GCE Computer Science Component 3

Edward Robert Karol Demkowicz-Duffy

May 9, 2018

Contents

1   Analysis

1.1   Problem Description

My client is a personal friend and the owner of a business that is based around the sale of novelty clocks and coasters manufactured from or based on vinyl records. They buy (often) second hand vinyl records from various sources, cut out the album art in the middle and use that to mount a clock mechanism on, then package it in its original sleeve modified to become a box as a clock to be hung or place on a surface. They also manufacture sets of coasters which are artificially manufactured to appear as vinyl records, sold in sets of matching bands or contemporary albums. Their present sales solution is in three parts:

- Online sales through third party storefronts Etsy and eBay

- Face-to-face sales at the clients stall at the regular Manchester Christmas markets

- Other face-to-face sales at any other events the client or their employees may wish to attend, irregularly

This solution is inconvenient for multiple reasons. The two storefronts provide quite different experiences and tools for people wanting to sell their products on their platforms and thus my client often finds it difficult to deduce trends and critical values like net profits in their overall online sales. Sales at the Christmas markets are difficult to keep track of as the square they are held in get crowded and the client/their employees frequently lose track of sales, which can be found at the end of the day by examining existing records, money gained and remaining stock; but is still an obstacle to clear analysis of the clients sales. Stalls at other events are usually set up at short notice, and similar logistical problems tend to arise as with the Christmas markets, but with less pre planning time and available information they also tend to be more difficult to solve.

The client would like a web app which provides a quality sales experience to customers, and powerful management tools and information for employees. It must have an online storefront, a (modifiable) storage method to track products and all their related information (most importantly stock and price) and sales of said products. Users should have to register to make purchases. Employees should be to view and modify products. There should be easy tools that specialize in managing stock and sales both at the beginning and the end of a day when the client sells their products at an event, be it unexpected or scheduled. The client also specifically requests that employees be able to view statistics and basic analysis of sales displayed in an easy-to-understand way. Employees with special permission should also be able to modify the accounts of both customers and other employees and delete or add products to the programs storage. Furthermore, the website must be secure, with both customers and employees data appropriately protected and must appear professional and easy to comprehend.

1.2 Stakeholders

Stakeholders are identified here as being people who must use, or will be directly affected by, the program. There are three primary stakeholders, who must be born in mind during the creation of the web app:

Customers These are the most important stakeholders as they as the clients source of income. They must have the easiest and most convenient experience possible, to retain their attention on the products the client wishes to sell to them. They will benefit from the project by having a quick and easy way to exchange money for goods they wish to buy. As such, the interface they use should be carefully and logically laid out, with no unneeded features or artefacts on it. Products should be described clearly and concisely so the customers know exactly what they are buying and all other information relevant to it.

Staff These are the people employed by the client who manage their stock and make sales in person. Their side of the web app will be used for swift and easy viewing of stock and statistics they may also need to modify stock data. Therefore it must provide the quickest and most powerful ways to manage current stock and sales at physical stalls, and helpful processed sales data.

Administrators These are senior employees who have all the responsibilities of other employees but have more power over the companys assets. They have the same requirements as regular employees, but also need to be able to view the data of customers and ordinary employees, be able to add whole new products to the set (and delete existing ones), be able to change the passwords of and delete accounts (be they employee or customer) and be able to view technical data such as logs.

1.3 Solving with Computational Methods

The client requires a centralised service for distributing their products en masse to customers who may be international a website or web app is perfect for this purpose as it can be accessed anywhere, can be easily translated, and requires little to no uncommon knowledge to use and access. It can be accessed from many places at once and run in parallel in all these places without compromising at all in any.

The client requires a service that will supply their company with processed data that will help them analyse their sales strategies and improve them. Computers excel at this as it is a repetitive numerical task which can be completed by a processor much faster than a human analyst. In addition to this, computers can scale easily to very large amounts of data, whereas other methods (aforementioned analyst) would likely struggle to cope in comparison.

The client requires a method for organising all their stock in a central location, with the ability for it to be accessed and changed from anywhere at any time. An always-on, internet-connected data source is an exceptional solution for this problem because besides matching all the required criteria since it can be accessed from anywhere by anyone with the appropriate credentials, it can add useful functionality to them by logging all information transfers and enforcing logical rules like relationships between data and correct formatting. The same data source can store information about both the customers and employees who use it for increased availability of data. This ties in with the previous requirement as all the data needing to be processed being stored in one (digital) location is extremely helpful for any computerised process trying to access it, keeping time taken to do that processing short.

Another advantage of the service being hosted online is that issues that may arise with the software are quick and easy to patch out as the patch can be deployed immediately to the only host with little to no downtime, meaning the service remains almost completely uninterrupted.

1.4 Research

1.4.1 The Client's Current Solution

As mentioned in the problem description, the client employs eBay and Etsy to sell their products online; however as I have covered these examples below I will focus on the physical sales element in this section.

Advantages   There are several advantages to this method of selling, most of which are related to the fact that the customer can talk to an employee in person and that they can view the product they wish to buy first hand.

The customer can examine the product on site, and view properties of it that they may not be able to judge through a website like size, weight and "true" colour. This increases the probability that the customer will make a purchase considerably, and so would be desirable to try to compensate for in the we application - the methods this could be accomplished through are essentially to ensure the product is represented on it's "page" as accurately and fully as possible. Dimensions should be included along with multiple images of the product from different angles and in different lighting conditions.

The customer can also talk to the employee for an extended period of time if they wish, which gives the employee the opportunity to convince them to purchase the product and to further enhance their image in the eyes of the customer (i.e. beyond what the customer has viewed of the array of products and the stall itself). Again, it is hard to provide a comparable service on a website to this. One solution would be to include a live text chat interface on the web application. This would require constant employee presence which is not feasible for my client as their company is not large enough or well-known enough to warrant having a team of support staff available. Another method would be to provide a phone number on the website and encourage people to call it if they have enquiries so as to reduce the requirements on the hosting machine and potentially the number of people needed to answer, but this would still suffer from the same problems as the first proposal (albeit less so).

Disadvantages   Employees can only bring and display so much stock to a physical event, which means the choice and availability customers see is severely stunted. A web application would inherently solve this issue because the entire inventory is displayed simultaneously so this does not need to be considered.

Furthermore, the inventory the client takes with them is vulnerable to theft and damage while it is on display at the event. This is another problem that is solved by developing a web application as a replacement because no stock is "exposed" until it has been sold to a customer.

1.4.2 Existing Successful Examples

This type of software is commonly used in ordinary life today. Multiple excellent examples already exist, some of which the client already utilizes. Three prominent extremely successful public examples are listed below:

- Amazon

- eBay

- Etsy

Amazon   This site is an enormous online marketplace which allows any verified entity to buy or sell goods there. It features extensive methods for displaying information about the goods listed on it, including images, hand written descriptions, multiple variants, stock amounts, delivery information (cost, areas available to deliver to) and in depth public user reviews. It also contains algorithms designed to promote other products to the user, displayed in the form of recommendations, usually labelled "people

who bought this bought" or other such categories. I would guess that these are generated from the logged-in user's purchase history and viewing history by categorising all products on the website, then suggesting the most popular products in the categories the user views and purchases from the most. This is an excellent method to retain user attention and encourage users to stay on the site and spend more money. The feature would be excellent to incorporate into my own application, however it could prove algorithmically quite complex to implement.

The site allows users to store payment information and delivery addresses for ease of use in future transactions - another great way to make user's experience much easier and quicker, which will in turn increase the chances of them making more purchases on the site. The site stores customer's order history in depth with information pertaining to when they were ordered, how much was paid, when they were delivered and even who to. In addition, they provide live tracking of orders which are in the process of being delivered.

Amazon also displays advertisements for currently running sales, new products and even new features. This draws the attention of users to popular products which they are more likely to buy. More so, the efficiency of shopping (i.e. the amount of time it takes to locate a specific product) is much improved by a large array of searching, sorting and filtering options when browsing available goods. Users can sort by price, name and search relevance to name but a few, and can search product names across the whole site or within specified categories.

eBay   This is a site similar to Amazon which allows buyers and sellers to make transactions in exchange for goods and services online. In contrast to Amazon, however, eBay facilitates time-limited auctions to be hosted which buyers can place bids on during a set time period, where the product will be sold to the highest bidder who will pay their named price to the seller. The user interface surrounding this feature is designed fantastically to make it as simple and easy as possible for customers to place bids - it shows them the starting price, the time remaining (live) and the current highest bid. It makes placing a bid as easy as entering a value in a text box and pressing a button with extremely low latency, so that competitive bidding can occur fairly towards the end of the time period which I frequently observed to drive the final price much higher within a small time frame. This means sellers can create highly competitive environments for optimum profit, something I thought my client would appreciate greatly, but upon contacting them they informed me they were not interested in auctioning their products. Still, there are definitely notes to be taken here as the design philosophies of the highly intuitive GUI displayed on the auction page can be applied well to the selling pages of my own project.
eBay offers many of the features Amazon does, namely the recommendations of other products, all the delivery related features and the saving of information related to those deliveries (addresses and payment methods).

Etsy   This is another online marketplace with a similar goal to the aforementioned two. It's distinguishing feature is that it allows sellers to design and construct their own personalised pages to display their products and their information, referred to as their "shop". This is a powerful function as it allows sellers to present themselves as they wish to prospective buyers, be it using particular colour palettes or shape designs, this sets the "attitude" of the seller and creates a "mood" relating to them. While this feature on it's own is not especially relevant to my client's needs as they wan to sell only as one entity in their own location, the core concept of presenting an impression to the user is a powerful way to get a consistent "feel" across the site which fits with my client's intended public image.

