Queen Mary
University of London

Behaviour Support App

Mobile Application

Project Report

Programme of Study: MSc Computing and Information Systems

Author: Pete Holdsworth

Supervisor: Dr Anne Hsu

Submission Date: 22nd August 2018

School of Electronic Engineering and Computer Science

Queen Mary University of London

Table of Contents

Abstract 4

Background 4

- Introduction 4
- The Challenge 6
- Review of Existing Potential Solutions 8
- Review Summary 13

Requirements 14

- Stakeholders Analysis 14
- Understanding Current Issues 15
- System Requirements 16
- Usability Requirements 16

Design 17

- Interface styling and user experience principles 17
- Front End 18
- Back End 25
- Architecture 27
- Development Tools 28

Process 29

- Requirements Development 29
- Planning 29
- Prototyping 30
- Minimum Viable Product 31

Implementation 35

- Project Structure 35
- Interface Builder 35
- External Libraries 36
- Sprint 1 37
- Sprint 2 41
- Sprint 3 46
- Sprint 4 47
- Sprint 5 49

Testing 51

- Usability Testing 51
- Implementation (2) 53

User-Experience Evaluation 55
Implementation (3) 59
Usability Testing (2) 61
Implementation (4) 61
Conclusion 62
Learning Points 63
Considerations for Future Development 65
References 66
List of Figures 67
List of Tables 69
Appendices 69

Disclaimer

This report, with any accompanying documentation and/or implementation, is submitted as part requirement for the degree of *MSc Computing and Information Systems* at the University of London. It is the product of my own labour except where indicated in the text. The report may be freely copied and distributed provided the source is acknowledged.

Abstract

This project has aimed to utilise the advantages of mobile-device systems to improve an existing system of behaviour-support tools at St Ann's School for children with complex profound to severe learning difficulties. A mobile app was developed for deployment on the School's proprietary iPads, which are mounted on classroom walls and carried by members of staff. The application is used to record observations of child behaviour during the course of each school day, and provide analytics tools for useful insight into behavioural patterns. Teachers and staff can record a 'Red', 'Amber' or 'Green' (RAG) assessment for every child after each of the day's seven school-periods and also record data relating to any 'incident' of extreme behaviour that may occur. The recorded data is analysed by the application to help identify patterns in behaviour for different children and groups, correlations between teacher's 'Red', 'Amber' and 'Green' observations in class and occurrences of incidents.

Background

Before launching into an exploration of the software Application and its development, it is worth providing some context. This section of the report aims to help explain why there is a practical need for ICT use in general behaviour-support practises in schools and why *special* schools have differing behaviour-assessment requirements to *mainstream* schools. Finally, we will discuss the specific problem to be addressed at St Ann's School.

Introduction

Effective assessment of pupils in Special Schools can be very different than in most mainstream schools. One of the main indicators of successful student progress in mainstream schools is academic progress - in most cases, the manner in which children are conducting themselves takes less of a priority than their performance in academic tests - which are ultimately designed to prepare children for the world of work ahead of them. In contrast, some students in Special Schools do not benefit much at all from this kind of academic progression. If a student's learning difficulties are so complex and profound that their literacy has progressed only as far as being able

to write their own name - in the entirety of their school career - then the mainstream approach to assessment of progress is probably unsuitable for them. For these students, a lifetime of work and independent adulthood might not be their expected future, so their time in school is used more appropriately to help them learn how to interact with other people, how to behave appropriately in social settings, how to maintain a positive mood, avoid distressing situations and lead an overall happier life.

Observation is one of the most frequently used assessment instruments in school settings; both naturalistic and systematic observation have proven to be useful in developing theory and practice related to assessment and intervention of student behaviour (Hintze, 2001). In contrast to naturalistic observation, where pupils are observed without a pre-determined set of behaviours or observation-points in mind, *systematic* observation requires assessment of behaviour related to a pre-defined set of criteria. Collection of quantitative data - as in systematic observation - lends itself particularly well to automated computational analysis. However, historical methods have often favoured paper-based, human methods - neglecting the potential to use ICT to enhance data analysis.

The discussion on how ICT can be used to enhance pupil progress in schools in not a new one. Semmel (1978) states that "few attempts have been made to explore and evaluate the potential contributions of computers in developing observation procedures for improving trainee interaction skills”. Several decades later and research in this area is still in its infancy, however, some organisations have been taking steps to improve this situation - as will be seen here.

Behaviour-tracking solutions for mainstream schools can remain fairly rigid/uniform and still be used effectively across different institutions as their students have relatively similar needs, academic performance and behavioural traits. Special schools, however, can have very specific requirements that can’t easily be catered for by adapting existing solutions, and the lower demand for services means that the development of data-recording tools has remained slow and the developmental costs high.

The Challenge

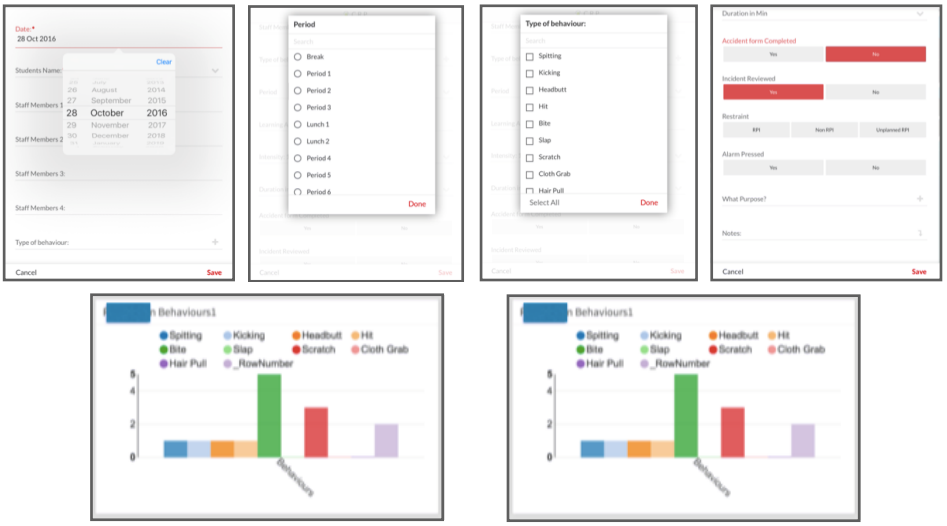
St Ann's School is a special school for ninety students aged 11 to 19 years with complex profound to severe learning difficulties. As part of their Positive Behaviour Support policy the School drives forward the use of positive and proactive approaches to support behaviour management and avoidance. Through understanding of how and why challenging situations may arise for each individual, the School tries to set up a behaviour support plan that aims to resolve behaviours before they arise.

Assessment of challenging behaviour in schools is a complex task. To assist in the process, the National Institute for Health and Care Excellence (NICE) provide several guidelines for conducting assessment, including that the person being assessed remains at the centre of concern and is supported throughout the process, and that the assessment is relative to the severity, impact, frequency and duration of the behaviour (NICE, 2015). In line with NICE's guidelines, and with greater focus on children with learning disabilities, The Challenging Behaviour Foundation (2018) has developed a range of Positive Behaviour Support tools designed to support children in practising positive behaviour, rather than 'fixing' the child or punishing the challenging behaviour. This approach is reflected in St Ann's School's (2017) Positive Behaviour Support policy.

As Deputy Headteacher responsible for Positive Behaviour Support, Timothy Holdsworth has been championing the use of ICT to support such behaviour analysis for several years. By moving away from the analysis of purely anecdotal evidence and towards the gathering of more objective data, he aims to reduce subjectivity and look for statistical patterns. Over the past few years he has developed a set of tools for staff to record information about classroom behavioural performance (RAG Assessments) and behaviour incidents. The incident data collected includes - the physical severity (such as biting, scratching, punching etc) and the emotional intensity (what is considered a normal intensity for one student might be very abnormal for another) - as well as some metadata (timescales and all the parties involved). The collected data is then analysed to identify trends in individual and school-wide behaviour.

So far this approach has been lauded by the School's stakeholders, other professionals involved within the special schools and disabilities community and last year featured in a publication of The SLD Experience - a journal distributed by the British Institute of Learning Difficulties. Imray's (2017) article, *Listening to behaviours: adopting a capabilities approach to education* commends the School's strategy and use of ICT tools to support data-driven analysis.

Figure 1: *Data capture forms and reports for the current system*



Despite the positive start to this new approach, the initiative faces considerable challenges. Currently, data is recorded in several different places - different forms or 'tools' made using Google Docs and Sheets. The tools are accessible from any web browser, but require several steps to navigate to them and are optimised for use on desktop devices. These obstacles are affecting the quality of the data being gathered. As the tools are not so easy to find, some staff at the school - who may not be particularly proficient computer users - are reluctant to use them or avoid them altogether. Even though iPads are widely used at the School, because the tools are optimised for desktop devices data recording is sometimes delayed until later on when access to a desktop becomes available. This delay risks degradation of the quality of collected data. If staff had better access to the tools, regardless of their computer-literacy level, the data gathered would be richer and more accurate, providing better insight into behavioural patterns.

In addition, the data is currently processed manually, straining on staff resources. As they develop the tools further, data requirements are also increasing. Automating the reporting process would enable staff to spend more time exploring ways to model the data and find ways to leverage the information to support the needs of School attendees.

A mobile-first system could enable a more discreet interaction between users and the system - with minimum disruption to their real-world activities. Harrison et al (2007) stated that the most recent paradigm of Human Computer Interaction (HCI), strives for better "support for action in the world" - as opposed to obtrusive and *explicit* interaction. Through the use of Multi-touch, Physics and Gestures (MPG), users can have a more natural and seamless user experience through mobile devices - known as *implicit Human Computer Interaction* (Poslad 2009) - therefore completing tasks more easily and quickly. Enhanced connectivity and utilisation of wearable devices could also provide richer data-gathering capabilities.

It was identified some time ago that a mobile-first system could resolve the accessibility and automation issues discussed here and take full advantage of the native features of modern mobile devices. Engineering the solution from the ground up would give increased design freedom, avoid the constraints of the Google Docs/Sheets currently performing tasks they are not specifically intended for and eliminate their unused tools/features that confuse users and waste space.

Review of Existing Potential Solutions

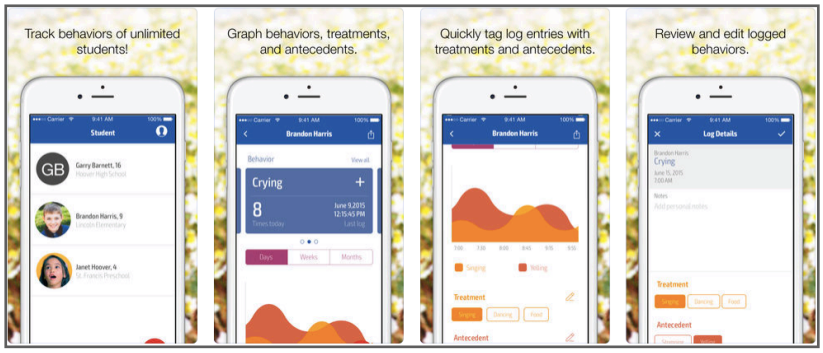
The first and most logical step in the development of a new system is to review any existing applications that might either provide an acceptable solution or help generate ideas for a bespoke design. As the School's mobile devices are all Apple iPads, the following applications were all picked from the Apple App Store.

App 1: **Best Behaviour** (eKrios Consulting, 2017)

eKrios Consulting's **Best Behaviour** app is designed for parents and teachers to record occurrences of different behaviours for children in their care. Users can add multiple children to the app and can add their own custom behaviours to record occurrences of. When the user records a behaviour the data is reflected in a visual chart to show occurrences over time. These features provide some of the functionality needed to support St Ann's behaviour-support system, but fall short of providing a viable solution. The structure and design of the student list (with images for even quicker identification) is clear and appealing, but in a school environment - where

students are organised into and managed in different classes - it would be necessary to sub-categorise the students. The user must also navigate into each student individually to record data, which would be time consuming if entering data for an entire class.

Figure 2: *Best Behaviour - User Interface*



While the ability to add custom behaviours provides advantageous flexibility this data is not detailed enough to provide the depth of data gathering and analysis potential that the School is currently achieving. Behaviour incidents need to be recorded with more information about who was involved, intensity/severity, reasons for the behaviour and so on. There is also no way to accommodate the Red, Amber, Green assessments for school-day periods.

Visually, the app is well designed with good use of blocked colours, a clean layout, simple navigation, and the familiar appearance of Google's *Material Design* (screen objects are given similar characteristics to physical design objects. eg. pieces of paper/cards - with realistic shadows and movement). The app also makes good use visual data representation. The charts used to display data over time are an effective way to convey information in a more compelling and engaging way than text alone

Table 1: *Best Behaviour app - advantages vs disadvantages*

App1: Best Behaviour	
Advantages	**Disadvantages**
• Clear and organised representation of entities • Clean layout and colours • Google's *Material Design*-style appearance makes for an intuitive user experience • Visual data representation	• Not detailed enough data recording for incidents • No provision for grouping entities • Slow navigation if recording data for groups of entities • No realistically feasible way to store Red, Amber, Green school-period assessments

App 2: **Autism Tracker Pro** (Track & Share Apps, 2018)

Figure 3: *Autism Tracker Pro - User Interface*

One particularly useful feature of this app is the ability to record severity of behaviour incidents, as well type of behaviours observed. However, it falls short of providing the level of detail the School needs. Incidents or behaviours are recorded as either occurring or not occurring on a given day - without ability to record the time of day (or school-period) that they occurred in. The app also provides the ability to record Red, Amber or Green assessments, but unfortunately these are too focused on the child's 'aggression' and are used for recording behaviour severity - the School needs to assess both school-period performance *and* behaviour incident severity. The app is also designed to be use to support a single child - not multiple students, let alone organised classes. From a usability perspective, the app's User Interface (UI) is a mixture of crowded and cluttered screens, with some fonts particularly hard to read as they are very small, and screens that are so sparsely filled that it is not easy to understand what is being presented or how to navigate them.

Table 2: *Autism Tracker Pro app - advantages vs disadvantages*

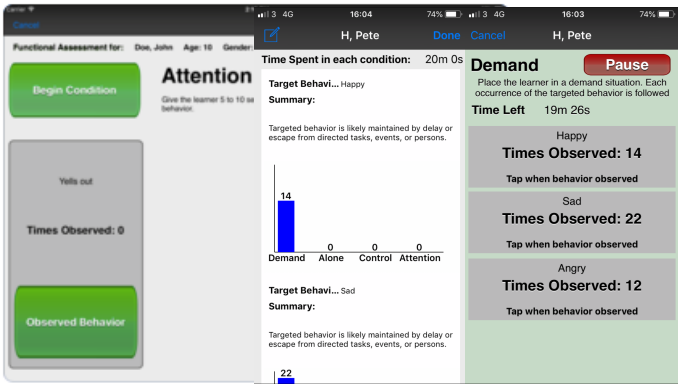
App 2: Autism Tracker Pro	
Advantages	**Disadvantages**
• Colour-coded representation of observation data (Reds, Ambers, Greens) • Visual representation of data where appropriate	• Designed for use with single child • Unbalanced use of screen space • Drab and clinical layout and colour scheme • Does not fulfil data recording requirements

App 3: **Functional Behaviour Assessment Wizard** (WhizzWhat Software, 2017)

WhzzWhat Software's **Functional Behaviour Assessment Wizard** allows users to add entities (children) to storage and define their own behaviours to observe and run observation sessions. The children are observed in a timed period and the user taps a button to record every time they observe the anticipated behaviours. The app then presents some simple charts to visualise the

frequency of occurrences of each behaviour. While this is useful if studying individual children under test conditions, it falls a long way sort of fulfilling the school's requirements.

Figure 4: *Functional Behaviour Assessment Wizard - User Interface*



The aesthetic design of the app is quite poor - it uses a drab colour-scheme, varied font-sizes and weights and has an unintuitive design - it can be hard to understand the overall layout of the app's navigation. However, it does make good use of instantly recognisable native iOS User Interface components - such as the CocoaTouch library's standard UISwitches, UIPickerViews and UITableViewCells and UIButtons. While they could have been formatted and positioned a lot better, their immediate recognition-value means that functionality will be quickly understood by the user.

Table 3: *Functional Behaviour Assessment Wizard app - advantages vs disadvantages*

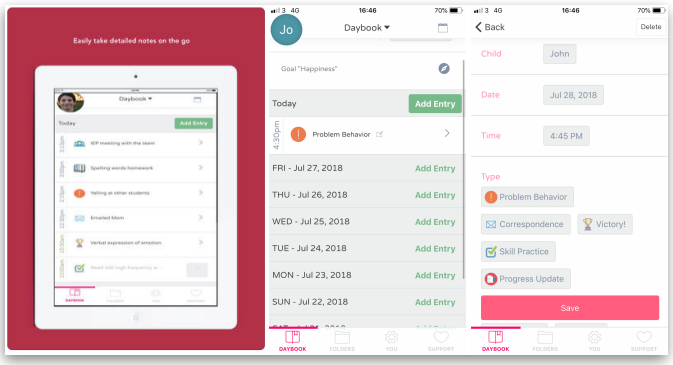
App 3: Functional Behaviour Assessment Wizard	
Advantages	**Disadvantages**
• Easily recognisable native iOS screen components convey functionality quickly	• Does not fulfil data recording and analysis needs • Poor aesthetic design

---

App 4: **Birdhouse - For Special Education Teachers** (Birdhouse, 2016)

Birdhouse's **Birdhouse - For Special Education Teachers** has a well designed User Interface. Navigation is smooth, seamless and logical - with good use of space, colours, screen components, appropriate imagery and icons. The app responds as expected to all inputs and provides good feedback (eg. displaying alerts when use cases take an alternative path or when loading data). The organisation of the

Figure 5: *Birdhouse - For Special Education Teachers - User Interface*



interface and navigation layout is familiar and easily learned - making good use of a Tab Bar across the bottom.

Functionally, the app appears to work well and receives high user-review ratings on the App Store. The app allows the user to record many different aspects of an incident, as well as make notes for others to read and there is a strong emphasis on the sharing of data between users - teachers can share data between themselves and with parents at home. However, there are a lot of features that are unsuited / irrelevant to St Ann's School so they would fill the app with unused clutter for users, and there is still no convenient way for teachers to record RAG assessment data for whole groups (classes) of students at regular intervals in the day.

Data analysis is limited to individual children and visual presentation of charts is poor compared with the rest of the UI. However, the app does allow users to export data in .csv format for use in external software packages for further analysis.

Table 4: *Birdhouse - For Special Education Teachers app - advantages vs disadvantages*

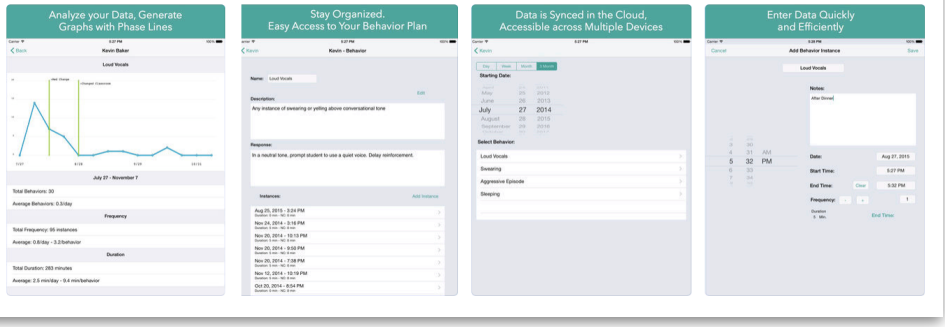
App 4: Birdhouse - For Special Education Teachers	
Advantages	**Disadvantages**
• Elegant, simple and appealing UI and UX • Detailed data recording facility for behaviour incidents • Multi-tenancy system design enables good collaboration on data • Data exporting capability allows users to extend analysis externally	• Lots of redundant features • No convenient way to record RAG assessment data

App 5: **Nulite Behaviour Tracker for Special Education** (Stephen Moy, 2016)

Stephen Moy's **Nulite Behaviour Tracker for Special Education** enables users to record detailed information about behavioural incidents and is (in theory) flexible enough to accommodate the recording of RAG assessments. However, most of the useful data is recorded in free-text format. The app does a good job of recording detailed metadata, such as dates and durations, but

as the rest of the information is recorded in free-text, it is much more difficult to analyse. As a result the app's chart feature fails to provide much useful insight into the data.

Figure 6: *Nulite Behaviour Tracker for Special Education - User Interface*



The application's UI design is adequate for use in a formal setting, but it is very clinical, with mostly drab grey colours and boxy screen components. It does not make the best use of space, and is not pleasing to use, however it is functional and usable.

