**Contents**

**Analysis - 3**

* Problem Identification - 3
  + Hardware and Software Requirements - 3
  + Computational Suitability - 4
  + Stakeholders - 6
* Stakeholder Questionnaire - 7
  + Survey Follow up - 13
* Research - 14
  + ASDA’s hardware / software - 14
  + League of Legends pinging system - 16
  + Proposed additions for a Solution - 17
* Success Criteria - 18
  + Limitations - 23

**Design - 24**

* Decomposition - 24
  + Decomposition Flowchart - 26
* Algorithms and Pseudocode - 27
  + Top layer - 27
  + Second layer - 28
    - Login screen - 28
  + Third layer - 32
  + Database Branch - 32
  + Hardware Branch - 38
  + Software Branch - 38
    - Communication - 38
      * Ping Sending - 38
      * Ping Receiving - 40
    - Shopfloor - 41
      * Item Lookup - 41
    - Backroom - 41
      * Picking - 41
    - Barcode Scanner - 43
* Key Variables - 45
* Key Classes - 46
* Key Data Structures - 47
* Inputs and Outputs - 47
* User Interface Design - 48
  + General Usability Features - 48
  + General Success Criteria - 48
  + Login Screen - 49
  + Barcode Scanner - 50
  + Pinging System (Inbox) - 51
  + Stakeholder Feedback - 51
  + Validation - 52
* Testing Methods - 53
  + Iterative Interface Inputs and Test Table - 56
  + Post Development Tests - 60

**Iterative Testing and Development - 61**

* Stage 1 - Login Activity - 61
  + Test 1 - 65
  + Midpoint Review - 65
  + Test 2 - 67
  + Review - 68
* Stage 2 - Main Menu - 70
  + Test 1 - 72
  + Midpoint Review - 73
  + Test 2 - 76
  + Review - 77
* Stage 3 - Barcode Scanner - 79
  + Test 1 - 81
  + Review - 82
* Stage 4 - Item Info - 83
  + Test 1 - 84
  + Midpoint Review - 85
  + Test 2 - 89
  + Test 3 - 91
  + Review - 92
* Stage 5 - Communication - 93
  + Test 1 - 96
  + Midpoint Review - 98
  + Test 2 & 3 - 102
  + Test 4 - 103
  + Test 5 - 104
  + Review - 104
* Stage 6 - Backroom - 105
  + Midpoint Testing - 109
  + Test 1 - 112
  + Test 2 - 113
  + Review - 113

**Evaluation - 115**

* Testing Post Development - 115
* Stakeholder testing - 116
* Criteria Met - 120
  + General Success Criteria -123
* Usability Features - 125
* Limitations - 125
* Maintenance - 126
* Conclusion - 127
* Bibliography - 127
* Links - 127

**Code - 128**

**Analysis**

Problem Identification:

For my computer science coding project, I have decided to remake software for the product scanning devices used in my place of work, a large chain supermarket. It will be called the “Automated Sorting System” or ASS for short. Currently, a system is used that can scan any barcode in order to stack shelves; audit item locations and information; rearrange zones and modulars; calculate waste and connect to printers that print barcodes. These run on handheld devices that use infra-red barcode scanners and an android operating system. The problem arises that these devices are often slow, won’t scan the larger barcodes seen on large boxes and there are no tools for communication between staff. They are also missing small quality-of-work improvements, such as the ability to create filter lists for bins and sort bins. This slows down work and productivity, as tricks and loopholes must be used to overcome these barriers, where the user must actively work against the system rather than with it. This system would allow the user to carry out their job faster.

My version of this software shall also operate on an android system, using a phone's camera instead to scan barcodes. Like the current system, it shall run on an existing database architecture, however new databases will be designed for the pinging and filter system that will be implemented. This makes it backwards compatible with older devices and allows for existing systems to be reused. I also plan to graphically update the user interface by adding consistent brand colours.

The code will be written in a mixture of Kotlin to work with the android OS and SQL to perform database checks.

Hardware and software Requirements:

* A camera enabled device- The device will need a camera in order to scan the barcode on products, of a high enough resolution in order to successfully scan the products. Without it, they will have to be manually inputted by searching the UPC of each product, which will drastically slow productivity.
* An internet enabled device - The device will need internet connectivity in order to connect to the servers running the databases, without it the user will not be able to log into the system and the software will be inaccessible. Ethernet will not work with the devices, as they are designed to be portable, but may be used as a last resort.
* Android 5.1 (Lollipop) - This is the operating system that the software will be developed for, chosen for the ability to make the application backwards compatible with the current devices, if barcode scanners are supported in the future.
* Databases running on a server - This is needed in order to facilitate the requests made by users. These should be running and accessible at all times so that users can access it even outside of work and should also sync across all devices when updated. This does not mean that devices will get live updates however after a change, it should show any new values upon the next access.

Computational Suitability:

This problem is suitable for computational solutions because several large databases must be maintained and upheld, through large amounts of changes that happen regularly, without damage or disfiguration of their integrity. In addition, communication features must be used that require devices in order to send and receive information. The main use of the computer in this system is taking in a wide variety of inputs from the user, such as barcodes and item information, then storing and performing tasks on that data for the user; as well as collecting the relevant information from a large database quickly and displaying it in a readable format. Algorithms can be reused from the previous system in order to select what items go on the shelves, however new ones must be created in order to access databases for the filter system as well as the distribution and storage of pings. Finally, it is suitable for computational solving because it requires interaction with the smartphone’s camera that cannot be done without a computer.

Computational Methods:

* Problem recognition- Overall, the main problems are being able to edit and display databases, not only efficiently and understandably but also while optimising the layout so that the user can obtain the specific information that they want in the quickest possible time. My initial impressions are that this will require a lot of communication and feedback from the stakeholders during the UI design process. Other than this, most of the functions of this system can be validation of inputs and outputs to maintain integrity of databases, by using algorithms on them to make sure they are inputted to the correct length (len checks) and contain the correct characters, either numbers or numbers and symbols.
* This lends itself well to computational methods, as repeatable SQL code can be used to extract the data from large databases. This can save time as the database can be updated and stored portably, quickly and with minimal storage. This also saves on resources, as the repeatable code means more time and focus can be spent working on the current problem, reducing errors in storage and retrieval. The data can also be represented in a number of ways by using computational methods rather than obtaining a mass of raw data that can be hard to interpret.
* Abstraction- This will be used in order to only display the essential information to the user, rather than all of the information that the computer must deal with. Data such as specific keys, designed to count items on the database, won’t be displayed to the user. This will help to speed up loading times and reduce confusion when using the software.

Inputs will be minimised to one or two buttons to input one or two points of data and outputs will be displayed graphically and only pertaining to the small scale impact. For example, adding a new item will not display the entire of the new database for items, but instead a small message saying “Item added” will be displayed as that is all the user needs. This allows anyone to use the software, regardless of intelligence or lack of technological prowess.

This is also suited to computational solutions as the large databases will have to be optimised in order to store only data that is necessary. This can cut down on money spent on data storage and time spent loading item information. This will also help keep the databases accurate, normalised and contained, reducing the chance for error or inaccuracies.

* Problem Decomposition- Most problems can be broken down into the following routine:
  + 1. The user selects the function of the device that they want and the correct menu is displayed
  + 2. The user inputs any data that is needed by the function, such as barcodes, numbers etc.
  + 3. The program validates this input internally by checking with format prerequisites and externally by comparing it with the database
  + 4. If the inputs are correct and valid, the function will be carried out to completion and produce the correct output.

Once these steps can be carried out so that only external factors, such as hardware limitations and internet connectivity, are the only factors drastically impacting speed (and that their impact will be as minimal as possible), then the software will be optimised and the solutions will be suitable for its purpose.

By decomposing the problem, the large database can be interacted with in such a way that only the information needed can be accessed and edited, reducing the chance for inaccuracies and the amount of misinformation it would store. This also helps to develop all the different ways to access and manipulate the database separately, so that they can be developed with shareable parts (such as repeatable SQL code). This will save on storage space and the size of the application, as well as ensuring all the parts are as optimised and the system is simple to use, yet it’s functions are wide reaching.

* Divide and Conquer- Each function in the software would be daunting to chain together, so subroutines (or multiple subroutines) will be designed for each function. This allows for them to be repeatable, callable at any time in the software and easier to program, as each problem can be tackled independently and modified independently of the rest. This also allows the user to access one function of the software directly, without having to pass through unnecessary steps, increasing productivity for the managers and simplicity and accessibility for the employees.
* Decomposing a large database will also keep it easier to maintain, reduce loading speeds and ensure that the correct functions on the system matches with the correct information it requires access to. This makes it suitable as it can reduce the amount of code that has to be repeated by sorting it into sections; while ensuring the the unique code for each part is written in order to allow each part to working independently of each other.

Stakeholders:

Overall, the stakeholders for the system will be anybody that has a stake in supermarket operations and their efficiency. This includes management, the employees and to a lesser extend the customers and suppliers. However, because of their smaller stake, the feedback obtained from stakeholders would optimally not include the latter, only employees and management. As management does not use this system, as they do not work on the shop floor (where the system is designed to be used), the sample will represent employees far more than management. However, management will be used more for consulting on new functions in order to assure that supermarkets would want their employees to be able to access these features.

* Management- A faster and more effective system means more profit as more stock gets on the shelves. The software should work with existing hardware to lower the cost with buying the existing software and it should not be slower than the current model despite the extra features. Running the software on personal devices lowers cost, as specialist hardware doesn’t have to be bought and maintained when expanding employees or replacing faulty hardware.
* They will use this proposed solution to update information about the item that would affect the wealth of the inventory of the store (how much money the store has in terms of products). This can include waste management and the total number of items in the store. They will also use this system to communicate with the employees by sending pings pertaining to information about work.
* The system is suited to their needs because it will be robust when working with large databases, ensuring that no corruption happens when interacting. It also has high security measures, such as a log in function to prevent unwanted access to parts of the application. It is also suited to their needs because it will allow them to make the changes necessary, quickly and easily while condensing steps.
* Employees- An easier system to use means they can be more effective with their work. The software should be intuitive so that they can focus on the task at hand rather than specific gun uses. If it is too complex then essential functions may go unused, resulting in the employee working against the software rather than with it. Adding new features should be designed with cutting down work time primarily, with streamlined functions that can condense multiple steps into one, as aesthetic and backend work aren’t as important to them.
* The employees will want to use all the “standard” functions on the system, such as picking, dropping and binning. They will also need to be able to log in, send and receive pings as well as use their phone camera to scan barcodes. All features on the device should be available to them other than the administrative functions that will be locked behind an admin check.
* The system is suited to their needs because it will be basic and intuitive to users. This is due to their high workload, requiring highly abstracted and informative information from the system. It will also prevent invalid inputs from being processed, allowing them to work without fear of making a mistake and allowing for training on the job to take place.

Stakeholder Questionnaire:

A questionnaire was issued to 8 asda colleagues; 1 management, 4 warehouse and 3 ambient. It consisted of 10 questions (8 multiple choice and 2 with feedback) and the results and implications are shown on the next page. Questions were designed to find out more about their thoughts on the current system as well as their thoughts on how new proposed functions could integrate with their work.

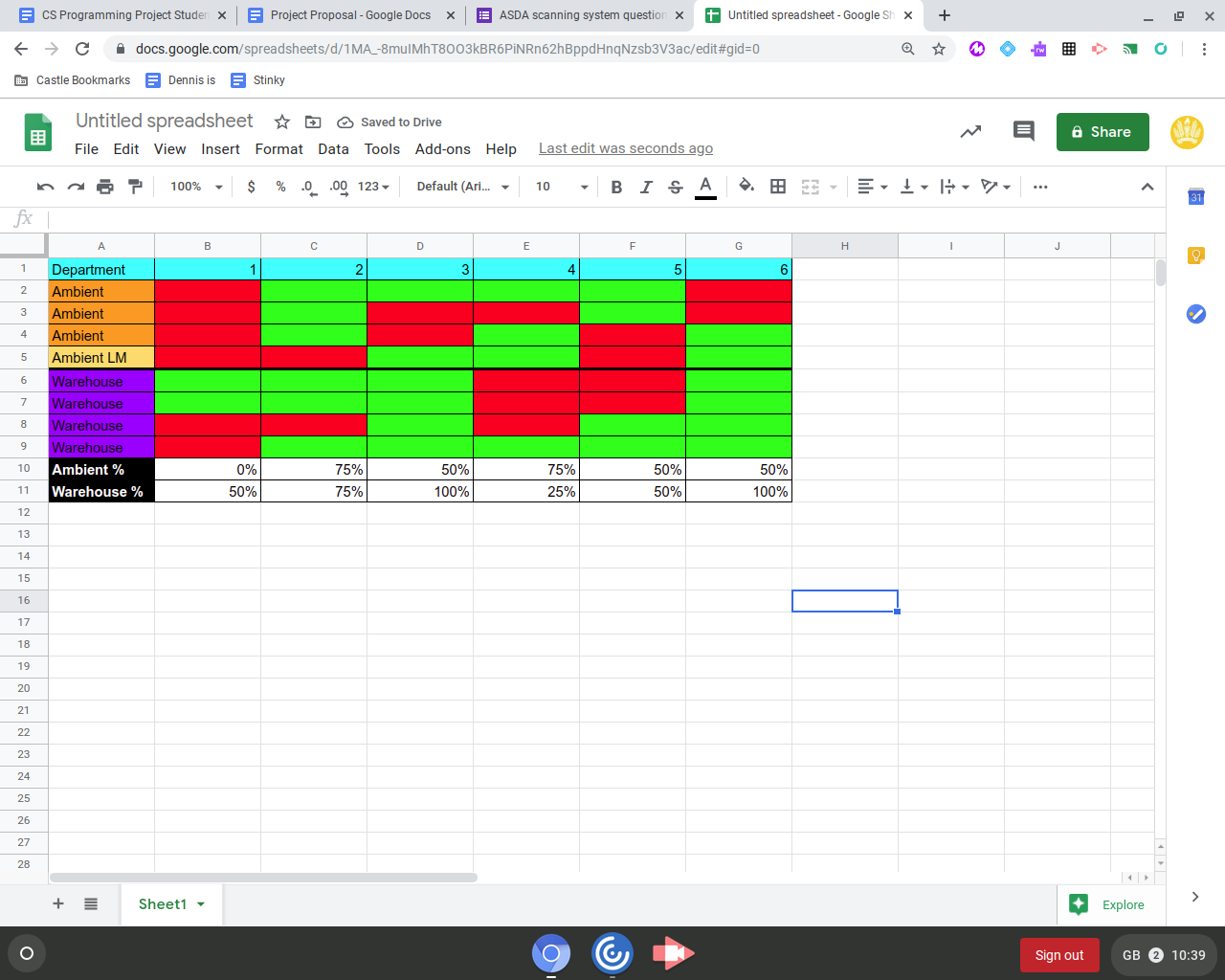
The 6 new functions proposed to colleagues were:

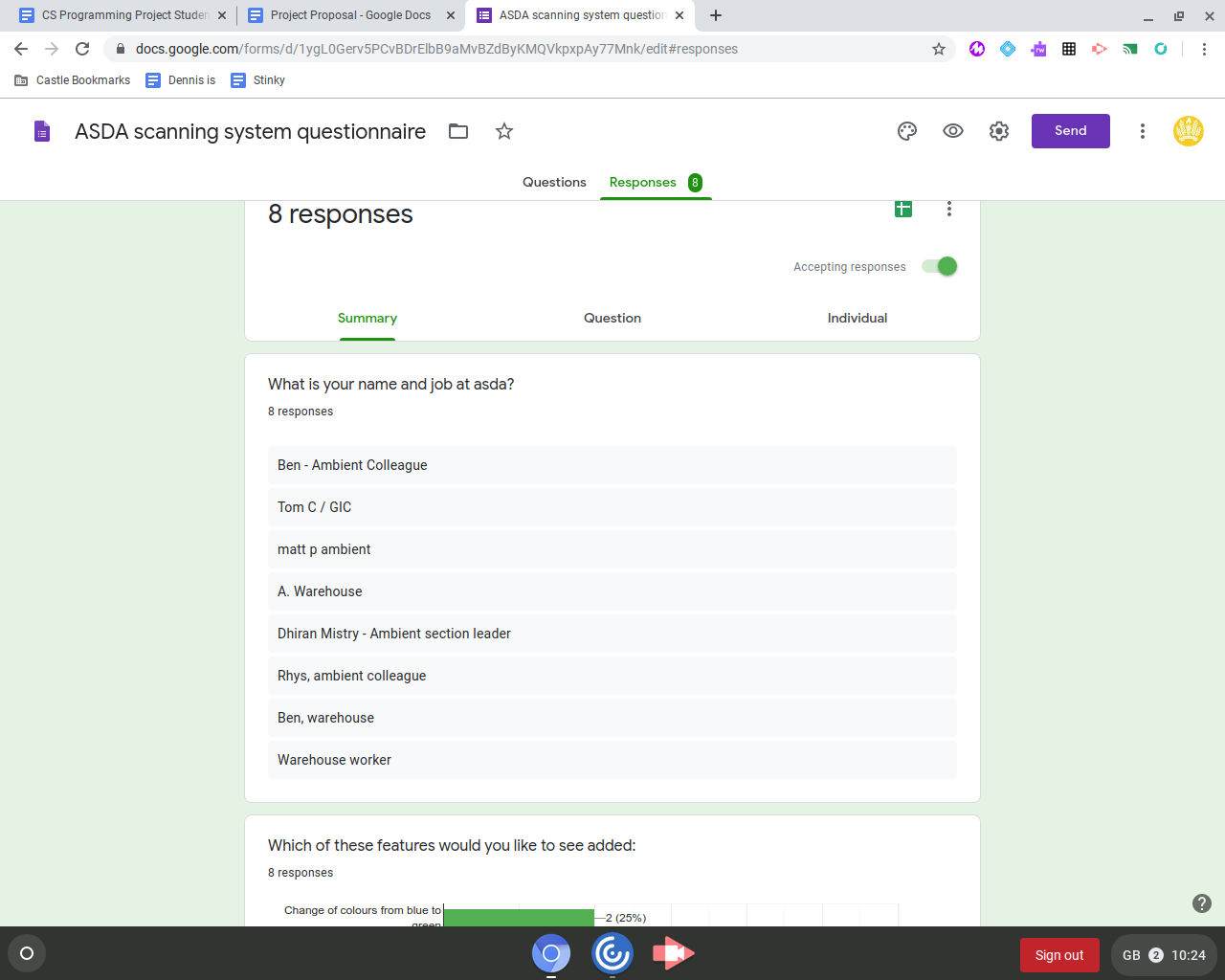
1. A change of colours from blue to green, so that the branding was more inline with their shop.
2. A system that lets you notify your managers and other colleagues of aisles, bins and jobs that need working, in order to have communication inside of the store; henceforth it shall be known as the “pinging system”.
3. Custom filters and sorting system for the list of bins and departments when picking, in order to reduce time spent redoing lists for different situations.
4. A countdown until the next picking drop time, in order to better manage time.
5. Using temporary numbers when updating a max shelf limit or total products rather than manager approval, in order to prevent the picking algorithm from creating picks that are incorrect and save the employees time.
6. ASDA radio, in order to listen to store sanctioned music while not in the public view.

After the survey was conducted, additional questions were asked in person about the results of the survey, and will be outlined after all the questions.

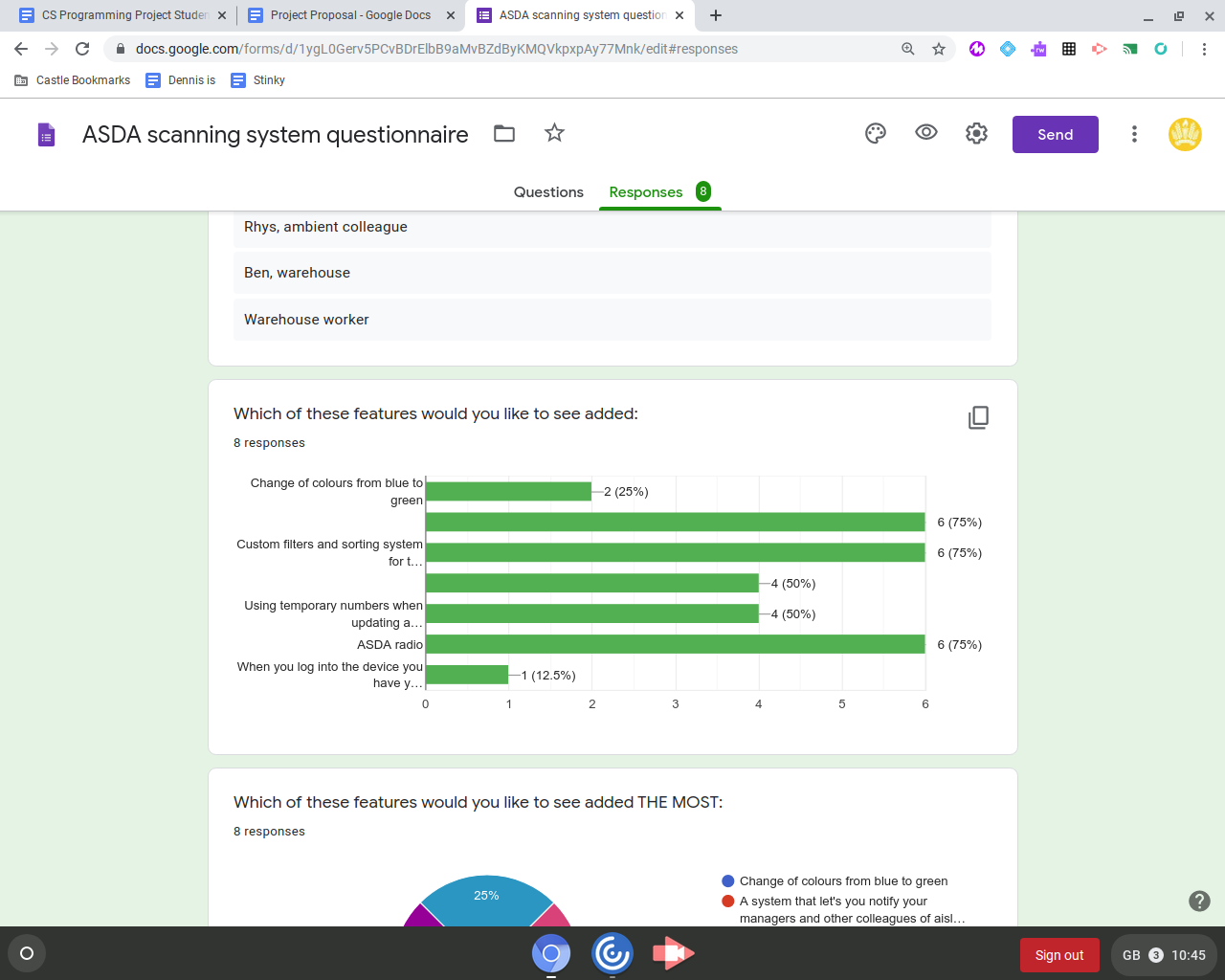
Question 1: What is your name and job at ASDA?

This question was added to enable me to understand where the feedback was coming from and how it would impact each department, as well as understand each department's needs. This table below displays the needs of each worker divided by department and the % of need for each function per department; where green represents an explicit need and red represents no explicit need.



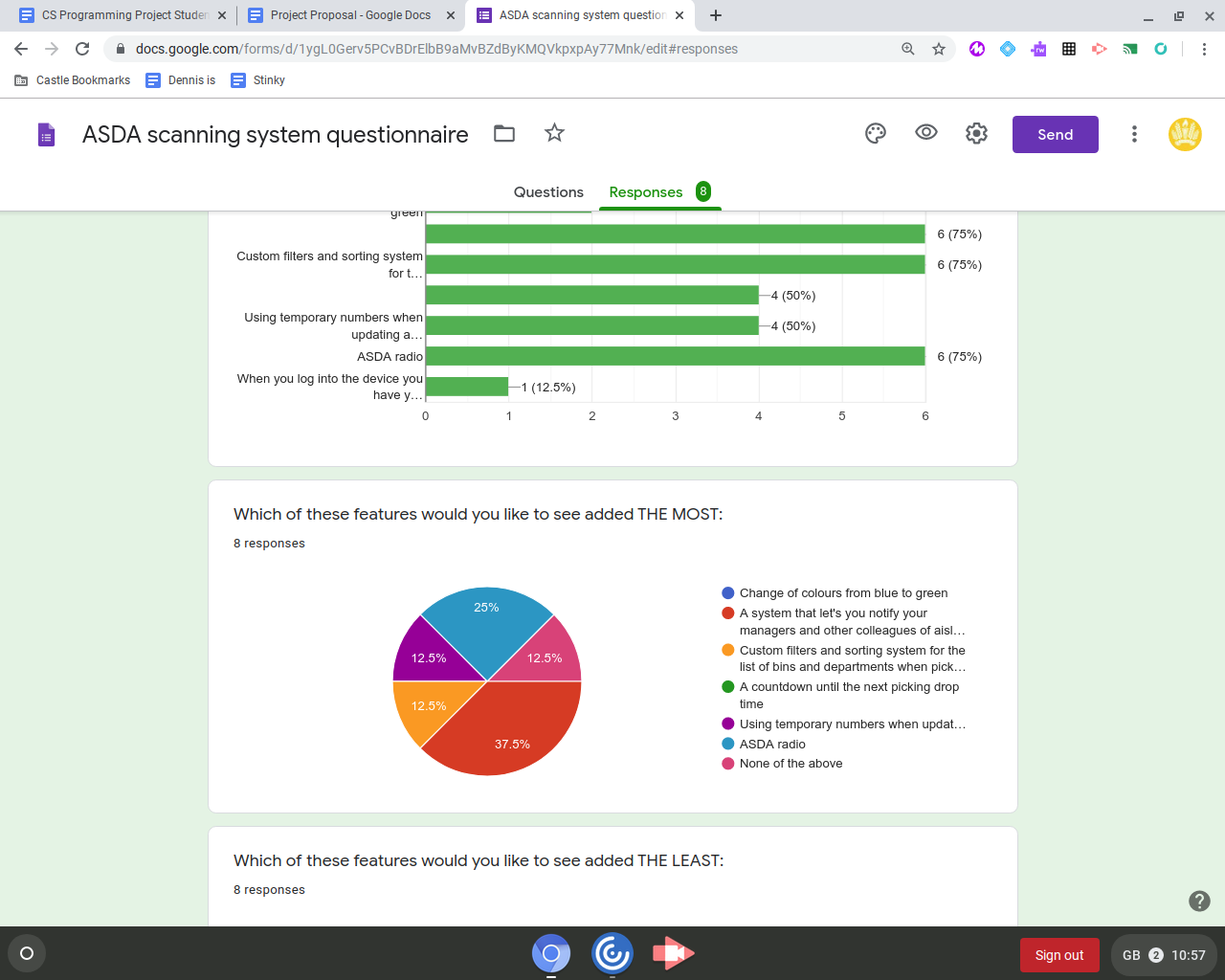
This table can show us that warehouse workers are more likely to want a change of colours, as they do not use the other apps and therefore need no differentiation between them. Likewise, ambient workers are more likely to want a countdown to the next drop of picks as shop floor workers are the only group of stakeholders to want this feature compared to warehouse workers who will not need it for their tasks. Both departments equally wanted the pinging system to notify managers, suggesting that the lack of any communication in ASDA is felt across all departments. An outlier is that function 3, while only benefitting ambient workers, was wanted unanimously by warehouse workers and only 50% by ambient. This suggests that it may be a good idea on paper but those with experience on the shop floor may find it unnecessary; so the function should be followed up after this initial meeting. Likewise, temporary numbers were a 50/50 split across all departments and individual departments, so should also be followed up by this initial meeting. ASDA radio was also unanimously requested by warehouse workers, who are the primary target audience, showing it’s popularity as a feature.

Question 2: Which of these features would you like to see added?



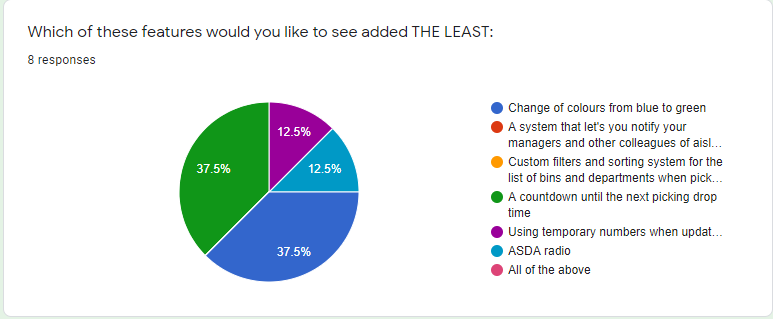
This question allowed me to see which features the employees would like implemented the most overall, across all departments. It shows 3 clear features that the workers felt were essential to add, custom filters and sorting system, asda radio and the pinging system (6 votes). Using this knowledge I can prioritize time into these 3 sections and ensure that the system is built with these functions in mind. Likewise, features with only 4 or fewer votes were dropped, proving to be divisive or to be counterintuitive to a system reliant on its ease of use. With limited time and a wide array of functions to code, I had to prioritise time into what mattered. Finally, only one additional feature was requested, a system that would enable the device to work like a phone with custom numbers. This idea was rejected however, as it is superseded by the pinging system, taking a longer time to convey ideas. In addition, the management were not comfortable with users appearing to be taking phone calls on their personal devices in front of customers, so it’s usability would be restricted and work against the user.

Question 3: Which of these features would you like to see added THE MOST?

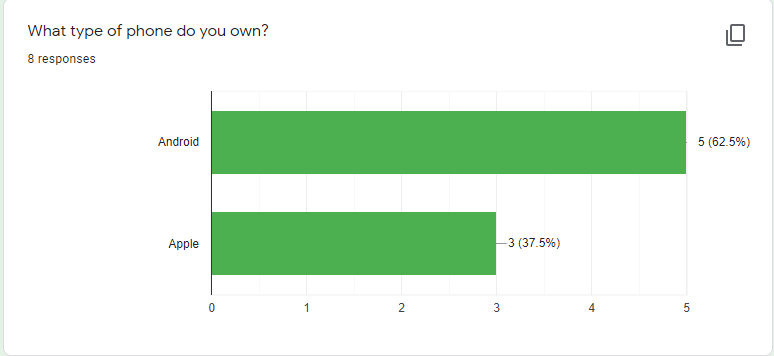


This question better allowed me to decide which features were considered essential and by which departments, as well as allowing me to cut features that were not needed by any department a significant amount. For example, nobody asked for the change of colours from blue to green or for a countdown timer until the next pick drops, allowing for justification of why they were dropped. These results show that the pinging system was the most highly requested feature to be added, followed by ASDA radio. As the pinging system is a communication feature, this question reflects what was already shown in my research, a clear lack of communication that is noticed heavily by staff. Some features that were not highly requested, such as custom filters or temporary numbers, were followed up after the survey. One response also said to add none of them the most, yet filled features that they wanted to see added, showing they had no preference rather than wanting none of them.

Question 4: Which of these features would you like to see added THE LEAST?

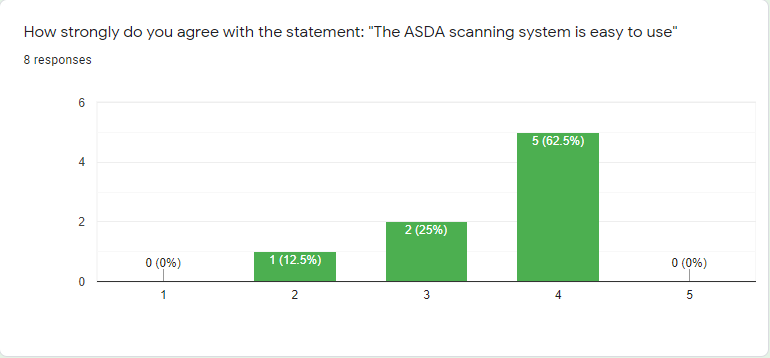


This question better allowed me to decide which features people didn’t want or wanted the least. Unlike the 3rd question, this showed 2 clear features that the stakeholders didn’t want and so these features were dropped after receiving this feedback, being the countdown and a change of colours. 2 other features with one vote each were followed up on.

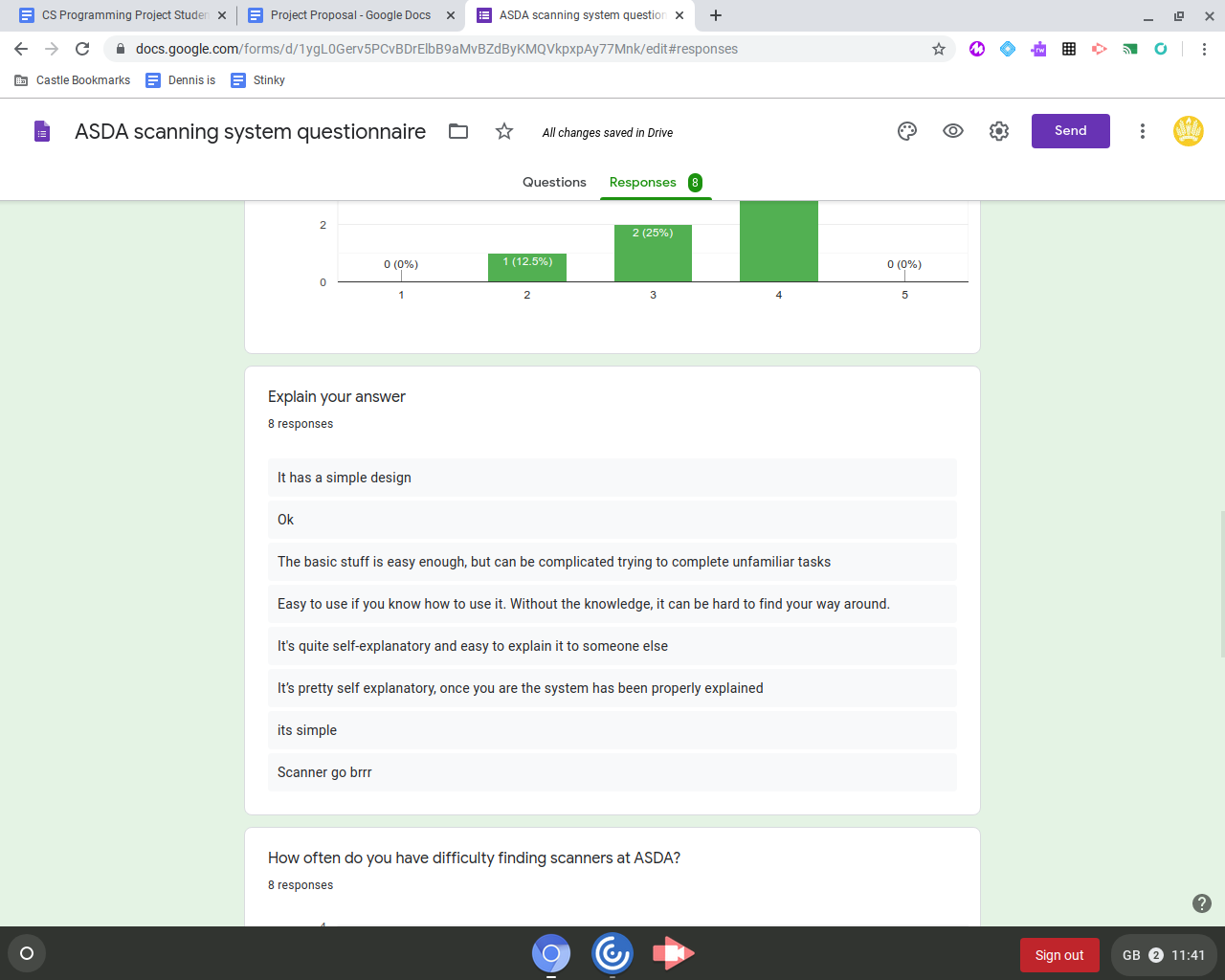
Question 5: What type of phone do you own?

This question allowed me to understand if I had to port the software to IOS as well as android, however with a 60% response rate of android I feel that it will remain the main OS and the only OS. Whereas IOS devices can be expensive if it was the main device, android devices are relatively inexpensive and readily available; stores could even use their old devices as substitutes. Therefore, Android will remain the primary OS.

Question 6: How strongly do you agree with the statement: "The ASDA scanning system is easy to use" ?

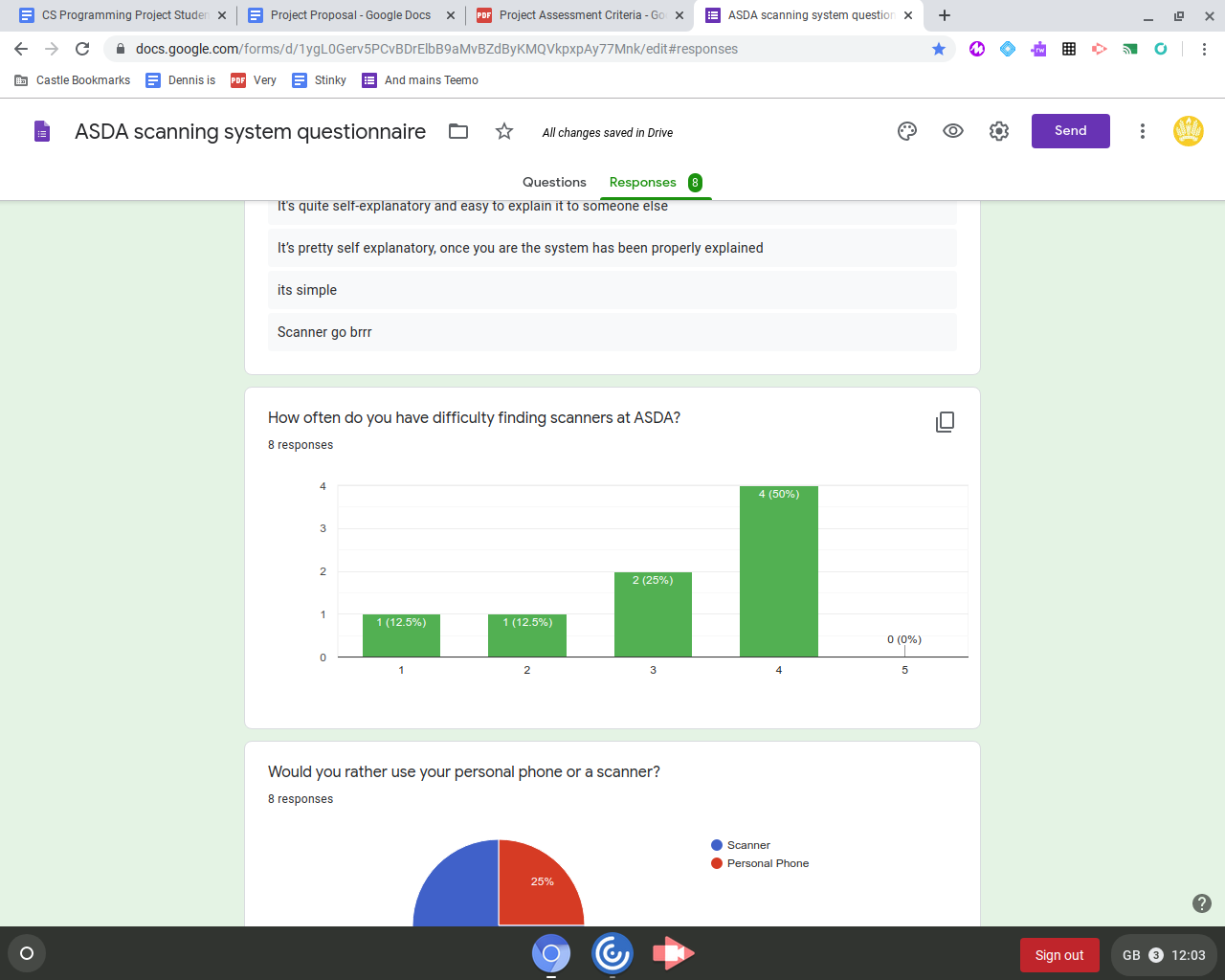


This question allowed me to better focus my success criteria for the software development, specifically on creating GUI. This feedback shows that most people feel the software is already simple to use, so may be a good model to base my own software off. However, 2 votes were neutral and 1 was negative of the system, showing that there are still improvements that can be made. For one of the most important success criteria on my list, obtaining software of this calibre is key. I will use the current system as a framework for my new system, as it will already be familiar to the user. However, with the addition of clearer buttons using icons rather than just 3 vertical dots, and more menus to confirm the choices of the user, I can make the device easier to use for new workers.

Question 7: Explain your answer [to the previous question]

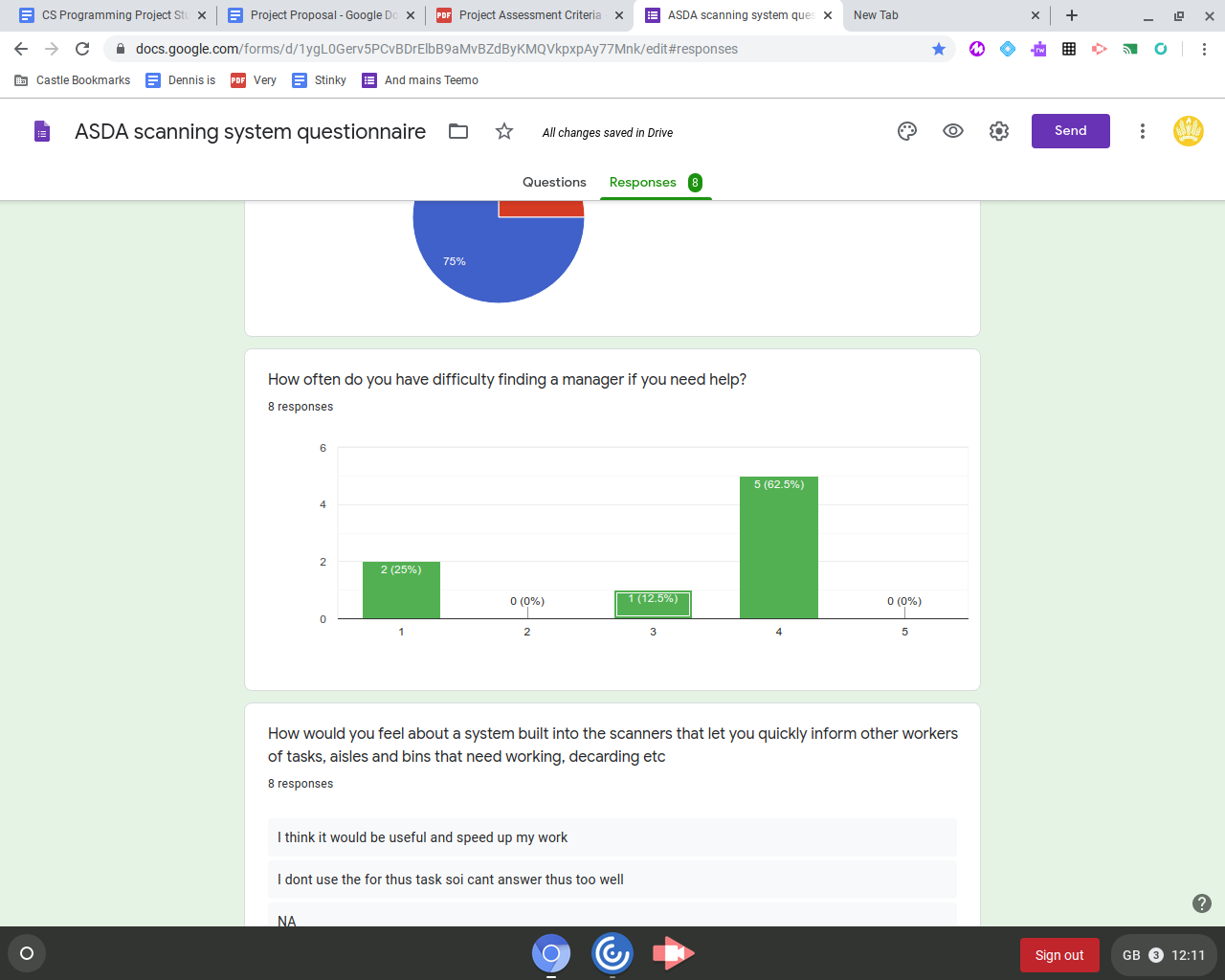
This question was designed in order to further understand the requirements of my stakeholders and their needs in a simple design. Not all feedback was helpful, such as “scanner go brrr”, showing the need for a more selective intake of feedback. 3 main themes can be derived from the feedback, the system has a simple design (mostly text based with images only as identifiers), the system is easy to understand and use, but also that the system can be hard to understand without knowledge of the device in unfamiliar tasks. This originally led me to believe that a “how to” menu or faq would be helpful, but as this would clutter the GUI. Instead, the system should be streamlined, so that one function or screen easily flows into the next with clear instructions that are jargon free.

