Computing Project – Matthew French

Contents

Project Summary

Summary

In the past, teachers have told me that they find it difficult to log problems with IT Support at school. Due to past changes to the systems, many are unsure about which methods of communication to use when requesting help from the technicians, and many feel left in the dark about the state of their problem, whether it has been resolved, or even if their message has been received.

To help solve these problems, there needs to be a simple, centralised way of reporting problems to IT support and a system for seeing how close those problems are to being fixed.

Many teachers have a reasonable grasp of technology, but the system that I design will need to cater to those with a minimum knowledge of computers. To that end, I have decided to go with a website, because it will likely be the most comfortable and familiar application for people with limited experience with technology. It can also be easily designed for user-friendliness, which I plan to achieve by using Google's Material Design[1] philosophy, which revolves around simplistic and understandable design, mainly based on real-world lighting and motion. The website will need to several different systems function properly:

1. **Log in/Sign up**
   Teachers and students will need to see personalised data, such as their reports. Any reports they make will also need to be tied to an account to allow IT support to contact those people to ask for more information. The school uses SIMS as an Information Management System[2]and may in the future want to use data in the IMS to create users on the site. This will require a specific API[3] that can modify users in the database.

2. **Problem Reporting**
   Teachers and students need to be able to create a report and submit it to a database where it can be accessed and modified by IT support.

3. **Database Management**
   Reports, Users, and other data will need to be stored in databases. As the school's requirements change, the structure of these databases may need to be modified.

4. **Client-Side UI**
   The users will need to be able to interact with the application. The easiest and most understandable way of doing this is by using a simplistic website that can be accessed via browsers. While more difficult to implement than a command-line interface, it is far easier for people with limited technical knowledge to use.

5. **Backend API**
   An API is a set of functions exposed by an application to allow other applications to interface with it. With the website, there will need to be code running on the user's computer (the client) as well as code running on a server. The client needs to be able to send data (like reports) back to the server. The easiest way of doing this is by writing a REST API[4] on the server. This will allow the client to connect and transfer data and could allow for two-way communication using WebSocket[5] technology.

I have implemented most of these systems in the past on various projects, but the main challenge for this system will be trying to combine many individual components into a coherent application. During development, I will likely be developing several features at a time. This could end up causing serious problems if I need to roll back to an earlier version. To prevent these issues, I plan to use a VCS[6] (Version Control System) to help manage the project. As well as file management, some VCSs such as GitHub provide many other tools, such as Continuous Development and Deployment.

---

[1] More details about Material Design can be found at https://material.io/design/
[2] IMSs are used by organisations to hold information about many things, such as rooms or employees. They can then be integrated into other applications via APIs, allowing 3rd party apps to use the data.
[3] An Application Programming Interface is a set of commands and functions exposed by one program to let other 3rd party applications to interact with the code in a programmatic way. A simple way to think of it is like a user-interface for computers.
[4] REST APIs are a set of URLs that are provided by a website that can be trigger code running on the server.
[5] WebSockets are a set of protocols that can be used for real-time two-way data transfer between a client and a server.
[6] VCSs track changes to a project, allowing *commits* to be made when the code is ready. This lets developers work in a team, as well as allowing them to *rollback* to previous code if mistakes are made.

Research Plan

There will be 3 main stages of research:

- Requirements
- Technologies
- Capabilities

The first stage will be to develop an understanding of the requirements needed by both the users who report the problems and those in IT Support who will be dealing with them. For this, I will be interviewing various people from both groups. Some of these interviews will be recorded and written down, but some of them will be in the form of casual conversations. These two techniques combine should allow me to understand what is needed from the application.

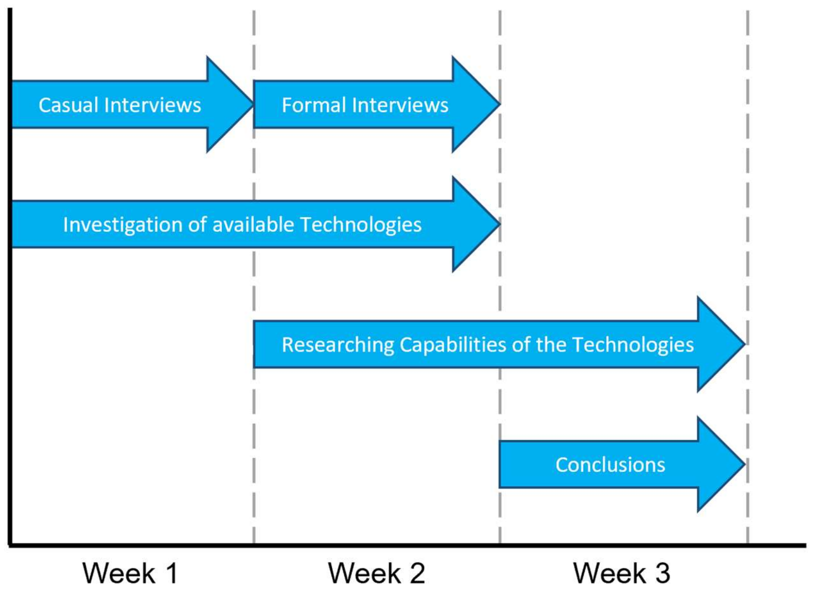
The second stage is closely related to the third. In this stage, I will be researching the different languages, frameworks and technologies that could be used as part of the application. The three main areas of research will be the backend, frontend, and database. Most of this research will be secondary and done through the internet, though I already have a good existing knowledge of many existing languages and frameworks.

The third stage is about assessing the capabilities of different technologies. Most of this will be done at the same time as the previous stage, as it is closely related to the technologies themselves. Again, this will be done via secondary sources on the internet.

Once this research has been done, I can move on to developing a plan and timeline for development.

As the project develops, more requirements may arise. These may introduce a need for new components to be added. These will have to be researched before they can be fully integrated. This will likely involve reading the documentation as well as conducting secondary research on sites like StackOverflow[7].

Below you can see how my time will be split during the research phase of the project. I plan for it to take a total of 3 weeks, with another week set aside for the write-up. To maximise my efficiency, I will be pursuing multiple lines of research simultaneously. Some of this will be primary research, such as interviewing teachers and students, while other parts will be secondary, such as researching the capabilities of different technologies.



[7] StackOverflow is a very well-known site within the development community. It allows people to post questions, while other developers can post answers and provide support.

End Users

For this application, there are two sets of end-users:

- Teachers & Students
- IT Support

Both sets of users will be using the application in different ways. Teachers and students will be using it to report problems they are having with technology, whereas IT Support will be using the website to communicate with users and helping to solve their issues.

Because of the different nature of the users, they will have different specific requirements, which will need to be addressed in the design of the website. In practice, this could mean two separate websites, or two different authentication systems, or just two different sides of the website. Before development can begin, I needed to talk with people from both sets of users to gain a deeper understanding of their problems and requirements.

Teachers & Students

To help understand the position of the people reporting problems, I decided to interview a teacher and two students. This should give a more detailed insight into the difficulties faced by each group of users.

Teacher – Joy Cheung

Joy Cheung is my A-Level maths teacher, and someone that I know has faced problems with IT in the past. I asked a series of questions, and I have done my best to record her answers word-for-word.

1. **How technically minded do you consider yourself?**
   I think that I'm pretty good. I use my Surface Pro in lessons to write notes on draw on the screen. I've used stuff like OneNote and Teams to share files with students and help with marking.

2. **If you have a problem with the technology you are using, who & how do you contact first?**
   I generally start by emailing James Russell (head of IT at Sidcot) and describing my problem to him. But if it's a smaller problem, I often ask one of my students to help sort it out.

3. **How long does it take to get a response / resolve the problem?**
   It depends, normally James is quite quick to reply, but it can be a day or so before he can give me any help with the problem.

4. **What if it is more urgent?**
   If it is urgent, I'll usually go and see IT Support at their office. However, quite regularly, there is no one there. This can be tricky if I'm in the middle of teaching a class and can be very disruptive.

5. **Would you know who to contact if James Russell was busy?**
   Well, I know there are a couple of guys in IT Support, but I don't know them by name, and therefore I don't know what their email addresses are.

6. **Would it be helpful to have a website where you could report problems?**
   Yes, definitely. Part of the problem I have is that I don't know what details they need to help me. I'm not particularly technical, which can be difficult when talking to the IT Support guys. If there was some kind of form I could fill in, that would be great.

Student – Cameron Wride

Cameron Wride is an A-Level student taking DT, Business Studies and Music Tech.

1. **How technically minded do you consider yourself?**
   Decent. I'm not as technical as Matt is, but I know how to use a computer, and can often help teachers with any problems they have in class.

2. **How often do you end up helping teachers in class?**
   Fairly often, but not very many teachers are that good when it comes to technology. Most of them aren't using it to a high level and losing out on lots of functionality and they usually have problems with Teams or OneNote.

3. **Who would you contact if you have a problem with technology?**
   On the rare occasion I can't figure it out, I'll normally find Matt and ask him. And on the even rarer occasion that he doesn't know the answer, I'll try and find Mr Russell, or I'll drop him an email.

4. **And if you can't find him?**
   I'll go and speak to Tom in IT Support. He normally knows how to fix it, or can at least tell me why he can't fix it.

5. **How often do you have problems?**
   Not that often, its more that if I can't connect to the WiFi, or I can't log into my school account. Most recently, I had to go and see them about my email, as I couldn't access it. If I have any problems with Teams or OneNote, they normally fix themselves after a while.

6. **Would it be helpful to have a website where you could report problems?**
   Kind of. As I said, I don't really speak to IT very much, but I think that it would be good for a lot of people I know. I guess it would make it easier for me to report problems as well. But it could be difficult, as most of my problems are to do with the WiFi, and I can't access a website without the internet.

Student – Jack

Jack is a Y10 student taking English, Maths, Science and German. He requested that his last name not be published in this document.

1. **How technically minded do you consider yourself?**
   Not too good, I often find myself struggling with Teams and OneNote. And it can be really difficult to find someone to help.

2. **What do you do if you have a problem?**
   I'll go and find one of my friends who are better at technology than I am. Usually, they know the answer, or they know who I can speak to.

3. **Do you know which staff members you could contact if you had problems?**
   I know that Mr Russell does the IT stuff, but I don't really know how to contact him.

4. **How would you feel about having a website where you could report problems?**
   That would be really helpful, it would make it much easier for me to get help when I'm having a problem. I don't really have access to my emails, so I struggle to contact people, even if I knew who to ask.

IT Support

I decided to interview James Russell about IT Support, as he oversees IT at Sidcot. I've worked with him in the past, and he seems to be quite understanding of the teachers' position, as he used to work in education.

Head of IT – James Russell

1. **How do you feel IT Support is going at the moment?**
   Well, it's significantly better than it once was, but it has quite a way to go. We've tried to set up systems in the past, but we haven't been able to get people to adopt them. Also, at the moment it's quite difficult to do anything about it, as teachers are already trying to adapt to using OneNote and Teams for Online Supported Learning[8] (OSL), and any new information we give them will likely be forgotten.

2. **What direction do you hope to head in with providing Support?**
   Over the past few years, I've looked at numerous solutions to help with IT Support, most of them are web-based. However, it has always been quite a low priority compared to the other projects. This isn't helped by how expensive the products are, as most of them are aiming towards large businesses who have maybe 20 staff working in IT Support.

3. **What is the main problem on your end?**
   Repeated request. When we have a problem with WiFi or some major system, we get lots of emails and people coming to see us saying basically the same thing. What would be really helpful is if we had a way of detecting large amounts of repeated problems, as it would let us easily see if we have a critical problem.

4. **What is the main thing you would like to see from a product?**
   Firstly, it needs to be usable for us. Many of the products I've seen have really fancy interfaces for the user, but it's basically unusable for us. Also, it would need to integrate with SIMS if it was to be a long term thing, as we would need to create and manage accounts etc.

5. **How are most problems fixed currently?**
   I often get a lot of emails from teachers that I have to forward to Tom in IT Support, as he has more time to be dealing with these issues. I'm also aware that students are crucial in helping teachers in the classroom, which really helps reduce the number of trivial problems we need to be dealing with.

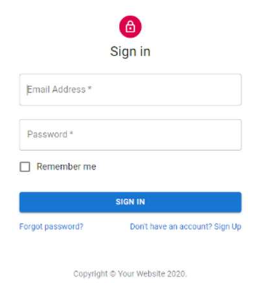
6. **Would a simple system like an FAQ help?**
   I thought it would, so we spent a lot of time creating one about a year back. Unfortunately, it didn't really do much to help, as people didn't bother looking at it before sending us emails. The main problem we have is communicating with staff, which often causes new projects like an FAQ to be forgotten about.

7. **How do you plan to improve communication in the future?**
   That's the thing, we're not quite sure. In the past, we've given training to teachers, which has helped, but most of the information gets forgotten about within a month. What would be really helpful is if we had one central website or system where all of our support goes through, which would reduce the number of stuff teachers and students need to remember.

[8] At the time of writing, the country is in the middle of the Coronavirus Pandemic. When the country went into lockdown, Sidcot provided lessons online via Teams and OneNote. This was a big shock to everyone, as we had to rapidly transition to a completely new way of working.

Project Objectives

User Management



Sign-In form from previous project

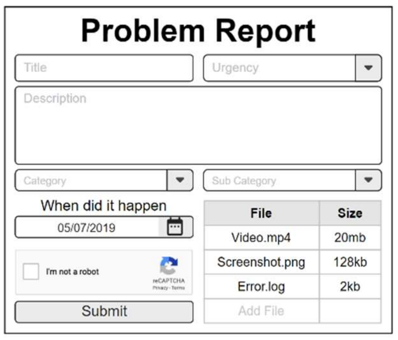
1. **Let users sign in and access their specific content**
   Users need to be able to use their school email and password to log into the website and see content specific to them. This will require a backend system to authenticate users.

2. **Multi-Level Authentication**
   The project will need to be able to determine between users and tech support members. Users will only be able to have access to certain functionality and parts of the website, whereas admins will need to have a different landing page and more access. This may need to be expanded in future to allow for more specific roles for IT Support.

3. **Password Reset & Account Recovery**
   Users will need to be able to reset their passwords and recover their accounts in case they forget their login. For this to be secure, the application will need to be able to verify the user's identity before resetting. This could potentially be achieved by using an email notification or SMS message.

4. **Two Factor Authentication**
   Two-factor authentication (2FA) works by requiring a user to complete an action on another account, typically by entering a code from an email or a phone. This provides an extra level of security, which could be used to prevent malicious password resets and similar destructive activities. However, as of writing, I have never attempted to implement a 2FA system, and it may be much more difficult than anticipated.

Problem Reporting – Teachers/Students



File	Size
Video.mp4	20mb
Screenshot.png	128kb
Error.log	2kb

Form mock-up made using moqups

1. **Title**
   The title will be used to give a brief overview of the project, allowing IT Support to quickly gain a brief understanding of the problem before addressing it. This field will mainly be text characters but may also contain numbers. Like the other text fields, this will need to be checked for escape characters[9] to prevent attacks such as SQL injections[10].

2. **Description**
   This is where teachers and students can provide a more detailed account of their problem(s). There is a whole range of characters that could be entered here, so rather than just removing escape characters, the code will need to be able to process them without causing any adverse effects.

3. **Photos/Video/Files**
   Users may want to upload files such as screenshots or video clips to help demonstrate the problem(s) they are having. These files could range from tiny error logs up to large video files. There will need to be a limit on the upload size to prevent the servers from being overloaded with data.

4. **Urgency**
   This will allow IT Support to sort to find the most urgent problems that need addressing. For the user reporting the problem, this will just be a simple dropdown field, which means there is no need to protect against malicious data.

5. **Status**
   This is a field that is filled in by IT Support when the report is received. This will allow the teacher or student who reported the problem to have a rough estimate of how long it will be until their problem is resolved.

6. **Category/Subcategory**
   These will be dropdown fields filled in by the user reporting the problem. The main use of these is for analysis and monitoring by IT Support. For example, if there were to suddenly be many reports in the email subcategory, they would be easily able to see that they may have a problem with their mail server.

7. **Messaging**
   As part of the problem report, there will be a basic two-way messaging system, allowing IT Support to easily request more information and provide better support for the problem.

[9] Escape characters are special characters recognised by the programming language; these can be used to manipulate the code without having direct access to it.
[10] SQL Injections are where escape characters are used to manipulate SQL queries to a database, which can cause damage to backend systems and data leaks.