Table 5: *Nulite Behaviour Tracker for Special Education app - advantages vs disadvantages*

App 5: Nulite Behaviour Tracker for Special Education	
Advantages	**Disadvantages**
• Detailed data recording capabilities • Simple features scope - without needless or redundant features • Clear and well presented charts • Functional interface design	• Unable to analyse data appropriately in the app, and data exporting option is made complicated by the capture of data in free-text fields. • Uninspiring interface design

Review Summary

It is clear that there are no suitable options that would satisfactorily replace the current system. Each of the applications reviewed has some advantageous features to consider, and some have particularly pleasing usability aspects to take note of, but none is able to offer a viable alternative to the current system's data-recording features. This means that development of a new, bespoke solution is worthwhile.

In summary, the advantageous design elements and features identified for a bespoke system are as follows:

1. Ensure data-recording features are appropriately detailed
2. Use clear and organised lists to represent entities (classes/staff/students)

3. Make use of Google's *Material Design*-style appearance for a familiar and intuitive user experience
4. Use well-presented visualisations (charts) to present data where appropriate
5. Use colour-coded symbols where possible - especially for recording observation data (Reds, Ambers, Greens)
6. Use easily recognisable native iOS screen components to quickly convey functionality
7. Favour a simple and appealing User Interface over complex and busy screen layouts
8. Ensure a positive and seamless User Experience through simple and logical navigation, efficient error-handling and informative user feedback
9. Use a multi-tenancy system design to enable positive collaboration on data
10. Enable data exporting capabilities so that users can extend analysis externally
11. Include only useful features (no need for redundant features)

Requirements

As the project sponsor and leader of Behaviour Support in the St Ann's School, the Deputy Headteacher was the main source of information when developing requirements for the project. Several unstructured interviews took place to understand the context of the current problem, the ambition of the School for the project, the available resources and the stakeholders involved. The author also visited the School to meet with staff and better understand the physical environment. Through the course of these meetings the author and sponsor were able define a set of project requirements that would ensure delivery of a useful system. This section of the report discusses the outcome of the requirements development exercise

Stakeholders Analysis

The primary stakeholders for this project are the staff at St Ann's School - they will be the ones using the application directly. As well as improving their experience of using the data gathering tools, this solution should enable them to take better preventative and support measures - which could improve the quality of their work day and improve job satisfaction.

Secondary stakeholders include the students themselves and their families and/or carers. Where appropriate they may be consulted for input or feedback and could influence the decision making of the School. The students are the ones who will benefit most from the success of the system. By enabling staff to take better preventative measures, student's should experience fewer, and less severe, behaviour incidents at the School and should feel happier and safer as a result. Obviously the experience of the students at St Ann's has a direct affect on the lives of their parents and/or carers - both from an emotional perspective and from a practical perspective (learning how to avoid extreme behaviour).

Tertiary stakeholders for this project include potential investors in the completed solution and other members of the special schools and disabilities communities. While they have no direct influence over the decision making for this project, it is still important to consider their potential interests, where possible, to minimise the need to re-engineering aspects of the product if the School proceeds to invest more into it and share with other organisations.

Understanding Current Issues

The purpose of this project is to design and develop an improved system for collecting and processing the data. As discussed in 'The Challenge' section, there are several technical, system issues that are having a negative effect on the quality of data and usability of the tools. To summarise these points, the main concerns that are to be addressed by the implementation of a mobile-oriented system are as follows:

- Data is recorded in several separate systems and needs manual consolidation for processing and analysis
- The systems are only properly accessible from non-mobile, desktop devices so recording observations can be delayed at the expense of data accuracy
- Access to the system is convoluted and slow so some users tend to delay recording observation and record data in batches later on at the expense of data accuracy

- Some low-skilled users find it difficult to access and use the system so minimise their contact with it at the expense of quality data collection
- Reports are generated manually, which can be time consuming

System Requirements

The system will:

1. Enable access from Apple iOS mobile devices (iPads)
2. Support portrait and landscape screen orientation - as iPads are fixed to classroom walls in landscape-orientation, but staff usually carry them in portrait-orientation
3. Provide a single, secure data storage location
4. Enable recording of all currently recorded data points for both RAG Assessments and behavioural Incidents. including:
    - Red, Amber, Green (RAG) assessment for each child for each school-day period
    - Incident start time, duration, student and staff involved, behaviour-type, reason/purpose for behaviour, intensity/severity of incident, type of restraint used, whether a school accident form was completed, whether an alarm was raised, and additional notes.
5. Provide data analysis capabilities and rapid presentation of analysis results
6. Provide data exporting capabilities - for further external analysis
7. Have potential to scale up appropriately to accommodate increased future usage and implementation of additional functionality

Usability Requirements

The system will:

1. Improve Human Computer Interaction (HCI)
2. Provide a quick access for recording observation data
3. Provide simple access to services to accommodate low-skilled users
4. Have a clean, simple to use interface
5. Improve collaboration between users on data
6. Provide a smooth, seamless and unobtrusive experience for users

7. Use visual representations where appropriate to convey meaning clearly

Design

This section of the report presents the proposed bespoke solution ideas put forward, including user interface, system architecture, supporting technologies and tools. All designs were developed with the objective of adhering to good interaction design and achieving usability and user-experience goals - such as being effective to use, efficient to use (with good utility), easy to learn and easy to remember how to use (Preece et al, 2002) - and avoiding, among other things, visual clutter, incomprehensible components, annoying distractions, confusing navigation, information overload, design inconsistency (Galitz, 2002).

The design proposed in this section is the full concept for the system. Most of the features described are being delivered in this project, however, some features have been left out of implementation (or adjusted) for this initial release. In the next section we will see how this concept was analysed to identify a Minimum Viable Product (MVP) for the application's first version, which is necessary to ensure that a realistic implementation target can be set for the deadline of this first iteration (the end of this project), and to maintain an agile approach to development.

Interface styling and user experience principles

To maximise user-acceptance, usability and, therefore, usage of the system, the author has aimed to keep the user interface as simple, clear and logical as possible - with large components where appropriate (eg. buttons and table cells) and 'flat' navigation structures to make access easy for users of all ICT-skill levels. As identified in the *Review of Existing Potential Solutions* above, Google's *Material Design* approach is a good choice for ensuring a positive user experience. It is used in many Google products, on countless Android devices and in Google docs, which the current system is built on. Following this design pattern should make the transition to a new system smoother, without users needing to learn a new set of design characteristics. The proposed design therefore incorporates uniform formatting of interface objects with shadows set to indicate height from the ground, providing association of grouped objects (and differentiation of

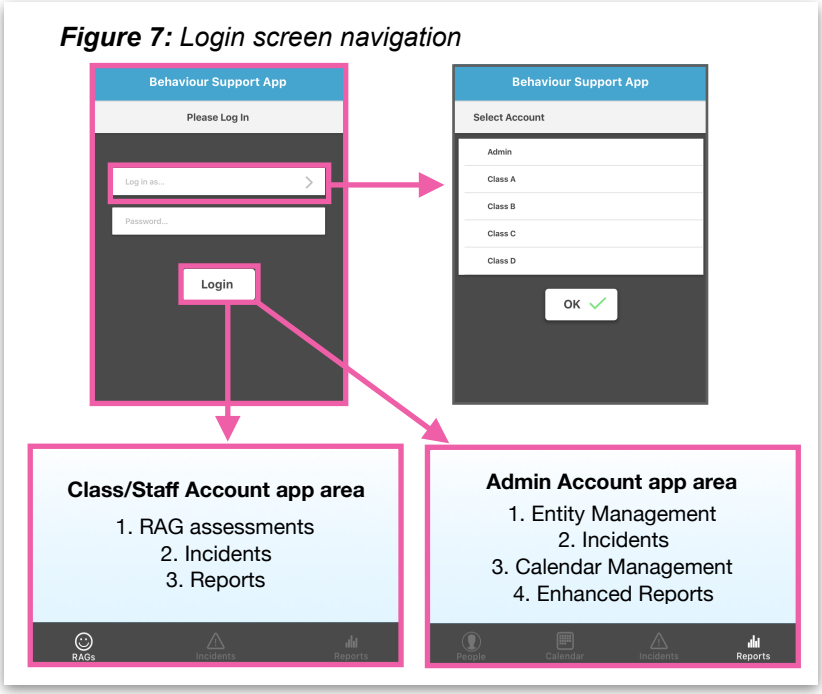
separate objects, eg. action buttons). Bright block colours are also used throughout, balanced with clear white objects and paper-like ‘cards’ for presentation of data. Space is used in a balanced way to frame key components and leave screens uncluttered for high-pressure situations, such as recording observations immediately after a distressing incident, or for low-skill users to access easily. Conversely, Screens that benefit from increased data presentation, and which are accessible for high-skilled users (eg. Admin’s enhanced analytics screens) are designed to maximise use of screen space and present a higher volume of data to the user.

Flattening the navigation is achieved by using Tab Bars to partition areas of the application that are on the same ‘level’, and by using moveable containers inside some single screen layouts (scroll views, swappable left/right content). This avoids the user needing to travel ‘up and down’ a hierarchical navigation structure, which can be confusing as they forget how many steps back they need to go to reach the home/root view before navigating to another area of the application.

The full interface concept can be found in the project’s Supporting Material: ‘Full_Concept_Design.png’

Front End

Figure 7: Login screen navigation



User Accounts / Login

Upon launching, the app will enable staff to select an appropriate User Account - either the Admin account or a School-Class account. The user will log in from a dedicated screen and will be directed from there to their appropriate interface and/or data space. User accounts for each of the school classes have identical user interfaces, but only access data relating to students in their class. An ‘Admin’ account will be provided, which has a different interface - without ability to record RAG assessments as this is performed by classroom staff, but with added functionality for

management of database entities (Classes, Staff Members, Students) and also enhanced reporting / analytics features. Authentication features will be provided for secure access to user accounts - password protection, with password management of all accounts available to the Admin account.

Classroom/Staff Accounts application area

For users logged into a school-class account or as a regular member of staff, the app has 3 sections - separated using a familiar iOS **UITabBar**. The first tab contains content for recording **RAG Assessments;** the second tab contains content for recording **Incidents** data; and the the third tab contains content providing some useful analysis - **Reports** and charts - targeted at the logged in user.

Admin Account app area

For the Admin account, the **UITabBar** provides access to different content than the class/staff accounts. The first tab contains entity (**People**) management features; the second tab contains a **Calendar** management view (for filtering active dates around holidays, weekends, term time etc); the third tab contains content for reporting **Incidents** data (same as for class/staff app area) and the fourth tab contains enhanced **Reports** for more in depth analysis for decision makers.

<u>RAG Assessments Screen</u>

The **RAG Assessments** screen for class/staff user accounts will present a list of 7 buttons - representing the school periods of each day. Each button will display a status for whether the RAG Assessment for that period has been completed or not (programmed to display as ‘Not Complete’ in red-colour - conveying urgency - if the user is late in completing it). The list of periods will automatically be set to represent the current day - as indicated by date labels at the top of the view. To the left and right of these date labels, there will be buttons to signify and enable navigation to alternative dates (initially the right-hand date button is greyed-out -as seen below - to show that it is not possible to navigate to future dates). Navigation between dates will happen inside this single view (using animated containers - holding separate sets of 7 school-period

buttons) to maintain the perspective of ‘flat’ navigation. Buttons will have action indicator arrows on their right-hand side to help users intuit that tapping the button leads to further functionality.

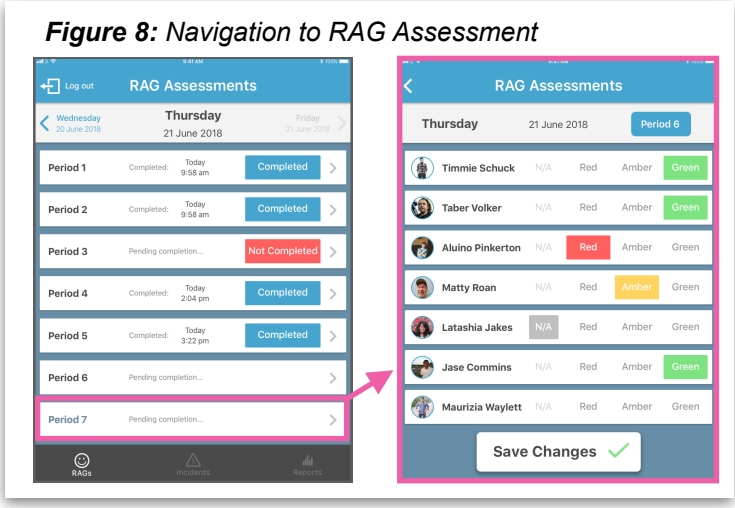


Figure 8: *Navigation to RAG Assessment*

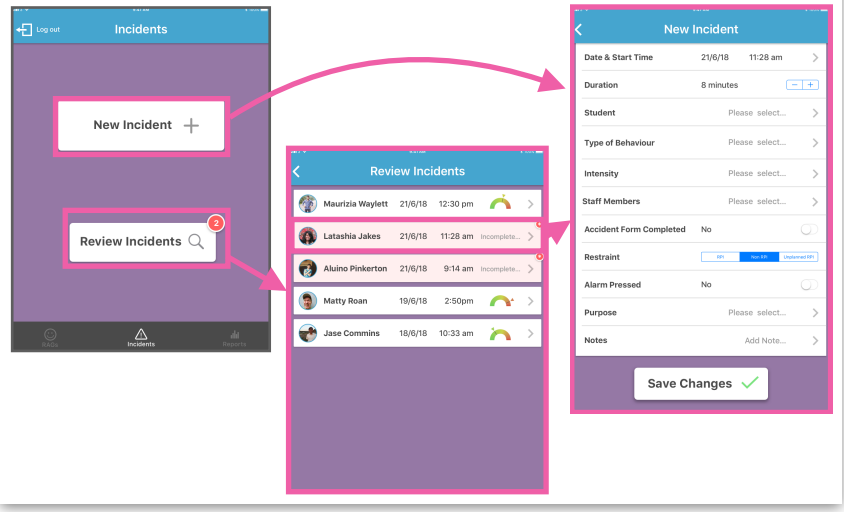
When a period-button is tapped the view navigates to a new screen where a list of students is presented (students are filtered to show only those who are associated with the class/staff member currently logged in). Here, the user can record RAG Assessments for all class students in a single screen - without having to record individual RAG Assessments for each student, navigating back and forth between them. This reduces the potential of performing the use case ‘Complete RAG Assessment’ repeatedly - instead allowing staff to perform the use case only once per period. This minimises the application’s obtrusiveness in the user’s day-to-day activities, improving the user experience and providing a more discreet HCI. Each student has four buttons associated with it - representing Red, Amber, Green and ’Not Applicable’ (eg. student was absent) assessments. When the user taps one of the buttons, it will be highlighted with appropriate colour-coding. If a user changes their selection, the highlighted buttons will animate in a compelling way to support a seamless, fluid interaction experience. When a RAG Assessment is completed, the user taps the ‘Save Changes’ button to commit the data. The app will check to make sure the assessment has been completed fully before committing to the database and provide appropriate feedback (in the form of **UIAlert**s) if data is incomplete or problems occur when updating the database. This helps prevent errors, avoids data inconsistencies in the system and keeps the user informed of system status.

Incidents Screen

The initial **Incidents** screen (also accessible from the Admin account’s Tab Bar) has been kept as minimal as possible - containing only two large buttons - to avoid visual distraction when recording data in a potentially stressful situation. The main button (positioned at top and largest) will take the user to a new, blank form for collecting Incident data. The secondary button (bottom) will take the

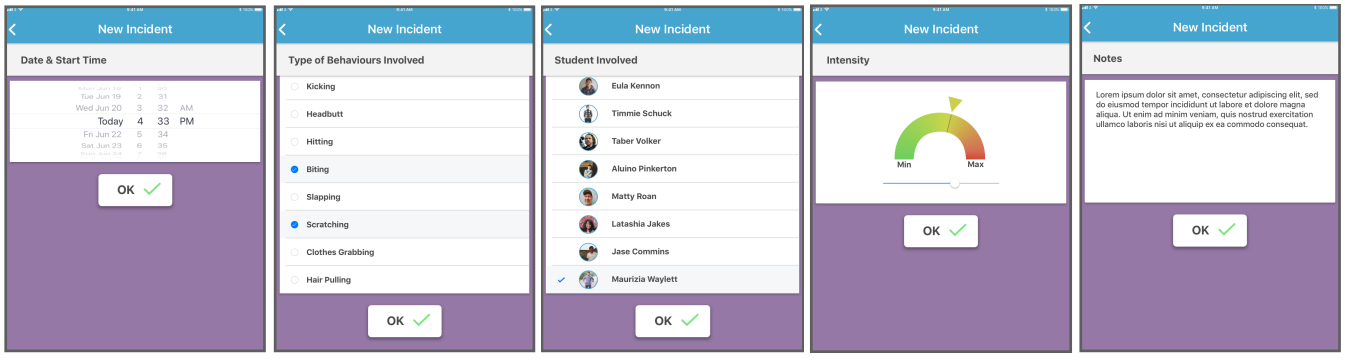
user to a view of historic incidents. If the user has tapped on the 'Review Incidents' button they are presented with a list of past incidents in chronological order (most recent first), with a few key pieces of data highlighted - including the name and profile image of the student involved, the date and time and the recorded intensity.

Figure 9: *Incidents Screen - navigation to Incident Form*



For each of the cells in the incident form, the user can record data about the incident. Some data can be input/updated from the cells themselves - where a yes/no value, incremental number, or low-choice value is needed - using familiar and easily recognisable iOS components (UISwitch, UIStepper, UISegmentedControl). Other data input requires a more complex process so the view will segue to a new screen dedicated to that data input if the user taps on one of the cells marked with a placeholder 'Please Select…' or 'Add Note…'

Figure 10: *Example of screens for value-selection for Incident Form*

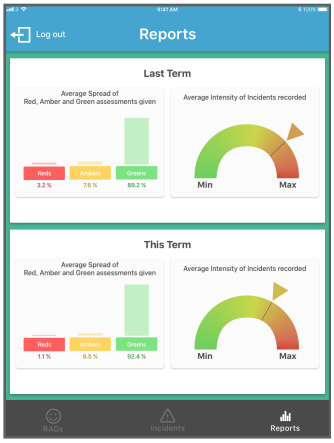


Each of these dedicated screens has either a date/time-picking tool, a list of values or objects (staff/students) to choose from, a text area for writing more detailed notes, or a custom 'intensity-indicator' chart with a slider control (to give a more compelling visual representation of the incident's intensity). Once data has been input or selection has been made, the user taps the 'OK' button to navigate back to the incident form, which will then be populated with the new data.

As with the RAG Assessments screen, the user commits entered data by tapping the ‘Save Changes’ button at the bottom of the incident form (with the app checking for data completeness and using Alerts to provide feedback and inform the user of system status for data uploading).

Class Reports Screen

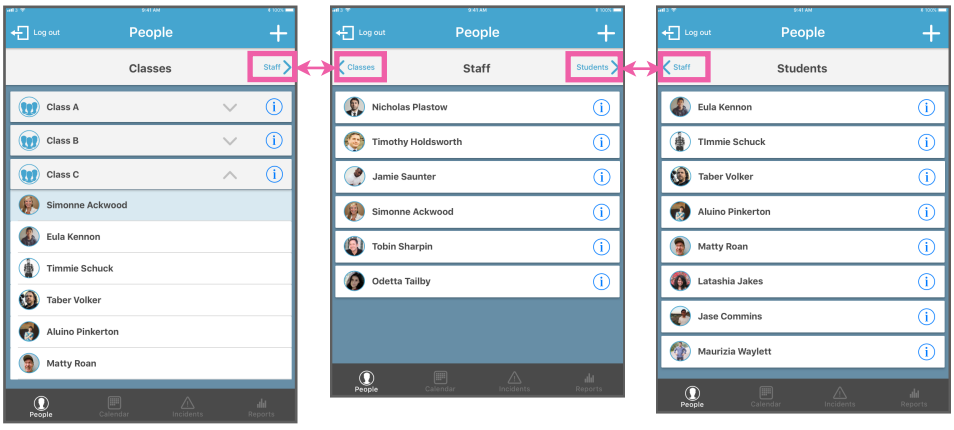
Figure 11: *Class Reports screen*



The **Reports** screen for class/staff accounts is a simple single screen display of two comparative views of data associated with current user. The top half of the screen shows two charts with data from a previous time frame (eg. Last Term), and the bottom half of the screen shows the same charts but with data from a more recent time frame (eg. This Term). This enables users to compare current class progress with historical progress. In both the top and bottom views, the left-hand chart displays the average number of Reds, Ambers, and Greens awarded for the class in the associated time period, and the right-hand-side chart shows shows the average intensity of incidents recorded for students in that class and in the associated time frame.