Question 8: How often do you have difficulty finding scanners at ASDA?



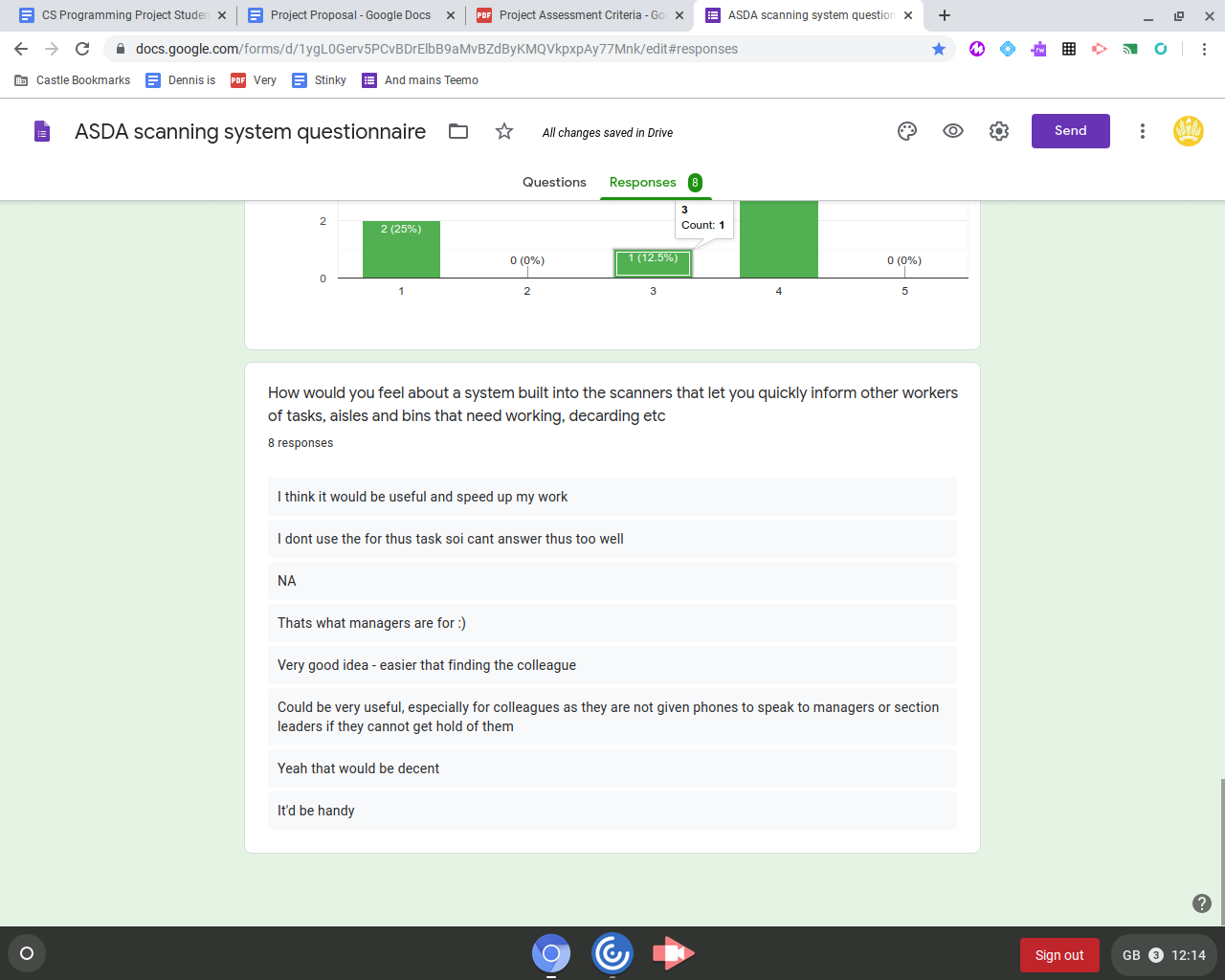
This question was asked in order to find out if the need for mobile portation of the software was needed, or if the load could be split between the two devices. With 50% of votes saying that they had problems with finding scanners and a further 25% saying that they were neither hard nor easy to find, this system was instantly recognised as outdated and that mobile portation was essential, whereas a continuation on scanners would not be in the interest of the stakeholders.

Question 9: How often do you have difficulty finding a manager?



This question was asked in order to understand the need for the pinging system and asda radio. With 62.5% of the respondents saying it was hard to find their manager, and 1 vote saying it was easy being from a manager, it is clear reinforcement that the communication aspects of the solution were essential.

Question 10: How would you feel about a system built into the scanners that let you quickly inform other workers of tasks, aisles and bins that need working, decarding etc?



In order to find out more information about the success criteria of the pinging system, this question was proposed to the respondents. However the feedback was not very useful for this function, instead only ushering support without any usable information, so next time a questionnaire is issued I will be sure to ask a question that is more suited to obtaining feedback.

Survey Follow Up:

* ASDA Radio - This feature was heavily requested by staff, warehouse workers in particular. They said that they would be interested in having the music out the front be played in the back of the house where it’s harder to hear. However, further research into this showed the function to be obsolete, as ASDA can already provide radios to warehouse workers if they wish to have ASDA radio as well as all the unique calls over the communication speakers (that cannot be found online); as well as ASDA having a policy that prohibits the use of earphones in their stores by employees. Thus, although a good idea on paper, in practical use this feature was dropped.
* Custom Filters - This feature was followed up, where most people did not see a use for it except in certain corner cases, “Most of the time I just look at everything and scroll through”. However, as it did not break any rules or impact the number of steps the user would have to take when not using this function, I have decided to follow through and add it, as it would still speed up work when it is needed.
* Temporary Numbers - This feature was found to be well liked among staff, as it would reduce the amount of false work or “ghost drops” when working the picks to be put on the shelf. However, the management said that the temporary numbers would impact their work in a negative way; the employees were often wrong when rectifying item numbers, “they could just find picks they don’t like and change the numbers for them” and that “Quite often the numbers on top are wrong as well, so by not having them drop down, there’s no like safety net and it’ll just stack up”. Hence, this feature was also dropped.

**Research:**

* **ASDA’s current hardware**:
  + They use the TC70/70X
    - TC70: Android 5.1 (Lollipop)
    - 1.7 GHz dual core processor
    - TC70x: Android 8.1 (Oreo)
    - 1.8 GHz hex core processor
* **ASDA’s software (Inventory Management):**
  + Current system functions:
    - Sales Floor:
      * Scan an item for look up history, location, general information, number on hand/on store/on backup
      * Counts: Check inconsistencies on the shop floor (if the number is negative, too much is on the shelf etc)
      * Manager Approval: Confirm a user submitted product volume or max shelf limit is correct and not the current stored
      * Zones: Configure modulars (places where product is stored), confirm the location of products without them
      * Waste Entry: Takes data off the database, stores it as waste instead, allows prices to be reduced on damaged goods
      * Shelf availability: Checks the shelf to see if there is empty space for new products to be put on display
    - Backup:
      * Binning: Adds products to a bin (back up where items are stored)
      * Auditing: Sets a bins contents to the products scanned
      * Picking: Compiles a list of products that can go out on the shop floor that must be picked off and put down before a set time
  + Strengths:
    - Easy to understand and use UI
    - Performs a wide variety of tasks, as shown above
    - Displays a wide variety of product information quickly and easily
    - Allows for modification to a central database without compromising the set up and layout of the normalisation.
    - The account protected software keeps it safe and secure from unwanted access and allows permissions to be distributed amongst management
    - Guns are usually held in one central location, meaning that when available they are easy to find
  + Weaknesses:
    - Slow, unoptimized and can easily break
    - Not every gun can scan the big barcodes
    - Competition to acquire the guns can lead to loss of time
    - Have to redo the same filters for picking section every time, leads to unnecessary repetition of basic tasks
    - Nothing can be sorted by number of products, intensity of aisles (% product to be worked), frequency of product dropping etc
    - Inability to be flexible with picks the algorithm detects
    - Over reliance on manager activity in order to correct mistakes, can lead to mistakes causing more damage than is necessary
    - Lack of communication options
    - Basic, blue and white UI is used for as the base rather than the white and green of asda branding
    - No confirmation screen after performing tasks
  + Solutions:
    - More robust code that is less prone to breaking under wifi lag or stress
    - Use of the camera on a mobile phone eliminates the inconsistent infra-red scanners
    - Use of personal android devices can mean that every employee should have a gun at hand, no time wasted on collecting. Also means there is no lack of available guns and that new guns to buy will be cheaper than dedicated devices.
    - Sortable filter lists that can be stored to the user removes the need to repeat tasks
    - Improved algorithms for sorting the collected data from the database of products
    - Using temporary numbers in conjunction with the stored numbers stops the over reliance on managers to make corrections but also maintains that no mistakes are made to the database
    - The addition of a “pinging system” where aisles, bins or jobs can be pinged to users increases workflow and productivity by allowing for faster and quicker communication
    - Changing the colours to green and white keeps with the asda branding and allows the business to maintain professional identity.
    - Shows a confirmation screen before each modification to the database. This can be toggled on or off to maximise efficiency.
  + Stakeholder Feedback on original system:
    - “The most important thing is that a new system must be able to always scan the large barcodes on the side of the box” - Phil, Product Line Manager
    - “The system we have now is simple to use though, new things could be complicated” - Matt, Ambient Colleague
    - “I hate having to filter the same aisles each time for picks. Why doesn’t it just save!” - Josh, Line Manager
    - “I wish the guns had, like, screens to confirm when you used them for stuff. It just kinda disappears and I don’t know if it happens or not.” - Dan, Warehouse Operator
  + Stakeholder Feedback on suggested improvements:
    - “[The pinging system] could be very useful, especially for colleagues as they are not given phones to speak to managers or section leaders if they cannot get hold of them ” - Rhys, Ambient Colleague
    - “The countdown seems unnecessary, it happens at the same time every day” - Dani, Health and Beauty Colleague
    - “Who even cares about the colours, I’d rather the more important features were put in quicker” - Chris, Deputy Store Manager

Further research was also done into the **League of Legends pinging system**: (Smart Ping, 2013)

* Pings are displayed in a radial format, where each of the 4 pings takes up a compass like position on the menu. This allows for each one to be selected as quick as each other and for them all to be seen at once.
* Each ping is carried out by selecting a location on the map or on the floor, then clicking and dragging to the ping type you want. These pings are then sent to all other players, where they make a distinct noise and last for a few seconds before disappearing.
* Each ping has a different colour, icon and sound; which allows users to quickly distinguish between them with only a quick glance. This allows them to remain focused on whatever task they are doing and not be distracted or disrupt their concentration.
* Each ping type is also distinct (“Enemy Missing, Danger, Help here, On my way”), so it conveys a wide amount of information without overlapping.

**Abstraction** was then carried out on this system to find out what made it useful and how to use this to implement my own system of pinging into ASDA:

* Each ping has a specific location
* Each ping has a colour and icon and is unique
* Each ping is easy to get to

Therefore my own system will reflect this style, where 5 presets, each with their own colour and symbol have been chosen:

* + - “Rumble” (Red, Trash bin)
    - “Drop and Audit” (Green, Upwards Arrow)
    - “Pick” (Blue, Downwards Arrow)
    - “Work here” (Purple, Rightwards Arrow)
    - “Meet here” (Yellow, Leftwards Arrow)

Each ping will be attributed to one specific location that the user can set when sending the ping. Pings will be easy to send, with each one requiring the use of only 3 inputs (recipient, location and type) and easy to reply to, as a reply function will also be built in.

**Proposed Additions for a Solution:**

Approved:

* The Pinging System - As the tied most requested feature as well as the most requested feature to be added by staff, this feature is the most important one to be added. The most time will be spent implementing this system, as it fulfills a role that is currently absent in the workplace and can even make the asda radio feature obsolete. It will require new architecture not seen in the current system, such as databases and menus, as well as new variables and SQL code, which will take a lot of time to design and implement.
* Custom Filters and Sorting System - With the tied most votes for requested feature and a vote for the most requested feature, this feature seems like a core component however not essential compared to other new features. Workers have expressed concern that this feature would complicate the application, so the design should be kept easy to use and it should work independently of other essential features.

Rejected:

* Changing the colours from blue to green - This modification has been abandoned completely, as the least requested addition with and the joint most votes for the one least needed in implementation. Upon reflection with staff, they said that it would be too similar to other apps on the device and that blue gave it a distinctive feel that enabled them to tell which app they were using quickly.
* Countdown to picks - Upon conversations with multiple stakeholders, this seemed only useful for half of the store and not everyone who would use it seemed to want to use this feature either. With the joint most votes as the least wanted addition, this feature will be dropped completely as it is redundant and creates unnecessary pressure on the workers.
* Using Temporary Numbers - A controversial addition, it was not very highly requested, but equally most asked for and least asked for. Upon reflection with the management, the response that asked for it the least, they said that the workload for them would increase massively compared to the workload for an employee. Upon this insight, the feature was rejected.
* Porting the application to scanners - Originally designed with backward compatibility on the original scanning devices, the new application will now only work on mobile devices because I do not have access to a barcode scanner. If it was to be ported, however, minimal changes would need to happen (use of a barcode scanner instead of the camera) and no variables or structural changes to the code would need to happen.
* ASDA radio - Originally proposed by a colleague, this feature would allow colleagues to listen to the music playing in store. Currently, it is the joint most requested feature to be added and was requested by multiple departments. However, due to details outlined previously, this feature will not be implemented due to time constraints, prohibition of earphones and the ability for this to already be implemented into the places that would use it.

Essential Features of the Design:

* Item look up - The core of the design, this is the most essential feature, as it enables the user to access the large database of items this software is based around. This displays all the information available about an item, as well as allowing the user to modify various attributes of the information.
* Picking - This feature is essential to users as it forms the basis of most of their work and is the main way they will modify the database. Picking is where the program will access a database of items that an external algorithm has decided upon, that need to be moved from bins and storage onto the shop floor.
* Pinging - This feature is an essential tool that will be added, as it is the only form of communication that workers have access to.
* Accounts - This is essential in order to facilitate saving settings and sending pings as well as prevent unwanted access to the device.

General Success Criteria Based on Feedback:

* The UI should be very simple to use. A survey will be conducted after the UI is designed and stakeholders will be asked to give feedback before adjustments are made.
* The UI should be quicker and more effective than the current system, providing all the same functionality as well as new functions and simplifying steps. This can be measured by timing tasks or counting the amount of buttons that need to be pressed and comparing results for the new and old system.
* The software should be able to correctly scan barcodes and communicate with an online server. This can be done by scanning a variety of barcodes and monitoring the inputs it receives. The server can be done by checking some generic responses of communication.
* All settings that are account specific should be saved correctly to the accounts, this means that they are account exclusive and shouldn’t show up on other user’s accounts. This can be checked by accessing accounts as different users and seeing what the UI allows us to use and see.
* All parts of the code should work independent of each other
  + The code should be robust, commented and adaptable

To check this, certain parts of the code could be turned off and then the system could be checked for functionality.

* The code should work on any android device. This could be checked by running the software on a variety of android devices and monitoring how fast it works.
* The code should be streamlined and one screen should lead smoothly into the next. This can be easily checked in testing by accessing different menus from different screens and recording the results.
* Labels should be jargon free, so that they do not complicate functions for the user. To check this, static outputs to the user should be stored and recorded and checked with stakeholders.

| **Requirement No** | | **Description** | **What it solves** | **Evidence** | **Justification** | **Success Criteria** |
| --- | --- | --- | --- | --- | --- | --- |
| 1 |  | Log in screen | Secure entry to the system  Stops unwanted entry or false identification | **If you’re reading this Sir, I have no idea what to put. :D** | You need to log in so that you can use the account specific features. Stops use of features unique to other accounts. |  |
|  | 1.1 | Username box | Requirement to enter a username |  | Ensures that the correct user is logging in. | * Username is rejected when it can’t be found * Should be labeled “Username” and be a text input field |
|  | 1.2 | Password box | Requirement to enter a password |  | Secure entry  using a login screen asking for  username and password  Stops unwanted access to accounts | * Password is disguised * Password is rejected when it doesn’t match * Should be labeled “Password” and be a text input field |
|  | 1.3 | Store Area | Requirement to select the correct store area |  | Ensures the user selects the correct store database. | * Should be a drop-down menu of 4 options * Should be rejected when it doesn’t match |
|  | 1.4 | Store Number | Requirement to enter correct store number |  | Stops users from accessing the wrong store  Speeds up database retrieval  Adds another layer of security | * Store is rejected when it doesn’t match * Should be labeled “Store Number” and be a text input field |
| 2 |  | Working barcode scanner | Allows the device to scan barcodes across functions for use in searching items |  | Allows the correct items to be used in the functions | * Should scan the correct barcode everytime * Should be able to be manually inputted or visually scanned |
| 3 |  | Pinging System | Allows staff to easily communicate with each other without spending time finding each other.  Currently a system exclusive to phones that would be made redundant |  | Speeds up the communication process.  Increases the efficiency of information between workers.  Allows workers to see who is logged in, who is working on what etc. increasing planning efficiency and communication. | * Databases should be fully normalised * Databases should not break when modified * Data should only be stored if it is relevant |
|  | 3.1 | Ping sending screen | Allows the user to send pings between each other. |  | Should only allow staff to send pings inside their own store. Should also show which staff members are in or not.  Should allow custom messages or pre sets. | * Should have a simple design * Should always send the message to the user in under 30 seconds (wifi dependent) * Should only list online users from their store * Should accurately update when loaded |
|  | 3.1.1 | Recipient input | Allows users to select who they wish to receive the ping. |  | Allows for the correct user to receive the ping and ensures the communication remains private. | * Should display online users in a drop down menu. * Should not allow the user to send messages to users that aren’t in the same store as them. |
|  | 3.1.2 | Ping Type Input | Sends the intention that the user has for the recipient |  | Allows for quicker communication by using presets | * Accepts all inputs * Should display options in a drop down menu. |
|  | 3.1.3 | Zone Location Input | Sends a location that the user decides on to the recipient |  | Should be a drop down menu  Allows the user to clarify their intentions even quicker. | * Accepts all inputs * Should display options in a drop down menu. |
|  | 3.2 | Ping receiving screen | Allows the user to receive pings. |  | This is needed in order for the user to access any pings they have been sent. | * Should always receive any sent messages in order of newest first * Should only load messages for the user * Should display the appropriate message if the user has no pings * Should allow the user to easily respond by taking them to the ping sending screen if they wish to reply. |
| 4 | 4.1 | Home Menu Screen | Having a contained and easy way of accessing the system |  | Keeps all the core functions in one place that is easy to access | * Should show all the menus available on the device * All the menus should be accessible * Should be easy and simple to understand |
|  | 4.2 | Side menu screen | A menu that can be accessed by swiping the left side of the screen right. This menu will display all the functions of the home menu in one list |  | Allows the user to access all the functions from any menu.  This cuts down on the amount of buttons that need to be pressed and simplifies the menu | * All the menus should be accessible and correctly labeled |
| 5 |  | Item Information Screen | Shows detailed product information:   1. UPC 2. Item Title 3. Item Number 4. Department 5. Size 6. Colour 7. Price 8. Delivery 9. Products in Backroom 10. Products on the shopfloor   Allows editing of the following information:   1. Products in Backroom 2. Products on the shopfloor |  | Allows the user to understand metrics of each item, such as where to find it or when the next delivery is.  Allows the user to easily update the database of each item.  Keeps the database accurate and up to date. | * Only the desired information should be editable * All the information displayed should be up to date * The correct item should be found every time |
| 6 |  | Picking Screen |  |  |  |  |
|  | 6.1 | Displayed list of departments -> bins -> items | A numerical (in ascending order of number from smallest to largest) list of each department that has picks. These can be clicked on to open the bins in that department, that can be clicked on to show the items. |  | An easy and understandable way for the user to see what work has to be done.  Each item in the hierarchy chain flows into the next one. | * All sections, and only those sections, of each category are shown if they have been chosen by the picking algorithm. * Each category is correctly linked * They are shown numerically * Buttons to move to the next screen are clear |
|  | 6.2 | Filterable bins that can work positively or negative on the list of departments | Won’t have to remake the same bin lists every time you use the software  Can easily swap between two separate lists  Can more easily see what needs to be done if sorted by department size |  | Will speed up time spent speculating totals for the work that needs to be done.  Managers can refer to specific lists rather than have to fill out the guns for each colleague. | * Should correctly save the filters to the user and only be accessible by that user * All filter lists should be accessible until they are deleted * Filter lists should correctly store any data inputted by the user for them * Filter lists should always be applied even if some of the departments specified aren’t on pick list |
|  | 6.3 | Can jump to the item information screen | Quickly links to section 5 (information on the item) |  | Allows the user to find the product easier if they’re having trouble.  Allows the user to correct mistakes on the item information screen quickly. | * Should correctly link back to the item information screen for that item * Should be able to return back to the pick list for the bin * Should be accessible for all items |
|  | 6.4 | Item scanning/ manual input to pick the item off the list | Allows the user to mark the pick as having been moved from the back up to the shop floor |  | Manual input ensures the user can always input the item if the barcode scanner is not working (due to lack of camera etc).  Visual input speeds up the process by removing long input times. | * Should scan the correct barcode everytime * Should be able to be manually inputted or visually scanned * Should work for all items and only modify the database for that item |

Limitations:

* New devices will have to be ordered in for stores that lack sufficient android devices for each staff member. This will be essential as it allows users to interact with the system and update work from home, as well as see their tasks before they come into work; this allows them to plan ahead.
* New usernames will have to be developed in the future, in order to make the ping system easier to understand and use. Without it, users may find it harder to search up other users on the system. This will have a financial effect in order to support this change.
* New users to the system may still require oversight or training from management, this will cost them time and money in order to improve the effectiveness of their workers.
* It lacks the ability to have live updates to the system as the servers will not be powerful enough to send out constant streams of data to multiple devices, this could be facilitated with more powerful servers but that would require expensive equipment.

**Design:**

**Decomposition:**

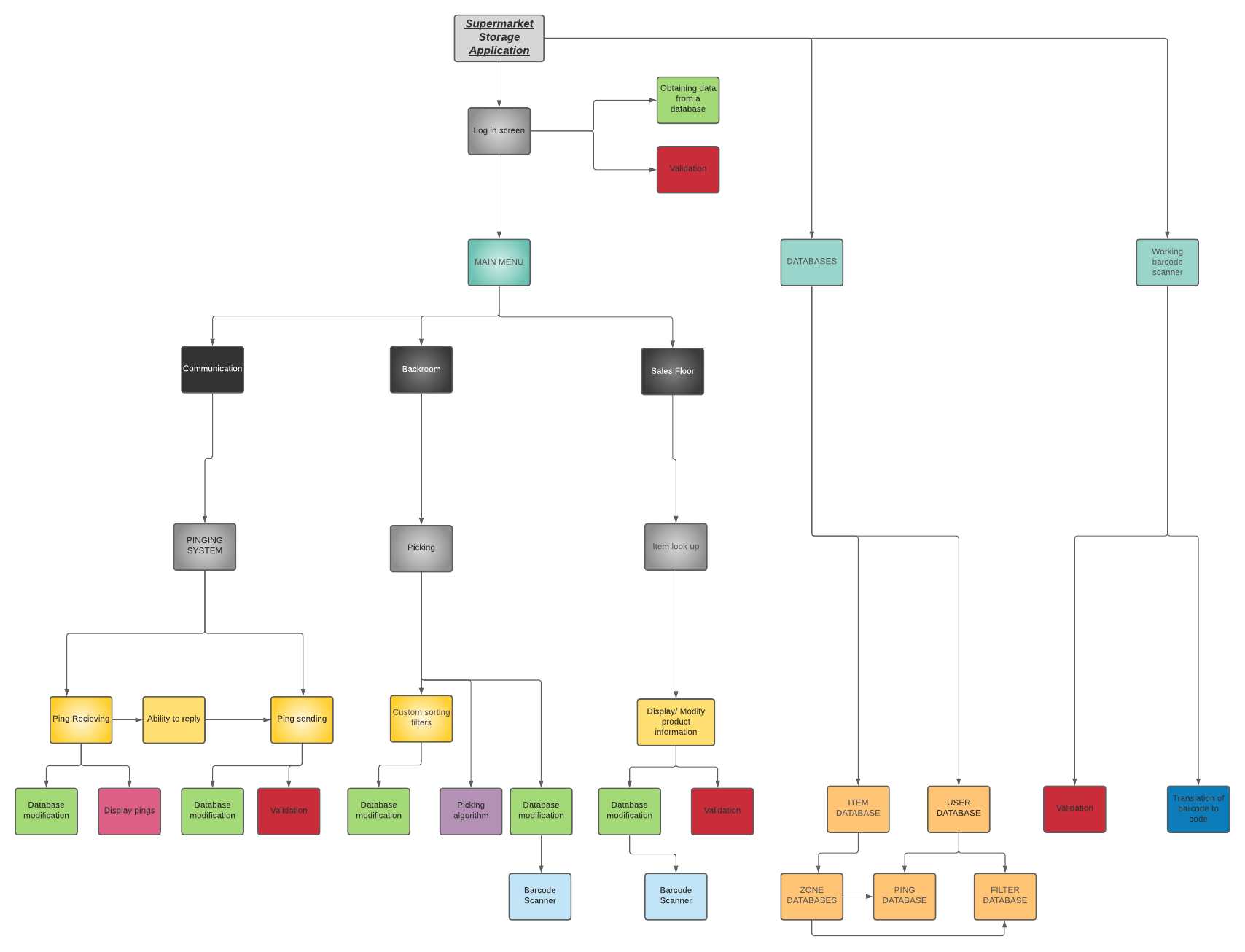
Decomposition has been used in order to break down the overall system into individual subproblems that can be managed and coded independently of each other. Each of these subproblems will be suitable for the solution by the computational solutions outlined in the analysis portion of this report. They have been broken down into sections pertaining to the role they play in the software; where each requirement of the user is a branch and those branches are then broken down into the functions they resolve. Fragments, individual screens in the system, can then be developed for each of these subproblems, where navigational fragments can be used for their parent problems. They can each be explained, acting as a verbal flowchart, for what each of them must do and then basic pseudocode can be written for each of these problems.

The chart below shows the structure of the solution to be solved, as a top down view of the application as a whole. The tasks of coding and development are generally labelled at the bottom, these are the largest programming tasks and algorithms that must be implemented.

The next layer up represents the main fragments, or groups of fragments, in the system. This represents the code that must be written in a more visual mindset and written with the stakeholders in mind more than any other layer. This is why stakeholder feedback will be gathered at this layer the most in order to ensure the system is as user friendly as possible.

The next layer up represents all the navigational fragments. These will be very easy to write, a simple button or two that links to the fragments they house. Most of these can even be copied and pasted, with only minimal changes to the button outputs and variable names but the overall structure will be the same. Therefore, these will not receive large design or algorithm portions in this report as they are essentially filler parts of the code and only need to operate on a basic level. So long as they have simple labels and navigate to the correct outputs, they will be considered successful according to the criteria.

The penultimate layer is an exception to the decomposition charts formula, as this will be a separate activity in the app. As it is a separate activity and not a separate fragment, it will be best to think of it as its own separate part in the app rather than think of it in conjunction with the other applications. This will be where the user logs into the system and so it’s own personal requirements of code are contained within this region.

Decomposition flowchart:[](https://lucid.app/documents/edit/847ef428-c515-4ef6-80c9-0b332361c6c3/0?callback=close&name=docs&callback_type=back&v=4164&s=854)

Colour Key:

* Solid Colours:
  + Light Grey: The application as a whole.
  + Grey: The main functions of the application
  + Dark Grey: The main sections of the application
  + Turquoise: The main branches that are umbrellas for sections
  + Green: Database manipulation or modification - accessing a database and changing parts of it. One universal database access code will be written for reading, modifying and deleting that can be modified per use of the code, with minimal changes.
  + Red: Validation of inputs from the user, making sure inputs are valid for the code and won’t invalidate the database.
  + Orange: Databases that will be used by the operator, these databases will need to be designed and normalised separately.
  + Light Blue: Denotes use of the barcode scanner in order to help with database modifications.
  + Purple, pink, blue: Custom algorithms that help the main functions they are under specifically
* Colour texture:
  + Radial colours: A menu that will need to be displayed, these will either be an activity, a fragment, or a series of fragments.
  + Solid colour: A general identifier of what type the functions below it are, that may need to be coded but will not need to be some displayable menu(s).

**Algorithms and Pseudocode:**

* Top Layer:

The SSA box represents the system as a whole and unifies the sections of the decomposition into one central location. One arrow goes to the first page that will open upon access to the system, while 2 other arrows branch off to the hardware and databases that it requires access to after login. This can be seen as the app’s icon on the home screen. Although it is not worth mentioning in the design, this will be changed to a recognisable icon in order to improve app visibility. No testing will be needed for this however, as the app will not run if this is wrong and as access to the actual icon used is not available, it will be a generic placeholder.

**Pseudocode:**

Android:icon = “drawable/icon”

(where the icon is storage as an image named “icon” in the drawable folder”)

* Second Layer:

This layer has one function that acts as a barrier to the application once opened, requiring you to log into the system in order to access it. This will be one of two activities in the system; in order to improve security in the software and prevent accidental access in code oversight. The grey box is radial, showing that it consists of a main menu which will have to be designed with a layout resource file and has green and red sub boxes, representing the validation of the data inputted and how it will need to access the user database in order to cross check usernames and passwords inputted with those stored on the system.

To log in, the user must input their correct username, password and store number as well as select the correct spinner option for their store area. After they have done this, the system can check their inputs against the database and use that to validate. If it is correct, it can then pass their data to the rest of the system and the user can continue to the main activity of the system.

**Basic Algorithmic Approach:**

1. Instantiate Variables and the spinner
2. Create and watch a button that begins the login process
3. Pass off the variables to the subroutine that handles the login process when this is clicked
4. Check to see if the user has correctly inputted the details, then pass back either a fail or success
5. Display the correct message based on passing or failing
6. Save the results of the login and move to the next activity if it was successful (log in)
7. Allow the user to retry on a fail

**General Pseudocode:**

**An important thing to note is that this pseudocode will be far more in depth than the following algorithms, in order to fully showcase all the parts of production. Due to the project being very large, this is not possible for all algorithms and would result in lots of replication of parts such as instantiation of variables and creation of spinners. After this, a focus on unique functions will be applied rather than a 100% walkthrough of each kotlin file. Pseudocode will also not be needed for the visual layout, as kotlin uses a drag and drop system for its assets and objects.**

Login Activity:

setContentView(activityLogin)

val username = findViewById<EditText>(username)

val password = findViewById<EditText>(password)

val spinner = findViewById<Spinner>(storeArea)

val storeNumber = findViewById<EditText>(storeNumber)

val login = findViewById<Button>(login)

val storeAreas = resources.getStringArray(storeAreas)

This code will instantiate the correct fragment layout for the login screen, as well as connect all the input fields to their correct variable counterparts. The final line connects to the array of strings that act as options for the spinner. These are set up first so they can be used later to pass their data into the subroutine for login attempts.

if spinner != null {

val adapter = ArrayAdapter(this, simple\_spinner\_item, storeAreas)

spinner.adapter = adapter

spinner.onItemSelected{

if (parent != null) {

storeArea = parent.getItemAtPosition(position)

}

}

}

The first thing that will need to be coded will be the creation of the spinner, so that all the input fields are ready at the start of the code to be operated on by the user and validated against the database. Here, the first line will check to ensure the spinner has not already been set up. It will then create an adapter for the spinner, where it will give the correct context, type of spinner it needs and what data it needs it to hold. Simple spinner item was used as it was the easiest to develop and would be the easiest for the end user to understand. It will then assign this new adapter to the spinner for the spinner to use. This will have then created the spinner and the focus of the code will shift towards extracting the data from the spinner to be used in the code. The code will check if the user has input a choice (although this is impossible as there will be a default selection) and then it will obtain the item selected (getItemAtPosition) and set it as their option each time they select a choice, or upon the first creation where it will set itself to the default (first) option.

if error != null {

loginFailed(error)

}

if success != null {

loginSuccess(success)

}

Selection has been used here to branch to the correct outcomes based on the results of the users attempt to log in. If the user has failed to log in, the correct subroutines will be ran and an error message will be displayed. However, if the user has succeeded in logging in, then that subroutine will be run and the program can continue down the branches shown in the decomposition diagram. These two algorithms are shown below:

FUNCTION loginSuccess(mode)

welcome = getString(welcome)

userName = GLOBALusername

storeNumber = GLOBALstorenumber

Toast(applicationContext, "$welcome $userName", Toast.LENGTH\_LONG)

val intent = Intent(this@LoginActivity, MainMenu::class.java)

startActivity(intent)

ENDFUNCTION

FUNCTION loginFailed(@StringRes errorString) {

Toast(applicationContext, errorString, Toast.*LENGTH\_SHORT*)

ENDFUNCTION

These two functions handle a successful login and a failed login respectively. The success algorithm obtains the welcome message from the strings file, then sets the correct username and store number to be global so that they can then be used throughout the program. Then it displays the correct message to the user in the form of a toast (a message that appears at the bottom of the screen). It passes through the correct context, message and the time duration of the toast. This is all that the failed attempt will do, as there is no correct data to save and no entry to the next section, and will have a short time when compared to the successful attempt as it has a shorter message. Then, the successful attempt will create an intent, where it designates it’s destination, current location and then moves between activities.

login.setOnClickListener {

login(username, password, storeArea, storeNumber)

}

This is the section of code that will run whenever the user clicks on the login button. It will pass through all the current values of the log in inputs to the subroutine that handles login requests so that it can begin the login process.

FUNCTION login(username, password, storeArea, storeNumber): {

val apiInterface = LAPIClient.client.create(LAPIInterface::class.java)

val call = apiInterface.loginAccounts()

call.enqueue(

object : Callback<LoginAccounts> {

FUCNTION onResponse(

call: retrofit2.Call<LoginAccounts>,

response: Response<LoginAccounts>

) {

val text = response.body()

**THIS CHUNK OF PSEUDOCODE ACTS AS A BLUEPRINT FOR ALL RETRIEVAL FROM THE DATABASE AND WILL NOT BE REPEATED IN LATER ALGORITHMS, INSTEAD REFERRED TO WITH THE LINE “TEXT = DATABASE.RESPONSE”.**

**WHERE SPECIFIC PARTS OF DATA IS NEEDED (NOT JUST THE WHOLE DATABASE), THE CALLBACK<VARIABLE> WILL BE CHANGED TO BE OF A TYPE THAT REFLECTS THIS CHANGE.**

for (i IN RANGE 0 ... text.values.size) {

val temp : Users = text.values[i]

if (username == temp.username) {

if (password == temp.password) {

if (storeNumber == temp.storeNumber) {

fail = 0

}

}

}

}

val finalUser = LoggedInUser(username, storeNumber)

return when (fail){

1 -> Result.Error(

IOException(

"Error logging in",

LogInException("Log in error")

)

)

else -> status == “online”

Result.Success(finalUser)

}

ENDFUNCTION

This function would make use of the LAPI client in order to create an online database with google sheets. The program would call to the online database and the database would return all of the users on the database. The program would then shift through the users and if it found a user that was correct and matched with the information it held, then it would turn off the fail checker, otherwise the fail check would be left on. At the end of the list, the final user would be stored (the users log in details) and it would either return a failed result or a successful result based on if it was successful. If it was successful, the status of the user would be changed to “online” and then the user would be logged in.

In the future, I would like to improve this software by using better search algorithms than the linear search which has a time complexity of O(N^2) and so is not suitable to work with the larger databases of 100s of employees. I may also like to do this search on the server, rather than send all the confidential information to the user which could be intercepted and then data mined for information. However, these improvements cannot be made as I cannot be certain the database it connects to will be sorted and sorting the list into alphabetical order would mitigate the time saved. In addition, doing all the checks server side is beyond the scope of this project, as I am unfamiliar with the language needed and the information is not being stored on a private server. I would use SQL on a private server if I had the option in the future and so is an improvement that I would recommend if this was to be used in the future.

* Third Layer:

This layer of code contains the 3 main branches that problems can be divided into. One branch contains all the code that must be written in order to carry out the functions from a software perspective; one branch contains all the databases that must be written in order to store the data from a data management perspective and a final branch contains all the code that must be written from a hardware perspective.

The software branch is known as the “Main Menu” because it is radial, meaning it will house a directory of all the functions and acts as a screen on the device that will display each task. This will be the second activity, as fragments will only be used after this point in order to keep the system within one navigation file and make navigation between functions on the system easier than having to create new activities and pass intents between each one.

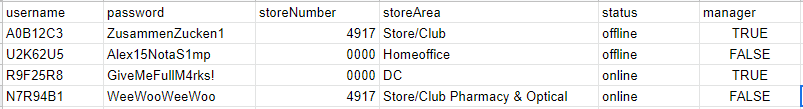
The other two branches will not require screens in order to function, as they will not be directly accessed by the user, so are plain.

Although there are lots of menus with only one function, and it would be faster to remove these menus and jump straight to their functions, it has been kept this way so that in the future more functions can be added. By keeping the menu set up, new functions can be developed independent of the other functions and then added by attaching it to a button on one of the menus. This will speed up future development time and allow for cohesion throughout the application to menus with multiple functions.

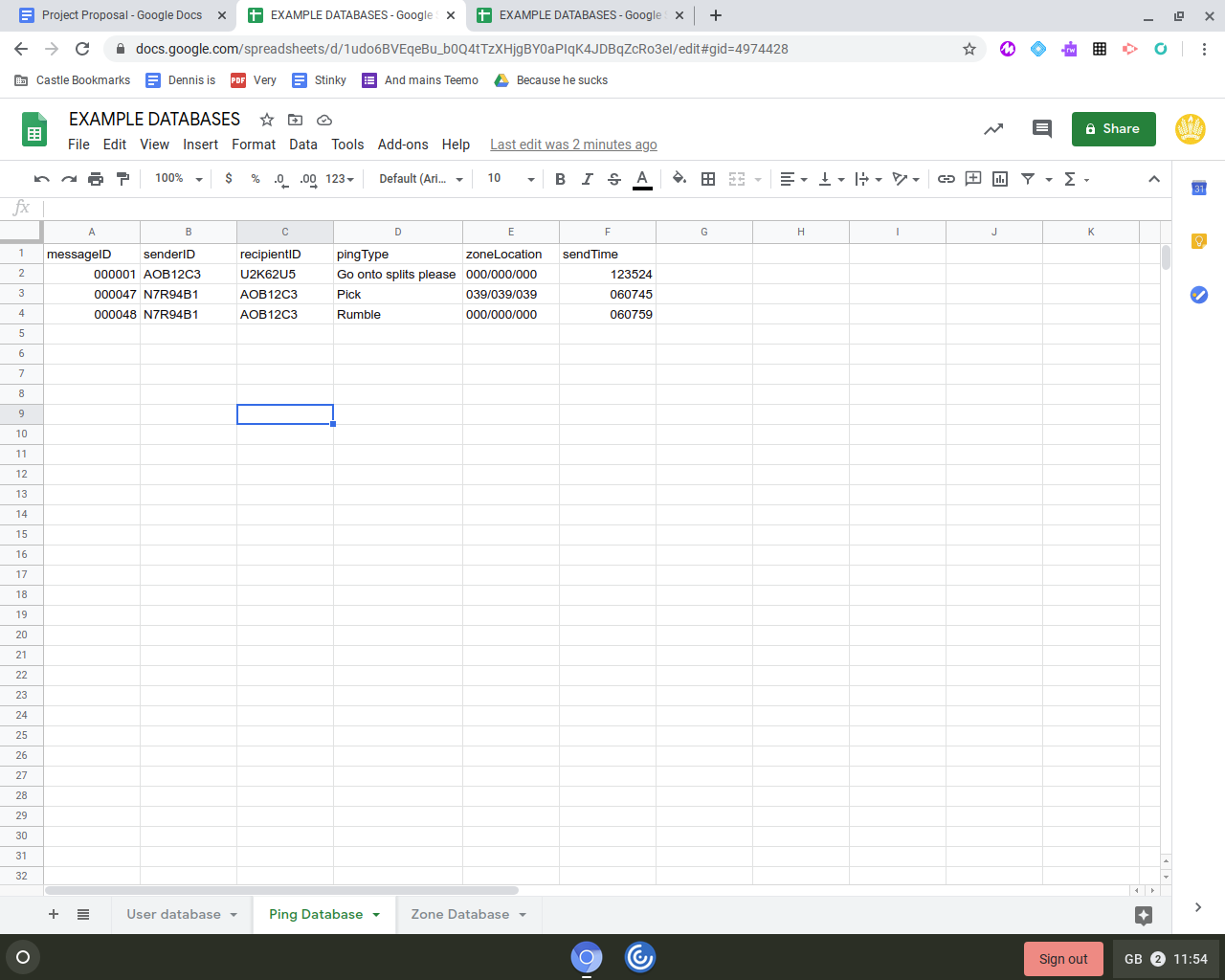
* Database Branch:

This branch contains four databases and a database collection: a database that stores information specific to the user; a database that stores all the information pertaining to items and a sub database collection for shop floor zones, departments and modulars; a database that stores all the pings while they are being sent between users and finally a database that stores the unique filter lists per user. The two main databases are the item information database and the user database, as the user database also acts as the parent for the ping database and filter database and the item database acts as a parent for the zone databases. The child databases were implemented as a way of simplifying data storage and allowing other applications or functions to use the databases without all the extra content that would be created by two singular databases. Google Sheets will be used to store the spreadsheets, as it is a free software that works well with Kotlin and is always online, with the example databases useful in testing shown below each outline. No validation will need to be carried out on the databases except when uploading, with the assumption that the database is already fully complete and managed by an exterior force.