1.4.3   Conclusion

Key features shared by all the successful examples are:

- Algorithmically-generated recommendations for the customer based on their order and viewing history. While a full implementation of this may prove to be out of the scope of my project, it is

clearly a useful feature to implement that may well increase the chance of purchases being made by users. Alternatives could be to show users "buy it again" or "next in the set" options.

- Convenience-based features. All the websites I examined were obviously designed to make the user experience as slick as possible. This is very helpful for the seller and website because it encourages the user to spend more time browsing and buying due to the whole process being easier. As such, it should be an important success criterion that the user experience is as "smooth" as possible to promote the most time spent on the website.

- Comprehensive searching and filtering tools. These are designed to assist customers finding the product that suits their needs. The products which will be displayed on my project will be more similar than those present on the websites examined, but the idea of searching through them can be carried over still. The web app I will build should offer similar functionality if it wants users to locate products as swiftly as possible.

- Information-laden lists of products. These worked exceptionally well when used in conjunction with the above tools. Each product is represented in short with crucial information clearly available in a small space, which further assists in finding the desired products.

- Plentiful information related to the item the user is viewing. Each website has an individual page for each item the user views containing a plethora of data that they may want to find about the product. All three website showed a "slide show" of photographs or other images of the product they were viewing, and allowed them to zoom in and examine them. Descriptions written by the sellers featured markdown formatting to allow text effects such as bolding, italics, links and underlining which help express information the seller writes more effectively and strikingly. Stock and prices are shown in depth, with exact numbers of stock left and the delivery costs displayed clearly and prominently in the foreground of the page's design. These would all be good ideas to follow from.

- Storage and easy access of the user's data. This includes past payments and addresses so the user doesn't have to go and retrieve their cards when making a purchase, which in turn keeps them browsing longer and improves their experience making transactions. The sites also store all a user's past orders and display them in a concise way upon request, with money spent, which product, the amount of that product and the date the order was placed all very visible. This is a feature which I can and plan to implement in my own website very effectively, as I already planned to store extended data about any orders made within the program.

Appearance appears to play a key role in usability too - all the websites I examined had consistent colour palettes and designs across all pages which gave a distinct impression of professionalism.

1.4.4 Laws

There are several laws pertaining to data hosted online. Since the service I intend to construct will store sensitive personal information, the Data Protection Act 1998 dictates that the service provider is responsible for any damage or theft of that data. My solution must be secure enough that the client can be confident their user's data is safe from people who shouldn't see it as they would be held liable should it be stolen.

1.5 Software and Hardware Requirements

As this software will be interacted with in different ways to how it is run at its actual location, I have included two sets of hardware requirements; one for the end user (be they employee or customer) and one for the server hosting the software:

1.5.1 Client Computers

- The latest version of a modern browser. I recommend Google Chrome, Microsoft Edge, Mozilla Firefox, Opera or Safari.

- A computer which can run that browser. For reference, I have taken the requirements[1] for Mozilla Firefox, which is regarded as an industry standard:

  - Pentium 4 or newer processor that supports SSE2
  - 512MB of RAM / 2GB of RAM for the 64-bit
  - 200MB of hard drive space

- An internet connection with download speed equal to or exceeding 5MB/s (megabits per second)

1.5.2 Host Computer

- At least 5GB of available secondary storage with a high read/write rate

- A modern high-performance server processor with 4 or more cores clocked at more than 3GHz

- At least 16GB of RAM

- An internet connection with download and upload speeds equal to or exceeding 1GB/s (gigabits per second)

---

[1]https://www.mozilla.org/en-US/firefox/58.0.2/system-requirements/

1.6 Success Criteria

1. Write a web application that holds information about orders, customers, products and employees.

2. The web application must show users the appropriate data for their role in using the program; products data to customers, managerial data to employees, all data to administrators.

3. The web application must allow customers to create their own credential combinations which they must use to identify themselves when interacting with the web application.

4. The web application must allow customers to select products they would like to buy and purchase them. Furthermore, it must provide them adequate data to make this decision on.

5. The web application must allow employees to view and edit information stored in the application which they are permitted to. They may view:

   - All data related to all products
   - All data related to placed orders
   - The usernames of all registered customers

   They may edit:

   - The stock value of all products
   - The passwords of registered customers

6. The web application must allow administrators to view and edit information stored in the application which is likely to be necessary. They may view:

   - All data related to all products
   - All data related to placed orders
   - The usernames and personal data of all registered customers
   - The usernames and personal data of all registered employees
   - The background logs kept by the application

   They may add:

   - Employee accounts
   - Products

   They may delete:

   - Customer accounts
   - Employee accounts
   - Products

7. The web application should show employees helpful data on sales, including but not limited to:

   - Sales per week
   - Sales per month
   - Sales per year
   - Sales per customer

   It should also be able to show not just sales volume but total income for all of the above. It should display these data in a graphical visualisation which allows quick and easy interpretation of the data.

8. The web application should provide a facility for helping employees manage sales in person. This facility should be able to track what products the employee has taken with them and "reserve" them from the main inventory. It should be able to record what was sold at the end of the event as specially marked orders, and return all unsold "reserved" items to the main inventory.

9. All the above should be accomplished such that it is hard enough for outside hostile elements to access sensitive data to allow the client to be confident that the service won't breach the Data Protection Act 1998.

2 Design

2.1 Problem Decomposition

This problem must be approached with computation in mind if I am to succeed.

I will begin by abstractifying the components of the problem while sticking to my success criteria to keep my goals in sight. For this I should focus on compartmentalising the key individual challenges of the project, i.e. login system, data storage method e.t.c. As I do this, I will record a plan of the project either on paper or digitally which lays out the requirements of each section and the implementation methods I intend to use. The plan should include a list of pages and background processes that the application needs to run.

Once I firmly believe my plan is suitable to the problem, I will begin to think about the appearance of the web application. I will prototype several sketches for the different pages that I think are appropriate, consult the client with them, use their feedback to improve some of them and show the client again. I can repeat this process until we reach a prototype me and the client are both satisfied with. When I have a final prototype, I will experiment with different core rules for all pages to follow based on that prototype - these is important because it retains a consistent look throughout the application, giving the user a smoother and more comfortable experience when interacting with it.

Having finalised these rules (which will likely be implemented in Cascading Style Sheets files), I shall move onto selecting the devices and methods to employ to develop the application in. Most prominently I need to choose a programming language or framework and a method for mass data storage. After this, I will progress to actually developing the application using the devices I have chosen. I should start with writing the central functions which many other processes or functions rely on first, like the handling of the products and the program's interaction with whatever data source I select, constantly referring to my plan to ensure I am conforming to it as well as adjusting it to compensate for any problems I may have to solve during development. Once these are complete, I can move on to writing the code for each of the pages, beginning with the most vital first (probably the login page).

At this point it would be wise to present the current working build to the client to gain their feedback, and adapt the application to it. This is valuable to do at this point because the project will still be at a stage where I can easily change parts of it without damaging the functionality of any others, something that will not remain true once I have started finalising the interface and re factoring the code.

Finally, I can fully implement the "nitty-gritty" of the user interface, adapting my rules to ensure no functionality on any page is compromised. I will start this with layout and structure, making sure all the elements of the page are located on it so as to follow my rules. The last step is configuring minor properties such as font and colour.

The project will then move onto testing and review, detailed later.

2.2 Problem Structure

Here begins my documentation of the plan I talked about above. I have decided to express the first part of my plan as a set of flowcharts, one for each discrete sub system within the application (please note that where user input is required followed by a decision, I have concatenated the input and decision elements into one decision element for the sake of brevity). I have decided to compartmentalize the problem as follows.

