {
      "presence": {
        "message": "^Address 1 is required.",
        "allowEmpty": false
      }
    },
    "address2": {
      "presence": {
        "message": "^Address 2 is required.",
        "allowEmpty": false
      }
    },
    "phone": {
      "regex": {
        "pattern": "^[0-9]*$",
        "message": "^Contact phone number is required. Only numbers are allowed."
      }
    },
    "email": {
      "email": { "message": "'%{value}' is not a valid email." },
      "presence": {
        "message": "^Email address is required.",
        "allowEmpty": false
      }
    },
    "postcode": {
      "regex": {
        "pattern": "^([A-Za-z][A-Ha-hJ-Yj-y]?[0-9][A-Za-z0-9]? ?[0-9][A-Za-z]{2}|[Gg][Ii][Rr] ?0[Aa]{2})$",
        "message": "^Postcode is required. Refer to the tooltip for the correct format."
      }
    },
    "lastContactProgramme": {
      "presence": {
        "message": "^Last contact programme is required.",
        "allowEmpty": false
      }
    },
    "lastContactBy": {
      "presence": {
        "message": "^Name of the last contact is required.",
        "allowEmpty": false
      }
    },
    "lastContactDate": {
      "presence": {
        "message": "^Last contact date is required.",
        "allowEmpty": false
      }
    },
    "sicCode": {
      "regex": {
        "pattern": "^\\d{5}$",
        "message": "^SIC Code is required. Please ensure a value from the lookup is selected."
      }
    }
  }
```

Figure 19: Complete constraints file

It was at this point a tooltip was then needed for the Postcode field so that users were told what formats of postcode were acceptable.

5.3.6 Multiple Contact Table:

The next task was to finally go from an Organisation only having one contact who had a name, phone number and email address as shown below:

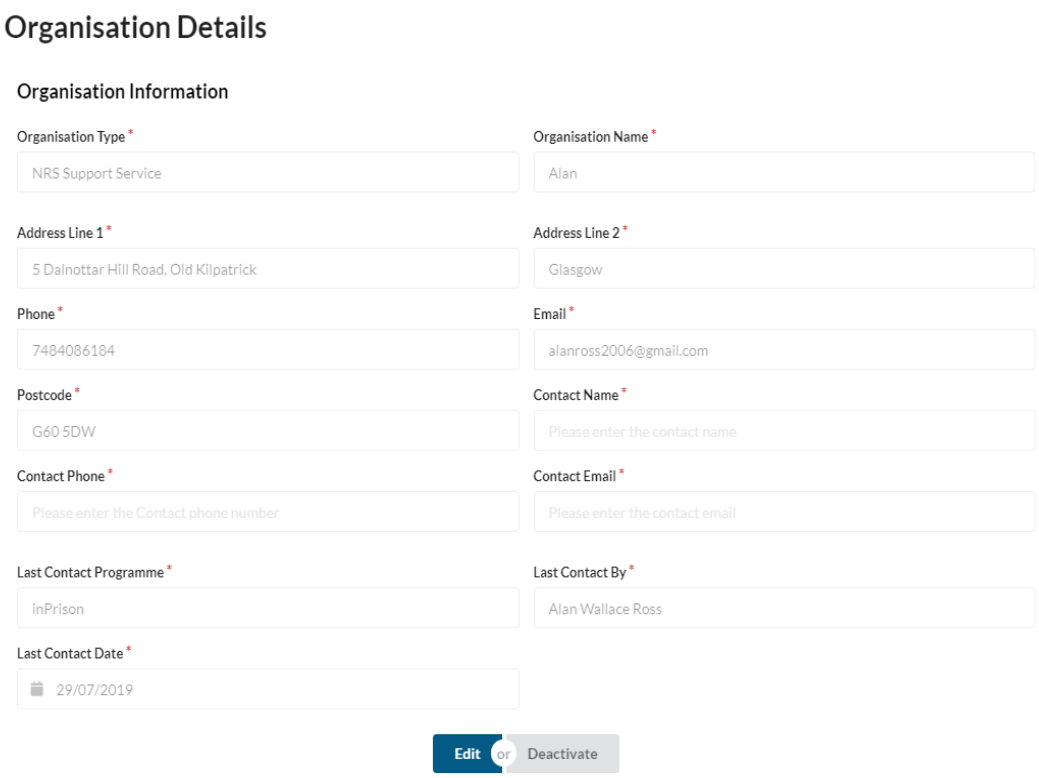

Figure 20: Initial single contact form

to an Organisation that could have multiple contacts (as many as desired) with these attributes:

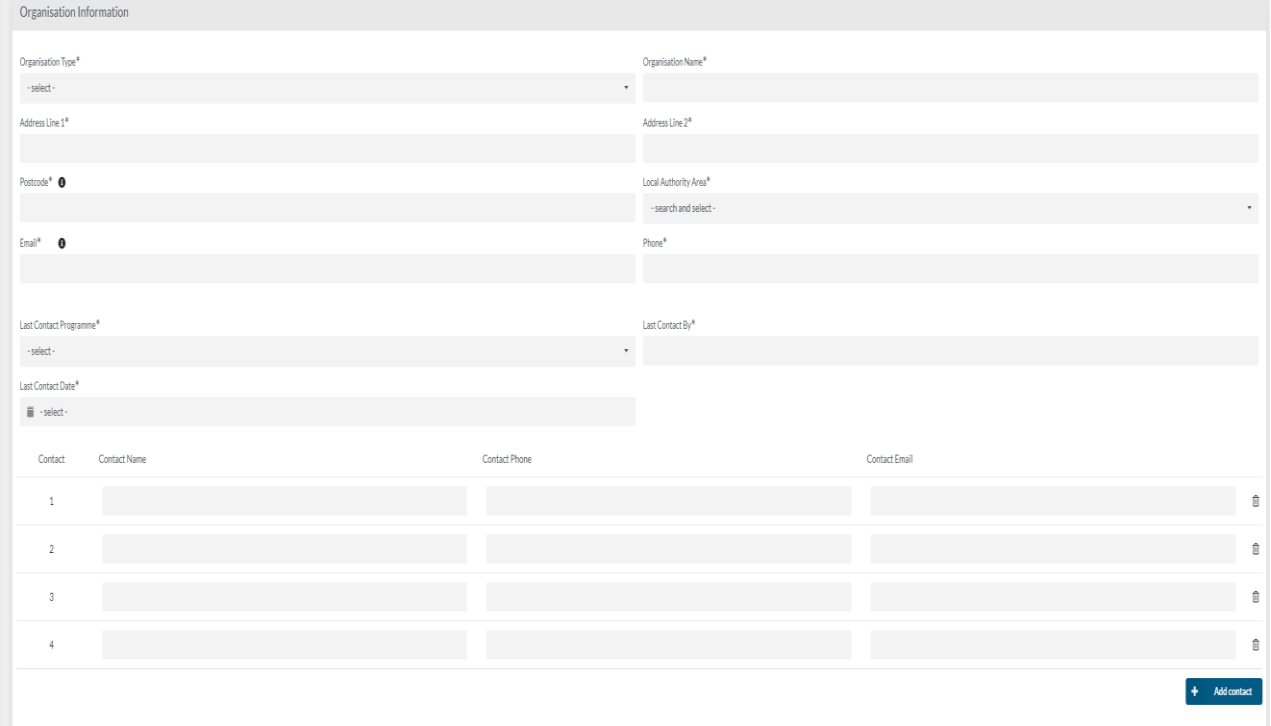

Figure 21: Final Organisation's form involving contacts. An enlarged version can found in the Appendix.

The initial step in implementing this was in actually rendering a list of three fields “name”, “phone” and “email”. The basic functionality was copied initially from the internet however with still a lack of understanding in how everything worked exactly as well as a lack of practice when it came to methods such as map and manipulating objects within an array that is within an object, there was still a long windy path in trying to get this to work where a lot of different ideas were tried and then discarded as well as time spent looking on google to try and find a solution to the problem as well as even looking at third party forms that might be able to handle this sort of complex form easily.

Initially it took a while to get the rendering to work but when that worked, it was then time to try and handle how the information that was entered in them rendered fields to be saved into the selectedOrganisationData object that contained the selected organisation that was clicked from the table or the organisation that was to be created.

The focus was to initially get it working for the Edit container first. Since the back-end didn’t support multiple contacts yet, the edit container had to invoke a function that would instead insert a placeholder object with the desired properties and structure that the multiple contacts could render.

The idea was to remove the properties “contactName”, “contactEmail” and “contactPhone” that was already there to support a single contact and instead have a property within the object called “contacts” which would be an array that would store objects that had the properties “contactName”, “contactEmail” and “contactPhone” so that if there was 2 contacts for that organisation, it would look like [{“contactName”:””, “contactPhone”:””, “contactEmail”:””}, {“contactName”:””, “contactPhone”:””, “contactEmail”:””}]. Before this was done though within the EditContainer, this was temporarily done within the

ModifyOrganisation component which is bad form since the pattern of containers and components within React is that containers handle state which I had within my component and also I had functions in my component that also should be passed down from the container. This was ofcourse fixed though so that all the local state (not redux state) as well as functions were stored within the container.

These functions were mostly “addContact”, “removeContact” and “handleContactChanges” which are quite self-explanatory. “addContact” simply pushed another contact object on to the array, “removeContact” removed the last contact to be added from the array and “handleContactChanges” was responsible for what was being entered in the fields to be reflected in the corresponding contacts and properties of them contacts, during the middle of the development of multiple contacts, the form looked like this:

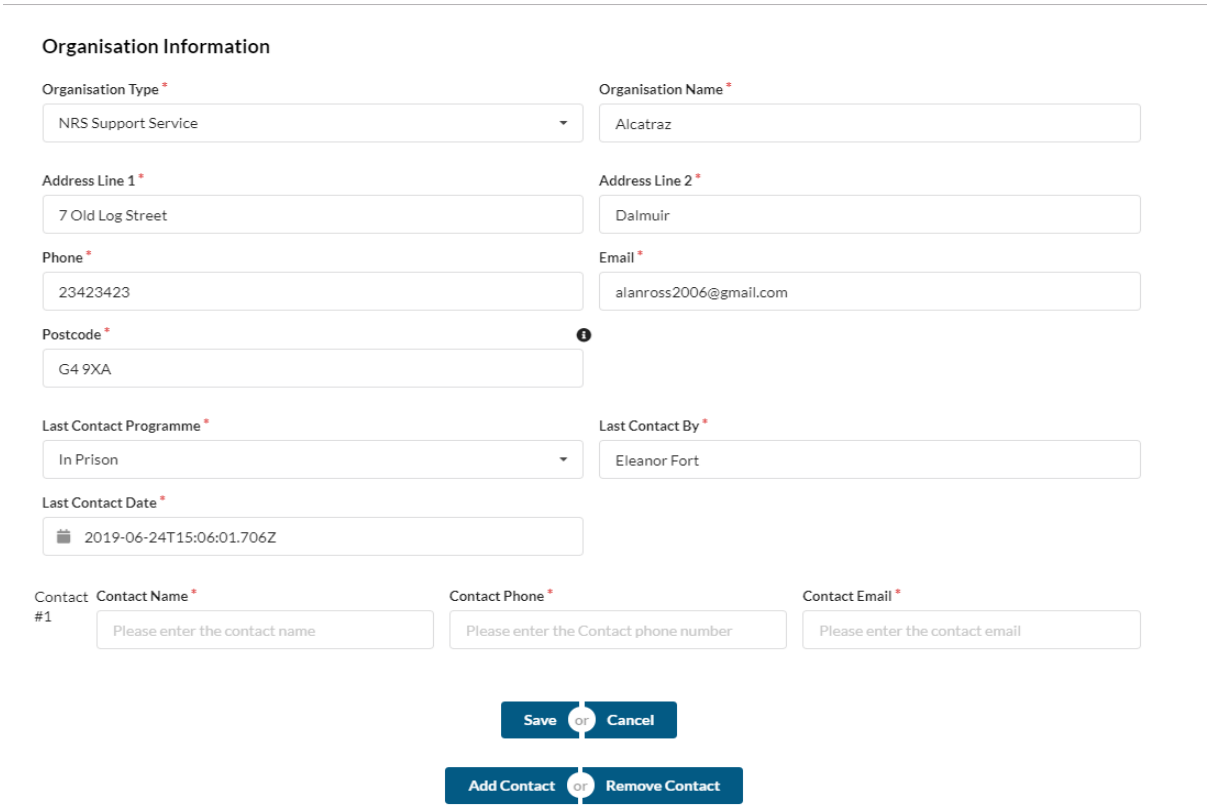

Figure 22: Appearance of form halfway through developing multiple contacts.

.

The plan initially was to make it so that when the form was submitted, the “contacts” array would be updated locally and then only when the save button was being pressed, that it would send the contacts information to redux. That would be the ideal solution however the way that the project had been set up was that each key press in a field updated Redux which wasn’t as simple due to the dynamic nature of contacts. Actions and Reducers were then created for updating Redux with the array.

This was still not something totally understood and some time was spent trying different things and debugging however no progress was really made with it.

This task was put on hold so I could refactor my previously existing “Last Contact Programme” field to be a dropdown instead as well as also implementing a field called “SIC Code” where SIC stands for “Standard Industrial Classification”. This field would only allow numerical characters as well creating validation constraints for the field so to enforce the rule that it had to be 5 characters long.

The field was also implemented to only be shown if the Organisation was an employer too due to SIC being a way to identify what kind of work an employer did and it didn’t make sense to include it if it wasn’t. This task offered a short breath of fresh air due to it not being as tricky as the task of implementing the feature of having multiple contacts.

Once multiple contacts was being worked on again, a day or two more was spent on trying to make it so that the corresponding values in the contacts array was being updated when the fields were being updated before finally being able to get it working. Once that had been done, the attempt to update Redux with the whole array started.

The way it was being done was that the whole array was being sent each time a field received a key press, so it wasn’t very efficient way of updating data, however the project was already not efficient because ideally form information is kept locally to that component rather than having each key input being sent to the redux store. The redux action and reducer that was created seemed to work though as the “selectedOrganisationData” object was being updated as it should have been however it was noted that for some reason “initialOrganisationData”, the object that stores data that is stored before any changes are made was being updated as well which wasn’t supposed to happen because, the “initialOrganisationData” object was used so that if you pressed cancel, the information within the fields would be reset to what the information was there before. The save button also had logic attached to it so that it would only be pressable if there was a difference between “selectedOrganisationData” and “initialOrganisationData”.

For now this concern was left aside to change the way contacts were removed, how the contacts were displayed and how to actually validate multiple contacts due to the dynamic nature of it due to the possibility of fields being added and removed and how a static JSON object would be used to validate these fields.

It was realized that the way contacts were removed was quite inflexible due to the fact there only being one button and that would remove the last contact added but what if it was only desired to remove the first contact that was added rather than the 3rd? It wouldn’t be very user friendly, so the remove button was refactored so that a bin icon was rendered for each contact so if that button was pressed, that contact would be deleted, the contrast in this can be shown in figure 20 and figure 19.

It was also decided that the list of contacts could look a lot better and so a semantic UI table component was used instead to render the multiple contacts and that is shown in figure 19 too.

Once these two things were done, it was then time to work on the validateForm function so that it would validate the multiple contacts as well. The constraints for contacts looked like this:

```
"contactConstraints": {
  "contactName": {
    "presence": {
      "message": "^Contact name is required.",
      "allowEmpty": false
    }
  },
  "contactPhone": {
    "regex": {
      "pattern": "^[0-9]+$",
      "message": "^Contact phone number is required. Only numbers are allowed."
    }
  },
  "contactEmail": {
    "email": {
      "message": "^Contact email address is required. '%{value}' is not a valid email."
    }
  }
}
```

Figure 23: Constraints for multiple contacts

The way these constraints were applied was done was by initialising a local variable called “contactResults” with an empty array and then the function would check whether the “contacts” property within the “selectedOrganisationData” object was truthy. If so, then it would loop through each of the contact objects in the array using the “forEach” method and within that loop, it would push the result of the validate function on to contactResults. If the value in the field passed the validation, the returning value would be pushed would be “undefined” however if it failed the validation, the message that would be pushed would be the related validation message.

Once that was done, a variable called “resultInvalid” was declared and the “contactResults” would be looped through and each value would be checked if it was undefined, if a value wasn’t undefined then it means that an invalid result came from the validation function and “resultInvalid” would be set to true.

The next part was to check whether “resultInvalid” was truthy and if so, the validation results would only then append the “contactResults” because in Javascript, an array of undefined values is considered truthy whereas the value “undefined” itself, is considered falsy and so it was causing the final validation “results” variable to be truthy meaning that the form was invalid when in fact it was valid.

Another change also had to be made in ModifyOrganisation’s where the validationResult attribute had to check whether the contacts property of validationResults was truthy and only then would it then check if that specific contact within the contacts property was truthy and only once that was truthy then the specific property (e.g contactName) of the specific contact (e.g of index 1) could be set as the value of that attribute.

If these checks weren't in place then trying to access the index of an undefined value that didn't exist or trying to access the property of an undefined value would result in a TypeError that would break the application.

With these changes in place, the validation now worked perfectly for multiple contacts and an assumption was made that the keys given to each contact to identify them had to be permanent rather than having it be based on the contact's index due to the fact that it can cause problems if you have a dynamic list of items (Pokorny, 2019). The problem was though that it wasn't desired to have permanent id's for these organisation contacts and there wasn't any other way to give them unique keys without using the index of the map function.

There was also the fact that the way the contacts were added and removed from the list would never cause an issue index wise and cause unexpected behaviour. The only way unexpected behaviour would occur would be if it was possible to add contacts to the start of the array rather than at the end so a key of greater than 0 would never be at the index 0. For these reasons, the index remained as the key however if in the future, this is to be changed, each contact will need to be given a permanent id.

With that decision decided, the issue still remained that changes to the multiple contacts fields could not be saved unless one of the other fields were changed. This was due to somehow the multiple contact data being updated within the "initialOrganisationData" object as the same time the multiple contact data within the "selectedOrganisationData" object was being updated in redux as observed by the Redux plugin for chrome dev tools.

This was very confusing and the first assumption that was made was that the new action and case statement that was created for updating Redux with the whole array was not behaving as it should and so instead, the Redux action and reducer case statement that was used for the other form fields was to be used to fix the problem. Once redux was eventually being updated again with that action and reducer case statement in place, the same issue was still there.

The only reason left now for this happening was that somehow the object "initialOrganisationData" was copying by reference the "selectedOrganisationData" object somewhere however the way "initialOrganisationData" was being set was through a setState function that was using destructuring which was the known way to create a clone and not a copy by reference. Some googling helped to clarify that actually destructuring was only for shallow copies than deep copies. This meant that any nesting properties were being copied by reference rather than having an actual copy of the actual values and this made sense due to the fact that the other fields weren't being updated in "initialOrganisationData" too.

What was required to create a deep copy of another object was the following: "initialOrganisationData = JSON.parse(JSON.stringify(selectedOrganisationData))" where JSON.stringify would expand the entire object out including the nested properties into a JSON string and then JSON.parse would read that JSON string and turn it into an object again. With this revelation, multiple contacts now fully worked. The new redux action and

reducer wasn't put back in use though because the current action and reducer was more efficient than the other one that updated Redux with the entire contacts array.

After a bit more polishing such as making validation box sizes consistent, fixing some regular expressions, the decision was made to try and unit test some components which will be mentioned in the verification and validation chapter.

While implementing some unit tests, there was other tasks in other pages such as Customers and Users that had to be done.

5.4 Users User Story Features:

5.4.1 User Roles and Geography:

One such task was simply adding another dropdown field in Customers upon further communication with the client as to how Users would interact with Customers. It was found that Users and Customers would be given an area attribute so that only Users from that area could see Customers from that area unless they were an admin user and in that case they could see customers from all areas.

Area was then also added to Users too which then caused for some conditional logic to then be applied.

The issue was that it was possible to assign a user a non-admin role while at the same time the ability to view Customer's from both area's which shouldn't be allowed to happen. This also meant if you assigned a User with the ability to see both Area's, the user then shouldn't be allowed to be assigned a non-admin role. So initially it looked like this where NR Manager is a non-admin role:

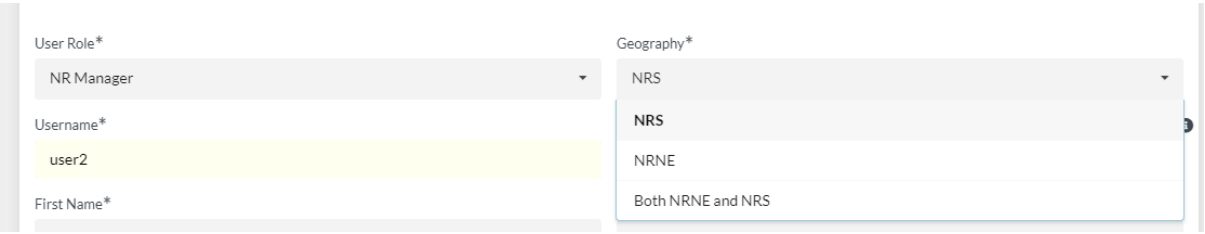

Figure 24: With NR manage (a non-admin role) selected, it was possible to select "Both NRNE and NRS" as a geography.

But it was changed so NR Manager and other non-admin roles could only see NRS or NRNE:

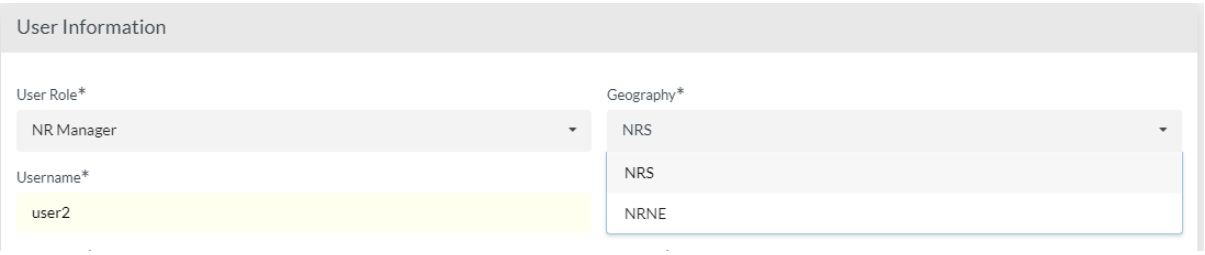

Figure 25: A user with the role of NR Manager is now limited to only NRS or NRNE geography.

The other issue was that there was two roles that could be assigned but each one was tied to each of the area's and so the idea was that if one of these roles were assigned then the corresponding value in the dropdown would be preselected and the dropdown would be disabled. The beginning of this functionality was being done with the actual form's attributes however it became clear that the complexity of it required for functions to be created in the Container instead and for them functions to be passed down to the form components themselves through props.

So initially it looked like this when you picked a NRS mentor:

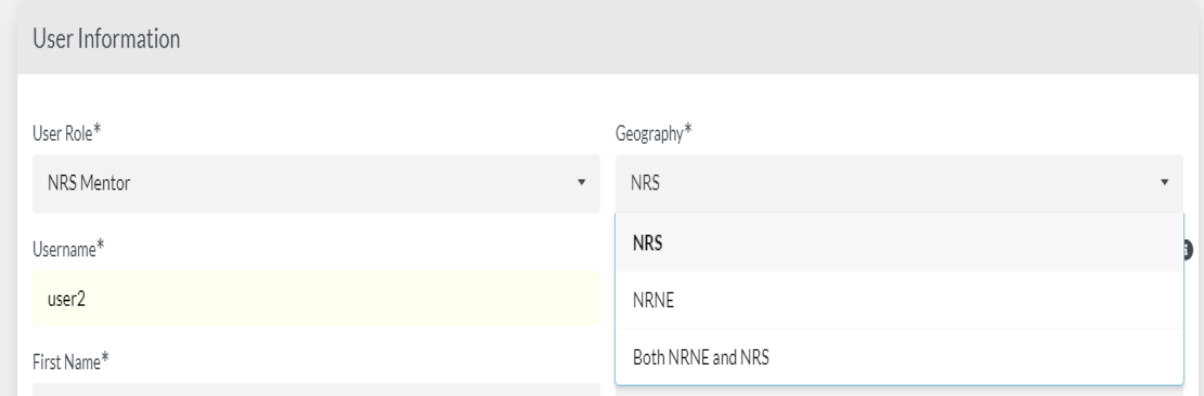

Figure 26: With NRS Mentor picked (a non-admin role) but specific to NRS, all geographies were allowed.

And then looked like this with geography field preselected and disabled:

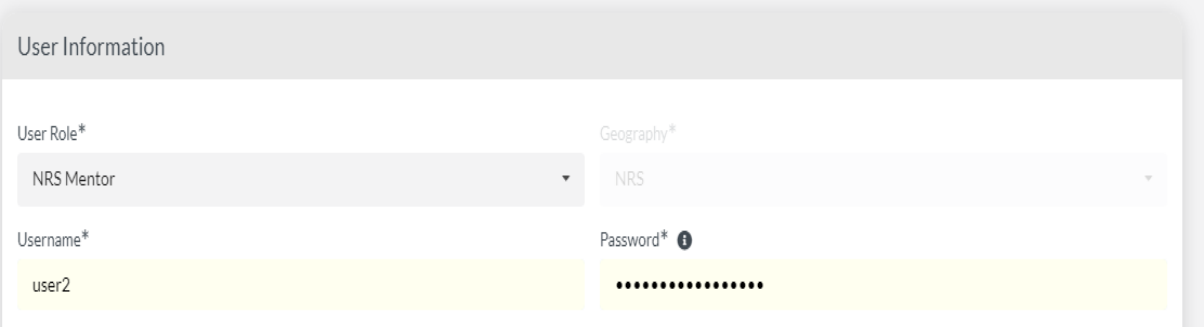

Figure 27: With the update to the logic, now the selection of a NRS mentor meant that "NRS" was preselected in the dropdown field and the dropdown was disabled.

Upon further reflection it was realized that the name of these two roles where the area was appended at the end could actually be changed so that the preselection logic was redundant so this logic was removed later on.

6 VERIFICATION AND VALIDATION

When a client is paying Pulsion Technology to build a software system, it is absolutely vital for Pulsion Technology to ensure that the system is working as the client is expecting it to. That is why the requirements document is more than 100 pages long so to specify the details of what they want and at the same time, more and more is being added as development is being carried out and questions are arising about the precise workflow of how entities such as a Customer is created with regards to a Referral which according to the client, the creation of a Referral should create a Customer. It is subjects like these that require the client to come in and meet with the senior software engineer to discuss the exact process of these issues so that the senior software engineer can then guide the rest of the team in the right direction. If the code isn't working as it should, the company loses money and if the company loses money then there isn't enough money to pay wages and then there could be loss of jobs so this is why verification and validation is absolutely essential for a company like Pulsion Technology.

Pulsion Technology hasn't been a champion of unit testing for a while, however recently there has been efforts to try and integrate it into development. There are even lunches dedicated to Clean Code videos by Robert "Uncle Bob" Martin[38] which include videos on Test Driven Development and now tutorials are in place for people to start practicing how to go between unit testing and development as it's often so tempting to just take 5 steps at a time in development rather than one step at a time to make sure that one step doesn't break anything. Front-end unit testing is something that has also became of interest and packages such as Jest and Enzyme allow for front-end components to be tested.

There are several ways to test the front-end but the method of best practice is to shallow render the components because that gets rid of the need to render dependencies which would otherwise require many more mock functions and data to be required. The other ways opposed to shallow rendering was attempted however after trying it and running into problem after problem, it was decided that time was better spent on development or shallow-rendering other components however this may be returned to in in future work. With shallow-rendering, snapshots of what is being rendered can be generated.

6.1 FRONT-END TESTING:

Another person on the front-end team had already did research into front-end unit testing and had been tasked with writing a document on it, so this document came in use when unit-testing other components. Some components have more depth though and shallow-rendering by itself wasn't able to test that without the use of a function called "dive" which allowed child components to be seen as well and this allowed for two tests to be done that checked whether a component was rendered or not based on a Boolean value. One of components tested was the ComponentHeader component and the tests done for these was

[38] https://cleancoders.com/

a mix between snapshot tests, normal shallow rendering tests to see if props are being passed in and also a test to check if an on-click event works:

```
const redirectButton = jest.fn();
let wrapper, props, componentHeader;    'wrapper' is declared but its value is never read

beforeEach(() => {
  props = createTestProps();
});

describe("<ComponentHeader /> will render correctly with a pageTitle and headerList", ()
  it("renders organisation's componentHeader correctly", () => {
    const renderer = new ShallowRenderer();
    componentHeader = renderer.render(
      <ComponentHeader
        pageTitle="Organisation Details"
        headerList="Organisations"
        {...props}
      />
    );
    expect(componentHeader).toMatchSnapshot();
  });
});

describe("<ComponentHeader /> will render a pageTitle and headerList", () => {
  it("should render the organisation's pageTitle and headerList", () => {
    wrapper = shallow(
      <ComponentHeader
        pageTitle="Organisation Details"
        headerList="Organisations"
        {...props}
      />
    );
  });

  it("should render the customer's pageTitle and headerList", () => {
    wrapper = shallow(
      <ComponentHeader
        pageTitle="Customer Details"
        headerList="Customers"
        {...props}
      />
    );
  });

  it("should render the user's pageTitle and headerList", () => {
    wrapper = shallow(
      <ComponentHeader pageTitle="User Details" headerList="Users" {...props} />
    );
  });

  it("The component will have two <ShowIfAuthorised>'s", () => {
    componentHeader = shallow(<ComponentHeader {...props} />);
    expect(componentHeader.find('[data-test="permissionsUpdate"]'));
    expect(componentHeader.find('[data-test="permissionsCreate"]'));
  });
});