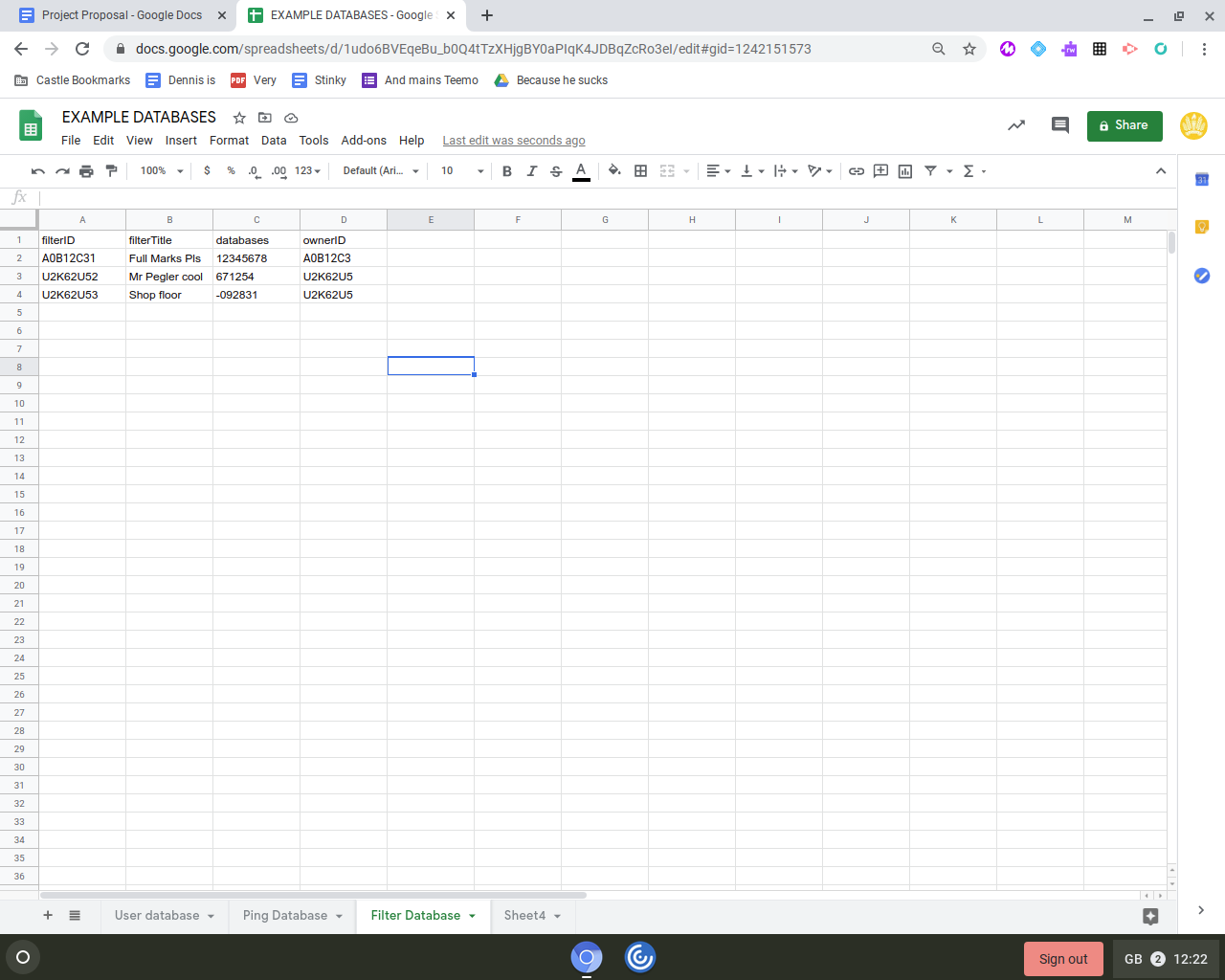
* The **user** database will have to store each user with a unique primary key of username (username), that it can use to link to the pinging and filter database. The user database will store each user's username (username), password (password), store number (storeNumber) and work category (storeArea). It will also keep note of if that user is currently online (status) and if they are a manager and thus have access to the manager specific functions (manager).
  + username - This constant will be a 7 character string that is unique to the user, such as “A0B12C3”.
  + password - This constant will be a string between 6 and 14 letters long that will be enquired for when the user makes a login attempt, such as “BenIsCool18!”. Although it can be changed via walmartone, for the purposes of the device it will be said to be constant and unchangeable.
  + storeNumber - This integer constant will be a number of 4 characters that will be enquired for when the user makes a login attempt, such as “1234”.
  + storeArea - This string constant will be 1 of 4 predefined strings will be enquired for when the user makes a login attempt, they are “Store/Club”, “Store/Club Pharmacy & Optical”, “Homeoffice” and “DC”. The last of the two do not require a store number to be stored if they are present so the store number “0000” will be used in order to avoid null errors.
  + status - This string variable will be updated to “Online” when the user makes a correct log in attempt and then turning “Offline” when the user makes a successful log out, however the user may also choose to set themselves to “Busy” or “On Break” while they are online which will reset when they log off.
  + manager - This boolean constant will be “True” if the user is a manager or “False” if they are not.



* The **ping** database will have unique message IDs (messageID) for each sent message, as well as storing the sender (senderID), recipient (recipientID), ping type (pingType), ping location (zoneLocation) and time of sending (sendTime).
  + messageID - This 6 digit constant integer will be unique for all pings on the list, taking the form of 1 + the most recent ping ID number. If no pings are found then the ping will be the number 0000000001; if this number should be 7 digits, then the system will check the database for an empty message ID, choosing the lowest available slot if one is found, else rejecting the ping
  + senderID - This 7 character constant string variable will contain the “username” of the sender, that is used to create the “messageID” as well as the find the “recipientID” if the recipient chooses to respond.
  + recipientID - This 7 character constant string variable will contain the “username” of the intended recipient, so that when the code checks to see if the user has any pings, it will check their “username” against this variable, then take all the messages where it matches this variable. When a user logs off the system, this will be checked to see if there are any remaining pings that the system will then delete.
  + pingType - This undetermined length constant string variable can be set to be one of 5 preset options, in order to speed up the process or can be a custom string of information from the user. Presets are set out in the research section of this report.
  + zoneLocation - This optional 11 length letter constant string variable will link to the zone databases, allowing the user to search for one from the list that is then attached to the ping. This can also be null if there is no location needed (“000/000/000”) or a custom location.
  + sendTime - This constant 6 character integer will store the time in the 24 hour format so that the user can know when they were asked this task and thus which is the most recent request if more than one was made. (12:25:35 would be 122535)



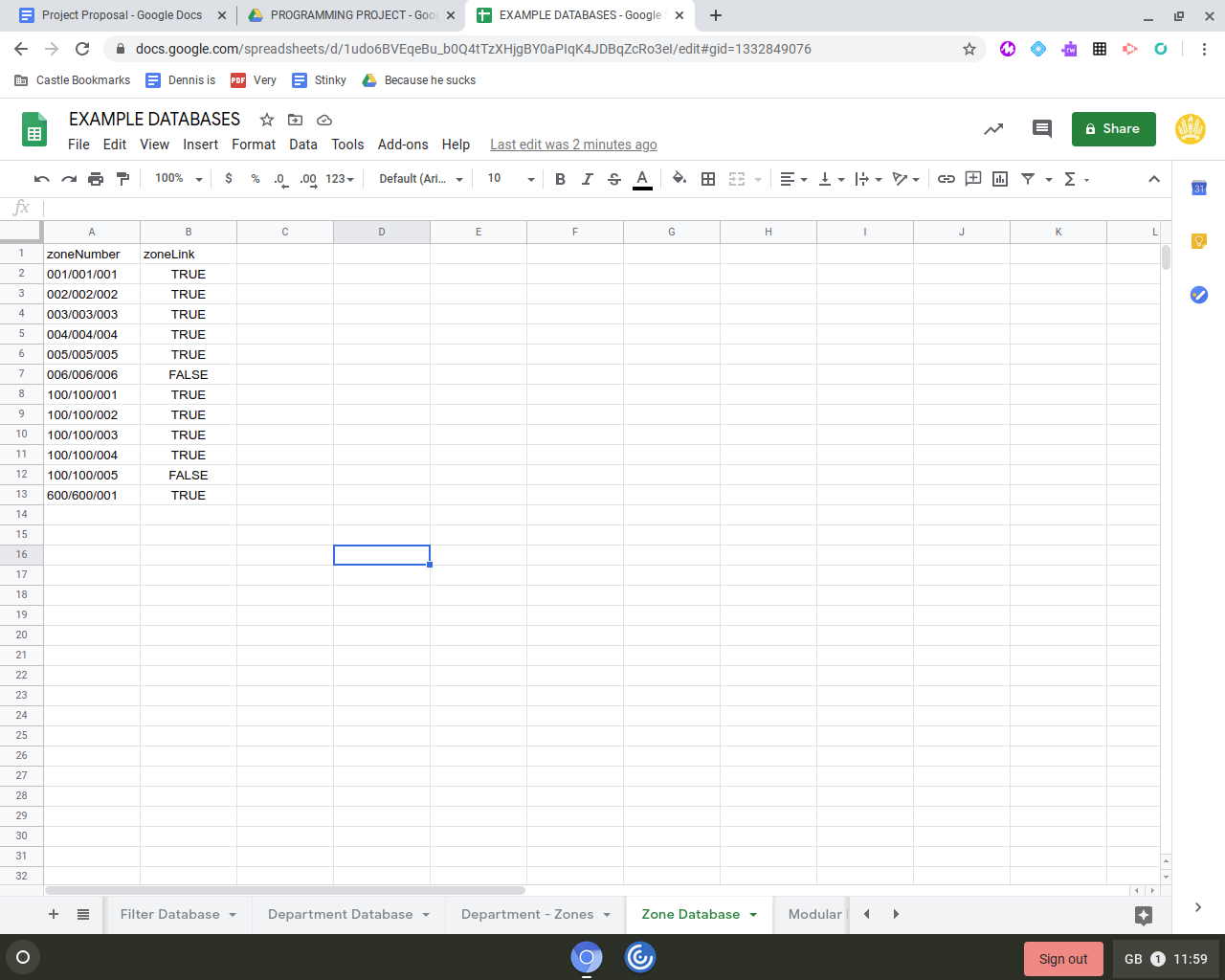
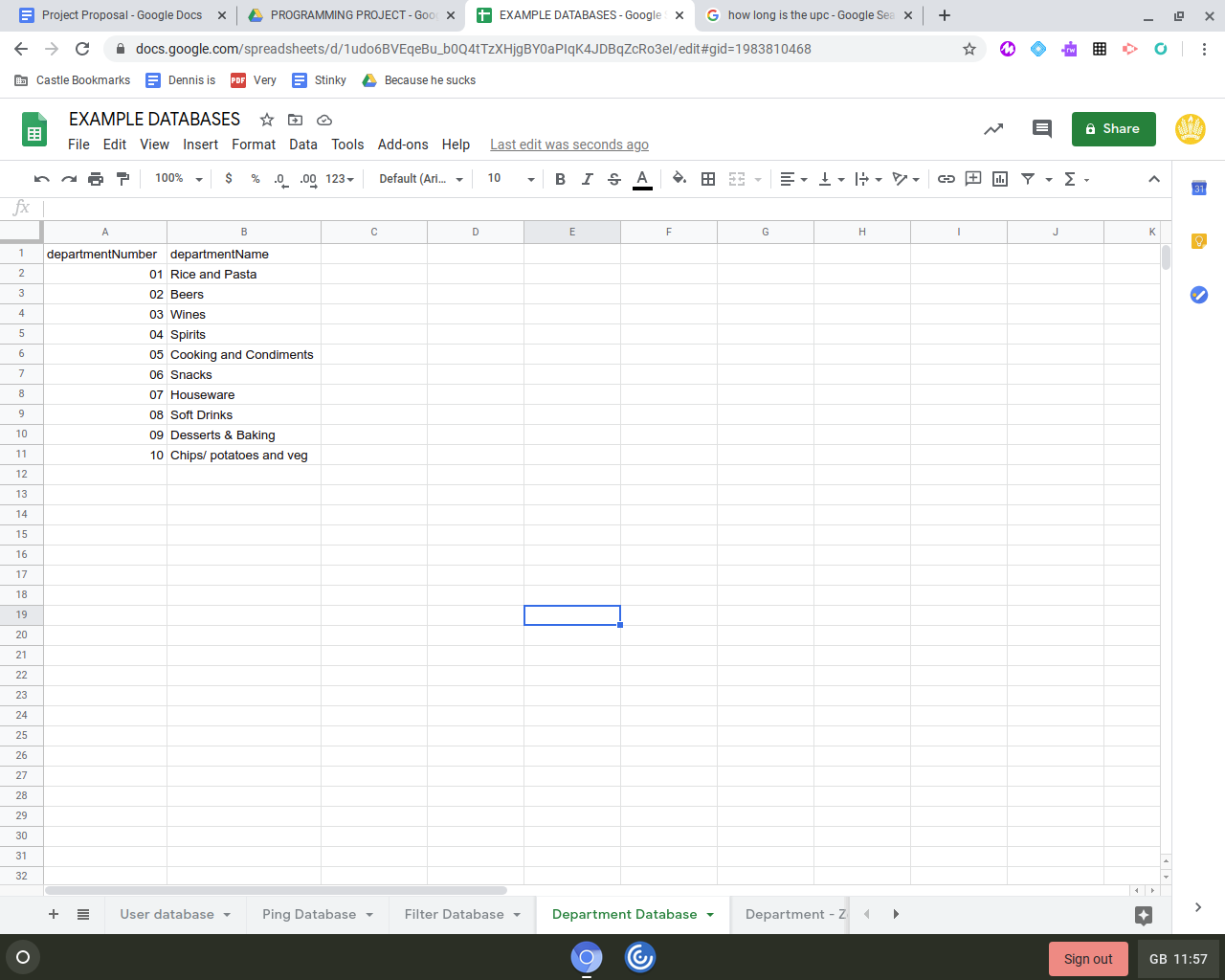
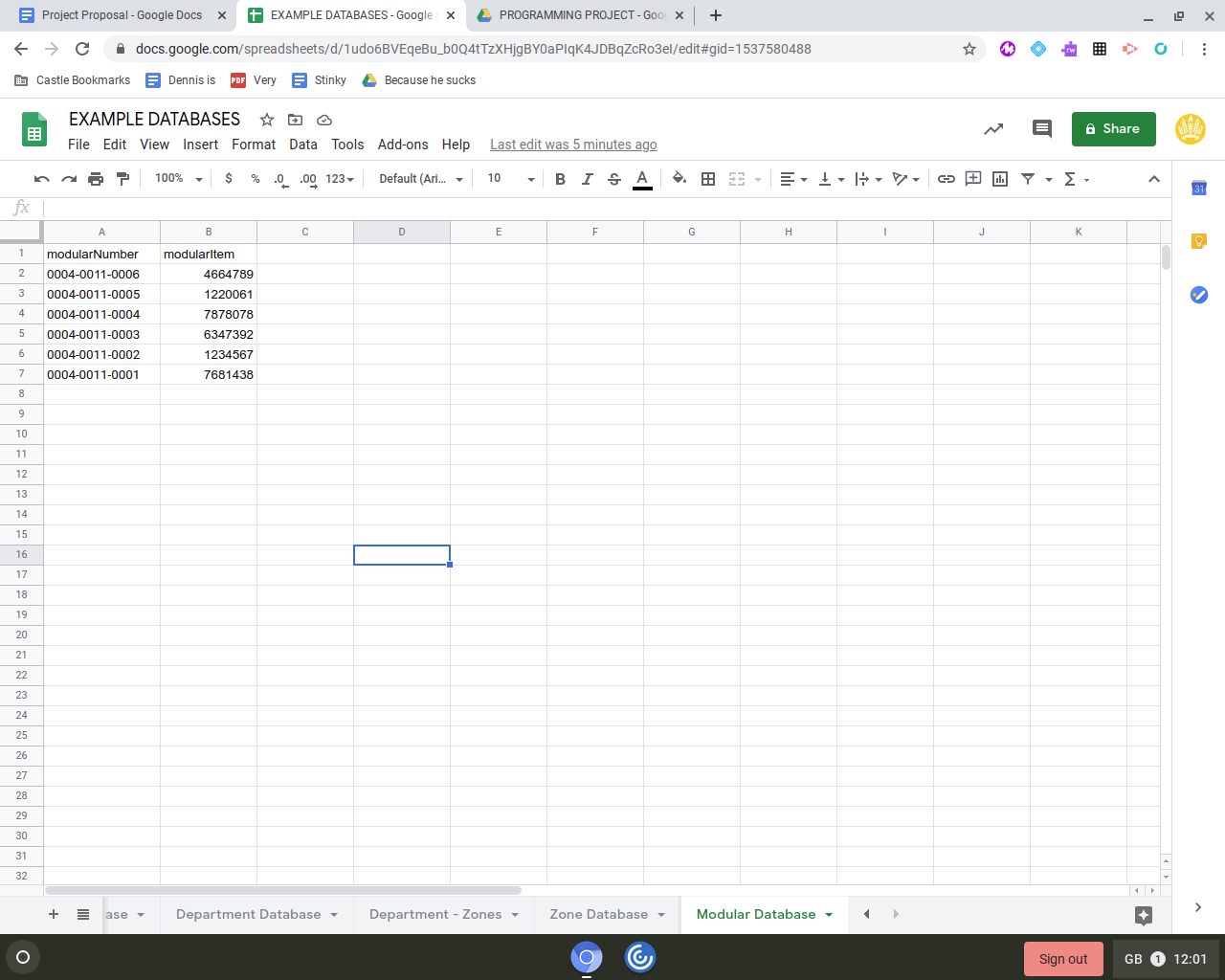
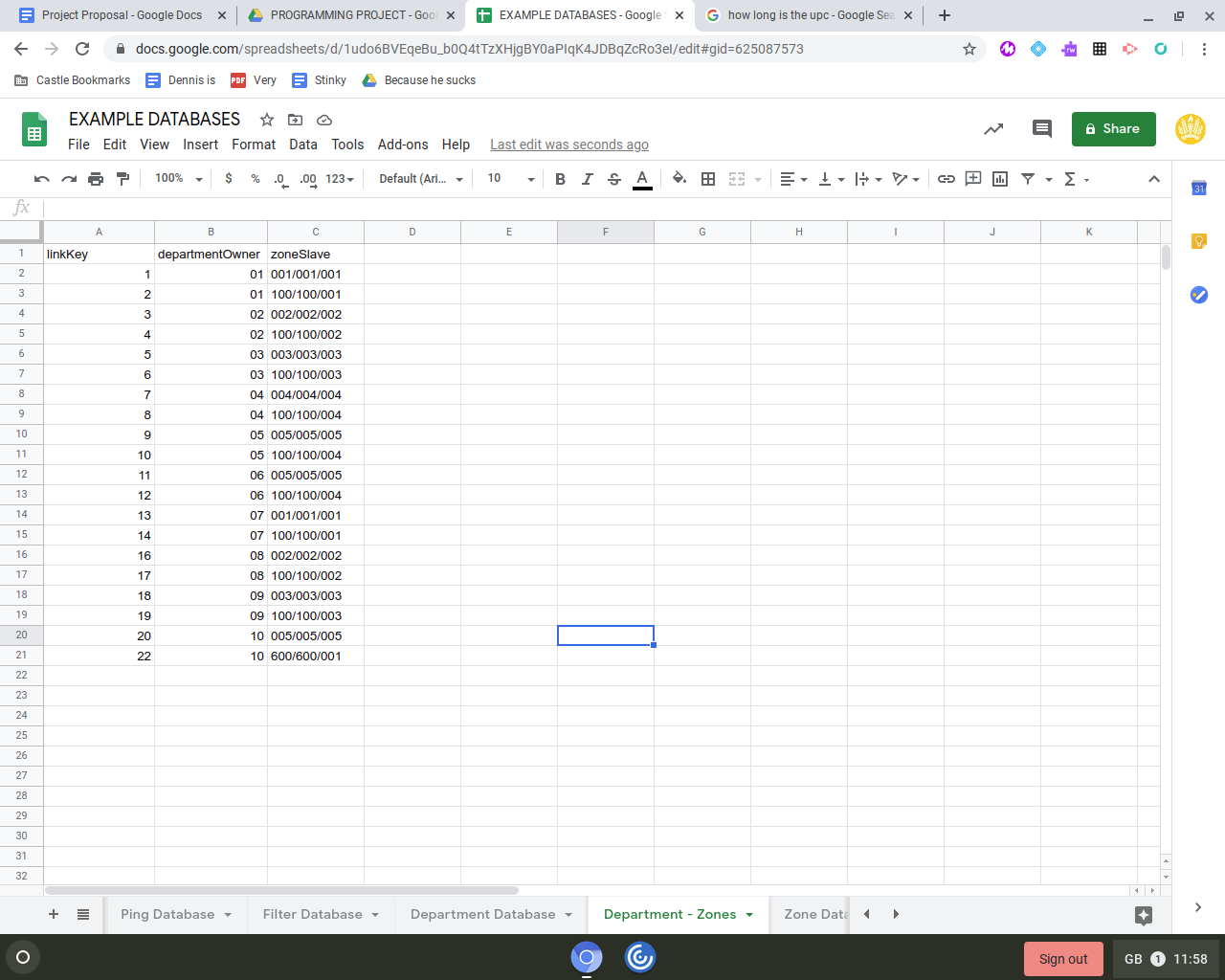
* The **filter** database will have unique filter list IDs (filterID) for each stored list, that stores a filter list title (filterTitle), the departments that it is composed of (databases) and a link will be made to the user (ownerID) that owns the preset. These will be permanent on the database unless manually removed. The user may also choose for these to be inclusive or exclusive filters, meaning that they will either remove the selected departments or include the selected departments and nothing else.
  + filterID - A constant string of unannounced length, composed of the ownerID + n where n is 1 + the final digit of their last list; unless they have created no lists in which case it will be 1. For example if A0B12C3 creates a list it will be stored as A0B12C31, then their second will be A0B12C32 and the third will be A0B12C33. If they then delete the first list, the fourth will be called A0B12C34 as they have still created 4 lists in total.
  + filterTitle - A constant string of characters between 4 and 15 characters in length, this will be the displayed title of the users filters rather than the filterID, in order for the user to more easily identify and access the correct filter list they need, e.g. “My Full Marks”
  + databases - A constant integer of undefined length, each 2 digit department will be chained one behind the other to form a list of the highlighted departments. If it is a negative integer, this will be an exclusive filter list and a positive denotes an inclusive filter list, “6874” will only display departments 68 and 74 whereas “-0612” will exclude departments 6 and 12.
  + ownerID - A constant string of length 7 that links the specific filter to the filter list, e.g. “A0B12C3”



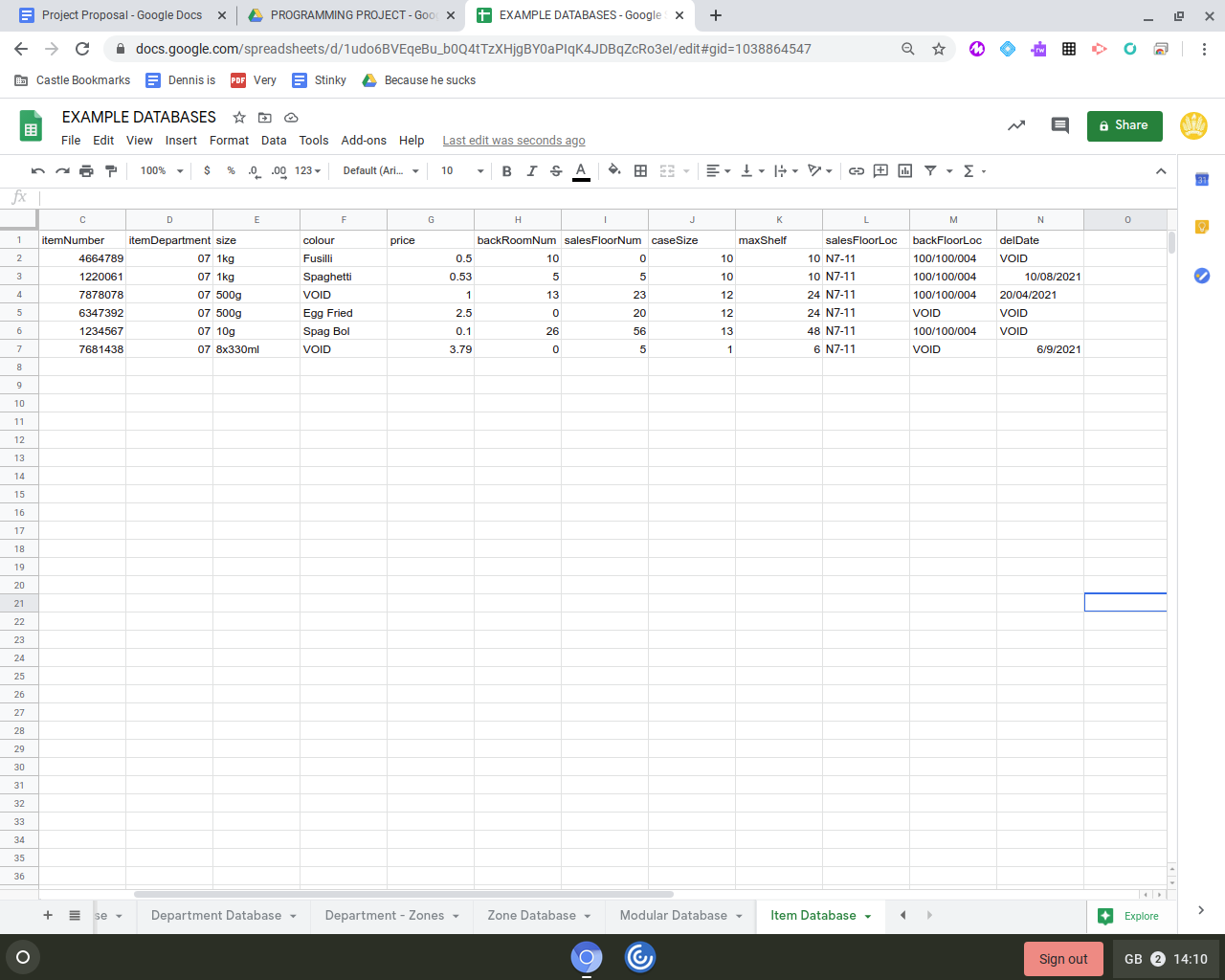
* The **zone** databases contain all the possible locations a product can take, consisting of 3 locations of increasing abundance (Departments, Zone, Modular) as shown in the pyramid below. The relationship diagram on the right showcases how each one relates to the one below, where it will contain many of the layers below but the zones can relate back to many above. Because of this, 1 database will be made showcasing the department to zone relationship; the zone to modular relationship will not be needed as they use their owners name in their variable denotation and 3 additional databases for further information on these sections in order to convey whether they are linked etc.



* No information will be stored about items here, instead links will be made with the database and it will keep track of if it is linked to anything or not.
  + departmentNumber - A two digit fixed number that stores the number that the department is associated with, such as “68”
  + departmentName - A unlimited length string that describes the department that the number is associated with e.g. “Cooking and Condiments”
  + linkKey - An unlimited integer key that acts as a unique identifier for each relationship, consisting of the number before it +1
  + departmentOwner - A two digit fixed number that denotes the department the elected zone links back to e.g. “21”
  + zoneSlave - An 11 character fixed string that denotes the zone that is the subordinate of the owner e.g. “100/100/003”, linking back to the zone number
  + zoneNumber - An 11 character fixed string that locates a bin or aisle e.g. “0039/0039/0039”
  + zoneLink - A boolean variable that checks to see if the zone has a master
  + modularNumber - A 13 character fixed string that denotes the aisle number, bay number and product-shelf number. For example, the 13 items in bay 26 on aisle 46 will read “0046-0026-0013”.
  + modularItem - This 7 digit integer will link back to the item number of the item stored in the modular. If no item is stored, the item number will be void (all 0s).



* The **item** database will store all the data needed about the item, taken straight from the existing system. This includes:
  + The UPC - the Universal Product Code represented by the barcode stored as a 12 digit integer in “upc”.
  + Item Title - the unlimited length string “itemTitle” will store the worded description of the item
  + Item number - A 7 digit number used locally in supermarket chains in order to save space and as a shorthand for the UPC. “itemNumber”
  + Department - A 2 digit integer used in order to identify the department that the item belongs to “itemDepartment”
  + Size - An unlimited character string used to represent how much one item will weigh, for example a 30pack of 330ml cans will be “30x330ml” but a 1kg bag will be “1kg”. It is denoted by “size”
  + Colour - An optional unlimited character string used to represent the flavour or colour of an item, if there are two items of the same size; denoted by “colour”. If there is no colour, “VOID” will be used.
  + Price - A mandatory double that stores the value of 1 item, denoted by “price”. The pound sign is not stored here.
  + Number of products in the back room, shop floor and on hand (total) - Two separate unlimited size integers, where the “shopFloorNum” and the “backRoomNum” can be updated manually and the total number is worked out on the device.
  + Case size - An unlimited sized integer that represents the amount of product per unit case, for example there are 30 walkers bags in a box, so the “caseSize” would be “30”.
  + Max Shelf - The maximum amount of products that can go on all of it’s shelves, represented by an unlimited integer maxShelf.
  + Sales floor location(s) - The modular numbers will be used here, if the product is located in a modular beginning 00xy, then the number will take the form of “Nxy”. Likewise, a product beginning in 01xy will take the form “Pxy” and a product with 02xy will take the form “Cxy”, for produce and checkouts respectively. The next part will consist of the bay it is on, for example if it is on aisle 48, bay 12 then it will be N48-12. If there is more than one modular on a different bay and/or aisle, they will be separated by commas. The variable is assigned to “salesFloorLoc”.
  + Back floor location(s) - The zone numbers will be listed here, if the product is present in any of them. If it is present in more than one location, they will be separated by commas. The variable is assigned to “backFloorLoc” and by default is “VOID”.
  + Sales/ Activity - These two parts of the original database have been removed in this updated version. They would require two more separate databases as well as more advanced coding to track purchase history, order history and account access that I do not have the time to implement. Feedback from the stakeholders showed that this information was only ever used by sales managers outside of the app, so this was an unnecessary feature that slowed down loading time. However, the delivery date will be implemented still as it is useful when dealing with customers needing missing items but the delivery database will not be coded as the software does not interact with it. It will be stored under “delDate” and will be void if there is no incoming delivery.



* Hardware Branch:

This branch contains only the barcode scanner, as it is the only hardware centric code that needs to be written. It has a unique blue sub box for the code that will detect the barcode on the screen and translate it to a usable item barcode. It also features a red sub box for validation of the barcode it scans, in order to differentiate between the zone barcodes and the item barcodes. It can then return what the barcode belongs to and then the software side will use this in order to accommodate for internal verification so that database integrity is maintained.

* Software Branch:

The software branch is then decomposed into 3 sections based on their overall functionality to the user, some will be used for communication between the workers (communication), some will be used to perform general tasks (shopfloor) and the final branch (backroom) will contain 3 functions that work in tandem with each other in order to change the stock size on the item database. These were grafted straight from the pre existing software, so should be intuitive to the users. Each of these main sections (other than communication) are radial, showing that they will be screens on the application, these will be menus where each function can be selected for usage. On each main screen there will be a side tab that links to each other category, with drop down functionality for the functions in that category.

* **Communication:**

This branch contains the main menu of the ping system, represented by the radial function box as well as the menus that comprise each of its components. The main menu for this will be accessible via the side tab. It will display a button for sending pings and a button for checking the ping inbox.

* + Ping sending - This will be a menu that takes in the desired user, either by selecting them from the list of workers assigned to their store; sorted by offline and then alphabetically. If the user is not assigned to a store, there will instead be no list. This will be the recipientID as shown above. It will then display a dropdown menu underneath, these will act as the pingType. Next, the user will select a zone by selecting from a dropdown menu; this will automatically set to the void zone “000/000/000” so if none is entered it will speed up sending. This will be the zoneLocation. Finally, the user will press send, at which point the user’s username will be taken for the senderID. It will also check the database to find out the messageID as shown above, and then store in the database. After this, the user will be taken back to the main menu for the pinging system. In the future, I would like to add more to this feature, such as custom pings or adding multi-user selection.

**Algorithm:**

1. Retrieve the username, store number and status of users from the database.
2. Create a list of recipient usernames with matching store number and “online” status.
3. Sort this list alphabetically (very small list so short sorting times)
4. Set up the spinners for username and ping type selection.
5. Retrieve the list of zones from the database, then set up the spinner for zone selection.
6. Set up a listener for the sending button then send the ping to the database once it has been clicked

**Unique Pseudocode:**

companion object {

recipient

pingType

location

}

The file will use a companion object throughout it’s runtime, this allows it to share the variables of user inputs throughout the functions it contains.

val recipients = ArrayList<String>()

val response = database.response

for(element in response){

if(element.status == "online"){

if(element.storeNumber == storeNumbe*r*){

recipients.add(element.username)

}

}}

recipients.sort()

This section of pseudocode will create an array list for the list of possible recipients that can then be fed into a spinner adapter for the recipient spinner. It takes an empty list and the list of all possible candidates, then fills the empty list with candidates that are both “online” and have the correct store number. Once this is done, it will sort the small list into alphabetical order so that it is easier for the user to find their item. This is a good usability feature to add, as it will allow for the user to use their own searching methods, or methods such as the binary search on the data items and decrease the time spent searching for users.

var locations = database.response

locations.add("000/000/000")

locations.sort()

This section of pseudocode is much more simplistic and is used to gather the list of locations for it’s spinner adapter. It is sorted for the same reasons as stated in the previous chunk of pseudocode, however the zone 000/000/000 has been added so that in instances where a location should not be stated, it can be left null.

sendButton.onClickListener{

val currentTime = Date()

val finalPing = Ping(userName,recipient,pingType,location,currentTime)

database.add(finalPing)

Toast(activity, "Ping has been sent to: $recipient", Toast.*LENGTH\_LONG*)

navigate(communication)

}

The above pseudocode showcases what will happen when the final button is pressed and the ping is ready to be sent to the database. Firstly, the current date is saved into a variable so that it can be attached to the final ping. Then, the final ping is compiled, adding together all of the variables of the ping into one object of the Ping class. This is so that it can be set up as one packet of data to the online database. Confirmation of this is then shown to the user, a toast is made displaying who the ping was sent to and that it has been successfully sent. Finally, the system reverts back to the navigation menu so that the user may then navigate to a new function or may check their inbox, construct a new ping etc. Although not required, this is more of a usability feature by providing further visual confirmation that the ping has been sent off.

* + Ping receiving - Upon accessing the main communication menu, the system will check the ping database for any pings that match the recipientID of the user. If none are found, the button will be disabled and this fragment will be inaccessible. If any pings are found, upon pressing the button they will be taken to a menu listing all of their pings. This will consist of a recycler list that displays the ping type’s image on the right, the sender ID at the top middle, the written ping type in the bottom middle, the send time at the top right and the zone location in the bottom right, as an example shows below. This will be stored as ping\_item.xml in the layout resources and will be inflated in the view for each ping item. Two buttons will also be shown at the bottom, one that allows the user to “reply” which just navigates them to the ping sending fragment. Another button will be used to clear the ping list of all pings, which will clear the database of all pings and then refresh the ping list manually to show the updates to the user.

**Algorithm:**

1. Retrieve all the pings from the database
2. Filter them out until only the pings for the selected user are left
3. Sort these pings so that the newest pings are at the top of the list
4. Pass the pings to the recycler view to be displayed
5. Display the pings
6. Set up the reply button to send the user to the ping sending screen
7. Set up the clear button to clear the pings on the database for that user



* **Shopfloor:**
  + Item look up - Firstly, the user will be taken to the generic barcode screen in order to input the UPC of the item they wish to look up. If the UPC number is on the database for items, the device will then navigate to the item information screen, setting the global upc to the upc returned.

**Algorithm:**

1. Access the barcode scanner so that a UPC number may be obtained, passing the destination as the item info fragment
2. Set up the text views for each attributes of the item and buttons to scan a new item, add items to the shop floor or add items to the backroom
3. Set the values of the text fields to the attributes of the item found in the database
4. Update the buttons whenever an item is added or taken away so that the minimum number in a zone is 0 and the maximum is the total items available and the buttons will be disabled to enforce this.

* **Backroom:**
  + Picking - The first stage of this will be to show the user the list of departments along with how many items in each department must be picked. The user can then select any of these departments to view the zones that these items are kept in. The user can then select any of these zones to view the unique items within that zone. Scanning or entering the UPC number of any of these items will then “scan” the item out of the pick list and change the number of items on the shop floor and backroom.
    - Picking (Filters) - This is a component of the department screen that will allow the users to only see specific departments based on the needs of the users. These can be saved as preset filters unique to the user, that can then be accessed and used repeatedly, saving time against the current system. To make a filter, the user must specify if the filter is a negative filter (none of the selected) or a positive filter (only the selected) and choose the departments to go on the filter. They will then save this and a spinner will be set up to allow the user to choose between their filters or make a new one.
    - Picking (Sorting) - This is a spinner that will be displayed on the department screen that will allow the user to filter their departments by pre set filters. This could be numerically, alphabetically or by size both ascending and descending.

**Algorithm (Picking):**

1. The department menu sets up the sorting spinner (described below), the filter button (described below) and the recycler view for the departments list.
2. When a department on the recycler view is clicked, it navigates to the bin list of that department.
3. The bin list then sets up a recycler view for the list of bins, passing through all bins with items and the amount of items in each bin.
4. When a bin on the recycler view is clicked, it navigates to the final list for that bin, where all the items inside the bin are listed.
5. When an item on the recycler view is clicked, it navigates to the item info fragment, passing through that UPC.
6. A floating action button is set up to access the barcode scanner. Any UPC number the barcode scanner returns that has a matching item on the bin list will remove that item from the list of items. If there are no more items in the list, it will instead return to the previous bin list.

**Algorithm (Filters):**

1. Set up a button that takes the user to a new filter screen
2. Set up a spinner on this new screen to handle filter selection
3. Allow the user to select which departments they want **FROM A RECYCLER LIST OF ALL DEPARTMENTS** when creating a new filter
4. Allow the user to select if the new filter will be positive or negative
5. Allow the user to save the filter then apply the filter and take the user back to the department screen

**Algorithm (Sorting):**

1. Set up a spinner with the 6 inputs
2. Apply the default sort to the list
3. Change the sorting of the list on a new spinner selection
4. Update the recycler view with the new sort

* **Barcode Scanner:**
  + UPC numbers can either be inputted manually or scanned using the camera on the device. There will need to be checks for the UPC, if it is 12 characters, if it is numbers and if it is on the database for items. The barcode scanner will only need to be checked against the database as it can be configured to only accept 12 character barcodes often used on items.
  + In addition, destinations will have to be attached when this fragment is called, so that the fragment knows where to pass on the UPC numbers that it returns. This is done by storing a global destination variable “destination” inside a companion object attached to the class. The fragment also attaches the UPC number it calls to a global variable “upc” inside this companion object in order to enable it as accessible to all other fragments that need it.

**Algorithm (Camera):**

1. A scan button is set up which is watched
2. When this button is clicked, the camera is opened and the camera scans for a barcode
3. When a barcode is detected, the camera closes and returns the value that it scanned.
4. If the camera is closes and no value is found, a message is output that states the scan was cancelled

**Pseudocode:**

ONCLICKLISTENER(scanButton)

val intentIntegrator = IntentIntegrator.forSupportFragment(this)

intentIntegrator.beepEnabled(FALSE)

intentIntegrator.cameraId(0)

intentIntegrator.prompt("SCAN")

intentIntegrator.barcodeImageEnabled(FALSE)

intentIntegrator.initiateScan()

END ONCLICKLISTENER

This is the pseudocode for the on click listener of the scanning button. It sets up an intentIntegrator, where it essentially details how it wants to operate the scanning button. It sets the camera to not beep when it is used (this is currently the case and is highly annoying), specifies the camera Id that it wants to use, selects a message to display to the user to help them understand what to do. It also sets the barcode image enabled to be false, so that it does not return a path of the image of the barcode but instead the UPC number. Finally, it can initiate the scanning process.

FUNCTION onActivityResult(requestCode, resultCode, data) {

val result = activityResult(requestCode, resultCode, data)

if result != null {

if result.contents == null {

Toast(context, "Scan Cancelled", LENGTH\_LONG)

} else {

Toast(context, "Scan results: " + result.contents,LENGTH\_LONG)

depart(result.contents)

}

}

ENDFUNCTION

This function will run once the UPC number has been collected (there has been a result of the barcode activity). Here, the result is stored into the value of result, where a value has been used to ensure the UPC number is not overwritten in testing. So long as there is a result, the next part of the function will run. This is to prevent the function from breaking when there is no result and so it validates the hardware and not the user input. If the user has canceled the scan, by use of the back button or otherwise, then the program will output a message to say that the scan was cancelled and to affirm the user’s decision. If the user has scanned the barcode however, and a result is obtained, then it will output to the user what UPC number they scanned and the depart function will run. Although not shown, the depart function will merely look at the destination variable and then choose which fragment to send the UPC number to.

Although the scan results are not always needed by the user, it is far more useful when testing the software and ensuring that the process works and the right barcode has been scanned and is left in as it does not obscure any other process and can be a helpful usability feature in ensuring the user knows exactly what inputs they have fed to the device. This can be helpful in instances where the wrong inputs have been given.

**Key Variables:**

| **Variable Name** | **Variable Type** | **Function** | **Location** |
| --- | --- | --- | --- |
| destination | String, non null | Stores the destination of the barcode fragment to take once it has retrieved the barcode it was trying to find. Kept in a companion object so that it can be modified by any fragment | ui > barcode > BarcodeScanner.kt  Companion object of BarcodeScanner |
| upc | String, non null | Stores the UPC retrieved from the barcode scanner, kept in a companion object so that any fragment may use it. | ui > barcode > BarcodeScanner.kt  Companion object of BarcodeScanner |
| storeArea | String, non null | Stores the storeArea of the user so that it can be retrieved from the dropdown menu and still modified outside of the menu as well as used in authorisation checks outside of the main log in function. | ui > login > LoginActivity.kt  Companion object of LoginActivity |
| userName | String, non null | Stores the username of the user in a companion object so that it is usable across multiple fragments. E.g. used in the side menu to display the current logged in user | ui > MainMenu.kt  Companion object of MainMenu |
| storeNumber | String, non null | Stores the store number of the user in a companion object so that it is usable across multiple fragments. E.g. to check the store for recipients of a ping | ui > MainMenu.kt  Companion object of MainMenu |

**Key Classes:**

| **Class Name** | **Function** | **Location** |
| --- | --- | --- |
| Item | Acts as a model for all items with their possible characteristics. Where possible, attributes have been set to non-null and unchangeable however attributes that may be null (such as delDate) can be null and attributes such as backRoomLoc are changeable | Data > items > item.kt |
| Users | Acts as a model for user logins, containing the username, password, store number and store area as unchangeable and non null. | Data > LoginAPI.kt |
| Ping | Acts as a model for pings that are retrieved from the database, containing the senderID, recipientID, pingType, zoneLocation and sendTime as unchangeable, non null attributes | Data > PingInboxAPI.kt |
| Recipients | Acts as a model for possible recipients of the ping retrieved from a database, containing the username, store number and status as unchangeable attributes | Data > PingSendingAPI.kt |
| FinalPing | Acts as a model for the attributes that must be included in the final ping, containing the senderID, recpientID, pingType and location as unchangeable, non null attributes | Data > PingSendingAPI.kt |

**Key Data Structures**

| **Structure name** | **Structure Type** | **Function** | **Location** |
| --- | --- | --- | --- |
| globalPings | ArrayList<Ping> | Stores the pings specific to that user, for use in the ping adapter as well as the ping inbox fragment, in order to be modified and displayed. | ui > communication > PingInboxFragment.kt  Companion object of  PingInboxFragment |

**Inputs and outputs:**

| Variable | Input Source | Expected Input | Validation |
| --- | --- | --- | --- |
| username | Text field in LoginActivity | A username in the correct format | None - but will be checked against the database to ensure it is a correct user |
| password | Text field in LoginActivity | A password that is correct for the user | None - but will be checked against the database to ensure it is a correct password for the user |
| storeNumber | Text field in LoginActivity | A store number that is correct for the user | None - but will be checked against the database to ensure it is a correct store number for the user |
| manualInput | Text field in BarcodeScanner | A UPC number of length 12. |  |

**User Interface Design:**

General Usability Features:

Button Consistency: In order to help users navigate the app easier, all the buttons have been coloured blue. This ensures that when the user sees a blue button, they can instinctively know it is a button and also ensures the buttons are clearly marked. This can help reduce confusion and improve efficiency when using the application. Buttons are also simple to understand, with a very easy input requirement, leaving validation to the system rather than to the user.

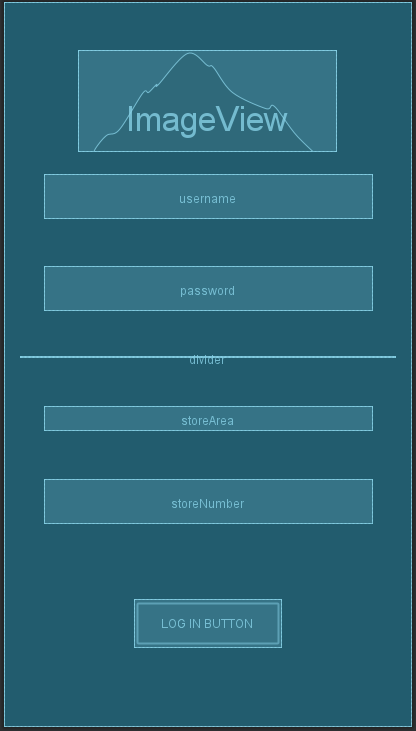
Clear Labels: By using clearly defined labels, of no more than one word, the purpose of each can be easily defined and understood without prior knowledge of the system. This can make it easier for newer employees to operate the system and allows them to work independently with less training on the system, increasing the efficiency of the store and decreasing costs. The software labels were designed in a way to be easy to understand by all workers, as ASDA employs a variety of workers of age and skill level where technological differences can vary and so by making it easy to understand in practical terms, more technological barriers can be overcome.