2.2.1 Pages and Areas

User Authentication This is key to ensuring the function of the application remains organised and track able. All major actions will be logged and tagged with the user that made them, which will help with identification of problems both in full operation, where administrators can trace problems to their source and find the perpetrator, and in development, where I can trace more obscure issues and help myself keep track of what I have done in debugging.
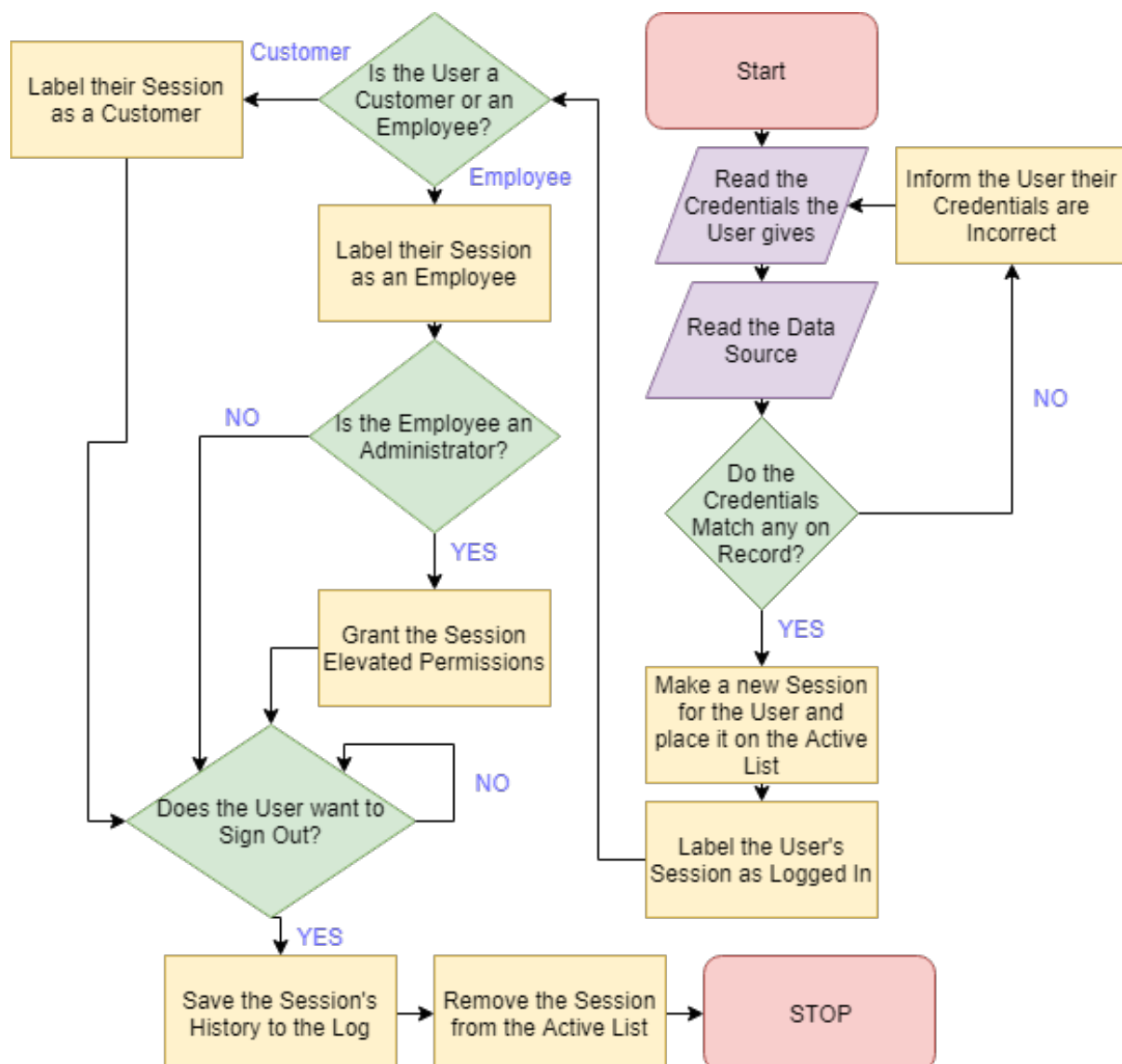
A Session refers to the user's connection to the host computer; it is persistent while the user remains connected to the web application and can have many properties and pieces of data attached to it. The Active List is the conceptual area in memory where the hosting machine will keep track of all the currently active Sessions.

Store Main Area This page/area represents the forefront of the website as it is presented to the customer, and thus must be "slick" and comprehensive. According to my research, I have planned out filtering and sorting features within the page.
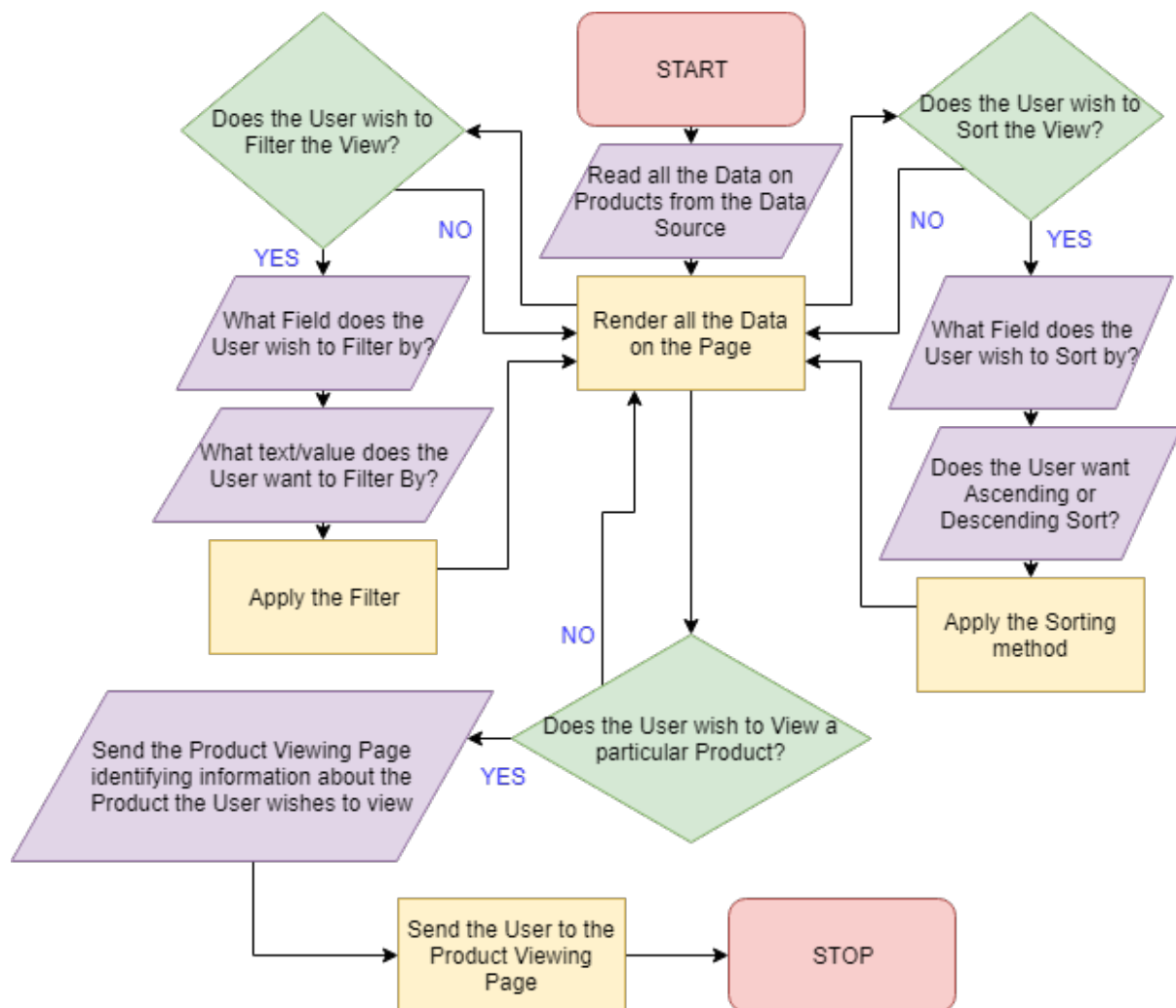
The algorithms used for sorting and filtering will be covered in more details in the algorithms section below.

Product View Page This page will be the "detailed" display of all the information regrading the product. It will show a set of images and a written, plain text description. Customers will be able to add a given amount of a product to their "cart" there. It is *imperative* that the information displayed on this page is plentiful, and very legible.
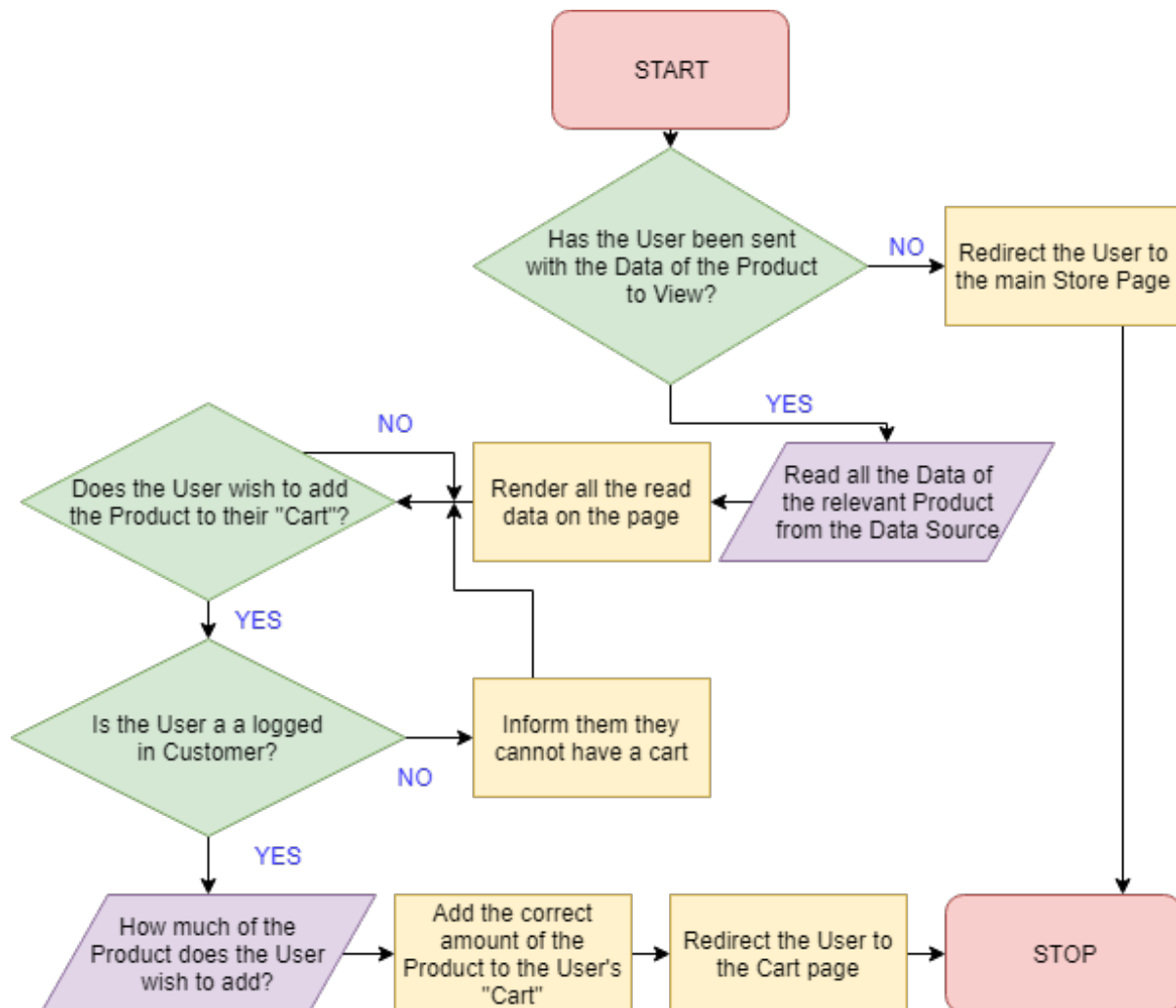
The cart should be a collection of products stored in a group labelled with what customer has them in their cart. It should be independent of the Sessions so that customers can leave the website or sign out without losing the items in their cart. Users who aren't logged in to a customer account may not possess a cart because employees are not authorized to make purchases on the site and people who are not logged in at all may not possess a cart because an account is needed to make a purchase (so purchases are track able).