People Screen

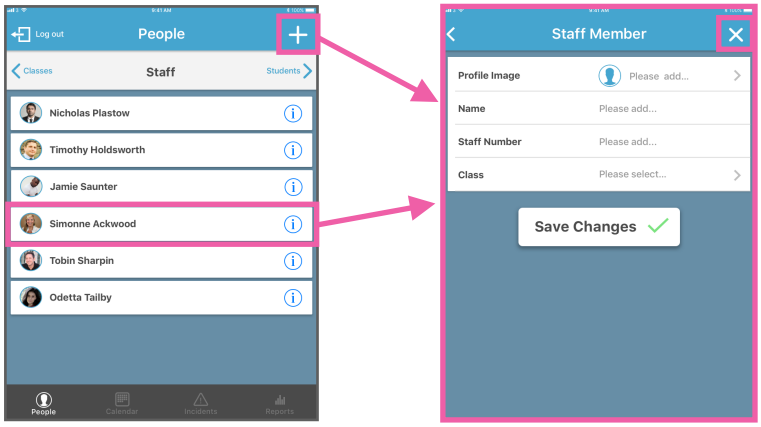
Figure 12: *People screen entity-container navigation*



The **People** screen in the Admin account consists of three separate containers with a list of entities in each one. The user can navigate between containers by tapping on the button in the subtitle bar - labelled according to the entity (Class, Staff, Students). These buttons animate the containers on and off screen - preserving the ‘flat’ navigation structure and managing the available screen space.

The initially displayed container has a list of school classes contained in the system, with any associated staff members or students listed underneath it in a collapsible, drop-down view. The next container (to the right) shows a list of all Staff in the system, and the final container, to the right again, shows all Students in the system. Staff and students are represented with their name and profile image, and all entity cells have an ‘i’ symbol to their right-hand side to indicate that information is available for that entity if the user taps that cell.

Figure 13: *Navigation to entity-details form - for existing or new entities*

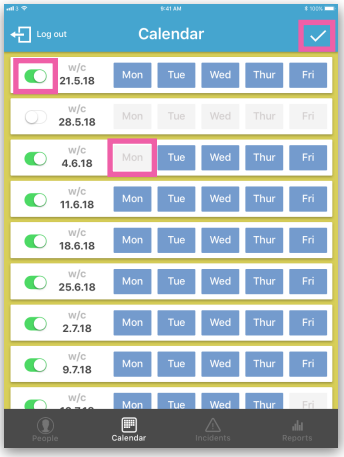


Tapping on any of the cells, or on the ‘+’ (plus) symbol in the screen’s top right corner will segue the view to an entity-details screen. If the user has tapped an existing entity it will be populated by data for that entity, otherwise the fields will be empty. When the user has completed all necessary fields, they tap the ‘Save Changes’ button to commit the new or updated data to the database. If the entity already exists, the user can tap the ‘x’ symbol in the top-right of this screen to delete the entity from the database.

Calendar Screen

The **Calendar** screen provides a way for the Admin user to activate/ de-activate certain days and weeks of the calendar that the class users can add RAG Assessment data to. This will help to avoid users adding data to incorrect days in error - such as committing RAG Assessments to the database for non-school days (weekends / holidays / staff-training days).

Figure 14: *Calendar screen*



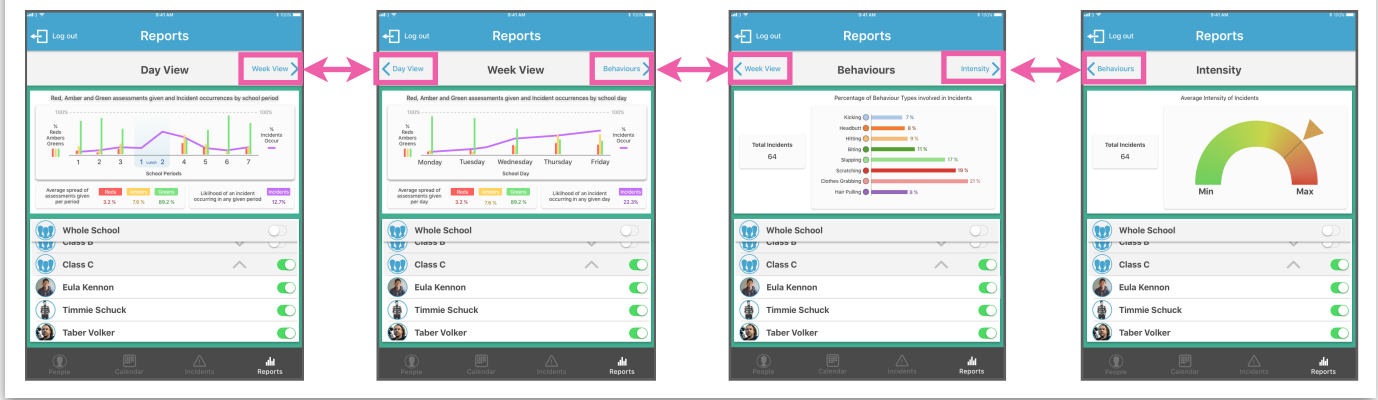
Weeks of the calendar year a represented by a row in a scrollable table, with each cell displaying a button for a day of the school week (Mon - Fri). The user can tap each individual day (useful for selecting public holidays / staff-training days) to toggle whether it is active and available for

committing data to (highlighted blue) or not (greyed-out). At the left-hand side of each cell there is also a switch to activate/de-activate entire weeks at a time (useful for indicating holiday periods. To save changes to the calendar, the user can the 'tick' symbol in the screen's top-right corner to indicate they are finished editing.

Admin Reports Screen

The **Reports** screen for the admin account consists of a list of entities filling the bottom half of the screen, and four changeable containers - each occupied by a different chart - positioned in the top half of the screen. The list of entities occupying the bottom half of the screen, which remains in position even when the charts at the top are being navigated between, enables the user to filter the chart data by turning on and off switched on each entity's cell. This way the user can add or remove, individual students, whole classes, or the whole school at once. When selections (and de-selections) are made in this list, the data in the chart above will be updated immediately to reflect the selected entities.

Figure 15: *Navigation between Admin Report containers*



Only one chart is displayed at a time - navigation between them provided by the buttons in the subtitle bar in the same way as navigation is achieved between entity lists in the *People* screen above. The first chart displays analysed data for selected students plotting RAG Assessments awarded against the occurrences of Incidents - over the course of an average day - on a combined clustered-column and line chart. The chart slices all RAG Assessment and Incident date/time by the period of the school day - so average number of Reds, Ambers, Greens awarded in period 1 (represented by red, amber, green columns) vs proportion of Incidents occurring in period 1 (represented by point of purple line data). The aim is to show any correlation between

class performance and occurrence of behavioural incidents over the course of a single day. It can help answer questions such as, ‘are there times of day that students’ behaviour is particularly likely to escalate into an incident?’ and ‘do RAG Assessments forewarn about times of day where poor class performance precedes occurrences of incidents?’ The second chart shows a different view of the same data. This time, the data is sliced by the different days of the week. It can help to answer questions such as ‘does behaviour and class performance peak at certain times of the week?’

The third and fourth charts display a breakdown of behaviour characteristics, using a horizontal bar chart to display percentage of incidents where an occurrence of each behaviour is recorded and an intensity-indicator showing average intensity of incidents.

Back End

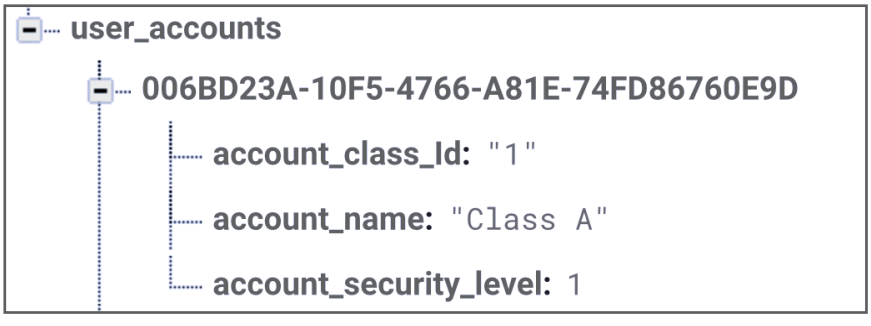
As the system is designed to share data across multiple devices, the most appropriate choice of data storage is a cloud-based data store, which can provide multiple clients access to the same, single set of data. To ensure the application remains flexible to changing future requirements, and easily scalable to accommodate higher volumes of data, a cloud-service-provider is the most appropriate option for back-end deployment. For initially low volumes of data these can be very cost-effective - compared with independent development and deployment of application servers. Should requirements change drastically in future, migration to a different type of service will be simpler and cheaper than re-engineering a bespoke server.

When considering the type of database to use, it is important to understand the criticality of the system’s accuracy - does it need to fulfil strong ACID (Atomicity, Consistency, Isolation, Durability)properties? Or will a BASE (Basically Available, Soft state, Eventually consistent) database be sufficient? In the case of this mobile application, accuracy and consistency at all times is not particularly critical (as it is in, say, a financial institution), and flexibility and scalability are important, so it is more appropriate to implement a back-end that has strong BASE characteristics.

For this project the author chose to use Google's **Firebase** service to support the app's back-end. Firebase is used by numerous successful applications and describes itself as a **Back-end as a Service (BaaS)**, which includes a NoSQL database and authentication services among other features and is hosted on Google's secure infrastructure. Installation is relatively simple using the code libraries provided with the service. Additionally, the NoSQL database features automated data caching for smooth asynchronous upload or download of data. If connectivity is lost, Firebase's upload/download methods provide the necessary callback-functions to handle incomplete transactions.

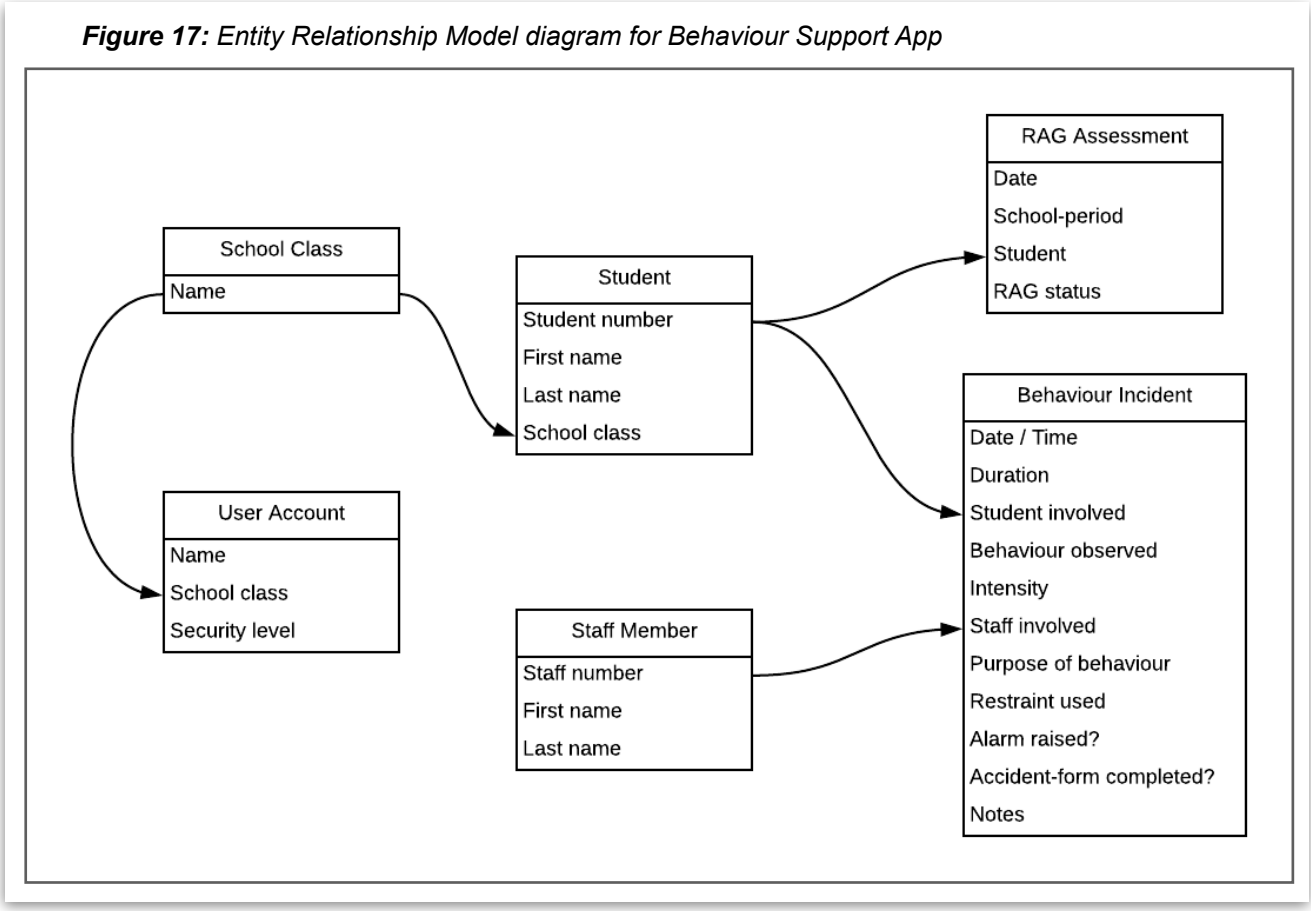
Firebase's database stores JSON (JavaScript Object Notation) documents, which behave like '<key : value>' dictionaries - where the key is a String (used to identify the document in the database) and the value is stored as 'Any' object - which can be cast locally to appropriate primitive data types - depending on the development language of the client application. Rather than storing data for predefined attributes - that must be programmed into the database when it is initially built - Firebase's database stores all data as nested dictionaries originating from a single root path in a hierarchical, tree-like structure. This way, additional data values can be associated with an entity at a later date, without needing to re-write the database. In the example below, we can see a JSON document object stored with a key of '*006BD23A-10F5-4766-A81E-74FD86760E9D*' and its value equivalent to an object containing 3 more key:value pairs <account_class_id : "1", account_name : "Class A", account_security_level : 1>
Additional attributes can be added to the object at a later time without any effect on the rest of the database - leaving greater scope for future development.

Figure 16: *Example JSON objects as stored in Firebase*



For the Behaviour Support App, the author has used an entity-relationship model, which can be represented with the following simplified diagram:

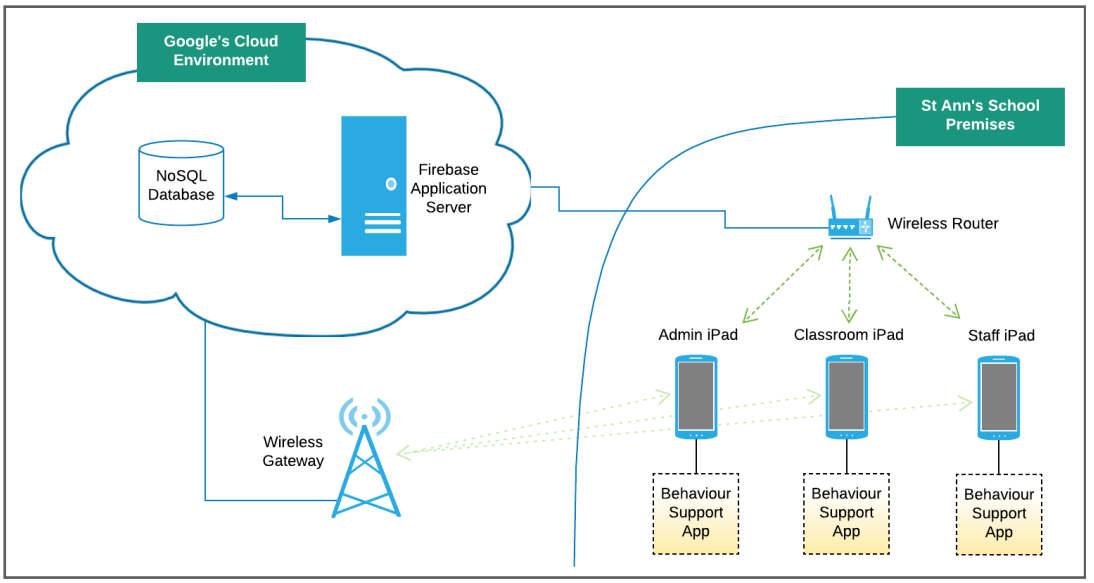
Figure 17: *Entity Relationship Model diagram for Behaviour Support App*



Architecture

The architecture for the system is fairly simple. The Behaviour Support Application will be deployed to proprietary Apple iPads at St Ann's School. These will connect wirelessly to the School's wireless router, which will provide connectivity to the Internet and Google's Cloud Services environment. The Firebase application will be configured to communicate with client applications and will coordinate access to the Firebase database - both of these entities hosted in the Cloud. If the School's connection to the Internet is interrupted for any reason, the mobile app will be able to reach the Firebase back-end through a GSM connection, provided they have been provisioned with 3G/4G network subscriptions.

Figure 18: *System Architecture for Behaviour Support App*



Development Tools

The main technologies used to develop the solution are described in the table below:

Table 6: *Development Support Tools*

Technology Type	Choice	Justification
Platform	iOS	• iPads widely available and used at St Ann's School by staff • Existing experience of developing iOS apps already
Integrated Development Environment (IDE)	Xcode	• Widely used IDE for building iOS applications • Developed and fully supported by Apple • Large online community of users • Free of cost for MacOS users • Already have experience of development with Xcode
Programming Language	Swift	• Powerful, compiled programming language for native iOS apps • Fast-growing in popularity - to supersede Objective-C as the primary programming language for native iOS apps • Open-source software supported by large online community • Extensive range of learning material available • Already have experience of development with Swift
UI Framework	Cocoa Touch	• Primary UI Framework for building iOS apps • Features tools designed to make use of the mobile devices in particular - including touches and gesture recognition • Easily incorporated into projects through the Xcode IDE • Already have experience of development with CocoaTouch
Data Storage	Google Firebase	• Cloud-based data storage owned and fully supported by Google • Benefits from security offered by Internet giant • Supports concurrent access from multiple devices • Manages connection volatility simply and easily • Free of cost for low volume - easily scalable if requirements grow • Already have experience of development with Firebase

Process

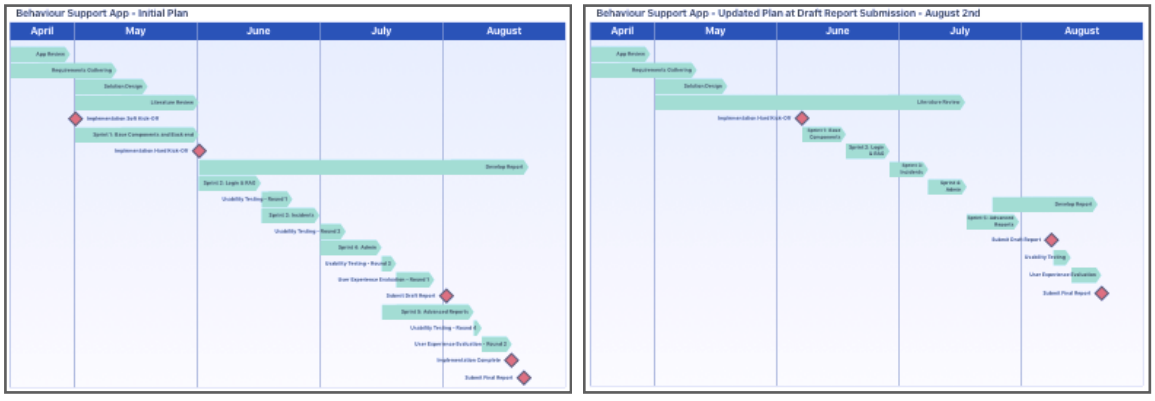
Requirements Development

A stable set of requirements was achieved through a combination of staff interviews, research and observation of the current system. Once the requirements were agreed, the Deputy Headteacher was consulted after each stage of the design process - and subsequently, the development process - to encourage a participatory approach to the design, ensure it was in line with expectations and to pick up on any additional requirements overlooked in the formal requirements-development exercise.

Planning

When planning timeframes for the project, the author aimed to balanced structure with flexibility. The initial plan aimed to provide a rigid structure for the pre-project preparation activities - including requirements gathering, research and product design - before transitioning to a more flexible series of ‘Sprints’. The aim was for the end of these sprints to mark completion of full sections/features of the application and offer a chance to review progress with the sponsor, before moving onto the next set of features. These sprints were left open to reorganising or rearranging, and development time for particular features within the sprints was not strictly allocated - meaning implementation could change up and down pace and direction to some extend - reacting to any unforeseen obstacles that arise. On August 2nd, the project’s plan was updated to reflect the actual timeframes for completed tasks, and rearranging of future tasks to ensure the project is still delivered on time.