describe("<ComponentHeader /> has a redirect button with an onclick event", () => {
  it("The text will be a button that has an on-click function", () => {
    componentHeader = shallow(<ComponentHeader {...props} />);
    componentHeader.find('[data-test="redirectButton"]').simulate("click");
    expect(redirectButton).toHaveBeenCalled();
  });
});
```

Figure 28: Unit tests for ComponentHeader

In the above figure, there is code that runs before each test so to pass in props (data) that is required for the components to function. The first test involves shallow rendering and then expecting the “ComponentHeader” component to match the snapshot that has already been generated from a past run of the test.

To better explain what a snapshot is, a typical snapshot test case would be when there is code that renders a UI component. The test then takes a snapshot (an image) and then compares the snapshot to a reference snapshot file which is stored inside the test folder, next to the test. The test will fail if the two snapshots don’t match. This then means either of two things: either the change is unexpected, or the reference snapshot needs to be updated to suit the

new version of the UI component. An issue that can arise is when something like Date() is used. This is because Date() returns a new Date each time it's called as expected and so if a snapshot is used of a component involving Date(), the snapshot will always fail:

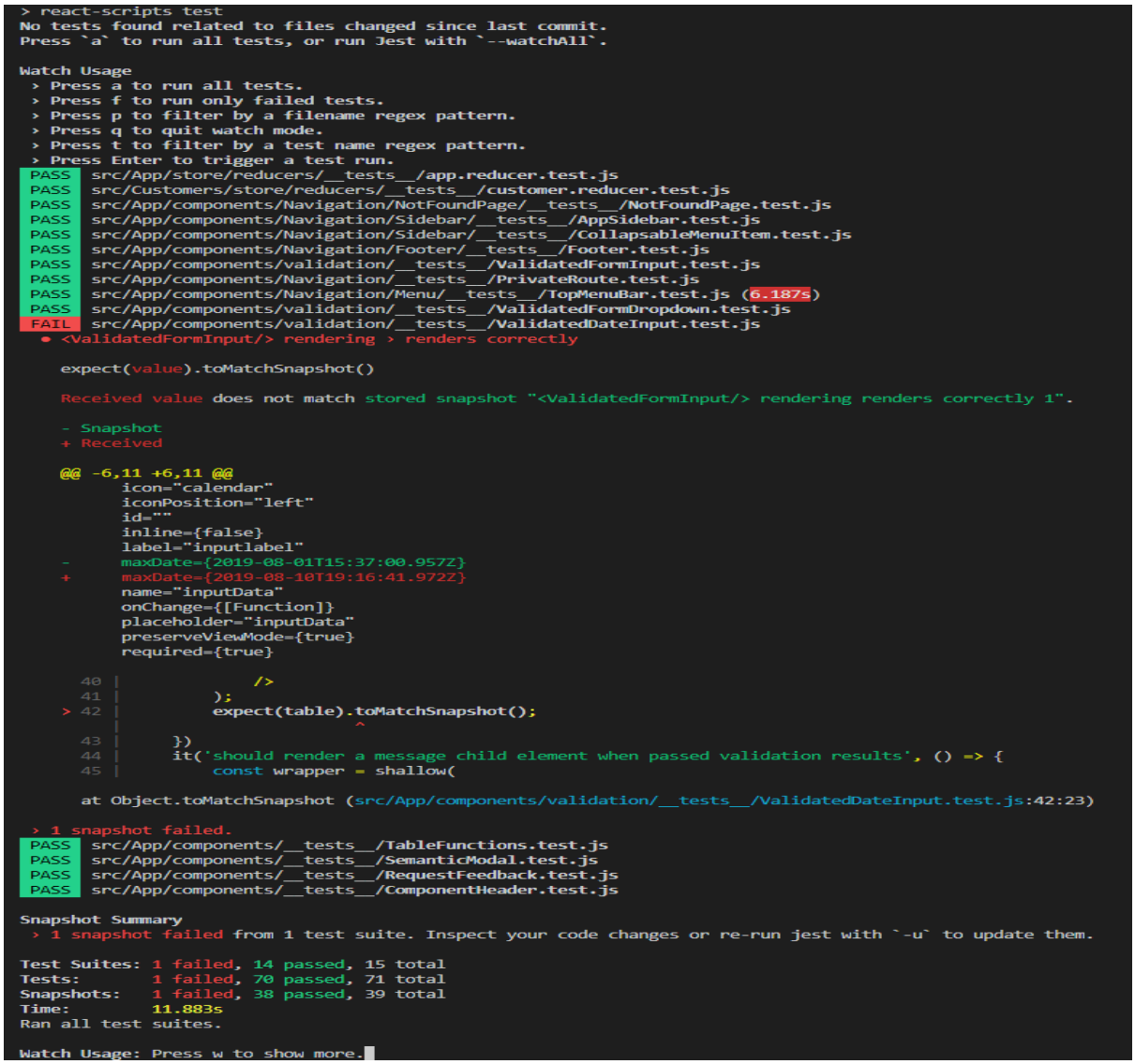

Figure 29: Front-end test results

The snapshot of ComponentHeader looks like this:

```
// Jest Snapshot v1, https://goo.gl/fbAQLP

exports[`<ComponentHeader /> will render correctly with a pageTitle and headerList renders organisation's componentHeader correctly 1`] = `
<Grid
  className="ComponentHeader"
  columns={3}
  data-test="ComponentHeader"
>
  <GridColumn>
    <div
      className="pageHeader"
      data-test="pageHeader"
    >
      <Button
        as="button"
        className="redirectButton"
        data-test="redirectButton"
        onClick={[MockFunction]}
        role="button"
      >

      </Button>
      <div
        className="headingDivider"
        data-test="headingDivider"
      />