Cart Page The cart page shows the customer what they currently have in their cart, lets them edit amounts and remove products.

More importantly, this is the page where users will make their actual purchases from. There are several factors which should be considered in regard to this feature:

- Whether there is enough of all products in the inventory to supply the user's purchase

- The user's payment method

- The user's delivery address

After poring over these, I produced the following method.
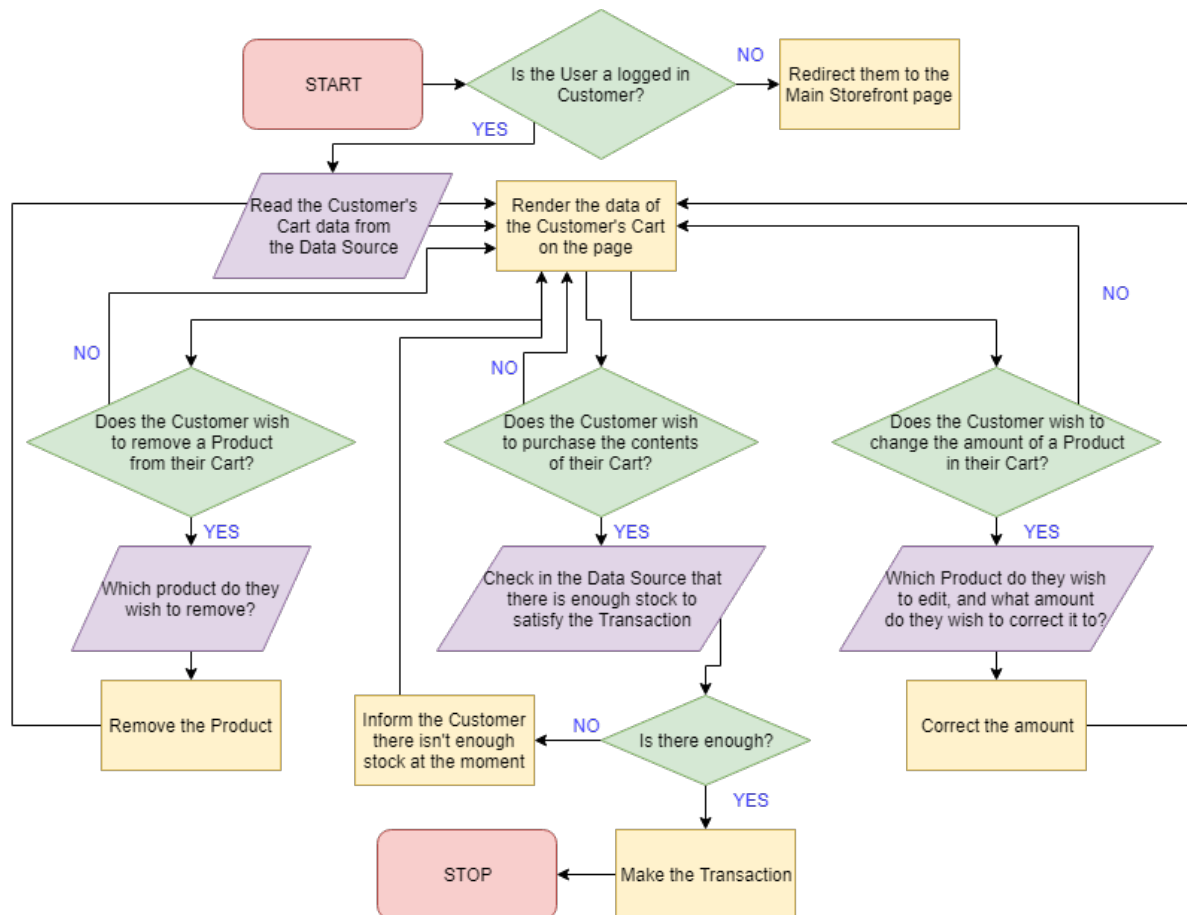
The transaction must be handled individually, as it's own process. While I cannot guess yet, managing monetary transactions may well prove to be out of the scope of this project, but this is yet to be proved.

Physical Market Management Page This page will be where the employee interacts with all features related to managing stock at third party events. Tt's appearance could quickly get confusing, so careful UI design is important.
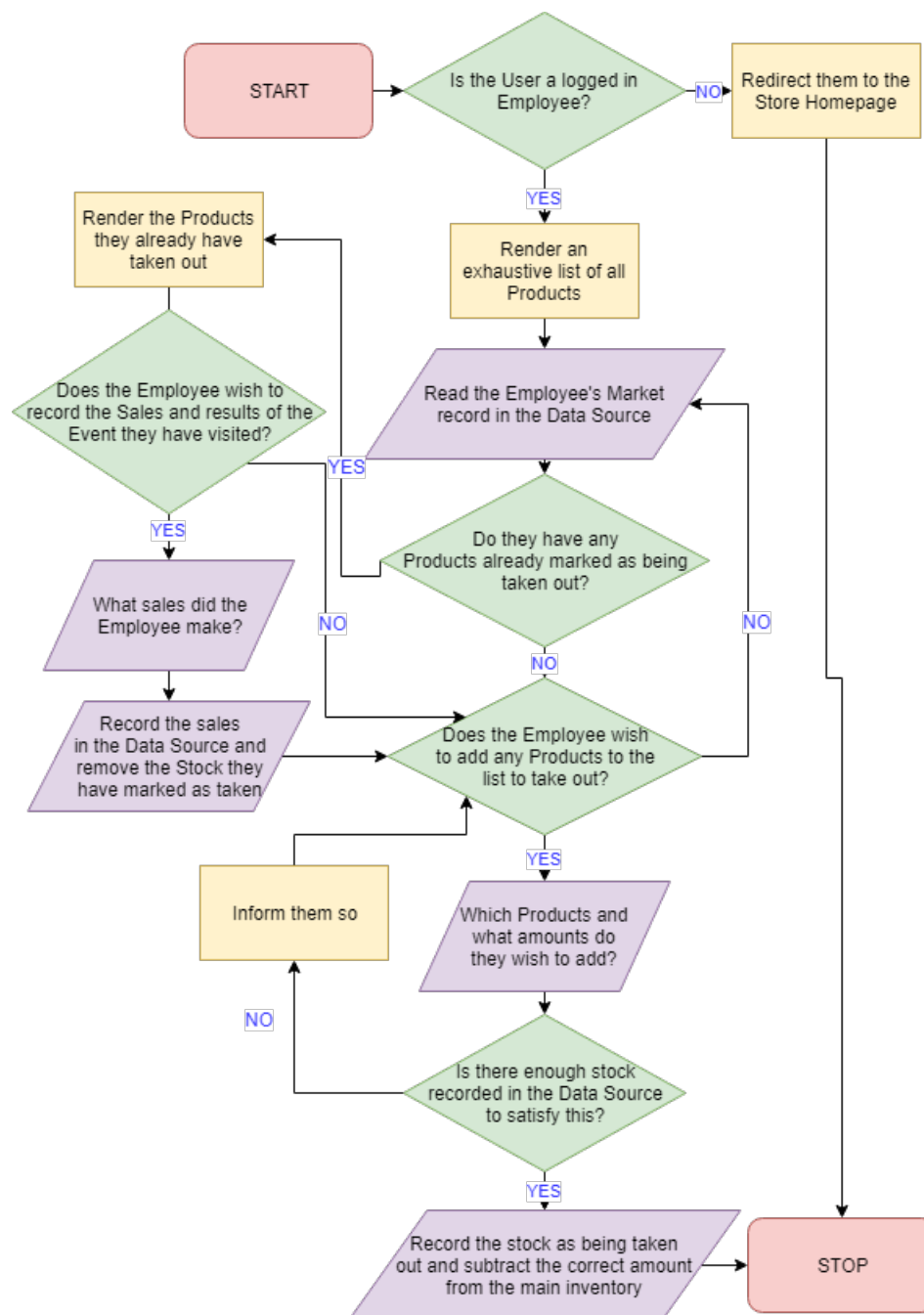
2.3 Algorithms

2.3.1 Quick Sort

One of the complex algorithms I will employ will be a sorting algorithm to sort lists of products to be displayed on the store front in good time. I chose the quick sort algorithm for this purpose because it is highly efficient across most datasets, having $O(log(n))$ average complexity and $O(n^2)$ worst case complexity, and because it offers the opportunity to be solved recursively. My pseudocode for the algorithm is as follows:

```
function quicksort(dataset)
    //Contingencies
    switch dataset.length:
        case 1:
            return dataset
        case 2:
            if dataset[1] < dataset[0] then
                dataset[0], dataset[1] = dataset[1], dataset[0]
            endif
            return dataset
    endswitch

    //Find the pivot
    pivotindex = dataset.length / 2
    pivotindex = rounddown(pivotindex)

    //Define the two lists
    list under
    list over

    //Main code block
    for i = 0 to (dataset.length - 1)
        if i == pivotindex then
            continue
        elseif dataset[i] < dataset[pivotindex] then
            under.add(dataset[i])
        elseif dataset[i] >= dataset[pivotindex] then
            over.add(dataset[i])
        endif
    next i

    //Convert to arrays and recur
    array underArray = quicksort(under.toarray())
    array overArray = quicksort(over.toarray())

    //Return
    return underArray + dataset[pivotindex] + overArray
endfunction
```

If the dataset (array) the function is passed is too short, the main code block won't function correctly. The first block of code labelled "Contingencies" (beginning line 4) is designed to circumvent this problem; if the function is passed a array of length 1, it will just return it immediately or if it is passed an array of length 2 it will check if they need swapping then return them concatenated into a new array. This can also be referred to as the "break case" of the recursive function - it is the place where the function decides whether or not to recur so that the recursion loop isn't infinite.