Figure 19: *Original Time Plan Gantt Chart vs Revised Time Plan Gantt Chart at Interim Report*



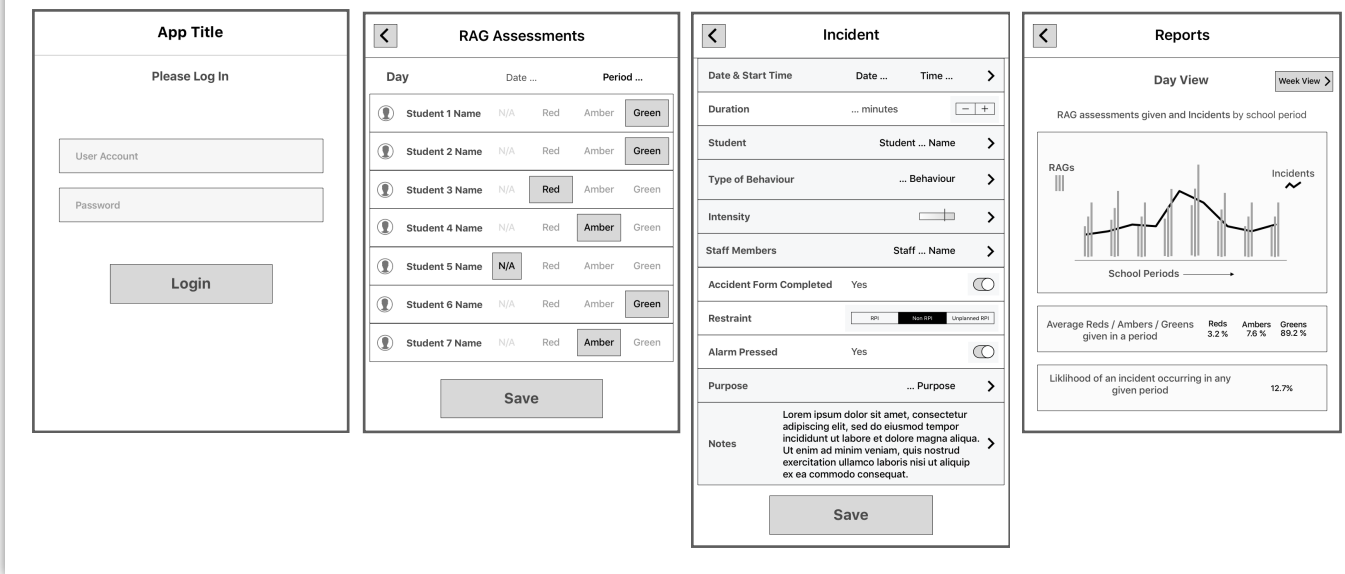
External circumstances (challenging exams) meant that during May it was actually very difficultly to find time to work on the project. This meant that implementation didn't start in earnest until mid June.

Full Gantt-chart time-plans can be found in Appendix 1 and Appendix 2

Prototyping

Before investing time and effort in expensive code implementation the app was designed thoroughly. By drafting concepts quickly and cheaply, it is still possible to get useful feedback and identify problems early, without much investment - avoiding wasted time implementing features and layouts that are ultimately unsuitable. For this project, the first stage of design was to create a wireframe, low-fidelity prototype and test it with the Sponsor. Many modern graphics-software packages now offer the ability to create and test prototypes on the target device. In this case, the author used the **Sketch** app for Mac to develop a detailed set of wireframe images to represent the interface. Sketch also allows the user to add hotspots to images which, when tapped, trigger transitions to other images. This allows the designer to take the potential end-user through a sequential set of low-fidelity designs with a realistic navigation flow at low cost.

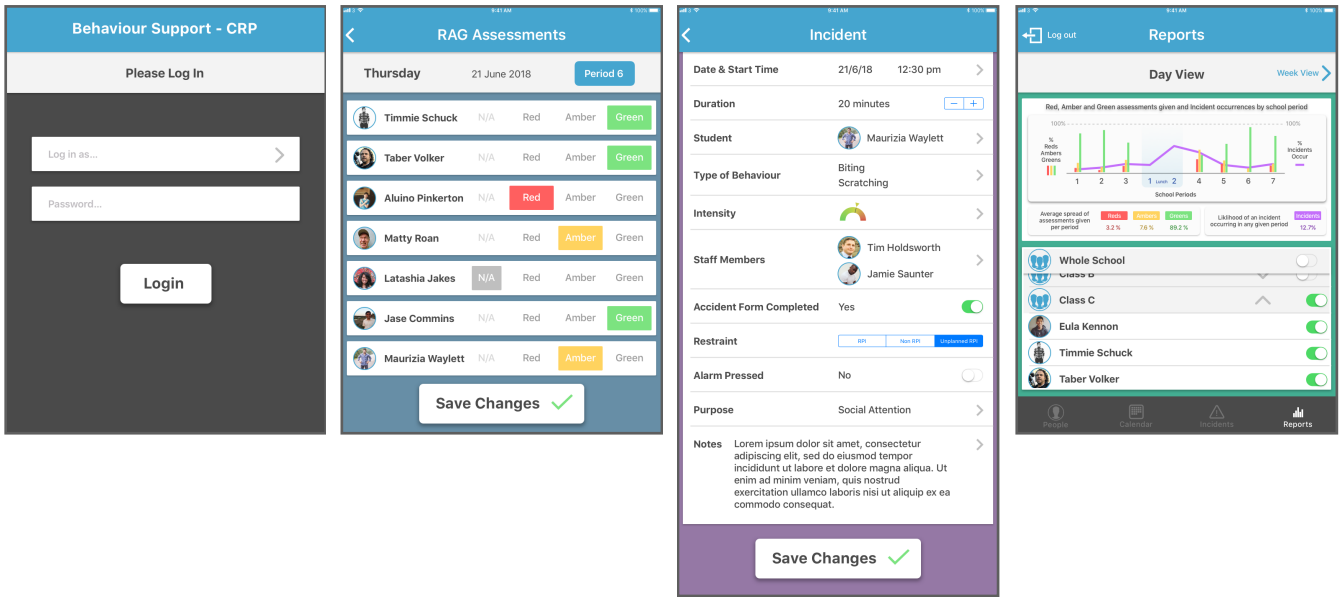
Figure 20: *Selection of UI layouts from Wireframe Prototype*



The full Wireframe Prototype can be found in the project's Supporting Material: 'Wireframe_Prototype.png'

The higher-fidelity prototype took a little more investment to produce than the wireframe - building on the wireframe with much more time spent on aesthetic qualities. However, it is still a more cost-effective way to test an *end product*-like version than forging ahead with implementation. This prototype was very similar to the end design and could be tested on-device in the same way as the wireframe, allowing changes to be spotted and made with relatively low consequence for the overall timeframe of the project.

Figure 21: *Selection of UI layouts from Full Concept Prototype*



The full Concept Prototype can be found in the project's Supporting Material: 'Full_Concept_Design.png'

Minimum Viable Product

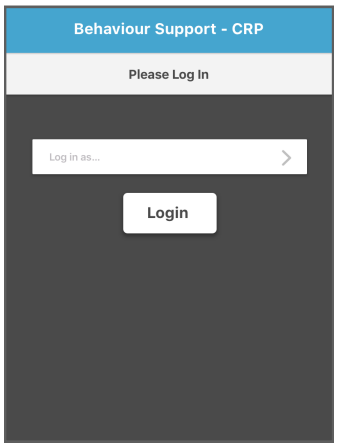
One of the principles of an agile methodology is to make iterative improvements to the product - release early and release often - therefore, it is necessary to consider which aspects of the full concept design could be removed and still provide viable, usable solution. This is known as the Minimum Viable Product (MVP) and will be the target for the first release - delivered through this project. Here, we will examine the changes to the full design and define the MVP.

The full interface for the MVP can be found in the project's Supporting Material: 'MVP_Design.png'

Login Screen

While password authentication will be a useful feature, it was removed from the MVP as it is not critical to the operation of the application - enforcement of permissions could be provided be workplace policy in the School for the time being.

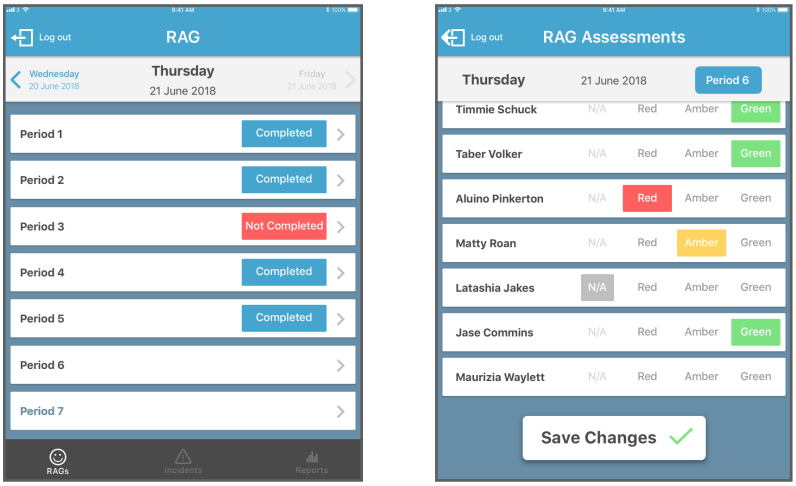
Figure 22: *Login screen for Minimum-Viable-Product*



RAG Assessments Screen

On the RAG Assessment screens, the metadata labels were removed from the school-period buttons as a result of feedback from the full-concept prototype testing. On the RAG Assessment table cells (as well as all other staff/student identification cells), profile images were removed - leaving just the student's name. Incorporation of profile images means programming them to display on screen, but also the means to capture, store and retrieve the images. They are not critical to operation so were left out of this release.

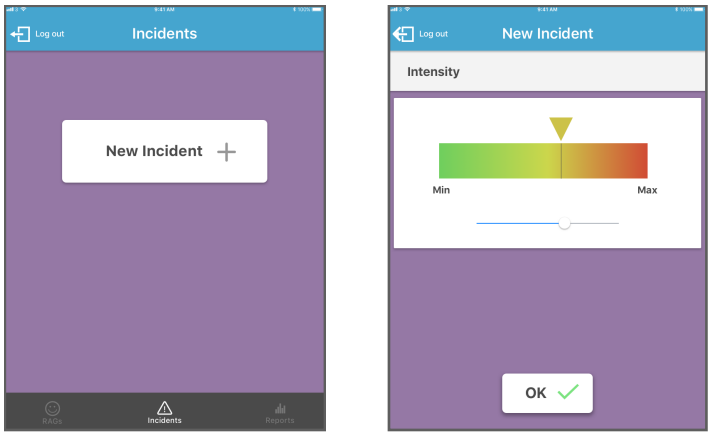
Figure 23: *RAG Assessments screens for Minimum-Viable-Product*



Incidents Screen

For the Incidents screens the option to view historic incidents was removed. They are not critical to operation of the system so the button 'Review Incidents' has been removed. Also, the 'intensity-indicator' chart for recording/displaying behaviour severity was reverted back to the original rectangular shape (instead of the semi-circular shape in the full concept) because it would be quicker to implement (an important factor in this time-pressured release) and still offer a similar user-experience.

Figure 24: *Altered Incident-related screens for Minimum-Viable-Product*



Class Report Screen

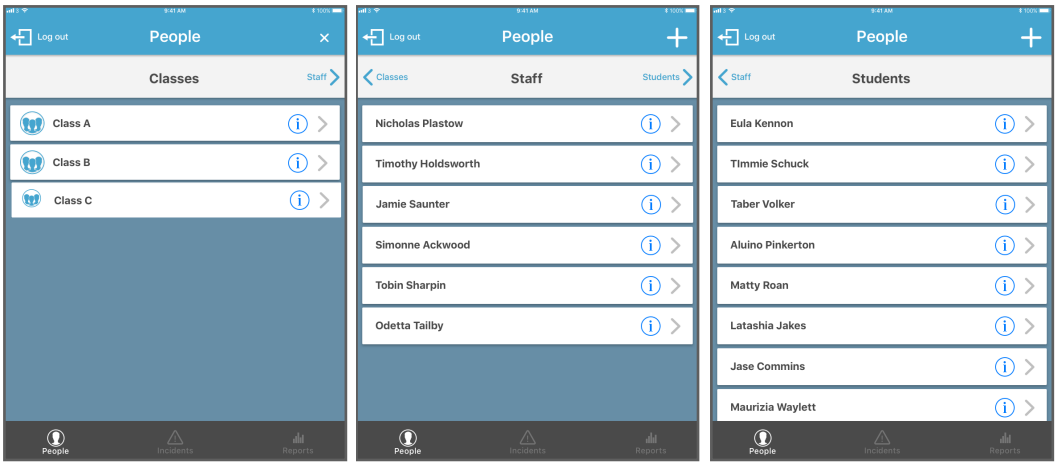
The class report screen remains visually the same as in the full concept (but instead incorporating the rectangular intensity-indicator chart design).

Figure 25: *Adjusted Class Reports screen for Minimum-Viable-Product*

People Screen

On the People screen, the view of school Classes has been adjusted to only show a list of all classes - without the collapsible list of associated staff and students underneath each one. While this is a positive user experience feature, it is not critical to operation at this stage and can be removed from the initial release.

Figure 26: *Adjusted People screen for Minimum-Viable-Product*



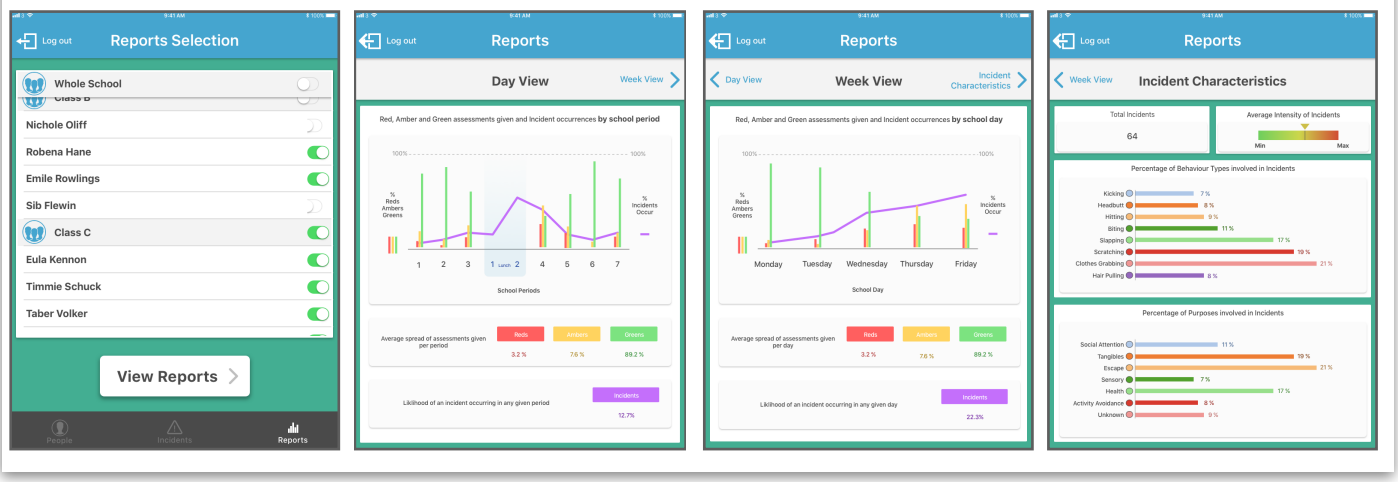
Calendar

The Calendar screen has been removed completely from this initial release because it would have a relatively complex and time consuming implementation and it does not provide critical functionality; the restriction of data entry on certain dates can be enforced by school policy for now.

Admin Reports Screen

For the Admin account's report screens, the layout has been adjusted to simplify the segmentation and slicing of data. The user is first presented with a full screen list of all classes and students and is required to select all those that they wish to see data for *before* generating the reports. When the user has finished selecting students and taps the 'View Reports' button, they are shown three full screen reports. The first two show the same 'Day View' and 'Week View' perspective of the selected students' data, while the third has been adapted (due to feedback from the full-concept prototyping testing exercise) to consolidate useful incident data onto a single screen - this time also including another horizontal bar chart displaying the percentage of incidents that each *Purpose* occurs in.

Figure 27: *Adjusted Admin Reports screens for Minimum-Viable-Product*



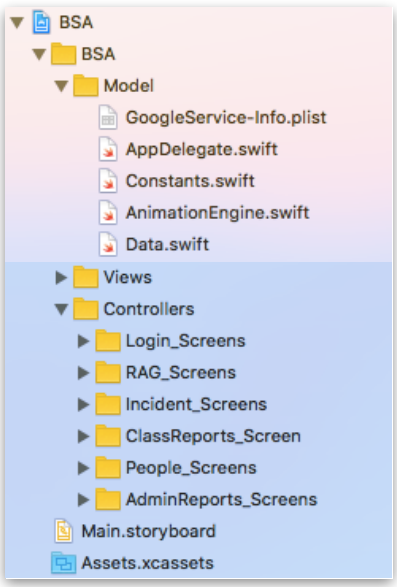
Implementation

The app's implementation was primarily conducted using Apple's *Xcode* software, the *Swift* programming language and Apple's *CocoaTouch* framework, which provides classes, methods and functionality specifically for iOS-targeted applications.

Project Structure

The project was structured using the Model-View-Controller (MVC) approach, with code for front-end/user-interface views abstracted away from the back-end/data model as much as possible. The objective of this model is to increase the modularity and scalability of the project. 'Controller' classes are used to mediate communication between the data 'Model' classes and the front-end 'View' classes, so that the model or the views can be developed independently of each other - without needing to re-engineer sprawling, monolithic classes every time the model or a view needs to be updated.

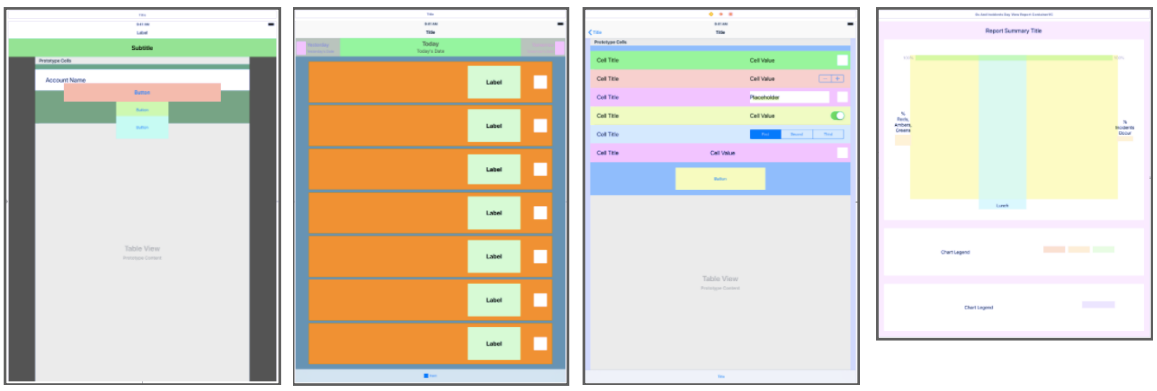
Figure 28: *Project folder structure*



Interface Builder

The Xcode IDE (Integrated Development Environment) has a range of powerful tools, including an *Interface Builder* for development of screen layouts in a visual way. It does not enable the developer to add much functionality, but allows them organise views through a Graphical User Interface (GUI) and link them to logic and methods written in the project's source code.

Figure 29: *Selection of Interface-Builder layout views*



External Libraries

Most of the source code for this project was written by the author. However, some aspects of development have been assisted by the inclusion of external libraries that provide specific methods and functionality.

Table 7: *External Code Libraries*

Library Name(s)	Type	Justification
Firebase/Core Firebase/Database	Firebase Connectivity	• Provided with Firebase service for easy configuration and connection to services
Charts 3.0.5	Chart Drawing - External Library	• Large community of users and online resources • Avoid time consuming task of drawing visual charts - where extraction of useful data/information is more important
Facebook pop 1.0	Animation	• Well supported and documented library for simple to complex visual animation

CocoaPods, which is a dependency manager for Swift and Objective-C *Cocoa* projects, was used to install these libraries through the Mac OS Terminal. With a 'Podfile' created in the project's root folder, which lists all the desired external libraries, a command of 'pod install' from the terminal downloads those dependencies (libraries) and installs them in the project. The libraries are then available to access once the 'import' keyword is used in source files to expose their interface.