General Success Criteria:

* All code should work independent of each other: By splitting the functions of the application into separate screens, each screen can be edited and changed without affecting the other screens of the application. This allows for easier editing and testing of the application without needing other parts of the application to be finished or without breaking other parts of the application in the process.
* The code should work on any android device: By only using features accessible to android systems running android 5.1, almost all android devices. Running the application on any device would be almost impossible, but by running the system on approximately 92.3% devices the same effect can be achieved.
* The code should be streamlined and one screen should lead smoothly into the next:

Not all the designs for the screens were changed however, instead I tried to keep as many of the original designs as possible, where possible.It was kept this way so that it would act as a pseudo usability feature, by allowing for the knowledge of the previous system to be retained and carried forward. This means that I only had to design new parts of the device from the ground up, as shown below. Although they are not explicitly shown in this report, however, user feedback was still taken for the menu designs and other current designs in case users wanted them to be changed.

**Log in screen:**



This is the first screen shown to the user when they log in. It allows the user to input all the required data and information needed to be able to access the system. Each of these are easily accessible and logically follow on from each other down the screen, reaching a natural conclusion at the login button.

A divider is used to help keep the flow of the screen easy to understand, dividing the information on the specific user and the information on the store.

The addition of the asda logo at the top (imageView) also helps to keep the login screen aesthetically pleasing.

The username, password and storeNumber boxes are string inputs, whereas the storeArea is a drop down menu. The password box also hides the input of the user so that shoulder surfing can be mitigated.

Links to success criteria:

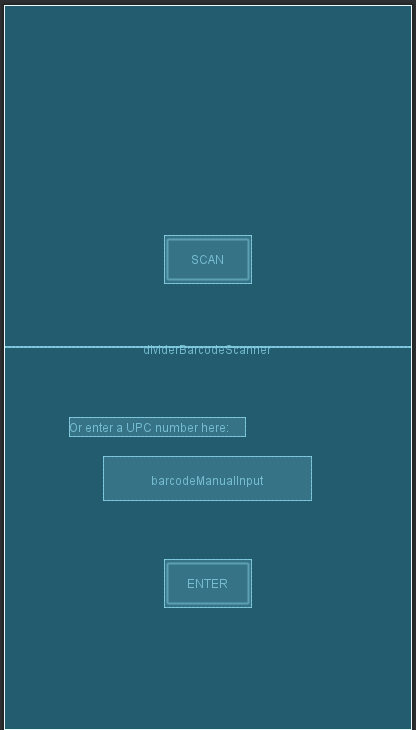
* [Store Area] should be a drop-down menu of 4 options
* Should be a text input field (for all of them other than storeArea)
* [Password] should be labeled “Password”
* [Username] should be labeled “Username”
* [Store Number] should be labeled ”Store Number”
* UI is simple to use
* Each screen leads smoothly into the next
* Labels are jargon free

Usability features:

I have considered and implemented usability features to ensure that the system can be used by all employees of adsa. The asda logo, and text inside the boxes are large and simple so that users with reading difficulty or poor eyesight can still read and understand their function. They are also large, including the large button (that will be coloured blue for visibility), making it easy for users to select the right input field that they require.

A problem with the dropdown menu is that the inputs can be quite close together, so during development work may be needed in order to give each input a larger hitbox if possible, especially as the user will have to select their option with their hands rather than a more precise input such as a mouse. The usage of the menu has also been made as easy as possible, as outlined in the design process, the fields have been set up to logically follow into each other.

**Barcode Scanner:**



This is the design for the screen shown when the user attempts to access the barcode scanner. It allows the user to input the upc of the desired item in the format that is easiest for them, by either manually typing in the bottom section or by scanning in the top section. Each of these are separated and correctly labeled to easily distinguish which method can be assessed in which half of the screen, this was developed by adding a divider and only implemented after user feedback, as shown in the next section. The barcode scanner will be used for multiple parts of the program, so should be as generic as possible in labelling to allow for no changes to be needed when calling this fragment.

The scan button is a simple input that links to the barcode scanning fragment. This will be clearly labeled with an easy to understand instruction, allowing it to be easy to use and understand.

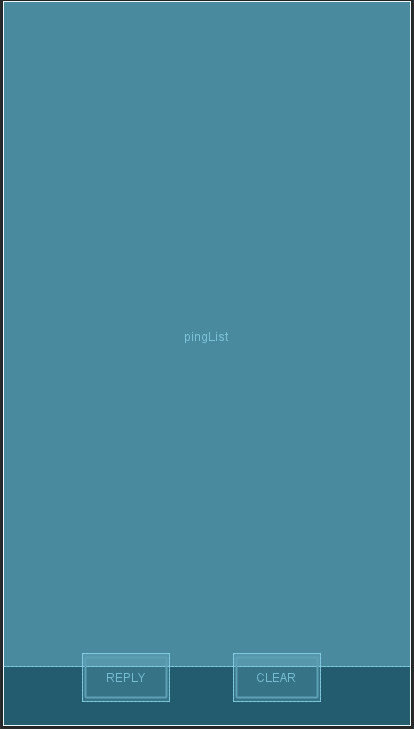
The barcodeManualInput field is an integer-only input field, that will allow users to input UPC numbers without needing a camera device. This will allow users to input items with damage, obscured or outdated barcodes. It is clearly labelled, so it is easy to understand it’s function and has a good screen flow from the scan button. It is self validating, where UPC numbers that do not correlate to an item will not appear and the user will be sent back to the previous screen.

Links to success criteria:

* Should be able to be manually inputted or visually scanned
* Should be callable from any function
* UI is simple to use
* Labels should be jargon free

Usability features:

I have considered the purpose of this screen in the context of the fragments it will be used by and have decreased the amount of possible inputs drastically so that it is easy to use by everyone. By only using two input fields, both taking in the same information, the user knows exactly what is needed of them but is still allowed to input it in the format that is easiest for them. For example, if their device doesn’t have a camera or if the barcode is damaged/obscured they can still use the features of the application. The buttons are also clearly labelled and distinct, so that is easy to understand and operate this screen.

**Pinging System (Inbox):**

This is the blueprint for the ping inbox screen. The majority of the list is taken up by a recycle list, which will display all the pings for that user in a scrollable list, sorted by the time that they were sent so that the oldest appear first.

The buttons are clearly labeled at the bottom, intending to be used as the eyes track the pings down the bottom of the list so that it naturally flows into them.

The clear button is essential and linked directly to the inboxes function. Clearing the pings allows the inbox to not become cluttered and hard to use, however they can still be kept until the task is done or while there are multiple tasks that would otherwise be hard to remember.

The reply button is non-essential, however has been included in order to make the application easier and faster to use. The reply button acts as a shortcut to the ping sending screen, which speeds up the amount of screens and inputs needed to get there.

Links to success criteria:

* Data should only be stored if it is relevant
* Should always receive any sent messages in order of newest first
* Should only load messages for the user
* Should allow the user to easily respond by taking them to the ping sending screen if they wish to reply.
* UI should be very simple to us

Stakeholder feedback:

After heavily designing these key parts of the program, I took the designs for these screens and the other designs for menus and showed them to stakeholders at my place of work. I did this in order to gather feedback to inform design choices and to justify choices made or any new adjustments. Below are the responses obtained:

* “ The ping inbox looks very simple to use, I like that you included the reply button, I wouldn’t even think to have one. My only problem would be if you had too many pings and they wouldn’t all show up.” They were then informed that the ping inbox was scrollable “Oh that’s fine then. Looks great.” - Ambient Colleague
* “I like the inclusion of the ASDA logo, it makes it look so much nicer than the current one.” - Health and Beauty Colleague
* “[the buttons] don’t have any of those stupid words that mean nothing unless you can work the device. I like that I shouldn’t have to teach anyone basic skills to use this.” - Ambient Manager
* “Having to press another button to scan things looks like a pain. I think I’d rather use the gun although the button is so large I’ll probably do it on instinct after a while and it wouldn’t matter anyway. So long as there’s nothing finicky or whatever after this then I don’t mind. Maybe you could add one of those divider things to it though.” - Warehouse worker

The general consensus of these new designs showed they were positively received, picking up on the simple and large buttons and that not too many objects crowded the screen. Although strong feelings were felt towards not having a dedicated scanning button on the side of the screen, it was decided that the design was probably the best it would be given it was a hardware limitation. The only change to the design was the addition of the divider on the scanning design.

**Validation:**

When using the software, the data inputted should be correct and validated in order to ensure the program stays robust and doesn’t allow for the user to break the code. This can be done by ensuring the keyboard type only supports integers, or using pre set menus rather than text input. The following list shows all the general types of input field and how they should be validated:

Buttons, Spinners and Navigation labels:

Buttons are a function of kotlin that is prebuilt and therefore they will be self-validating and robust without any additional coding. They do not pass through user inputs, other than an onclick effect, so when the user clicks on them, they will not need to be tested. The buttons can also be disabled until the user inputs the correct field to ensure atomicity in the databases, and therefore the only validation that will come with these will be setting pre-conditions for their use, so that they can only be used when all fields of a form have been inputted.

Other features that act like buttons, such as the navigation menu labels or spinners are also part of the kotlin language and will not have to be validated. This means that inputs done by fields of text for preset inputs, can instead utilise spinners to reduce the available inputs and the chance the user will put an incorrect input. It also means that additional validation is not needed.

The only checks that will need to be made for these is that the labelling and options are correct, which can be validated in the strings file.

Text Fields:

Test fields are components of the code that will have to be validated, so should be used only when completely necessary. They will be used in this code only when logging in or when inputting a UPC number. Validation would be performed at a database level, by comparing the inputs of each box with the string stored in the database. By performing checks only at database level, less time is wasted on validation inside the program and less code will need to be written. This means that the inputs themselves do not need to be tested, but the format of the keyboard could be changed based on what input it is. For example, the UPC text input could take numbers and integers and still compare but the keyboard that appears when the user clicks on this field could only show numbers in order to make it easier for the user to use.

By performing checks at database level, code can also be written to give back specific feedback based on what input was wrong. For example, the UPC number input, when input incorrectly, can feedback that an item with that UPC number could not be found and the user can adjust their inputs accordingly.

Camera Image:

The camera imaging is an inbuilt function of the language and so does not require validation, as it can be set up to only accept barcodes in the correct format (UPC-12). This means that it will only not work if the barcode is unscannable (damaged, obscured, incorrect etc) or if there is no barcode shown to it. To prevent program locks, it will be instantiated as a fragment so that the backstack can be used if the user cannot scan the barcode.

To prevent the camera image from becoming unusable when the barcode is unusable, a text field will be used in conjunction with the camera option, so that validation does not remove function without cause.

If neither cannot be input, the program will not work but will not crash, and other methods such as scanning the barcode of the item label on the shelf can be used, as they share UPC numbers.

**Testing Methods:**

A variety of testing methods will be carried out throughout the programming of each fragment, and as each function is developed in fragments with lots of functions. This will ensure that testing is as effective as possible in highlighting design oversights and bugs instead of incomplete design.

Constructive and Destructive Testing

Constructive testing will record all the inputs into the data fields and buttons as well as their intended and actual results. Unfortunately, the console is not always helpful when crashes occur, instead only sending the program back to the home screen of the device or the login screen, as those will be the most recent activities (assuming the user has logged in). Due to this, console outputs will only be displayed when helpful in informing future fixes and solutions to bugs. At this point, destructive testing may also be used to test how the program will respond to incorrect inputs and data. This will involve inputting incorrect usernames, passwords and UPC numbers and other incorrect data into the text fields. All of these inputs will be recorded in a table, along with expected outputs, actual outputs and any adjustments that were made.

Iterative Development and Testing:

During development, iterative development will also be used; where fragments will be designed and developed in rapid succession, in order to build up a functional fragment through rapid testing and changes. This will show how each function in the fragment is developed and tested independently of the fragment as a whole and will ensure each one is hopefully bug free as an independent unit. This will make bugs easier to isolate as they can be traced to that function rather than having a web of multiple functions it could come from.

Post Development Testing:

Post-development testing will involve both destructive testing and feature checking; where destructive testing is used to test the robustness of the program and feature checking will be used with the success criteria to measure how well the program fulfills the targets. Feature checking may also involve showing the program to the stakeholders after development, to ensure the software meets expectations and that there is nothing they would like to be implemented or changed.

General Testing Checklist:

Below is a generic testing checklist that can be used with all fragments in order to ensure that it is functional. I have used a generic checklist rather than a program checklist due to the large scale nature of the program, as well as each fragment being similar. If any actions do not relate, they can be marked as “N/A” and ignored. By making one repeatable checklist, I can save time when testing and developing. Any features that I do not feel are covered by this checklist for each fragment can be added to the checklist.

| **Tested Part** | **Operative Status** |
| --- | --- |
| Text formatting is correct (spelling, grammar, function label) |  |
| All navigation buttons lead to the correct fragment/activity |  |
| All form submission buttons will submit inputs correctly |  |
| All spinners show the correct options and all options available |  |
| The correct user details are shown on the navigation tab |  |

All fragments in the program will be tested against this checklist in order to ensure it is fully operational and that the coding of that fragment can finish. By having all fragments in the program adhere to this checklist, the development of the program as a whole can finish and time can be spent gathering user feedback from stakeholders and adjusting accordingly.

Barcode inputs:

For the barcode inputs that will be tested, I have taken wasted barcode labels from my place of work and uploaded their UPC numbers as the UPC numbers of items in the database. I have then posted these items onto my whiteboard, so that I can simulate a shelf by having lots of barcodes together, where the system will have to choose the correct one.

While consistency is usually good in testing, the barcodes themselves have been purposefully tilted or damaged, as this is often the case in a supermarket and testing “clean” barcode labels will not help test the versatility of the software. Instead, different angles and lighting conditions will be used when testing the barcodes to ensure that the reader can work under a variety of conditions. The set up of these barcodes is shown below.



Iterative Interface Inputs and Test Tables:

| **LoginActivity** | Variable | Input | Expected Output | Justification |
| --- | --- | --- | --- | --- |
| Test 1 | userName  password  storeArea  storeNumber | B0B02U0  abc123  Store/Club  4917 | The user details are correct and the user is logged in. | To test if the application will let users log in who have the correct credentials. |
| Test 2 | userName  password  storeArea  storeNumber | B0B02U  abc123  Store/Club  4917 | The user details are incorrect and the user is not logged in. | To test if the application will not let in users with an incorrect username. |
| Test 3 | userName  password  storeArea  storeNumber | B0B02U0  abc123  DC  4917 | The user details are incorrect and the user is not logged in. | To test if the application will not let in users with an incorrect store area. |
| Test 4 | userName  password  storeArea  storeNumber | B0B02U0  abc123  Store/Club  4916 | The user details are incorrect and the user is not logged in. | To test if the application will not let in users with an incorrect store number. |

| **MainMenu** | Variable | Input | Expected Output | Justification |
| --- | --- | --- | --- | --- |
| Test 1 | username.text  storenumber.text | B0B02U0  4917 | Username: B0B02U0  Store Number: 4917  This would be displayed on the top of the side menu. | This test was to make sure that the bar would update on the submission of new usernames and store numbers. |
| Test 2 | status | Log in with an offline user and then logout. | The user is offline | This test is to ensure the the database is correctly updated when the user logs out |
| Test 3 | Side bar navigation variables | Clicking on each | The correct fragment is shown | This test was done to ensure the sidebar was set up correctly. |

| **ShopfloorFragment** | Variable | Input | Expected Output | Justification |
| --- | --- | --- | --- | --- |
| Test 1 | barcodeScannerButton | Click | Navigation to barcode scanner | Check to see if the destination would appear when the button was selected |

| **CommunicationFragment** | Variable | Input | Expected Output | Justification |
| --- | --- | --- | --- | --- |
| Test 1 | userName | B0B02U0 as userName | The inbox button is enabled. | Checks that the ping inbox is only accessible if the user has pings. |
| Test 2 | userName | R9F25R8 as userName | The inbox button is disabled. | Checks that the ping inbox is only accessible if the user has pings. |
| Test 3 | inboxButton | Click | Navigation to the inbox menu | Check to see if the destination would appear when the button was selected |
| Test 4 | sendingButton | Click | Navigation to the sending fragment | Check to see if the destination would appear when the button was selected |

| **BackroomFragment** | Variable | Input | Expected Output | Justification |
| --- | --- | --- | --- | --- |
| Test 1 | pickButton | Click (picks available) | Navigation to the department list fragment | Check to see if the destination would appear when the button was selected |
| Test 2 | pickButton | No picks available | Button is disabled | Check to see if the button would be clickable when there are no picks |

| **PingSendingFragment** | Variable | Input | Expected Output | Justification |
| --- | --- | --- | --- | --- |
| Test1 | Recipient  pingType  Location | A0B012C3 (given A0B012C3 is logged in)  Pick  004/004/004 | A ping is constructed with the correct time, sender, recipient, type and location. This is then stored on the ping database and can be retrieved by A0B012C3 | To check that the correct type of ping is sent and stored. |

| **BarcodeScanner** | Variable | Input | Expected Output | Justification |
| --- | --- | --- | --- | --- |
| Test1 | result | Scan cancelled | The correct message is output “scan cancelled”. | To ensure that the barcode scanner can handle cancelling the scan |
| Test2 | - | Attempt to scan a nice magic the gathering trading card | The camera would stay on the nice looking mtg card | To ensure that the camera would not scan something that was not a barcode.  To look at nice cards. |
| Test 3 | result | Scan a barcode with upc number 400092167399 | The correct message is output “Scan results: 400092167399”. | To ensure that the barcode scanner could correctly scan barcodes. |
| Test 4 | manualInput.text | An incorrect string length (“A”) | The correct message is output “UPCs must be 12 long” | To ensure the barcode scanner would not take the wrong lengths. |
| Test 5 | manualInput.text | A correct string length (“400092167399”) | Nothing | To ensure the barcode scanner would take the correct lengths. |

| **BarcodeScanner - Item lookup** | Variable | Input | Expected Output | Justification |
| --- | --- | --- | --- | --- |
| Test 1 | UPC | Barcode with no matching item | The correct output message is displayed “item not found” | This is to check that the validation of the barcode scanner works when an incorrect item is scanned |
| Test 2 | UPC | Barcode with a matching item | The fragment is moved to the item look up fragment | This is to check that the validation of the barcode scanner works when a correct item is scanned |

| **Item Info Fragment** | Variable | Input | Expected Output | Justification |
| --- | --- | --- | --- | --- |
| Test 1 | item | A correct upc number was passed to the fragment | The correct item is displayed | To ensure that the item info screen would show the correct item |
| Test 2 | addShopButton | Clicked | The shop floor increases by one case and the backroom decreases by one case. The shopfloor button is disabled when there is not a case in the back or no space. | To check to see if the add shop button would work correctly and perform its function |
| Test 3 | addBackButton | Clicked | The shop floor decreases by one case and the backroom increases by one case. The backroom button is disabled when there is not a case on the shop floor. | To check to see if the add shop button would work correctly and perform its function |
| Test 4 | newScanButton | Clicked | Navigation to the barcode scanning function. | To ensure that the correct navigation was set up for the button. |
| Test 5 | addShopButton | No shopfloor location | The button is disabled | To check to see if the add shop button would be disabled when there was no location for it. |
| Test 6 | addBackButton | No backroom location | The button is disabled | To check to see if the add back button would be disabled when there was no location for it. |

| **PingSendingFragment** | Variable | Input | Expected Output | Justification |
| --- | --- | --- | --- | --- |
| Test 1 | Recipient  pingType  Location | A0B012C3  Pick  004/004/004 | A ping is constructed with the correct time, sender, recipient, type and location. This is then stored on the ping database and can be retrieved by A0B012C3 | To check that the correct type of ping is sent and stored. |
| Test 2 | simulatedLocations | “000/000/000” | The list of zones is correct with the additional zone. | To ensure that the zone is only added once |
| Test 3 | sendButton | Click (nav) | The fragment would return to the communication menu | To ensure it goes to the correct fragment. |

| **PingInboxFragment** | Variable | Input | Expected Output | Justification |
| --- | --- | --- | --- | --- |
| Test 1 | userName | B0B02U0 | Two pings are displayed with the correct information | To ensure the ping inbox screen can correctly display pings on the database. |
| Test 2 | replyButton | Click | The user is taken to the ping sending fragment. | To ensure that the navigation of the ping sending fragment was correct. |
| Test 3 | clearButton | Click | The pings for the user was cleared | To ensure that the clear button correctly clears all the pings |

| **Communication Branch** | Variable | Input | Expected Output | Justification |
| --- | --- | --- | --- | --- |
| Test 1 | simulatedItems | Upc number on the pick list | The item is removed from the list and the list is updated | To ensure the picks would be scanned off when the upc number was correct |
| Test 2 | simulatedItems | Upc number not on the pick list | The item is not removed from the list and the correct message is output. The user is returned to the pick list. | To ensure the picks would not be scanned off when the upc number was incorrect |

Post Development Tests:

Video Walkthrough:

A video walkthrough of the program will be undertaken at this stage. Here, the functionality of the program will be tested by attempting to use every input and recording the results. This recording can then be used as evidence of the functions working and how they operate.

Usability tasks:

* Log in to the device and send someone else a ping (I ensured at least 1 other user would always be logged in). Ask them to rumble aisle 3 and then meet you in the backroom, location 2. Then log out.
* Log in to the device and find out all the information you can on Spaghetti. Then, move a case onto the shopfloor. Then log out.
* Log into the device and pick off all the items in the pasta department. Then log out.

These tasks were added so that evidence could be gathered for the usability of the devices, gathering user feedback so that I am sure that the device meets their needs when carrying out their job, as well as more general information about the ease of use.

**Iterative Testing and Development:**

Due to the nature of Kotlin Development, the development of this project will be split into 3 main folders that house the code. These sections of code must be developed alongside each other but will be kept separate in order to improve readability of the code.

The first folder, the data folder, will store adapters for the recycler views, the APIs for the database connectivity and the global variables in the system. These are kept separate as they act as a sort of data system, not carrying out any logical decisions but instead handling the storage and formatting of data.

The second folder, the UI folder, will handle the main bulk of logical code and decision making. This will act as a bridge between the other two folders, connecting the data behind the scenes to the visual outputs that the user sees. This will be what is largely tested and developed, as this is the only complex part that solves computational problems.

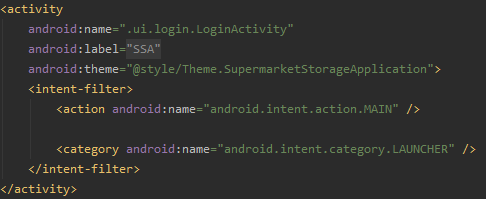
Finally, the resources folder will house all of the string resources, theme colours, layouts, images and navigational maps used within the code. This does not need to be tested as it is not programmable, only the variable names need to be formatted correctly in order for it to work. In cases where the layouts are constraint layouts, constraints will need to be set up between each object but the program will allow you to drag allows between objects and other objects or the sides of the screen to achieve this and hence no actual programming is needed here.

**Stage 1 - Login Activity**

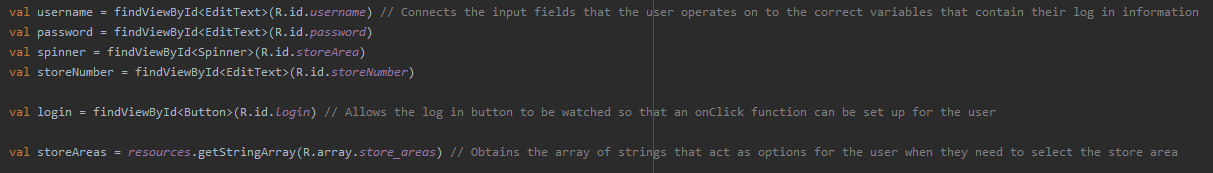
****

The first step is to import all of the requirements of the system, which is automatically done by the IDE. As it is automatically done by the IDE and is not a required development task **it will not be shown again** but has been included here as an example of what it looks like.

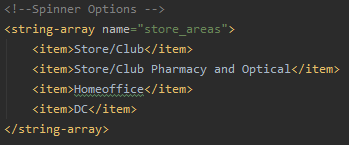
Here a class LoginActivity has been created, of the type AppCompatActivity, as it is an activity and not a fragment. This is further shown in the manifest.xml file where this section of code must be added, shown below. In addition, the correct viewmodel has been assigned to the activity, this essentially gives the code extra options in the code, but has not been used outside of the first activity as it was found to be tedious to code, resulted in code being developed over multiple classes and files and reduced the efficiency of the final product. All fragments and classes must use a viewmodel however, but they have been left blank outside of this one as a result of this prototype.

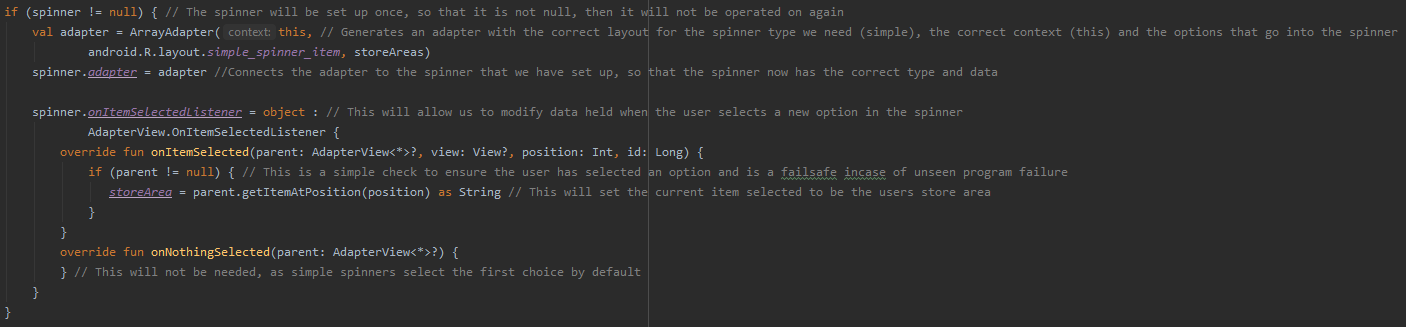


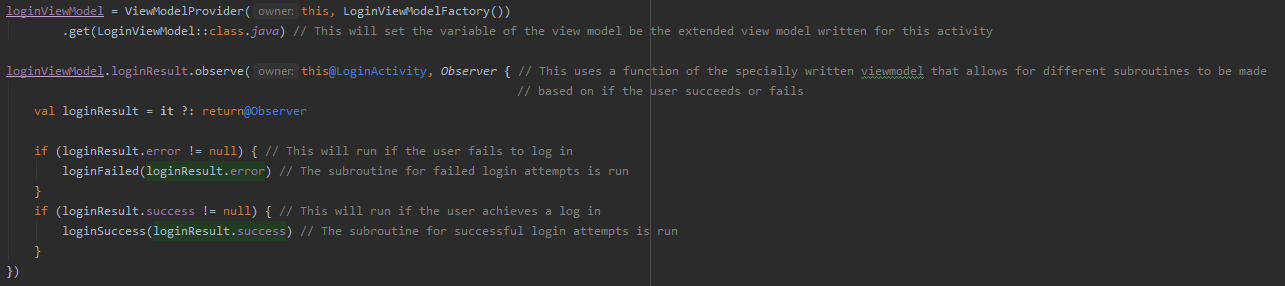
This shows the activity name, the label (what the user will see in the top left hand corner of the action bar), the style it uses (theme) and an additional intent tag is used; this is to define it as the launcher activity of the application, what the application should open on.

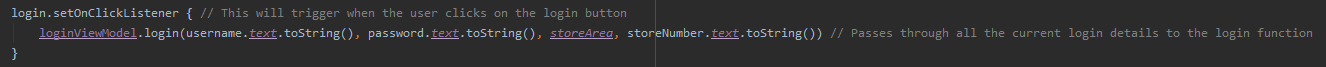
Next, the variables have been set up for use throughout the code. As shown by the pseudocode in the design section, values (unchangeable variables) have been set up to find the paths to each text view, spinner or button and link to their values. This means that although their data values may change, they will not be overwritten accidentally in code and will always point to those values. In development of this initial prototype, I found this to be extremely useful and I intend to keep all pointers as values throughout the development for this reason. 

The final variable obtains the strings stored in the strings.xml file, as shown below and stores them as an unchangeable array. I have found that by storing all potential strings in the .xml file and then accessing them when I need them, I can more easily manage the outputs I give to the end user, save space on repeated strings and also test these outputs more easily, meaning there should be no spelling mistakes in the development.

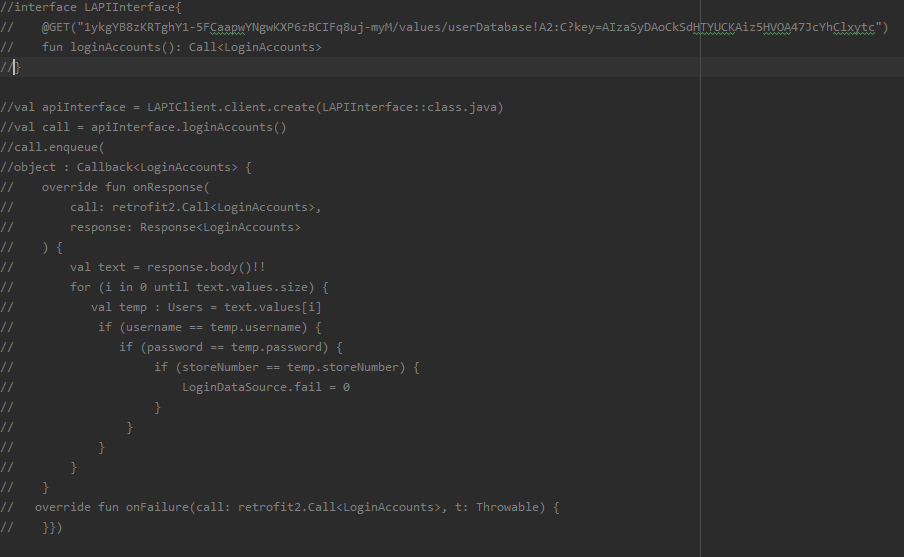


Next, the spinner will be set up; it will always be set up as although selection is used, the spinner will never be null thanks to its contents being a predefined string array of 4 options. The adapter for the spinner is then created, so that it knows what is using it, what type of spinner it needs to be and what string options it needs to use. The adapter is then attached to the spinner and an onClickListener is set up. This listener has two functions, although only the onItemSelected function needs to be used as the spinner keeps its original selection if they cancel their choice. The onItemSelected function will perform one last failsafe check to ensure the spinner is not empty and then it will set the storeArea of the user as the current selection. A change was made when developing the software, as it was found that time could be saved if the storeArea was set to its global variable now rather than further on in the code.

This next section of the code deals with setting up an observer inside the activity that watches for the result of a login attempt. If the login attempt is successful, it will run the loginSuccess function and if the login attempt is unsuccessful then it will run the loginFailed function.

This is the section of code that connects the input button declared earlier in the development to the login function provided by the viewmodel. It passes through all of the login details as a string, as a form of validation for the requirements of the function. This is done within the parameter declaration to save on time and cut down on code length.

Although there is one last part of the login activity to code, this is the end of the main function and the last part will be saved until the login function is developed. This is because there is currently no main menu for our program to move to and also it allows for this section of the code to be tested before the next activity is developed. To do this, the login function was written next (without the status update) to check if the database could be accessed and checked against the data obtained.

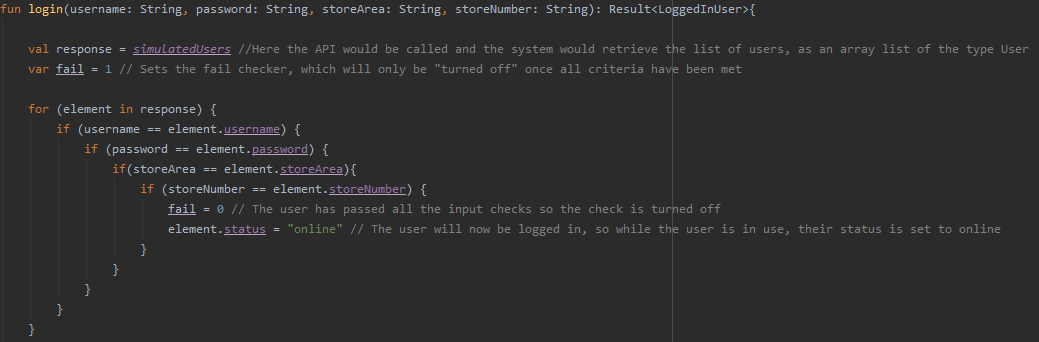
An immediately noticeable aspect of this code is that it is all commented out, this is because the code shown above does not work and more details of this are explained with the table below.

| **LoginActivity** | Variable | Input | Expected Output | Actual Outcome | Justification | Action taken |
| --- | --- | --- | --- | --- | --- | --- |
| Test 1 | userName  password  storeArea  storeNumber | B0B02U0  abc123  Store/Club  4917 | The user details are correct and the user is logged in. | The user details are retrieved from the database correctly, shown in the console. However, the user is not logged in and the data is not checked. | To test if the application will let users log in who have the correct credentials. | The database will be housed inside the application and the device will not have internet capabilities for the spreadsheets it modifies. |

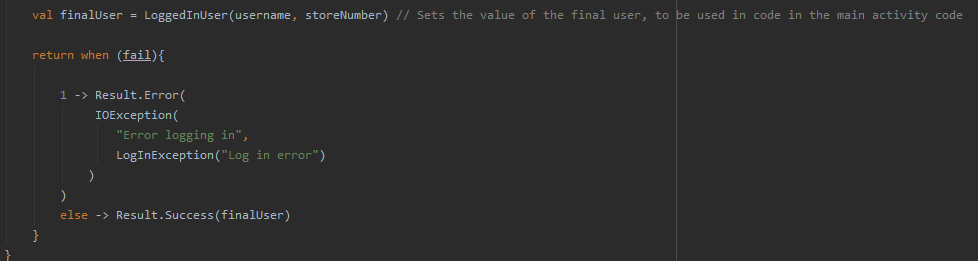
**Midpoint Review:**

The intentions of the above code were to set up an interface that the system could connect to, which would then retrieve all the data in the class specified (loginAccounts). The code below that was then almost identical to the pseudocode, where it is enqueued and the response was checked against to validate the user inputs. Unfortunately, it was found that the data would be retrieved but then no operations would occur on the data and it would always return an unsuccessful login. As I could not locate the source of the problem or solution to the problem it was decided that I would use a series of global variables inside the program in order to ensure the project would be complete in the time frame allotted. If I was to repeat this problem again, I would make sure to use a programming language I was more familiar with, as this is my first time coding anything with Kotlin and the retrofit plugin and I would also do more research on other solutions to databases and data storage. However, all the data is being stored in the format it would be retrieved in and minimal changes have been made to the algorithms and solution as a whole; this ensures that it can be compatible with the retrofitter plugin in the future and so would work if it was able to obtain the data in another way. This means that the database and the LAPI client written will be kept commented so that they can be used as a basis in the future of improvements but also as proof of attempt. 

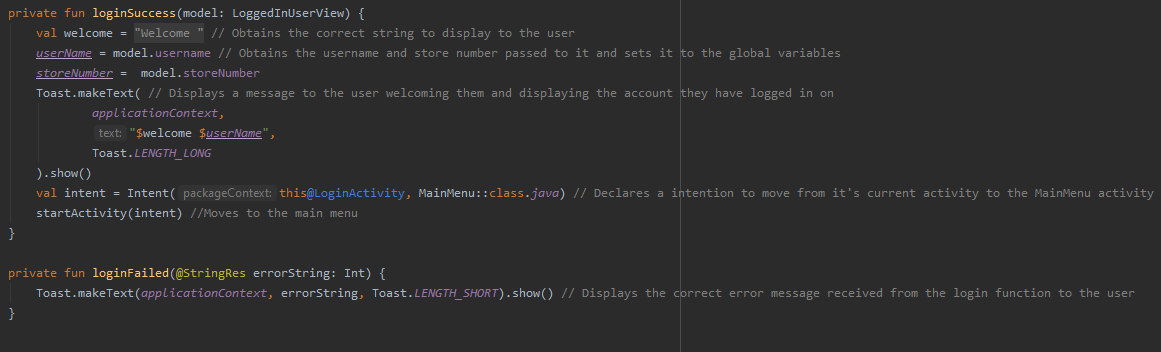
Instead, new code was written in its place and global objects were set up to house all of the databases required for the solution to be developed. The new global objects were housed in the global variables folder. The new code is shown below.



The first line of the login function “calls” for the responses from the global array of users and also sets a variable of fail to 1. This acts as a fail checker and is turned on (1) unless the user has correctly entered a username that then has a matching password, store area and store number and is turned off (0). The user is also changed to be online in the database, by changing their status to online.



At this point of the prototype, it can finally be linked up to the main program. The final user is set to equal the results of the user’s input, regardless of whether they are successful or not. This is so that they can potentially be used by both the unsuccessful and successful functions later in the code. Next, the function decides what to return; if the fail check was not turned off, then it says there was an error logging in and returns to the result.error intermediate function and if it was a success then it returns to the result.success intermediate function. These functions are just stepping stones for the viewmodel however, which then takes them directly to the login failed or login successful functions based on the response.

These are the last two functions that must be written for the login activity. Both of them make toasts (outputs messages) to the user based on if they were successful or unsuccessful. The loginFailed takes in the error string passed through to it from the login function and outputs it. Although there is only one string it can receive, it was kept this way in case improvements for more specific feedback was to be made possible in the future. Although I debated having “password incorrect” type feedback, I thought this would be detrimental to security and decided not to keep it but left the string resources and code options in the code in case I changed my mind from stakeholder feedback.

The loginSuccess function also makes a toast to the user but also has more functionality. It collects the username and store number given to it and then changes the global variables to these values for the future. Although the data is not changed between the initial collection and pass through to the function, it was kept this way for any checks to be made and added in future iterations of this software. It also creates the correct intent and starts the intent in order for the application to move to the next activity.

| **LoginActivity - 2** | Variable | Input | Expected Output | Actual Outcome | Justification | Action taken |
| --- | --- | --- | --- | --- | --- | --- |
| Test 1 | userName  password  storeArea  storeNumber | B0B02U0  abc123  Store/Club  4917 | The user details are correct and the user is logged in. | The user is logged in but it is realised they are not set to be online. | To test if the application will let users log in who have the correct credentials. | The code has been modified to add *simulatedUsers* = response  After the line  element.status = "online"  This has now fixed the issue and this is fully operational. |
| Test 2 | userName  password  storeArea  storeNumber | B0B02U  abc123  Store/Club  4917 | The user details are incorrect and the user is not logged in. | The user is logged in. | To test if the application will not let in users with an incorrect username. | The variable fail was found to only be 1 (indicating fail) on the first log in attempt. The variable would then always be 0 after, even if the user logs out, so long as the program remained open. The variable was updated to change back to 1 at the start of all login attempts, not just the first one. |
| Test 2.1 | userName  password  storeArea  storeNumber | B0B02U0  abc12  Store/Club  4917 | The user details are incorrect and the user is not logged in. | A toast appears saying “login failed” | To test if the application will not let in users with an incorrect password. | None |
| Test 3 | userName  password  storeArea  storeNumber | B0B02U0  abc123  DC  4917 | The user details are incorrect and the user is not logged in. | A toast appears saying “login failed” | To test if the application will not let in users with an incorrect store area. | None |
| Test 4 | userName  password  storeArea  storeNumber | B0B02U0  abc123  Store/Club  4916 | The user details are incorrect and the user is not logged in. | A toast appears saying “login failed” | To test if the application will not let in users with an incorrect store number. | None |

**Review:**

What has been done:

The ability for the user to log into the system has now been completed. The user can now input all of their required information and have the system securely handle and validate this information in order to give them potential access to the application as a whole. It achieves this by gathering data, checking it against a large database for validation and then handling the result it receives from these checks. It also stores their information in global variables so that it can be used by other processes in the application.

Prototype Summary:

Running the application causes the user to face a login screen. Correctly logging in will take them to a blank screen and incorrectly logging in will cause them to remain where they are.

How it has been tested:

Although not ideal, the functions were tested once the login function had been produced, as this would be the only time the application could produce test results. Erroneous, boundary and normal test data was put into each of the fields in turn so that they could be tested for full functionality. This data was then stored and compared in test tables from the design portion of this report.

The online database had been tested by monitoring results from the console and traffic to the database, although ultimately this ended up providing no helpful evidence of note.

Each function was also informally tested as they were developed, with the IDE performing syntax and spelling checks as well as imports and other tasks as I developed. This ensured that each function was only subject to logical errors and not simple errors that would not show up as easily in testing.

How it meets the success criteria and user expectations:

For the user, this fully meets their expectations; it logs them in correctly and has features such as hidden passwords and spinners to make their login experience easier. Comparing the code written to the general success criteria checklist, it also completes all applicable criteria on there as well as the success criteria in the login screen requirements in the analysis section.

The glaring criteria it fails to meet is the online database connectivity but unfortunately, this is unsolvable and a huge focus for future improvements. However, once online connectivity can be achieved then it can be easily integrated without many changes to the code and logic structure so it has been met to the best of my ability with this language.

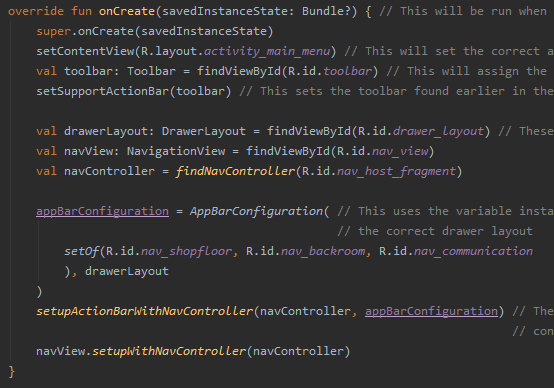
It also meets all of the applicable success criteria outlined in the general success criteria of the analysis.

Changes in the design:

The design will no longer feature online support, places where the pseudocode interacted with the online database will now cause it to interact with a global array. However, the algorithms and code are unchanged in terms of logical steps and the designs for all of them are unchanged.

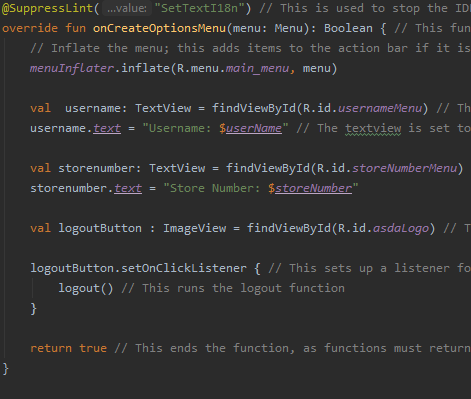
**Stage 2 - Main Menu:**

The first process of developing the main menu is to set up the navigational inputs. These include the buttons for each function as well as the navigational sidebar. This will be set up using dummy fragments as the end destinations for each of these buttons. To achieve this, the menu bar must first be set up so that each of these navigational menus can be accessed. This stage will also handle the process of logging out, as this is expressed by a large button on the main menu. Then, the navigational menus themselves can be set up, configuring the buttons to each (empty for now) destination fragment and adding checks to see if the buttons should be disabled. Once all of this has been developed and tested, the second prototype will be complete.