Report Management – IT Support

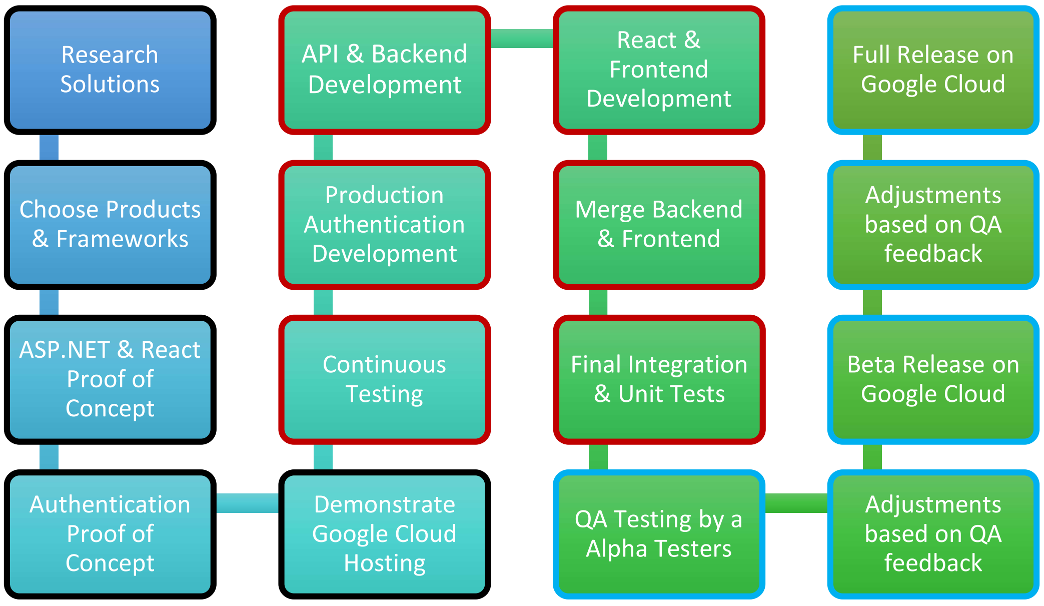
1. **View Reports**
   IT Support need to be able to view an overview of currently open reports, as well as viewing each report individually to view more information about that specific problem. The titles of the reports could easily be displayed in a list, which would provide an easy way for Support members to find specific reports. These entries in the list could be linked to another page, displaying the description of the problem, as well as any attached files or other details.

2. **Public/Private State Management**
   Members of IT Support will need to be able to change the public state of the problem, this is the value that users can see attached to their report, such as "pending", "closed", "open" etc. However, another helpful feature would be a private state only accessible to the IT Support team. This could help with cooperation between multiple support members, as it could provide more detailed information about the problem without having to read the entirety of the description. It would also help if Support Staff could attach their private notes, allowing them to save time by only reading the crucial information.

3. **Report Grouping**
   If the school were to experience a problem with its mail server, there may be dozens of reports filed regarding the issue. Not all of these will need to be addressed individually, as all their statuses are the same. To help Support Staff with this, there should be a feature to group problems together, so that their statuses can be controlled by just one master status.

4. **Messaging**
   To help with communication between the teacher or student having the problem and IT Support, it would be beneficial to have an asynchronous messaging system. This means that both users do not have to be talking at the same time, and a conversation could take place over several days as more information comes to light.

Project Timeline


Research Solutions
Choose Products & Frameworks
ASP.NET & React Proof of Concept
Authentication Proof of Concept
Demonstrate Google Cloud Hosting
Continuous Testing
Production Authentication Development
API & Backend Development
React & Frontend Development
Merge Backend & Frontend
Final Integration & Unit Tests
QA Testing by a Alpha Testers
Adjustments based on QA feedback
Beta Release on Google Cloud
Adjustments based on QA feedback
Full Release on Google Cloud

The project timeline is split into 3 distinct sections, each marked on the graphic by a different coloured outline:

- Black indicates the pre-development stage, which is primarily concerned with proofs-of-concept. These will allow me to develop a comprehensive solution to a specific problem, without having to worry about how it will integrate with other parts of the program.
- Red indicates the development state. This is where most of the programming is done, but it also includes continuous testing that will be going on throughout the project. Before the project shifts to the final stage, there is some final testing to make sure that the product does not have any critical problems.
- Blue indicates the release & testing stage. This is where the bulk of the product has been developed, and testers are brought in to check the frontend functionality and design. After testing is finished, a full release can be published to Google Cloud.

Analysis & Research

Technical Research

Frontend

Requirements

- **Dynamic-Refreshing**
  Elements of the website will need to dynamically refresh when new data is available without needing the page to reload. This prevents the use of pure HTML and instead demands a more flexible approach, such as using JavaScript.

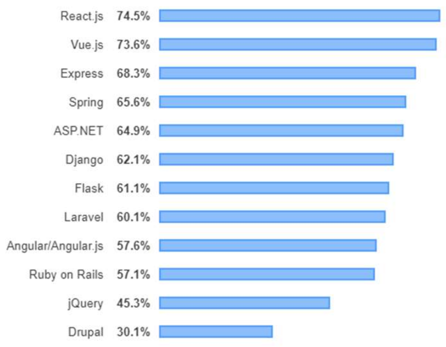
- **Single-Page**
  Part of the requirement of the website is to be simple and usable, one way to achieve this is by having a whole site as a single page application[11]. This massively reduces loading times, making the website less infuriating to use.

- **Element Functions & State**
  Some elements will need JavaScript functions to request data from the backend. This data will then need to be stored in an element's state and displayed dynamically.

Potential Solutions

ReactJS

React is a *JavaScript* framework that allows you to easily create dynamic and single page websites. It is just a framework rather than a full environment like other Angular, so it will need to be extended using plug-ins such as *Redux* to provide state-management and *React Router* for page loading and routing[12]. I have some pre-existing experience with React, which would make development quicker. React is also lightning-fast and can easily manage dynamic refreshing and component states. There are no costs associated with using React. One of the biggest benefits of using React is the knowledge base around it, according to the 2019 StackOverflow Developer Survey, React is the most popular web framework, closely followed by Vue.JS.



Angular

Angular is another *JavaScript* framework but is hugely different to React. Apart from being written in TypeScript, a variant of JavaScript, it provides a much more detailed and all-in-one approach to building websites. While React can be used in a couple of components and the rest of the website can be written in native HTML, Angular requires the entire website to be written using their tools. This includes state management, along with routing. This can be beneficial as it reduces the need for plugins, but it makes the learning curve significantly steeper. Angular does satisfy all the requirements, but due to the complexity of components, it is noticeably slower than React and Vue.

Vue

Vue.js is another *JavaScript* framework and sits in the middle-ground between Facebook's React and Google's Angular. The main benefit of Vue is the shallow learning curve, rather than having a complicated system like Angular or using JSX like React, Vue uses modified HTML along with CSS and JavaScript. Vue is also incredibly lightweight and fast but lacks some of the eco-system features of Angular. While plugins exist, they are not as robust and well documented as React's plugin library. Vue is also not as well supported by ASP.NET, a major backend web framework.

[11] A Single-Page-Application works by loading all the data for the website on the first request and then displaying the correct pages when needed. This increases the time needed to initially load the site, but eliminates delays when switching to new pages, making the site smoother and easier to use.
[12] Routing is where users are directed to different pages based on their URL.

Backend

Requirements

- **Serving Webpages**
  Pages written in the frontend framework will need to be served to the browser when requested. Depending on which framework is chosen, some routing may need to be provided to make sure the user gets the data they requested.

- **API & Frontend-Backend Communication**
  An API will be needed to provide communication from the frontend to the backend. This is necessary to send forms to the database.

- **Database Interaction**
  Forms and data will need to be stored on a persistent database; therefore, any backend system will need to be able to access and manipulate data from that database.

Potential Solutions

Flask

Flask is a *Python* library for writing backend systems for websites. The library itself is very lightweight, making it perfect for small, fast applications, but it must be extended with other plugins and libraries to provide much basic functionality. Flask can serve webpages to the front end, along with React scripts. I have lots of existing knowledge of Flask, allowing me to easily write a REST API and frontend routing. Using another Python library called *SQLAlchemy*, I can interface with various databases to pull and push data. The main downside of Flask compared to other potential backend frameworks is the lack of Server-Side Rendering[13] for React. SSR drastically speeds up website loading times, which would make the website easier to use. Flask is also an exceedingly popular library, so any problems I come across should already have Stack Overflow answers.

Django

Django is another Python backend, but compared to Flask, it is much more comprehensive. Rather than just providing library code like Flask, Django is more of a framework, providing a structured way to implement a Model-View-Controller[14] system. This is beneficial when working with databases, as Django has built-in compatibility with most SQL data sources such as MySQL or Postgres. Using the MVC framework, it would be much easier to control the flow of data between the frontend and the database. Django also manages most of the security for the website, so I do not need to worry about hashing data or sorting out SSL certificates[15]. Like Flask, Django is very popular.

ASP.NET

ASP.NET is the web framework for the .NET ecosystem. The framework is based on the C# language, one of the most popular languages currently. C# is significantly more complicated than Python but is also much more flexible. I have some basic experience of programming in C#, but it will likely be a steep learning curve. ASP subscribes to the MVC system, like Django, but provides a much more detailed method of implementing it. ASP is all open source and provides many official plugins like SignalR, which allows real-time bi-directional communication. Writing APIs in ASP is also much easier, as pulling data from databases is massively simplified. ASP also has built-in support for React and Angular.

[13] Server-Side Rendering is where part of the webpage's content is calculated by the server before it is sent. This means that the frontend needs to spend less time fetching the data, as it has already been provided as part of the request.
[14] MVC is a design system used with many backend systems. The data is pulled out of the request or database in the form of JSON and placed into the *model*. This object can be passed to the *controller* which turns the necessary data back into JSON and sends it as part of the request. A *view* is effectively a webpage the user can visit, these can be embedded with code so that when they are requested, the server renders the data that should go there.
[15] SSL certificates are a way of guaranteeing the website you are visiting is who they are claiming to be.

Data Storage

Requirements

- **Storing Forms**
  Any reports from the frontend will need to be stored persistently, which rules out just simply holding them in variables in memory. Modifications may also need to be made to the entries after they have been submitted.
- **File Storage**
  Some files may be submitted alongside error reports, these will need to be stored with reference to the problem. After a form has been closed, the files will still need to be kept (for archival purposes) but could be compressed.
- **API for the Backend System**
  A crucial requirement for the datastore is to be able to be accessed by the backend. Without this, the system is pointless.

Potential Solutions

JSON – JavaScript Object Notation

JSON is a text-based method of storing objective data. Most programming languages can serialize JSON into an Object-Oriented object[16], this data can then be easily manipulated and fed into a website. The MVC system would work fabulously for this kind of data storage. The JSON would need to be stored on a persistent disk somewhere, but this should not be an issue if working with a cloud provider like Google or Azure. I have plenty of experience dealing with JSON files, so it would be easy to get up and running. The main downside of this is continuity. Most web-based application are running many instances of their service at the same time (some use a system called Containerisation[17]), but this means that there could be many read/write operations happening at the same time. This can cause problems when working with one central file. In this situation, a database would be much more beneficial, as it can handle many I/O operations simultaneously.

MySQL

MySQL is one of the most popular SQL databases currently. Maintained by Oracle, it has plugins that provide compatibility with ASP, Django, and Flask. The main benefit of a database is that it can easily make concurrent I/O operations, allowing the system to be scaled to any size. MySQL is a relational database[18], allowing me to easily link users to their error reports. Like all the major databases, it can be hosted with any respectable cloud provider. I already have pre-existing knowledge of MySQL and its variant of SQL, so the learning curve should be very shallow.

MongoDB

MongoDB is a document-based database rather than a conventional database. Data is stored and returned in JSON syntax rather than a table like with MySQL or Postgres. This could be useful when working with the backend, as JSON is much easier to serialize into a model than data from a table is. Mongo has drivers for Python, JavaScript, and C#, so it will work with any of my backend/frontend choices. Despite being document-driven, Mongo can end up being more relational than most standard databases. I do not know how MongoDB or document-based databases work, so it may take a while to fully understand its potential.

Firebase Cloud Storage

Firebase Cloud Storage is a No-SQL method of storing structured data. This can be teamed with another Firebase product called Cloud Storage, which could easily be used to store pictures and screenshots. The biggest benefit of using Firebase is authentication. I have used the authentication system in the past, and it is second-to-none. It supports email sign-up, along with Google, GitHub, and other OAuth[19] providers. While OAuth will not be necessary for this specific project, it is certainly a benefit. Firebase also provides something known as a Real-Time database, which allows code to subscribe to changes, which would be very helpful when programming the frontend.

[16] Objects are a way of storing data programmatically. These objects can have many variables, but the most powerful aspect is that they can have functions. This is very useful when dealing with text-based data, as it can be placed into an object, which can then provide functions based on the data.
[17] Containerisation is a method of splitting a large program into smaller functions. This allows for more flexibility when hosting the project, as you can have many small instances rather than a few massive ones.
[18] Relational databases allow data from multiple tables to be linked, making it easier design objects.
[19] OAuth is a internet standard for authentication, as is provided by several major companies, such as Google and Facebook.

Authentication

Requirements

- **Create & Manage Users**
  The authentication system will need to be able to create, delete and manage users. There is also a lot of other things, such as password resets, email resets etc.

- **Store Permissions**
  Some users will have greater permissions than others. I need to be able to identify between teachers, students, and admins.

- **Backend Authentication**
  When requests are made to the backend, I need to be able to authenticate those requests.

Potential Solutions

Firebase Authentication & JWT

Firebase provides the best all-in-one authentication system that I have ever worked with. Backed by Google, the platform is under constant development. The system is far more secure than anything I would be able to design and implement. Authentication libraries are provided for every major language, along with a REST-API. I would be able to directly authenticate from the frontend, then send a JWT to the backend to authorise my requests. JSON Web Tokens (JWTs) are a 3-part base64 encoded string used to transfer authentication data. They are signed by a registered certifier and can be decoded on the backend. Firebase provides all necessary components of authentication, such as password resets and email changes.

Database Authentication

Instead of using Firebase, I could implement a custom authentication system. This is possible, but the security could easily be compromised by an amateur hacker. A custom authentication database is a highly ill-advised option for a production environment. However, for a general proof-of-concept, it would be a possibility. I have only built basic authentication systems in the past, without any kind of hashing[20] and salting[21].

[20] Hashing is an unreversible method of encryption and means that only the person who created the data knows what it is. This is a common approach for password-storage, as you do not need to keep a copy of the password, only a hash of it to compare when the user tries to login.
[21] The problem with hashing is while it is unreversible, you can easily create look-up tables of the most common passwords. This is done by taking the most common passwords, putting them through the hashing algorithm and then storing them in a database. One method the tackle this is by salting the password before putting it through the hashing algorithm. A salt is a random string of characters that is added to the end of the password. This makes it much hard to crack the password, as for each one, you need to regenerate the look-up table again.

Non-Technical Research

Pre-Existing Products

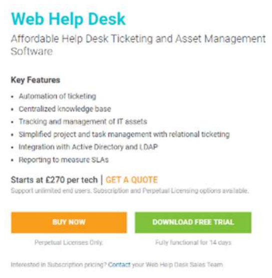
GitHub Issues

Error: "Commit tree already contains fiber 19587. This is a bug in React DevTools."

GitHub is an online VCS based on the SCM tool *Git*. However, it provides many other features, such as issue tracking. GitHub's issue tracker is not all that similar to the product I am creating, as it is more designed for open-source collaboration rather the bi-directional communication. However, the key thing I want to take from GitHub Issues is the layout. They created a really easy way to see and understand the development of a problem, allowing you to see when the status changed, who responded etc.

SolarWinds Web Help Desk

SolarWinds sells a tool for setting up help desks. It is highly customisable on both the frontend and backend, allowing companies to set up specialised ticket creation. You can easily add new users and manage support tickets. All of this is done through a web interface that is hosted on-premise.



Web Help Desk

Affordable Help Desk Ticketing and Asset Management Software

Key Features

- Automation of ticketing
- Centralized knowledge base
- Tracking and management of IT assets
- Simplified project and task management with relational ticketing
- Integration with Active Directory and LDAP
- Reporting to measure SLAs

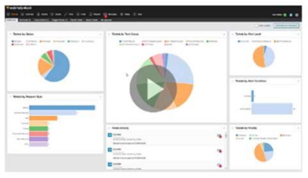
Starts at £270 per tech | GET A QUOTE

Support unlimited end users. Subscription and Perpetual Licensing options available.

BUY NOW | DOWNLOAD FREE TRIAL

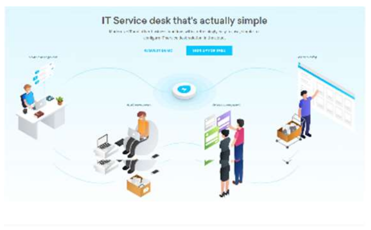
Perpetual Licenses Only | Fully functional for 14 days

Interested in Subscription pricing? Contact your Web Help Desk Sales Team.



Freshservice

Freshservice is like SolarWinds, but with a much cleaner and clearer web interface. Still highly customisable, their website is advertised as adhering to Google's *Material Design* policy, which is focused on making apps easy and intuitive to use. It is also significantly cheaper than SolarWinds but is designed for smaller businesses.