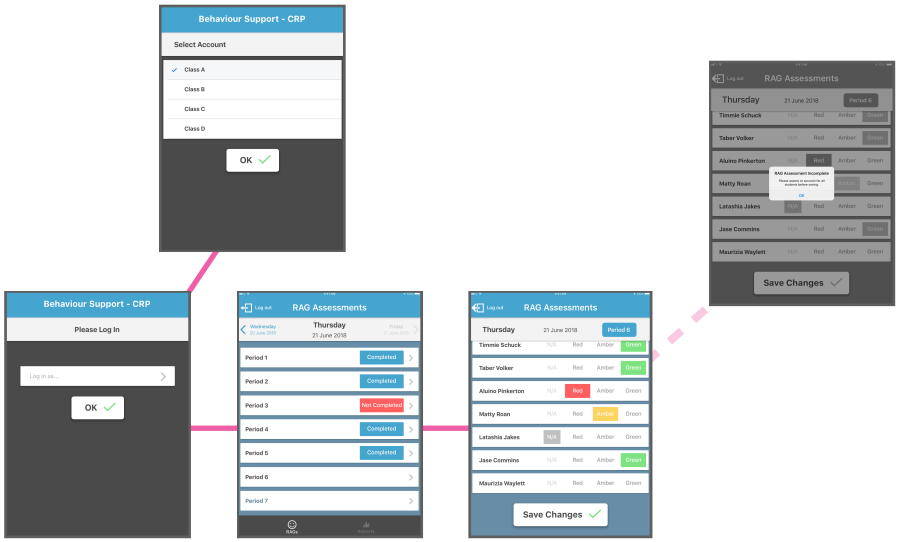
Figure 30: *CocoaPods Podfile contents*

```
target 'BSA' do

  use_frameworks!

  pod 'pop', '~> 1.0'
  pod 'Charts', '~> 3.1.1'
  pod 'Firebase/Core'
  pod 'Firebase/Database'

end
```

Sprint 1

The first sprint of implementation was used to develop the app's login screen and the RAG Assessments screens. To achieve this, several components were made in each area of the Model-View-Controller structure.

Figure 31: *User Interface for views developed in Sprint 1*



In the *Model*, classes were created for each **User Account**, **School Class** and each **RAG Assessment**. A **DBSeed** (database seed) class was also created to act as a temporary local data store to feed objects to the *Views* (via the *Controller* classes) during development.

To assist with error prevention, a static **Constants** class was created to store global values that help maintain consistency throughout the app (eg. colours and fonts) and reduce the number of places code needs to be update when changes are made (eg database keys).

Finally, an **AnimationEngine** class was created to provide animation controls for display objects. Rather than cluttering up the View-Controller classes, layout constraints for specific screen objects can be passed into a new instance of this animation engine, which can then be called upon to manipulate those constraints. This was mainly used to implement the left/right animation of content on and off screen - passing in the x-position constraints of various screen objects / object-containers.

Figure 32: *Example of AnimationEngine methods being used to animate screen layouts*

```
// Animates the sub-screen views to simulate 2 separate view containers sliding left and right
func setScreenAlignment(to screen: String) {
    switch screen {
        // animate view to the left - to show 'Login' container (username button and login button)
    case "login" :
        subtitle.text = "Please Log In"
        AnimationEngine.animate(constraint: subtitleXAlign, by: 200)
        loginAnimation?.animateOnFromScreenLeft()
        selectAccountAnimation?.animateOffScreenRight()

        // animate view to the right - to show 'Account' container (table of account names)
    case "selectAccount" :
        subtitle.text = "Select Account"
        AnimationEngine.animate(constraint: subtitleXAlign, by: -200)
        loginAnimation?.animateOffScreenLeft()
        selectAccountAnimation?.animateOnFromScreenRight()
    default :
        print ("unknown screen-alignment requested")
    }
}
```

Most of the reusable base-component *Views* were developed for the app in this sprint. These were created as subclasses of native CocoaTouch classes (*UIView*, *UIButton*, *UITableView*, *UITableViewCell* etc) with additional formatting provided to achieve the **Material Design** style identified during the requirements development stage. These base components were used throughout the app to give a consistent experience and minimise repetition of code.

Figure 33: *Example of Material Design formatting of objects*

```
import UIKit

class MaterialObject: UIView {

    // Configure view appearance when loaded
    override func awakeFromNib() {
        super.awakeFromNib()

            // set color, shadow and rounded corners
        self.layer.backgroundColor = UIColor.white.cgColor
        self.layer.shadowColor = UIColor.black.cgColor
        self.layer.shadowOffset = CGSize(width: 1, height: 2)
        self.layer.shadowRadius = 4
        self.layer.masksToBounds = false
        self.layer.shadowOpacity = 0.5
    }

}
```

Various custom *UIButton* and *UITableViewCell* subclasses were also developed in this sprint. For the button classes, custom drawing was used to provide the additional visual clues to the button's functionality. While this could have been achieved using imported external image files (which would have been simpler to code), performing simple drawing programmatically is a more efficient use of memory and can avoid problems with changeable screen resolutions.

The school-day period buttons and the RAG Assessments table cells, were a bit more complex and required some methods to update their appearance according to model data or user input.

Figure 35: *Example updating method for period-button status*

```
// Update label color and text value to reflect when a given period's RAG assessment has been completed
func setStatusComplete() {
    self.backgroundColor = Constants.BLUE
    self.text = "Complete"
}
```

Figure 36: *Example of setting status of RAG Assessment cell*

```
@IBAction func greenButtonPressed(_ sender: UIButton) {
    UIView.animate(withDuration: Constants.RAG_SELECTION_SPEED, animations: {
        self.selectionIndicator.layer.backgroundColor = Constants.GREEN.cgColor
        self.selectionIndicatorTrailingConstraint.constant = 15
        self.layoutIfNeeded()
    })
    sender.setTitleColor(.white, for: .normal)
    self.naButton.setTitleColor(Constants.GRAY, for: .normal)
    self.redButton.setTitleColor(Constants.GRAY, for: .normal)
    self.amberButton.setTitleColor(Constants.GRAY, for: .normal)
    selectionIndicatorPosition = "green"
    rAGSelectionDelegate.didSelectRAG(selection: "green", forCellWith: self.tag)
}
```

The RAG Assessment cells also required use of the **delegation pattern**, which is similar to *interfaces* in other languages and can be used to pass data between classes through invocation of pre-defined, required methods. In this case, a protocol was defined named ‘RAGSelectionDelegate’, which included a function named ‘didSelectRAG’ that takes a String and and Integer as arguments. When the cell is loaded, its parent view-controller (VC) sets itself as the cell’s delegate, allowing the cell to call the required ‘didSelectRAG’ function in the parent VC’s class - passing in the user’s selection (Red, Amber, Green or NA) as the String and the cell’s row number (which corresponds to a student in one of the parent VC’s arrays).

Figure 37: *Example definition of a protocol for the delegate pattern*

```
protocol RAGSelectionDelegate {
    func didSelectRAG(selection: String, forCellWith: Int)
}
```

Three view-controllers were developed in this sprint: one for the login screen, one for the selection of school-day periods and one for completion of RAG Assessments for a selected school-day period. The main purposes of the login VC are:

1. To handle the selection of account that the user would like to log in to by animating left and right (via the animation engine) between the main login screen and an account-selection table,
2. Caching the account's data to local storage (for quick, global access) - using **UserDefaults**, which provides a small amount of easily accessible local data storage - without the need to set up a new database instance.
3. Segueing to the appropriate *UITabBar* for the logged in user (Admin account area or Class/ Staff account area).

Figure 38: *Caching of selected-user data and and segue to appropriate account area*

```
// Segue to appropriate tab-bar controller - according to the account selected
@IBAction func loginButtonTapped(_ sender: Any) {
    guard selectedAccount != nil else { return }

        // record account details
    UserDefaults.standard.set(selectedAccount?.id, forKey: Constants.LOGGED_IN_ACCOUNT_ID)
    UserDefaults.standard.set(selectedAccount?.accountName, forKey: Constants.LOGGED_IN_ACCOUNT_NAME)
    UserDefaults.standard.set(selectedAccount?.securityLevel, forKey: Constants.LOGGED_IN_ACCOUNT_SECURITY_LEVEL)
    UserDefaults.standard.set(selectedAccount?.schoolClassId, forKey: Constants.LOGGED_IN_ACCOUNT_CLASS_ID)

    switch selectedAccount!.accountName {
    case "Admin" :
        // go to Admin Tab Bar Controller)
        performSegue(withIdentifier: "showAdminAccountTabBarVC", sender: nil)
    default :
        // go to Class Tab Bar Controller
        performSegue(withIdentifier: "showClassAccountTabBarVC", sender: nil)
    }

}
```

The purpose of the view controller for school-day period selection is to allow the user to 'scroll' back and forth through calendar days and present 7 buttons (one for each period) for that day. Only two sets of 7 buttons are actually used, but by using the animation engine to reposition sets and reload button data before each scroll, an illusion of scrolling through multiple sets is created. Once a user taps a button, the VC triggers a segue to the view controller for competing RAG Assessments and passes in to it the selected date and school period.

The view controller for completing RAG Assessments handles selection of RAG statuses for each student in a class. By first checking the cached User-Account data the RAG Assessments VC fetches a list of Student objects (currently from the DBSeed class) that are associated with the logged in user. Once the assessments are complete and the user taps the 'Save Changes' button, a new **RAGAssessment** object is created (to be uploaded to the database once the back-end is

implemented). If the user has not assessed all students, the VC presents an alert to prompt them

Figure 39: *Alert presented if RAG Assessment is incomplete*

```
// check to make sure all students have been assessed
guard !rAGSelections.contains("none") else {

        // if not, show alert to inform all students must be assessed
    let alert = UIAlertController(title: "Assessment Incomplete", message: "Please assess
        all students before saving changes", preferredStyle: UIAlertControllerStyle.alert)
    alert.addAction(UIAlertAction(title: "OK", style: .default, handler: { action in
        alert.dismiss(animated: true, completion: nil)
    }))
    self.present(alert, animated: true, completion: nil)
    return
}
```

to do so before requesting to save changes.

Sprint 2

During the second sprint, the screens needed for recording incidents were created, as well as screen for displaying school-class reports.

Figure 40: *User Interface for views developed in Sprint 2*

In the *Model,* new classes were created to represent each **Student**, each **Staff** member and each **Incident**. Students and Staff need to be retrieved from storage and selected by the user for association with an Incident. The incidents themselves are created once the user has completed all fields in the incident-form and requests to save the values to storage. A class was also created

to hold data for class reports (**Class Report Data Set)** in a way that is clean and easily accessible for the class-report view. This means that the report view can simply request a data set to be returned from the model, without being cluttered by performing analysis of data locally (inside the view-controller's class)

Various different cell and button *Views* were created, using the custom drawing approach described earlier, as well as native UI elements such as *UISwitch*es, *UISegmentedControl*s and *UIStepper*s.

For the Red, Amber and Green columns of the class report screen, a **Report Item** class was created. This is a reusable class with several features. Upon creating an instance of the class, it can be set to a 'type' (using a Swift **enum** to define a list of all possible types).

Figure 41: *Enum used to define all possible types of 'Report Item'*

```
// Provide constrained values for different available report item data types
enum ReportItemType {
    case reds
    case ambers
    case greens
    case incidents
}
```

When the type is set, the class modifies its appearance to suit the prescribed type and make the data value more quickly recognisable. The core item only displays a colour-coded title rectangle

Figure 42: *Setting of Report Item type and modifying appearance*

```
// Assigns a report data-type to the item and adjusts appearance accordingly
func setType(to type: ReportItemType) {
    self.type = type
    switch type {
    case .reds:
        self.title.text = "Reds"
        self.layer.backgroundColor = Constants.RED.cgColor
        figure.textColor = Constants.RED_DARK
    case .ambers:
        self.title.text = "Ambers"
        self.layer.backgroundColor = Constants.AMBER.cgColor
        figure.textColor = Constants.AMBER_DARK
    case .greens:
        self.title.text = "Greens"
        self.layer.backgroundColor = Constants.GREEN.cgColor
        figure.textColor = Constants.GREEN_DARK
    case .incidents:
        self.title.text = "Incidents"
        self.layer.backgroundColor = Constants.PURPLE.cgColor
        figure.textColor = Constants.PURPLE_DARK
    }
}
```

and a programatically-generated value label beneath it. However, the class also has methods that generate, and animate, a column above the item - representative of the data value that the item has been set to. In the case of the class report, three report item instances were created, with their types set to ‘reds’, ‘ambers’ and ‘greens, their values set according to the retrieved ClassReportDataSet and their columns activated - and animated every time the screen is shown.

To create the **Intensity Indicator** view, a linear gradient was applied and a programmatically generated ‘arrow’ view was added and positioned according to the value of a given ‘intensity’ variable. An animation method was also provided to give a more compelling view of the data.

Figure 43: *Adding a gradient layer to the Intensity-Indicator*

```
// Configure view appearance on load
override func awakeFromNib() {

        // add color gradient to the chart
    let color1 = UIColor(red: 111/255, green: 207/255, blue: 94/255, alpha: 1.0).cgColor
    let color2 = UIColor(red: 204/255, green: 215/255, blue: 76/255, alpha: 1.0).cgColor
    let color3 = UIColor(red: 209/255, green: 76/255, blue: 53/255, alpha: 1.0).cgColor
    gradient.colors = [color1, color2, color3]
    gradient.startPoint = CGPoint(x: 0.0, y: 0.5)
    gradient.endPoint = CGPoint(x: 1.0, y: 0.5)
    self.layer.addSublayer(gradient)

        // create and add an Indicator Arrow view above the chart as a visual representation of the its 'intensity' value
    indicator = IndicatorArrow(frame: CGRect(x: 0 - (self.frame.height * 0.5 / 2) + (self.frame.width * CGFloat(intensity)),
        y: 0 - self.frame.height * 0.5, width: self.frame.height * 0.5, height: self.frame.height * 1.5))
    self.addSubview(indicator)
}
```

Figure 44: *Example of chart column animation*

```
// Animates a displayed column from a height of zero to its representative value
func animateColumn() {
    guard column != nil else { return }

    column!.frame =  CGRect(x: self.bounds.midX - 20, y: -3 , width: 40, height: 0)

    UIView.animate(withDuration: Constants.REPORT_ANIMATION_SPEED , delay:
        Constants.REPORT_ANIMATION_DELAY, options: UIViewAnimationOptions.curveEaseOut,
        animations: {
        self.column!.frame =  CGRect(x: self.bounds.midX - 20, y: -3 - CGFloat(self.value!),
            width: 40, height: CGFloat(self.value!))
    },completion: nil )
}
```

Several view controllers were created in this sprint. Primarily, the main Incidents screen was very simple - with only a single button for navigation to the incident-form screen. The incident-form screen contains a fairly complex UITableView, which includes various custom UITableView cells that are programmed to receive data from value-selection view controllers through the delegation pattern.

Figure 45: *Protocol definition for Date/Time-selection delegate*

```
// Allows a selected Date value to be sent to the delegate
protocol DateTimeSelectionDelegate {
    func setDateTime(to selection: Date)
}
```

Figure 46: *Implementation of protocol method for Date/Time-selection delegate*

```
// Sets the new Incident Form's 'Date/Time' property to the value selected in delegated VC - and updates the 'Date/
    Time' cell's value label
func setDateTime(to selection: Date) {
    if let dateTimeCell = tableView.cellForRow(at: IndexPath(row: 1, section: 0)) as? IncidentFormSelectionCell {
        dateTimeCell.setValue(to: "\(DataService.getShortDateString(for: selection))        \
            (DataService.getTimeString(for: selection))")
        self.incidentDateTime = selection
    }
}
```

A value-selection view controller was created for all cells that can not be updated from the actual cell itself - via a UIStepper, UISwitch or UISegmentedControl. These value-selection view-controllers are fairly simple - containing a UIDatePicker, UITableView, UITextArea or the custom IntensityIndicator view - controlled by a UISlider. Once selection is made the value is passed back to the incident form VC via the delegate pattern.

Figure 47: *Invocation of delegate protocol method to pass data between View-Controllers*

```
// Sets parent VC's 'Date/Time' value to thic VC's selected value, before segueing back to parent VC
@IBAction func okButtonPressed(_ sender: Any) {
    dateTimeSelectionDelegate?.setDateTime(to: selectedDateTime)
    self.navigationController?.popViewController(animated: true)
}
```

If no selection is made, or if an incident form is submitted, a UIAlert is presented to the user to prompt them to complete the task in question - as described above.

Finally, the view-controller class for Class Reports was programmed to receive ClassReportDataSet objects and pass the contained data through to the appropriate *ReportItem* and *IntensityIndicator* views.

Figure 48: *Passing of data from Class-Report dataset to individual chart elements*

```
// Configures the 2 charts at the top of the view
func setupTopCharts() {

        // set main title
    topChartsTitleLabel.text = "\(topChartsData!.entityName!) - \(topChartsData!.timePeriod!)"

        // set chart 1 Reds data
    topChart1Legend.text = "Average Spread of\nRed, Amber and Green Assessments Given"
    topChart1Reds.setType(to: .reds)
    topChart1Reds.setValue(to: topChartsData!.averageReds)
    topChart1Reds.showColumn()

        // set chart 1 Ambers data
    topChart1Ambers.setType(to: .ambers)
    topChart1Ambers.setValue(to: topChartsData!.averageAmbers)
    topChart1Ambers.showColumn()

        // set chart 1 Greens data
    topChart1Greens.setType(to: .greens)
    topChart1Greens.setValue(to: topChartsData!.averageGreens)
    topChart1Greens.showColumn()

        // set chart 2 legends and data
    topChart2Legend1.text = "Average Intensity of Incidents"
    topChart2IntensityChart.animateIntensity(to: topChartsData!.averageIntensity)
    topChart2Legend2.text = "Likelihood of an Incident Occurring in Any Period"
    topChart2IncidentLikelihoodItem.setType(to: .incidents)
    topChart2IncidentLikelihoodItem.setValue(to: topChartsData!.likelihoodOfIncident)

}
```

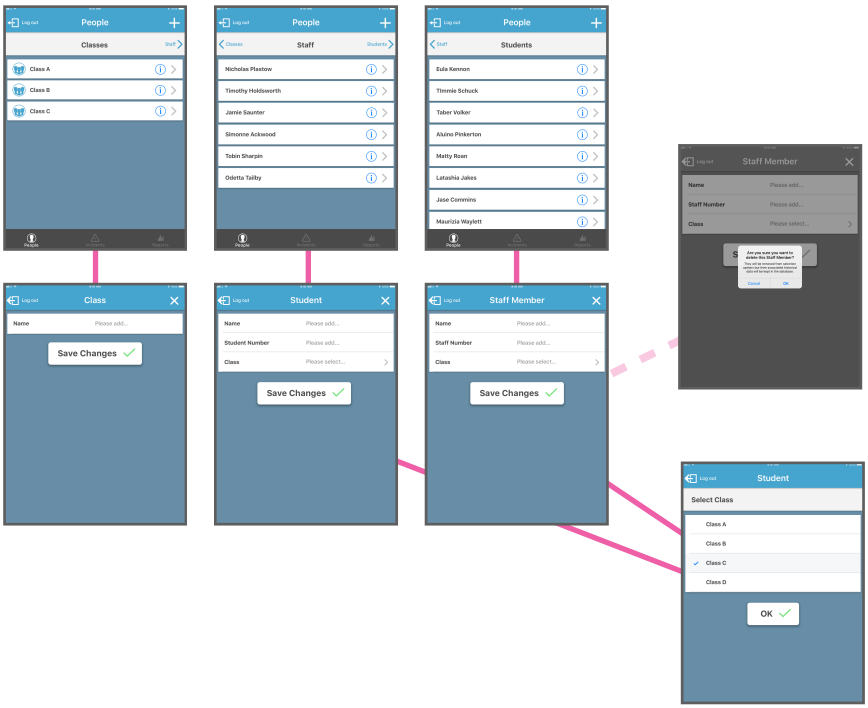
When the user returns to the screen after navigating away from it, the view-controller is programmed to re-animate the charts.

Figure 49: *Animation of charts every time time the view is presented*

```
// Immediately before the view appears, sets the chart data visual representations to animate
override func viewDidAppear(_ animated: Bool) {

        // top chart 1
    topChart1Reds.animateColumn()
    topChart1Ambers.animateColumn()
    topChart1Greens.animateColumn()

        // top chart 2
    if topChartsData != nil {
        topChart2IntensityChart.animateIntensity(to: topChartsData!.averageIntensity)
    }
```

Sprint 3

Figure 50: *User Interface for views developed in Sprint 3*



The third sprint was used to create the screens required for the Admin user to manage system entities - classes, staff and students. No new implementation was needed in the project's *Model* for this sprint.

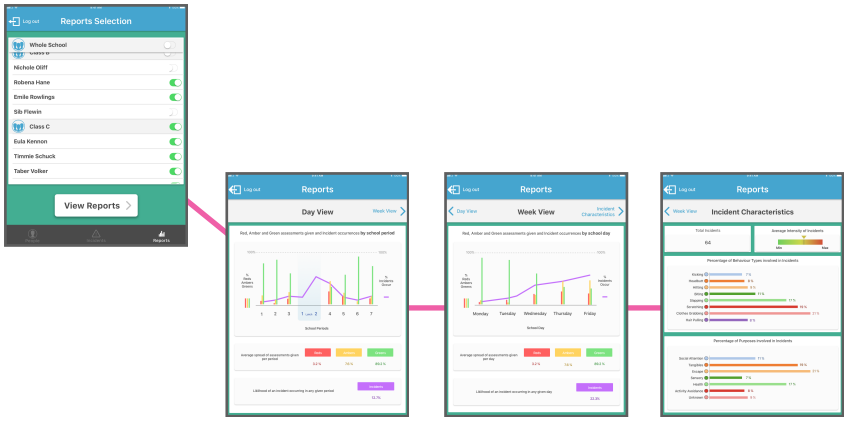
A new UITabBar was created to provide navigation around the Admin account's area of the app - replacing the 'RAG Assessments' tab with a 'People' tab, and changing the destination of the 'Reports' tab to display the *Admin*-reports view-controller .Some custom UITableViewCell *Views* were created for the different entity tables - one for Classes and another for both Staff and Students.