The method I chose for finding the pivot in my implementation of quick sort was just to find the element which was halfway through the array; there is much debate over what is the best method, but this one seems the simplest to implement and the most useful for the application at hand. I have used

rounddown() as a placeholder for a function which rounds down non-integer numbers into integers. [2]

When the code refers to the "list" type on lines 18 & 19, I have called the type list to placehold for the "List" type in C#, the language I intend to implement this algorithm in. A list is an open-ended array which can be appended to and turned into a "true" array at any time.

Beginning line 23 is the main block of code where the comparisons are made; if the current element is the pivot (checked against the pivot's index as found before) it is skipped, if it is smaller than the pivot it is added to the list of smaller elements and if it is higher or equal then it is added to the list of bigger elements. This is the primary logic of quicksort.

Beginning line 34 is the "recursive case" of the function, i.e. where it recurs (calls itself within itself). The function converts the two lists to arrays and passes them to new instances of itself. This is powerful because any individual function need only focus on one iteration of the whole sorting process, which makes the code easier to read and understand. The algorithm will spawn instances of the function until the break case is tripped, i.e. the array length it is passed is less than 3, when it will just return the array without creating another instance. This break case will "travel up" the "stack" of instances and all will return their section of the dataset, sorted, in turn. When the initial instance is reached by this phenomenon, it will return the full dataset to the script which called it initially.

---

[2]I choose this over rounding up because the arrays are zero based but I expect the lengths to be one based, and this function will compensate for both that and any .5s that may occur in datasets of odd length.

2.4 Key Variables and Structures

Variables are named references to sections of memory. They are the label for a data value - referring to them or using them will return you the value of that section of memory. Due to this functionality, they are essential to developing a complex program like mine, for storing commonly accessed or changed sections of data and are in fact commonplace in all code scripts for practically all purposes.

2.4.1 Global Variables

The only variables which need to carry between pages are those related to the system controlling logged in users because all pages need to be able to "tell" information about the user that is viewing them, and all other variables will not be needed by (all) pages. These variables will be:

- A string holding the user name of the user that is currently logged in.

- A string or boolean representing whether or not the user is an employee or a customer. Either of these types could be used; the string would use more memory than necessary but would make the code more understandable whereas the boolean would do the opposite (less memory and less readability).

- A boolean holding "true" to show is the user is an administrator, or "false" if they are not. Another option to replace both this and the above variable would be something like an Enumerator (a user defined type which can have a set number of enumerable values) which could have the predefined values "Customer", "Employee" and "Administrator" or similar. However, using two booleans or a bool and a string makes boolean logic on each page slightly less complicated and reduces the need for iteration where it isn't necessary.

- Optionally, a bool showing whether there is a user logged in or the current session is a "guest", ie. not authenticated or formally identified. The need for such a variable could be circumvented by simply setting the value of the string holding the user name to be blank or null, but that method could allow people to put exploits in their username string and perform unauthorized and/or unintended acts on the application.

2.4.2 Structures

The data source is the single key structure in this project. It is critical that the data source be able to store all information the website needs, and make it easily accessible and logically organised so that it can be searched, sorted and manipulated. To this end, I have chosen to use a relational data base as my data source. The additional advantage of using this structure is that within a relational database, the database engine can force referential integrity, which reduces the need for consistency checks at the frontend. Within the database I will have six tables, they will be as follows:

Customers I have chosen to store customers and employees in separate tables because the data attached to each type is different from the other. I am going to investigate the viability of different specific implementations of these tables during development because I do not presently know enough about Microsoft Access to predetermine the method I will use to store customers' data. It should, however, contain basic information:

User Name The user name or handle the customer wishes to be identified by on the site. This should be unique.

Real Name The real name of the user in some format, so their online account can be easily linked to their real world identity for purposes such as shipping and customer support.

Password The private string which the customer uses to verify it is really them whenever they try to log in to the application. This information is very sensitive and so I will not store the password verbatim, instead, I will store a hash of the true password (a hash is a value that can be generated

from a piece of data, but is generated so that the original data cannot be directly obtained from the hash). This means that if the data base's security is breached then attackers will only obtain the hashes, which are useless for logging into the website.

Address The address will allow me to implement the feature I described above where the website remembers the customer's delivery infomation for convenience during checkout. It could also be used for user verification for customer support.

Other Personal Info These could be contact information such as phone numbers or email addresses. It is generally helpful to my client's company that this information be attached to their customers, for the aforementioned customer support.

Employees The employees table can be a little more concise since not as much personal information is required; that and the inclusion of the administrator boolean are the only differences between it and the customer table.

User Name The user name or handle the customer wishes to be identified by on the site. This should be unique.

Real Name This is simply for the sake of simplicity and convenience. It could be helpful for identifying employees when a lot are registered.

Password This will be a hash of the employee's actual password. It will be used for checking their credentials when they try to log in.

Administrator This is the boolean variable I described above. It will label whether or not the employee is an administrator.

Products The products table is very important as it is the table most pages related to the web application's primary function will access.

Product Name The name of the product; should be unique.

Type A string showing whether the product is a clock, coaster or other.

Stock An integer storing the amount of the product available in stock (not including the stock which has been taken to a physical market).

Price The amount of money, in GBP, that the product will cost a customer. Stored as a currency format.

Creator The username of the employee who created it. This helps with accountability for errors and keeps the history of the database traceable.

Description A long string with lots of details about the product which will be displayed on the individual product page.

Image Image file(s), one of which will be displayed as the product's thumbnail on the product listing page and all of which will displayed in a "slide show" on the individual product page.

Band A string containing the name of the band the product is linked to. This allows products to be searched or sorted by what band they are based on.

Orders The orders table will record every individual order that has ever been placed on the web application.

ID This will be an automatically generated number used as the primary key for the database. This serves little front-end purpose.

Date Placed Quite simply, a string or date format which will store the date and time that the customer placed the order. This is essential for statistics, order tracking and customer support.

Product Ordered The name of the product which the customer ordered. This will be referentially enforced against the names of products registered in the products table.

Volume Bought An integer storing how many units of the product the customer ordered.

Transaction Value A decimal or price format which stores how much money was requested and received by the application. While this could be calculated on the spot whenever the record was accessed, by just multiplying the volume bought by the price in the relevant record in the products table, this method allows for sales or discounted purchases to be stored accurately and eliminates the need to run queries on two different tables to get one piece of data.

Buyer The username of the customer who placed the order. This should be referntially enforced.

Market Items A method to "reserve" items which are taken to a market by an employee is needed; and I have decided to create a separate, simple, table to store those.

Product Name The name of the product which has been taken out.

Volume The amount of the product which has been taken out.

Employee The username of the employee who is currently responsible for/in possession of these products.

2.4.3 Intended Implementation

Microsoft Access I have chosen the Microsoft Access software to create and manage the database which will support the application. Access is freely accessible to me as a student and provides professional level database software which integrates perfectly with my other choices for implementation. It allows me to create an automatically enforced relational database which can be queried at any time by any MySQL-compatible code.

C# I have chosen the .NET-based language Visual C# to implement the majority of all logic-based algorithms and scripts in this project. My reasoning for this is:

- I am already familiar with C# and it's syntax, types and object-oriented problem solving method. This reduces the possibility of coming across problems I haven't encountered before meaning development wil hopefully be faster. It is still inevitable that I will encounter such obstacles, but C# should throw fewer than other choices.

- The .NET foundation provides the excellent ASP.NET libray which contains a set of elements for .NET languages to be used for web development. ASP.NET comprises a set of JavaScript elements that imitate HTML elements while simultaneously giving them "hooks" (methods and properties) which are accessible by the .NET-based language behind each page.

- ASP also provides the functionality of Master pages, which are a perfect method of implementing the base layouts I describe in a later section. These are pages with set elements on them which all pages inherit from, and add their own content to.

- C# integrates well with the Microsoft Access database engine with the use fothe Microsoft-provided OleDB library, which contains many functions for running powerful SQL queries on MS access databases.

2.5   Usability Features

Since this project is a web application with a public-facing GUI, usability features are a vital consideration at all stages of development. The primary focus of that consideration will be UI layout and design; all pages must only show what they need to, with no excess information, in a clear and concise format.

Base Layout   To this end, I intend to have a "base layout" for all pages. This will be a blank layout with minimal features (only those which all pages will require) and a large amount of blank space for the content of each page to be placed in. There will be two iteration of this "base layout" - one for all employee-only pages, and one for all others. The two will have starkly colour schemes so they are clearly distinguished when viewing them in order to clearly show when an employee is viewing a customer area or an exclusive employee only page.