Research Conclusion & Next Steps

After considering many different methods and languages for the project, I have decided which will be most appropriate for the project. I will detail my basic ideas and go into more detail in the next section. All of the code for the project can be found at https://github.com/MattFrench019/Helpdesk

For the backend systems, I have chosen to use ASP.NET. While I have more experience with Flask, I feel that it may become seriously limiting when working with certain aspects of the website. One of the notable problems I have had with Flask in the past has been trying to determine whether to pass a request to React, or whether to send it to an API endpoint in my code. ASP.NET manages all this routing for me, making it much easier to work with frontend routing, as it just works rather than requiring a lot of setup.

I have decided to use React for the frontend, as it just provides so much flexibility when it comes to writing websites. This mainly comes down to its functional approach, treating different elements as individual components, allowing me to attach code to them much more easily than if I was using pure JavaScript. I have worked with React in the past; I already know my way around many of the most important libraries.

The authentication system is a bit more complicated. I plan to use Firebase to provide the authentication, but the trouble is having different permissions. There are ways of achieving this, namely by linking the authentication system to Firebase's Realtime-Database. However, this is very challenging and is currently beyond my level of React knowledge. Instead, I plan to use two different login systems, which will allow admins and users to be authenticated separately. I chose Firebase because of how easy it is to use, and the only real alternative was to write an entirely custom authentication system, which is well beyond the scope of this project.

For the database, I will be using MySQL. This was an easy choice as it is an industry-standard tool and one of the few databases that I have existing experience with. I had hoped to use MongoDB, but the amount of time needed to set it up and learn it was well in excess of what I had hoped to spend on a relatively small section of the overall project.

Design-Development

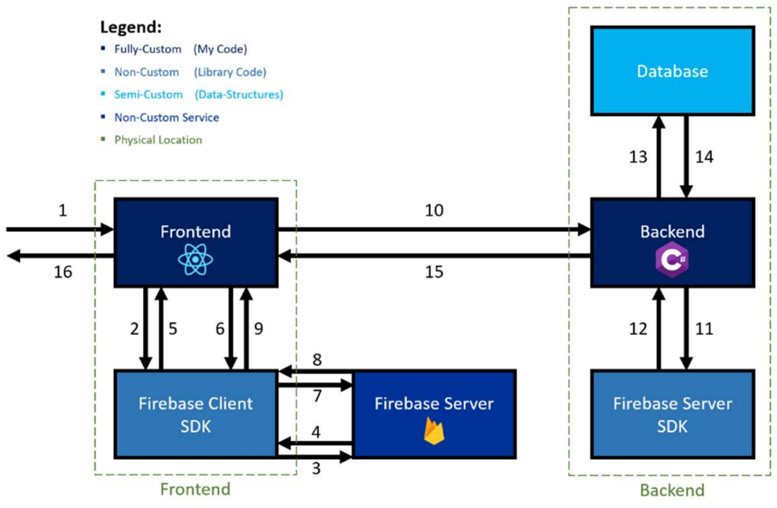
Authentication Flow

Authentication is a massive part of this project, so I fully expect the approach to change throughout the project. At the time of writing, the current approach is to use two separate Firebase projects, however, this may prove to be complicated and cumbersome. A potentially better approach may be to use the Firebase Realtime-Database to store permissions and various other properties.

The diagram below shows how the authentication will be handled for each system. It was created before the two-project approach was devised but is still accurate. However, it does miss off how the frontend will choose how to route each request, as this has still not been figured out.

A lot is going on in the diagram, as well as the more specific steps shown next to the diagram, this will be a quick walkthrough of a request:

> To start with, a user inputs their data and presses the submit button. This data is then interpreted by the frontend and fed into a function provided by Firebase's SDK. The SDK is a library of code provided by Firebase to help developers interact with their services. Firebase then sends a request to their authentication server to verify the user and will then return a user object back to my code. Using that UID, we can then call another function on the SDK to generate a JWT. This is a long string consisting of 3 parts: the type, the data, and the signature. It is basically a secure way of authenticating a request[22], as they cannot be faked due to public-private key encryption[23]. This JWT is then attached to a request sent to my server, which is interpreted and verified using the backend SDK. If the JWT is valid, I can be sure that it has been issued by *my* frontend and no one else. This allows me to have 100% confidence in the identity of the user before sending them any user-specific data.



Request Stages:

1. User inputs their data
2. Data sent to Client SDK to authenticate
3. *Authenticate with Firebase Server*
4. *Authentication Response*
5. Client SDK creates a logged-in user object
6. Request JWT from Client SDK
7. *JWT creation*
8. *JWT Response*
9. JWT returned to code
10. Request sent to backend with JWT
11. *Authenticate JWT*
12. *Authentication Response*
13. Any necessary data requested from database
14. Data returned
15. Authenticated Response to Request
16. Frontend updates the interface

Firebase hidden requests are marked in italics

[22] More details at https://en.wikipedia.org/wiki/JSON_Web_Token
[23] Public-Private key encryption is a way of securing data by using 2 keys. It allows messages to be encrypted and easily decrypted, while being incredibly difficult to

User-Interface Design

The main goal of the UI design is to be easy to use. To that end, I have decided to strictly follow the Material Design philosophy. This is a concept that was developed by Google to provide UI designers and developers with a rule book on how to build responsive websites and applications. Its main focus is usability, which fits well with the scope of my project. Material Design covers many different areas, from colour to animation, all based around making applications behave predictably.

I am sure most people have had the frustration of using a website when a simple action takes 10 clicks, or when a progress button is red instead of green. These are basic problems that many websites suffer from due to bad design. For my project, I desperately want to avoid these pitfalls as much as possible.

User Homepage

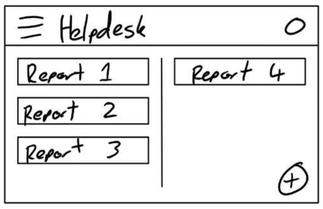
This will be the most used page, so the design must be as intuitive and simplistic as possible. Its goal is to provide users with an overview of their pending and completed reports, as well as giving them an option to create a new report.

The page will be split up into two columns, each containing either pending or completed reports. Material Design specifies that logic should flow from left to right, so the pending reports should be on the left and the completed ones on the right. This is the same logic as those "before and after" photos, subconsciously you expect the "before" picture to be on the left and the "after" one to be on the right.

Each report will have a small overview, likely containing the name, a brief description, and the urgency. I am planning on using a traffic-light system, where red means that the problem is awaiting support, amber means it is currently being supported, and green means it has been closed.

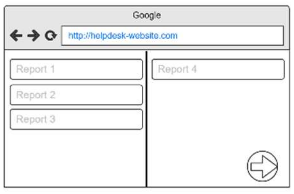
Early Prototypes

The prototypes on the left are quick sketches I made in OneNote just to demonstrate how the page would look. The one on the right is a more formal design I made using Moqups. I showed these designs to a small selection of people for their feedback. Most of the responses were positive, except for a few people who did not like how far away the "add" button was from the rest of the page.



↑ Early Sketch

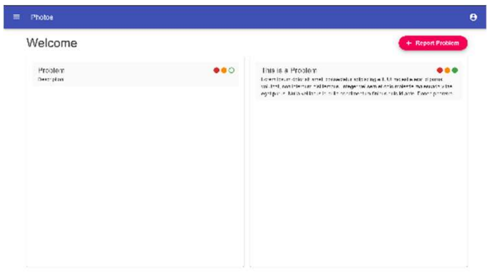
Title
Description...
↑ Problem Overview



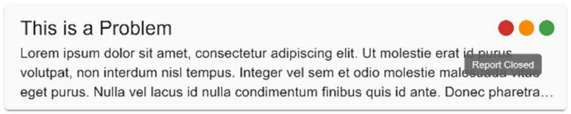
↑ Mock-up of GUI

Mid-Development

Based on the feedback I got earlier in the project, I moved the "Add" button to the top right of the screen, where it was noticeably more visible. I also used CSS to adjust the colours and designs of the different components to make them fit more with the Material Design philosophy. These screenshots are from dummy data, as the page has not yet been integrated with the database. I used different shades of whites and greys to indicate the different components, as well as adding a shadow to the report descript when the user hovers over it. These kinds of changes help make the website more responsive and intuitive.



↑ User Homepage


This is a Problem

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut molestie erat id purus volutpat, non interdum nisl tempus. Integer vel sem et odio molestie malesuada vitae eget purus. Nulla vel lacus id nulla condimentum finibus quis id ante. Donec pharetra…

↑ Problem Overview

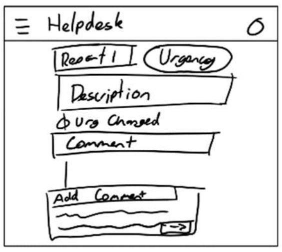
View Report Page

This will also be a very commonly used page, so the aim is to make it as simple as possible. After researching other websites and issue-trackers, the design I liked the most was GitHub's issue tracker. While it is not perfect, it provides a very understandable way for a user to see the timeline of their issue as well as any comments or changes that have been made.

The page will roughly follow a vertical timeline, consisting of comments and changelogs, such as when a support member changes the status of the issue. Each of these likely be a "card" which is a Material Design term for a component with a different z-height[24]. This timeline will start with the oldest events at the top and as the user reads down the page they will get to the newer events.

Each card will have the comment as well as the username of the person and the time and date it was posted at. This will help users see who is providing support as well as how long it has taken. At the bottom of the page, there will be a card that will contain a field for adding a new comment. This will be how users reply to the support team, as well as how the support team provides help for the user.

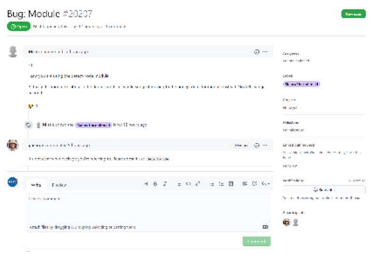
This new model of comments may change how I plan to store data on the backend, as rather than using a static class object or plain JSON, I will likely end up using a list of smaller comment objects, each with their own data structure. These will then be parsed by the frontend and displayed to the user.



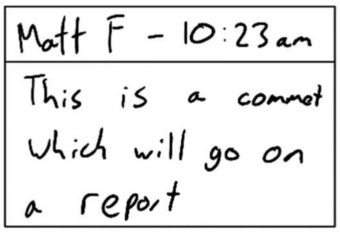
↑ *Early Sketch*

Early Prototypes

Again, these are a collection of rough designs from the very start of the project. The sketch on the left is what I intend the page to look like. It follows the same design as GitHub's issue tracker, which can be seen underneath the sketch. As I mentioned earlier, the timeline flows from top to bottom, which makes it intuitive for users to understand. The other sketches are rough ideas of what a comment will look like as well as the card for adding a comment at the bottom of the page.



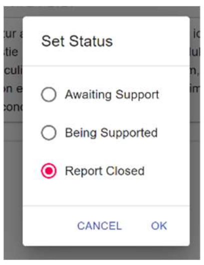
↑ *GitHub Issue Tacker*



↑ *Example Comment*

Add Comment
Description...
Add

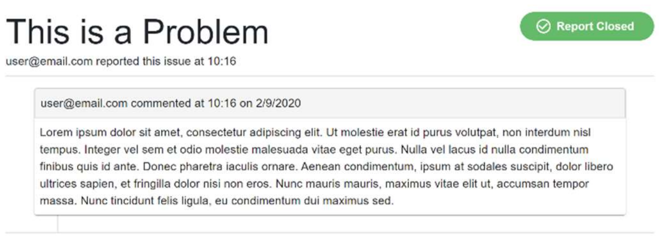
↑ *Add Comment Box*



↑ *Status Drop-Down*

Mid-Development

At the time of writing, the comment functionality has not been implemented, so these images are missing the "Add Comment" box which will be placed at the bottom of the screen. However, all other functionality works, including the usernames and the time the comment was created. Also, if the user has admin permissions, they can click on the "Status" button to change the status of the report by opening a drop-down menu. All the data is being pulled from the database and is not hardcoded.



This is a Problem

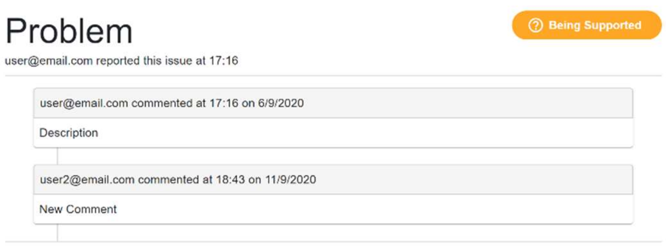
Report Closed

user@email.com reported this issue at 10:16

user@email.com commented at 10:16 on 2/9/2020

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut molestie erat id purus volutpat, non interdum nisl tempus. Integer vel sem et odio molestie malesuada vitae eget purus. Nulla vel lacus id nulla condimentum finibus quis id ante. Donec pharetra iaculis ornare. Aenean condimentum, ipsum at sodales suscipit, dolor libero ultrices sapien, et fringilla dolor nisi non eros. Nunc mauris mauris, maximus vitae elit ut, accumsan tempor massa. Nunc tincidunt felis ligula, eu condimentum dui maximus sed.

↑ *Example Problem*



Problem

Being Supported

user@email.com reported this issue at 17:16

user@email.com commented at 17:16 on 6/9/2020

Description

user2@email.com commented at 18:43 on 11/9/2020

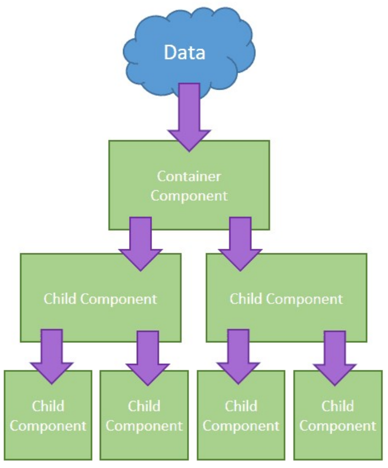
New Comment

↑ *Problem with Comments*

[24] Material Design uses the term "z-height" to indicate how big the shadow underneath the component is. This allows users to easily tell components apart.

Manager Context

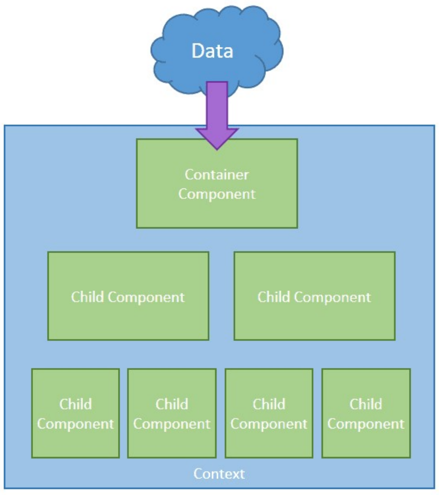
In React, a webpage is built up of many components, each with their own pieces of code and call-backs that allow them to function properly. These components are assembled into a tree, much like a folder structure. This is done to split up the functionality of the webpage into reusable sections, such as buttons or a drop-down list. Data can be passed down the component tree by using parameters, which can be given to a component when it is rendered. However, this can become cumbersome if data needs to be passed down many levels.



↑ *Dataflow without Context*

In my project, there will need to be many shared functions, needed for making requests or authenticating, and a substantial amount of shared objects/data, such as the database adaptor[25] and Firebase object. I plan to put all this shared data into a *Manager* class. This is an approach typically used in game design, where a single component is responsible for allowing various parts of a program to interact. The *Manager* object will then need to be provided to all components that need it. This is where a context comes in.

In React, a *Context* is a method of providing data to lower-level components, without having to pass it down multiple levels using parameters. It consists of 2 parts: the provider, and consumers. The provider is a component that is used to wrap the entire component tree, allowing consumers within its scope to access the data passed to the provider. A consumer is then used to wrap the component that needs access to the data, it takes the data from the provider and inserts it as a parameter into the wrapped function. This wrapping process can be done with a function that takes a component as a parameter and returns the component wrapped with the consumer.