    </div>
  </GridColumn>
  <GridColumn />
  <GridColumn
    className="FormButtons"
  >
    <Connect(ShowIfAuthorised)
      allowedRoles={Array []}
      data-test={Array []}
    />
    <Connect(ShowIfAuthorised)
      allowedRoles={Array []}
      data-test={Array []}
    />
  </GridColumn>
</Grid>
`;
```

Figure 30: Snapshot of Component Header

In the above snapshot of the Component Header file, It shows all the components at the very top layer of the component that have been rendered as well as elements like that are a layer below the components such as <GridColumn></GridColumn>

6.2 Back-end Request Testing:

Also another form of testing would be testing the back-end through Postman. If there was no information being presented in the front-end after a request for data, or the data appears to

be wrong, Postman is a way to send requests and retrieve that data back while skipping the need to use the front-end which might not work, and by using this, it could then be determined whether it is the front-end that isn't working or it is the back-end that isn't working where the top body shows body of request and the bottom body shows the response from AWS:

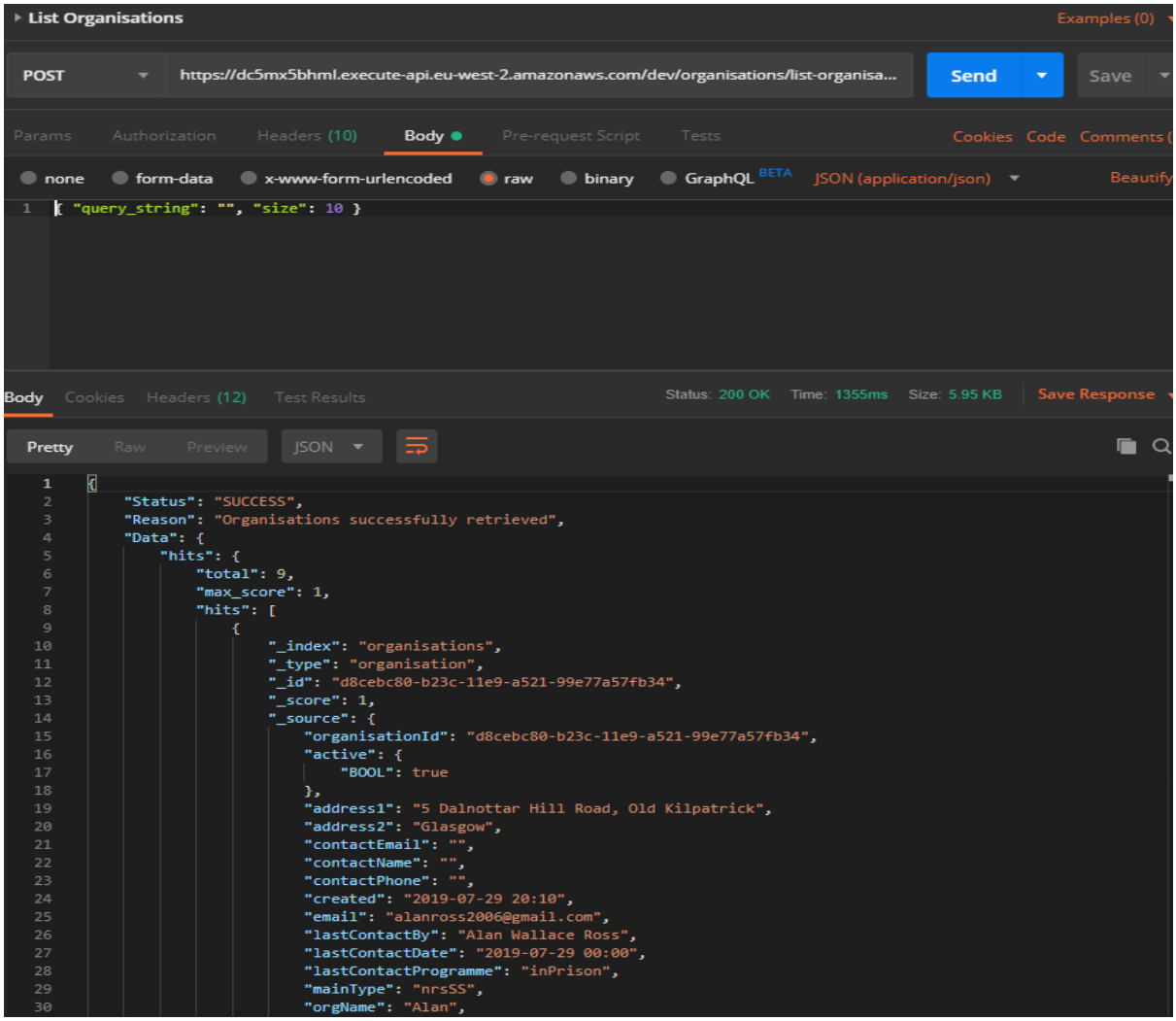

Figure 31: Postman sending a request to retrieve a list of organisations

There is a lot of information presented in figure 31, however the request is "List Organisations" as can be found at the top left, to send this request to the AWS server, you click the bright blue "Send" button near the top right. This sends the object in the top body of code "{ "query_string": "", "size": 10}" and this code is handled by the server, telling the server to only send 10 organisation's back, if there are more than 10 organisations. This then sends back the object that starts with "Status: SUCCESS". If the back-end isn't working though, an error is often seen.

The reason for why only 10 organisation's are returned is to keep server costs down, as the more data returned back from AWS, the more AWS charges Pulsion so limits are put in place to keep costs down.

6.3 INTEGRATION TESTING:

Aside from unit-testing, there is quite a lot of integration testing that is undertaken when merge requests are made. Every time a merge request is opened, the team is tagged in the merge request description so to let people know a merge request has been opened because if a person is tagged, they receive an email. The process is then to pull from master so to make sure the most up-to-date version of that branch is on the computer to then check out locally. When checked out, the entire system is being ran and then it is time to tinker around as much as reasonable with the new feature that has been added, if it is a functional feature, and try and break it if possible.

It is not always so simple though as there are two repositories, a front-end repository and a back-end repository. Sometimes the front-end won't work properly without also having to make sure you have checked out the correct branch and made sure it's on the latest version too. And then once that is done, each service needs to be entered into through "cd <service_name>" and then have "npm install" called to make sure all the packages have been installed for the code to work and then "serverless deploy" to make sure that the code in the repository is synced with the code that is being hosted on AWS Lambda so that the updated code is actually the code executing on AWS. Before this is done however, sometimes all currently existing Users, Customers and Organisation's would have to be deleted if there was a change in how the back-end processed these entities and so these entities would need to be deleted so that new default entities could replace them when "serverless deploy" is called. The reason for why it doesn't work sometimes and that previous "entities" like Organisations have to be deleted is due to the back-end or the front-end needing to change the structure of the object of an entity such as Organisation to solve a problem and so the code to handle this new structure needs to be changed. If the code changes to handle this new structure, entities with the older structures will not be handled properly and so the application wouldn't behave properly by not showing "entities" properly or breaking on compile.

Sometimes there are problems and this process of checking out can take longer than expected. There can also be cache issues where a browser caches the previous version of the web app so that even though the web app has been updated, the browser still runs the out-of-date version. The easy way to tell if it is a cache issue though is by running the web application through Chrome's incognito browser which doesn't cache anything.

Aside from just checking out a branch, a team member can also inspect the code on GitLab and quote code snippets asking for the reason why the code is the way it is and this helps verify that the code is doing what it is supposed to be doing but also to make sure that the code's purpose is in line with the requirements.

6.4 Q/A TESTING:

After a sufficient amount of progress has been made through branches being merged with the master branch, it is then time for an in-house test release where in-house testers write up tests solely from the User Stories and while they test separate functions they also do things

like end to end testing making sure the application behaves as expected throughout which is called positive testing as well as also doing negative testing where they check that they can't do things they're not supposed to be able to do. Overall a much more rigorous test since engineers can't afford the time to go through all test cases, for example there are multiple permissions for different roles of user and so it is important to make sure that a user with a certain role can only do what's expected, nothing more and nothing less.

An example of a test by the in-house testing staff can be shown below where a functional test is being written up with a priority of the user story being high, and an estimate of how long the test should take. There is also a reference to the ticket shown on Jira as well. Below that is the preconditions, for example, what user was logged in as, what pages were clicked to arrive at the steps that were being tested. The below test was the testing of creating an organisation where there are multiple steps to creating one shown.

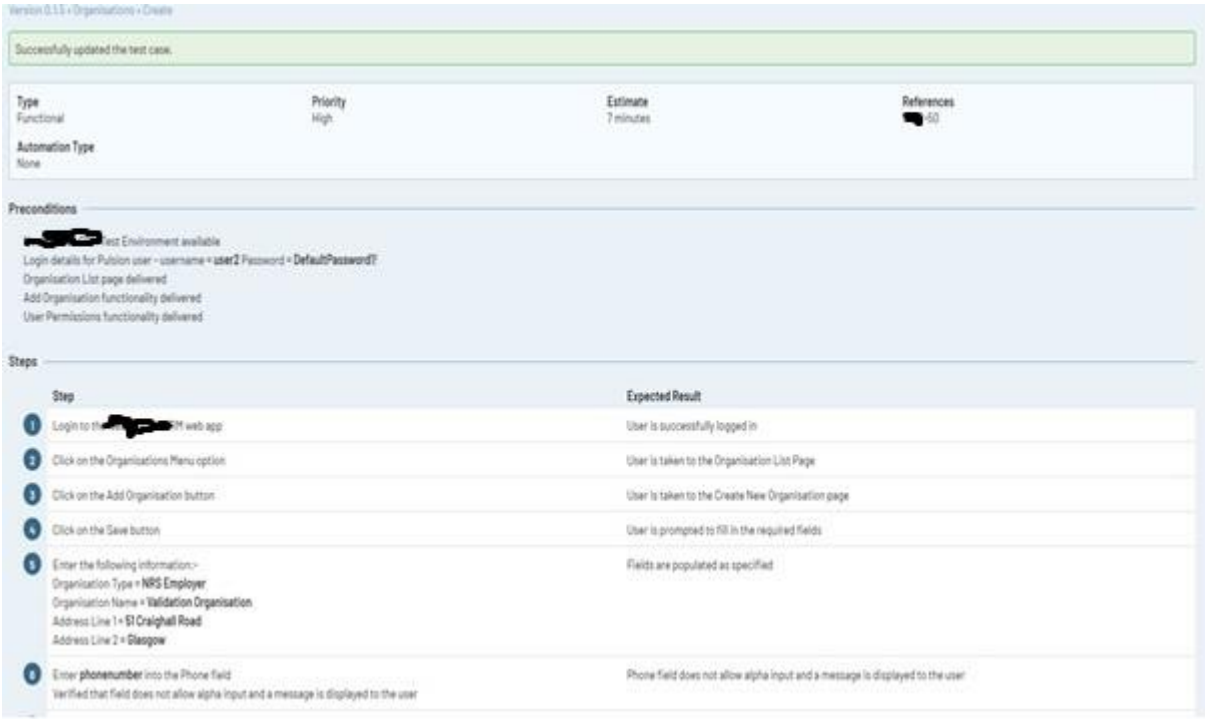

Figure 32 Screenshot of steps for a test on creating an organisation that a Q/A has written up and followed.

Once any bugs that have been brought up by the testers have been fixed, then it is time for them to test again and if there are no bugs, then it can then be time for an user acceptance test (UAT) to take place. Each test release and UAT release has an accompanying document which documents what changes have been made as well as the version number. A user acceptance test hasn't been released yet so this part of the validation and verification can't be discussed as in-depth as the other testing done however it would be quite similar in the sense that any bugs caught would be brought up as a ticket for the engineers to fix and then once done, the cycle continues to the test release and then to the UAT again. It's also to be expected that a user acceptance test will provide a lot more validation feedback than verification feedback.

6.5 User Evaluation:

One example of user evaluation feedback that was given was based on a question posed around multiple contacts. As part of the requirements, a table of multiple contacts was to be developed and so it was developed. There was no detail on how many contacts a table should have though. The question of how many contacts a table should then have was raised to the senior software engineer who then asked the client. Once they asked the client, the senior software engineer gave back the feedback and it turned out that an Organisation can have zero contacts as a minimum and 3 contacts as a maximum. This hasn't yet been applied due to other work being higher priority however the change is planned and has been added as a ticket on Jira.

Limitations and Future Work

6.6 Limitations:

Due to priorities, not all functionality has been totally implemented in the Organisation's page. This is mainly due to the back-end team having other things to work on such as Search functionality rather than being focussed on adding API's that would feed some form components data, for example, the SIC code field should let a user type in numbers and based on them numbers, display what kind of Organisation that is. Both address lines should also be filled based on a postcode API that would offer addresses based on what postcode was entered.

6.7 Future work:

As part of future work for this particular CRM, there is a lot more work to do however there is no set deadline compared to another project that does and so commands a higher priority. For Organisation's alone, it still hasn't been completely finished. There is still no ability to display last contact information over the last two months, no ability to search on local authority area and the ability to migrate data from spreadsheet into an organisation's record has still not been implemented. In addition to that, when assigning an organisation its address, there should be an API that retrieves the Postcode as well as the local authority area of that postcode so that these values are automatically inserted. Going forward, there might also be additional 'type' options for programmes as well as a secondary 'type', further adding complexity to the application.

On top of organisation's, there is Task Management, Risk Assessment, Baseline Assessment, Diary Management

1. Task Management – an area of the CRM to tell the user what upcoming tasks they have for the week so that no tasks are missed.

2. Baseline Assessment – the ability to record baseline assessments for customers so that their needs can be identified.
3. Diary Management – the ability to record meetings in a diary so that the user's team knows where they are
4. Sharing Records – the ability to share records between partner organisation's
5. Audit history – the ability to see when any field has changed on the system so that changes can be tracked to identify mistakes or potential fraudulent activity
6. System features – the ability to take a picture of a form that contains a customer signature and be able to attach a copy of the picture to the customer record
7. System features – the ability for the system to allow the user to use a report builder and download results to a document on an excel spreadsheet so that the user can manipulate data if needed.
8. Mail merge – The ability to produce mail merge letters for weekly contact so that a user can stay in touch with customers who do not have a phone and do not attend meetings.

There are 59 of these user stories in total and in the requirements document they are detailed much more and so to say 8 of these or more may be done in the future as an individual in a team may be a fair amount to do considering there are only 4 people doing the front-end. The work for the client is ideally to be finished by December/January 2020 so there is a lot more of development to be done.

In the future, there will be more specification on how React components should look like, as there are different types of components, there are components that are Classes and can have lifetime cycle functions that are called at specific times of their lifetime, e.g componentDidMount, componentShouldUpdate, componentDidUpdate and componentDidUnmount which are quite self-explanatory however these functions add complexity, and need to be treated carefully or else an infinity loop of rendering can be caused. There are also components called functional components that's only function is to render a UI component and so if there a component that is made from a Class but there is no need for that component to hold state or to have lifetime cycles apart from "render" then that component can be made into a more readable form with less code. This was spotted a lot throughout some of the code and due to time constraints, not all code has been refactored to reflect best practice according to the React community.

7 CONCLUSION

To conclude, it is best to refer to the objectives that were set at the start of this paper.

With regards to learning, which was the first part of the objectives set, the basics of JavaScript, React, AWS as well as languages such as HTML and CSS have been learnt. With the basics learnt, and 3 months in, a deeper understanding of the languages is slowly but surely coming. Not only have languages been learnt though but a lot of other things such as following the

practices of a company that has stand-ups each day, how to write JIRA tickets, how to write descriptions of merge requests, the process of deploying an app, and much more has been learnt too or still in the progress of learning. Overall enough has been learnt to be able to contribute to the project and so this objective could be considered to be completed.

Following the objective of learning, was the objective of developing the Organisation's section of the CRM. Based on the fact that all three pages of the Organisation's section have been developed as well as functioning correctly, this objective can also be considered completed.

Overall React is very fast and provides the ability for developers to provide users with a lot of responsiveness. React has also been quite intuitive as it has been easy to learn and understand the main principles and patterns of building a React application. The use of microservices as well seems to be the way forward as the way user verification is handled can be copied over and also at the same time avoiding the need to invest in physical architecture which isn't as easy to scale as a cloud solution can. So with all that said, these technologies are perfectly suited to being able to develop CRM systems that involve a lot of replicated themes such as create, read, update and delete (CRUD) operations as well as permissions and having a dashboard to show what work a user has to do for that week or month and with React's ability to create components, a lot of re-use should occur.

With stand-ups every day, there is plenty of communication on what everyone is doing and enough progress is being made based on the estimates that are devised by the head of project delivery as well as the senior software engineer so while it is still early, it seems like the project is going to be a success.

8 BIBLIOGRAPHY

Howard, M. (2019). *14 CRM Stats That Sales Professionals Need to Know | Nutshell*. [online] Nutshell. Available at: https://www.nutshell.com/blog/crm-stats/ [Accessed 30 Jul. 2019].

http://comparecamp.com/history-of-crm-software/. (2019). *History of CRM Software*. [online] Available at: http://comparecamp.com/history-of-crm-software/ [Accessed 30 Jul. 2019].

Raab, G., Ajami, R.A., Gargeya V. & Goddard, G.J. (2008) "Customer relationship management: a global perspective" Gower Publishing

Peppers, D. & Rogers, M. (2011) "Managing Customer Relationships: A Strategic Framework" John Miley & Sons

Mathur, U.C. (2010) "Retail Management: Text and Cases" I.K. International Pvt Ltd

Nucleusresearch.com. (2019). *CRM pays back $8.71 for every dollar spent – Nucleus Research*. [online] Available at: https://nucleusresearch.com/research/single/crm-pays-back-8-71-for-every-dollar-spent/ [Accessed 7 Aug. 2019].

Ivey, J. (2019). *CRM Software UserView | 2014*. [online] Softwareadvice.com. Available at: https://www.softwareadvice.com/crm/userview/report-2014/ [Accessed 7 Aug. 2019].

Cleveroad Inc. - Web and App development company. (2019). *How to Create a CRM System for Your Business [An Extensive Guide]*. [online] Available at: https://www.cleveroad.com/blog/how-to-build-your-own-crm-system-avoiding-common-mistakes#benefits-of-custom-crm-development [Accessed 7 Aug. 2019].

Canalys.com. (2019). *Canalys Newsroom- Cloud market share Q4 2018 and full year 2018*. [online] Available at: https://www.canalys.com/newsroom/cloud-market-share-q4-2018-and-full-year-2018 [Accessed 3 Aug. 2019].

Pokorny, R. (2019). *Index as a key is an anti-pattern*. [online] Medium. Available at: https://medium.com/@robinpokorny/index-as-a-key-is-an-anti-pattern-e0349aece318 [Accessed 4 Aug. 2019].

9 APPENDIX:

9.1 FOLDER STRUCTURE TREE:

```
C:.
|    index.js
|    index.scss
|    serviceWorker.js
|    tree.txt
+---App
|   |    App.js
|   |    App.scss
|   |    App.test.js
|   |    routes.js
|   +---assets
|   |   \---images
|   |            logo.png
|   |            logo_dark.png
|   |            logo_small.png
|   |            logo_tiny.png
|   +---classes
|   |        index.js
|   |        SystemHeaders.js
|   |        User.js
|   +---components
|   |   |    ComponentHeader.js
|   |   |    index.js
|   |   |    RequestFeedback.js
|   |   |    SemanticModal.js
|   |   |    ShowIfAuthorised.js
|   |   |    TableFunctions.js
```

```
|   |   ├──Layout
|   |   |   index.js
|   |   |   Layout.js
|   |   |   ResponsiveContainer.js
|   |   └──css
|   |           Layout.scss
|   |           ResponsiveContainer.scss
|   |   ├──Navigation
|   |   |   PrivateRoute.js
|   |   |   ├──Footer
|   |   |   |   Footer.js
|   |   |   └──css
|   |   |           footer.scss
|   |   |   ├──Menu
|   |   |   |   TopMenuBar.js
|   |   |   └──css
|   |   |           topMenuBar.scss
|   |   |   ├──NotFoundPage
|   |   |       NotFoundPage.js
|   |   |   └──Sidebar
|   |   |           AppSidebar.js
|   |   |           CollapsableMenuItem.js
|   |   |           Sidebar.scss
|   |   ├──validation
|   |   |       index.js
|   |   |       ValidatedDateInput.js
|   |   |       ValidatedFormDropdown.js
|   |   |       ValidatedFormInput.js
|   |   |       ValidatedMultipleContacts.js
|   |   └──__tests__
|   |       └──__snapshots__
|   ├──containers
|   |   ├──Login
|   |   |   index.js
|   |   |   Login.js
|   |   |   Logout.js
|   |   |   PasswordReset.js
|   |   └──css
|   |           login.scss
|   |   └──ResetPassword
|   |           index.js
|   |           ResetPassword.js
|   ├──resources
|   |       AppPermissions.js
|   |       DropdownOptions.js
|   |       Enums.js
|   |       index.js
|   |       Strings.js
|   ├──services
|   |       auth.service.js
|   ├──store
|   |   index.js
|   |   ├──actions
|   |   |       app.actionTypes.js
|   |   |       auth.actions.js
|   |   |       auth.actionTypes.js
|   |   └──reducers
|   |           app.reducer.js
```