Nav-Bar   A nav bar is an area at the top of the page containing links for navigating the web application. These links will be relevant on all pages, they will include a link to the home page, login page and product browsing page; or the statistics page and market page on the employee pages. It will also show an element showing the real name of the currently logged in user, with options to sign out and view their profile. If no user is logged in, it will show a login button in place of this. I plan to implement this feature as the central component of both base layouts.

Content   This is a reserved area of the page where the unique content of each page will be shown, according to the page the user is viewing, for example: on the analytics page graphs will be shown whereas on the product browsing page the product list will be shown. It will be the only place each individual page is allowed to place elements and content.

Background   While this may not strictly be a usability "feature", I think there should be an image background at the edges of the page to break up the monotonous colours. This has the dual advantage of both making the website more pleasing to look at and easier to read.

Since this is a web application, the best method to implement these features is, without a doubt, Cascading Style Sheets. CSS is a styling language that dictates the appearance, colour and layout of web pages. I have chosen it because:

- It allows powerful control of many aspects of web pages as mentioned above with only a few lines of code

- It is simple to read, learn and understand because it has little to no syntax variation

- It can set presets for elements of particular user defined classes, allowing it to retain consistency across the site with minimal extra effort

2.5.1 Customer Base Layout
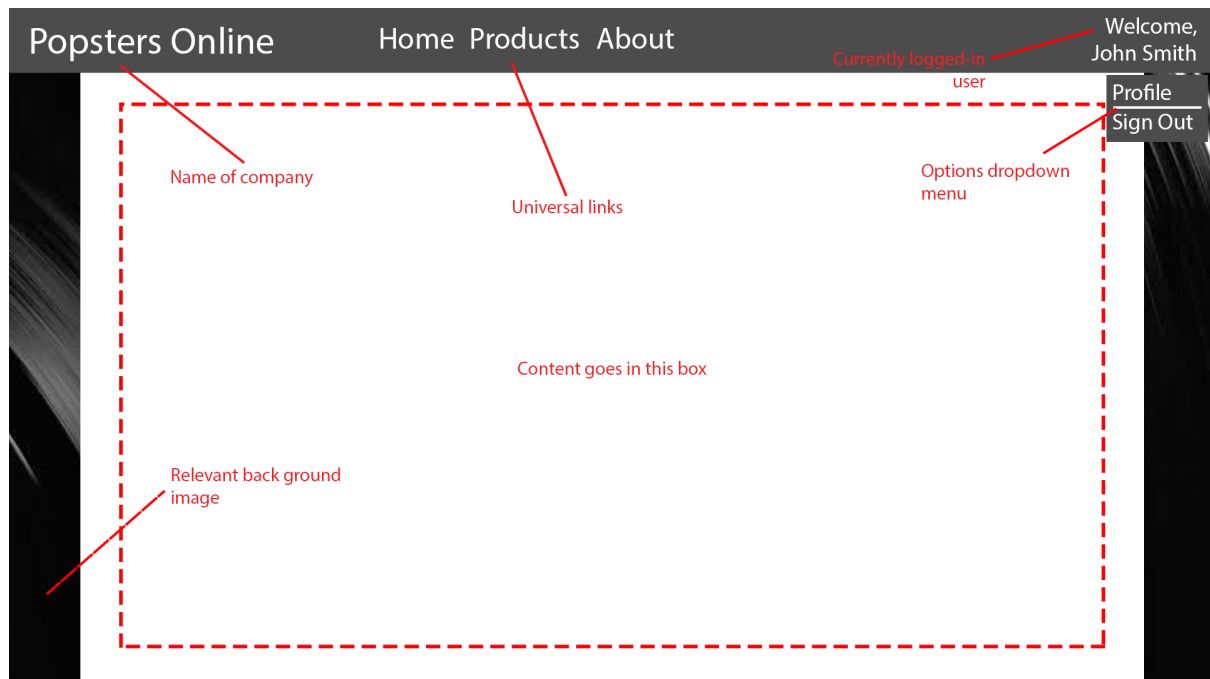
I decided on a muted colour scheme to match the idea of vinyl records for the website. Note that the red text and elements in the image are labelling for the purpose of this document, and will not be present in the finished product. There may also be objects at the bottom of the page containing things like copyright notices, contant information and disclaimers although I am as yet undecided and will consult the client.

2.5.2 Employee Base Layout
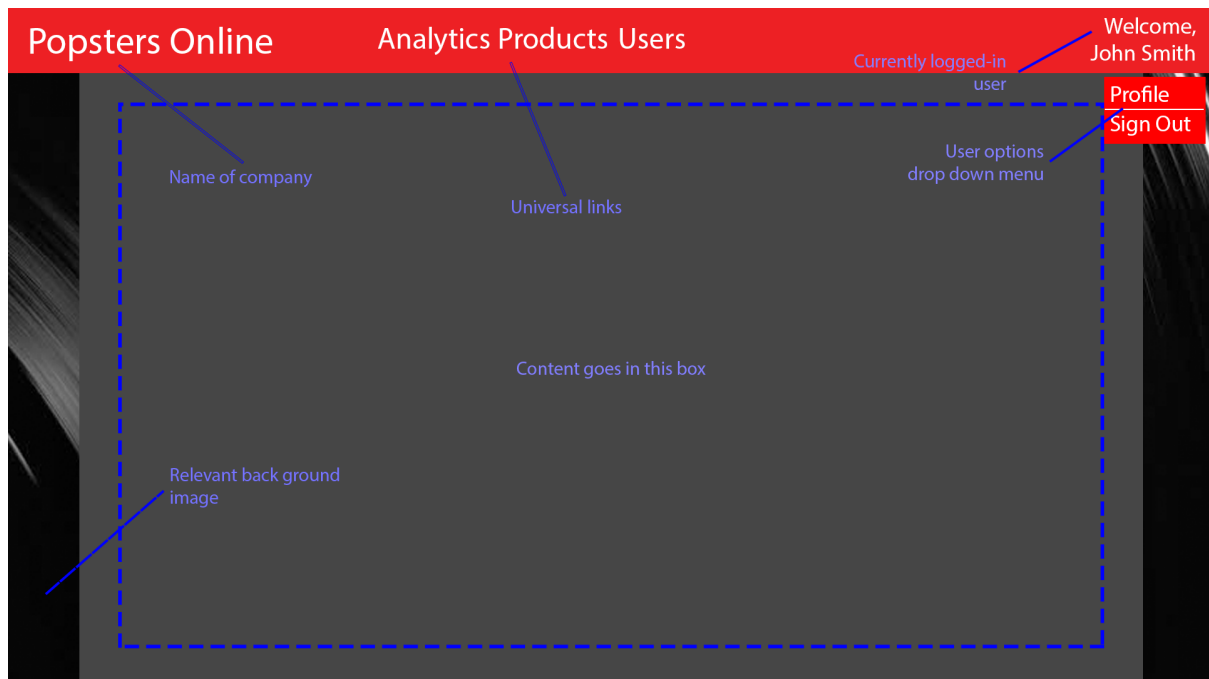
As before, the blue text and boxes are labels for the sake of illustration.

3   Iterative Development

3.1   Data Base

Above, I described the contents of each table in the database. This section is here to provide a visualisation of the relationships between the tables with a screenshot of the Access relationships screen:

3.2 Annotated Code (Initial Solution)
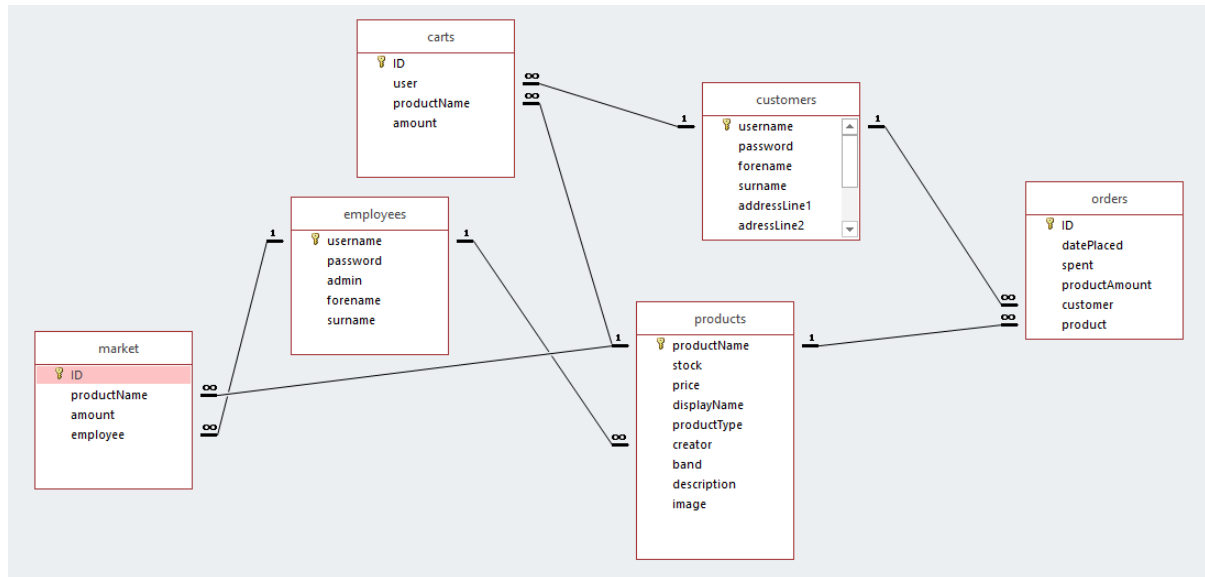
3.2.1 Preface

Code Structure My project is a web application, with the web pages written in ASP.NET and the logic behind them in Visual C#; as such, an optimal structure and organisation of the code is available. Points to remember:

1. All pages are composed of a .aspx file, and a .cs file. The .aspx file stores the markup language style code (ASP) that dictates what elements are permanently present on the page, and all their properties. The .cs file stores the logic attached to those elements and contains all the C# code which can refer to all elements present on the ASP page as if they were ordinary objects; it always contains a predefined method called Page_Load which will be called when the page loads.