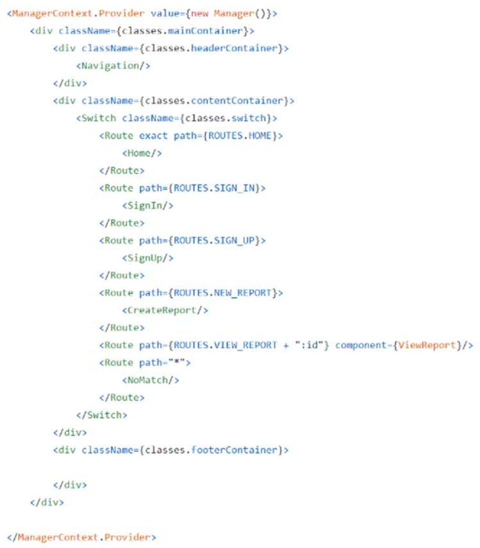


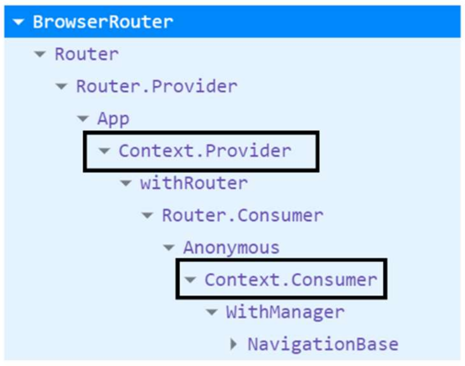
↑ *Dataflow with Context*

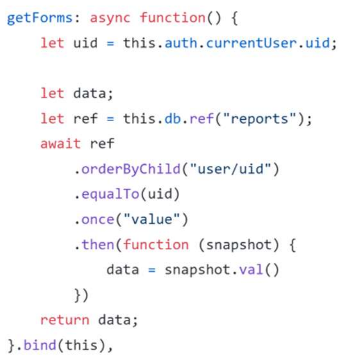
This will be expanded on in the development section, but if the explanation did not make sense, there is also a set of diagrams, demonstrating how a *Context* speeds up rendering and development.

In the diagram below and to the left, you can see how a basic example of how *Contexts* can be used to pass data down multiple levels. This is a component view of a small segment of my project, specifically the Navigation Bar at the top of the screen.

To the right, you can see a snippet of the component tree that is used to render my application. The top-level component is the *Context* provider, meaning all of my components can access the *Manager* object if they are wrapped with a *Context* consumer.


```
<ManagerContext.Provider value={new Manager()}>
    <div className={classes.mainContainer}>
        <div className={classes.headerContainer}>
            <Navigation/>
        </div>
        <div className={classes.contentContainer}>
            <Switch className={classes.switch}>
                <Route exact path={ROUTES.HOME}>
                    <Home/>
                </Route>
                <Route path={ROUTES.SIGN_IN}>
                    <SignIn/>
                </Route>
                <Route path={ROUTES.SIGN_UP}>
                    <SignUp/>
                </Route>
                <Route path={ROUTES.NEW_REPORT}>
                    <CreateReport/>
                </Route>
                <Route path={ROUTES.VIEW_REPORT + ":id"} component={ViewReport}/>
                <Route path="*">
                    <NoMatch/>
                </Route>
            </Switch>
        </div>
        <div className={classes.footerContainer}>

        </div>
    </div>

</ManagerContext.Provider>
```
↑ *Component Tree*


```
▾ BrowserRouter
  ▾ Router
    ▾ Router.Provider
      ▾ App
        ▾ Context.Provider
          ▾ withRouter
            ▾ Router.Consumer
              ▾ Anonymous
                ▾ Context.Consumer
                  ▾ WithManager
                    ▸ NavigationBase
```
↑ *Context Demonstration*


```
getForms: async function() {
    let uid = this.auth.currentUser.uid;

    let data;
    let ref = this.db.ref("reports");
    await ref
        .orderByChild("user/uid")
        .equalTo(uid)
        .once("value")
        .then(function (snapshot) {
            data = snapshot.val()
        })
    return data;
}.bind(this),
```
↑ *Shared Function*

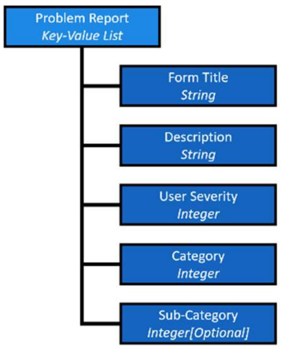
[25] A database adaptor is typically an object or class that can be initialised to connect to a database, and also has functions that allow programmatic access to said database.

Data Structures

Report

This data structure will hold information about a problem reported by a user. When the user logs in and accesses their homepage, a request will be made to the server which will return a list of these objects. As the project evolves, some of the details of this object will change to match future requirements. At a basic level, the data structure needs to be able to store:

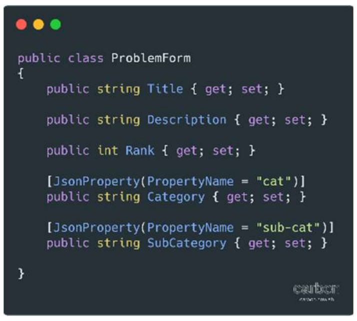
- User who created it
- Current Status
- Title/Short Description
- Longer Description



↑ *Data Visualization*

Early Plans

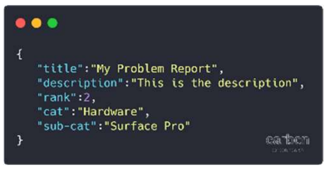
The initial idea is to use a JSON object with various properties. This can then be interpreted into a JavaScript or C# object using various libraries. This would then allow me to use this data within my project. Functionality like this is provided as part of ASP.NET, but for JavaScript, I would have to use an external library.



```
public class ProblemForm
{
    public string Title { get; set; }

    public string Description { get; set; }

    public int Rank { get; set; }

    [JsonProperty(PropertyName = "cat")]
    public string Category { get; set; }

    [JsonProperty(PropertyName = "sub-cat")]
    public string SubCategory { get; set; }

}
```

↑ *ASP.NET Model*

On the right, there is a small data structure diagram I have created to show the different properties of the object. This will then be stored as JSON, an example of which can be seen under the diagram. When the frontend makes a request, the JSON it passes will then be interpreted using the code on the left which will turn it into a C# object which can then be used to insert into a database.

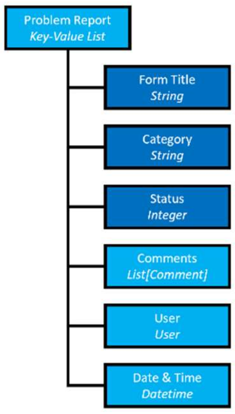


```
{
   "title":"My Problem Report",
   "description":"This is the description",
   "rank":2,
   "cat":"Hardware",
   "sub-cat":"Surface Pro"
}
```

↑ *JSON Object*

Revision #1 – Comment System

As the project progressed, it became necessary to revise how problems were stored at a low level. When the frontend goes to render a report, it creates a small textbox for each comment and places various bits of information in them. To make the frontend code simpler, I adjusted the structure.



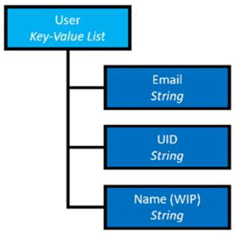
↑ *Revised Report*

The title, status and category all remained the same, but I added a new "Comments" field which stores a list of comment objects. There is also a new "User" and "DateTime" object, which I will explain in the next section.

Comment

The comment object is a representation of a comment that may be posted on a report. It will also be used to hold the description and other properties that were originally part of the report object. With only 3 fields, it is very simple and only holds the actual comment, the user who created it, and the time it was created.

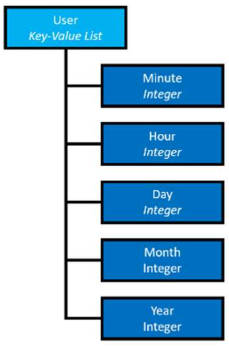
User



↑ *User Datatype*

This object stores a representation of the user who created a comment, some of this data is for internal reference, such as the *unique identifier* of the user. However, some fields will be used to display various properties of the user on the report page. At the time of writing, there are only 2 fields, the email of the user and the UID. This will likely change to include the name of the user as well.

Datetime



↑ *Datetime Datatype*

The DateTime object is used to store a programmatically neat version of the time and date. This time can refer to several things, but for this example, we are using it to represent the time and date when a comment was created. It splits the time and date down into individual fields, allowing me to easily change the formatting of the date and time on various pages, without having to update all of the DateTime objects in the database.

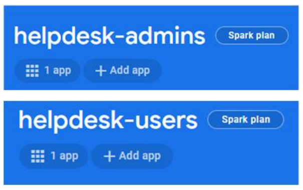
Development

Authentication

Overview

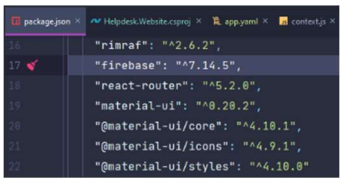
Before other systems can be implemented, I need to set up the authentication system. The plan is to set up the Firebase projects, then write a Context for React that with provide.

Setting Up Firebase

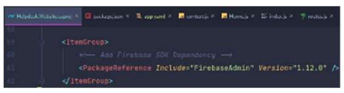


I already have a Firebase account linked to my personal email. With the free account, almost all the authentication API is provided, except the SMS system. To start, we set up two Firebase projects (right).

For both of these projects, I then enabled the *Email and Password* sign-in option. On a less important note, I created a user for each system, to allow me to test the authentication system.



```
"rimraf": "^2.6.2",
"firebase": "^7.14.5",
"react-router": "^5.2.0",
"material-ui": "^0.20.2",
"@material-ui/core": "^4.10.1",
"@material-ui/icons": "^4.9.1",
"@material-ui/styles": "^4.10.0"
```



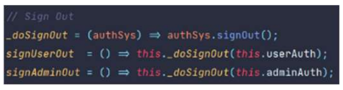
```
<ItemGroup>
    <!-- Add Firebase SDK Dependency -->
    <PackageReference Include="FirebaseAdmin" Version="1.12.0" />
</ItemGroup>
```

After the projects were set up, I needed to install the SDK for both the frontend and backend. For React this is as simple as listing the Firebase package in the *package.json* file (top left). For ASP.NET, the dependence must be listed in the *.csproj* file (left). After running some CLI commands for *npm* (Node package manager) and *dotnet*, all dependencies were installed.

To finish the setup for the backend, I had to install two key files to be stored. These allow the backend to establish connections with the Firebase servers. I had to do a similar thing for the frontend, I stored the access tokens for the Firebase projects into *.env* files, which are then pulled into React when the application runs.

Writing the Context

As mentioned earlier, a React Context is a component that providers global values to smaller components. The *manager context* (as I will now refer to it), holds a *manager object*, which is a class with many functions on it. These functions will provide a way for other components to talk to the authentication server or API without having to do the complicated logic.

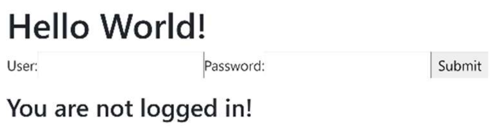


```
// Sign Out
_doSignOut = (authSys) => authSys.signOut();
signUserOut  = () => this._doSignOut(this.userAuth);
signAdminOut = () => this._doSignOut(this.adminAuth);
```

The basic format for most functions is that there is a private function (marking with a leading underscore) that takes the necessary parameters as well as an *authSys* variable. There is then a function for each specific authentication system (either admin or user), which calls the private function with the arguments passed to it, along with the appropriate authentication system. For the example of signing out, see left.

The code for the manager at the time of writing can be found at the following link:
https://github.com/MattFrench019/Helpdesk/blob/9675dabacfd40afb02636f96bef9d65d71598a12/Helpdesk.Website/ClientApp/src/components/Manager/manager.js

Writing the Form



Next, I need to write the form for the sign-in page. As this is a proof of concept, it does not need to be flashy. A basic HTML and CSS form will suffice for this, wrapped by a React component. This component will keep track of field values and will call the *manager object* with the entered email and password. You can see the form to the right.

This form will be revisited later in the development process.

The code for the form at the time of writing can be found at the following link:
https://github.com/MattFrench019/Helpdesk/blob/a5eb78d2fc872fa528d0338675ebb754b415499b/Helpdesk.Website/ClientApp/src/components/SignIn/SignIn.js

Compiling & Deployment

Important Design Change

This is where it got tricky. I had written the application based on the .NET 3.1 SDK, which is not currently (June 2020) supported by Google App Engine. I spent several hours trying to create custom runtimes with .NET 3.1 Core using Dockerfiles[26]. Eventually, after no progress, I abandoned 3.1 and decided to rewrite the project in .NET 2.2. This was more difficult than I had expected, but only took a couple of hours.

Once the project was rewritten, I had to compile the .NET app and the React into a production version. Before this could happen, I needed to create an *app.yaml* file which defines how my app with be hosted on Google Cloud. This file needed to be specified in the *.csproj* file, as I need it to be copied to the build directory. I then ran the command:

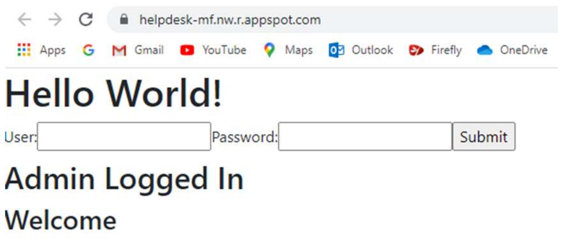
dotnet publish -c Release

This compiles everything to an output directory. After this happens, I need to publish to code to Google App Engine. This is done using the command:

gcloud app deploy

This command takes the code in the directory and uploads it to a Virtual Machine in one of Google's servers. It also sets up load balancing and redirects traffic to the new version.

Eventually, after many hours, the project was finally hosted on Google Cloud.


helpdesk-mf.nw.r.appspot.com

Hello World!

User: Password: Submit

Admin Logged In

Welcome

Continuous Testing & Deployment

Overview

CT & CD are practices that are rapidly gaining popularity in the software development world due to their benefits when it comes to working in teams. The idea is that when you create a release on GitHub etc, you have scripts set up to run unit and integration tests on the code and then automatically deploy it to a hosting provider or local server.

For me, this means that after testing the application locally to make sure it runs, I can create a GitHub release that will take the source code, compile it using the *dotnet publish* command, then deploy it to Google Cloud.

This will save me a vast amount of time in the long run, but it can be difficult to set up in the short term.

Precautions & Safety Measures

To prevent deployment of non-working code, there must be several precautions put in place:

1. Test code locally before releasing on GitHub
2. Run unit[27] & integration[28] tests before deployment to check code functionality
3. Prevent deployment if errors during compiling
4. Deploy with *-no-promote* command flag, which makes Google Cloud keep the old version of the code running until it is manually disabled. This allows for code-rollbacks if the code does not work on Google Cloud specifically.

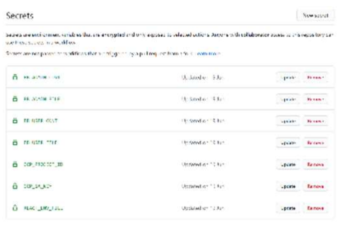
Other safety measures need to be put in place to prevent the leakage of sensitive API tokens. Due to the use of Firebase APIs, I need to store certain tokens to allow for authentication. These tokens **MUST NOT BE LEAKED** under any circumstances, as they can then be used to access the authentication system. These tokens are stored in JSON and Env files, which are currently just listed in the *.gitignore* file. These will need to be stored as GitHub secrets.

[26] Docker is a system that allows virtual machines to be created based on a file, known as a Dockerfile.
[27] Unit tests focus on testing very specific parts of the program, such as an individual function.
[28] Integration tests focus on a larger section of code, such as a whole module.

Development

GitHub Actions is a system provided by GitHub, my Version Control System, to help with the Continuous Development process. To set up the Continuous Deployment section of the pipeline, I need to write a script in YAML which GitHub can then interpret. I started by researching how GitHub Actions work, mainly by looking at other examples of CD scripts and reading the documentation.
The script will have 3 main tasks:

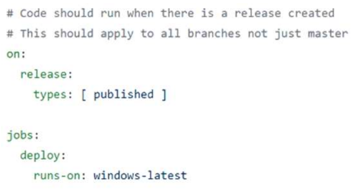
1. Insert the secret API tokens into the code
2. Build the ASP.NET app into an executable
3. Deploy to the Google Cloud



The first job was to add my API secrets to the secure part of my GitHub project. This was relatively simple and just involved me copying the data from my computer and adding it as GitHub Secrets. I gave each Secret an obvious name that would allow me to access them from my CD scripts.

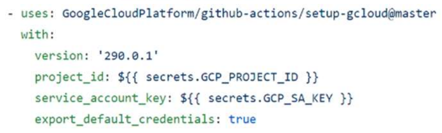
The next task was to set up API access from the Google Cloud, which would allow me to upload files from GitHub. To do this, I just had to log into my Google Cloud project and generate an API Key. I put this and the project id into Secrets.

Once the secrets were added, I could begin to start with the development of the CD script. The first task was to define what operating system the script should run on, and when it should run. For this script, I want it to run on Windows so it can use the dotnet utilities, and I want it to run when I create a release on GitHub.



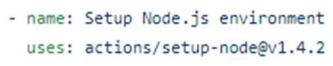
```
# Code should run when there is a release created
# This should apply to all branches not just master
on:
  release:
    types: [ published ]

jobs:
  deploy:
    runs-on: windows-latest
```

I next needed to set up the gcloud utility, which allows me to interface with Google Cloud through a command-line script. To do this, I found a GitHub action that someone had already created. This is like using a library in a piece of code. It works by taking a few options and will then install gcloud on the virtual machine.