One view-*Controller* was created to display lists of all existing entities - navigation between the the list containers achieved in the same way as for the school-day period selection screen in sprint 1. Another was created for each of the entities - to show details of that entity (name, identification-number etc). These 'details' view-controllers are also used to capture data for a new entity in the system. A final view-controller was created to enable scalable selection of existing classes - to associate with a Student entity - using the delegation pattern.

Sprint 4

The fourth sprint was used to develop the Admin accounts Reports section. In the *Model* a new class was created to hold data for the admin reports (**Admin Report Data Set**) with the same principle as for the Class Report Data Set.

Figure 51: *User Interface for views developed in Sprint 4*



Several new *Views* were implemented, including custom UITableViewCells and a custom UIButton for the student selection screen, and a selection of chart views that makes of the 'Charts' external library - installed with CocoaPods as described earlier.

The combined clustered-column and line charts needed to be configured to take the correct data values from the AdminReportDataSet object used to populate it. They also required some custom legend labels, axis-label formatting and some custom views to highlight the 'lunch' periods (on the day-view chart) and indicate the maximum possible value (100% marker at the top of the charts). The horizontal bar charts were configured to take data from an AdminReportDataSet, have custom value labels (suffixed with a '%' symbol) and colour coordinated with the custom legend views. Careful configuration was needed for the screen layout - using nested *UIStackViews* to achieve tidy layout - and allowing the layout to be changed when the device's screen orientation changes.

Figure 52: *Configuration of Day-View chart to read values from given Dataset*

```
    // congfigure layout of bars
let groupSpace = 0.7
let barSpace = 0.0
let barWidth = 0.1

    // add values for 'red' bars
let bar1Entry1 = BarChartDataEntry(x: 0, yValues: [redBarValues[0]])
let bar1Entry2 = BarChartDataEntry(x: 0, yValues: [redBarValues[1]])
let bar1Entry3 = BarChartDataEntry(x: 0, yValues: [redBarValues[2]])
let bar1Entry4 = BarChartDataEntry(x: 0, yValues: [redBarValues[3]])
let bar1Entry5 = BarChartDataEntry(x: 0, yValues: [redBarValues[4]])
let bar1Entry6 = BarChartDataEntry(x: 0, yValues: [redBarValues[5]])
let bar1Entry7 = BarChartDataEntry(x: 0, yValues: [redBarValues[6]])
let bar1Entry8 = BarChartDataEntry(x: 0, yValues: [redBarValues[7]])
let bar1Entry9 = BarChartDataEntry(x: 0, yValues: [redBarValues[8]])
let barDataSet1 = BarChartDataSet(values: [bar1Entry1, bar1Entry2, bar1Entry3, bar1Entry4, bar1Entry5, bar1Entry6, bar1Entry7,
    bar1Entry8, bar1Entry9], label: nil)
barDataSet1.setColor(UIColor(red: 255/255, green: 96/255, blue: 96/255, alpha: 1.0))
```

Figure 53: *Method set to unpack given Dataset values for use in Day-View and Week-View charts*

```
// Unpacks data sets for use in the charts - first checking to make sure that a valid data set has been given
func addChartData(chartData: (redValues: [Double], amberValues: [Double], greenValues: [Double], incidentValues: [Double])) {

        // check to make sure data set is valid
    guard chartData.redValues.count == 9, chartData.amberValues.count == 9, chartData.greenValues.count == 9,
        chartData.incidentValues.count == 9 else {
            // invalid data given
        print("invalid data for day view report")
        return
    }

        // unpack data
    redBarValues = chartData.redValues
    amberBarValues = chartData.amberValues
    greenBarValues = chartData.greenValues
    incidentLineValues = chartData.incidentValues
    redAverageValue = DataService.getAverage(of: redBarValues)
    amberAverageValue = DataService.getAverage(of: amberBarValues)
    greenAverageValue = DataService.getAverage(of: greenBarValues)
    incidentAverageValue = DataService.getAverage(of: incidentLineValues)

    // update charts
    updateCharts()
}
```

Only two view-controllers were created in this sprint: one to manage selection of students to show report data for, and another to display the reports themselves. The first view-controller actually contains a simple single-cell table with the ‘whole-school’ selection cell in it, and a ’container view’ with a second table in it that shows all the classes and students fetched from storage. These are positioned to give the impression of a single table, but allows the ‘whole-school’ selection cell to be pinned to the top of the view. Selected students (identified by the state of their provided UISwitch) are passed from the container-view to the main view controller using the delegate patten when the user taps the ‘View Reports’ button. Those selected Students are then passed to the report-displaying view-controller when performing the segue.

The reports-displaying view-controller provides navigation between the three report containers themselves - using the same ‘flat-navigation’ approach described earlier. It is also responsible for retrieving the AdminReportDataSet for the received selection of students and passing the appropriate data to the relevant charts for display.

Figure 54: *Method to unpack given values for use in the Incident Characteristics charts*

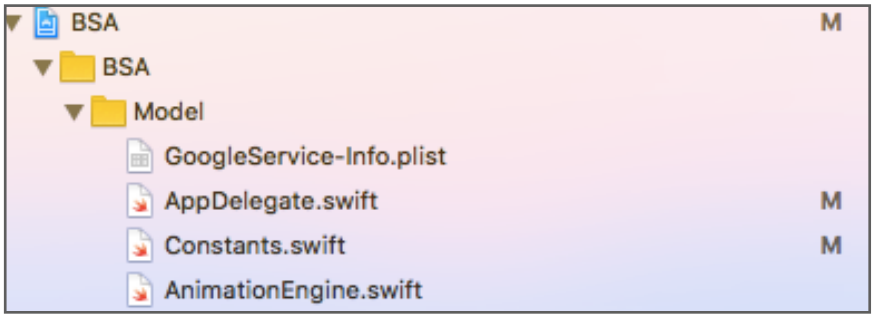
```
// Unpacks data sets for use in the charts - first checking to make sure that a valid data set has been given
func addChatData(chartData: (totalIncidents: Int, averageIntensity: Float, behaviourPercentages: [Double], purposePercentages: [Double]))
    {

        // check to make sure data set is valid
    guard chartData.behaviourPercentages.count == 8, chartData.purposePercentages.count == 7 else {
            // invalid data given
        print("invalid data for Incident Characteristics report")
        return
    }

        // unpack total incidents data
    totalIncidentsValue = chartData.totalIncidents

        // unpack average intensity data
    averageIntensityValue = chartData.averageIntensity

        // unpack behaviours percentages data
    kickingValue = chartData.behaviourPercentages[0]
    headbuttValue = chartData.behaviourPercentages[1]
    hittingValue = chartData.behaviourPercentages[2]
    bitingValue = chartData.behaviourPercentages[3]
    slappingValue = chartData.behaviourPercentages[4]
    scratchingValue = chartData.behaviourPercentages[5]
    clothesGrabbingValue = chartData.behaviourPercentages[6]
    hairPullingValue = chartData.behaviourPercentages[7]

        // unpack purposes percentages data
    socialAttentionValue = chartData.purposePercentages[0]
    tangiblesValue = chartData.purposePercentages[1]
    escapeValue = chartData.purposePercentages[2]
    sensoryValue = chartData.purposePercentages[3]
    healthValue = chartData.purposePercentages[4]
    activityAvoidanceValue = chartData.purposePercentages[5]
    unknownValue = chartData.purposePercentages[6]

        // update charts
    updateCharts()
}
```

Sprint 5

In the final sprint, no new views or view-controllers were created, all the time was spent developing the *Model* - configuring the app's back-end (setting up the Firebase application and configuring the Cloud database) and creating classes to analyse stored data and populate the app's charts.

With a project created for this app in Firebase's browser-portal, a 'GoogleService-Info.plist' file, which contains all necessary configuration data, is downloaded and added to the app's root folder.

Figure 55: *GoogleService-Info.plist file in product folder*



To connect the app to the project created in the Firebase console, a simple 'FirebaseApp.configure()' command is added to the *AppDelegate* class's *didFinishLaunchingWithOptions* method.

Figure 56: *Firebase configuration command*

```
FirebaseApp.configure()
```

Figure 57: *Example of 'constant' values used for consistent database referencing*

```
/// Firebase Database values:
static let FIREBASE_USER_ACCOUNTS = "user_accounts"
static let FIREBASE_USER_ACCOUNTS_NAME = "account_name"
static let FIREBASE_USER_ACCOUNTS_SECURITY_LEVEL = "account_security_level"
static let FIREBASE_USER_ACCOUNTS_CLASS_ID = "account_class_Id"
```

Once appropriate database key values were added to the *Constants* class (to prevent errors when identifying database nodes in different firebase-access methods), the database was ready to use. To provide access to the database, a dedicated **DataService** class was created to handle all communication with Firebase - as well as some other data-related and data-type extensions methods. This DataService class has a varied interface with methods supporting retrieval of individual data objects, whole collections and filtered collections of objects from Firebase. It also has several methods for handling Dates - returning different String-formatted values for given dates, or Date values for historic moments in time - and some basic analysis methods (for finding averages of a range of numeric values).

Figure 58: *Example view of the project's Firebase database with data present*

Finally, two classes were implemented for analysis of the class reports (**ClassReportAnalysis**) and admin reports (**AdminReportAnalysis**). These classes both use the DataService class to retrieve data for a set of students (and, in the case of the class reports, for a given time period), then perform filtering, sorting and aggregation of data in order to return useful data sets for populating charts. The algorithms for performing this analysis and generating these data sets are too long to include in-line so can be found in the appendix.

Figure 59: *Example delegate method for handling data received from the DataService class*

```
func finishedAnalysingAdminReportData(dataSet: AdminReportDataSet) {
    data = dataSet
    updateChartData()
}
```

Testing

Once the development of a viable solution was complete, the product was tested with end users for **Usability** and evaluation of **User-Experience**. General bugs and errors were picked up as implementation went along - with each newly implemented feature tested for functionality at the time, but this formal testing phase is an opportunity to test the usability and user-experience of the product with other people that are representative of the end users. The results of these tests are used to inform future rounds of implementation - to refine the product further and improve both of those aspects. This cycle of 'implement -> test usability -> evaluate user-experience -> implement' will be repeated as many times as is necessary for each iteration of the product.

Usability Testing

The initial round of usability tests was conducted with three people and required them to complete various use-cases in the app un-guided. The aim being to watch the participants as they complete the tasks, taking note of any observed issues or difficulties with the interface and asking for feedback at the end of each session.

A full list of the use-cases tested can be found in Appendix 3

The main issues observed and fed back from testers are as follows:

1. Navigation. Several times the testers were observed trying to swipe the screen to navigate (eg. swipe-right to navigate back to previous screen) as they are accustomed to doing in other apps they use on daily basis. While navigation was already provided through buttons, adding this swipe-navigation feature would probably offer an enhanced and more natural experience.
2. Button labelling. In the navigation bar, the ‘Log-out’ button is marked with a graphical icon and the text-label ‘log-out’. However, the entity-details screens have buttons for adding a new class/staff-member/student (marked with a ‘plus’ symbol) and a delete button (marked with a ‘cross’ symbol) - but with no text-label. This ambiguity led to some confusion when testers were asked to ‘add a new Class entity’ and ‘delete a Student entity’. Text-labels were purposefully left out of the initial design here to maintain, where possible, a minimal approach to screen design and reduce clutter. However, this usability test highlighted that the symbols alone were not enough to convey the buttons’ actions clearly enough.
3. Incident behaviours/staff/purposes selection. In each incident form, the user must select a single student that was involved, before going on to select one *or more* behaviours, staff members or purposes that were also involved. It was observed that testers only ever selected a single behaviour/staff-member/purpose - and when asked after the session, they fed back that they had not realised they could select more than one. Already, the interface design uses different checkmarks/selection indicators to differentiate single vs multi-selection in the different tables, and presents an alert to the user, which says they must select one or more, if they try to press ‘OK’ without making any selection. However, this testing phase revealed that these measures were not clear or pre-emptive enough to fully inform the user of functionality before using that particular interface.
4. Meaning of Admin Report charts. During feedback, two of the testers asked for clarification of the ‘Day View’ and ‘Week View’ charts - unsure whether they were showing data for the current day/week or some sort of average. Once it was explained to them that they show aggregate data that represents the ‘average day’ and the ‘average week’ from all data collected, they both suggested that the title could be changed to more clearly state that it was showing generalised/ aggregate data.

5. Colour scheme for ‘Behaviour-Type’ and ‘Purpose-Type’ charts in the ‘Behaviour Characteristics’ report page. After the full-prototyping exercise revealed the additional requirement for a chart in the Admin reports to show purpose-types - similar to the behaviour-types chart - another horizontal-bar-chart was added to the interface. It was feedback that by using the same colour-scheme (which used a random selection of colours to differentiate the elements of the chart - whilst avoiding the red/amber/green/purple colours of the other report elements) some users found the charts confusing - unsure whether they represented the same incident characteristics, or were grouped/related in some way.

Implementation (2)

In response to this usability testing, several adjustments were made to the interface:

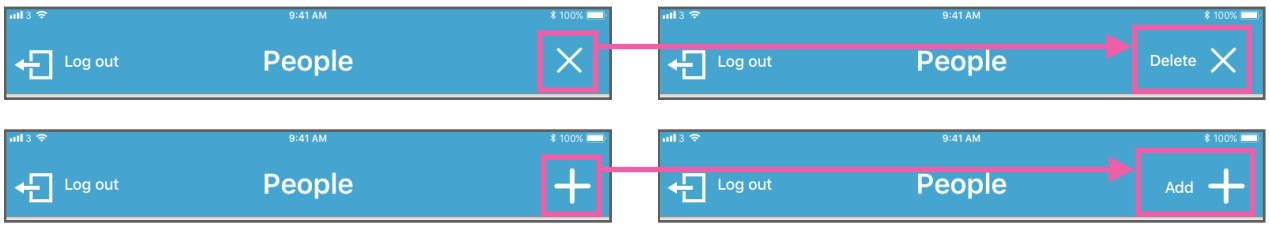
1. Navigation: Where appropriate, swipe-driven navigation was added to screens using UISwipeRecogniser’s to trigger the same navigation functions as the already provided navigation buttons.

Figure 60: *Example of adding a swipe-gesture recogniser to the view*

```
    // add left-swipe recogniser
let swipeLeft = UISwipeGestureRecognizer(target: self, action: #selector(processGesture))
swipeLeft.direction = UISwipeGestureRecognizerDirection.left
self.view.addGestureRecognizer(swipeLeft)
```

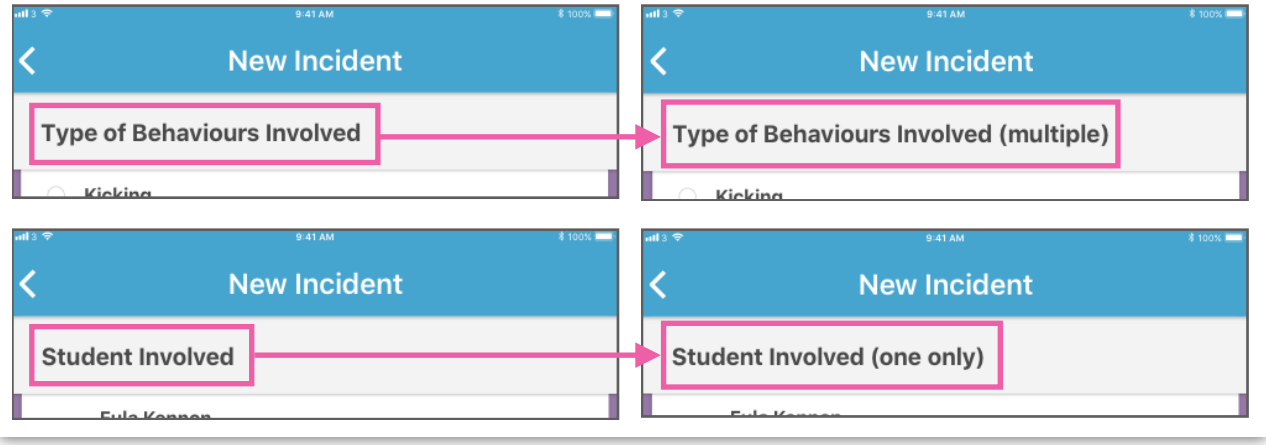
2. Button-labelling: On the ‘Admin’ user’s ‘People’ screens, the ‘Add New Entity’ and ‘Delete Entity’ buttons were given additional labels to improve conveyance of their functionality.

Figure 60: *Adding text-labels to the ‘Delete’ and ‘Add’ buttons in the People screen*



3. Incident Behaviour/Staff-Member/Purpose selection: In the subtile-bar for each of these selection-screens presented by the Incident-form, additional text was added to explicitly state whether the table allowed single or multiple selection.

Figure 61: *Adding text to specify number of selectable items in Incident-Form selection lists*

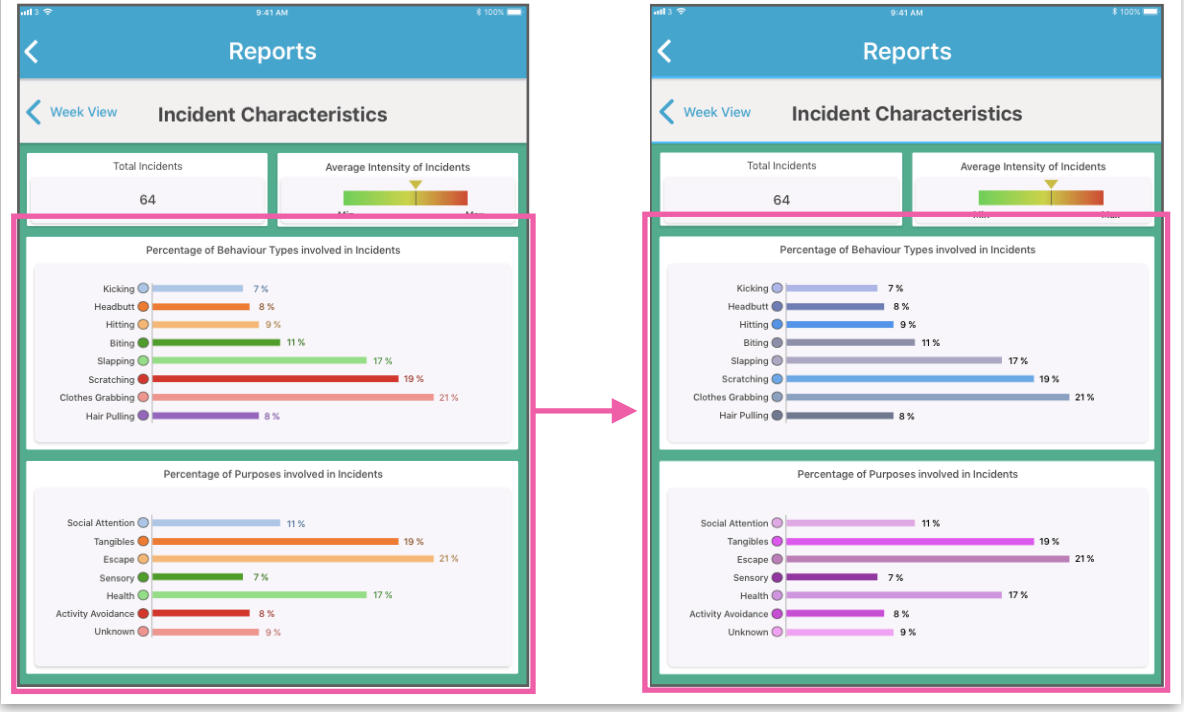


4. Meaning of Admin Report charts. The subtitle text for each of the 'Day View' and 'Week View' report pages was altered to read "General Day View" and General Week View", respectively.

Figure 62: *Adding text to help clarify meaning of Admin Report charts*

5. Colour scheme for 'Behaviour-Type' and 'Purpose-Type' charts in the 'Behaviour Characteristics' report page: The author felt it was still important to use different colours for each item in each of the charts (to clearly differentiate and make identification of each item easier), and also to avoid using the red/amber/green/purple of other elements of the report. Therefore, the 'Behaviour-Type' chart was given different hues of blue as its colour scheme, and the 'Purpose-Type' chart was given different hues of pink for its colour-scheme.