2. There is a master file (in my case two) which all pages inherit from. This has common elements like the nav-bar and the content at the bottom of the page. It, just like all other pages, has a .cs file attached containing code which applies elements on that page, which are active on all pages dependant on that master file.

Sessions Within ASP, the Session object refers to a class containing an array of objects which are universal across all pages - a set of global variables, if you will. This is helpful for implementing code which keeps the user authenticated, as described in the design section and is conveniently named the same thing. There are four variables stored in it which I refer to constantly:

isLoggedIn This is a boolean simply representing whether there is currently a user logged in.

currentUser This is a string storing the username of the user currently logged in. It's default value (i.e. when no one is logged in) is "" (nothing).

userType This is another string storing whether the logged in user is an employee, or a customer. It's default value is also "".

userIsAdmin A boolean storing whether the logged in user is an administrator. Default value false.

These are all the classes which contain code needed by many different pages for similar reasons. They are not in the master files because not all pages need them, and those that do will likely not need all of them. Instead, they are grouped into their own .cs file in order to keep them accessible and grouped together without any other logic making them more difficult to read and change.

3.2.2 Logging

The following code controls all the logging of all actions in the software, all log entries are processed through this class. This is not a webpage, this is just a section of shared code.

```
public class customLogging
{
    public static void newSession()
    {
        string result;
        string dividerASCII = "―――――――――――――――――――――――――――――――――――――";
        result = dividerASCII + generateTimestamp() + dividerASCII;
        writeEntry("");
        writeEntry(result);
    }

    public static void newEntry(string entryText)
    {
        string result = generateTimestamp() + " " + entryText;
        writeEntry(result);
    }

    public static void newException(Exception except)
    {
        string result = generateTimestamp() + " An error occurred in " + except.Source +
    " with message " + except.Message;
        writeEntry(result);
    }

    private static void writeEntry(string entryText)
    {
        try
        {
            using (StreamWriter logFile = new StreamWriter(@"\\albert \2011\R04637\
    Computer Science\coursework\mainCoursework\App_Data\log.txt", true))
            {
                logFile.WriteLine(entryText);
            }
        }
        catch (IOException)
        {
            using (StreamWriter logFile = new StreamWriter(@"C:\Users\Edward\Source\
    Repos\coursework\mainCoursework\App_Data\log.txt", true))
            {
                logFile.WriteLine(entryText);
            }
        }
    }

    private static string generateTimestamp()
    {
        string output = "[" + Convert.ToString(DateTime.Now) + "]";
        return output;
    }
}
```

The methods are as follows:

writeEntry   The method that writes the text passed to it to the log file. Here, I have used a try-catch statement to bypass a problem I encountered when working on this project between two workstation in different locations - at school and at home. Since the application does not run out of the directory containing the pages it is showing (the same directory which contains the logfile) because it executes on the pre installed IIS express server, referential file paths serve no purpose. My solution to this was to simply try the file path which would ordinarily contain the logfile on the school hard drive, and switch to that at home if it failed.

The method takes a string containing the text it should enter into the file, opens a StreamWriter class (the system class for writing to binary files) and uses it to send the requested text to the file.

generateTimestamp   This is a string method that returns the current time with square brackets around it, ready to be prefixed onto a log entry.

newSession   A void method which puts a dividing line in the log file indicating that the server has just started, and the history below it until the next divider is a new instance of the server. It also tags it with generateTimestamp's output to mark when the session started. This method is only called by the master page constructor.

newEntry   Another void method which creates a string from the string it is passed and the output of generateTimestamp, and writes it to the log with writeEntry.

newException   Similar to newEntry but it writes an exception's contents to the log. It uses the properties of the exception passed to it "source" and "message" to generate an appropriate entry to the log, timestamp it and write it.

3.2.3 Custom Security

The customSecurity class only covers MD5 hashing and SQL injection checking.

```
public class customSecurity
{
    public const string sanitizeErrorMessage = "All fields must be full. The (, ), +, -,
     = and ' characters are not allowed in ANY fields";

    public static bool sanitizeCheck(string[] input)
    {
        bool isClean = true;
        foreach (string i in input)
        {
            if (i.Contains("(") || i.Contains(")") || i.Contains("'") || i.Contains("=")
     || i.Contains("-") || i.Contains("+"))
            {
                isClean = false;
                break;
            }
            if (i == "")
            {
                isClean = false;
                break;
            }

        }
        return isClean;
    }

    public static string generateMD5(string input)
    {
        var md5 = System.Security.Cryptography.MD5.Create();
        byte[] inputBytes = System.Text.Encoding.ASCII.GetBytes(input);
        byte[] hash = md5.ComputeHash(inputBytes);
        string output = "";
        for (int i = 0; i < hash.Length; i++)
        {
            output = output + hash[i].ToString("x2");
        }
        return output;
    }
}
```

sanitizeErrorMessage   A custom error message stored so that error returns can use it. This exists to retain consistency, i.e. all returns show the same message when SQL - relevant characters are found.

sanitizeCheck   According to my success criteria, the application must adequately protect the users' data. A common method of trying to breach application which use SQL to interact with their databases is called "SQL Injection" - this is when an attacker enters an SQL command into an input they think will be checked against the database. This can be used to issue commands to the database without correct authentication - attackers can log in, drop tables and commit other destructive actions. The most common way to prevent this is referred to as Sanitization, when inputs are checked for characters that could be used in an SQL statement and either remove them or refuse the input if they are found. I have adopted this as no inputs require any of these characters as part of their function; this code takes an array of input box contents and returns true or false depending on if any of those inputs contain SQL sensitive characters (specifically (,),=,-,+). It also serves to make inputs required as it will return false if any of them are blank.

generateMD5   As stated in the key structures section of my design, all passwords stored in the database will be hashed so in the event that they are comnpromised, the password values are of little

use.

The code uses the System.Security.Crypography library to access the algorithms required to generate MD5 hashes from byte data. The function takes a string and creates a new instance of the MD5 class from the cryptography library. It converts the input into an array of bytes using the System.Text.Encoding.ASCII class (which changes ASCII values into byte data, i.e. strings into binary) Another array of bytes is generated using the ComputeHash method of our Cryptography.MD5 instance (called md5). The string output is defined and assigned as "" (this is because the next for loop requires an already existing string to append to). A for loop is run up to the length of the array of bytes (the output data) which appends the string output with each byte using the .ToString method which most objects in C# possess. The "x2" argument specifies that each string output should be capitalised (I use this for consistency).

3.2.4   Data Set

C# and the .NET framework in general provide DataSets for the use of developers to easily interface with a data source. The DataSet automatically generates code using OleDB (the Microsoft database engine) for each query you add to it. I have constructed one central DataSet which holds all my tables and several queries for each. The SQL query statements will be shown and annotated as they are used. The full dataset is shown below: Each table is shown in a box, with the adaptor (the containing object
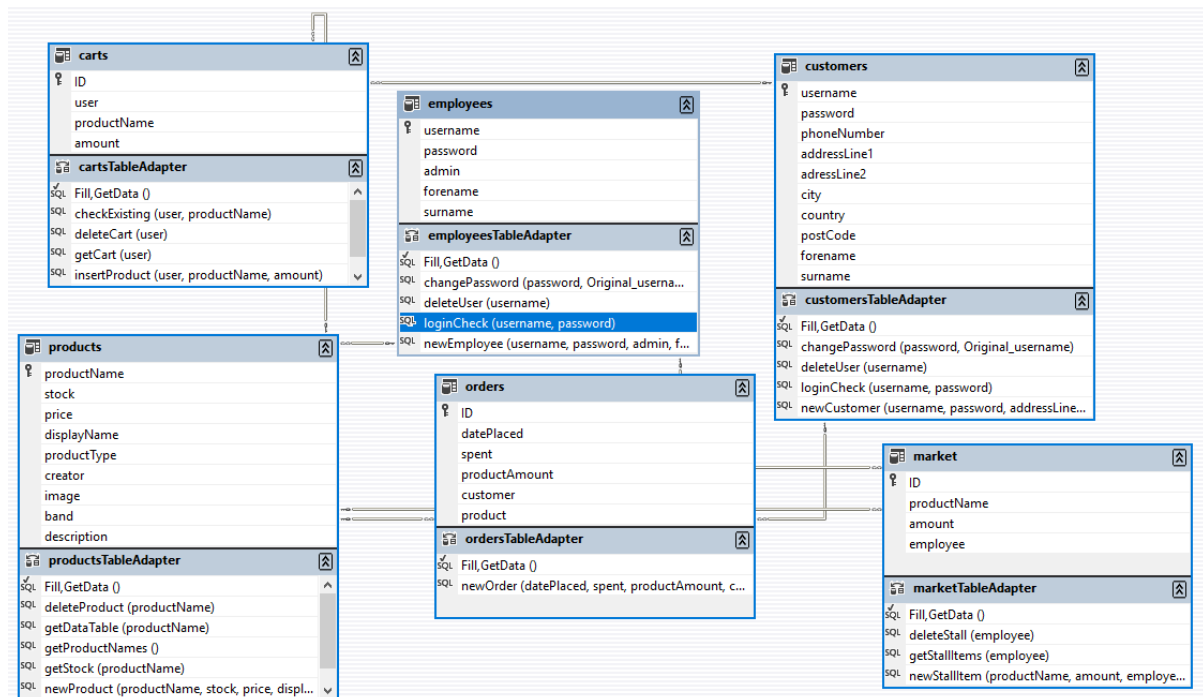
with all the user-defined queries in it) in an adjoining box below that Data relationships are shown with lines connecting the tables, data fields by rows in the table box and queries in the adaptor box.