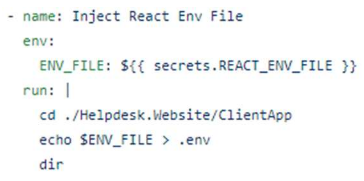
```
- uses: GoogleCloudPlatform/github-actions/setup-gcloud@master
  with:
    version: '290.0.1'
    project_id: ${{ secrets.GCP_PROJECT_ID }}
    service_account_key: ${{ secrets.GCP_SA_KEY }}
    export_default_credentials: true
```

To finalise the setup, I just need to install Node.js. This is used to compile the React source code into a production version. Out of all the build stages, the JavaScript takes the longest to compile, often around 3-4 minutes compared to the 50 milliseconds needed for the backend.



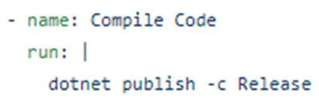
```
- name: Setup Node.js environment
  uses: actions/setup-node@v1.4.2
```

Next, we can start to add the files. To add the React *.env* file, we read from the GitHub secret and pipe it into a file. This is simple and can be achieved with a few common CLI commands. Then we need to add the Firebase files, which is done by creating a new directory and then piping the secret into a file.



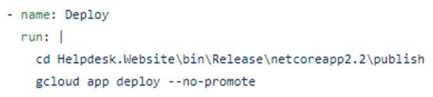
```
- name: Inject React Env File
  env:
    ENV_FILE: ${{ secrets.REACT_ENV_FILE }}
  run: |
    cd ./Helpdesk.Website/ClientApp
    echo $ENV_FILE > .env
    dir
```

We then run the command to compile the code into a build, this is done with the *dotnet* utility, which takes the source code and turns them into a collection of DLL and EXE files. Because of the type of project, this will also call React's compile command, which is defined in *package.json*.



```
- name: Compile Code
  run: |
    dotnet publish -c Release
```

Finally, we deploy the code to Google Cloud using the *gcloud* utility. When I do this, I use the *–no-promote* flag, which tells Google not to push any traffic to the new code. This allows me to test the new build before it is pushed to the website.



```
- name: Deploy
  run: |
    cd Helpdesk.Website\bin\Release\netcoreapp2.2\publish
    gcloud app deploy --no-promote
```

Testing

To make sure that my script worked properly, I created a release on GitHub and attached my source code. I then had to wait for the system to pick up the job, which takes about 10 minutes as I am currently using the free version of GitHub Actions.

Once it started, it took about 20 minutes to fully compile the source code into a production-ready version, most of this time was taken up by React's build script, which is painfully slow. After the code had finished injecting the secrets, it began to upload the application files to Google Cloud – which failed. In the time between writing the script and then testing it, Google had disabled my API key. This was not a major issue, as I just had to regenerate the key and read it to the GitHub Secrets.

When I then reran the script, everything worked perfectly. The new application version appeared separately on Google Cloud, rather than overwriting the pre-existing version, which was exactly the behaviour I wanted. This then gives me a chance to test the new build before pushing it to production. For most major applications, this is how Continuous Development is achieved, as the application still needs extensive testing before being rolled out fully. This is why companies may have open Alpha/Beta releases where users can test the application.

Firebase Permissions

Overview

Important Design Change

After trying to plan how my code would work with the current Firebase system, I decided to change how I handle permissions and authentication. Previously, I had used two separate Firebase projects to authenticate users. Now, I wanted to change to using just one system but backed up by a database. The easiest option for this database was the Firebase Realtime Database, which has some very useful features such as linking directly in with the authentication system and also providing a system of call-backs to allow my UI components to change dynamically when new data becomes available.

At the same time, I will need to change how my routing system works to allow users to only access pages specific to them, as well as updating my Sign-In and Sign-Up forms to cope with the new system.

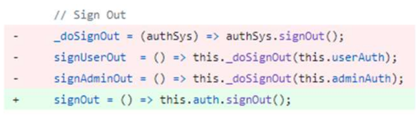
I fully expect this code to change drastically throughout the project, which means I will need a way of abstracting the functionality of the system into a *manager* component, which will mean I only need to change code in one place rather than all over the application.

Development

Manager

The code for this change can be found at:
https://github.com/MattFrench019/Helpdesk/commit/f6a245ade04297d3408c3dc1ef75d242d559316e#diff-821f6378efbfbb4ee46065ea53761578c93cb8fb926bb52ad52eab0439f61061

To start with, I needed to adjust the existing Manager component to work with the new system. Most of this just involved simplifying the existing code, as I now only need one function for each piece of functionality rather than two different ones.



```
      // Sign Out
-     _doSignOut = (authSys) => authSys.signOut();
-     signUserOut  = () => this._doSignOut(this.userAuth);
-     signAdminOut = () => this._doSignOut(this.adminAuth);
+     signOut = () => this.auth.signOut();
```

I idea of these small functions like *signIn* and *passwordReset* is to abstract as much functionality as possible into the Manager, and other components will just call these functions, without having to worry about how the system is working behind the scenes. In theory, this will mean that in future, I can completely change how the system works without needing to change the individual components that use the functions.

I also created a function that will allow components to subscribe to the current user. This is crucial to allow components to be able to see data related to the current user. Without this, the application would not be able to tailor the content for the current user.

withManager

The code for this can be found at:
https://github.com/MattFrench019/Helpdesk/commit/f6a245ade04297d3408c3dc1ef75d242d559316e#diff-209f64e90238f01993da437cdccab639fa9a80a9ff58b667cfdb4fca26008c9f

The *withManger* function works by taking another component and providing a reference to the Manager through a property on the child component. This system is known as using a *context* in React. More details can be found at https://reactjs.org/docs/context.html. The basic idea is that rather than needing to pass down a property through the whole component tree, you wrap the component where you want to use the property with a context consumer.

This context provider does two major things:

1. It provides the child component with access to the Manager class, and therefore all of the functionality of that object.
2. Provides the current user as a property that the child component can access

The benefit of providing the current user as a property is that I've created another layer of abstraction. This means that I will only need to change code in my *withManager* wrapper rather than in each component.

This is deemed bad practice in React, as components should ideally subscribe to the current user individually, as this allows them to add their own callbacks, rather than having to re-render the entire component tree when a new user logs in. However, this will work well in this situation, as I don't plan for this to be a permanent solution, it just means that if I need to change the code between now and the end of the project, it reduces the probability of me breaking other parts of the code. Also, normally when a new user logs in, the whole tree has to be re-rendered anyway, so I am not necessarily slowing down the application by using this approach.

withAuth

The code for this can be found at:
https://github.com/MattFrench019/Helpdesk/blob/f6a245ade04297d3408c3dc1ef75d242d559316e/Helpdesk.Website/ClientApp/src/components/Manager/withAuth.js

The *withAuth* component can be used to wrap another component. The goal is to only allow users with the correct roles to access a webpage. It works by subscribing to an event provided by the manager component. This is called when the *currentUser* property provided by Firebase changes. It then uses the user returned by the event to determine whether to render the component. It does this by using a condition passed to it when it is created. These conditions are stored in a constants file, where new ones can be added without changing the actual code.

Once this code was written, some of the existing components were wrapped with it. These included the home page which was wrapped with the condition that a user must be signed in.

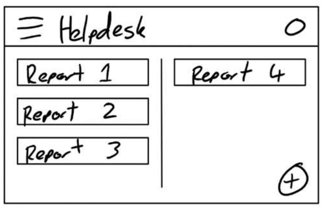
User Homepage, View Report & Create Report

Overview

In this stage of development, the aim was to develop and complete a homepage where users could see an overview of their reports. This also includes supporting features such as the ability to create a new report. It needs to be easy to navigate and immediately understandable, as it is the first page users will see after logging in.

Development

Homepage

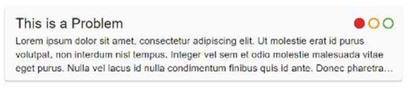


As mentioned earlier in the project, I wanted the homepage to look like the diagram on the left, with two columns each containing reports that were either in progress or completed. Each report would have a small button that could be clicked to view it in more detail. This button will have some details such as the title and the first part of the description.

To achieve this, I will need to create a homepage component that will be responsible for fetching the data from the data source and placing it into the smaller components. I will also need to create a component for the report button as well as a button to create a new report.

To start the development, I programed the title and action button at the top of the page. I moved the button on advice from several people, as it seemed like it was quite far away from the main page content. This was done by using a grid to separate the title text and button. The button was then modified from a component provided by the component library I'm using. This was done using CSS as well as properties defined within react.

I then created the columns using a "paper" component provided by the component library. Within React, I defined a piece of code to loop through the fetched data and place each report into the report button component.

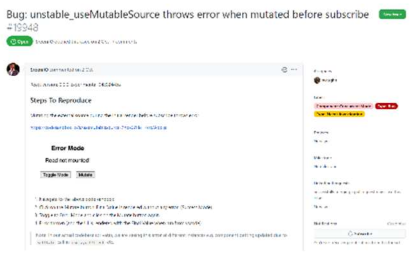

This is a Problem

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut molestie erat id purus volutpat, non interdum nisl tempus. Integer vel sem et odio molestie malesuada vitae eget purus. Nulla vel lacus id nulla condimentum finibus quis id ante. Donec pharetra...

The next job was to create the report button component. The main functionality of this component is to take the data provided by the homepage and display it in a small button-like object. As well as this, it needs to route the user to the correct report when clicked on.

While the formatting of this object is likely to stay the same, the data structure for the report may change, which will mean I will need to revise this component. When writing the component, one particular consideration was how data should be passed from the homepage to the button. This could be achieved by providing the entire form data structure as a property, and all the formatting could be done by the button. However, this is regularly considered bad practice in React, as it often leads to more data being passed around than necessary, leading to slower loading times. Instead, I've chosen to only pass down important variables of the data structure, such as the title, status and description.

View Report


Bug: unstable_useMutableSource throws error when mutated before subscribe
#19948

Steps To Reproduce

Error Mode

Read not mounted

Toggle Mode Mutate

When the user clicks on one of the buttons, they will be directed to a page that displays a full description of the problem, along with any associated data, such as dates and comments. The plan for this page is to have a component that receives all requests to pages with the URL "/report/view/<id>". It then will take the URL and find the report's id, then makes a request to the database to find the data associated with that id.

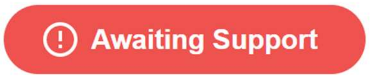
As mentioned earlier in the project, I would like this page to resemble the GitHub Issues page, which is based on a timeline of comments and changes (see left). I find this a particularly easy way to ingest information about an issue and its progression to being solved.

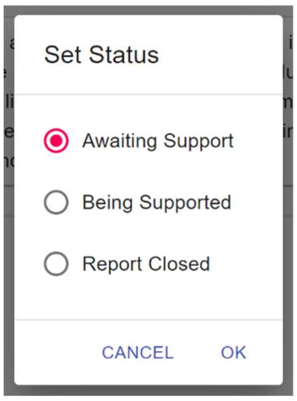
```
<Route path={ROUTES.VIEW_REPORT + ":id"} component={ViewReport}/>
```

The starting point of this page was to sort the routing out. React supports routing based on URL parameters, but advises against it due to the single-page design philosophy. To make this easier, I set up the switch component[29] to pass all requests which conformed to the "/report/view/<id>" schema to the *ViewReport* component.

When the component runs, it pulls the URL from React Router and breaks it down by backslash characters. It then selects the last element of the list and uses that for the id of the form. After that, it uses the id to make a request to the *Manager* component, which then requests the data from the database.

Currently, the *Manager* takes a snapshot of the database and then pulls the necessary form out of that snapshot. This is not how the Realtime-Database was designed to work, as Firebase intents developers to subscribe to a particular section of data. This then means that components can dynamically update when new data is added, or existing data is modified. In future, I may change this to use a subscription system.





Once I had tested and verified that the routing was working correctly, I then began work on the sub-components of the page. I started by creating a small button-like object at the top right of the page. This will show the user/admin what the status of the problem is. Admins can then click on that button, opening a small dialogue that can be used to change the status.

The chip itself was easy enough to code, it just needed a few tweaks to the CSS to get it to format correctly. I also had to set up a small switch statement to set the properties of the chip based on the status. This could be handled by putting the chip and its logic into a component of its own, which I may do later.

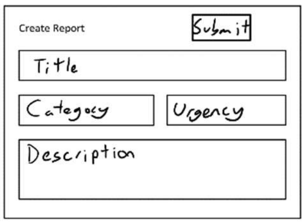
I then had to set up the dialogue, which was significantly more difficult. I tried many different approaches for this, but the one I settled for was using the chip to change the visibility of another component by using a state variable on the main component. When the chip is clicked, it changes the state, forcing the component to re-render. The new status changes the visibility of the dialogue component, making it appear on the screen.

The dialogue component holds the different options as selectable buttons. When created, the *ViewManager* component takes several call-backs as properties. When the user clicks on the confirmation button, this then triggers a call-back on the *ViewManager* component. This then changes the state controlling the dialogue visibility and calls a function on the *Manager* component to update the value on the database.

The main structure of this page is the column that holds the comments, each comment has several properties, such as the user who created it, the time and also the comment itself. The comment component is fairly simple in terms of programming, but the CSS was very difficult to format. I ended up using two *divs*, the top one has a slight grey shade and rounded top corners, and the bottom one is white with a box shadow and rounded bottom corners. I took inspiration from the GitHub Issue tracker, which uses a similar kind of style. This also fits in with the Material Design style that I'm using for the project.

[29] React Router provides a Switch component that selects which component to display based on the current URL

Create Report

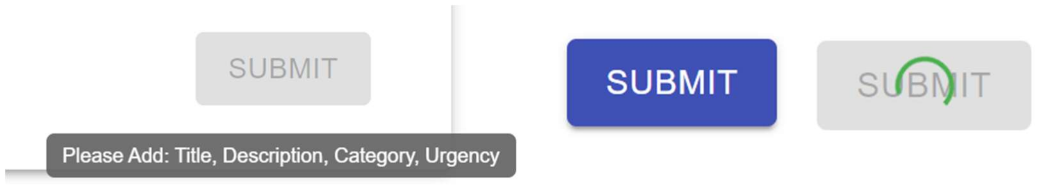


The final main piece of functionality is the page to create a report. Like the rest of the project, I aim to make it as simple and intuitive as possible. However, unlike the ViewReport page, I don't have anything to base the design off, so I started by creating a few rough sketches of potential layouts. I liked the idea of having a few smaller boxes for the properties such as the title, urgency, and the report category, then having a much larger box for the description.

Most of the backend functionality was in place by this point, I just needed to create a new function in the Manager class to post the data to the backend. I also built a lot of error checking into the post function to make sure that invalid data couldn't accidentally be sent to Firebase. This makes it easier for me to write the CreateReport component, as I don't need to do so much checking before I send the data to the Manager. It also makes it easier for another developer to come in and make changes, without having to worry about breaking anything.

I won't be going into all of the details of development, as most of the process is just a repeat of what I've covered before, but with even else interesting features. However, there is one particular feature I do want to talk about. I created a custom button that turns into a loading wheel until the page has finished the task. This will be helpful to indicate to the user whether a task has finished successfully or has failed.

Even though I added error checking to the Manager class, I still programmed a small function to check which fields still needed to be filled in. This allows me to create a tooltip that can be wrapped around the Submit button, indicating to the user why they can't currently submit the form



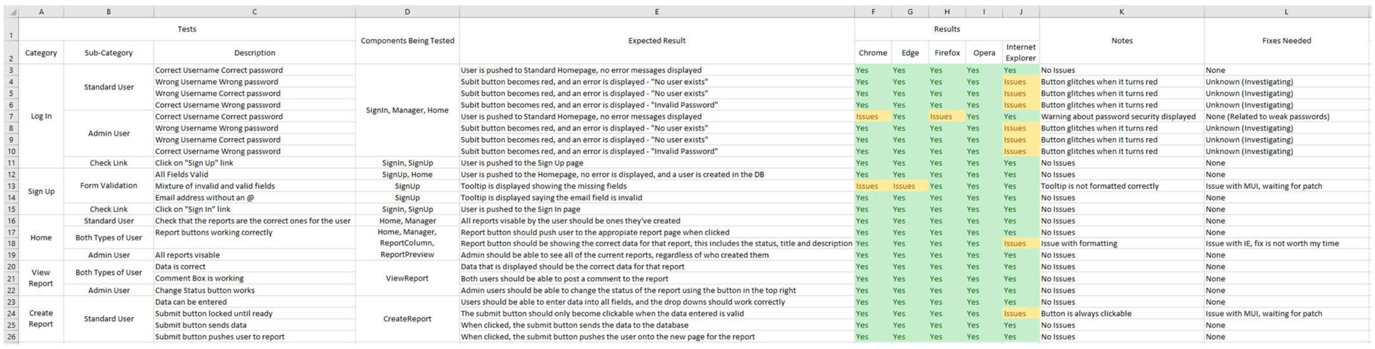
Please Add: Title, Description, Category, Urgency

Testing Report

More details about the testing process are mentioned in the "Testing, Issues & Beta Release" section of the documentation.