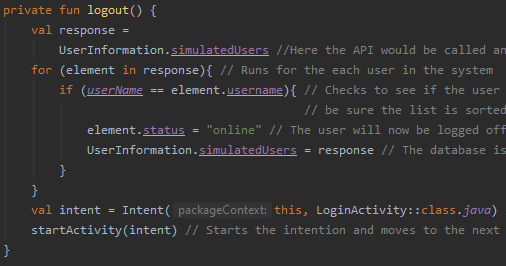


On the creation of the new menu activity, this section of code will be run first. Firstly, it collects the correct layout and sets it to the screen. Then it finds the toolbar, drawer layout and navigational components. These can be used to generate the action bar on the side of the screen. This will act as a tray that can be pulled out at any point by clicking on its icon, where a list of available menus will then be displayed. The navigational controller will be used throughout the program to move from one fragment to another fragment. This means that all fragments must be present on the mobile\_navigation.xml file, but no coding is needed to achieve this, other than app:startDestination="@+id/nav\_shopfloor" which will set the shopfloor menu to be starting fragment of this navigational “map”.

This code also configures the inputs on the drawer layout, matching up the shopfloor, backroom and communication menus to the corresponding labels on the drawerLayout layout. Finally, it can set up the action bar with the correct controller and configuration.



This function is also present in the main menu screen, and overrides the main menu creation to add a header section at the top. Firstly, the username and store number text views are obtained and then updated to represent the current user logged in. Finally, the image view is obtained so that a listener can be set up to watch for a click. When the button is clicked, the following function is ran:



This function acts as the logout function for the device, allowing the user to change to a different account or just represent the fact that they are not working anymore. Ideally, the database would be checked for the user and then directly edited, however as the system is now operating offline I have taken a different approach. The array is taken as a response of “calling” for the database and then the database is searched until the user is found. The status of the user is then set to be online and the database is then updated to reflect this change by overwriting the original database. Here a check could also be made, where once it has been changed the for loop ends, however this would increase the time factor due to the additional if statements needed in a worse case scenario so has not been implemented.

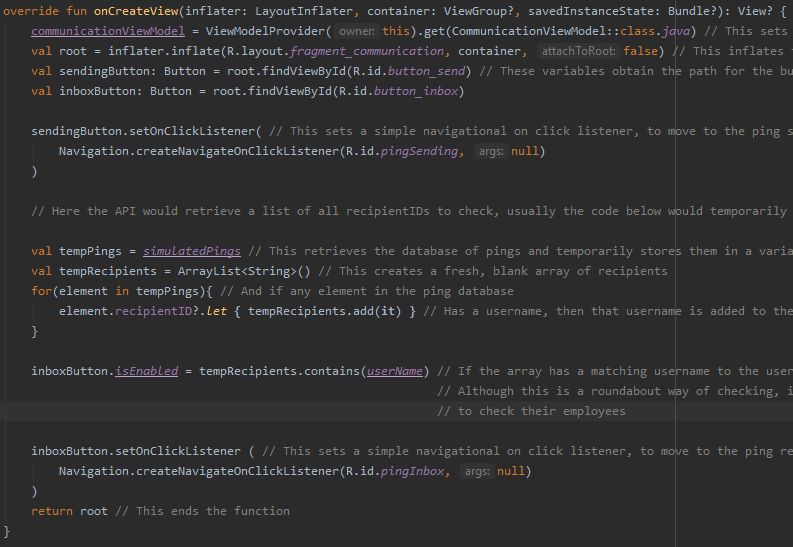
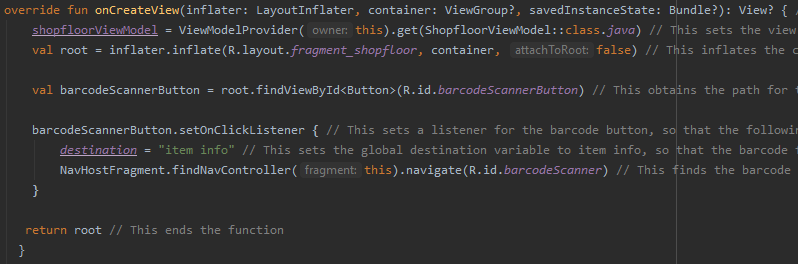
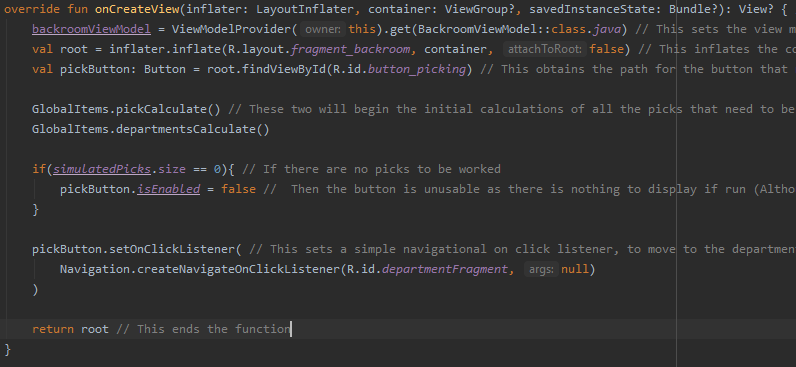
After this has happened, the correct intention to leave the main menu activity is made and the program moves to the login activity.

| **MainMenu** | Variable | Input | Expected Output | Actual Outcome | Justification | Action taken |
| --- | --- | --- | --- | --- | --- | --- |
| Test 1 | username.text  storenumber.text | B0B02U0  4917 | Username: B0B02U0  Store Number: 4917  This would be displayed on the top of the side menu. | Username: B0B02U0  Store Number: 4917  Was displayed on the top of the side menu. | This test was to make sure that the bar would update on the submission of new usernames and store numbers. | None |
| Test 2 | status | Log in with an offline user and then logout. | The user is offline | The user is online | This test is to ensure the the database is correctly updated when the user logs out | Changed the line element.status = “online”  To  = “offline” so that the user’s status was correctly represented. This fixed the bug. |
| Test 3 | Sidebar navigation variables | Clicking on each | The correct fragment is shown | The correct fragments were shown | This test was done to ensure the sidebar was set up correctly. | None |
| Unexpected Test | screenOrientation | “nosensor” | Screen won’t rotate and crash the application. | Screen didn’t rotate or crash the application. | During this stage of testing it was found that the application would crash on rotation. | android:screenOrientation="nosensor"  This was added to the Manifest file so that the screen would never rotate from the default and the device would not crash. |

**Midpoint Review:**

At this point, the prototype is now able to implement functions branching out from this set of menus. The navigational component of the prototype is generally complete and new functions only need to be added to the set of menus and built off this skeleton. This has been done by setting up the navigational files upon initial creation of the activity, making the main\_menu more of a holder for each fragment to be displayed on. However, I have kept it as main\_menu and not main\_activity because I consider that to make the most sense when coding, as it is mostly viewed with the sidebar in mind. It has been tested by viewing it through different accounts and ensuring it works the same for all. Nothing has changed in the design yet, other than that the program is unable to be used in landscape (something not considered in the analysis or design and so is negligible) and the success criteria is being met as of this part.

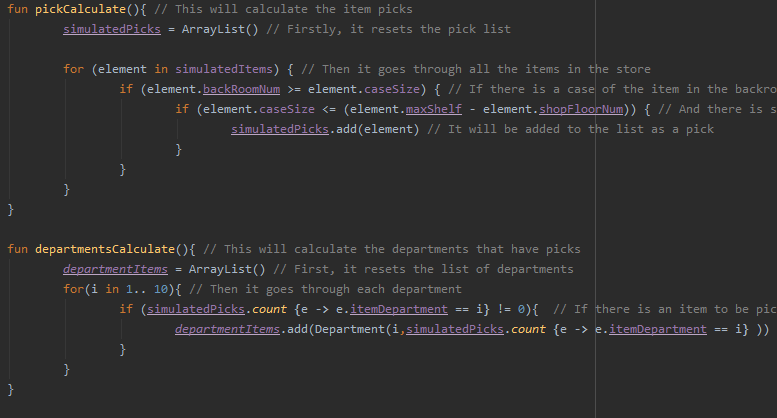
**Stage 2 - Continued:**

Next, the menus for the shopfloor, backroom and communicational aspects were properly set up. They are very similar, so will be shown below together although each unique part will be addressed in turn.

The first section of code will run the shopfloor menu. This is the simplest menu, where a simple button is set up. This button will set the end destination of the barcode scanner and then will navigate to the barcode scanning fragment.

The next menu of code is more complex, where two buttons are set up for ping sending and receiving. The ping sending button is always enabled, as the user accessing it must always be online and therefore there is always one user online. However, the ping receiving button shouldn’t be accessible if there are no pings for that user. Because of this, the program will retrieve a list of all pings and then put all of the recipientIDs into a list. This list will then be checked against the user’s username and if there is a match the button will be enabled. Otherwise, it will be disabled. This is a roundabout method of solving this problem, but it’s future proofing for further additions to the software. By gathering all recipientIDs, managers could check the inboxes of their employees in the future and so it was set up in this way instead. Finally, the ping inbox button has a simple on click listener to navigate to the inbox when clicked.

The last menu of code uses outside functions in order to operate, with pickCalculate() generating the list of items to be worked on the shop floor and the departmentCalculate() function generating the list of departments to be worked. These two functions are explored in depth after this but would be replaced by extraction from a database, if internet connectivity occurred. Next, a simple selection occurs where the pick button is disabled when there are no more picks to work. Finally, a listener is set up to take the user to the department fragment using the navigational map.



These two functions are set to be global, so that the pick list can constantly be generated and updated as the list of items changes. Firstly, pick calculate resets the list of picks to be made. Then, for each item on the database it adds it to the array so long as there is a case available in the back room and there is space available on the shop floor.

Departments calculate works very similarly, firstly it resets the list of departments with picks, then for each department in the store, it adds the departments and the number of items in that department (in an object of the Department class) to the list of departments so long as the number of items on that department is not 0.

| **CommunicationFragment** | Variable | Input | Expected Output | Actual Outcome | Justification | Action taken |
| --- | --- | --- | --- | --- | --- | --- |
| Test 1 | userName | B0B02U0 as userName | The inbox button is enabled. | The inbox button was enabled. | Checks that the ping inbox is only accessible if the user has pings. | None |
| Test 2 | userName | R9F25R8 as userName | The inbox button is disabled. | The inbox button was disabled. | Checks that the ping inbox is only accessible if the user has pings. | None |
| Test 3 | inboxButton | Click | Navigation to the inbox menu | Navigated to the sending menu | Check to see if the destination would appear when the button was selected | See below |
| Test 4 | sendingButton | Click | Navigation to the sending fragment | Navigated to the inbox fragment | Check to see if the destination would appear when the button was selected | See below |

When doing test 3 and test 4, it was found that the blank fragments were incorrectly labeled and swapped for each. As they were blank fragments, their ID’s were simply swapped and then the code was fully operational again. No significant code changes were needed.

| **BackroomFragment** | Variable | Input | Expected Output | Actual Outcome | Justification | Action taken |
| --- | --- | --- | --- | --- | --- | --- |
| Test 1 | pickButton | Click (picks available) | Navigation to the department list fragment | Navigated to the department list fragment | Check to see if the destination would appear when the button was selected | None |
| Test 2 | pickButton | No picks available | Button is disabled | Button was disabled | Check to see if the button would be clickable when there are no picks | None |

| **ShopfloorFragment** | Variable | Input | Expected Output | Actual Outcome | Justification | Action taken |
| --- | --- | --- | --- | --- | --- | --- |
| Test 1 | barcodeScannerButton | Click | Navigation to the barcode scanner fragment | Navigated to the barcode scanner fragment | Check to see if the destination would appear when the button was selected | None |

**Review:**

What has been done:

The main menu has now been fully set up and blank fragments for each activity have now been attached to the menus. The user can also only access fragments if there is information available to them on those fragments. Otherwise, the user will be locked out and will not break the application. The program has also set up methods of logging out and generating picks and departments for use in later parts of the program. The user can interact with the logging out function and can now passively change their status without thinking about it. These functions have been made global where appropriate but encapsulation has tried to be maintained as much as possible.

Prototype Summary:

Running the application causes the user to face a login screen. Correctly logging in will take them to the shopfloor menu. They can access a side menu to access the communication, backroom or shopfloor menu. They can use the power button at the top of this menu to log out. The shop floor menu has a scan button that takes the user to a blank fragment. The communication menu has an inbox button that can only be accessed when there is a ping, that takes the user to a blank fragment. It also has a ping sending button that takes the user to a blank fragment. The backroom menu has a pick button that is only enabled when there are picks and takes the user to a blank fragment.

How it has been tested:

Testing the buttons were achieved by clicking on them and logging the fragments that the user ended up on. This allowed me to be certain that each fragment was set up to follow each other correctly. The conditional buttons were tested by setting their conditions to ones that would either switch them on or switch them off completely and their availability was then logged.

Testing the user specific features (store number and username) was done by checking the fields against matching inputs and ensuring that they matched. Testing the status was done by outputting the status of all the users and checking this to ensure it matched against the inputs and outputs given to the database.

I believe that these testing methods were inclusive of all scenarios (e.g. you cannot have -1 messages or picks) and that the code is now robust and will not break because of the conclusive testing. The testing did not result in any problems thanks to the good planning and decomposition made earlier in the design

How it meets the success criteria and user expectations:

Showing this to users, the overall response was positive. Many liked the multi-menu layout and that the side menu was easy to use and understand. All of the stakeholders liked the large log out button that was easy to understand although a few mentioned that a simple pop up toast of “logged out successfully” would be appreciated. This was added on by copying the previously written toasts and so does not warrant further testing as I already know this code works. However the code is supplied below:



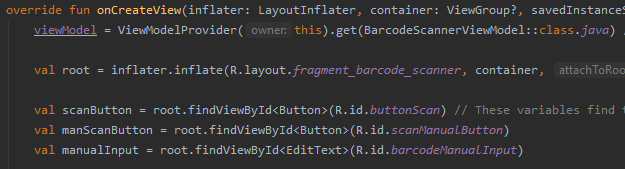
The current code however, meets all the success criteria outlined in the requirements table, feedback has shown it is easy to use and simple to understand and it also shows all the menus correctly, labeled correctly and they are all accessible.

In terms of the general success criteria, the UI is as quick as the current system, if not quicker due to it’s large button choices, where each menu will be the same format despite being on different devices unlike the current system. The account settings are also account specific and the global functions can function independently of which menu they are coming from, so the code is independent, commented and adaptable. This code is also streamlined and can work on any android device and the labels are free of jargon. Overall, it has met all the specific and general criteria.

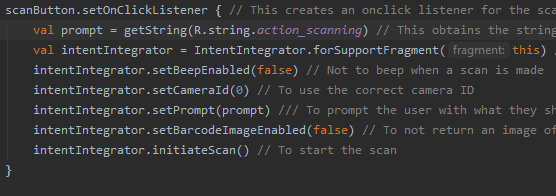
Changes in the design:

There have been no changes made to the overall design because of this stage.

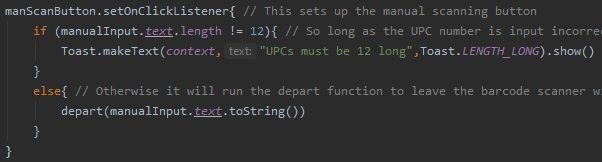
**Stage 3: Barcode Scanner**

****

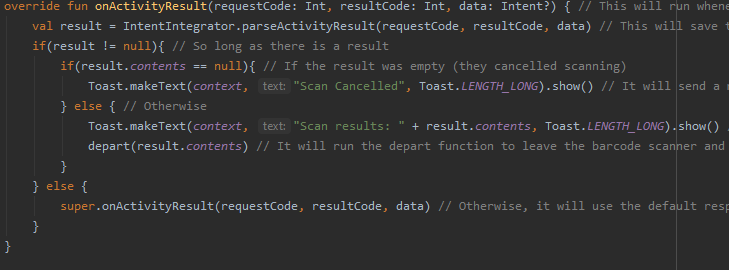
The first step of development was to set up all the buttons and text fields, the inputs that the user can use for UPC numbers. As before, the correct layout and view model are inflated and then the paths for the scan button, manual scan button and manual input fields are collected and attached to sensibly named variables.



Next, a simple intent integrator is set up in order to run this activity on a fragment. It is important to keep all of the main functions as fragments and not main activities so that they can all use the navigation map. This intent integrator is set up inside a simple on click listener, that waits for the user to click the scanning button and then will start the intent. Before it starts the intent, on the last line, it must first declare some properties of the barcode scanner it is about to use. Firstly, the “beep” sound effect that the camera makes is disabled, this is done because the scan sound effect is annoying for users and customers. Stakeholder feedback for this includes glares, angry comments and awkward looks. Next, the correct camera ID is set up, if it was a negative value then the program would have no preference however we need to use ID 0. Next, it sets the prompt to be the message gathered earlier in the button, so that the user knows what to do when they are accessing the camera. This is purely a usability feature and does not actually have to be in the code. Next, the .setBarcodeImageEnabled property is set to false, this is important as it tells the scanner that we only want the upc number and not an image of the barcode scanner that the camera would otherwise take. Finally, the scan is initiated and the barcode can return a upc number.



This is the final part of the fragment function, where the user instead chooses to use the manual option. Here a more complex listener is set up to run whenever the user clicks the manual scan button. If the upc number is not 12 letters long, then an appropriate message is output and the user is informed that it is wrong. This is done so that the network, if connected to one, is not used up by upc requests that will never be right and also to save time for the user who would have to wait for this request to be fulfilled. Otherwise, the depart function is called and the barcode scanner is almost ready to take the upc number to its destination.



This function must be written next, in order to handle the result of the barcode scanning activity that was supported by the fragment. This function collects the result of the intent set up earlier and stores it in an unchangeable variable. It is very important that this variable cannot be accidentally changed as it may become unusable after it has been scanned (it has been damaged or the box has been opened for instance). Then, a check is made to see if the camera has successfully returned a value. If it hasn’t, the default response is used otherwise, it must then check if the value is null. If the value is null, it means the user has cancelled the scan and the result is unusable. In this instance, it informs the user that the scan was cancelled and the user will remain on the barcode scanning fragment. They can then choose to either retry or use the backstack (back button) to return to their previous fragment. If the contents are not null however, and a upc number has been obtained, then the scan results will be shown to the user. This may not always be handy to the user, who may not care about the upc number, but the knowledge that a correct scan was made is and the extra information cannot be a detriment, it’s a unique usability feature the current system does not provide. Then, it will start the departure function to leave the barcode scanner.

Currently, the barcode scanner does not have anywhere to send the upc numbers to however, so the depart function is a blank function that returns null.

| **BarcodeScanner** | Variable | Input | Expected Output | Actual Outcome | Justification | Action taken |
| --- | --- | --- | --- | --- | --- | --- |
| Test1 | result | Scan cancelled | The correct message is output “scan cancelled”. | The correct message was output “scan cancelled”. | To ensure that the barcode scanner can handle cancelling the scan | None |
| Test2 | - | Attempt to scan a nice magic the gathering trading card | The camera would stay on the nice looking mtg card | Nothing happened and the camera was always on. | To ensure that the camera would not scan something that was not a barcode.  To look at nice cards. | Put the card back into the protective binder so that it was not damaged.  No change in code |
| Test 3 | result | Scan a barcode with upc number 400092167399 | The correct message is output “Scan results: 400092167399”. | The correct message was output “Scan results: 400092167399”. | To ensure that the barcode scanner could correctly scan barcodes. | None |
| Test 4 | manualInput.text | An incorrect string length (“A”) | The correct message is output “UPCs must be 12 long” | The correct message was output “UPCs must be 12 long” | To ensure the barcode scanner would not take the wrong lengths. | None |
| Test 5 | manualInput.text | A correct string length (“400092167399”) | Nothing | Nothing | To ensure the barcode scanner would take the correct lengths. | None |

**Review:**

What has been done:

The barcode scanner has been fully set up, the user can now input barcodes manually or visually in order to obtain upc numbers for the device to use. The scanner will only accept manual inputs if they are of the correct length, or barcodes if they are of the correct type. It can then take these barcodes and pass them to the depart function, although the function is not written yet as there is nowhere for the barcodes to go. This can be expanded in further iterations of the barcode scanner however, and for now all it needs to do is the validation.

Prototype Summary:

Running the application causes the user to face a login screen. Correctly logging in will take them to the shopfloor menu. They can access a side menu to access the communication, backroom or shopfloor menu. They can use the power button at the top of this menu to log out. The shop floor menu has a scan button that takes the user to **the barcode scanner. The barcode scanner will then collect the upc number input either visually or manually and prepare to give it away**. The communication menu has an inbox button that can only be accessed when there is a ping, that takes the user to a blank fragment. It also has a ping sending button that takes the user to a blank fragment. The backroom menu has a pick button that is only enabled when there are picks and takes the user to a blank fragment.

How it has been tested:

The barcode scanner has been tested by passing through both incorrect and correct data, as this is another instance where there is no boundary data. By passing these through for all circumstances, the robustness and validational aspects of the program have been fully tested and the code will not break under usage.

The scanning camera itself, as well as the buttons and text field, have simply been tested by using them to perform their functions and so their usability has been fully explored and they are known to work.

How it meets the success criteria and user expectations:

Comparing the developed solution to the requirements set out in the analysis, the project is found to be able to scan the correct barcode every time and can be manually input or visually scanned. This means it has met both the criteria set out in the requirements table.

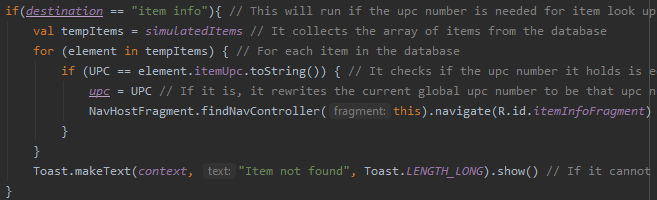
As with the other sections it has met the general criteria set out for it in the analysis and with the checklist in the design portion of this guide. However, as this is an essential part of the system I went and checked this with stakeholders, allowing them to try scanning the barcodes using their phone. Many people commented on the auto focus being “very noticeable in helping me, otherwise I’d have been here for ages setting it up” and there was a general census of the device being easy to use and simple to understand. This stakeholder meeting where I showed them the prototype allowed me to further assess my prototype to meet success criteria.

Changes in the design:

There have been no planned changes to the design, at this point or in the future. However, a consideration is being made to handle validation for the upc numbers inside this fragment, to allow the user to retry inputting numbers. This was suggested by a stakeholder and will be considered in further prototypes of the model as the more optimal solution.

**Stage 4 - Item Info:**

Before the item information fragment can be set up, the upc number must first be validated and collected from the barcode scanner. As outlined in the previous iteration, a change in design was made where the validation would now be made before the upc number is passed through. This will be done in the depart function of the barcode scanner, which can now be written as such:



This code performs validation specific to the item information fragment based on the destination being the iterm information fragment. It does this by collecting the items from the “online database” and temporarily storing the response. Next, it checks through the list of items to see if any of them have the matching upc number. As the upc number is given as an integer, it must be converted to a string in order to be compared with the upc number of the item object. Next, if this matches then it overwrites the current global upc number and uses the navigation controller generated in the main menu activity to move to the item information fragment, which is currently blank. If the list of items is searched and if there is no matching upc number for an item, it displays that the item could not be found and the user is kept on the barcode screen.

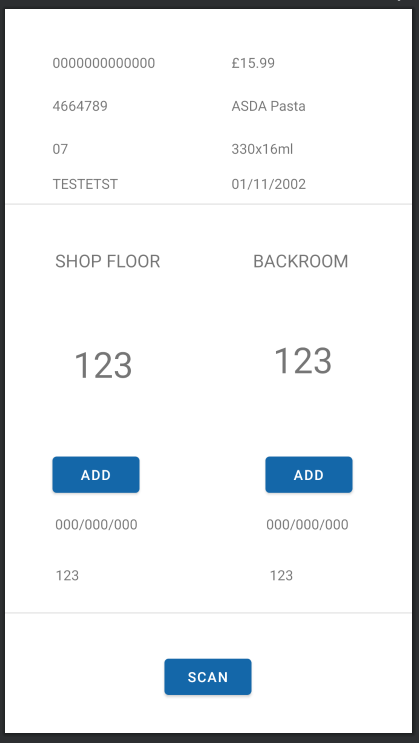
| **BarcodeScanner - Item lookup** | Variable | Input | Expected Output | Actual Outcome | Justification | Action taken |
| --- | --- | --- | --- | --- | --- | --- |
| Test 1 | UPC | Barcode with no matching item | The correct output message is displayed “item not found” | The correct output message was displayed “item not found” | This is to check that the validation of the barcode scanner works when an incorrect item is scanned | None |
| Test 2 | UPC | Barcode with a matching item | The fragment is moved to the item look up fragment | The fragment was moved to the item look up fragment | This is to check that the validation of the barcode scanner works when a correct item is scanned | None |

**Midpoint Review:**

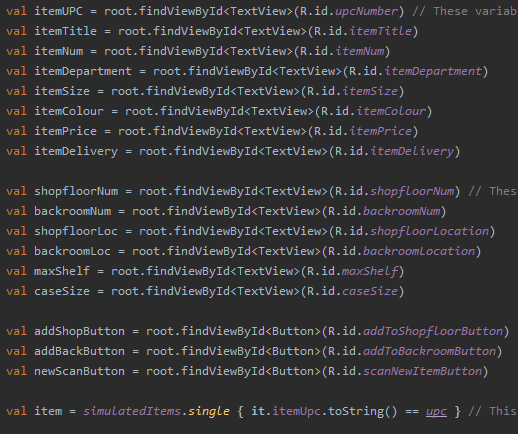
At this point in the development cycle, the barcode scanner can now validate the upc numbers it collects for the item look up screen and correctly determine if they are on the database or not. The upc number is now “safe” to pass to the global variable of the upc number and the fragment can be moved to the item look up screen. Thanks to the good design of the code, there have been minimal error tests when developing this section and other barcode sections. If I was to repeat this project, I would research all sections as thoroughly as this section which was well planned in pseudocode.

**Stage 4 - Continued:**

The next part of this iteration was to develop the screen that would display the information of the items that were looked up. This section will feature the design layout that I decided on, as it is important to see this to understand how it made the code easier to develop.



This is the design for the item look up screen layout used by the fragment. At the top, the attributes of the item have been replaced with filler data so that they can be spaced out properly but are null in the code. Dividers have also been used to separate the item information into 3 parts of different functions. The first section will display all the information of the item that the user searched up with the barcode scanner. The second section displays the number of items on the shop floor and in the back room. These can then be changed by adding cases to each with the buttons provided below. The areas that these items are stored in are then given below, with the case number and shelf number included (case for backroom and shelf for shop floor). The final section allows the user to search up another item. If this screen is too large for the user’s device, it has been made scrollable. If this was not the case, the scan button and divider (as well as the top informational pieces) would start to overlay onto other objects on the screen.

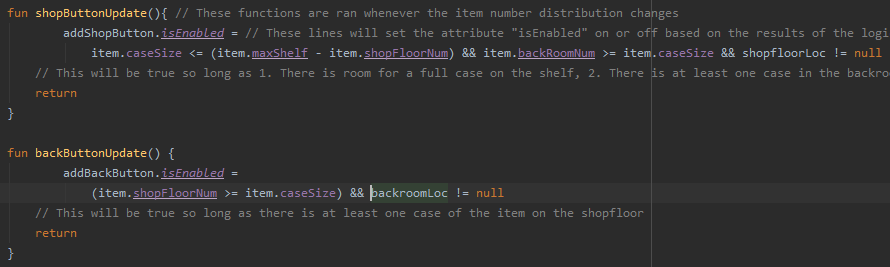


This section of code grabs all of the item’s attribute text views (the top section) and gives each of their paths to a variable that allows them to be overwritten.

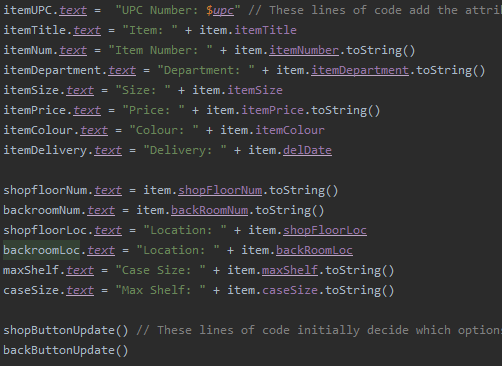
The next section of code deals with the text views for the items locations and numbers in the shop (the second section).

The next section of code grabs all the button inputs that the user can interact with, by using buttons and not allowing the user to manually input their own numbers, database integrity can be kept.

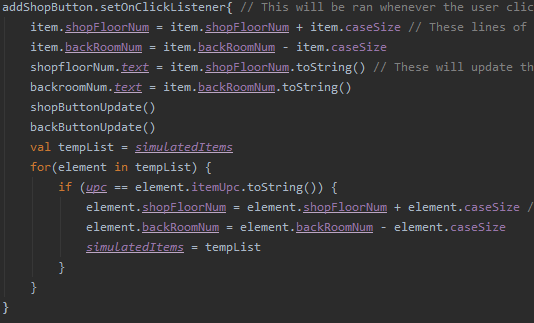
The final line of code grabs the item from the database that matches the upc number it has been given.

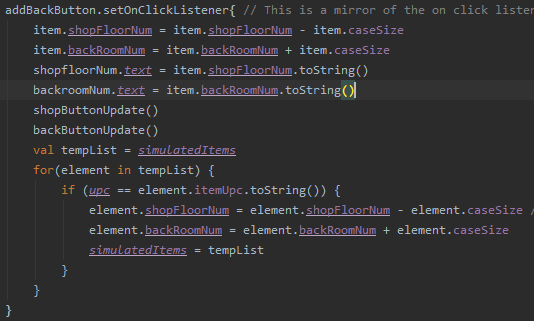


Two functions have now been written to help with the buttons that were obtained. They ensure that the buttons can only be used when it would be possible to move a case to that location. If there is not space for a case on the shop floor and there is a case in the backroom and the shop floor has a location for the item, then the button to add a case to the shopfloor will be enabled. Likewise, if there is a case on the shopfloor and there is a backroom location, then a case can be moved to the backroom, otherwise the button is disabled.



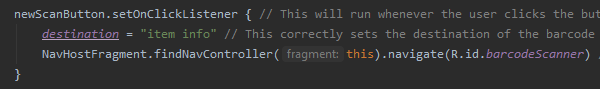
Next, all of the fields on the screen can be created, as well as the buttons which are set up at the end. This is done by setting all of their text in two sections, again reflecting the two sections shown to the user. Where the values are not stored as a string, they must be converted to a string in order to be shown.





Next, the two item buttons can be properly set up to accept inputs and change the database accordingly. The two buttons are a mirror of each other, where both will change the number stored by the temporary items, update the buttons and then update the simulated item database. The only difference between them is that the addShopButton will add the items to the shopfloor and take away from the back room whereas the addBackButton does the opposite.

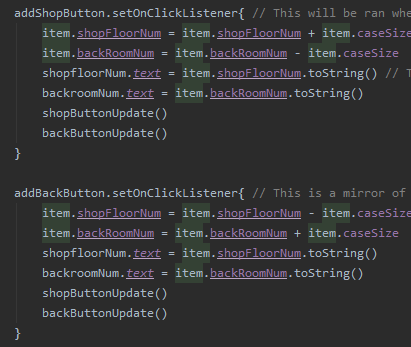
The database is updated by collecting the list of items, matching the item being changed to the item on the database, then changing the item numbers in the temporary list and uploading that list to the database. As this is such a short transaction, ACID in the database should be kept, as two users do not often work on picks at once.



Finally, the last button is set up so that the correct destination is set and then it navigates to the barcode scanner so that another barcode can be scanned.

| **Item Info Fragment** | Variable | Input | Expected Output | Actual Outcome | Justification | Action taken |
| --- | --- | --- | --- | --- | --- | --- |
| Test 1 | item | A correct upc number was passed to the fragment | The correct item is displayed | The correct item was displayed | To ensure that the item info screen would show the correct item | None |
| Test 2 | addShopButton | Clicked | The shop floor increases by one case and the backroom decreases by one case. The shopfloor button is disabled when there is not a case in the back or no space. | The shopfloor increases by two cases and the backroom decreases by one case. The button also takes two clicks before it comes into effect. | To check to see if the add shop button would work correctly and perform its function | See below |
| Test 3 | addBackButton | Clicked | The shop floor decreases by one case and the backroom increases by one case. The backroom button is disabled when there is not a case on the shop floor. | The backroom increases by two cases and the shopfloor decreases by one case. The button also takes two clicks before it comes into effect. | To check to see if the add shop button would work correctly and perform its function | See below |
| Test 4 | newScanButton | Clicked | Navigation to the barcode scanning function. | The barcode scanning fragment was accessed. | To ensure that the correct navigation was set up for the button. | None |
| Test 5 | addShopButton | No shopfloor location | The button is disabled | The button was not disabled. | To check to see if the add shop button would be disabled when there was no location for it. | See below |
| Test 6 | addBackButton | No backroom location | The button is disabled | The button was not disabled. | To check to see if the add back button would be disabled when there was no location for it. | See below |

Tests 5 and 6 showed an oversight in design, where items without a shop floor or backroom location could still be put onto the shelves or backroom. This is because the check that was used to ensure they had a location was using the variable of the displayed value rather than the actual value. As there is always a displayed value, even when there is no actual value, this meant it would always be true. The code was updated to use the items value and not the text field and then tested again. This is shown below:



Tests 3 and 2 also showed flaws in the design. This was because the item variable that was made directly linked to the item in the database. Because of this, the items values were updated twice. In addition, the for loop was after the code that updated the values so the buttons were also not updating properly.

| **Item Info Fragment** | Variable | Input | Expected Output | Actual Outcome | Justification | Action taken |
| --- | --- | --- | --- | --- | --- | --- |
| Test 2.2 | addShopButton | Clicked | The shop floor increases by one case and the backroom decreases by one case. The shopfloor button is disabled when there is not a case in the back or no space. | The shop floor increased by one case and the backroom decreased by one case. The shopfloor button was disabled when there was not a case | To check to see if the add shop button would work correctly and perform its function | None |
| Test 3.2 | addBackButton | Clicked | The shop floor decreases by one case and the backroom increases by one case. The backroom button is disabled when there is not a case on the shop floor. | The shop floor decreased by one case and the backroom increased by one case. The backroom button was disabled when there was not a case on the shop floor. | To check to see if the add shop button would work correctly and perform its function | None |
| Test 5.2 | addShopButton | No shopfloor location | The button is disabled | The button was disabled. | To check to see if the add shop button would be disabled when there was no location for it. | See below |
| Test 6.2 | addBackButton | No backroom location | The button is disabled | The button was disabled. | To check to see if the add back button would be disabled when there was no location for it. | None |

**Review:**

What has been done:

The first destination of the barcode scanner has now been set up. The barcode scanner can use the validated upc number and pass this to the global variable, where the item information screen can then display all of the information gathered on that item from the database. The database can then be modified as the user uses two simple buttons to add cases to the backroom or the shopfloor. The numbers shown to the user are also updated in real time and the database is only accessed when necessary for short periods of time in order to improve atomicity. Where possible, validation has been carried out with the temporary numbers rather than the numbers from the database in order to cut down on network usage. The user can also use the button at the bottom of the screen to scan a new item, forming one continuous loop with itself.

Prototype Summary:

Running the application causes the user to face a login screen. Correctly logging in will take them to the shopfloor menu. They can access a side menu to access the communication, backroom or shopfloor menu. They can use the power button at the top of this menu to log out. The shop floor menu has a scan button that takes the user to the barcode scanner. The barcode scanner will then collect the upc number input either visually or manually and **the upc number then undergoes validation to ensure it corresponds to an item. If it doesn’t the user stays on the barcode scanner but if it does then the user can see the item info for that item. They can update the numbers on the shopfloor or in the backroom and can also choose to scan a new item**. The communication menu has an inbox button that can only be accessed when there is a ping, that takes the user to a blank fragment. It also has a ping sending button that takes the user to a blank fragment. The backroom menu has a pick button that is only enabled when there are picks and takes the user to a blank fragment.

How it has been tested:

The code has been robustly tested to cover all the user inputs as well as all the item inputs. The testing in this portion of the development was all about the 3 buttons that the user can interact with. Testing was done to ensure that the buttons were only active when they could be used, this was achieved by using the buttons until they no longer logically worked or by feeding it items which would automatically fail (no shopfloor location or back room location).

The final button was just tested by simply using it and ensuring that the correct output was achieved.

How it meets the success criteria and user expectations:

Firstly it meets all the success criteria set out for it in section 5 of the requirements table:

* Only the desired information should be editable
* All the information displayed should be up to date
* The correct item should be found every time

Where the program allows the item numbers to be edited, takes the most recent iteration of that item from the database and will find the correct item every time thanks to validation done for it in the barcode scanning function.

The program also meets the large checklist set in the design section.

Finally, the item information screen was shown to the stakeholders in order to help gather evidence for meeting the general success criteria set out in the analysis.

“I like that the delivery time is as the top and not buried beneath useless information” - Rosie (ambient colleague)

“The buttons are kinda fun to use, if this was our actual device it would be too tempting not to play with them a little” - Matt (ambient colleague)

“Finally, you got rid of that useless info at the bottom that we have. This is so easy to use and understand.” - Dhiran (ambient manager)

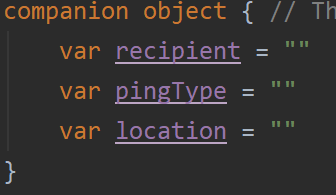
The following feedback shows that I have achieved the targets set out in the general success criteria, most notably making it more simple and easy to use as well as making it faster and jargon free.

Changes in the design:

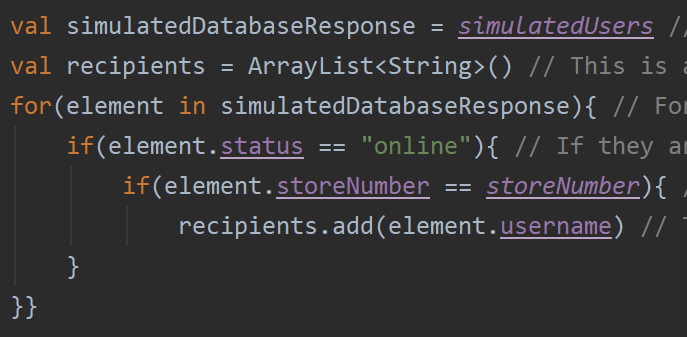
There are no planned changes to future designs due to this process, although a button was considered that would return you to the home menu. However, as this screen will be accessed by functions other than the main menu, this is not a planned implementation.

**Stage 5 - Communication (Ping sending):**

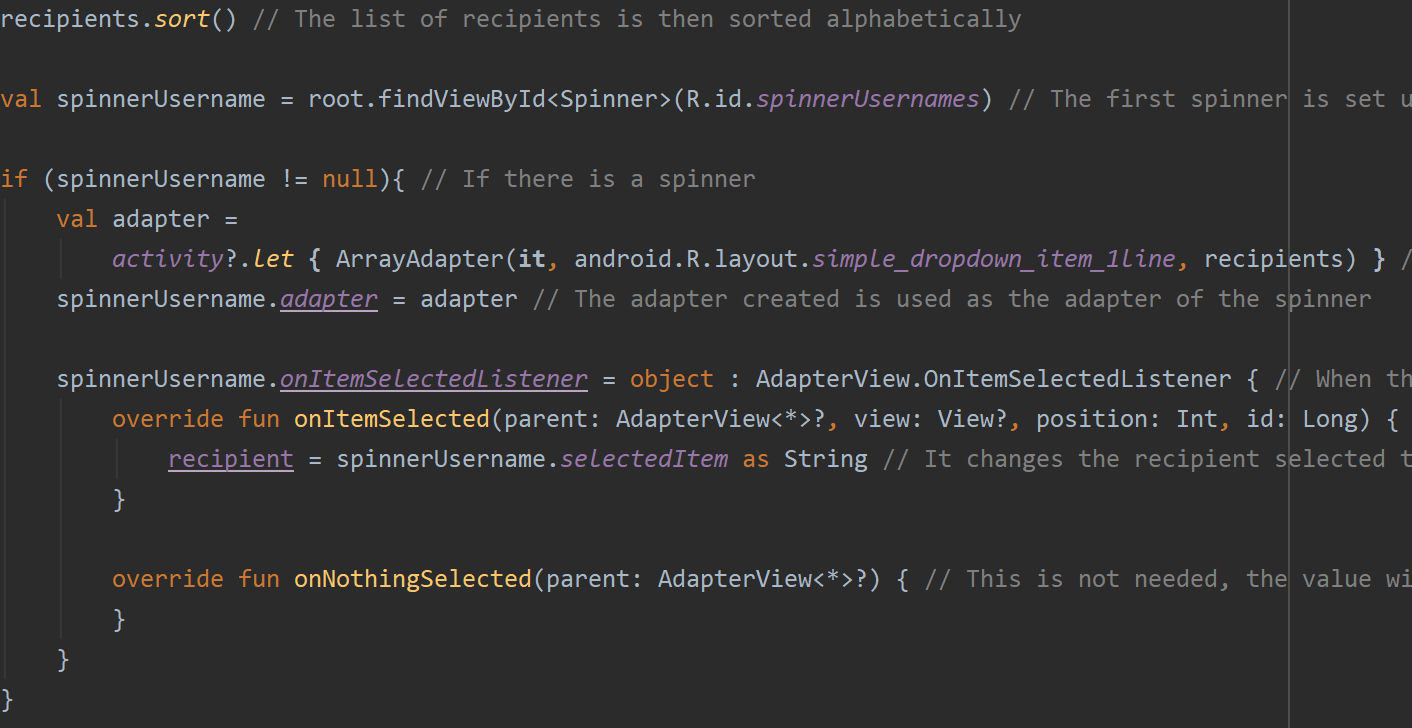
The first step in the communication branch was to set up the ping sending fragment. Before the fragment is set up, it is give unique global variables that can be used throughout the function and between functions



These three functions are defined in the companion object of the function and so are callable from any part of the function. These hold the variables of the 3 spinners that must be set up, in order to collect the 3 aspects of the ping that the user would send. These are set up in the order: recipients, ping types, locations. Unlike the other functions which initialised the variables, then performed operations, then handled leaving, this function will only initialise the spinner variables with their operations, in order to make the code more manageable and readable by giving it a more segmented view.

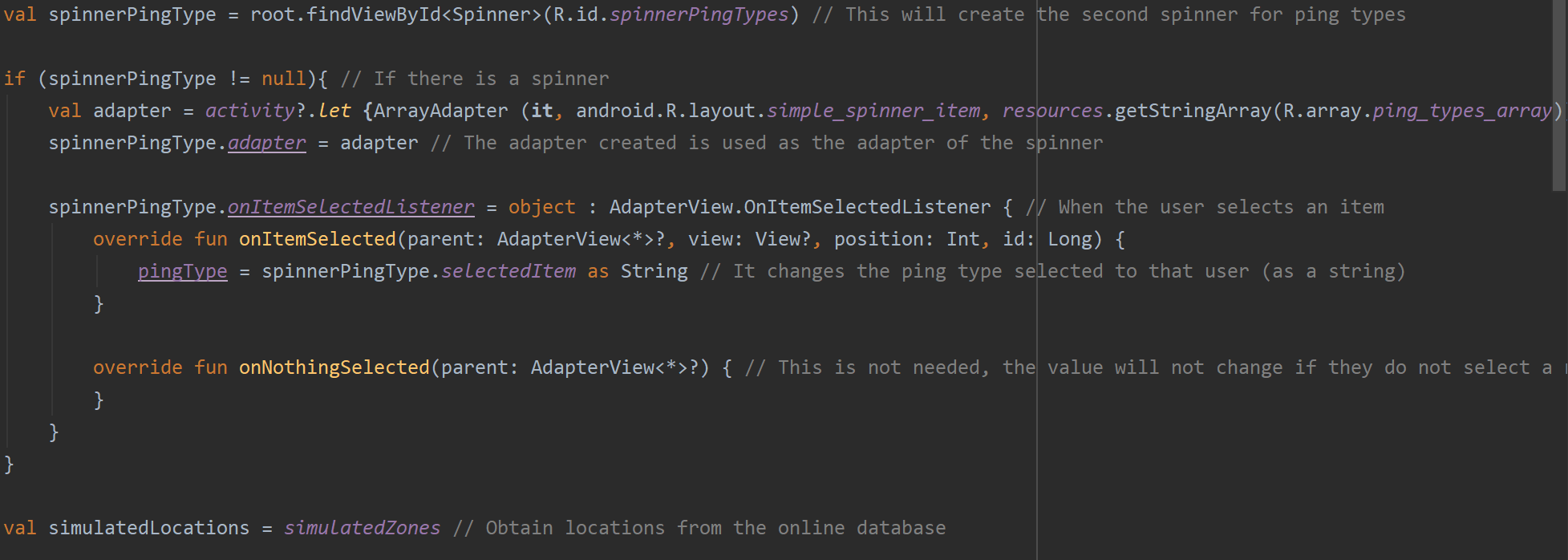


The first part of the recipient spinner is to generate the recipients that will go into the spinner. As with all the database queries, the list of users is taken from the database and then stored in a temporary variable. An empty variable is then created that will be given to the spinner to display. For each user on the database, if they are online and they share a store number they are added to the list of recipients. Although it may seem odd, as this would include the user’s own username, sending yourself a ping may allow the database to double up as a form of a to do list when tasks are given in real life and so this was kept to be this way. This increases the usability and versatility of the software and creates a neat hidden feature.