Figure 63: *New colour schemes to help distinguish horizontal-bar-charts*



User-Experience Evaluation

To test the app's user-experience 10 people were asked to use it each day for a week before providing feedback on their experience. Ideally, this round of testing would have taken place in the classroom setting, preferably at St Ann's School. However, the timing for this project meant that this round of testing needed to take place in the middle of the UK school summer holidays. Therefore, a user-group with similar characteristics was selected to represent the end-users - people, some with low ICT-skills, who have children in their care, or who have experience teaching classes of students and tracking their progress. Another challenge faced was that most testers did not have access to iPad devices, and those that did have them, didn't have access to them all day - as the end-users would. So, to overcome this device-accessibility issue, and to make the content more relevant to the testers' daily lives, the application interface was adapted slightly. For the most part this was a resizing-of-content exercise, but the admin-reports screen was also moved to the tab-bar of the class-account area, so that once the initial administration exercise was complete (adding entities for each new user) then the tester could perform all actions from inside their class-account. Some of the content (language) was also changed so that it was more relevant to the tester's daily activities (eg. instead of behaviours relating to children with complex profound to

severe learning difficulties, they could record more relevant behaviours like tiredness etc). These adaptations were kept to a minimum in order to preserve the app's interface but were necessary to make it testable, relevant and engaging for the testers.

Unlike the usability testing, the testers in this group were given instruction on how to use the app and perform use cases before the test started. The objective of this round of testing was to measure how the product was perceived by users through repeated use. After the testing period, the testers were then asked to participate in a survey regarding their experience, and were also given the opportunity to express any overseen usability issues or improvement suggestions. The questionnaire provided for the survey was obtained from *UEQ-Online* - a website providing UX resources, including "A fast and reliable Questionnaire to measure the User Experience of interactive products" (Hinderks et al, 2018). The questionnaire has 26 questions, each with a numeric scale related to one of six usability and user-experience aspects. The questionnaire is accompanied by analysis tools, including a spreadsheet that aggregates the scores for each question and provides an overall score for the users' perception of the following aspects:

1. *Attractiveness*: Overall impression. Do users like or dislike the app?
2. *Perspicuity*: Is the app easy to get familiar with and learn how to use?
3. *Efficiency*: Can users complete tasks without unnecessary effort?
4. *Dependability*: Does the user feel in control?
5. Stimulation: Does using the app feel exciting and motivating?
6. Novelty: Does the app feel interesting, innovative and creative?

The full Questionnaire provided by UEQ-Online can be found in Appendix 4, and the accompanying, completed analysis spreadsheet can be found in the project's Supporting Material: 'UX_Analysis.xlsx'

The results of the survey for this user-experience testing phase were as follows:

Table 8: *Behaviour Support App - User-Experience aspect scores*

Behaviour Support App - User-Experience aspect scores					
Attractiveness	Perspicuity	Efficiency	Dependability	Stimulation	Novelty
2.00	1.70	2.23	1.80	1.98	1.80

(where highest score = 3.00 and lowest score = -3.00)

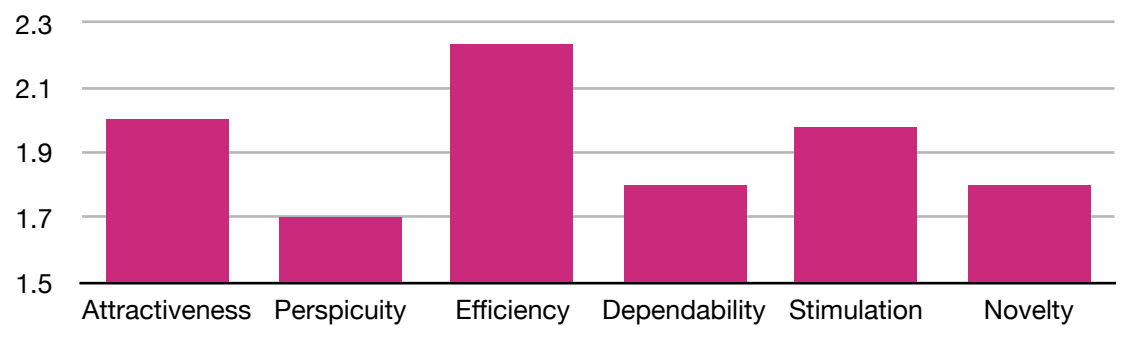
The analysis spreadsheet also provides a comparison to benchmark data that the site has gathered from (at the time of writing) 9905 people in 246 studies of different products. The results of this survey compare very well against the benchmark data provided by UEQ-Online:

Figure 64: *Behaviour Support App User-Experience scores vs UEQ-Online benchmark data*

Scale	Mean	Comparisson to benchmark	Interpretation
Attractiveness	2.00	**Excellent**	In the range of the 10% best results
Perspicuity	1.70	**Good**	10% of results better, 75% of results worse
Efficiency	2.23	**Excellent**	In the range of the 10% best results
Dependability	1.80	**Excellent**	In the range of the 10% best results
Stimulation	1.98	**Excellent**	In the range of the 10% best results
Novelty	1.80	**Excellent**	In the range of the 10% best results

The results show a very solid positive perception of the app. However, it must be acknowledged that there is potential for the results to be slightly skewed as the testers in this case all know the author personally so may have been inclined to answer more positively than if they were completely removed from the product's developer. Having said this, the results are still useful; because the testers' potential bias would affect *all* of the user-experience aspects, the scores *in relation to each other* are a good indicator of the app's user-experience strengths and weaknesses. When viewed graphically (as shown in figure 65), we can clearly see that testers perceived the app's strongest user-experience aspect to be its efficiency - perhaps in some part attributed to the flat-navigation design and easy data processing. In contrast, we can see that the weakest perceived aspect is the app's perspicuity, followed by it's dependability and novelty.

Figure 65: *Column-chart to show Behaviour Support App User-Experience scores*



Going forward, this result can be used to inform development of the app - being fed into the design-implementation-test iterative process; to improve the app's user-experience, highest priority should be given to making the app easy to learn/use (perspicuity), making the user feel in control (dependability), and making the app feel interesting, innovative and creative (novelty).

In order to explain and address the highest priority user-experience aspect, perspicuity, it is important to consider that some testers participated remotely - following a .pdf quick-start guide to explain how to use it and with no face-to-face demonstration of the app - before starting the test period. This probably contributed to a feeling of difficulty in learning the app - which is intended for a private organisation where, unlike most consumer apps, full training will be given by staff to end users. While a step-by-step tutorial could be implemented for new users, it would be costly and time consuming to create and maintain, so may not be the most most appropriate solution when there will be staff on hand to train new users at no extra cost. This is an aspect of the user-experience that should be monitored closely in subsequent iterations of the development process, to see if its relative low score persists even among fully trained users.

In addition to the survey results, several issues and suggestions were made by the testers - some of which relate well to the prioritised user-experience aspects identified through the questionnaire analysis. The most critical issue that arose during the test period was that one tester accidentally logged in as another user and was uploading data to the database on behalf of that other person. This reflects the user-experience aspect of *Dependability* as the users involved clearly did not have proper control of the app and their data. During development of the *Minimum Viable Product* (MVP) it was decided that user-authentication was not necessary for the first version as usage rules could be enforced through workplace policy, but this exercise highlighted how easy it is for mistakes to be made - mistakes which would severely compromise the integrity of the School's data.

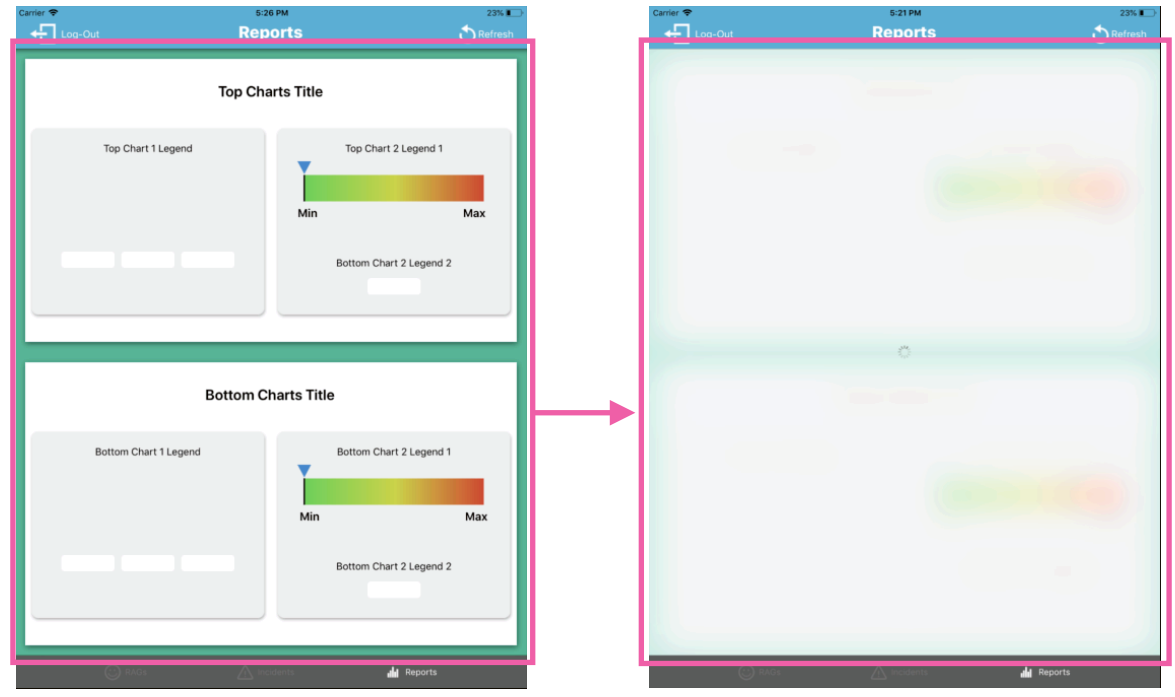
On the screen which enables the user to select a school period on a given day, for which they would like to complete a RAG Assessment, the buttons for each period have 'status labels' which appear with text 'Not Complete' and a red-coloured background if the user is late in completing the

assessment for the period - to urge the user to complete the task. One of the testers suggested that the app could go even further and present Notifications (that appear in-app and on the device's lock screen) to serve as more urgent reminders - and are effective even if the user has put the app into the background or locked the device.

Implementation (3)

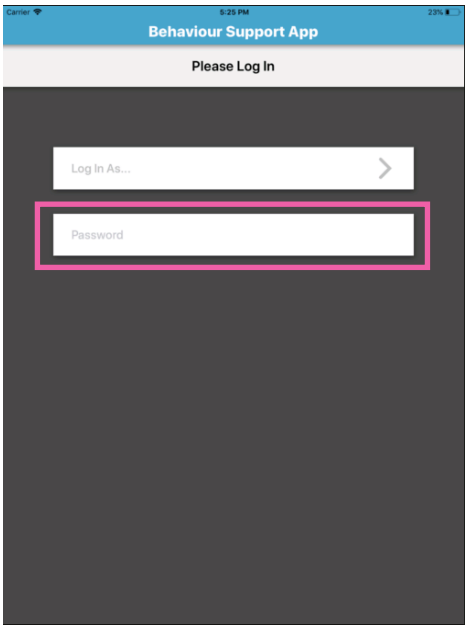
To enhance the dependability of the app and make the users feel more in control, an effort was made to increase app-status feedback to the user. While data is being fetched asynchronously from the database over the network, there can sometimes be a short delay with the interface appearing empty or incomplete in places. Wherever this occurs in the app, a lightly-blurred view has been placed on top of the empty/incomplete interface, with a spinning activity-indicator centred on top of it. The spinning indicator helps to show the user that the app is still working - has not frozen - even if the network is slow and data takes a long time to be returned. When the fetched data is received, the activity indicator stops and disappears - along with the blurred view - to reveal the now loaded and complete interface beneath.

Figure 66: *Adding blur and activity-indicator while data is being fetched over the network*



To improve the feeling of novelty - with innovation and creativity - animation was added to the charts so that each time a screen with charts on is presented to the user, the values are animated from zero to the set level. This helps make the data more compelling to read and gives life to the interface.

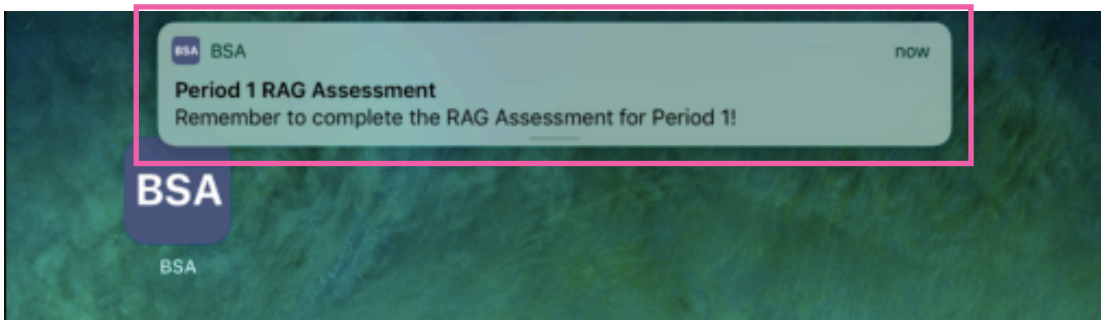
Figure 67: *Login screen with added 'Password' field*



In response to the issue of mistaken identity, password authentication was implemented - with management of passwords added to the Admin account's 'Class Details' screen. For clarity, the authentication currently provided is for users to log into the app only - to avoid this issue of mistakenly logging into the wrong account - it does not yet provide authentication with the Firebase database.

The idea of using notifications seemed like a useful suggestion that could help improve the app's perceived dependability, stimulation and novelty, so local Notifications were programmed into the app to be presented one hour after each school-period has ended, and only when the user is logged into a class-account - as the Admin account does not complete RAG Assessments.

Figure 68: *Notifications displayed at top of screen to remind user to complete RAG Assessments*



Assessing the app's user-experience is an ongoing exercise and should be undertaken alongside future iterations and releases. Some of the features that were stripped from the original design - while identifying the MVP - may help to enhance the user-experience and could be implemented in

the next version. For instance, implementing the originally designed curved (pressure-gauge-style) intensity-indicator, and also the staff/student profile-images, could improve the app's novelty feeling.

Usability Testing (2)

Once the user-experience improvements were made, one more round of usability testing was conducted - to test the new changes and make sure that the app was ready for release. The same testers were used as for the original round of usability testing. All the new changes were responded to positively by the testers and all fed back that they felt the app was more pleasing to use and easier to understand.

However, in addition to the usability and user-experience enhancements, this round of testing included two new features that were not previously active. The first feature enables the user to export .csv files with RAG Assessment and Incident Form data via email from the Admin Reports pages (this feature was added while user-experience testing was taking place). The second feature enables the user to delete entities (Classes, Staff, Students) from the database. This deletion feature was disabled during testing to prevent users accidentally deleting other users' data and ruining the test.

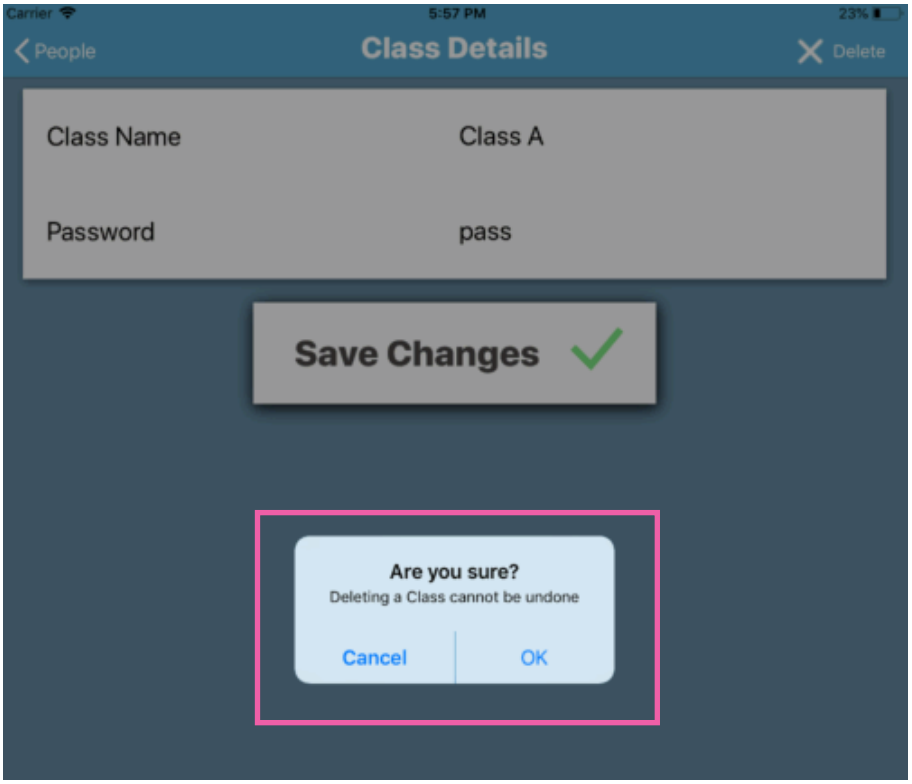
The data-export feature presented no problem to testers, they used it with ease. However, when asked to delete a Student entity, one tester accidentally deleted a staff member from the database. When the user presses the button, the action is immediately executed - the data removed from the database and the screen navigates back to the previous screen. This is an irrecoverable mistake, so it was very important to catch.

Implementation (4)

To improve the usability of the 'Delete entity' feature (and improve the app's perceived dependability), an alert was programmed to appear every time the user presses the 'delete' button - to ask for confirmation of the action and to issue a warning that it can not be undone. By adding

this extra step in the process and informing the user of the consequences, the app proactively attempts to prevent the user-error (accidental deletion of data).

Figure 69: *Alert displayed asking for confirmation before an entity is deleted from the database*



Conclusion

Having successfully conducted a round user-experience testing and two rounds of usability testing, the Behaviour Support App is now ready for deployment of version 1. Overall the author feels that the project has been a real success. Implementation of the Minimum Viable Product went as planned, even the features that were particularly daunting; the charts were a relatively high risk because the author had no experience with programming of charts or familiarity with the external library used. The testing phases uncovered some overseen usability issues and provided constructive insight to areas for improvement - insight that would have been missed by the developer alone. Finally, the project has successfully met all of the client's system and usability requirements. The client was kept up to date with progress throughout the process and is very pleased with the result. The existing available app options analysed early in the report fell far short

of meeting the client's needs, but the finished Behaviour Support App provides all of the functionality that the School has currently achieved, plus some additional functionality - in the form of the programmed reports/charts - that opens the client's mind to possible future analysis and reporting capabilities.

Learning Points

There were several particularly valuable learning points for the author during this project. These included:

1. Rigid time planning has no real advantage over an agile approach for software development. The original time plan for the project was comprised of a mixture of structured tasks and milestones, with periods of flexible time for implementation. While having a documented plan at the start of the project provided some structure and peace of mind, the reality during the project was that many tasks and milestones had to be moved around on-the-fly due to unforeseen circumstances or unexpected development times with unfamiliar features/tools. Ultimately, these adjustments were done mentally - there was no time to keep updating the documentation - so the documented plan did not always reflect the actual progress of the project (apart from when it was updated for the interim report submission). While it is important to understand the tasks required to complete a project, and any dependencies or hard milestones, the author also learned the importance of allowing for flexibility in time planning and task completion.
2. Better knowledge of dynamic-layout tools will save a lot of time and effort when accommodating different screen sizes. Fortunately, the School all use devices (iPads) with the same size screen so the app could be developed for a single set of dimensions. However, when adapting the app for use on iPhones during the testing phases, the project was duplicated and a lot of time and effort was spent re-engineering the interface and rebuilding with Xcode's auto-layout tools. When maintaining a project it could lead to quite a messy situation if code is duplicated between different projects for different devices and all updates need to be rolled-out to each one in turn. If If used correctly, there are 'Universal-layout' tools

and techniques that can accommodate all screen dimensions and device models - but they are not quick enough to learn to be made use of in this project.

3. Thorough planning of the data model before implementation would save time. In the original plan, the data model and back-end were expected to be developed before implementing any of the front-end. As it turned out, the back-end was implemented after the majority of the front-end was completed. However, the Firebase database turned out to be only compatible with and optimised for certain data types - so when connecting the front-end to the back-end, a lot of time was spent re-engineering the way data is passed, stored and associated with objects in the front-end. This cost a relatively large amount of time and effort to reconfigure.
4. It would be useful to track the app's User-Experience Evaluation over time. The User-Experience Evaluation conducted in this project provided a useful snapshot, but it would be even more useful to see how the users' perception changes over time - through repeated use as well the addition of any new features. In hindsight it would have been beneficial to ask testers to complete the same questionnaire at the start of the test period (to get a view of their initial impressions) so that the data could be compared to that gathered at the end of the test period.
5. It would be useful to gather test data from a larger, and preferably anonymous, user group. While the testing conducted here provided useful insight into the product, a larger sample of users - who also don't know the developer - would likely give more accurate representation of the app's usability and user-experience.
6. The back-end could be used to track usage and measure effectiveness of the product. The Firebase back-end has many analytics features and if properly configured could be used to track targeted outcomes (eg. completion of RAG Assessments or Incident forms in good time). If data could be collected about the current system, these tools could have a particularly powerful impact; one of the main reasons for developing a new solution was to improve the timeliness of RAG Assessment/Incident data recording to preserve data integrity, so tracking these outcomes digitally could provide the data necessary to prove the value of the new system.