There were only a few minor issues, most of which were not related to my code directly. Several of them were caused by differences in the way Internet Explorer renders HTML. But as IE only has a 4% consumer market share at the time of writing, it's not worth my time to try and fix.

I believe these issues arise because of the outdated rendering engine that Internet Explorer uses. Most others use the Chromium engine, which supports almost all features defined in the HTML standard, whereas the older IE engine only has partial support for newer features, such as CSS Flexboxes, which my website relies on heavily.



Tests			Components Being Tested	Expected Result	Results					Notes	Fixes Needed
Category	Sub-Category	Description			Chrome	Edge	Firefox	Opera	Internet Explorer		
Log In	Standard User	Correct Username Correct password	SignIn, Manager, Home	User is pushed to Standard Homepage, no error messages displayed	Yes	Yes	Yes	Yes	Yes	No Issues	None
		Wrong Username Wrong password		Subit button becomes red, and an error is displayed - "No user exists"	Yes	Yes	Yes	Yes	Issues	Button glitches when it turns red	Unknown (Investigating)
		Wrong Username Correct password		Subit button becomes red, and an error is displayed - "No user exists"	Yes	Yes	Yes	Yes	Issues	Button glitches when it turns red	Unknown (Investigating)
		Correct Username Wrong password		Subit button becomes red, and an error is displayed - "Invalid Password"	Yes	Yes	Yes	Yes	Issues	Button glitches when it turns red	Unknown (Investigating)
	Admin User	Correct Username Correct password		User is pushed to Standard Homepage, no error messages displayed	Issues	Yes	Issues	Yes	Yes	Warning about password security displayed	None (Related to weak passwords)
		Wrong Username Wrong password		Subit button becomes red, and an error is displayed - "No user exists"	Yes	Yes	Yes	Yes	Issues	Button glitches when it turns red	Unknown (Investigating)
		Wrong Username Correct password		Subit button becomes red, and an error is displayed - "No user exists"	Yes	Yes	Yes	Yes	Issues	Button glitches when it turns red	Unknown (Investigating)
		Correct Username Wrong password		Subit button becomes red, and an error is displayed - "Invalid Password"	Yes	Yes	Yes	Yes	Issues	Button glitches when it turns red	Unknown (Investigating)
	Check Link	Click on "Sign Up" link	SignIn, SignUp	User is pushed to the Sign Up page	Yes	Yes	Yes	Yes	Yes	No Issues	None
Sign Up	Form Validation	All Fields Valid	SignUp, Home	User is pushed to the Homepage, no error is displayed, and a user is created in the DB	Yes	Yes	Yes	Yes	Yes	No Issues	None
		Mixture of invalid and valid fields	SignUp	Tooltip is displayed showing the missing fields	Issues	Issues	Yes	Yes	Yes	Tooltip is not formatted correctly	Issue with MUI, waiting for patch
		Email address without an @	SignUp	Tooltip is displayed saying the email field is invalid	Yes	Yes	Yes	Yes	Yes	No Issues	None
	Check Link	Click on "Sign In" link	SignIn, SignUp	User is pushed to the Sign In page	Yes	Yes	Yes	Yes	Yes	No Issues	None
Home	Standard User	Check that the reports are the correct ones for the user	Home, Manager	All reports visable by the user should be ones they've created	Yes	Yes	Yes	Yes	Yes	No Issues	None
	Both Types of User	Report buttons working correctly	Home, Manager, ReportColumn, ReportPreview	Report button should push user to the appropiate report page when clicked	Yes	Yes	Yes	Yes	Yes	No Issues	None
				Report button should be showing the correct data for that report, this includes the status, title and description	Yes	Yes	Yes	Yes	Issues	Issue with formatting	Issue with IE, fix is not worth my time
	Admin User	All reports visable		Admin should be able to see all of the current reports, regardless of who created them	Yes	Yes	Yes	Yes	Yes	No Issues	None
View Report	Both Types of User	Data is correct	ViewReport	Data that is displayed should be the correct data for that report	Yes	Yes	Yes	Yes	Yes	No Issues	None
		Comment Box is working		Both users should be able to post a comment to the report	Yes	Yes	Yes	Yes	Yes	No Issues	None
	Admin User	Change Status button works		Admin users should be able to change the status of the report using the button in the top right	Yes	Yes	Yes	Yes	Yes	No Issues	None
Create Report	Standard User	Data can be entered	CreateReport	Users should be able to enter data into all fields, and the drop downs should work correctly	Yes	Yes	Yes	Yes	Yes	No Issues	None
		Submit button locked until ready		The submit button should only become clickable when the data entered is valid	Yes	Yes	Yes	Yes	Issues	Button is always clickable	Issue with MUI, waiting for patch
		Submit button sends data		When clicked, the submit button sends the data to the database	Yes	Yes	Yes	Yes	Yes	No Issues	None
		Submit button pushes user to report		When clicked, the submit button pushes the user onto the new page for the report	Yes	Yes	Yes	Yes	Yes	No Issues	None

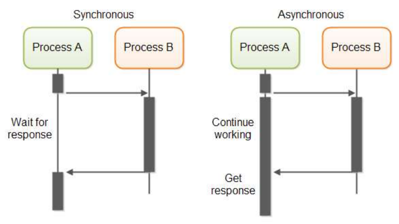
Data Improvements & Caching

Overview

At the time of writing the project has been running for nearly 8 months, and over that time there have been quite a few changes. One of the key ones has been my programming skills, which have improved measurably since I starting the development of the application. Due to this, there is a considerable amount of older code that needs to be rewritten or updated. A major part of this is the code that pulls data out of the Firebase database and uses it to render the components. Some of the older components still only fetch the data once, whereas I now know how to add callbacks. I also want to use this time to implement a caching component that pulls as much data as possible on the first load.

Development

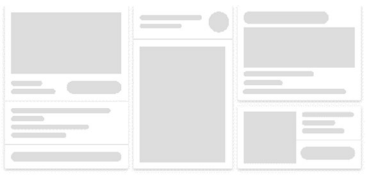
Dynamic Fetching



↑ *Synchronous vs Asynchronous*

Currently, when my components first load in, they call a function on my *Manager* class, which makes an asynchronous[30] request to the Firebase database. This means that React can continue rendering other elements while the data is being fetched. However, this approach was causing some issues due to JavaScript Promises.

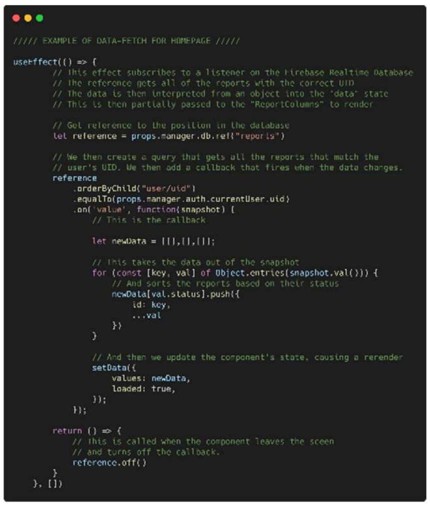
A *promise* is effectively telling another part of the program that it will get its data, but not right now. That's where the name comes from, it is promising to supply the data at a later point.



↑ *Example of a Skeleton*

The main issue was that when the component was rendered in for the first time, it would have to wait until the Manager had finished fetching the data. This was not the behaviour I desired, as ideally, I wanted to draw in an empty template (known as a skeleton) until the data arrived, and then adding pieces of data to the template as they arrive. The benefit of this approach is that it indicates to the user that something is happening, even if the data hasn't loaded yet.

Another issue was errors caused by React trying to update components that were no longer on the screen. This could happen when the data-fetching function was still running, even though the user had changed screens. However, this is solved relatively easily by adding a check to make sure the component is still loaded before updating the state.



```
///// EXAMPLE OF DATA-FETCH FOR HOMEPAGE /////

useEffect(() => {
        // This effect subscribes to a listener on the Firebase Realtime Database
        // The reference gets all of the reports with the correct UID
        // The data is then interpreted from an object into the 'data' state
        // This is then partially passed to the "ReportColumns" to render

        // Get reference to the position in the database
        let reference = props.manager.db.ref("reports")

        // We then create a query that gets all the reports that match the
        // user's UID. We then add a callback that fires when the data changes.
        reference
            .orderByChild("user/uid")
            .equalTo(props.manager.auth.currentUser.uid)
            .on('value', function(snapshot) {
                // This is the callback

                let newData = [[],[],[]];

                // This takes the data out of the snapshot
                for (const [key, val] of Object.entries(snapshot.val())) {
                    // And sorts the reports based on their status
                    newData[val.status].push({
                        id: key,
                        ...val
                    })
                }

                // And then we update the component's state, causing a rerender
                setData({
                    values: newData,
                    loaded: true,
                });
            });

        return () => {
            // This is called when the component leaves the sceen
            // and turns off the callback.
            reference.off()
        }
    }, [])
```

To get the behaviour I wanted, and improve the quality of the code, I decided to de-abstract the logic out of the Manager and into the individual components. While this approach is fine for my application, as most of the code is only being used once, I appreciate that this would not necessarily be the best for larger projects.

First, I added a *useEffect* function to each of my data-fetching components. This tells React to run this bit of code outside of the rendering cycle, allowing me to only fetch data when the component is first loaded in.

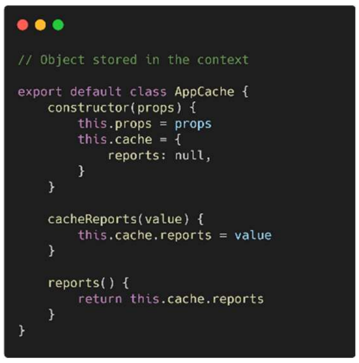
I then reprogrammed the data-fetching logic to subscribe to the database rather than just doing a one-off fetch. This means that I can update the state on the component when new data becomes available, which will force React to rerender the component with the new data.

Finally, to finish the data-fetching, I programmed an *if* statement to check if the component is still on screen before updating the state. This prevents the errors I mentioned earlier and also prevents potential memory leaks, where React can accidentally update a different component.

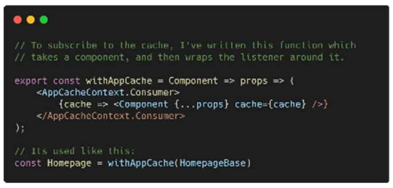
[30] Asynchronous coding is when a piece of code is being executed in parallel with the rest of the program, meaning that the program doesn't have to suspend until the request is finished.

Caching Data

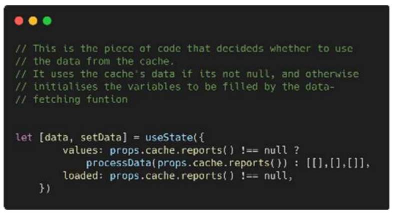
If I was deploying my application in a production environment, I would be charged by Firebase based on the quantity of data I download, and the number of requests I make. Currently, I haven't optimised my requests, meaning that every time the user goes to the homepage, the program will fetch all of the reports again. While in development, this doesn't cause many issues, as there are only a few reports from a handful of users, meaning the time it takes to request the data is relatively quick (around 10ms). However, in production, this could cause very slow load times, particularly for admins, who would have to download the entire list of reports.



```
// Object stored in the context

export default class AppCache {
    constructor(props) {
        this.props = props
        this.cache = {
            reports: null,
        }
    }

    cacheReports(value) {
        this.cache.reports = value
    }

    reports() {
        return this.cache.reports
    }
}
```



```
// To subscribe to the cache, I've written this function which
// takes a component, and then wraps the listener around it.

export const withAppCache = Component => props => (
    <AppCacheContext.Consumer>
        {cache => <Component {...props} cache={cache} />}
    </AppCacheContext.Consumer>
);

// Its used like this:
const Homepage = withAppCache(HomepageBase)
```



```
// This is the piece of code that decideds whether to use
// the data from the cache.
// It uses the cache's data if its not null, and otherwise
// initialises the variables to be filled by the data-
// fetching funtion


let [data, setData] = useState({
        values: props.cache.reports() !== null ?
            processData(props.cache.reports()) : [[],[],[]],
        loaded: props.cache.reports() !== null,
    })
```

To combat this, I've decided to implement a caching[31] system. When the user first loads their homepage, the application will make a request to Firebase and wait for the data to arrive. When the data becomes available, we add it to the cache, allowing it to be accessed in the future.

If the user then goes to a specific report, the component will make a request to Firebase for that report, but instead of waiting, it will use the data in the cache to render. The idea of making the new request is to check if new data has become available.

To start, I create a React context (see the explanation of the manager context for more info), that stores a basic class which has functions to store and retrieve the data. When a component that needs access to the cache is created, it'll subscribe by wrapping a listener around itself. This means that when the contents of the cache update, all of the components subscribed will also update.

The next job was to store the data in the cache when it arrives. This will be the responsibility of the Homepage component, which will call the *cacheReports* function as pass it the data it's received from Firebase.

The other piece of functionality needed is the ability to check the cache before waiting for the React data. This is relatively easy, as I just need to check if the cache's *reports* function returns a *null* value, then I know if there's any data. I did this by writing a pair of boolean expressions that checked whether the cache is not empty, and if so sets the variables to the cache's data.

One feature I may add at a later date is an expiry limit on the cache so that after a set length of time, the program deems the cache to be invalid, and there will wait until the data is back from Firebase before rendering.

Testing

As this caching system is a major change to the backend systems, a lot of testing was required to ensure that everything remained stable. To start with, I implemented an artificial delay in the Firebase requests, making each one take 5 seconds. This allows me to easily see when requests are made, and when data is being fetched from the cache.

This showed up some issues where the system would still fetch data from Firebase when it tried to access a report. On closer inspection, this was due to the UID of the report being a different datatype to the UID being returned by Firebase, which was causing a mismatch and leading the code to think that it couldn't find the report it was looking for.

I also had to adjust the order in which components fetched data, as sometimes it would not bother to check the cache and go straight to Firebase instead. This is because React does not work in a sequential way, but rather with many different functions running concurrently, which can lead to some things happening before previous operations had finished. I solved this by adding dependencies to the *useEffect* functions, which forces them to wait for the component to finish setting up before beginning to fetch the data.

[31] Caching is the process of storing regularly used data prevent unnecessary requests

Hosting & Deployment

Overview

Important Design Change

When I started the project, I had intended to do a lot of processing on the backend of my application. To help with this, I decided to ASP.NET which makes it much easier to write code to respond to requests. However, ASP.NET is reasonably difficult to publish and host, especially when compared to alternatives such as Node.js, Flask or Django.

So for the final deployment, I've changed from ASP.NET to Flask, a Python framework I've mentioned earlier in the project. This section will be about the issues with ASP.NET, the migration to Flask, and how I'm hosting the application.

Development

Issues with ASP.NET

ASP.NET is a fabulous backend for many different applications and is used by some of the largest companies to host their websites and REST APIs. Because of this, it has a particularly complicated deployment process, as it's intended to be hosted on a production-grade web server like Apache, rather than just running as an executable on a Virtual Machine.

One of the major issues I faced was that by the time I came to host the application, many hosting providers such as Google and AWS had stopped supporting the older version of ASP.NET I was using. The only real solution for this was to migrate to the 3.1 version of ASP. This required significant changes to the backend but eventually did work.

However, in the process, my IDE[32] configurations got changed, which meant that when I went to run my deployment scripts, it didn't compile correctly. I spent a long time trying to correct this, but I eventually decided to give up and migrate to Python.

Migration to Flask

From now on, I will be developing in a different GitHub project, which will allow me to revert to the ASP.NET project if necessary: https://github.com/MattFrench019/Helpdesk-Py

The first big change to be made was to reorganise the file/folder structure of the project. Instead of having all of the React application in a separate folder, it needs to be in the root of the project, otherwise, it won't compile to the correct place. Several files became redundant, such as multiple *README.md* and *.gitignore*. These were removed and their content was merged into one file.

As well as adjusting the code, I also needed to change my IDE from Rider to PyCharm, and I'm no longer developing in C#. PyCharm has better support for Python, as the name suggests.

The first thing to do was to import the code from GitHub and put it into a local Git repo. I then needed to set up a local Python environment. This can be done in different ways, but I prefer to use a Virtual Environment, which is a copy of your main Python interpreter that can be modified with different modules and packages depending on the project. In PyCharm this only takes a few clicks.

I then needed to set up Node.js, which is needed for running and compiling the React application. Node was already installed on my computer, so I just needed to install the dependencies. This is just done by running *npm install*, which looks in the *package.json* file and finds all of the packages necessary.

Finally, I wrote a small batch script that will make it easier to run the project, as before Python can start, React must first compile into a single HTML file. The batch script just runs the commands necessary to do this.