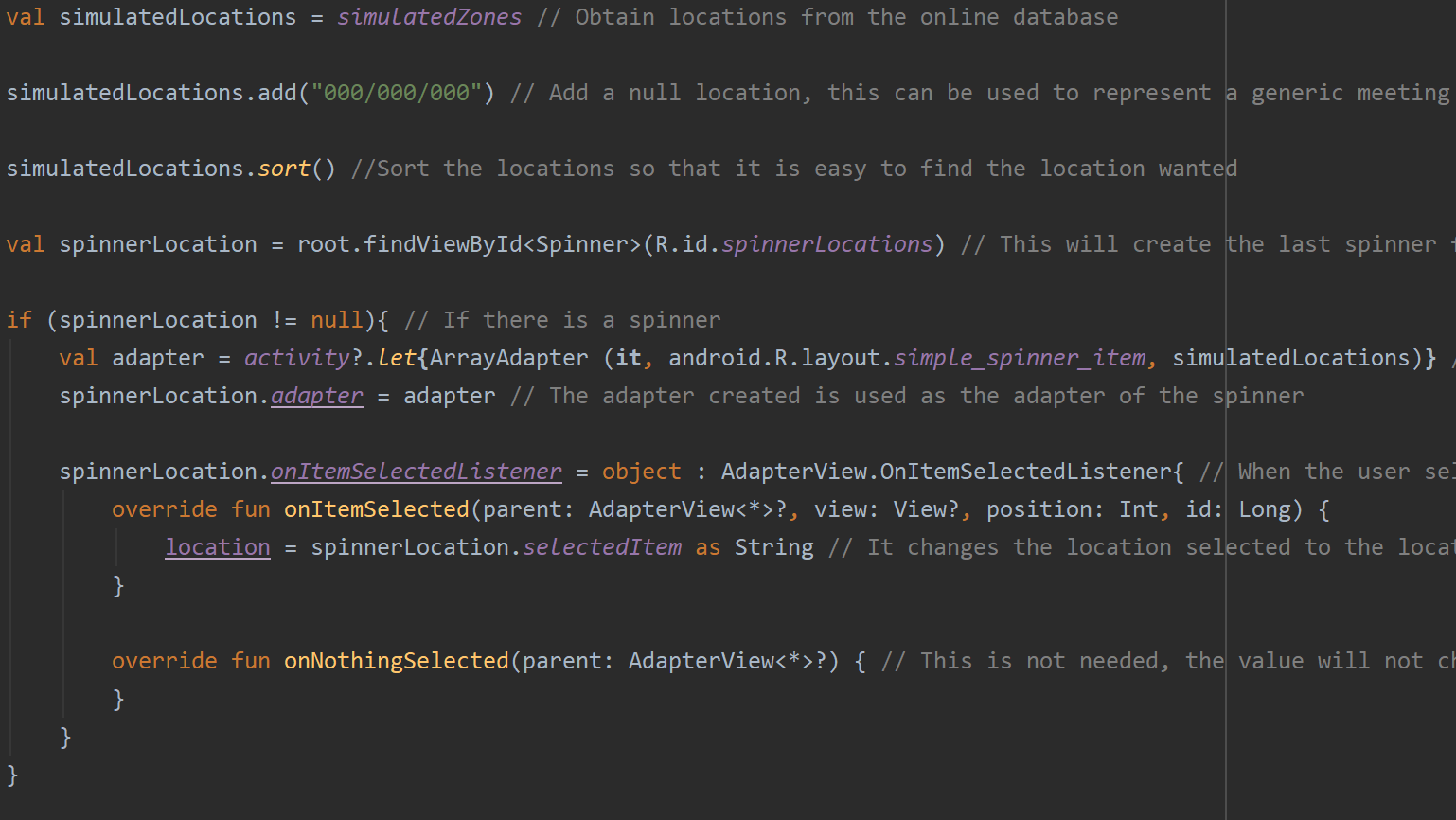


Before the array of recipients is used, it is sorted for usability, so that it is easier to find the username that you want. This is not needed by the program but has been included for the end user when there are large numbers of recipients. Next, the path of the first spinner is obtained from the layout declared earlier in the file. So long as there is a spinner, it will be set up by the program, where the adapter to be used by the spinner is made, passing through the current fragment as the context, the type of spinner needed and the recipients array generated earlier in the code. The type of spinner used here is different, as I found there was a spinner that used a greater spacing per item and have decided to use it here, to gather feedback on it when this iteration is shown to users. If they prefer this set up, then all of the spinners generated so far and in the future will be changed to this type.

Next, the adapter created is attached to the spinner, so that the spinner can use it on set up. Finally, the recipient is changed to be the current selected item, that will be changed whenever the user selects a new item.



The ping type spinner functions in the same way as the previously outlined spinner, the only difference is that it collects the string array that is uses for the adapter variable from the resources file. This is done to save on code space and not create unneeded variables that could break or accidentally interact with other parts of the code. The strings are kept in the strings file as it improves the maintainability of the code and means that any changes to the strings can be done independently of affecting the code and can be easily accessed and obtained.



The zone spinner works the same as the other spinners outlined, but the way it obtains its information is different. It does this by collecting the most recent list of zones from the zone database and adding a null zone to it. Next, it sorts the zones into numerical descending order to improve usability for the user. Then the spinner can be set up the same as the other spinners, the only difference being that it passes the list of zones as the string resource.

| **PingSendingFragment** | Variable | Input | Expected Output | Actual Outcome | Justification | Action taken |
| --- | --- | --- | --- | --- | --- | --- |
| Test 1 | Recipient  pingType  Location | A0B012C3  Pick  004/004/004 | A ping is constructed with the correct time, sender, recipient, type and location. This is then stored on the ping database and can be retrieved by A0B012C3 | Ping was not displayed | To check that the correct type of ping is sent and stored. | The storage for the pings (simulated pings) was changed from val to var so that it could be added to and changed.  The time was also wrong (12h format) so the sdf was updated (hhmmss -> HHmmss) |
| Test 1.2 | Recipient  pingType  Location | A0B012C3  Pick  004/004/004 | A ping is constructed with the correct time, sender, recipient, type and location. This is then stored on the ping database and can be retrieved by A0B012C3 | Ping was displayed | To check that the correct type of ping is sent and stored. | None |
| Test 2 | simulatedLocations | “000/000/000” | The list of zones is correct with the additional zone. | The zone was added each time the fragment was run. | To ensure that the zone is only added once | if("000/000/000" !in simulatedLocations) { simulatedLocations.add("000/000/000") }  The code was changed to ensure it was only added once by checking if it had already been added. |
| Test 2.1 | simulatedLocations | “000/000/000” | The list of zones is correct with the additional zone. | The zone was only added once | To ensure that the zone is only added once | None |
| Test 3 | sendButton | Click (nav) | The fragment would return to the communication menu | The fragment returned to the communication menu | To ensure it goes to the correct fragment | None |

**Midpoint Review:**

At this point in the development stage, half of the communication functions have been written. The prototype can now successfully add pings to the database of pings in a form that will not invalidate the database. All of this data has been formatted correctly for uploading and even been sent in the correct object type.

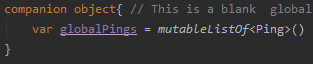
It has been thoroughly tested to handle all of the different instances it can take, where correct pings are sent, the navigational buttons are all properly connected and all of the inputs that should be available to the end user are available. This means it is now robust and finalised.

It also meets all of the checklist criteria and the general criteria set out in this report, although this will be commented on further in the final review.

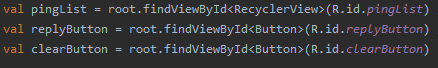
For now, it works enough to move onto the second part of this stage.

**Stage 5 - Continued - Ping Inbox:**

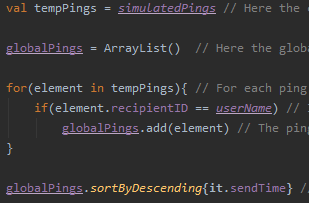
The first step of the development process is to set up any global variables that it requires, this is the key data structure that was defined earlier in the design process.



This is set up as a global ping array so that it can be validated in the ping inbox screen but also so that it can be passed to the adapter of the recycler view. The list is made mutable so that it can be changed when the user clears the list of pings.



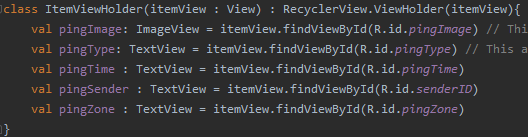
Next, the values of the paths for all of the variable views are obtained so that the correct outputs can be taken or so that the correct items can be input. Again, the procedure of global variables, view variables, validation or operation code and then button handling has been used here.



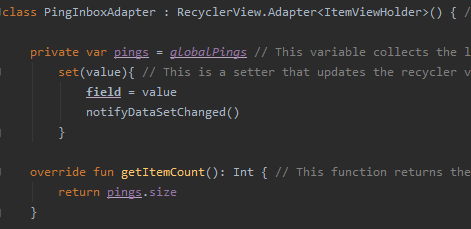
This is the next section of code that must be written, where the ping database is called to and the program stores the response in a temporary array. This array is then searched, ping by ping, for any that match with the user’s username. If there is a match, the ping is added to the list of pings belonging to that user. Once the list of pings has been searched, the list is sorted into order of when the ping was sent, so that the most recent pings are at the top of the list.



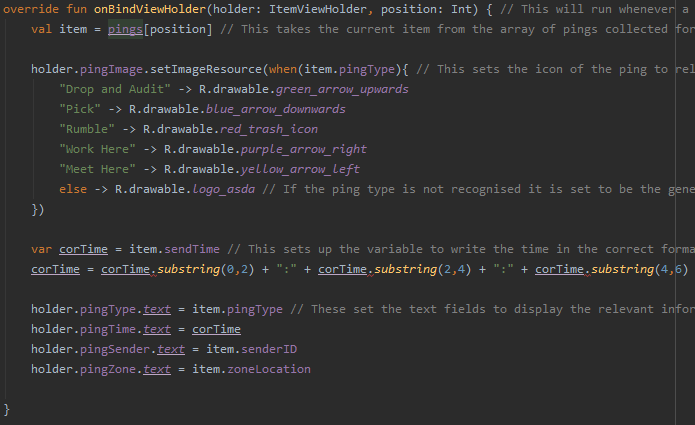
These two lines of code will create the adapter for the recycler view, so that it knows how many items to display, what type of items, how to set them out and what data to use. It will then attach this adapter to the recycler view for it to use. The PingInboxAdapter function is created in a separate file, to improve maintainability and the modular nature of the code.

****

Before the code for the adapter of the recycler view for the items can be written, the variables in each item view (each item in the list) must be created. This way the program can access all of these variable fields in each view and update them accordingly. Here, the text view that displays the icon of the ping and all of the fields for that ping’s information have been obtained. This means they can now be edited as each view is made. This is the same as how a fragment is generated, except each of these pings must be edited separately as they are created., as each have different values for their attributes.



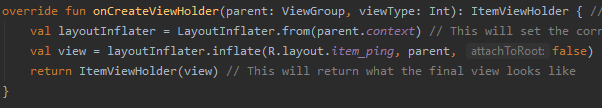
Now the recycler view can be created, where it is of the view holder created earlier, showing that it uses those views for its items. Here, we set up a private variable that copies the global ping array into local memory. This means that if internet connectivity was ever achieved, then it could be passed directly into it here without having to rewrite any code and the global pings array could be deleted without affecting the code. Next, a setter is written that simply sets the value of the item view that the recycler view is currently editing to the value passed in a setter and causes the recycler view to update. The final function is also simple, where the amount of items in the recycler view must be declared. Here, we simply declare it to be the length of the list of pings for the user, as this is the number of item views that we need.



Next, we write the function that edits the variables pathed in each item view, simply creating a new unique ping item whenever a ping item is created for the recycler view. Firstly, we obtain the current item from the ping list that we are creating a view for and save it as the current item. Next, we match up the ping type of the ping to the icon that must be displayed for that ping type. If the ping type is not understood (in future iterations, there may be more ping types) then the asda logo is used instead. This helps keep the program usable in the future and increases its compatibility with potential future iterations.

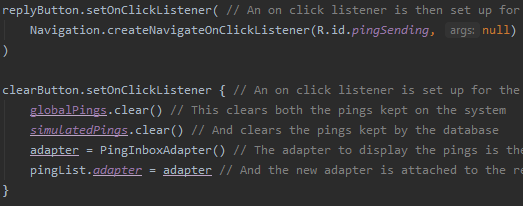
We also need to convert the string of numbers that represent time into its corrected form, that is easier to read. Here, the variable corTime (corrected Time) is set up and then colons are injected into the correct parts of the string. Again, the code does not need to do this but it is done for usability purposes and to make the time easier to read.

Finally, the text variable fields can be updated to reflect the relevant information from the database as well as the new corrected time information.



This is the last function that must be written, and simply unifies all of the components written above. This will create the recycler view as a whole and tie all of the unique item views into the scrollable list. Firstly, it gathers the fragment layout of the fragment that called this function (parent.context), this is done so that it can then inflate the correct item view for the recycler view and return the view that should appear in the recycler view. When this view is bound to the recycler view, it will then be updated with the previous function and the pings can be displayed in the recycler view. This is the end of the adaptor, which now lets the recycler know: how many views to generate, how to edit each view, the data to put in each view and can update when the information changes.

Now, the last section of the ping inbox can be written, where the listeners for each button will cause them to act out their functions when clicked.



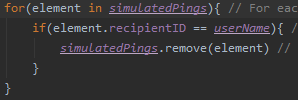
The first button, the reply button, is a simple navigational button used several times already throughout this development. It will navigate the user to the ping sending screen when it is clicked, so that they may send a ping of their own. The clear button has a more complex listener, where the local and “online” databases for the pings are cleared and then the ping inbox adapter is updated so that the recycler view can reflect these changes, by updating the adapter’s contents and then attaching the new adaptor.

| **PingInboxFragment** | Variable | Input | Expected Output | Actual Outcome | Justification | Action taken |
| --- | --- | --- | --- | --- | --- | --- |
| Test 1 | userName | B0B02U0 | Two pings are displayed with the correct information | The program crashes on accessing the ping inbox. | To ensure the ping inbox screen can correctly display pings on the database. | See below |

It was discovered that the ping inbox fragment would crash on start up due to the adaptor changing the contents of the time value. As there was no guarantee that corTime was not empty, and the text view cannot be empty, it would cause an error to crash the system. In order to avoid this, the variable corTime was changed to read var corTime = item.sendTime!! Where the variable corTime was now never empty because sendTime could not be empty. This will fix the crashes and the ping inbox screen should run normally.

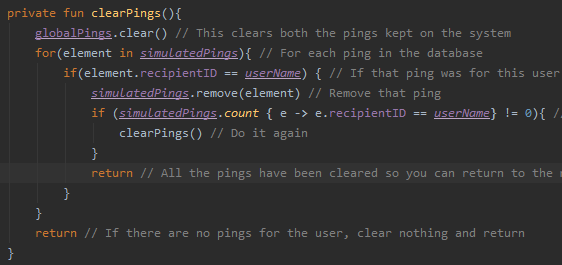
| **PingInboxFragment** | Variable | Input | Expected Output | Actual Outcome | Justification | Action taken |
| --- | --- | --- | --- | --- | --- | --- |
| Test 1.2 | userName | B0B02U0 | Two pings are displayed with the correct information | The two pings were displayed correctly. | To ensure the ping inbox screen can correctly display pings on the database. | See below |
| Test 2 | replyButton | Click | The user is taken to the ping sending fragment. | The user was taken to the ping sending fragment. | To ensure that the navigation of the ping sending fragment was correct. | None |
| Test 3 | clearButton | Click | The pings for the user was cleared | The pings for all users were cleared | To ensure that the clear button correctly clears all the pings | See below |

Here, it was discovered that the clear button would clear all of the pings for all users and not just the user that cleared them. This is because the line *simulatedPings*.clear() would clear all of the pings on the database. This was then taken out and replaced with the following code.

****

Now all of the pings in the global database that belong to the user are removed and not the ones belonging to all of the users.

| **PingInboxFragment** | Variable | Input | Expected Output | Actual Outcome | Justification | Action taken |
| --- | --- | --- | --- | --- | --- | --- |
| Test 3.2 | clearButton | Click | The pings for the user was cleared | The program crashed. | To ensure that the clear button correctly clears all the pings | See below |

The program crashed because the database was changed mid loop and it could not handle this change. Therefore, the new code was put into a function that could then call itself until the elements had been removed and then unwind once all the pings had been deleted.

| **PingInboxFragment** | Variable | Input | Expected Output | Actual Outcome | Justification | Action taken |
| --- | --- | --- | --- | --- | --- | --- |
| Test 3.3 | clearButton | Click | The pings for the user was cleared | The pings for the user are cleared. | To ensure that the clear button correctly clears all the pings | None |

**Review:**

What has been done:

In this portion of code, the communication branch of the device has been fully set up and written. The user can now send and receive pings correctly, as well as delete their own ping. Each ping type is fully set up to be unique, with identifiable logos used. This was achieved by accessing and modifying the database of pings that reflect what an online database would look like. This allows the users to communicate with each other more efficiently.

Prototype Summary:

Running the application causes the user to face a login screen. Correctly logging in will take them to the shopfloor menu. They can access a side menu to access the communication, backroom or shopfloor menu. They can use the power button at the top of this menu to log out. The shop floor menu has a scan button that takes the user to the barcode scanner. The barcode scanner will then collect the upc number input either visually or manually and the upc number then undergoes validation to ensure it corresponds to an item. If it doesn’t the user stays on the barcode scanner but if it does then the user can see the item info for that item. They can update the numbers on the shopfloor or in the backroom and can also choose to scan a new item. The communication menu has an inbox button that can only be accessed when there is a ping, that takes the user to **their ping inbox. They can then view their pings, reply to their pings or clear their pings**. It also has a ping sending button that takes the user **to the ping sending fragment. Here, they can choose the recipient, ping type and location and then send their ping to another user**. The backroom menu has a pick button that is only enabled when there are picks and takes the user to a blank fragment.

How it has been tested:

The functionality of the code has been tested by simulating access to the ping database and the sending of pings between users. This was done by simply testing all of the buttons and spinners to ensure they would function correctly under usage. This allowed me to test the criteria set out in the design checklist and the requirement success criteria.

The usability of this code was tested by allowing my manager (who is most likely to use this the most) to send pings and check his inbox on this part of the code. They were very excited by this, but commented that the larger spinner type used in the recipient list was the worst. Overall they agreed that the menu was very easy to use, free of any jargon and simple to understand.

How it meets the success criteria and user expectations:

The success criteria:

* Should always send the message to the user in under 30 seconds (wifi dependent)

Could not be met as there is no internet connectivity, but all other success criteria in this section was met.

The general criteria based on feedback set out in the analysis was also met, as evidenced by the testing with the use of human interaction with the system. Overall, I feel that the prototype fully meets all available criteria to the best of it’s ability at this point and this stage of the design is complete.

Changes in the design:

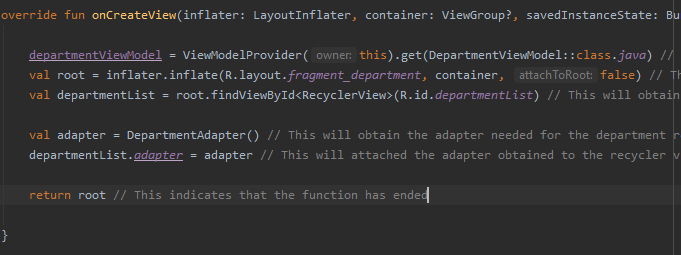
All the spinners will not use the spinner type “simple\_dropdown\_item\_1line” as this was found to be harder for the end user. There are no changes to the design.

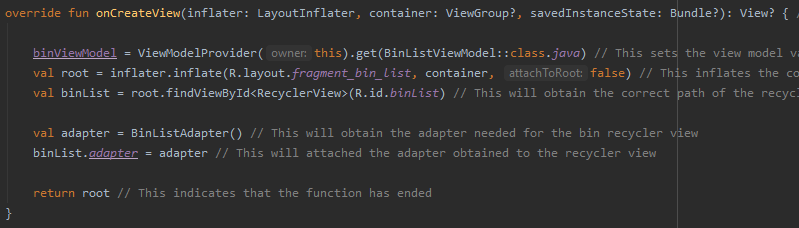
**Stage 6 - Backroom:**

This is the final stage of the design process, where the pick list must be accessible and then the filters and sorters are developed. However, due to time constraints and the project’s size being too large, I began to run out of time allotted to the development of this program. Luckily, the other stages of the development were complete and so would not be affected but to finish this stage in time (and the development of the solution as a whole), cuts had to be made.

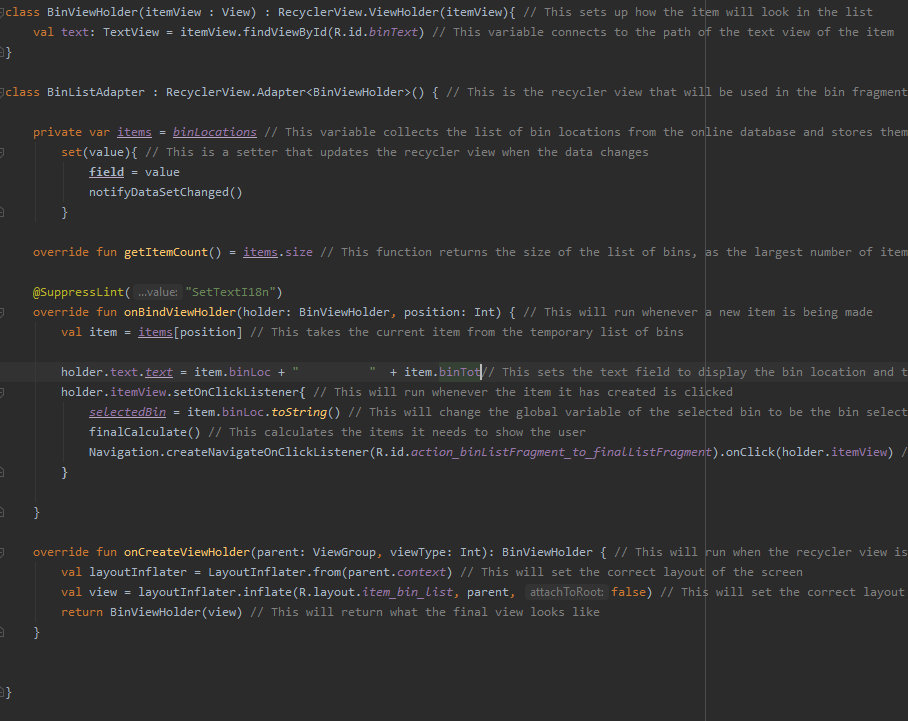
This section comprises two main features; the pick list which is an essential feature and the sorting capabilities which were not. In order to finish the project within the time allotted for the coursework, the sorting capabilities were dropped from the project overview and this stage looked solely to finish the pick list and thus end the project with a completed prototype. Although this prototype is not perfect, it was operational and would fulfill the success criteria to the best of my programming ability and the time frame allotted. This will be reflected in the review of this stage and the general evaluation at the end of the report.

Now that a recycler view has been created in the previous stage, it can be copied and reused to create 3 recycler views that feed into each other. The first two fragments and recycler views are the same, so developing one will allow the second one to be developed simultaneously. Their fragments are shown below:

****

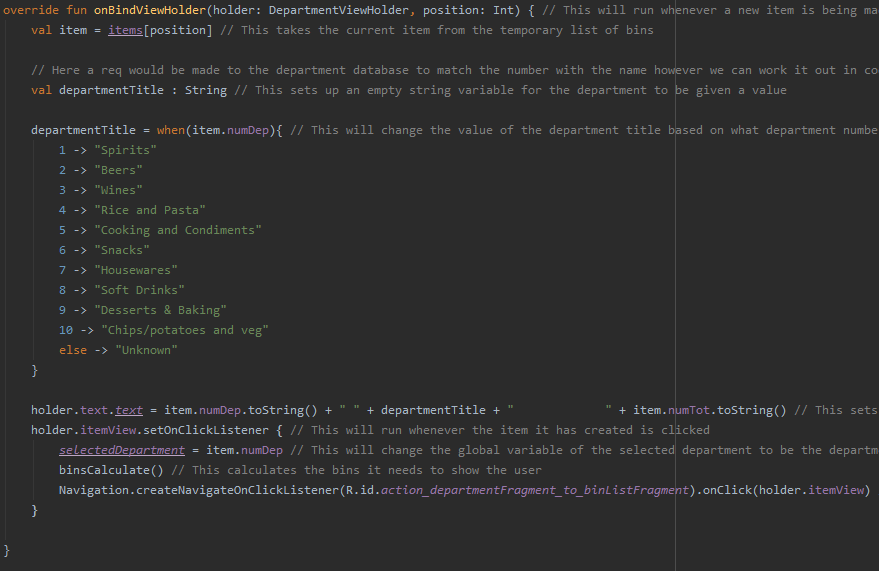
****

Here, both fragments will set up their corresponding view models, then bind their respective view layouts and obtain their recycler views. Both will then run their respective adapter functions before finally adding the adapter to the recycler view. As both fragments only display a recycler view, the code to create them was simple and could be copied over with only small changes to the variable names, in order to keep them sensible and correct.

****

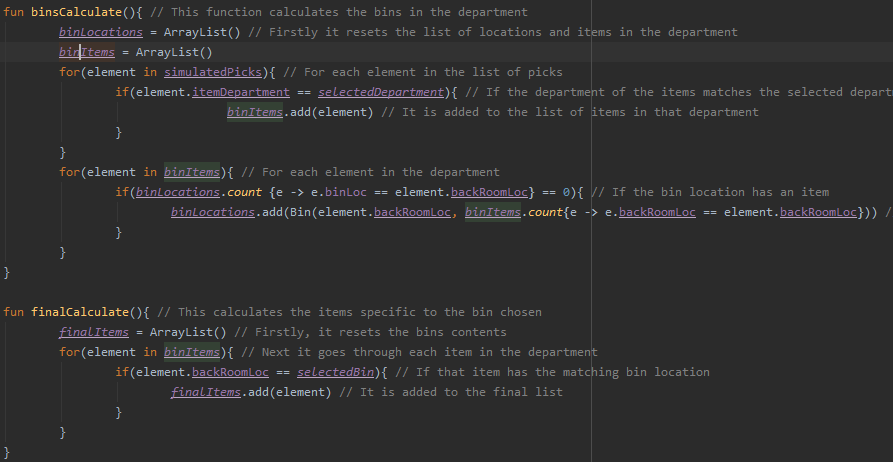
Now that one recycler view adapter has been made, it can be copied over and small changes to it’s variable names and functions can be made to tune it for the bin list. Here, the first class of code will define the variable fields inside the item display. Only one text field is shown and so has been named text for simplicity. Next, the adapter grabs the list of bins and their bin numbers generated by the binsCalculate function (that has not been written yet). The final function onCreateViewHolder, is also a copy of the initially created recycler view. The major difference is the contents of the onBindViewHolder function, which again takes the current bin to be displayed and sets it’s unique item. It sets up the text view to display the bin’s identifier and the number of items in that bin. It also sets up a unique on click listener for that bin, so that when it is clicked, the selected bin becomes that bin, it calculates the list of unique items in that bin and then navigates to the empty item view list.

The same process has also been carried out for the department recyclerview adapter.

****

The only difference between these two adapters (other than their variable names reflecting which fragment they belong to), is their onBindViewHolder function. Here, the department title is set up to be a string variable and then changed to reflect which department number the item has. If the departmentTitle cannot be recognised (perhaps in the future there are departments added), then the department is set to be unknown. This is so that the recycler view does not crash on handling new data and the program can continue to be compatible even when outdated in the future. Then, the department number, title and number of items are displayed to the user. As with the previous adapter, spaces have been inserted to separate these values and make them easier to read for the end user. Finally, a similar on click listener is set up that also calculates the items for the next screen (the bins) and then navigates to that screen.

So that these can be tested, the functions that calculate the bins and the items must now also be written.

****

The first function deals with first calculating all of the items in the bins as a collective. It achieves this by first resetting the current lists of bin objects and items in the bin objects. Next, it takes the simulated list of picks and calculates which objects belong to the selected department. If it does, it then adds it to the list of items in all of the bins. Next. it takes the list of all of the items in the bins and checks to see if the bin location of that item has been added to the list of bin locations already. If it hasn’t, then it adds a bin object to the list of bins for that bin location, consisting of the bin identifier and the number of items in the list of items in the bin that matches that bin's location.

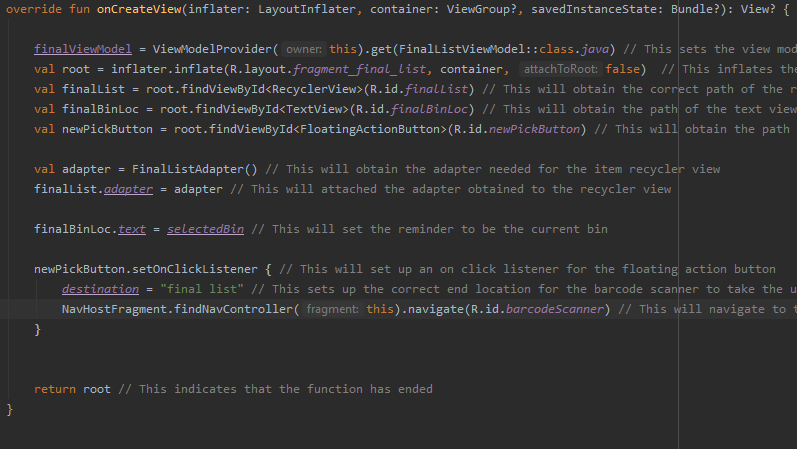
The latter function deals with calculating all of the items in that specific bin and adding them to a list of items in that bin. This is done by first resetting the items in that bin, then for each element in the bin list calculated earlier, it checks to see if it belongs to the selected bin. If it does, it adds it to the list of items in that bin. As the finalCalculate is not done without the binsCalculate being done immediately before it, it can be sure that the binItems is as updated as possible and reuse the assets from that function, as bin items is a global variable. It does not use the simulated picks list because although the bin zones may match, the departments may not and so it is easier to take the already thinned list and perform only one check. Currently, the ASDA system does not do this and so I consider this to be an improvement on the current system, where incorrect logs can cause random departments to appear in the pick list and slows down the work.

**Midpoint testing:**

At this point, the recyclerview’s onclick listeners were checked to ensure that they would work and navigate correctly. This was not put into the table because, at this point, the navigation system and listeners had been reused so much that it was impossible for them to break, but it has been mentioned here so that it is reported that they were tested. It is much more important to check that they update correctly rather than that they work, which is what will be tested at the end of this report.

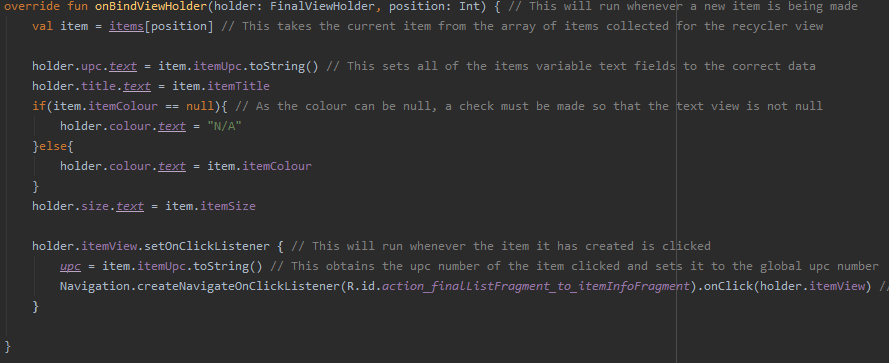
**Stage 6 - Continued - Final Recyclerview and Item Deletion**

The final section of this development stage is to set up the final recycler view for the items that can be picked off and to handle the depart function when the destination for the barcode scanner is set to be the pick list.



This is the code written for the final fragment in the backroom branch (and also in the code). This last fragment operates in a similar fashion to the two prior, other than two new pieces of code. These two new parts are: the line that sets the text of the heading at the top of the layout to the currently selected bin and the line that sets an onclick listener for a floating action button, or FAB. A FAB was used here as the list of items can often be long and this button will remain in place while the user is scrolling through the list, without obscuring items that may need to be viewed. The fab is set to correctly change the destination variable for the depart function in the barcode scanner and then navigate to the barcode scanner.

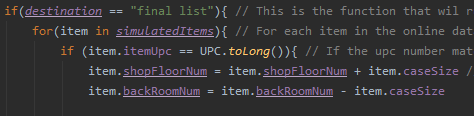
Before we can write the depart function, we must first write the adapter for the final list recycler view. This recycler is a copy of the other two, tested, recyclers and so testing for this section is not required to be explicitly shown in code, as it was copied and reused.



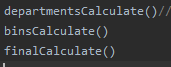
The only change is made on writing the code for the item that is displayed. Here, the item’s information is set to fill the text views in the recycler view item. A check is made to ensure that the colour of the item is not null, as it can be if the item has no colour and the text view cannot be null. In this case, a filler “N/A” text is shown instead in order to stop crashes from occurring.

Next, a simple onclick listener is set up for the item view, so that when it is clicked it sets the global variable for the upc number to the upc number of the item clicked and then navigates to the item information fragment for that item.

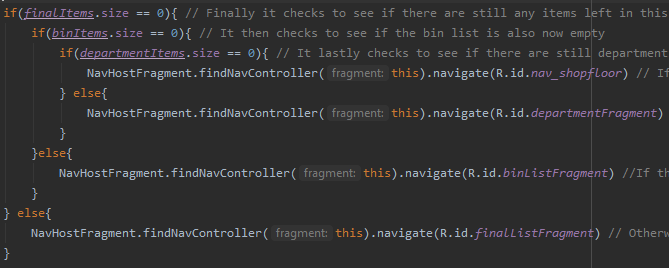
Lastly, the depart function for the barcode scanner must be written. This is the final function of code that must be written before the prototype is complete.



Firstly, the code checks each element in the list of simulated items to find the upc number of the item it is updating. If this item is found, then it adds a case of the item to the shop floor and takes away a case from the backroom.



Then, it recalculates all of the items in the departments, then the bins and then the final list of items. This is done to update the temporary arrays on the device, as the “online” database has been updated.



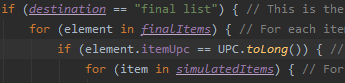
Next, it decides on which fragment to send the user. If there are no pick items left, they go back to the shopfloor menu, not the backroom menu as this would be pointless (because it’s only button is disabled). If there are no more items left in that department, but still items left, then they go back to the list of departments. If there are no items left in that bin, but items left in the department, then they go back to the list of bins in the department. Otherwise, they are returned back to the pick list.



Finally, if the item cannot be found then the correct message is displayed to the user, that their item could not be found and they are returned to the list of items, so they can ensure the item that they are scanning is the correct one. The code for the system is now complete.

| **Communication Branch** | Variable | Input | Expected Output | Actual Outcome | Justification | Action taken |
| --- | --- | --- | --- | --- | --- | --- |
| Test 1 | simulatedItems | Upc number on the pick list | The item is removed from the list and the list is updated | The item was removed from the list and the list was updated | To ensure the picks would be scanned off when the upc number was correct | None |
| Test 2 | simulatedItems | Upc number not on the pick list | The item is not removed from the list and the correct message is output. The user is returned to the pick list. | The item was removed from the list and the list was updated. | To ensure the picks would not be scanned off when the upc number was incorrect | See below |

At this point, it was noticed that the upc number scanned would always be correct, regardless of if it was on the pick list or not. This was a problem, as it meant items could be added when there were none in the back and negative numbers could occur. To combat this, the code was changed to check if the upc number was on the temporary list of items and then if that was true, it would find the upc number in the global database and change it there.



This was accomplished by adding a simple check that would search the list of final items until the desired item was found, or until it was certain that it was not in the list and so would reject the item.

| **Communication Branch** | Variable | Input | Expected Output | Actual Outcome | Justification | Action taken |
| --- | --- | --- | --- | --- | --- | --- |
| Test 1.1 | simulatedItems | Upc number on the pick list | The item is removed from the list and the list is updated | The item was removed from the list and the list was updated | To ensure the picks would be scanned off when the upc number was correct | None |
| Test 2.2 | simulatedItems | Upc number not on the pick list | The item is not removed from the list and the correct message is output. The user is returned to the pick list. | The item was not removed from the list and the correct message was output. The user was returned to the pick list. | To ensure the picks would not be scanned off when the upc number was incorrect | None |

**Review:**

What has been done:

The final branch of the solution, the backroom branch, has been coded. This allows the user to break down the list of items to be picked into department and bin specific lists that are more easily manageable. From here, they can click on the item to go to the item info screen for that item or they can use a floating action button to scan an item to be taken off the picklist. This will automatically adjust the numbers of that item for the user. I achieved this by using the adapter for the recycler view that was made earlier in the development cycle and reusing it, changing only a few key variable names and the set up of their items per adapter. I also used the navigational map that was tried and tested at this point to connect the items to their end destinations. It uses global variables for each of the sublists that are generated to create the lists that the end user sees, as well as global functions to generate these lists.

Unfortunately, the custom filters and sorting was not implemented due to time constraints.

Prototype Summary:

Running the application causes the user to face a login screen. Correctly logging in will take them to the shopfloor menu. They can access a side menu to access the communication, backroom or shopfloor menu. They can use the power button at the top of this menu to log out. The shop floor menu has a scan button that takes the user to the barcode scanner. The barcode scanner will then collect the upc number input either visually or manually and the upc number then undergoes validation to ensure it corresponds to an item. If it doesn’t the user stays on the barcode scanner but if it does then the user can see the item info for that item. They can update the numbers on the shopfloor or in the backroom and can also choose to scan a new item. The communication menu has an inbox button that can only be accessed when there is a ping, that takes the user to their ping inbox. They can then view their pings, reply to their pings or clear their pings. It also has a ping sending button that takes the user to the ping sending fragment. Here, they can choose the recipient, ping type and location and then send their ping to another user. The backroom menu has a pick button that is only enabled when there are picks and takes the user to a **list of departments that have picks. This list of departments can then be clicked on to access their bins. These bins can then be clicked on to access their items. These items can then be clicked on to access their information screens. A floating access button can also be clicked on, that takes them to a barcode scanner. When an upc number is input that matches an item on the pick list, a case of that item is moved to the shopfloor and the lists are updated. The user is then taken to the lowest list in the chain with items, or to the shopfloor menu if there are no more items**.

How it has been tested:

The navigational components were tested just by using the system, where I found play around with setting the numbers in departments, bins etc to be zero and track the results. These were not explicitly recorded, as at this stage I had tested the navigation system numerous times and so it was almost guaranteed to work.

The database components were tested by directly accessing them and modifying them, with attempts to incorrectly and correctly access them made. Once the correct outputs from each of these attempts were achieved, the validation was certain to be correct and this was considered tested.

The usability component of the device did not need to be tested, as this is a direct clone of the original system’s layout. This meant that I knew people could use and operate it and that it was as free from jargon as possible.

How it meets the success criteria and user expectations:

Unfortunately, requirement 6.2 from the list of requirements set out in the analysis portion of the project was unable to be met. This was disappointing for users, but many pointed out that the lists were often very small and so this was not essential for the final product. One user even pointed out that many people prefer to see the list of all of the departments so that they know which departments they need to go to next but I am still disappointed with my mismanagement of time.

Despite this, all the other requirements set out in the requirements table, the general criteria based on feedback and the design checklist have been met and so this can be said to adequately satisfy user needs, at least the very essentials, even if it does not do it to the standard I wished it to accomplish this at.

Changes in the design:

Unfortunately, time constraints meant that the custom sorting and filters could not be implemented. No other changes were made to the design.

**Evaluation:**

**Testing (Post Development):**

The first step of the evaluation is to carry out post development testing in order to accumulate evidence in order to form a conclusion on the project's success. I started by carrying out typo checking, in order to improve usability and ensure the finish project looked polished. This was done by checking the strings in the string file, which I had set up for this very reason to improve maintainability, but also the navigational map so that the headers of each fragment were not their file names.



Here the main menu string was missing a space, and the string was incorrectly named. This was changed to:





The header of this fragment was hard encoded into the attribute, so was changed to reflect a strings file location.



Now, the walkthrough test of the device could be carried out. This was outlined in the design phase, where each branch of the program will be tested by using the device and thus the functions and usability of the device could be measured. The link to this video is at the end of this project, after the bibliography and app link.

Next, a final test of all the test tables was undertaken to ensure that the navigational map was operational. This was done by accessing all of the buttons when they were not disabled and pressing each, ensuring I ended up at the correct destination. Finally, the button's conditions were set so that they would be inactive and they were tested again. Each one was found to be inactive when it’s conditions had been met.

I then restarted the program to test the database one final time. Each item was checked by swapping their values to have no cases in the back room (or no space on the shopfloor). The pick list was now empty and I ensured that the button was not turned on for the pick list. Next, the items were all moved to the backroom (where possible) and I navigated the pick list to check each navigational component. They were all found to be operational.

Lastly, I checked the ping sending and receiving fragments to double check that they correctly sent, received and deleted pings. Fortunately, this code had not changed since the first prototype and so it was still fully functional.

At this stage, syntax and functionality had been fully tested and I could use the test data from the iterative process to prove that each function was fully operational and polished in appearance. The next stage is stakeholder testing.

**Stakeholder testing:**

For the stakeholder testing, I sent my stakeholders and students at my school who had never used the asda devices copies of the app, barcodes and user details so that they could try the functions. I told them they could do whatever they wanted on it, so long as they carried out specific tasks, with each task I assigned someone who was already familiar and who was not familiar with the asda system. I told the people who were already familiar that I would not help them at all and the people who were not familiar I gave a quick outline of how asda operated (what the zones meant, what a upc was etc) but not how to operate any aspect of the device. The tasks I gave them were divided into each branch, so that one group tested the functions of each branch. I then gave the application to my manager, who I asked to test all 3. The tasks were:

* Log in to the device and send someone else a ping (I ensured at least 1 other user would always be logged in). Ask them to rumble aisle 3 and then meet you in the backroom, location 2. Then log out.
* Log in to the device and find out all the information you can on Spaghetti. Then, move a case onto the shopfloor. Then log out.
* Log into the device and pick off all the items in the pasta department. Then log out.

Task 1:

Both the users, familiar and unfamiliar, would not understand how to send a ping initially. This is because the ping sending feature is new and so the experience with the system will not help.

Unfamiliar user:

This user struggled with logging on, as they could not work out the difference between store/club and store/club Pharmacy and Optical but eventually logged on. This was not used as evidence and is instead a problem with the current asda system. They then were unsure as to what a ping actually was, but explored the menus until they saw the button labeled “send a ping”. They could not figure out why their inbox was greyed out but assumed they had no mail. They then sent a ping to themselves, to rumble aisle 3. When they saw the ping in their own inbox, after it was no longer grey, they realised their mistake. They then went back and sent the two pings correctly. They then correctly logged out, after noticing the login button at the top of the menu rather than an option of the navigation. They remarked that the button was “fun to use”.