Considerations for Future Development

This project constitutes the first iteration of a product that could evolve through subsequent versions in response to the client's changing requirements and feedback from users. At this stage, there are several apparent ways that the product could be improved for the next version and beyond:

1. The features that were stripped from the original full-concept design when identifying the Minimum Viable Product could be implemented. These features include the enhanced intensity-indicator chart, staff/student profile images, Calendar screen, collapsible and embedded student-selection table for Admin reports, and a historical views of completed Incident forms.
2. Bluetooth wearables could be paired with the application - a wearable alert-button (on wristband/pendant) could be used by staff to unobtrusively and accurately notify the app that an incident has started/ended, or Fitbit-style wristbands could be worn by students to track and send heart-rate data about the individual to the app (to investigate possible ways to detect and prevent incidents before they occur).
3. The system could be made more 'active' - in the sense that rather than waiting for the users to view the analysed data and make decisions, it could proactively notify users (inside the app or externally via email etc) when certain conditions are detected. For instance, if a student's data shows they are having a more turbulent time than the same time period last week/term/year, then the user could be automatically notified to investigate why. Notifications could also be sent to senior management whenever incidents occur, so that they are always up to date with extreme behaviour issues - without needing to manually check the app many times a day.
4. A Web browser-based portal could be developed for more detailed management of the database and to provide desktop-device access.
5. Another app (or separate interface) could be developed to allow parents/carers to access data that is relevant to the child in their care. This could be optimised to provide appropriate data privacy (so that parents/carers can't see data associated with other children) and use tailored language to present the most appropriate and positive view/outlook for each child - rather than the current, purely objective language and style used.

6. The ability to generate reports specifically designed for presenting to parents/carers during ‘parents’ evening’. Since the requirements development phase, the client has expressed how useful this feature would be - to create and print reports that show selected children’s historic data and highlights the progress they have made during their time at the School.

References

Hintze J.M. (2001). *Best Practices in the Systematic Direct Observation of Student Behaviour 63.*

Semmel M.I. (1978). *Systematic Observation.* Journal of Teacher Education. vol.29.

NICE guideline (2015). *Challenging behaviour and learning disabilities: prevention and interventions for people with learning disabilities whose behaviour challenges.* pp. 12-13.

The Challenging Behaviour Foundation (2018). *Positive Behaviour Support Planning: Part 3.* Viewed July 2018. <http://www.challengingbehaviour.org.uk/learning-disability-files/03---Positive-Behaviour-Support-Planning-Part-3-web-2014.pdf>

St Ann’s School (2017). *Positive Behaviour Support Policy.* Viewed July 2018. <http://docs.wixstatic.com/ugd/2090b6_236d8e77016e496f83b76bc8ca39b327.pdf>

Imray P, Colley A, Holdsworth T, Carver G and Savory P (2017) *Listening to Behaviours: adopting a Capabilities Approach to education.* The SLD Experience. 76: 3-9.

Harrison S, Tatar D, Sengers P (2007). *The three paradigms of HCI.* pp .10.

Poslad S (2009). *Ubiquitous Computing: Smart Devices, Environments and Interactions.* John Wiley & Sons. pp. 135-178.

eKrios Consulting (2017). *Best Behaviour* (Version 1.0.8) [Mobile application software]. Retrieved from https://itunes.apple.com/gb/app/best-behavior/id975986231?mt=8

Track & Share Apps (2018). *Autism Tracker Pro* (Version 7.5.0) [Mobile application software]. Retrieved from https://itunes.apple.com/gb/app/autism-tracker-pro/id478225574?mt=8

WhizzWhat Software (2017). *Functional Behaviour Assessment Wizard* (Version 1.2.0) [Mobile application software]. Retrieved from https://itunes.apple.com/us/app/functional-behavior-assessment-wizard/id573375887?mt=8

Birdhouse (2016). *Birdhouse - for Special Education Teachers* (Version 3.0.0) [Mobile application software]. Retrieved from https://itunes.apple.com/gb/app/birdhouse-for-special-education-teachers/id1060115001?mt=8

Moy S (2016). *Nulite Behaviour Tracker for Special Education* (Version 1.2) [Mobile application software]. Retrieved from https://itunes.apple.com/us/app/nulite-behavior-tracker-for-special-education/id1034110542?mt=8

Preece J, Rogers Y, Sharp H (2002). *Interaction Design: beyond human-computer interaction.* John Wiley & Sons. pp. 14-15.

Gallitz W.O. (2002) *The Essential Guide to User Interface Design*. John Wiley & Sons. pp.55-58.

Hinderks A, Schrepp M, Thomaschewski J (2018) *User Experience Questionnaire (UEQ)*. UEQ_S_Data_Analysis_Tool.xls, UEQ Handbook_V4.pdf. Viewed and retrieved July 2018 <https://www.ueq-online.org>

List of Figures

Figure 1: *Data capture forms and reports for the current system*

Figure 2: *Best Behaviour - User Interface*

Figure 3: *Best Behaviour - User Interface*

Figure 4: *Functional Behaviour Assessment Wizard - User Interface*

Figure 5: *Birdhouse - For Special Education Teachers - User Interface*

Figure 6: *Nulite Behaviour Tracker for Special Education - User Interface*

Figure 7: *Login screen navigation*

Figure 8: *Navigation to RAG Assessment screen*

Figure 9: *Incidents Screen - navigation to Incident Form*

Figure 10: *Example of screens for value-selection for Incident Form*

Figure 11: *Class Reports screen*

Figure 12: *People screen entity-container navigation*

Figure 13: *Navigation to entity-details form - for existing or new entities*

Figure 14: *Calendar screen*

Figure 15: *Navigation between Admin Report containers*

Figure 16: *Example JSON objects as stored in Firebase*

Figure 17: *Entity Relationship Model diagram for Behaviour Support App*

Figure 18: *System Architecture for Behaviour Support App*

Figure 19: *Original Time Plan Gantt Chart vs Revised Time Plan Gantt Chart at Interim Report Stage*

Figure 20: *Selection of UI layouts from Wireframe Prototype*

Figure 21: *Selection of UI layouts from Full Concept Prototype*

Figure 22: *Login screen for Minimum-Viable-Product*

Figure 23: *RAG Assessments screens for Minimum-Viable-Product*

Figure 24: *Altered Incident-related screens for Minimum-Viable-Product*

Figure 25: *Adjusted Class Reports screen for Minimum-Viable-Product*

Figure 26: *Adjusted People screen for Minimum-Viable-Product*

Figure 27: *Adjusted Admin Reports screens for Minimum-Viable-Product*

Figure 28: *Project folder structure*

Figure 29: *Selection of Interface-Builder layout views*

Figure 30: *CocoaPods Podfile contents*

Figure 31: *User Interface for views developed in Sprint 1*

Figure 32: *Example of AnimationEngine methods being used to animate screen layouts*

Figure 33: *Example of Material Design formatting of objects*

Figure 34: *Example of custom drawing on views*

Figure 35: *Example updating method for period-button status*

Figure 36: *Example of setting status of RAG Assessment cell*

Figure 37: *Example definition of a protocol for the delegate pattern*

Figure 38: *Caching of selected-user data and and segue to appropriate account area*

Figure 39: *Alert presented if RAG Assessment is incomplete*

Figure 40: *User Interface for views developed in Sprint 2*

Figure 41: *Enum used to define all possible types of 'Report Item'*

Figure 42: *Setting of Report Item type and modifying appearance*

Figure 43: *Adding a gradient layer to the Intensity-Indicator*

Figure 44: *Example of chart column animation*

Figure 45: *Protocol definition for Date/Time-selection delegate*

Figure 46: *Implementation of protocol method for Date/Time-selection delegate*

Figure 47: *Invocation of delegate protocol method to pass data between View-Controllers*

Figure 48: *Passing of data from Class-Report dataset to individual chart elements*

Figure 49: *Animation of charts every time time the view is presented*

Figure 50: *User Interface for views developed in Sprint 3*

Figure 51: *User Interface for views developed in Sprint 4*

Figure 52: *Configuration of Day-View chart to read values from given Dataset*

Figure 53: *Method set to unpack given Dataset values for use in Day View and Week View charts*

Figure 54: *Method to unpack given values for use in the Incident Characteristics charts*

Figure 55: *GoogleService-Info.plist file in product folder*

Figure 56: *Firebase configuration command*

Figure 57: *Example of 'constant' values used for consistent database referencing*

Figure 58: *Example view of the project's Firebase database with data present*

Figure 59: *Example delegate method for handling data received from the DataService class*

Figure 60: *Example of adding a swipe-gesture recogniser to the view*

Figure 61: *Adding text to specify number of selectable items in Incident-Form selection lists*

Figure 62: *Adding text to help clarify meaning of Admin Report charts*

Figure 63: *New colour schemes to help distinguish horizontal-bar-charts*

Figure 64: *Behaviour Support App User-Experience scores vs UEQ-Online benchmark data*

Figure 65: *Column-chart to show Behaviour Support App User-Experience scores*

Figure 66: *Adding blur and activity-indicator while data is being fetched over the network*

Figure 67: *Login screen with added 'Password' field*

Figure 68: *Notifications displayed at top of screen to remind user to complete RAG Assessments*

Figure 69: *Alert displayed asking for confirmation before an entity is deleted from the database*

List of Tables

Table 1: *Best Behaviour app - advantages vs disadvantages*

Table 2: *Autism Tracker Pro app - advantages vs disadvantages*

Table 3: *Functional Behaviour Assessment Wizard app - advantages vs disadvantages*

Table 4: *Birdhouse - For Special Education Teachers app - advantages vs disadvantages*

Table 5: *Nulite Behaviour Tracker for Special Education app - advantages vs disadvantages*

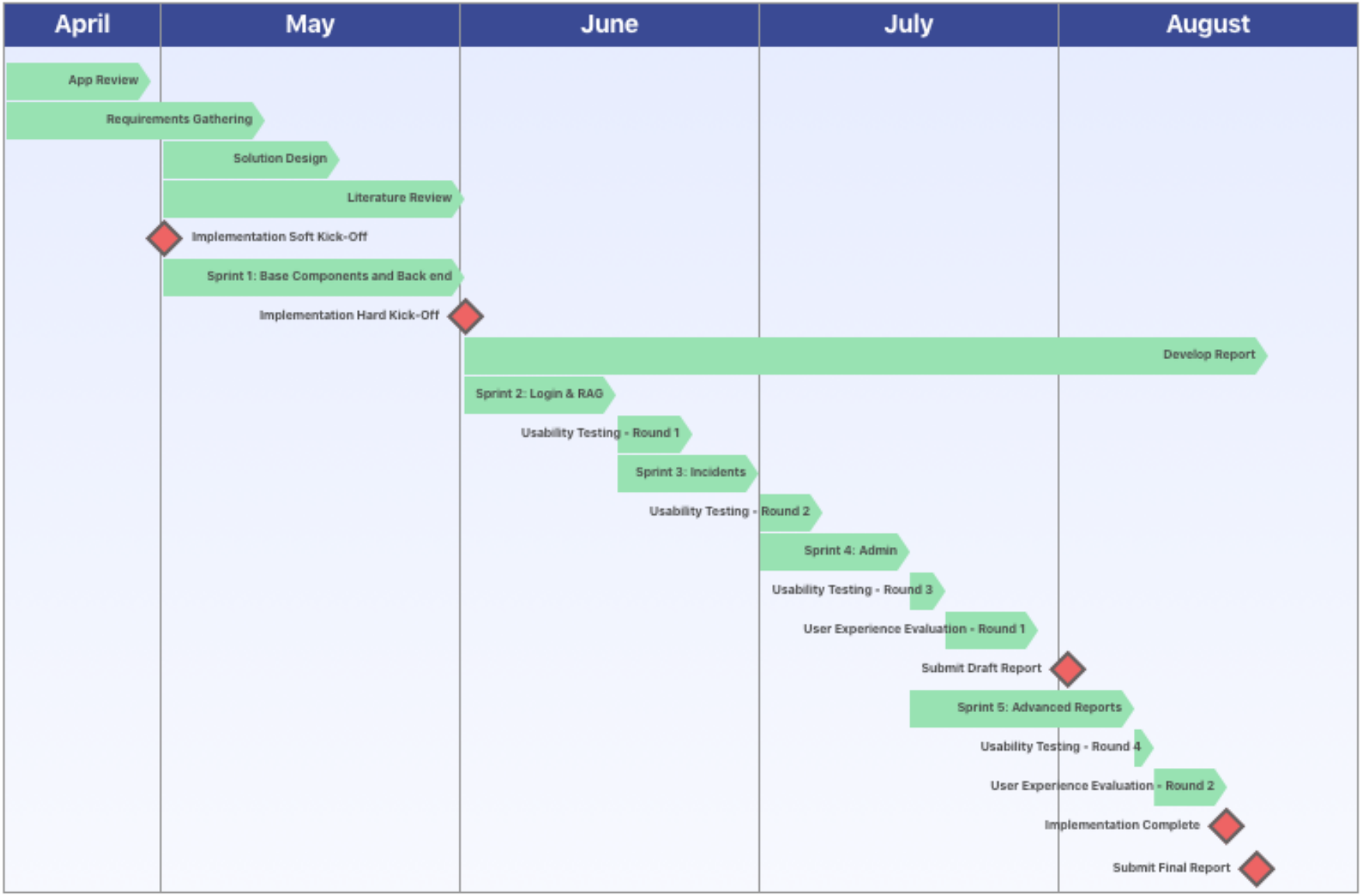
Table 6: *Development Support Tools*

Table 7: *External Code Libraries*

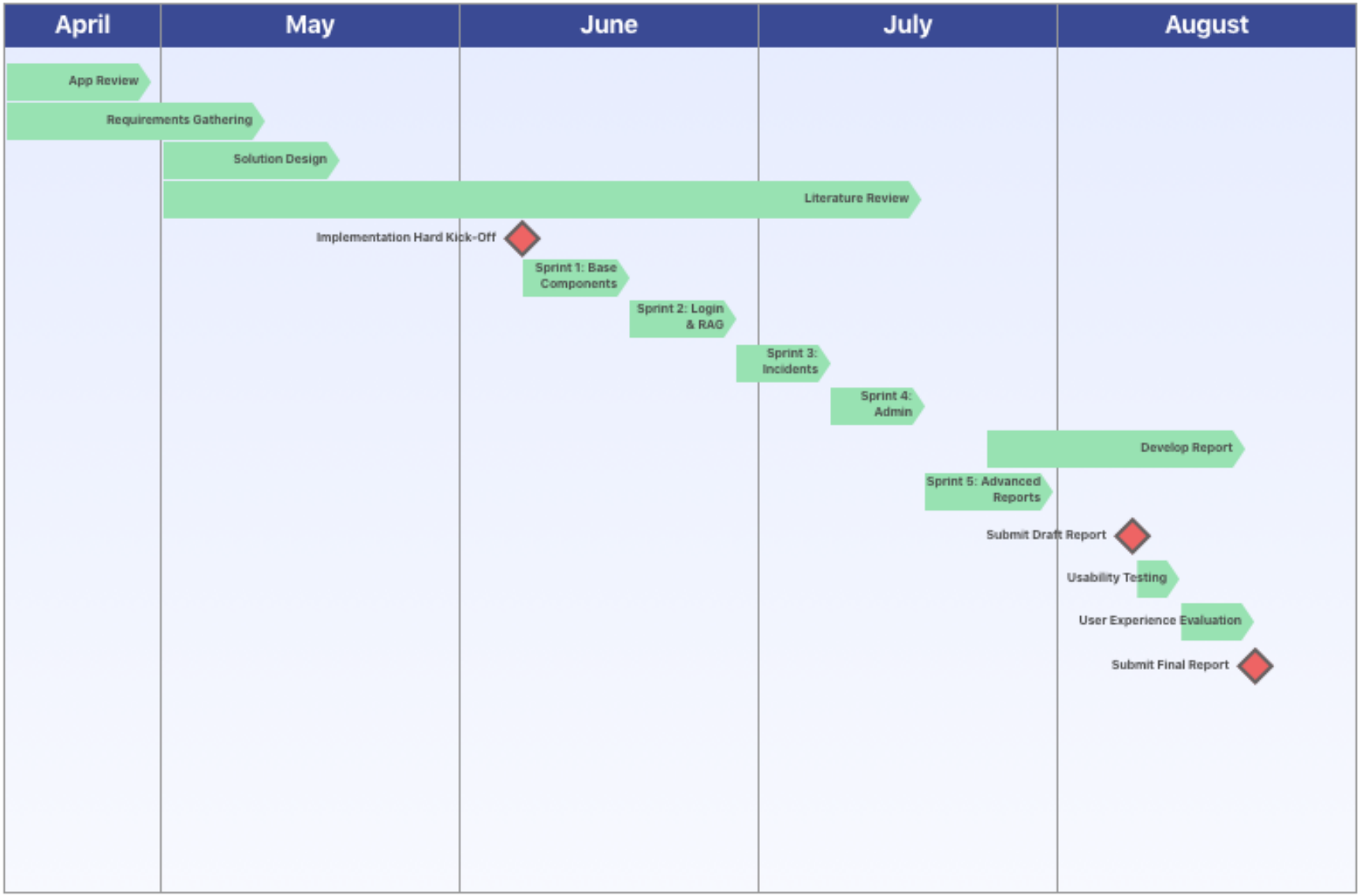
Table 8: *Behaviour Support App - User-Experience aspect scores*

Appendices

Appendix 1: Initial Project Time-Plan



Appendix 2: Revised Project Time-Plan at Interim Report stage



Appendix 3: Usability Testing use-case tasks:

- Log-in to the ‘Admin’ account and add a new ‘Class’ entity
- Log-in to a ‘Class’ account and record a new Incident
- Complete a RAG Assessment for Class A, period 6, today (all Green)
- Check the quick report for a ‘Class’ account
- View ‘Incident Characteristics’ report for the whole school
- Add a new Staff Member
- Add a new Student to ‘Class C’
- Move a Student to a different Class
- Complete the RAG Assessment for yesterday, Period 4 (with at least on Red)
- Amend the RAG Assessment for two days ago, Period 2 (with at least 2 Ambers)
- View the Day-View Admin Report for Class B
- View the Week-View Admin Report for 3 students only
- View the Admin Report of a single Student and export the data in .csv format
- Delete a Student from the database

Appendix 4: User-Experience Evaluation Questionnaire

Please rate the Behaviour Support App on the following scales (tick one for each)

	1	2	3	4	5	6	7		
annoying	○	○	○	○	○	○	○	enjoyable	1
not understandable	○	○	○	○	○	○	○	understandable	2
creative	○	○	○	○	○	○	○	dull	3
easy to learn	○	○	○	○	○	○	○	difficult to learn	4
valuable	○	○	○	○	○	○	○	inferior	5
boring	○	○	○	○	○	○	○	exciting	6
not interesting	○	○	○	○	○	○	○	interesting	7
unpredictable	○	○	○	○	○	○	○	predictable	8
fast	○	○	○	○	○	○	○	slow	9
inventive	○	○	○	○	○	○	○	conventional	10
obstructive	○	○	○	○	○	○	○	supportive	11
good	○	○	○	○	○	○	○	bad	12
complicated	○	○	○	○	○	○	○	easy	13
unlikable	○	○	○	○	○	○	○	pleasing	14
usual	○	○	○	○	○	○	○	leading edge	15
unpleasant	○	○	○	○	○	○	○	pleasant	16
secure	○	○	○	○	○	○	○	not secure	17
motivating	○	○	○	○	○	○	○	demotivating	18
meets expectations	○	○	○	○	○	○	○	does not meet expectations	19
inefficient	○	○	○	○	○	○	○	efficient	20
clear	○	○	○	○	○	○	○	confusing	21
impractical	○	○	○	○	○	○	○	practical	22
organized	○	○	○	○	○	○	○	cluttered	23
attractive	○	○	○	○	○	○	○	unattractive	24
friendly	○	○	○	○	○	○	○	unfriendly	25
conservative	○	○	○	○	○	○	○	innovative	26

This Questionnaire provided by Hinderks et al (2018) through the UEQ-Online website was transposed to a Google-Form for distributing to testers and collecting results. The Google-Form used can be found at https://goo.gl/forms/GbdN6Hhk5F5YIgMb2