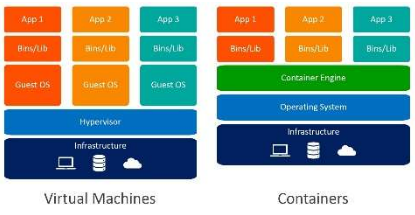
[32] An IDE is an application used to develop large projects, it provides many advanced features such as autocomplete for code

Deploying with Docker

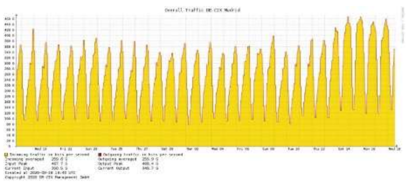
What is Docker?

Normally, when applications are deployed in production[33], are run in Virtual Machines[34] (VMs) on large servers. Some companies will host their applications on-premise using their own servers, but more and more companies are choosing to use hosting providers, such as Google or Amazon, which have vast arrays of computers that can be virtualised into small or large VMs.

Self-hosting works fine when you have a steady stream of traffic, as you can just add new servers to run the application when demand starts to reach the number of requests you can handle. However, it doesn't work well when dealing with traffic that spikes suddenly and at times can drop to virtually nothing. This is because you need enough servers to deal with the highest amount of traffic you receive, but most of the time they'll be sitting around with very little usage.



Docker provides a solution to this problem, as it allows an application to be easily packaged up into a small *container*, which can then be easily run. A container is like a small VM but works slightly differently. Instead of having an entire operating system, it uses part of the host machine's operating system, making them much smaller than a traditional VM. While this means that the host's OS must be the same as the container's OS, it means that many different applications can be running in containers on the same server.

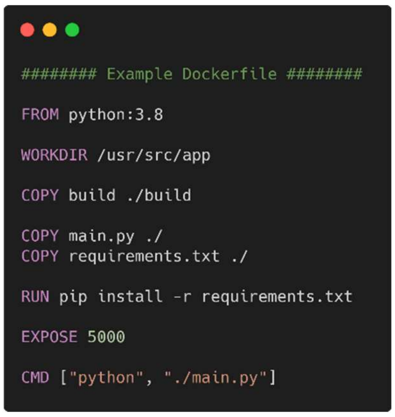

↑ *Netflix Traffic*

Other tools can then be used to dynamically increase the number of containers when demand increases. A key pioneer of this technology was Netflix, which notoriously suffers from huge spikes in demand. They've containerised all aspects of their applications, and use a tool called Kubernetes, which allows them to easily scale to meet demand.

Containers also make the famous phrase "it runs on my computer" redundant, as the container holds all of the necessary data and programs to run itself. This means that if a container runs on one person's computer, it should run on *all* computers. However, this is not always the case, as I'll explain later.

How do you make a container?

To make a container, you define a *Dockerfile* which is effectively a list of instructions and commands that will be run on the container to set everything up.



```
######## Example Dockerfile ########

FROM python:3.8

WORKDIR /usr/src/app

COPY build ./build

COPY main.py ./
COPY requirements.txt ./

RUN pip install -r requirements.txt

EXPOSE 5000

CMD ["python", "./main.py"]
```

They start by defining the base image, which is a slimmed-down pre-configured OS that is specifically designed for Docker containers. This base image may just be as simple as a Linux distribution, or it may already come pre-installed with applications such as Python, Node.js or even a fully configured database.

From then on, all the instructions will be executed on the container. Common commands are ones such as *COPY*, which takes files from the computer you're building from and copies them to the container; *RUN* which will run a command defined by the user; *EXPOSE* which will tell Docker to expose a particular port[35]; and *CMD* which defines the command that will be used to start the application on the container.

When the Dockerfile is all set up, it can be run by Docker to compile a container. This container can then be easily deployed to a server by using a *docker-compose.yml* file, which tells Docker how to configure the container to run properly. These are often used to configure applications in which certain settings, like how a *.config* file would typically be used.

[33] Production is referring to when an application is actually being actively used by users, by this point it needs to be very stable and able to handle errors if they arise.
[34] A Virtual Machine is a computer than runs within another computer, it works by hosting a separate operating system within a virtual hard-drive, which is a large file that acts as the file system for the VM.
[35] A port is a place where computers can listen and respond to requests.

How am I hosting the Application?

For this project, I've decided to self-host the application rather than using a hosting provider such as Google Cloud. The main reason for this is the much higher costs of using cloud-hosting rather than self-hosting. It would have almost certainly been easier using a hosting provider, as they pre-configure all of their VMs, and I would have been able to deploy my Docker container without any need for managing keys or moving files, as will be discussed in the next section.

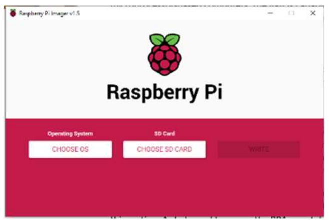
↑ *My Raspberry Pi*

To self-host the project I went out and bought a Raspberry Pi 4B (RP4), which I'll use to run the compiled Docker container. This would not be suitable if the application was being run in production, as there is absolutely no redundancy, and also because the RP4 will only be able to manage about 10 concurrent connections at once. However, as this application is for a small project, the RP4 should be able to cope.

The RP4 is a small computer, costing between £40-80. It has a small amount of RAM (my version has 2GB), an ARM[36]-based CPU and a selection of ports such as Ethernet, USB and HDMI. Unlike other small, microprocessor-based computers, the RP4 is capable of running a full operating system like Linux, which makes it much easier for me to deploy applications, as I don't have to compile binary executables specifically for the microprocessor.

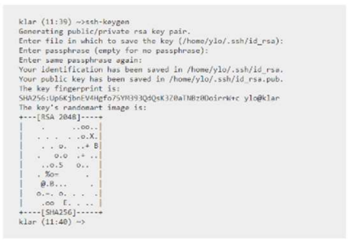
Setting up the RP4 & Deploying

This was where my plan to self-host started to fall apart. I completely underestimated how much work would have to go into setting up the RP4 for deployment. The entire process took about 10 hours, mainly because Docker refused to work on the ARM chip.



↑ *Raspberry Etcher*

I started by plugging the RP4 into my router with an Ethernet cable and powering it up. I then needed to create the boot drive, which for this Pi is a MicroSD card. To do this, I plugged the card into my PC and downloaded Raspberry's *etcher* tool. The etcher tool takes a particular operating system and writes all the necessary files to the MicroSD card. This is a fairly compilated process, as binary files need to be compiled and placed in specific directories on the card to make sure the RP4 can boot from it.

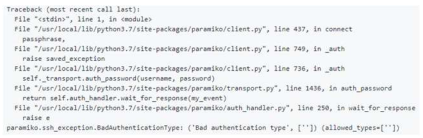


```
klar (11:39) ~>ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/ylo/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/ylo/.ssh/id_rsa.
Your public key has been saved in /home/ylo/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:Up6KjbnEV4Hgfo75YM393QdQsK3Z0aTNBz0DoirrW+c ylo@klar
The key's randomart image is:
+---[RSA 2048]----+
|    .      ..oo..|
|   . . .  . .o.X.|
|    . . o.  ..+ B|
|   .   o.o  .+ ..|
|    ..o.S   o..  |
|   . %o=       . |
|    @.B...     . |
|   o.=. o. . .  .|
|    .oo  E. . .. |
+----[SHA256]-----+
klar (11:40) ~>
```

↑ *Raspberry Etcher*

I chose to use the Raspbian Linux distribution, which is a 32-bit[37] version of Debian[38] specifically designed for the Raspberry Pi's limited resources. This would later cause some issues, which I will discuss later in this section. As I planned to access the RP4 remotely, I needed to enable SSH. SSH is a protocol that allows commands to be sent to one computer from a terminal on another computer, it provides encryption using either a password or public-private key pair. To do this, I just had to place a file called *ssh* in the root directory of the RP4.

After putting the MicroSD card into the RP4, I then remoted into it using the default username and password. I immediately changed the password to prevent the Pi from being compromised. Next, I made sure the RP4 was using a static IP address rather than one assigned by the router. This makes sure I can always access the Pi, as otherwise, the IP may change, making it virtually impossible for me to remote in to change it back.



```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python3.7/site-packages/paramiko/client.py", line 437, in connect
    passphrase,
  File "/usr/local/lib/python3.7/site-packages/paramiko/client.py", line 749, in _auth
    raise saved_exception
  File "/usr/local/lib/python3.7/site-packages/paramiko/client.py", line 736, in _auth
    self._transport.auth_password(username, password)
  File "/usr/local/lib/python3.7/site-packages/paramiko/transport.py", line 1436, in auth_password
    return self.auth_handler.wait_for_response(my_event)
  File "/usr/local/lib/python3.7/site-packages/paramiko/auth_handler.py", line 250, in wait_for_response
    raise e
paramiko.ssh_exception.BadAuthenticationType: ('Bad authentication type', ['']) (allowed_types=[''])
```

↑ *Example SSH Error*

I then installed Docker and configured the correct permission for my user. Because my Pi uses a 32-bit OS, I needed to build the Docker container on my PC, as the compiling process requires a 64-bit processor. This is where the problems with Raspbian started to become apparent. Due to a bug with the Docker build-script for Windows, I couldn't remotely deploy to the RP4, and as my Pi was running a 32-bit OS, I couldn't run the build-script on the Pi either. After much searching, I was unable to find a fix for the bug, which effectively meant I needed to scrap Raspbian and use a 64-bit OS like Ubuntu.

I then needed to redo all the steps above for Ubuntu, which took a considerable amount of time. I then was able to transfer the files to the RP4 and build the Docker application. After a bit of configuration, I was able to get the app to host on port 5000, as the default HTTP port was being used by another application. I then just needed to test the stability of the RP4 by making several concurrent requests using a tool called Postman. Even when handling 30 simultaneous requests, the RP4 didn't increase above 10% CPU usage, which makes me relatively confident that this would be a solution that could be deployed into a production environment, for example in a school or small business.

[36] ARM is a processor architecture typically used for small microprocessors, as it takes less power to run and is easier to manufacture. However, it is also gaining traction in the mainstream market, with companies like Apple creating their own ARM-based M1 processors.
[37] 32 and 64 bit operating systems refer to the number of bits being used for the Address Bus, an important note is that a 32-bit program can run on a 64-bit OS, but a 64-bit program **cannot** run on a 32-bit OS.
[38] Debian is a very common version of Linux.

Testing & Notable Issues

Testing & Issues

As most of the project is based around the user-interface, the majority of testing was done by visual inspection through the development process. However, some of the backend systems needed a different approach, such as a system of unit and integration tests, as well as using debugging statements to check the code was working correctly.

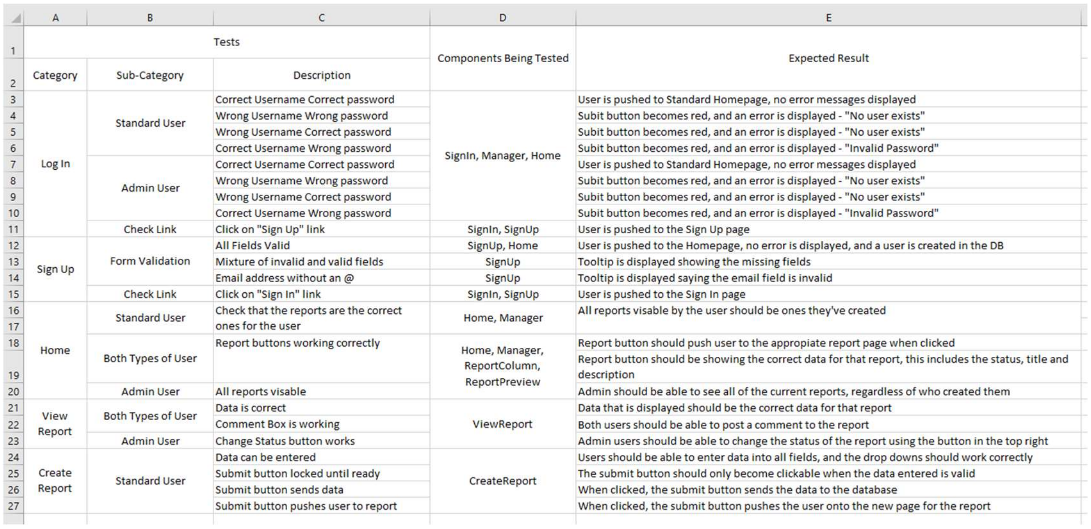
The overall process, while effective, was reasonable informal and undocumented. I based my approach on my extensive experience with programming, allowing me to maximise the time spent on development while retaining quality and keeping the code robust and solid.

When working with CSS, some behaviour is particularly unusual. To help avoid this, I use multiple browsers and screen sizes to ensure that the application is compatible with different computers. Some major problems can arise when these inspections are carried out. Below, I have documented some of the more serious, bizarre, and interesting issues that I found throughout the course of development.

Test Plan Evolution

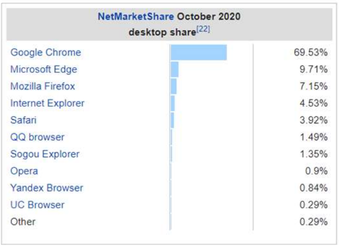
At the start of the project, I created a basic test plan that covered the basic features I needed to test on each revision. At first, there were very few items, only the very fundamentals of the authentication system needed to be tested. As the project developed and new features were added, I would create new sections in the test plan. I also added a section for testing each major browser, as explained in the next section.

Most of the testing throughout the development was done continuously. Generally, I would make some small changes, and then just test the bits of functionality they would affect to make sure everything is still working. The only time when I would do semi-formal testing was at the end of the development of a major feature, such as a new page. This would be when I would create the new sets of tests and would go back and check all the site's functionality.



	A	B	C	D	E
1	Tests			Components Being Tested	Expected Result
2	Category	Sub-Category	Description		
3	Log In	Standard User	Correct Username Correct password	SignIn, Manager, Home	User is pushed to Standard Homepage, no error messages displayed
4			Wrong Username Wrong password		Subit button becomes red, and an error is displayed - "No user exists"
5			Wrong Username Correct password		Subit button becomes red, and an error is displayed - "No user exists"
6			Correct Username Wrong password		Subit button becomes red, and an error is displayed - "Invalid Password"
7		Admin User	Correct Username Correct password		User is pushed to Standard Homepage, no error messages displayed
8			Wrong Username Wrong password		Subit button becomes red, and an error is displayed - "No user exists"
9			Wrong Username Correct password		Subit button becomes red, and an error is displayed - "No user exists"
10			Correct Username Wrong password		Subit button becomes red, and an error is displayed - "Invalid Password"
11		Check Link	Click on "Sign Up" link	SignIn, SignUp	User is pushed to the Sign Up page
12	Sign Up	Form Validation	All Fields Valid	SignUp, Home	User is pushed to the Homepage, no error is displayed, and a user is created in the DB
13			Mixture of invalid and valid fields	SignUp	Tooltip is displayed showing the missing fields
14			Email address without an @	SignUp	Tooltip is displayed saying the email field is invalid
15		Check Link	Click on "Sign In" link	SignIn, SignUp	User is pushed to the Sign In page
16 17	Home	Standard User	Check that the reports are the correct ones for the user	Home, Manager	All reports visable by the user should be ones they've created
18		Both Types of User	Report buttons working correctly	Home, Manager, ReportColumn, ReportPreview	Report button should push user to the appropiate report page when clicked
19					Report button should be showing the correct data for that report, this includes the status, title and description
20		Admin User	All reports visable		Admin should be able to see all of the current reports, regardless of who created them
21	View Report	Both Types of User	Data is correct	ViewReport	Data that is displayed should be the correct data for that report
22			Comment Box is working		Both users should be able to post a comment to the report
23		Admin User	Change Status button works		Admin users should be able to change the status of the report using the button in the top right
24	Create Report	Standard User	Data can be entered	CreateReport	Users should be able to enter data into all fields, and the drop downs should work correctly
25			Submit button locked until ready		The submit button should only become clickable when the data entered is valid
26			Submit button sends data		When clicked, the submit button sends the data to the database
27			Submit button pushes user to report		When clicked, the submit button pushes the user onto the new page for the report

Compatibility with Mozilla Firefox / Microsoft Edge


NetMarketShare October 2020 desktop share[22]

Google Chrome	69.53%
Microsoft Edge	9.71%
Mozilla Firefox	7.15%
Internet Explorer	4.53%
Safari	3.92%
QQ browser	1.49%
Sogou Explorer	1.35%
Opera	0.9%
Yandex Browser	0.84%
UC Browser	0.29%
Other	0.29%

At the heart of any browser application, there is code that handles the interpretation of HTML and the rendering of web pages. This is commonly known as the browser engine. There are many different engines that different browser applications use, but one of the most common ones is Chromium.

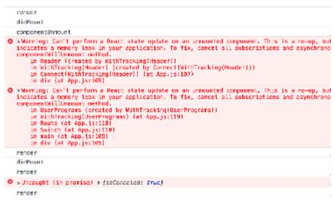
Chromium was developed by Google and published to the internet as an open-source project, allowing anyone to use it without a license. Because of this, it quickly became the most common browser engine and is currently used by Google Chrome, Opera and Apple's Safari. Also, during the development process, Microsoft Edge changed from using Microsoft's proprietary engine to using Chromium.