Familiar user:

This user logged in easily, although accidentally put in their own account details initially. They remarked they “really need to get a new job”. They then immediately went to the communication tab as this was the tab that was unfamiliar to them. They remarked that the ping inbox button was not turned on and asked if I had not coded the menu yet. When I informed them that I had, they said they “must not have any jobs yet” and sent off the two pings. When trying to log out, they looked for the log out option on the side of the menu, then went to the shopfloor menu to look for it. When they did not find it there, they noticed it at the top of the menu and logged out successfully. They marked that they “much preferred the large button although they didn’t notice it at first”.

Manager:

The manager immediately logged in and then located the send a ping button. They liked that “this seems like something I won’t have to train people on” and were able to send pings very quickly. They remarked that they would “be able to send pings very quickly after no time” and “I can’t wait to put my feet up and send out a bunch of these from a nice office.” They experienced no troubles and “I don’t care if there’s an inbox or not, I won’t use it.”

Task 2:

This task was more suited to the familiar user and manager, who I wanted to comment on if they thought it was easier with this system or the old system. With the unfamiliar user I wanted to test how easy it was for them to find the item screen.

Unfamiliar:

This user immediately logged in and found the scanner button, they commented “that was easy” when the first button they saw was the button that said “scan an item” and liked how centralized it was on the screen. Next, they choose instead to input the upc number manually stating “I can’t be bothered to find the barcode” and found the spaghetti item. They did not check if they had the correct item, but instead clicked the button to add a case to the shopfloor. When it became greyed out they said that “it must be loading hold on” and waited for a minute before realising it was grey because they couldn’t add any more. They then went back to the main menu and promptly logged out.

Here it was noted that the system was very easy to use for an unfamiliar user however the buttons could be updated to reflect their status on the label rather than just becoming disabled. If this is a consistent problem with users, I will add this as a usability feature.

Familiar:

This user took a while to log in, as they kept entering their own store number rather than the store number I made up for the system. Next, they initially checked that they were on the shopfloor fragment before noticing the button to scan a product. They then scanned the item’s barcode and immediately scrolled down (as this is what they were used to on the previous system). Once they noticed the buttons to add to the backroom and store front, they pressed the button once before closing the fragment and leaving.

Here they said that the system was a lot easier to use than the previous system, they did not have to specify how big a case was before it was added and the buttons were a lot larger than the old system. When asked about the greyed button, they just said that “well there was nothing in the backroom anymore” and said that “it was disabled because there was nothing there”.

Manager:

This user was already logged in and did this task after the pinging task. First, they accessed the navigational menu and went straight to the shopfloor. When they saw there was only one button they “didn’t even read what it did, I just knew” and then promptly scanned the correct barcode. Like the familiar user, they then scrolled straight down and pressed the add to shopfloor button before finally returning to the main menu.

This user said that “it felt the same, although the buttons were clearer” when compared to the current system. When asked about the grey buttons they said that, “I didn’t even notice it was disabled. I wouldn’t wanna do it if I know nothing is there, so I didn’t even look.”.

I feel the results of testing how users found the grey buttons were inconclusive so I tested this with multiple more users. Overall, the feedback was that the grey buttons were “weird the first time but after a few seconds I understood that whenever anything was grey, it just meant it couldn’t be used anymore for whatever reason.” Because of this, I decided not to add the change in labels, as I feel it could ruin the subtlety of the buttons and draw users to them when they shouldn’t be considered as the program is in use.

Task 3:

This task was more suited to the familiar user and manager, who I wanted to comment on if they thought it was easier with this system or the old system. With the unfamiliar user I wanted to test how easy it was for them to complete the picks for pasta.

Unfamiliar:

This user successfully logged in but was stumped by the term “pick”. They looked through the menus one by one (shopfloor then backroom) until they saw the pick button. They then used their common sense and figured that the pasta was in the rice and pasta department. They then were confused by the bin section, asking me “I don’t actually gotta go to these places, right?”, I responded with “If they were there, yes. But they’re not so no.” they then successfully picked off the first bin and were then stumped for a bit as to why there was a second. Realising it was spread out over locations, they scanned off the final item and were returned to the department list. They were surprised by this, but after realising that the rice and pasta was not there anymore, they logged out.

They commented that the barcode scanner was easy to use and that they mostly understood what was happening at each stage of the process, but would probably understand more if they were actually using this in the store.

Familiar:

This user had a small amount of problem logging in, where their device was too small and they did not realise that they had to scroll down to get the login button. After realising this, they then used the rest of the pick list without problem, using their prior experience to achieve all of the tasks.

I was surprised to find that they managed the swaps between screens without confusion, as this is not what happens in the current system, but they remarked that they “did not mind this change as it allowed them to focus on the next step rather than cleaning up for the prior step”.

Overall, they agreed that the new system was easier to use than the current one as it removed redundant steps.

Manager:

This was the final task for the manager, who did not have much to add that the familiar user did not. They had nearly the same experience, minus the login stage and the same comments were made, regarding no longer having to go back to the previous stage.

They also agreed that the new system was easier to use and quicker than the current system

Review:

In conclusion, this series of tests has allowed me to gather evidence to satisfy the more general parts of the success criteria, set out in the analysis.

The feedback gathered from users unfamiliar with the current system were taken to see if the device was easy to use, free of jargon and that each screen was set up in a way to make sense to users who had not used it before.

The feedback gathered from users familiar with the current system, and managers, was taken to see if the device was faster than the current system, that the knowledge and experience gathered on the current system could be applied to this new system and to see if they thought it was usable in the stores.

Overall, the feedback obtained from these post development tests clearly show that it can be used as evidence to meet the criteria set out in the analysis, as both the unfamiliar and familiar users struggled little if anything with the new system and could easily perform tasks they may have never done before.

**Criteria Met:**

| **Requirement No** | | **Description** | **Success Criteria** | **Met?** | **Evidence** |
| --- | --- | --- | --- | --- | --- |
| 1 |  | Log in screen |  |  |  |
|  | 1.1 | Username box | * Username is rejected when it can’t be found * Should be labeled “Username” and be a text input field | * Yes * Yes | * Iterative Testing Data (ITD) * ITD & Video |
|  | 1.2 | Password box | * Password is disguised * Password is rejected when it doesn’t match * Password should be between 6 and 14 letters * Should be labeled “Password” and be a text input field | * Yes * Yes * Yes * Yes | * ITD & Video * ITD * ITD * ITD & Video |
|  | 1.3 | Store Area | * Should be a drop-down menu of 4 options * Should be rejected when it doesn’t match | * Yes * Yes | * ITD & Video * ITD |
|  | 1.4 | Store Number | * Store is rejected when it doesn’t match * Should be labeled “Store Number” and be a text input field | * Yes * Yes | * ITD * ITD & Video |
| 2 |  | Working barcode scanner | * Should scan the correct barcode everytime * Should be able to be manually inputted or visually scanned | * Yes * Yes | * ITD & Video * ITD & Video |
| 3 |  | Pinging System | * Databases should be fully normalised * Databases should not break when modified * Data should only be stored if it is relevant | * Yes * Yes * Yes | * N/A * ITD & Video * Data is only changed when changed by the user explicitly and all data stored is used. |
|  | 3.1 | Ping sending screen | * Should have a simple design * Should always send the message to the user in under 30 seconds (wifi dependent) * Should only list online users from their store * Should accurately update when loaded | * Yes * No * Yes * Yes | * Post-Development Data (PDD) * Wifi connectivity was not possible * ITD & Video * ITD & Video |
|  | 3.1.1 | Recipient input | * Should display online users in a drop down menu. * Should not allow the user to send messages to users that aren’t in the same store as them. | * Yes * Yes | * ITD & Video * ITD & Video |
|  | 3.1.2 | Ping Type Input | * Accepts all inputs * Should display options in a drop down menu. | * Yes * Yes | * ITD & Video * ITD & Video |
|  | 3.1.3 | Zone Location Input | * Accepts all inputs * Should display options in a drop down menu. | * Yes * Yes | * ITD & Video * ITD & Video |
|  | 3.2 | Ping receiving screen | * Should always receive any sent messages in order of newest first * Should only load messages for the user * Should display the appropriate message if the user has no pings * Should allow the user to easily respond by taking them to the ping sending screen if they wish to reply. | * Yes * Yes * No * Yes | * ITD & PDD & Video * ITD & Video * Instead, the ping inbox is greyed out. Although this is an indicator, it is not a message. * ITD & Video |
| 4 | 4.1 | Home Menu Screen | * Should show all the menus available on the device * All the menus should be accessible * Should be easy and simple to understand | * Yes * Yes * Yes | * ITD & Video * ITD & Video * PDD & Video |
|  | 4.2 | Side menu screen | * All the menus should be accessible and correctly labeled | * Yes | * ITD & PDD & Video |
| 5 |  | Item Information Screen | * Only the desired information should be editable * All the information displayed should be up to date * The correct item should be found every time | * Yes * Yes * Yes | * ITD & PDD & Video * ITD & Video * ITD & PDD & Video |
| 6 |  | Picking Screen |  |  |  |
|  | 6.1 | Displayed list of departments -> bins -> items | * All sections, and only those sections, of each category are shown if they have been chosen by the picking algorithm. * Each category is correctly linked * They are shown numerically * Buttons to move to the next screen are clear | * Yes * Yes * Yes * Yes | * ITD & PDD & Video * ITD & PDD * ITD & PDD & Video * ITD & PDD & Video |
|  | 6.2 | Filterable bins that can work positively or negative on the list of departments | * Should correctly save the filters to the user and only be accessible by that user * All filter lists should be accessible until they are deleted * Filter lists should correctly store any data inputted by the user for them * Filter lists should always be applied even if some of the departments specified aren’t on pick list | * No * No * No * No | * This section was unable to be developed due to time constraints |
|  | 6.3 | Can jump to the item information screen | * Should correctly link back to the item information screen for that item * Should be able to return back to the pick list for the bin * Should be accessible for all items | * Yes * Yes * Yes | * ITD & Video * ITD & Video * ITD |
|  | 6.4 | Item scanning/ manual input to pick the item off the list | * Should scan the correct barcode everytime * Should be able to be manually inputted or visually scanned * Should work for all items and only modify the database for that item | * Yes * Yes * Yes | * ITD & PDD & Video * ITD & PDD & Video * ITD & PDD & Video |

**Overall, all of the criteria but one has been met and the project can be considered an overall success.**

| **General Success Criteria** | **Met?** | **Evidence** |
| --- | --- | --- |
| The UI should be very simple to use | Yes | Post development tests with the familiar and unfamiliar users show that it was simple.  Stakeholder feedback in the design stage shows that it was understood to be simple to use. |
| The UI should be quicker and more effective than the current system, providing all the same functionality as well as new functions and simplifying steps | Yes | Post development tests with the familiar users showed that they were able to complete tasks quicker than they could on the current system. |
| The software should be able to correctly scan barcodes and communicate with an online server | No - Only partially | The barcode scanner was correct, as shown in the iterative test data and partially shown in the post development user tests.  However, online functionality could not be met so this criteria was only partially met. |
| All parts of the code should work independent of each other  -The code should be robust, commented and adaptable | Yes | This is shown in the iterative testing, where each part of the code that was written was fully annotated and tested to work.  The code was written in stages, so that each function of code was independent of the other stages.  Functions can be added in the future due to the menu layout designed, so it is adaptable. |
| The code should work on any android device | Yes | Post development testing by sending different people the app to use on their personal device shows that it is usable across all android devices. |
| The code should be streamlined and one screen should lead smoothly into the next | Yes | Each fragment uses the navigational map set up in the iterative development stage, to swap out fragments smoothly and it lead to a smooth experience in the post development testing with the users. |
| Labels should be jargon free, so that they do not complicate functions for the user | Yes | This is shown in the post development stakeholder tests, where the familiar users did not even have to pay much attention to understand the functions. This is also shown with the unfamiliar users being able to use the program without understanding the functions, as their labels were jargon free. |

Overall, I feel that I met the success criteria to an acceptable amount. Other than the internet connectivity, and the requirement 6.2, all of the success criteria was met in full. In addition, the essential features of the design were met in full. The device has also been set up in such a way that these features can easily be added in the future and so I am not worried that they have not been met in this iteration of the program.

**Usability Features:**

* The buttons and spinners used in the program for inputs are very large, colour cohesive and are labeled correctly and without jargon. This makes navigation of the device very easy and it is easy to understand which function is stored where. Evidence of this includes how easy it was for the stakeholders to use the device in the post development testing.
* The pinging system can also be used as a to do list. By pinging themselves, their own private to do list can be set up and they can keep track of their own tasks.
* Pings are easy to understand and text fields are large and cohesive. This was achieved in the design section and partially in the iterative testing section, where the pings were shown to stakeholders to ensure that the stakeholders could quickly understand what each one meant and what each text field meant.
* All of the text input fields and spinners are labeled correctly so that the user understands what they do. This is shown using unchangeable text fields in the layout and helps the users understand what each input will do.
* All of the data validation and database handling is carried out by the device, so that the user only needs to focus on what they should do and not how it should function. This is shown by some user’s ability to ignore parts of the software when testing in the post development testing.

**Limitations:**

* No printer connectivity - I would not be able to test this part of the development out as I do not have access to a portable, hand held printer. If given more time or resources, I could have enabled this function and so if I was to redo this project I would look to acquire one.
* No backwards compatibility - No access to a barcode scanner means that the device shall only work on a camera enabled device. Research showed however, that most colleagues would rather use their own device, so this is acceptable.
* No waste support - Without access to a printer, new barcodes for the wasted products cannot be scanned. This should not impact the validity of the product stock numbers however, as these can be manually updated by staff when products break.
* No internet connectivity - Without a full understanding of the retrofit library, I was unable to come up with a solution to the problem that I faced.
* No zones addition - They can be updated on the database manually which will lead to better product management and decision making in terms of storage but can be long.
* No custom filters or sorting - I am very sad that I could not make this portion of the project work but this would be easy to add if given more time.

Limitation Avoidance:

The first 3 limitations could be avoided by having access to these peripherals (scanner and printer) during the development process. If these devices were available, I could have used them in testing in order to accommodate their unique inputs and outputs, as well as their unique functions. The next limitation was caused by a lack of understanding of the retrofit library, where I was unable to formulate a solution when a problem occurred. This was due to poor analysis of the kotlin code in the analysis section of this document. I would adjust this with further readings and practice using the retrofit library for smaller scale problems before using it in this large scale problem. However, this was not achievable due to time constraints in developing that stage.

The last 2 limitations could have been avoided with better time management, where if I had more time I could have designed and implemented these functions in full. I have however kept the menu system and pinging branch open with the code in such a way that these can still be added in the future.

**Maintenance:**

The largest maintenance that must happen is the manual management of the database. This includes setting the item’s shop floor location, back room location and general information about the items. This must be done “offline” from the problem, however, by accessing the databases directly and modifying them there. In the future, special functions that are only for managers could be created which could modify these aspects from the device and not on the database directly. Further maintenance could update how the user interfaces with these databases, where the codes for validation and modification happen serverside, instead the only code that functions like the barcode scanner would need is to move the system to the item it receives from the database online.

Each menu is set up in such a way that it allows the stakeholders to add unlimited functions to them. If the buttons become too much, they can always change this aspect to a scrollable list instead. By developing the code in this modular nature, the stakeholders can write their own functions without affecting the rest of the code. Using a theme, colour and string resource file also allows the stakeholders to change the labels and colour set up of this program without having to understand how the code works.

The code is fully annotated, with each line (or block of variable creation) having its own separate annotation. This makes it highly readable for future developers to understand what is happening and why.

Future versions of this software could offer IOS support, or fix any of the limitations given above. This could be done by rewriting the code in a language that supports IOS and the fixes for the limitations were given in the limitation reduction portion of this evaluation. They could also use the backstack better, as right now you must move through all of the fragments to go back to the menu.

**Conclusion:**

In conclusion, I am very happy with the application. It had received positive feedback from all the groups of stakeholders and it fulfilled all but one of its overall success criteria. I feel that this allows me to critically say it was a success, hindered only by its inability to connect to an online database but even as a proof of concept it works remarkably well. I am happy with my choice to use kotlin and feel that the experience gained from this app will allow me to create new apps in the future, using the invaluable experience gained from this project.

I am happy to label this coursework a success as a proof of concept but a lack of internet connectivity prevents me from saying that this coursework would be usable in place of the current system, however given access to their dedicated servers I feel that this would be easily rectified and so is fit for purpose in that regard.

**Bibliography:**

support-leagueoflegends.riotgames. 2013. *Smart Ping*. [online] Available at: <https://support-leagueoflegends.riotgames.com/hc/en-us/articles/201752974-Smart-Ping#:~:text=is%20Warded%20Ping.%22-,Smart%20Ping%20Menu,on%20the%20terrain%20or%20minimap.> [Accessed 18 June 2020].

**Link to the App:**

https://drive.google.com/file/d/17\_3plxUJ7WL1\_cZbObH8KibwgKJ9B9R8/view?usp=sharing

**Link to the Post Development Video:**

https://youtu.be/-MrjlBCr-s8

**Link to the Application Project as a Compressed Folder:**

https://drive.google.com/file/d/1zt9eKCI5HHtAmSjt0eHLWzJqZLrhQtYt/view?usp=sharing

**Code:**

**BinListAdapter.kt**

package com.barrow.ssa.data.adapters

import android.annotation.SuppressLint

import android.view.LayoutInflater

import android.view.View

import android.view.ViewGroup

import android.widget.TextView

import androidx.navigation.Navigation

import androidx.recyclerview.widget.RecyclerView

import com.barrow.ssa.R

import com.barrow.ssa.data.globalvariables.GlobalItems.Companion.finalCalculate

import com.barrow.ssa.data.globalvariables.GlobalItems.Companion.*simulatedPicks*

import com.barrow.ssa.data.globalvariables.GlobalVariables.Companion.*binLocations*

import com.barrow.ssa.data.globalvariables.GlobalVariables.Companion.*selectedBin*

class BinViewHolder(itemView : View) : RecyclerView.ViewHolder(itemView){ // This sets up how the item will look in the list

val text: TextView = itemView.findViewById(R.id.*binText*) // This variable connects to the path of the text view of the item

}

class BinListAdapter : RecyclerView.Adapter<BinViewHolder>() { // This is the recycler view that will be used in the bin fragment

private var items = *binLocations* // This variable collects the list of bin locations from the online database and stores them on the device for later use

set(value){ // This is a setter that updates the recycler view when the data changes

**field** = value

notifyDataSetChanged()

}

override fun getItemCount() = items.size // This function returns the size of the list of bins, as the largest number of items it can hold

@SuppressLint("SetTextI18n")

override fun onBindViewHolder(holder: BinViewHolder, position: Int) { // This will run whenever a new item is being made

val item = items[position] // This takes the current item from the temporary list of bins

holder.text.*text* = item.binLoc + " " + item.binTot// This sets the text field to display the bin location and the total number of items in that bin

holder.itemView.setOnClickListener**{** // This will run whenever the item it has created is clicked

*selectedBin* = item.binLoc.*toString*() // This will change the global variable of the selected bin to be the bin selected

finalCalculate() // This calculates the items it needs to show the user

Navigation.createNavigateOnClickListener(R.id.*action\_binListFragment\_to\_finalListFragment*).onClick(holder.itemView) // This will navigate the user to the list of final items

**}**

}

override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): BinViewHolder { // This will run when the recycler view is made

val layoutInflater = LayoutInflater.from(parent.*context*) // This will set the correct layout of the screen

val view = layoutInflater.inflate(R.layout.*item\_bin\_list*, parent, false) // This will set the correct layout of the item

return BinViewHolder(view) // This will return what the final view looks like

}

}

**DepartmentAdapter.kt**

package com.barrow.ssa.data.adapters

import android.annotation.SuppressLint

import android.view.LayoutInflater

import android.view.View

import android.view.ViewGroup

import android.widget.TextView

import androidx.navigation.Navigation

import androidx.recyclerview.widget.RecyclerView

import com.barrow.ssa.R

import com.barrow.ssa.data.globalvariables.GlobalItems.Companion.binsCalculate

import com.barrow.ssa.data.globalvariables.GlobalVariables.Companion.*departmentItems*

import com.barrow.ssa.data.globalvariables.GlobalVariables.Companion.*selectedDepartment*

class DepartmentViewHolder(itemView : View) : RecyclerView.ViewHolder(itemView) { // This sets up how the item will look in the list

val text: TextView = itemView.findViewById(R.id.*binText*) // This variable connects to the path of the text view of the item

}

class DepartmentAdapter : RecyclerView.Adapter<DepartmentViewHolder>() { // This is the recycler view that will be used in the department fragment

private var items = *departmentItems* // This variable collects the list of departments from the online database and stores them on the device for later use

set(value){ // This is a setter that updates the recycler view when the data changes

**field** = value

notifyDataSetChanged()

}

override fun getItemCount(): Int {

return items.size // This function returns the size of the list of departments, as the largest number of items it can hold

}

@SuppressLint("SetTextI18n")

override fun onBindViewHolder(holder: DepartmentViewHolder, position: Int) { // This will run whenever a new item is being made

val item = items[position] // This takes the current item from the temporary list of bins

// Here a req would be made to the department database to match the number with the name however we can work it out in code as well to save on network space

val departmentTitle : String // This sets up an empty string variable for the department to be given a value

departmentTitle = when(item.numDep){ // This will change the value of the department title based on what department number it is

1 -> "Spirits"

2 -> "Beers"

3 -> "Wines"

4 -> "Rice and Pasta"

5 -> "Cooking and Condiments"

6 -> "Snacks"

7 -> "Housewares"

8 -> "Soft Drinks"

9 -> "Desserts & Baking"

10 -> "Chips/potatoes and veg"

else -> "Unknown"

}

holder.text.*text* = item.numDep.toString() + " " + departmentTitle + " " + item.numTot.toString() // This sets the text field to display the department number, title and the total number of items

holder.itemView.setOnClickListener **{** // This will run whenever the item it has created is clicked

*selectedDepartment* = item.numDep // This will change the global variable of the selected department to be the department selected

binsCalculate() // This calculates the bins it needs to show the user

Navigation.createNavigateOnClickListener(R.id.*action\_departmentFragment\_to\_binListFragment*).onClick(holder.itemView) // This will navigate the user to the list of bins

**}**

}

override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): DepartmentViewHolder { // This will run when the recycler view is made

val layoutInflater = LayoutInflater.from(parent.*context*) // This will set the correct layout of the screen

val view = layoutInflater.inflate(R.layout.*item\_department*, parent, false) // This will set the correct layout of the item

return DepartmentViewHolder(view) // This will return what the final view looks like

}

}

**FinalListAdapter.kt**

package com.barrow.ssa.data.adapters

import android.annotation.SuppressLint

import android.view.LayoutInflater

import android.view.View

import android.view.ViewGroup

import android.widget.TextView

import androidx.navigation.Navigation

import androidx.recyclerview.widget.RecyclerView

import com.barrow.ssa.R

import com.barrow.ssa.data.globalvariables.GlobalVariables.Companion.*finalItems*

import com.barrow.ssa.data.globalvariables.GlobalVariables.Companion.*upc*

class FinalViewHolder(itemView : View) : RecyclerView.ViewHolder(itemView){ // This sets up how the item will look in the list

val upc: TextView = itemView.findViewById(R.id.*finalUPC*) // This variable connects to the path of the text views of the item

val title: TextView = itemView.findViewById(R.id.*finalTitle*) // This allows the text views to be edited for each item

val colour: TextView = itemView.findViewById(R.id.*finalColour*)

val size: TextView = itemView.findViewById(R.id.*finalSize*)

}

class FinalListAdapter : RecyclerView.Adapter<FinalViewHolder>() { // This is the recycler view that will be used in the final list fragment

private var items = *finalItems* // This variable collects the list of items and stores them as the data for the recycler view

set(value){ // This is a setter that updates the recycler view when the data changes

**field** = value

notifyDataSetChanged()

}

override fun getItemCount() = items.size // This function returns the size of the list of items, as the number of items to be displayed

@SuppressLint("SetTextI18n")

override fun onBindViewHolder(holder: FinalViewHolder, position: Int) { // This will run whenever a new item is being made

val item = items[position] // This takes the current item from the array of items collected for the recycler view

holder.upc.*text* = item.itemUpc.toString() // This sets all of the items variable text fields to the correct data

holder.title.*text* = item.itemTitle

if(item.itemColour == null){ // As the colour can be null, a check must be made so that the text view is not null

holder.colour.*text* = "N/A"

}else{

holder.colour.*text* = item.itemColour

}

holder.size.*text* = item.itemSize

holder.itemView.setOnClickListener **{** // This will run whenever the item it has created is clicked

*upc* = item.itemUpc.toString() // This obtains the upc number of the item clicked and sets it to the global upc number

Navigation.createNavigateOnClickListener(R.id.*action\_finalListFragment\_to\_itemInfoFragment*).onClick(holder.itemView) // This navigates to the item info fragment for that item

**}**

}

override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): FinalViewHolder { // This will run when the recycler view is made

val layoutInflater = LayoutInflater.from(parent.*context*) // This will collect the correct layout of the screen

val view = layoutInflater.inflate(R.layout.*item\_final*, parent, false) // This will set the correct layout of the item

return FinalViewHolder(view) // This will return what the final view looks like

}

}

**PingInboxAdapter.kt**

package com.barrow.ssa.data.adapters

import android.view.LayoutInflater

import android.view.View

import android.view.ViewGroup

import android.widget.ImageView

import android.widget.TextView

import androidx.recyclerview.widget.RecyclerView

import com.barrow.ssa.R

import com.barrow.ssa.ui.menus.communication.PingInboxFragment.Companion.*globalPings*

class ItemViewHolder(itemView : View) : RecyclerView.ViewHolder(itemView){ // This sets up how the item will look in the list

val pingImage: ImageView = itemView.findViewById(R.id.*pingImage*) // This variable connects to the path of the text views of the item

val pingType: TextView = itemView.findViewById(R.id.*pingType*) // This allows the text views to be edited for each item

val pingTime : TextView = itemView.findViewById(R.id.*pingTime*)

val pingSender : TextView = itemView.findViewById(R.id.*senderID*)

val pingZone : TextView = itemView.findViewById(R.id.*pingZone*)

}

class PingInboxAdapter : RecyclerView.Adapter<ItemViewHolder>() { // This is the recycler view that will be used in the bin fragment

private var pings = *globalPings* // This variable collects the list of pings for the user and stores them as the data for the recycler view

set(value){ // This is a setter that updates the recycler view when the data changes

**field** = value

notifyDataSetChanged()

}

override fun getItemCount(): Int { // This function returns the size of the list of pings, as the number of items to be displayed

return pings.size

}

override fun onBindViewHolder(holder: ItemViewHolder, position: Int) { // This will run whenever a new item is being made

val item = pings[position] // This takes the current item from the array of pings collected for the recycler view

holder.pingImage.setImageResource(when(item.pingType){ // This sets the icon of the ping to relate to the ping type of the ping

"Drop and Audit" -> R.drawable.*green\_arrow\_upwards*

"Pick" -> R.drawable.*blue\_arrow\_downwards*

"Rumble" -> R.drawable.*red\_trash\_icon*

"Work Here" -> R.drawable.*purple\_arrow\_right*

"Meet Here" -> R.drawable.*yellow\_arrow\_left*

else -> R.drawable.*logo\_asda* // If the ping type is not recognised it is set to be the generic asda logo

})

var corTime = item.sendTime!! // This sets up the variable to write the time in the correct format

corTime = corTime.*substring*(0,2) + ":" + corTime.*substring*(2,4) + ":" + corTime.*substring*(4,6) // This correctly injects the colons into the time

holder.pingType.*text* = item.pingType // These set the text fields to display the relevant information that each field should show

holder.pingTime.*text* = corTime

holder.pingSender.*text* = item.senderID

holder.pingZone.*text* = item.zoneLocation

}

override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ItemViewHolder { // This will run when the recycler view is made

val layoutInflater = LayoutInflater.from(parent.*context*) // This will set the correct layout of the screen

val view = layoutInflater.inflate(R.layout.*item\_ping*, parent, false) // This will set the correct layout of the item

return ItemViewHolder(view) // This will return what the final view looks like

}

}

**LoginAPI.kt**

package com.barrow.ssa.data.apis

//import com.barrow.ssa.ui.login.LoginDataSource

//import com.google.gson.annotations.SerializedName

//import okhttp3.OkHttpClient

//import okhttp3.logging.HttpLoggingInterceptor

//import retrofit2.Call

//import retrofit2.Callback

//import retrofit2.Response

//import retrofit2.Retrofit

//import retrofit2.converter.gson.GsonConverterFactory

//import retrofit2.http.GET

// //import java.util.concurrent.TimeUnit

//data class LoginAccounts(

// @SerializedName("range")

//var range: String,

//@SerializedName("majorDimension")

//var majorDimension: String,

//@SerializedName("values")

//var values: ArrayList<Users>

//)

data class User( // This is the data class for the user item

var username: String,

var password: String,

var storeArea: String,

var storeNumber: String,

var status: String,

)

//internal object LAPIClient {

// lateinit var retrofit: Retrofit

//val client: Retrofit

// get() {

// val interceptor = HttpLoggingInterceptor()

// interceptor.level = HttpLoggingInterceptor.Level.BODY

//val client = OkHttpClient.Builder()

// .addInterceptor(interceptor)

//.connectTimeout(2, TimeUnit.MINUTES)

//.readTimeout(2, TimeUnit.MINUTES)

//.build()

//retrofit = Retrofit.Builder()

// .baseUrl("https://sheets.googleapis.com/v4/spreadsheets/")

//.addConverterFactory(GsonConverterFactory.create())

//.client(client)

//.build()

//return retrofit

// }

//}

//interface LAPIInterface{

// @GET("1ykgYB8zKRTghY1-5FCaapwYNgwKXP6zBCIFq8uj-myM/values/userDatabase!A2:C?key=AIzaSyDAoCkSdHTYUCKAiz5HVOA47JcYhClxytc")

// fun loginAccounts(): Call<LoginAccounts>

//}

//val apiInterface = LAPIClient.client.create(LAPIInterface::class.java)

//val call = apiInterface.loginAccounts()

//call.enqueue(

//object : Callback<LoginAccounts> {

// override fun onResponse(

// call: retrofit2.Call<LoginAccounts>,

// response: Response<LoginAccounts>

// ) {

// val text = response.body()!!

// for (i in 0 until text.values.size) {

// val temp : Users = text.values[i]

// if (username == temp.username) {

// if (password == temp.password) {

// if (storeNumber == temp.storeNumber) {

// LoginDataSource.fail = 0

// }

// }

// }

// }

// }

// override fun onFailure(call: retrofit2.Call<LoginAccounts>, t: Throwable) {

// }})

**PingInboxAPI.kt**

package com.barrow.ssa.data.apis

data class Ping( // This is the dataclass for the ping set up

val senderID: String?,

val recipientID: String?,

val pingType: String?,

val zoneLocation: String?,

val sendTime: String?

)

**PingSendingAPI.kt**

package com.barrow.ssa.data.apis

data class Recipients ( // This the dataclass for the recipients object

var username: String,

var storeNumber: String,

var status: String

)

**globalPings.kt**

package com.barrow.ssa.data.globalvariables

import com.barrow.ssa.data.apis.Ping

class GlobalPings { // This serves as the local database for the pings and creates an array of the pings that can be added to and removed from

companion object{

private val Ping1 = Ping("A0B012C3","U2K62U5","Drop and Audit","000/000/000","123524")

private val Ping2 = Ping("N7R94B1","A0B012C3","Pick","005/005/005","060745")

private val Ping3 = Ping("N7R94B1","A0B012C3","Rumble","000/000/000","060759")

private val Ping4 = Ping("A0B012C3","B0B02U0","Work Here","100/100/001","091112")

private val Ping5 = Ping("U2K62U5","B0B02U0","Meet Here","003/003/003","120005")

var simulatedPings = *arrayListOf*(Ping1, Ping2, Ping3, Ping4, Ping5)

}

}

**globalVariables.kt**

package com.barrow.ssa.data.globalvariables

class GlobalVariables { // This code stores all of the global variables used throughout the program that are not user specific

companion object{

var selectedDepartment = 0

var selectedBin = ""

var destination = ""

var upc = ""

var departmentItems = ArrayList<Department>()

var binLocations = ArrayList<Bin>()

var binItems = ArrayList<Item>()

var finalItems = ArrayList<Item>()

}

}

**Items.kt**

package com.barrow.ssa.data.globalvariables

import com.barrow.ssa.data.globalvariables.GlobalVariables.Companion.*binItems*

import com.barrow.ssa.data.globalvariables.GlobalVariables.Companion.*binLocations*

import com.barrow.ssa.data.globalvariables.GlobalVariables.Companion.*departmentItems*

import com.barrow.ssa.data.globalvariables.GlobalVariables.Companion.*finalItems*

import com.barrow.ssa.data.globalvariables.GlobalVariables.Companion.*selectedBin*

import com.barrow.ssa.data.globalvariables.GlobalVariables.Companion.*selectedDepartment*

import com.barrow.ssa.data.globalvariables.Zones.Companion.*simulatedZones*

data class Item( // This is the data class for an item object

val itemUpc: Long,

val itemTitle: String,

var itemNumber: Int,

var itemDepartment: Int,

val itemSize: String,

val itemColour: String?,

val itemPrice: Double,

var backRoomNum: Int,

var shopFloorNum: Int,

val caseSize: Int,

var maxShelf: Int,

var shopFloorLoc: String,

var backRoomLoc: String?,

var delDate: String?

)

data class Department( // This is the dataclass for a department object

val numDep: Int,

val numTot: Int

)

data class Bin( // This is the data class for a bin object

val binLoc: String?,

val binTot: Int

)

class GlobalItems {

companion object { // This is the global location of all the items, that are then compiled into an editable array

private val item1 = Item(

400065399048,

"ASDA Pasta",

4664789,

4,

"1kg",

"Fusilli",

0.5,

10,

0,

10,

10,

"004/004/004",

"100/100/003",

null

)

private val item2 = Item(

400501683014,

"ASDA Pasta",

1220061,

4,

"1kg",

"Spaghetti",

0.53,

10,

0,

10,

10,

"004/004/004",

"100/100/001",

"10/08/2021"

)

private val item3 = Item(

400507755531,

"ASDA Pancake",

7878078,

8,

"500g",

null,

1.00,

13,

9,

12,

24,

"003/003/003",

"100/100/002",

"20/04/2021"

)

private val item4 = Item(

400500253096,

"ASDA Micro Rice",

6347392,

7,

"500g",

"Egg Fried",

2.50,

0,

8,

12,

24,

"004/004/004",

null,

null

)

private val item5 = Item(

400500252853,

"Lloyd Sauce",

1234567,

7,

"10g",

"Spag Bol",

0.10,

26,

56,

13,

48,

"004/004/004",

"100/100/003",

null

)

private val item6 = Item(

400092167399,

"Tropical Sun Coco",

7681438,

6,

"8x330ml",

null,

3.79,

1,

5,

1,

6,

"005/005/005",

"100/100/003",

"6/9/2021"

)

var simulatedItems = *arrayListOf*(item1, item2, item3, item4, item5, item6)

var simulatedPicks = ArrayList<Item>() // This is the array that houses all of the picks that would be generated by the server

// As I do not have access to a server, they will be generated on the device

fun pickCalculate(){ // This will calculate the item picks

simulatedPicks = ArrayList() // Firstly, it resets the pick list

for (element in simulatedItems) { // Then it goes through all the items in the store

if (element.backRoomNum >= element.caseSize) { // If there is a case of the item in the backroom

if (element.caseSize <= (element.maxShelf - element.shopFloorNum)) { // And there is space for a case on the shop floor

simulatedPicks.add(element) // It will be added to the list as a pick

}

}

}

}

fun departmentsCalculate(){ // This will calculate the departments that have picks

*departmentItems* = ArrayList() // First, it resets the list of departments

for(i in 1.. 10){ // Then it goes through each department

if (simulatedPicks.*count* **{**e **->** e.itemDepartment == i**}** != 0){ // If there is an item to be picked on the department

*departmentItems*.add(Department(i,simulatedPicks.*count* **{**e **->** e.itemDepartment == i**}** )) // Then it adds that department and how many items there are to the array

}

}

}

fun binsCalculate(){ // This function calculates the bins in the department

*binLocations* = ArrayList() // Firstly it resets the list of locations and items in the department

*binItems* = ArrayList()

for(element in simulatedPicks){ // For each element in the list of picks

if(element.itemDepartment == *selectedDepartment*){ // If the department of the items matches the selected department

*binItems*.add(element) // It is added to the list of items in that department

}

}

for(element in *binItems*){ // For each element in the department

if(*binLocations*.*count* **{**e **->** e.binLoc == element.backRoomLoc**}** == 0){ // If the bin location has an item

*binLocations*.add(Bin(element.backRoomLoc, *binItems*.*count***{**e **->** e.backRoomLoc == element.backRoomLoc**}**)) // It adds the bin and the amount of items in that bin

}

}

}

fun finalCalculate(){ // This calculates the items specific to the bin chosen

*finalItems* = ArrayList() // Firstly, it resets the bins contents

for(element in *binItems*){ // Next it goes through each item in the department

if(element.backRoomLoc == *selectedBin*){ // If that item has the matching bin location

*finalItems*.add(element) // It is added to the final list

}

}

}

}

}

**userInformation.kt**

package com.barrow.ssa.data.globalvariables

import com.barrow.ssa.data.apis.User

class UserInformation { // This class holds the global variables for the information of the user as well as the database of all user details

companion object{

var userName = ""

var storeNumber = ""

var storeArea = ""

private var user1 = User("A0B012C3", "ZusammenZucken1", "Store/Club", "4917", "offline")

private var user2 = User("U2K62U5", "Alex15NotaS1mp", "Homeoffice", "0000", "offline")

private var user3 = User("R9F25R8", "GiveMeFullM4rks!", "DC", "0000", "online")

private var user4 = User("N7R94B1", "WeeWooWeeWoo", "Store/Club Pharmacy and Optical", "4917", "online")

private var user5 = User("B0B02U0", "abc123", "Store/Club", "4917", "offline")

var simulatedUsers : ArrayList<User> = arrayListOf(user1, user2, user3, user4, user5)

}

}

**Zones.kt**

package com.barrow.ssa.data.globalvariables

class Zones { // This class holds the device's database for all of the available zones in the store

companion object {

private const val zone1 = "001/001/001"

private const val zone2 = "002/002/002"

private const val zone3 = "003/003/003"

private const val zone4 = "004/004/004"

private const val zone5 = "005/005/005"

private const val zone6 = "006/006/006"

private const val zone7 = "100/100/001"

private const val zone8 = "100/100/002"

private const val zone9 = "100/100/003"

private const val zone10 = "100/100/004"

private const val zone11 = "100/100/005"

private const val zone12 = "600/600/001"

val simulatedZones = *arrayListOf*(zone1, zone2, zone3, zone4, zone5, zone6, zone7, zone8, zone9, zone10, zone11, zone12)

}

}

**LoggedInUser**

package com.barrow.ssa.data.model

*/\*\**

*\* Data class that captures user information for logged in users retrieved from LoginRepository*

*\*/*

data class LoggedInUser(

val username: String,

val storeArea: String

)

**LoggedInUserView**

package com.barrow.ssa.data.model

*/\*\**

*\* User details post authentication that is exposed to the UI*

*\*/*

data class LoggedInUserView(

val username: String,

val storeNumber: String

)

**LoginFormState**

package com.barrow.ssa.data.model

*/\*\**

*\* Data validation state of the login form.*

*\*/*

data class LoginFormState(val usernameError: Int? = null,

val passwordError: Int? = null,

val storeNumberError: Int? = null,

val isDataValid: Boolean = false)

**LoginResult**

package com.barrow.ssa.data.model

*/\*\**

*\* Authentication result : success (user details) or error message.*

*\*/*

data class LoginResult(

val success: LoggedInUserView? = null,

val error: Int? = null

)

**LoginViewModelFactory**

package com.barrow.ssa.data.model

import androidx.lifecycle.ViewModel

import androidx.lifecycle.ViewModelProvider

import com.barrow.ssa.ui.login.LoginDataSource

import com.barrow.ssa.ui.login.LoginRepository

import com.barrow.ssa.ui.login.LoginViewModel

*/\*\**

*\* ViewModel provider factory to instantiate LoginViewModel.*

*\* Required given LoginViewModel has a non-empty constructor*

*\*/*

class LoginViewModelFactory : ViewModelProvider.Factory {

@Suppress("UNCHECKED\_CAST")

override fun <T : ViewModel> create(modelClass: Class<T>): T {

if (modelClass.isAssignableFrom(LoginViewModel::class.*java*)) {

return LoginViewModel(

loginRepository = LoginRepository(

dataSource = LoginDataSource()

)

) as T

}

throw IllegalArgumentException("Unknown ViewModel class")

}

}

**Result.kt**

package com.barrow.ssa.data.model

*/\*\**

*\* A generic class that holds a value with its loading status.*

*\** ***@param*** *<T>*

*\*/*

sealed class Result<out T : Any> {

data class Success<out T : Any>(val data: T) : Result<T>()

data class Error(val exception: Exception) : Result<Nothing>()

override fun toString(): String {

return when (this) {

is Success<\*> -> "Success[data=$data]"

is Error -> "Error[exception=$exception]"

}

}

}

class LogInException(message:String): Exception(message)

**BarcodeScanner**

package com.barrow.ssa.ui.barcode

import android.content.Intent

import android.os.Bundle

import android.view.LayoutInflater

import android.view.View

import android.view.ViewGroup

import android.widget.Button

import android.widget.EditText

import android.widget.Toast

import androidx.fragment.app.Fragment

import androidx.lifecycle.ViewModelProvider

import androidx.navigation.fragment.NavHostFragment

import com.barrow.ssa.R

import com.barrow.ssa.data.globalvariables.GlobalItems.Companion.binsCalculate

import com.barrow.ssa.data.globalvariables.GlobalItems.Companion.departmentsCalculate

import com.barrow.ssa.data.globalvariables.GlobalItems.Companion.finalCalculate

import com.barrow.ssa.data.globalvariables.GlobalItems.Companion.pickCalculate

import com.barrow.ssa.data.globalvariables.GlobalItems.Companion.*simulatedItems*

import com.barrow.ssa.data.globalvariables.GlobalVariables.Companion.*binItems*

import com.barrow.ssa.data.globalvariables.GlobalVariables.Companion.*departmentItems*

import com.barrow.ssa.data.globalvariables.GlobalVariables.Companion.*destination*

import com.barrow.ssa.data.globalvariables.GlobalVariables.Companion.*finalItems*

import com.barrow.ssa.data.globalvariables.GlobalVariables.Companion.*upc*

import com.google.zxing.integration.android.IntentIntegrator

class BarcodeScanner : Fragment() { // This is the fragment that is run whenever the barcode scanner is accessed

private lateinit var viewModel: BarcodeScannerViewModel // This creates a variable to store the view model for the barcode

override fun onCreateView(

inflater: LayoutInflater,

container: ViewGroup?,

savedInstanceState: Bundle?