```
│   │           auth.reducer.js
│   └──styles
│           react-table.scss
│           _colours.scss
├──Customers
│   ├──components
│   │   │   CustomerTable.js
│   │   │   index.js
│   │   │   ModifyCustomer.js
│   │   ├──css
│   │   │       CustomerComponents.scss
│   │   └──validation
│   │           index.js
│   ├──containers
│   │   │   index.js
│   │   ├──CustomerCreateContainer
│   │   │       CustomerCreateContainer.js
│   │   │       index.js
│   │   ├──CustomerEditContainer
│   │   │       CustomerEditContainer.js
│   │   │       index.js
│   │   └──CustomerListContainer
│   │       │   CustomerListContainer.js
│   │       │   index.js
│   │       └──css
│   │               CustomersTable.scss
│   ├──resources
│   │       DropdownOptions.js
│   │       index.js
│   │       Strings.js
│   ├──services
│   │       API.js
│   │       axios-customers.js
│   │       index.js
│   ├──store
│   │   ├──actions
│   │   │       customer.actions.js
│   │   │       customer.actionTypes.js
│   │   └──reducers
│   │           customer.reducer.js
│   └──styles
│           _colours.scss
│           _components.scss
│           _list.scss
├──Organisations
│   ├──components
│   │   │   index.js
│   │   │   ModifyOrganisation.js
│   │   │   OrganisationTable.js
│   │   ├──css
│   │   │       OrganisationComponents.scss
│   │   ├──validation
│   │   │       index.js
│   │   └──__tests__
│   │       └──__snapshots__
│   ├──containers
│   │   │   index.js
│   │   │
```

```
|   |   ├──OrganisationCreateContainer
|   |   |       index.js
|   |   |       OrganisationCreateContainer.js
|   |   ├──OrganisationEditContainer
|   |   |       index.js
|   |   |       OrganisationEditContainer.js
|   |   └──OrganisationListContainer
|   |       |   index.js
|   |       |   OrganisationListContainer.js
|   |       └──css
|   |               OrganisationsTable.scss
|   ├──resources
|   |       DropdownOptions.js
|   |       index.js
|   |       Strings.js
|   ├──services
|   |       API.js
|   |       axios-organisations.js
|   |       index.js
|   ├──store
|   |   ├──actions
|   |   |       organisation.actions.js
|   |   |       organisation.actionTypes.js
|   |   └──reducers
|   |           organisation.reducer.js
|   └──styles
|           _colours.scss
|           _components.scss
|           _list.scss
├──Programmes
|   └──containers
|           Programmes.js
├──Referrals
|   └──containers
|           Referrals.js
├──Reports
|   └──containers
|           Reports.js
├──Users
|   ├──components
|   |   |   index.js
|   |   |   ModifyUser.js
|   |   |   UserTable.js
|   |   ├──css
|   |   |       UserComponents.scss|
|   |   └──validation
|   |           index.js
|   ├──containers
|   |   |   index.js
|   |   ├──UserCreateContainer
|   |   |       index.js
|   |   |       UserCreateContainer.js
|   |   ├──UserEditContainer
|   |   |       index.js
|   |   |       UserEditContainer.js
|   |   └──UserListContainer
|   |       |   index.js
|   |       |   UserListContainer.js
```

```
│   │       └──css
│   │           UsersTable.scss
│   ├──resources
│   │       DropdownOptions.js
│   │       index.js
│   │       Strings.js
│   ├──services
│   │       API.js
│   │       axios-users.js
│   │       index.js
│   ├──store
│   │   ├──actions
│   │   │       user.actions.js
│   │   │       user.actionTypes.js
│   │   └──reducers
│   │           user.reducer.js
│   └──styles
│           _colours.scss
│           _components.scss
│           _list.scss
├──Work
│   └──containers
│           Work.js
├──_axios
│       interceptors.js
├──_config
│       aws.js
│       index.js
│       systemHeaders.js
├──_store
│       rootReducer.js
│       store.js
└──_styles
        _colours.scss
        _components.scss
        _list.scss
        _react-table.scss
```

9.2 IMAGE OF ORGANISATION FORM:

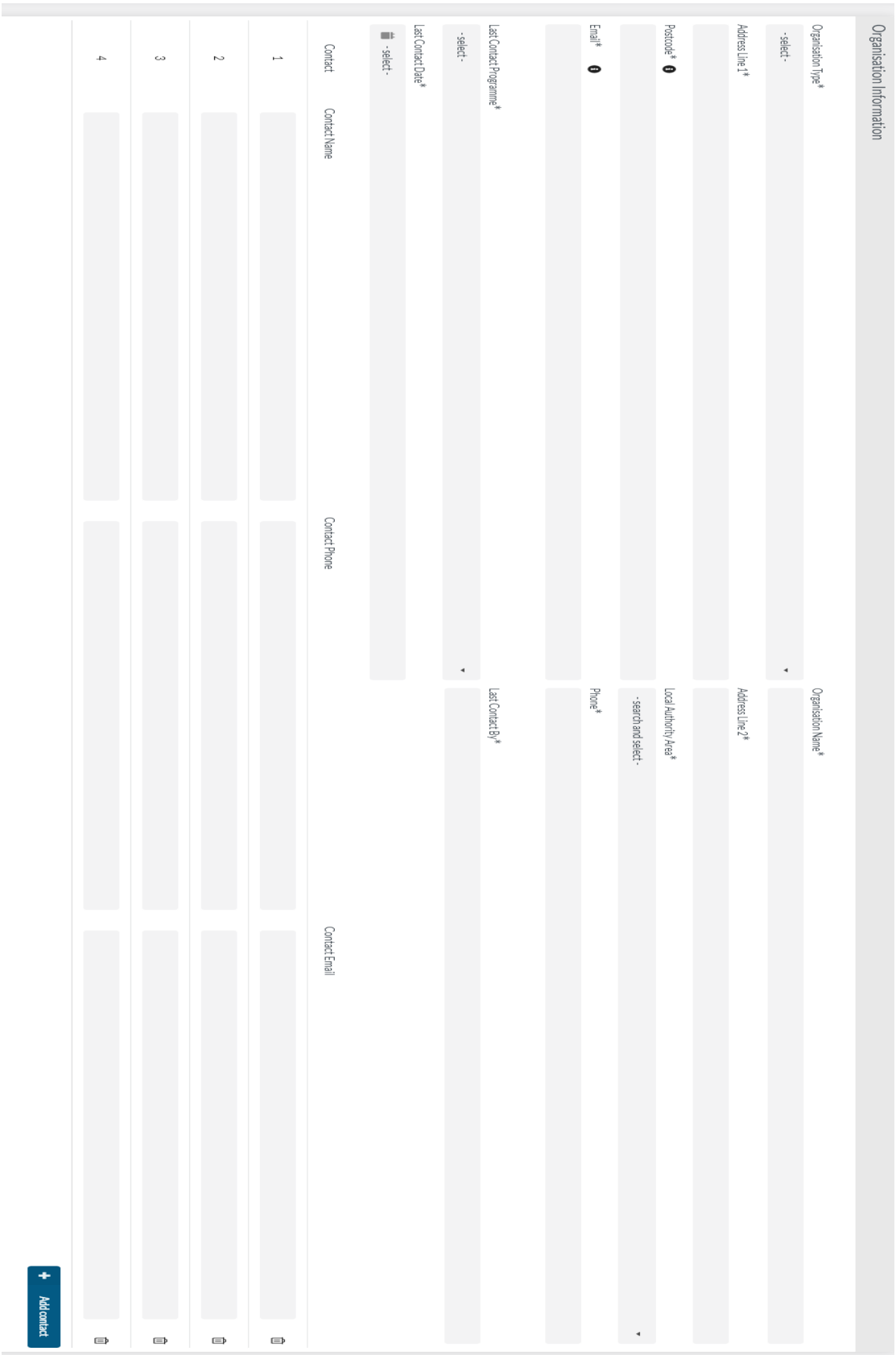

Figure 33: Enlarged version of Organisation form