The one notable exception is Mozilla Firefox, which currently accounts for around 7% of the browser market. Because it uses a different engine, some features that are supported by Chromium, work inconsistently or not at all. These features are generally not critical, as they are normally for controlling browser-specific elements like scroll bars or tab colour, but it is important to be aware of them while developing.

For these reasons, at the end of each development phase, I spend around 10 minutes testing the application on multiple browsers. The main ones I use are Google Chrome, Microsoft Edge and Mozilla Firefox. I would also like to be able to test on Apple's Safari, but I am currently unable to as I don't own a Mac.

Issues with Callbacks & React Hooks

To understand the issues in this section, you need some information about how React behaves at a low level. React is a framework written in JavaScript that provides the ability to change components without reloading the screen. This is achieved by having an *engine* that runs continuously, updating parts of the screen when data changes. To tell this engine how to render your webpage, you supply React with JavaScript files that define a *component*.

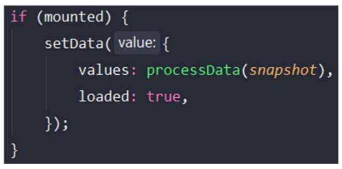


```
render
didMount
componentUnmount
▸Warning: Can't perform a React state update on an unmounted component. This is a no-op, but
indicates a memory leak in your application. To fix, cancel all subscriptions and asynchrono
componentWillUnmount method.
    in Header (created by WithTracking(Header))
    in WithTracking(Header) (created by Connect(WithTracking(Header)))
    in Connect(WithTracking(Header)) (at App.js:107)
    in div (at App.js:105)
▸Warning: Can't perform a React state update on an unmounted component. This is a no-op, but
indicates a memory leak in your application. To fix, cancel all subscriptions and asynchrono
componentWillUnmount method.
    in UserPrograms (created by WithTracking(UserPrograms))
    in WithTracking(UserPrograms) (at App.js:119)
    in Route (at App.js:118)
    in Switch (at App.js:110)
    in main (at App.js:109)
    in div (at App.js:105)
render
didMount
render
▸Uncaught (in promise) ▸{isCanceled: true}
render
```

↑ *Example Error*

These components are typically made of subcomponents, and those are again made of smaller components, all the way down to the basic HTML tags. Each component is dependent on its parent, whether that's for data or formatting, the engine doesn't care, it only knows that if something changes with the parent component, it must re-render all the child components. This fundamental behaviour allows Single-Page-Applications to be created using React, as you can dynamically update components as new data becomes available.

For my application, I want to update components when the data I'm fetching from Firebase returns. To achieve this, I need to set up a system to re-render the component when the data returns. I did this by setting up functions that run when the component is first loaded which make requests to Firebase for the data. The function then stores this data in a state variable, which triggers the re-render.



```
if (mounted) {
    setData( value: {
        values: processData(snapshot),
        loaded: true,
    });
}
```

↑ *Checks to make sure the component is still mounted*

However, this caused certain issues that were remarkably difficult to track down. The problems arose when the component's data-fetching function returned the data *after* the component had left the screen. React can get confused when this happens, as it reuses low-level memory addresses to reduce the footprint of the application. For example, if the code is running in the background, and React then unloads the component it belongs to from memory, and then places a different component in that same memory address, the function may inadvertently end up sending data to the wrong place. This is known as a memory leak, which can be dangerous when dealing with sensitive data such as passwords, as hackers are occasionally able to exploit these.

These issues regularly came up when testing, but as React has already got rid of the component, it doesn't provide much information when the error is logged, making them difficult to track down. To prevent these memory leaks, I added a statement that checks the component is still loaded before updating the state.

Evaluation & Conclusion

This project has been going on for nearly 10 months now, and my abilities as a programmer have noticeably improved since the beginning. When I was rewriting older parts of the application, I was shocked to see some of the shoddy code that I had written only a few months ago. Even now there are almost certainly parts of the app that could be improved, but I have attempted to make the code as supportable as possible, so if this had been deployed in production, another developer could come in and build on what I've started. Now I have reached the end of the project, I will be evaluating its success based on 3 factors: **Completeness**, **Usability**, and **Supportability**.

Completeness

The first point I'll be evaluating is how well the application met the goals I set at the beginning of the project. For each goal I will be summarising how well the project achieves it, any issues I had with development, and whether it was worth the time. While I managed to achieve most of the objectives, due to time constraints, some features had to be prioritized to ensure a smooth user experience.

User Management

1. **Let users sign in and access their specific content**
The authentication works fabulously on the application; it is remarkably fast and provides adequate feedback about any errors. This was crucial for the project and took several attempts to get right. However, I am extremely glad I spent the time to migrate from my older system, as it would have caused no end of issues later in development.

2. **Multi-Level Authentication**
Again, the multi-level authentication is handled excellently, originally, I had planned to have two separate sign-in systems, but now any user can log in on the same page and be redirected to the appropriate homepage. If I had more time, I would have liked to set up a separate set of permissions that would have allowed someone to modify the permissions of users, rather than having to edit them in the database.

3. **Password Reset & Account Recovery**
At the time of writing, the project has no way for users to reset their password or recover their account. This was a decision I took early on, as I wouldn't be able to guarantee the security of the application. However, I have written most of the necessary backend functionality for these features, so it would be possible to build a new page where users could sort out account issues.

4. **Two Factor Authentication**
This was a feature I had intended to build when I started the project, but after researching the potential options for 2FA, I quickly dropped the idea, hence why it is only mentioned in the project objectives. I could quite easily integrate Google Authenticator into my application, but it would also make the app more difficult to use and support.

Problem Reporting – Teachers/Students

5. **Title, Description, Urgency, Status, Category**
These fields were all implemented in the Create Report page, which has a neat interface, making it easy to use.

6. **Photos/Video/Files**
This was another feature that was dropped during the project. The main reason for this was the lack of any decent pre-build options that I could have integrated into the application. Firebase provides a service called CloudStore, which provides a great programmatic way to store files, but the issue was the cost. For such a small project, the cost was just unviable.

7. **Status**
This is a feature of the application that I'm particularly proud of. Each report has its own status that can be updated by admins, but the clever part is that it will instantly update on another computer without the need for a refresh. This makes the site much easier to use and provides a far better experience for the user.

8. **Messaging**
This is another feature that worked very well. By using Firebase's callbacks I was able to create a system where users could have a conversation in real-time, rather than having to constantly refresh the page. This allows for both synchronous and asynchronous communication between the user and technical support, providing a better experience for the user.

Report Management – IT Support

9. **View Reports**
When an admin logs in, they are presented with a list of all the currently open reports. Each report can be clicked on to be open, making it easy for tech support members to navigate many different reports. I am incredibly pleased with the UI for the homepage, particularly for the admins, and it presents just the right amount of data, in a clear and concise way.

10. **Public/Private State Management**
When I started the project, I significantly underestimated the time it would take to develop the other features, which mean that as the project progressed, some features had to be shelved. This is one of those features; it would have taken a lot of time and effort to implement, for a relatively minor reward.

11. **Report Grouping**
Again, this was another feature that was shelved to make way for others. It would have required a redesign of the database to work properly, which would have taken too much time to make the feature viable.

Usability

After assessing the project objectives, I will be attempting to gauge how successful the User-Interface design has been. I've been using the application for nearly a year, so it follows that I would have a very solid understanding of how to navigate and use different features. Another key thing being tested will be the responsiveness of the interface, such as loading times and the delay between clicks and actions.

I'll be speaking to 2 of the people I originally interviewed. For each user, I'll ask them to create a new account, sign in, create a new report, and then add a comment to it. Unless they get stuck, I won't be providing any input or hints, as this would defeat the point of testing how intuitive the interface is. Throughout the process, I'll make notes about points where the users struggled, and any feedback they have.

Cameron Wride – Standard User

I spoke to Cameron at the start of the project for input and recommendations for the user side of the website. Throughout the project, he has helped me to improve the user-interface and test the robustness of the system. Many of the bugs and issues discovered were due to his "thorough" testing.
To test the user side of the website, I asked him to create an account, sign in, and then create a new report. I watched as he did this to check there weren't any issues with the UI design, and also to make sure the website was responsive enough to use.
I also got him to try using the admin side of the website, as even though he wouldn't be using it day-to-day, he does have a similar skill-level to the tech-support members.

Observations

Cameron was easily able to navigate the website, this is somewhat understandable, as he is familiar with many of the applications I took inspiration from. It took him about 1 minute and 30 seconds to complete the whole process, and most of this time was used for typing. He even remarked on how quick the website responded to his clicks.

The only minor issue raised with the user-interface was that there was no easy way to go from the report page after creating the report, back to the homepage. To get there you had to click the back button twice in the browser. This is far from ideal and is the kind of small issue that I find particularly irritating when using websites, and I will certainly be fixing it in the next patch.

There was also a minor issue raised with the admin-interface. When trying to change the report's status, he was stumped for about 10 seconds, as he wasn't aware that the status indicator was also a button. This was a major oversight for me, as I had never added any kind of responsiveness to the indicator, such as turning a different colour when hovered over, or just simply having a drop shadow. I view this as a major issue for anyone trying to use the interface, as it could easily prevent someone from being able to use the application.

Feedback

I asked Cameron a few questions regarding the website and the user experience, his answers are recorded below.

1. **How would you rate the user experience?**
   It's genuinely one of the easiest applications I've ever used, it helps that the functionality is just limited to what is necessary, but the user interface was a big part of that. There were a few things that weren't so intuitive when I first used them, but once you walked me through them, they did make quite a bit of sense.

2. **Would be happy to use the applications to log reports?**
   From a user point of view, it's really simple to use, and I would definitely be happy to use it. However, it could use some extra features, such as email notifications when the status of reports changes, or the ability to add photos or screenshots, but those features could always be added later. While I wouldn't be using it from an admin point of view, I do think some of the functionality is a bit lacking. It'd probably be fine for Sidcot, as we only have one or two guys working tech-support, but for a larger company, it would probably struggle.

3. **Do you have any other feedback or potential improvements?**
   I would kind of like to see a 'dark mode' implemented, but that's far less important that stuff like screenshot sharing. I'm not such a fan of the blue-pinkish colour scheme, but it is really easy to understand.

James Russell (Head of IT) – Admin User

I spoke to James Russell at the start of the project to develop an understanding of the kind of system a small business would need. As someone who has been in the industry for a long time, he has plenty of similar applications and provided especially useful information about features other applications did well, and ones where others failed. With this in mind, I've also developed a list of potential features that could be added in the future (that section is at the end of the evaluation).

As he would not be using the system on a day-to-day basis, rather than getting him to use the application, I talked about and demonstrated the different features and functionality. This would be the kind of format I would use if I were attempting to sell the software to a business.

Observations

One of the biggest take-aways from our conversation was that he was most interested in two pieces of core back-end functionality. These were the cloud-based database and authentication, as well as the lack of a central server. From a System-Administrator point of view, these are both major selling points of the application: firstly, as the database is hosted in the cloud, it doesn't need to be regularly backed up; and secondly, there's no need for a costly server to host the application.

He was also impressed by the sheer speed and responsiveness of the user interface, which he believed would be easily navigable even by users who hadn't been trained to use the application. However, he did point out a few pieces of missing admin-side functionality.

Feedback

After our conversation, we summarised the missing pieces of functionality into these three points:

1. **User-Reported Urgency**
   On the Create Report page, I noticed there was a box for users to set the urgency of their report. After working with many of these systems, I've always found that users tend to have a very different understanding of how urgent an issue as compared to someone doing tech-support. It might be worth removing that in favour of an urgency assigned by a support member when the report is first read. This would make it easier to prioritise issues, as it would mean that only truly urgent reports will be dealt with first.

2. **Grouping Reports**
   In the past, when we've had large technical issues, such as the email server going down, we often will get dozens of different reports about the issue. With the current system, the application provides, we would need to go into each issue and change the status. What could be better is having a way of grouping reports that are all regarding the same issue, and then have a button that changes the status of all of them at the same time. This would massively cut down on the time it would take for users to get feedback after the issue is solved.

3. **Screenshots & Photos**
   When dealing with issues, we often need to ask for a screenshot or photo of the problem so that we can accurately diagnose the issue. It would be useful if the application provided a way to attach screenshots to a comment, as at the moment we would have to get them to send the photo in another way, say an email. This would complicate matters, as there would be two separate conversations going on at the same time.

Supportability

This is a lot more difficult to measure than the other two areas as it is far more subjective. When I talk about "Supportability" I am talking about how easy it would be for another developer to come in and build on top of my existing code, and how easy it is for someone else to fix the application if it ever goes wrong. This was why I created the manager class, a place where I could put all my *backend* functionality, such as authentication systems and functions to interact with the database. The idea was that a developer could come in, read the documentation on the manager class, and then should be able to go off and create a new page without too many issues.

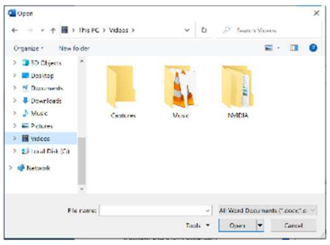
I would say that I've done a relatively good job of documenting the manager class, almost all of the functions are highly commented, and I've used as many easily understandable variable names as possible. I've also tried to centralise as much functionality in the manager class as possible so that if anything needs to be changed, say a company wants to use their own authentication system, it only has to be changed in the manager class.

Future Functionality

Screenshots & Attachments

One of the biggest features I wasn't able to implement was the ability to attach files to a report. The limiting factor was not the complexity, but the cost, as Firebase's product for storing files (CloudStore) is not free, unlike the database or authentication system. Because of this limitation, I've decided to write up about how I would implement a file-storage system, but without actually coding it.

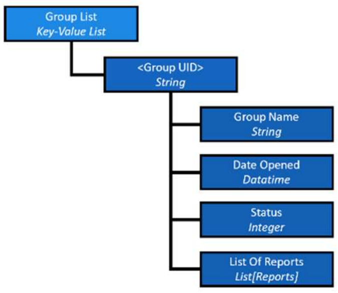
The first change that would need to be made is to enable CloudStore on the Firebase project, and then I would need to initialise it in my Manager class. I would then need to create a new function on the manager to upload a selected file, I would probably use the UID of the report to create a sub-folder in CloudStore and then add files with sequential names, such as "file1" and "file2". This would make it very easy to access the files when a report needs to be viewed, as I can use the UID of the report to locate the files.



To upload the file, I need a reference to where the file is stored on the user's computer. To do this, JavaScript provides a file API that allows you to create a pop-up dialogue box for users to select their file with. This is like how you might save a file in an application like Word or Notepad. Once I have the reference, I can pass that to Firebase which will upload it to the server.

The rest of the functionality relies upon how the UI is designed. It would be quite easy to implement a system where the code goes and fetches the files from CloudStore at the same time it's fetching the data from the database. Rather than downloading the files, the code can just fetch a URL where they are available. This URL can then be embedded in a component and displayed to the user.

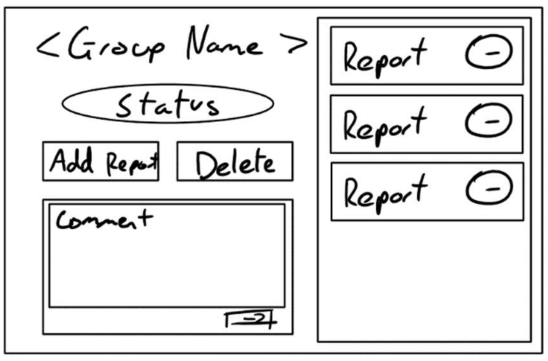
Grouping Reports



A major admin feature that hasn't been implemented is the ability to group similar reports and deal with them as one homogenous clump rather than changing the status of each one individually. To achieve this, the best way is to create a new section in the database where I can store a list of which reports are currently assigned to a certain group.

Potential, the data structure for this part could look like the diagram on the left, where we have a parent that holds a key-value list of current groups. Each of these groups would have a unique identifier (this is the key) and a set of properties (this is the value). These properties could include but are not limited to: the date the group was created, the display name of the group, and the current status.

For the group feature to be used, I would need to create a page to display a list of the current groups. This could be done in a similar way to how I display the reports on the Homepage. I actually have a specific component that contains all the functionality for those two columns (see the Homepage section of the Design chapter for more info). This could quite easily be adapted so it can be used to display groups instead of reports, as they have a fairly similar data structure.



I would also need a page to view a group, at a very minimum, this would need to display the reports assigned to that group, have buttons to add and remove reports, and a button the change the status of the reports. To the left, you can see a potential layout of this page, which has a clear structure with large buttons, making it easy to use. Each report in the list would have a remove button instead of the status indicator used in other parts of the project.

Most of these components already exist in other parts of the project, so I could easily borrow and adapt them to this new page, so the amount of work for a developer to create this feature is relatively low and is mainly focused on the backend rather than the UI design.