): View? { // This is ran when the barcode scanner fragment is created

viewModel =

ViewModelProvider(this).get(BarcodeScannerViewModel::class.*java*) // This saves the correct view model to the variable created earlier

val root = inflater.inflate(

R.layout.*fragment\_barcode\_scanner*,

container,

false

) // This inflates the correct fragment layout for the screen

val scanButton =

root.findViewById<Button>(R.id.*buttonScan*) // These variables find the inputs of the screen and saves their paths to variables to obtain the inputs

val manScanButton = root.findViewById<Button>(R.id.*scanManualButton*)

val manualInput = root.findViewById<EditText>(R.id.*barcodeManualInput*)

scanButton.setOnClickListener **{** // This creates an onclick listener for the scanning button so that it runs the barcode scanner when it is pressed

val prompt =

getString(R.string.*action\_scanning*) // This obtains the string value from the string file for the scanner to use

val intentIntegrator =

IntentIntegrator.forSupportFragment(this) // This sets up the barcode scanner and puts in the correct parameters

intentIntegrator.setBeepEnabled(false) // Not to beep when a scan is made

intentIntegrator.setCameraId(0) // To use the correct camera ID

intentIntegrator.setPrompt(prompt) /// To prompt the user with what they should do

intentIntegrator.setBarcodeImageEnabled(false) // To not return an image of the barcode but the UPC number instead

intentIntegrator.initiateScan() // To start the scan

**}**

manScanButton.setOnClickListener **{** // This sets up the manual scanning button

if (manualInput.*text*.length != 12) { // So long as the UPC number is input incorrectly

Toast.makeText(*context*, "UPCs must be 12 long", Toast.*LENGTH\_LONG*)

.show() // It tells the user and does not waste time or network bandwidth

} else { // Otherwise it will run the depart function to leave the barcode scanner with the input as the upc number

depart(manualInput.*text*.toString())

}

**}**

return root // This ends the fragment function

}

override fun onActivityResult(

requestCode: Int,

resultCode: Int,

data: Intent?

) { // This will run whenever a result is returned from the barcode scanner

val result = IntentIntegrator.parseActivityResult(

requestCode,

resultCode,

data

) // This will save the result from the barcode scanner to a variable

if (result != null) { // So long as there is a result

if (result.*contents* == null) { // If the result was empty (they cancelled scanning)

Toast.makeText(*context*, "Scan Cancelled", Toast.*LENGTH\_LONG*)

.show() // It will send a message to the user to inform them that they have cancelled the scan

} else { // Otherwise

Toast.makeText(*context*, "Scan results: " + result.*contents*, Toast.*LENGTH\_LONG*)

.show() // It will tell them the scan was successful and output the results of their scan

depart(result.*contents*) // It will run the depart function to leave the barcode scanner and return the upc number it got from the scanner

}

} else {

super.onActivityResult(

requestCode,

resultCode,

data

) // Otherwise, it will use the default response for the barcode scanner

}

}

private fun depart(UPC: String) { //This is the function that will handle the outgoing UPC numbers

if (*destination* == "item info") { // This will run if the upc number is needed for item look up

val tempItems = *simulatedItems* // It collects the array of items from the database

for (element in tempItems) { // For each item in the database

if (UPC.*toLong*() == element.itemUpc) { // It checks if the upc number it holds is equal to that item's upc number

*upc* = UPC // If it is, it rewrites the current global upc number to be that upc number

NavHostFragment.findNavController(this).navigate(R.id.*itemInfoFragment*) // Then it navigates to the item information screen

return // Then end the function

}

}

Toast.makeText(*context*, "Item not found", Toast.*LENGTH\_LONG*).show() // If it cannot be found, then a message is displayed showing this and the user remains on the scanner

}

if (*destination* == "final list") { // This is the function that wil run if the upc number is used for picks

for (element in *finalItems*) { // For each item in the list of items to be picked

if (element.itemUpc.toString() == UPC) { // If the item upc matches with the upc number scanned

for (item in *simulatedItems*) { // For each item in the online database

if (item.itemUpc.toString() == UPC) { // If the upc number matches the element of the list it is currently accessing

item.shopFloorNum =item.shopFloorNum + item.caseSize // It adds a case to the shop floor and removes a case from the backroom

item.backRoomNum = item.backRoomNum - item.caseSize

pickCalculate() // Next it updates the values in the picks, bins, departments and final item lists

departmentsCalculate()

binsCalculate()

finalCalculate()

if (*finalItems*.size == 0) { // Finally it checks to see if there are still any items left in this bin

if (*binItems*.size == 0) { // It then checks to see if the bin list is also now empty

if (*departmentItems*.size == 0) { // It lastly checks to see if there are still departments with picks on them

NavHostFragment.findNavController(this)

.navigate(R.id.*nav\_shopfloor*) // If there is nothing left, they are returned to the main menu

return // Then end the function

} else {

NavHostFragment.findNavController(this)

.navigate(R.id.*departmentFragment*) // Otherwise they are returned to the list of departments

return // Then end the function

}

} else {

NavHostFragment.findNavController(this)

.navigate(R.id.*binListFragment*) //If there are no items left, it returns to the list of bins

return // Then end the function

}

} else {

NavHostFragment.findNavController(this)

.navigate(R.id.*finalListFragment*) // Otherwise, it returns to the list of items

return // Then end the function

}

}

}

}

}

Toast.makeText(*context*, "Item not found", Toast.*LENGTH\_LONG*).show() // Otherwise, if the item cannot be found on the pick list then the correct item is displayed

NavHostFragment.findNavController(this).navigate(R.id.*finalListFragment*) // It is more helpful to return them to the list of picks at this point, so they can check what to scan

}

}

}

**BarcodeScannerViewModel**

package com.barrow.ssa.ui.barcode

import androidx.lifecycle.ViewModel

class BarcodeScannerViewModel : ViewModel() {

}

**LoginActivity**

package com.barrow.ssa.ui.login

import android.content.Intent

import android.os.Bundle

import android.view.View

import android.widget.\*

import androidx.annotation.StringRes

import androidx.appcompat.app.AppCompatActivity

import androidx.lifecycle.Observer

import androidx.lifecycle.ViewModelProvider

import com.barrow.ssa.R

import com.barrow.ssa.data.globalvariables.UserInformation.Companion.*storeArea*

import com.barrow.ssa.data.globalvariables.UserInformation.Companion.*storeNumber*

import com.barrow.ssa.data.globalvariables.UserInformation.Companion.*userName*

import com.barrow.ssa.data.model.LoggedInUserView

import com.barrow.ssa.data.model.LoginViewModelFactory

import com.barrow.ssa.ui.menus.MainMenu

class LoginActivity : AppCompatActivity(){

private lateinit var loginViewModel: LoginViewModel // Sets up the correct view model for the fragment to be used throughout the code

override fun onCreate(savedInstanceState: Bundle?) {

super.onCreate(savedInstanceState)

setContentView(R.layout.*activity\_login*) // Sets up the correct fragment layout for the program to display to the user

val username = findViewById<EditText>(R.id.*username*) // Connects the input fields that the user operates on to the correct variables that contain their log in information

val password = findViewById<EditText>(R.id.*password*)

val spinner = findViewById<Spinner>(R.id.*storeArea*)

val storeNumber = findViewById<EditText>(R.id.*storeNumber*)

val login = findViewById<Button>(R.id.*login*) // Allows the log in button to be watched so that an onClick function can be set up for the user

val storeAreas = *resources*.getStringArray(R.array.*store\_areas*) // Obtains the array of strings that act as options for the user when they need to select the store area

if (spinner != null) { // The spinner will be set up once, so that it is not null, then it will not be operated on again

val adapter = ArrayAdapter(this, // Generates an adapter with the correct layout for the spinner type we need (simple), the correct context (this) and the options that go into the spinner

android.R.layout.*simple\_spinner\_item*, storeAreas)

spinner.*adapter* = adapter //Connects the adapter to the spinner that we have set up, so that the spinner now has the correct type and data

spinner.*onItemSelectedListener* = object : // This will allow us to modify data held when the user selects a new option in the spinner

AdapterView.OnItemSelectedListener {

override fun onItemSelected(parent: AdapterView<\*>?, view: View?, position: Int, id: Long) {

if (parent != null) { // This is a simple check to ensure the user has selected an option and is a fail safe in case of unseen program failure

*storeArea* = parent.getItemAtPosition(position) as String // This will set the current item selected to be the users store area

}

}

override fun onNothingSelected(parent: AdapterView<\*>?) {

} // This will not be needed, the original choice will be kept

}

}

loginViewModel = ViewModelProvider(this, LoginViewModelFactory())

.get(LoginViewModel::class.*java*) // This will set the variable of the view model be the extended view model written for this activity

loginViewModel.loginResult.observe(this@LoginActivity, *Observer* **{** // This uses a function of the specially written viewmodel that allows for different subroutines to be made

// based on if the user succeeds or fails

val loginResult = **it** ?: return@Observer

if (loginResult.error != null) { // This will run if the user fails to log in

loginFailed(loginResult.error) // The subroutine for failed login attempts is run

}

if (loginResult.success != null) { // This will run if the user achieves a log in

loginSuccess(loginResult.success) // The subroutine for successful login attempts is run

}

**}**)

login.setOnClickListener **{** // This will trigger when the user clicks on the login button

loginViewModel.login(username.*text*.toString(), password.*text*.toString(), *storeArea*, storeNumber.*text*.toString()) // Passes through all the current login details to the login function

**}**

}

private fun loginSuccess(model: LoggedInUserView) {

val welcome = getString(R.string.*welcome*) // Obtains the correct string to display to the user

*userName* = model.username // Obtains the username and store number passed to it and sets it to the global variables

*storeNumber* = model.storeNumber

Toast.makeText( // Displays a message to the user welcoming them and displaying the account they have logged in on

*applicationContext*,

"$welcome $*userName*",

Toast.*LENGTH\_LONG*

).show()

val intent = Intent(this@LoginActivity, MainMenu::class.*java*) // Declares a intention to move from it's current activity to the MainMenu activity

startActivity(intent) //Moves to the main menu

}

private fun loginFailed(@StringRes errorString: Int) {

Toast.makeText(*applicationContext*, errorString, Toast.*LENGTH\_SHORT*).show() // Displays the correct error message received from the login function to the user

}

}

**LoginDataSource**

package com.barrow.ssa.ui.login

import com.barrow.ssa.data.globalvariables.UserInformation.Companion.*simulatedUsers*

import com.barrow.ssa.data.globalvariables.UserInformation.Companion.*userName*

import com.barrow.ssa.data.model.LogInException

import com.barrow.ssa.data.model.Result

import com.barrow.ssa.data.model.LoggedInUser

import java.io.IOException

*/\*\**

*\* Class that handles authentication w/ login credentials and retrieves user information.*

*\*/*

class LoginDataSource {

fun login(username: String, password: String, storeArea: String, storeNumber: String): Result<LoggedInUser>{

val response = *simulatedUsers* //Here the API would be called and the system would retrieve the list of users, as an array list of the type User

var fail = 1 // Sets the fail checker, which will only be "turned off" once all criteria have been met

for (element in response) {

if (username == element.username) {

if (password == element.password) {

if(storeArea == element.storeArea){

if (storeNumber == element.storeNumber) {

fail = 0 // The user has passed all the input checks so the check is turned off

element.status = "online" // The user will now be logged in, so while the user is in use, their status is set to online

*simulatedUsers* = response // The database is updated to reflect these changes

}

}

}

}

}

val finalUser = LoggedInUser(username, storeNumber) // Sets the value of the final user, to be used in code in the main activity code

return when (fail){

1 -> Result.Error(

IOException(

"Error logging in",

LogInException("Log in error")

)

)

else -> Result.Success(finalUser)

}

}

}

**LoginRepository**

package com.barrow.ssa.ui.login

import com.barrow.ssa.data.model.Result

import com.barrow.ssa.data.model.LoggedInUser

*/\*\**

*\* Class that requests authentication and user information from the remote data source and*

*\* maintains an in-memory cache of login status and user credentials information.*

*\*/*

class LoginRepository(val dataSource: LoginDataSource) {

var user: LoggedInUser? = null

private set // This is a value that stores the logged in user with it's own private setter so it is not accidentally changed

init {

user = null // When it is first created, it is set to be empty

}

fun login(username: String, password: String, storeArea: String, storeNumber: String): Result<LoggedInUser> {

val result = dataSource.login(username, password, storeArea, storeNumber) // It takes the log in result

if (result is Result.Success) { // If the result was successful

setLoggedInUser(result.data) // It sets the logged in user to the user details passed

}

return result

}

private fun setLoggedInUser(loggedInUser: LoggedInUser) { // This is the function called in the code above

this.user = loggedInUser // That acts as a private setter for the loggedInUser value

}

}

**LoginViewModel**

package com.barrow.ssa.ui.login

import androidx.lifecycle.LiveData

import androidx.lifecycle.MutableLiveData

import androidx.lifecycle.ViewModel

import com.barrow.ssa.data.model.Result

import com.barrow.ssa.R

import com.barrow.ssa.data.model.LoggedInUserView

import com.barrow.ssa.data.model.LoginResult

class LoginViewModel(private val loginRepository: LoginRepository) : ViewModel() { // This is the view model used in the log in fragment

private val \_loginResult = MutableLiveData<LoginResult>() // This is a private value that tracks the result of the log in attempt

val loginResult: LiveData<LoginResult> = \_loginResult // This will set the value of the tracker to the result

fun login(username: String, password: String, storeArea: String, storeNumber: String) { // This is called when the user tries to log in

val result = loginRepository.login(username, password, storeArea, storeNumber) // It obtains the result

if (result is Result.Success) { // If the result was successful

\_loginResult.*value* = LoginResult(success = LoggedInUserView(username = result.data.username, storeNumber = storeNumber)) // It carries out the successful log in result function

} else { // Otherwise

\_loginResult.*value* = LoginResult(error = R.string.*login\_failed*) // It carries out the unsuccessful log in attempt function

}

}

}

**BinListFragment**

package com.barrow.ssa.ui.menus.backroom.picking

import androidx.lifecycle.ViewModelProvider

import android.os.Bundle

import androidx.fragment.app.Fragment

import android.view.LayoutInflater

import android.view.View

import android.view.ViewGroup

import androidx.recyclerview.widget.RecyclerView

import com.barrow.ssa.R

import com.barrow.ssa.data.adapters.BinListAdapter

class BinListFragment : Fragment() { // This is the bin list fragment that will run when the user accesses a department

private lateinit var binViewModel: BinListViewModel // This creates a variable to store the path of the correct view model for the list

override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View? { // This is ran when the list is created

binViewModel = ViewModelProvider(this).get(BinListViewModel::class.*java*) // This sets the view model variable to the correct view model for the fragment

val root = inflater.inflate(R.layout.*fragment\_bin\_list*, container, false) // This inflates the correct layout for the list to be shown to the user

val binList = root.findViewById<RecyclerView>(R.id.*binList*) // This will obtain the correct path of the recycler view used to show the list of bins

val adapter = BinListAdapter() // This will obtain the adapter needed for the bin recycler view

binList.*adapter* = adapter // This will attached the adapter obtained to the recycler view

return root // This indicates that the function has ended

}

}

**BinListViewModel**

package com.barrow.ssa.ui.menus.backroom.picking

import androidx.lifecycle.ViewModel

class BinListViewModel : ViewModel() {

}

**DepartmentFragment**

package com.barrow.ssa.ui.menus.backroom.picking

import android.os.Bundle

import android.view.LayoutInflater

import android.view.View

import android.view.ViewGroup

import androidx.fragment.app.Fragment

import androidx.lifecycle.ViewModelProvider

import androidx.recyclerview.widget.RecyclerView

import com.barrow.ssa.R

import com.barrow.ssa.data.adapters.DepartmentAdapter

class DepartmentFragment : Fragment() { // This is the department list fragment that will run when the user accesses the pick list

private lateinit var departmentViewModel: DepartmentViewModel // This creates a variable to store the path of the correct view model for the list

override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View? { // This is ran when the list is created

departmentViewModel = ViewModelProvider(this).get(DepartmentViewModel::class.*java*) // This sets the view model variable to the correct view model for the fragment

val root = inflater.inflate(R.layout.*fragment\_department*, container, false) // This inflates the correct layout for the list to be shown to the user

val departmentList = root.findViewById<RecyclerView>(R.id.*departmentList*) // This will obtain the correct path of the recycler view used to show the list of departments

val adapter = DepartmentAdapter() // This will obtain the adapter needed for the department recycler view

departmentList.*adapter* = adapter // This will attached the adapter obtained to the recycler view

return root // This indicates that the function has ended

}

}

**DepartmentViewModel**

package com.barrow.ssa.ui.menus.backroom.picking

import androidx.lifecycle.ViewModel

class DepartmentViewModel : ViewModel() {

}

**FinalListFragment**

package com.barrow.ssa.ui.menus.backroom.picking

import androidx.lifecycle.ViewModelProvider

import android.os.Bundle

import androidx.fragment.app.Fragment

import android.view.LayoutInflater

import android.view.View

import android.view.ViewGroup

import android.widget.TextView

import androidx.navigation.fragment.NavHostFragment

import androidx.recyclerview.widget.RecyclerView

import com.barrow.ssa.R

import com.barrow.ssa.data.adapters.FinalListAdapter

import com.barrow.ssa.data.globalvariables.GlobalVariables.Companion.*destination*

import com.barrow.ssa.data.globalvariables.GlobalVariables.Companion.*selectedBin*

import com.google.android.material.floatingactionbutton.FloatingActionButton

class FinalListFragment : Fragment() { // This is the item list fragment that will run when the user accesses a bin

private lateinit var finalViewModel: FinalListViewModel // This creates a variable to store the path of the correct view model for the list

override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View? { // This is ran when the list is created

finalViewModel = ViewModelProvider(this).get(FinalListViewModel::class.*java*) // This sets the view model variable to the correct view model for the fragment

val root = inflater.inflate(R.layout.*fragment\_final\_list*, container, false) // This inflates the correct layout for the list to be shown to the user

val finalList = root.findViewById<RecyclerView>(R.id.*finalList*) // This will obtain the correct path of the recycler view used to show the list of pings

val finalBinLoc = root.findViewById<TextView>(R.id.*finalBinLoc*) // This will obtain the path of the text view at the top of the screen that shows the bin the user is currently on

val newPickButton = root.findViewById<FloatingActionButton>(R.id.*newPickButton*) // This will obtain the path of the floating action button that allows the user to scan an item

val adapter = FinalListAdapter() // This will obtain the adapter needed for the item recycler view

finalList.*adapter* = adapter // This will attached the adapter obtained to the recycler view

finalBinLoc.*text* = *selectedBin* // This will set the reminder to be the current bin

newPickButton.setOnClickListener **{** // This will set up an on click listener for the floating action button

*destination* = "final list" // This sets up the correct end location for the barcode scanner to take the upc number

NavHostFragment.findNavController(this).navigate(R.id.*barcodeScanner*) // This will navigate to the barcode scanner

**}**

return root // This indicates that the function has ended

}

}

**FinalListViewModel**

package com.barrow.ssa.ui.menus.backroom.picking

import androidx.lifecycle.ViewModel

class FinalListViewModel : ViewModel() {

}

**BackroomFragment**

package com.barrow.ssa.ui.menus.backroom

import android.os.Bundle

import android.view.LayoutInflater

import android.view.View

import android.view.ViewGroup

import android.widget.Button

import androidx.fragment.app.Fragment

import androidx.lifecycle.ViewModelProvider

import androidx.navigation.Navigation

import com.barrow.ssa.R

import com.barrow.ssa.data.globalvariables.GlobalItems

import com.barrow.ssa.data.globalvariables.GlobalItems.Companion.*simulatedPicks*

class BackroomFragment : Fragment() { // This is the backroom menu fragment that will run when the user accesses this menu

private lateinit var backroomViewModel: BackroomViewModel // This creates a variable to store the path of the correct view model for the menu

override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View? { // This is ran when the menu is created

backroomViewModel = ViewModelProvider(this).get(BackroomViewModel::class.*java*) // This sets the view model variable to the correct view model for the fragment

val root = inflater.inflate(R.layout.*fragment\_backroom*, container, false) // This inflates the correct layout for the menu to be shown to the user

val pickButton: Button = root.findViewById(R.id.*button\_picking*) // This obtains the path for the button that needs to go to the department list

GlobalItems.pickCalculate() // These two will begin the initial calculations of all the picks that need to be done, and the first calculations for the departments that must be visited

GlobalItems.departmentsCalculate()

if(*simulatedPicks*.size == 0){ // If there are no picks to be worked

pickButton.*isEnabled* = false // Then the button is unusable as there is nothing to display if run (Although the chances of this happening are 0)

}

pickButton.setOnClickListener( // This sets a simple navigational on click listener, to move to the department list screen when clicked

Navigation.createNavigateOnClickListener(R.id.*departmentFragment*, null)

)

return root // This ends the function

}

}

**BackroomViewModel**

package com.barrow.ssa.ui.menus.backroom

import androidx.lifecycle.ViewModel

class BackroomViewModel : ViewModel() {

}

**CommunicationFragment**

package com.barrow.ssa.ui.menus.communication

import android.os.Bundle

import android.view.LayoutInflater

import android.view.View

import android.view.ViewGroup

import android.widget.Button

import androidx.fragment.app.Fragment

import androidx.lifecycle.ViewModelProvider

import androidx.navigation.Navigation

import com.barrow.ssa.R

import com.barrow.ssa.data.globalvariables.GlobalPings.Companion.*simulatedPings*

import com.barrow.ssa.data.globalvariables.UserInformation.Companion.*userName*

class CommunicationFragment : Fragment() { // This is the communication menu fragment that will run when the user accesses this menu

private lateinit var communicationViewModel: CommunicationViewModel // This creates a variable to store the path of the correct view model for the menu

override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View? { // This is ran when the menu is created

communicationViewModel = ViewModelProvider(this).get(CommunicationViewModel::class.*java*) // This sets the view model variable to the correct view model for the fragment

val root = inflater.inflate(R.layout.*fragment\_communication*, container, false) // This inflates the correct layout for the menu to be shown to the user

val sendingButton: Button = root.findViewById(R.id.*button\_send*) // These variables obtain the path for the buttons that go to the ping sending and receiving screen

val inboxButton: Button = root.findViewById(R.id.*button\_inbox*)

sendingButton.setOnClickListener( // This sets a simple navigational on click listener, to move to the ping sending screen when clicked

Navigation.createNavigateOnClickListener(R.id.*pingSending*, null)

)

// Here the API would retrieve a list of all recipientIDs to check, usually the code below would temporarily store however, we have to take all the recipients manually then store

val tempPings = *simulatedPings* // This retrieves the database of pings and temporarily stores them in a variable

val tempRecipients = ArrayList<String>() // This creates a fresh, blank array of recipients

for(element in tempPings){ // And if any element in the ping database

element.recipientID?.*let* **{** tempRecipients.add(**it**) **}** // Has a username, then that username is added to the ping array

}

inboxButton.*isEnabled* = tempRecipients.contains(*userName*) // If the array has a matching username to the user, the button is enabled

// Although this is a roundabout way of checking, it can be used for more complex checks in the future or for managers

// to check their employees

inboxButton.setOnClickListener ( // This sets a simple navigational on click listener, to move to the ping recieving screen when clicked

Navigation.createNavigateOnClickListener(R.id.*pingInbox*, null)

)

return root // This ends the function

}

}

**CommunicationViewModel**

package com.barrow.ssa.ui.menus.communication

import androidx.lifecycle.ViewModel

class CommunicationViewModel : ViewModel() {

}

**PingInboxFragment**

package com.barrow.ssa.ui.menus.communication

import android.os.Bundle

import android.view.LayoutInflater

import android.view.View

import android.view.ViewGroup

import android.widget.Button

import androidx.fragment.app.Fragment

import androidx.navigation.Navigation

import androidx.recyclerview.widget.RecyclerView

import com.barrow.ssa.R

import com.barrow.ssa.data.adapters.PingInboxAdapter

import com.barrow.ssa.data.apis.Ping

import com.barrow.ssa.data.globalvariables.GlobalPings.Companion.*simulatedPings*

import com.barrow.ssa.data.globalvariables.UserInformation.Companion.*userName*

class PingInboxFragment : Fragment(){ // This is the ping inbox fragment that will be ran when the user accesses this screen

companion object{ // This is a blank global list that can be used to temporarily store the ping list for the user

var globalPings = *mutableListOf*<Ping>()

}

override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View? { // This is ran on creation of the fragment

val root = inflater.inflate(R.layout.*fragment\_ping\_inbox*, container, false) // This will take the correct layout and inflate it

val pingList = root.findViewById<RecyclerView>(R.id.*pingList*) // This will obtain the correct path of the recycler view used to show the list of pings

val replyButton = root.findViewById<Button>(R.id.*replyButton*) // These variables will obtain the paths of the buttons used to reply and clear

val clearButton = root.findViewById<Button>(R.id.*clearButton*)

val tempPings = *simulatedPings* // Here the code would retrieve all the pings from the database and temporarily store them

// Here the global ping variable is reset, so that it can be updated to the most recent data on the database and so is kept up to date

for(element in tempPings){ // For each ping on the database

if(element.recipientID == *userName*) // If that ping is a ping for the user of the device

globalPings.add(element) // The ping is added to the list of pings for that user

}

globalPings.*sortByDescending***{it**.sendTime**}** // The pings are then sorted so that the most recent ping is first

var adapter = PingInboxAdapter() // An adapter is set up for the ping inbox fragment, calling the correct adapter function

pingList.*adapter* = adapter // This adapter is then attached to the recycler view that is needed for the pings

replyButton.setOnClickListener( // An on click listener is then set up for the reply button

Navigation.createNavigateOnClickListener(R.id.*pingSending*, null) // The user is then navigated to the ping sending fragment when the button is pressed

)

clearButton.setOnClickListener **{** // An on click listener is set up for the button to clear the pings

clearPings()

adapter = PingInboxAdapter() // The adapter to display the pings is then refreshed to reflect these changes

pingList.*adapter* = adapter // And the new adapter is attached to the recycler view

**}**

return root // This indicates that the function has ended

}

private fun clearPings(){

globalPings.clear() // This clears both the pings kept on the system

for(element in *simulatedPings*){ // For each ping in the database

if(element.recipientID == *userName*) { // If that ping was for this user

*simulatedPings*.remove(element) // Remove that ping

if (*simulatedPings*.*count* **{** e **->** e.recipientID == *userName***}** != 0){ // If there are still pings in the list for this user

clearPings() // Do it again

}

return // All the pings have been cleared so you can return to the main function

}

}

return // If there are no pings for the user, clear nothing and return

}

}

**PingSendingFragment**

package com.barrow.ssa.ui.menus.communication

import android.annotation.SuppressLint

import android.os.Bundle

import android.view.LayoutInflater

import android.view.View

import android.view.ViewGroup

import android.widget.\*

import androidx.fragment.app.Fragment

import androidx.lifecycle.ViewModelProvider

import androidx.navigation.fragment.NavHostFragment

import com.barrow.ssa.R

import com.barrow.ssa.data.apis.Ping

import com.barrow.ssa.data.globalvariables.GlobalPings.Companion.*simulatedPings*

import com.barrow.ssa.data.globalvariables.UserInformation.Companion.*simulatedUsers*

import com.barrow.ssa.data.globalvariables.UserInformation.Companion.*storeNumber*

import com.barrow.ssa.data.globalvariables.UserInformation.Companion.*userName*

import com.barrow.ssa.data.globalvariables.Zones.Companion.*simulatedZones*

import java.text.SimpleDateFormat

import java.util.\*

import kotlin.collections.ArrayList

class PingSendingFragment : Fragment() { // This is the ping sending fragment that will be ran when the user accesses this screen

companion object { // These are blank user variables that can be carried throughout the program for use in all functions

var recipient = ""

var pingType = ""

var location = ""

}

private lateinit var viewModel: PingSendingViewModel // This will create a variable to store the view model in later

@SuppressLint("SimpleDateFormat")

override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View? { // This function will run when the fragment is created

viewModel = ViewModelProvider(this).get(PingSendingViewModel::class.*java*) // This will attached the correct view model to the variable created earlier

val root = inflater.inflate(R.layout.*fragment\_ping\_sending*, container, false) // This will take the correct layout and inflate it

// At this part the program would retrieve the data it needs from the database

val simulatedDatabaseResponse = *simulatedUsers* // This variable acts as a storage of the response from the database

val recipients = ArrayList<String>() // This is a blank variable that will hold all of the usernames that the user can send pings to

for(element in simulatedDatabaseResponse){ // For each user on the database

if(element.status == "online"){ // If they are online

if(element.storeNumber == *storeNumber*){ // And they work in the same store

recipients.add(element.username) // They are added to the list of potential recipients

}

}}

recipients.*sort*() // The list of recipients is then sorted alphabetically

val spinnerUsername = root.findViewById<Spinner>(R.id.*spinnerUsernames*) // The first spinner is set up to handle username selection

if (spinnerUsername != null){ // If there is a spinner

val adapter =

*activity*?.*let* **{** ArrayAdapter(**it**, android.R.layout.*simple\_spinner\_item*, recipients) **}** //The correct spinner layout, context and strings as passed to it

spinnerUsername.*adapter* = adapter // The adapter created is used as the adapter of the spinner

spinnerUsername.*onItemSelectedListener* = object : AdapterView.OnItemSelectedListener { // When the user selects an item

override fun onItemSelected(parent: AdapterView<\*>?, view: View?, position: Int, id: Long) {

recipient = spinnerUsername.*selectedItem* as String // It changes the recipient selected to that user (as a string)

}

override fun onNothingSelected(parent: AdapterView<\*>?) { // This is not needed, the value will not change if they do not select a recipient

}

}

}

val spinnerPingType = root.findViewById<Spinner>(R.id.*spinnerPingTypes*) // This will create the second spinner for ping types

if (spinnerPingType != null){ // If there is a spinner

val adapter = *activity*?.*let* **{**ArrayAdapter (**it**, android.R.layout.*simple\_spinner\_item*, *resources*.getStringArray(R.array.*ping\_types\_array*))**}** //The correct spinner layout, context and strings from the strings file are passed to it

spinnerPingType.*adapter* = adapter // The adapter created is used as the adapter of the spinner

spinnerPingType.*onItemSelectedListener* = object : AdapterView.OnItemSelectedListener { // When the user selects an item

override fun onItemSelected(parent: AdapterView<\*>?, view: View?, position: Int, id: Long) {

pingType = spinnerPingType.*selectedItem* as String // It changes the ping type selected to that user (as a string)

}

override fun onNothingSelected(parent: AdapterView<\*>?) { // This is not needed, the value will not change if they do not select a recipient

}

}

}

val simulatedLocations = *simulatedZones* // Obtain locations from the online database

if("000/000/000" !in simulatedLocations) { // Checks to see if the null location has not been added

simulatedLocations.add("000/000/000") // Add a null location, this can be used to represent a generic meeting place

}

simulatedLocations.*sort*() //Sort the locations so that it is easy to find the location wanted

val spinnerLocation = root.findViewById<Spinner>(R.id.*spinnerLocations*) // This will create the last spinner for the zone

if (spinnerLocation != null){ // If there is a spinner

val adapter = *activity*?.*let***{**ArrayAdapter (**it**, android.R.layout.*simple\_spinner\_item*, simulatedLocations)**}** //The correct spinner layout, context and strings as passed to it

spinnerLocation.*adapter* = adapter // The adapter created is used as the adapter of the spinner

spinnerLocation.*onItemSelectedListener* = object : AdapterView.OnItemSelectedListener{ // When the user selects an item

override fun onItemSelected(parent: AdapterView<\*>?, view: View?, position: Int, id: Long) {

location = spinnerLocation.*selectedItem* as String // It changes the location selected to the location of the ping (as a string)

}

override fun onNothingSelected(parent: AdapterView<\*>?) { // This is not needed, the value will not change if they do not select a recipient

}

}

}

val sendButton = root.findViewById<Button>(R.id.*buttonSend*) // This will find the path of the button used to send the ping

sendButton.setOnClickListener**{** //When the send button is pressed

val sdf = SimpleDateFormat("HHmmss") //It creates the time format that we want for the current time

val currentTime = sdf.format(Date()) // Then it takes the current time in the format specified above

val finalPing = Ping(*userName*,recipient,pingType,location,currentTime) // Lastly it compiles all of the information inputted by the user

*simulatedPings*.add(finalPing) // And adds this ping to the database

// Code that returns the final ping to the server that will then add the messageID and the date and time

// Date and time requires API level 26 to access and the program runs on a minimum of API 22, so will be have to be compiled server side

Toast.makeText(*activity*, "Ping has been sent to: $recipient", Toast.*LENGTH\_LONG*).show() // This will display a message to the user to reinforce that they have sent off the ping and show the user

NavHostFragment.findNavController(this).navigate(R.id.*nav\_communication*) // Then it navigates back to the communication menu

**}**

return root // This indicates that the code of the fragment is done

}

}

**PingSendingViewModel**

package com.barrow.ssa.ui.menus.communication

import androidx.lifecycle.ViewModel

class PingSendingViewModel : ViewModel() {

}

**ItemInfoFragment**

package com.barrow.ssa.ui.menus.shopfloor

import android.annotation.SuppressLint

import android.os.Bundle

import androidx.fragment.app.Fragment

import android.view.LayoutInflater

import android.view.View

import android.view.ViewGroup

import android.widget.Button

import android.widget.TextView

import androidx.navigation.fragment.NavHostFragment

import com.barrow.ssa.R

import com.barrow.ssa.data.globalvariables.GlobalItems.Companion.*simulatedItems*

import com.barrow.ssa.data.globalvariables.GlobalVariables.Companion.*destination*

import com.barrow.ssa.data.globalvariables.GlobalVariables.Companion.*upc*

class ItemInfoFragment : Fragment() { // This is the class containing all of the functions for the item info fragment

@SuppressLint("SetTextI18n") // Stops the IDE from reporting a problem when variables are put directly into text views

override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View? { // This will run when the fragment is created

val root = inflater.inflate(R.layout.*fragment\_item\_info*, container, false) // This will find and inflate the item info layout

val itemUPC = root.findViewById<TextView>(R.id.*upcNumber*) // These variables find the paths for the text views for the attributes of the item, at the top of the screen

val itemTitle = root.findViewById<TextView>(R.id.*itemTitle*)

val itemNum = root.findViewById<TextView>(R.id.*itemNum*)

val itemDepartment = root.findViewById<TextView>(R.id.*itemDepartment*)

val itemSize = root.findViewById<TextView>(R.id.*itemSize*)

val itemColour = root.findViewById<TextView>(R.id.*itemColour*)

val itemPrice = root.findViewById<TextView>(R.id.*itemPrice*)

val itemDelivery = root.findViewById<TextView>(R.id.*itemDelivery*)

val shopfloorNum = root.findViewById<TextView>(R.id.*shopfloorNum*) // These variables find the paths for the storage attributes of the item, at the bottom of the screen

val backroomNum = root.findViewById<TextView>(R.id.*backroomNum*)

val shopfloorLoc = root.findViewById<TextView>(R.id.*shopfloorLocation*)

val backroomLoc = root.findViewById<TextView>(R.id.*backroomLocation*)

val maxShelf = root.findViewById<TextView>(R.id.*maxShelf*)

val caseSize = root.findViewById<TextView>(R.id.*caseSize*)

val addShopButton = root.findViewById<Button>(R.id.*addToShopfloorButton*) // These variables find the paths for all of the input buttons used in the fragment

val addBackButton = root.findViewById<Button>(R.id.*addToBackroomButton*)

val newScanButton = root.findViewById<Button>(R.id.*scanNewItemButton*)

val item = *simulatedItems*.*single* **{ it**.itemUpc.toString() == *upc* **}** // This variable finds the item in the database that corresponds to the item upc acquired by the barcode scanner

fun shopButtonUpdate(){ // These functions are ran whenever the item number distribution changes

addShopButton.*isEnabled* = // These lines will set the attribute "isEnabled" on or off based on the results of the logical problem

item.caseSize <= (item.maxShelf - item.shopFloorNum) && item.backRoomNum >= item.caseSize && item.shopFloorLoc != null

// This will be true so long as 1. There is room for a full case on the shelf, 2. There is at least one case in the backroom and 3. There is a location for them on the shopfloor

return

}

fun backButtonUpdate() {

addBackButton.*isEnabled* =

(item.shopFloorNum >= item.caseSize) && item.backRoomLoc != null

// This will be true so long as there is at least one case of the item on the shopfloor

return

}

itemUPC.*text* = "UPC Number: $*upc*" // These lines of code add the attributes of the items to their respective text views in the layout, converted to strings where appropriate

itemTitle.*text* = "Item: " + item.itemTitle

itemNum.*text* = "Item Number: " + item.itemNumber.toString()

itemDepartment.*text* = "Department: " + item.itemDepartment.toString()

itemSize.*text* = "Size: " + item.itemSize

itemPrice.*text* = "Price: " + item.itemPrice.toString()

itemColour.*text* = "Colour: " + item.itemColour

itemDelivery.*text* = "Delivery: " + item.delDate

shopfloorNum.*text* = item.shopFloorNum.toString()

backroomNum.*text* = item.backRoomNum.toString()

shopfloorLoc.*text* = "Location: " + item.shopFloorLoc

backroomLoc.*text* = "Location: " + item.backRoomLoc

maxShelf.*text* = "Case Size: " + item.maxShelf.toString()

caseSize.*text* = "Max Shelf: " + item.caseSize.toString()

shopButtonUpdate() // These lines of code initially decide which options should be available to the user

backButtonUpdate()

addShopButton.setOnClickListener**{** // This will be ran whenever the user clicks on the button to add a case to the shop floor

item.shopFloorNum = item.shopFloorNum + item.caseSize // These will update the numbers in the backroom and on the shop floor

item.backRoomNum = item.backRoomNum - item.caseSize

shopfloorNum.*text* = item.shopFloorNum.toString() // These will update the values shown and the buttons

backroomNum.*text* = item.backRoomNum.toString()

shopButtonUpdate()

backButtonUpdate()

**}**

addBackButton.setOnClickListener**{** // This is a mirror of the on click listener for the button that adds to the shop floor

item.shopFloorNum = item.shopFloorNum - item.caseSize

item.backRoomNum = item.backRoomNum + item.caseSize

shopfloorNum.*text* = item.shopFloorNum.toString()

backroomNum.*text* = item.backRoomNum.toString()

shopButtonUpdate()

backButtonUpdate()

**}**

newScanButton.setOnClickListener **{** // This will run whenever the user clicks the button to scan a new item

*destination* = "item info" // This correctly sets the destination of the barcode scanner to scan a new item

NavHostFragment.findNavController(this).navigate(R.id.*barcodeScanner*) // This navigates to the barcode scanning fragment

**}**

return root

}

}

**ShopfloorFragment**

package com.barrow.ssa.ui.menus.shopfloor

import android.os.Bundle

import android.view.LayoutInflater

import android.view.View

import android.view.ViewGroup

import android.widget.Button

import androidx.fragment.app.Fragment

import androidx.lifecycle.ViewModelProvider

import androidx.navigation.fragment.NavHostFragment

import com.barrow.ssa.R

import com.barrow.ssa.data.globalvariables.GlobalVariables.Companion.*destination*

class ShopfloorFragment : Fragment() { // This is the shopfloor menu fragment that will run when the user accesses this menu

private lateinit var shopfloorViewModel: ShopfloorViewModel // This creates a variable to store the path of the correct view model for the menu

override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View? { // This is ran when the menu is created

shopfloorViewModel = ViewModelProvider(this).get(ShopfloorViewModel::class.*java*) // This sets the view model variable to the correct view model for the fragment

val root = inflater.inflate(R.layout.*fragment\_shopfloor*, container, false) // This inflates the correct layout for the menu to be shown to the user

val barcodeScannerButton = root.findViewById<Button>(R.id.*barcodeScannerButton*) // This obtains the path for the button that needs to go to the barcode scanner

barcodeScannerButton.setOnClickListener **{** // This sets a listener for the barcode button, so that the following code is ran on user input

*destination* = "item info" // This sets the global destination variable to item info, so that the barcode function knows where to pass the results to

NavHostFragment.findNavController(this).navigate(R.id.*barcodeScanner*) // This finds the barcode scanner fragment on the navigation map and navigates to it

**}**

return root // This ends the function

}

}

**ShopfloorViewModel**

package com.barrow.ssa.ui.menus.shopfloor

import androidx.lifecycle.ViewModel

class ShopfloorViewModel : ViewModel() {

}

**MainMenu**

package com.barrow.ssa.ui.menus

import android.annotation.SuppressLint

import android.content.Intent

import android.os.Bundle

import android.view.Menu

import android.widget.ImageView

import android.widget.TextView

import android.widget.Toast

import com.google.android.material.navigation.NavigationView

import androidx.navigation.findNavController

import androidx.navigation.ui.AppBarConfiguration

import androidx.navigation.ui.navigateUp

import androidx.navigation.ui.setupActionBarWithNavController

import androidx.navigation.ui.setupWithNavController

import androidx.drawerlayout.widget.DrawerLayout

import androidx.appcompat.app.AppCompatActivity

import androidx.appcompat.widget.Toolbar

import com.barrow.ssa.R

import com.barrow.ssa.data.globalvariables.UserInformation

import com.barrow.ssa.data.globalvariables.UserInformation.Companion.*simulatedUsers*

import com.barrow.ssa.data.globalvariables.UserInformation.Companion.*storeNumber*

import com.barrow.ssa.data.globalvariables.UserInformation.Companion.*userName*

import com.barrow.ssa.ui.login.LoginActivity

class MainMenu : AppCompatActivity() { // The class that holds the code of the main menu

private lateinit var appBarConfiguration: AppBarConfiguration // This will set up a variable now for use in the program later that allows

// me to join navigational end points to the app bar

override fun onCreate(savedInstanceState: Bundle?) { // This will be run when the activity is created

super.onCreate(savedInstanceState)

setContentView(R.layout.*activity\_main\_menu*) // This will set the correct activity layout to the screen

val toolbar: Toolbar = findViewById(R.id.*toolbar*) // This will assign the path of the toolbar we'll use to a variable

setSupportActionBar(toolbar) // This sets the toolbar found earlier in the code to be the current active toolbar

val drawerLayout: DrawerLayout = findViewById(R.id.*drawer\_layout*) // These variables find the paths of the drawer layout, navigational views and controllers

val navView: NavigationView = findViewById(R.id.*nav\_view*)

val navController = *findNavController*(R.id.*nav\_host\_fragment*)

appBarConfiguration = *AppBarConfiguration*( // This uses the variable instantiated earlier in the code to configure the items in the appbar to end destinations and attaches

// the correct drawer layout

*setOf*(R.id.*nav\_shopfloor*, R.id.*nav\_backroom*, R.id.*nav\_communication*

), drawerLayout

)

*setupActionBarWithNavController*(navController, appBarConfiguration) // These two lines of codes set up the action bar to use the navigational controller with the

// configured endpoints

navView.*setupWithNavController*(navController)

}

@SuppressLint("SetTextI18n") // This is used to stop the IDE from reporting an error in the developed code, where the variable is directly put in "" into the text view

override fun onCreateOptionsMenu(menu: Menu): Boolean { // This function will run on the creation of the action bar

// Inflate the menu; this adds items to the action bar if it is present.

*menuInflater*.inflate(R.menu.*main\_menu*, menu)

val username: TextView = findViewById(R.id.*usernameMenu*) // This variable obtains the path for the text view on the top of the side bar for the username

username.*text* = "Username: $*userName*" // The text view is set to accurately display the users username from the global variable

val storenumber: TextView = findViewById(R.id.*storeNumberMenu*) // This is the same as the previous part of code, except for the store number

storenumber.*text* = "Store Number: $*storeNumber*"

val logoutButton : ImageView = findViewById(R.id.*power\_button*) // This will obtain the path for the image used when logging out

logoutButton.setOnClickListener **{** // This sets up a listener for the image, so that when it is clicked the following code is ran

logout() // This runs the logout function

**}**

return true // This ends the function, as functions must return something

}

override fun onSupportNavigateUp(): Boolean { // This is more code to set up the navigation, giving the correct navigational layout to the controller

val navController = *findNavController*(R.id.*nav\_host\_fragment*)

return navController.*navigateUp*(appBarConfiguration) || super.onSupportNavigateUp()

}

private fun logout() {

val response = *simulatedUsers* //Here the API would be called and the system would retrieve the list of users, as an array list of the type User

for (element in response){ // Runs for the each user in the system

if (*userName* == element.username){ // Checks to see if the user is the current user. I would like a binary search rather than a linear but I cannot

// be sure the list is sorted

element.status = "offline" // The user will now be logged off, so while the user is not in use, their status is set to offline

*simulatedUsers* = response // The database is updated to reflect these changes

}

}

val message = getString(R.string.*logOut*)

Toast.makeText(*applicationContext*, message, Toast.*LENGTH\_LONG*).show()

val intent = Intent(this, LoginActivity::class.*java*) // Declares a intention to move from it's current activity to the Login activity

startActivity(intent) // Starts the intention and moves to the next activity

}

}