3.2.5 Customer Master Page

There are two master pages on my project. Each of them contains all the references needed for an ASP page, and some C# code to supplement them. The master customer page contains the navbar, which consists of static links with no code behind them but HTML and a label showing the login status of the current user, a placeholder panel for the content and an area under a horizontal rule which contains a button to sign them out (if they are signed in) and a button to redirect them to the employee side of the site if they are logged in as an employee. The code behind this is:

```
1  protected void Page_Load(object sender, EventArgs e)
2  {
3      if (Convert.ToString(Session["currentUser"]) == "")
4      {
5          usernameLabel.Text = "No user logged in";
6          signOut.Visible = false;
7          userProfileNavbar.Visible = false;
8      }
9      else
10     {
11         usernameLabel.Text = "Welcome,\n" + Session["currentUser"] + "!";
12         signOut.Visible = true;
13     }
14
15     if (Convert.ToString(Session["isLoggedIn"]) == "True")
16     {
17         loginNavbar.Visible = false;
18     }
19
20     if (Convert.ToString(Session["userType"]) != "employee")
21     {
22         employeeRedirect.Visible = false;
23     }
24 }
25
26 protected void signOut_Click(object sender, EventArgs e)
27 {
28     Session["isLoggedIn"] = "False";
29     Session["currentUser"] = "";
30     Session["userIsAdmin"] = "False";
31     Session["userType"] = "";
32     customLogging.newEntry("User logged out");
33     Server.Transfer("~/default.aspx", false);
34 }
35
36 protected void employeeRedirect_Click(object sender, EventArgs e)
37 {
38     Server.Transfer("~/staffOverview.aspx", false);
39 }
40
```

Page_Load Determines whether to display various controls at the bottom of the page. It first checks if a user is logged in by converting the currentUser session to a string and seeing if it is blank. If it is, the label at the footer is set to show "No User Logged In", the sign out button is set to be invisible (and therefore uninteractable; there is no need to hide the employee redirect button because this is invisible by default) and an element on the navbar which would redirect the user to their profile is set to be invisible. If not, the button is confirmed to be visible and the label is set to show the current user's name with a welcome message. If the "isLoggedIn" session is set to true, the login link on the navbar is hidden. If the "userType" session isn't marked as employee, the employee redirect button is set to be invisible.

signOut_Click is the click event for the sign out button. It resets all the user-related sessions to their default state, logs the sign out and redirects the user to the home page.

employeeRedirect_Click just binds the employee redirect button to redirect the user to the employee page.

3.2.6 Employee Master Page

The employee master page contains the navbar, the content placeholder and the footer with the sign out button on it just as the customer master does.

```
1  protected void Page_Load(object sender, EventArgs e)
2  {
3      if (Convert.ToString(Session["userType"]) != "employee")
4      {
5          Server.Transfer("~/default.aspx", true);
6      }
7      if (Convert.ToString(Session["isLoggedIn"]) == "True")
8      {
9          loginNavbar.Visible = false;
10     }
11     usernameLabel.Text = "Welcome\n, employee " + Session["currentUser"] + "!";
12     signOut.Visible = true;
13 }
14
15 protected void signOut_Click(object sender, EventArgs e)
16 {
17     Session["isLoggedIn"] = "False";
18     Session["currentUser"] = "";
19     Session["userIsAdmin"] = "False";
20     Session["userType"] = "";
21     customLogging.newEntry("User logged out");
22     Server.Transfer("~/default.aspx", false);
23 }
24
25 protected void employeeIn_Click(object sender, EventArgs e)
26 {
27     Server.Transfer("~/employeeLogin.aspx", false);
28 }
29
30 protected void customerRedirect_Click(object sender, EventArgs e)
31 {
32     Server.Transfer("~/default.aspx", false);
33 }
34
```

Page_Load serves a similar purpose to on the customer page, checking sessions and performing the appropriate actions. First and foremost, if the user isn't marked as an employee with userType, they are immediately redirected to the customer home page. This prevents unauthorized access to the employee side of the website, so all page which inherit from this master page will be inaccessible to anyone not marked as an employee. The username label is set to show a welcome message and the sign out button is set to be visible.

signOut_Click is the event method for when the sign out button is clicked. It does exactly the same thing as the sign out button on the customer master page, clearing all sessions and redirecting the user to the customer home page.

employeeIn_Click just redirects the user to the employee login page.

customerRedirect_Click just redirects the user to the customer half of the site because this is difficult to access for an employee otherwise.

3.2.7 Log In

```csharp
public partial class customerLogin : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if (Convert.ToString(Session["isLoggedIn"]) == "True")
        {
            Server.Transfer("~/default.aspx", false);
        }
    }

    protected void submitCustomerCredentialsButton_Click(object sender, EventArgs e)
    {
        if (customSecurity.sanitizeCheck(new string[] { customerUsernameBox.Text,
customerPasswordBox.Text }))
        {
            string attemptedName = customerUsernameBox.Text;
            using (var checkCredentials = new defaultDataSetTableAdapters.
customersTableAdapter())
            {
                if (checkCredentials.loginCheck(attemptedName, customSecurity.
generateMD5(customerPasswordBox.Text)) != null)
                {
                    Session["isLoggedIn"] = "True";
                    Session["currentUser"] = attemptedName;
                    Session["userType"] = "customer";
                    Session["userIsAdmin"] = "False";
                    customLogging.newEntry("Customer " + attemptedName + " logged in");
                    Server.Transfer("~/default.aspx", false);
                }
                else
                {
                    customerLoginReturnLabel.Text = "The Username or Password is
incorrect.";
                    customLogging.newEntry("Someone attempted to login as a customer
with username '" + attemptedName + "' but the credentials were incorrect");
                }
            }
        }
        else
        {
            customerLoginReturnLabel.Text = customSecurity.sanitizeErrorMessage;
        }
    }

    protected void registerButton_Click(object sender, EventArgs e)
    {
        Server.Transfer("registerCustomer.aspx", false);
    }

    protected void employeeRedirect_Click(object sender, EventArgs e)
    {
        Server.Transfer("employeeLogin.aspx", false);
    }
}
```

The Page There are two log in pages in my project, one for the customers and one for the employees; because the customer and employee tables are seperate, it is more convenient to seperate the logic for each table into two seperate pages/files. In terms of appearance, the two pages are identical but for the inclusion of a button on the customer page which redirects the user to the customer register page. Otherwise, both pages contain a text box for the user name of the user and their password (customerUsernameBox and customerPasswordBox or equivalent), a submit credentials button (submitCustomerCredentialsButton or equivalent) and finally a button to redirect the user to the employee login or customer login as appropriate (employeeRedirect or equivalent). Both pages inherit from Customer.Master.

Page_Load This page should not be accessible by people that are already logged in, as logging in again is unnecessary if they are already logged in and could cause unforeseen problems. This checks the session "isLoggedIn", and if it tagged as true then it redirects the user to the relevant homepage with the Server.Transfer method (this is used on almost all pages to redirect people - it's arguments are the path of the page to redirect to and whether to "preserve" the form it is currently loading, for my purposes this is always false). In the employee file, it will redirect to the employee homepage, which will then redirect to the customer homepage if it is a customer logged in.

submitCustomerCredentialsButton_click A method called when the submit button is clicked (ASP automatically generates blank methods linked to each button's onclick event, comprised of the name of the button followed by "_click"). Within this method, I have used two nested if statements with boolean logic to check the inputs for validity. The first ensures that both input boxes are free of SQL characters with customSecurity.sanitizeCheck and returns the appropriate error message to the return box if not, while the second runs a query against the database:

```
1  SELECT COUNT(*) AS Expr1
2  FROM customers
3  GROUP BY username, [password]
4  HAVING (username = ?) AND ([password] = ?)
5
```

This query will return a DataRow type if an entry with matching credentials is found, and null if not. The boolean logic in the if statement checks if something was returned, and if it was it runs the code within it. If null is returned then it will not run anything and change the error message label to inform the user they have input incorrect credentials. The code within the if statement sets all the appropriate session data as described previously, using the username string the user gave as the username and setting the administrator & type values to false and customer respectively on this page. It will also log the user entering the site. On the employee page, lines 24 to 26 are replaced with:

```
1  Session["userIsAdmin"] = Convert.ToBoolean(loginCheck);
2  if (Convert.ToBoolean(Session["userIsAdmin"]))
3  {
4      customLogging.newEntry("Admin " + attemptedName + " logged in");
5  }
6  else
7  {
8      customLogging.newEntry("Employee " + attemptedName + " logged in");
9  }
10 Server.Transfer("~/staffOverview.aspx", false);
11
```

The changes this code introduces are that the return of the query is converted into a boolean with the default static class Convert - this works because the query run for that page is

```
1  SELECT admin
2  FROM employees
3  GROUP BY username, [password], admin
4  HAVING (username = ?) AND ([password] = ?)
5
```

Which returns just the value of the admin field instead of the whole DataRow so the only value for Convert to operate on is the one we want. The resulting value is stored in the userIsAdmin session. The same session is immediately checked again to decide what to post to the log; if it returns true then the log is told an administrator logged on and if false that an employee logged on. The last line redirects the newly-logged in user to the staff overview page instead of the customer home page.

3.2.8 Product Class

My project needs to deal with lots of information about the products of the client constantly as part of it's base function. The code which deals with this is best abstracted enough to be omni-purpose across the site, with little modification needed to use it on different pages for different purposes. Most pages related to products need to be able to display a list of products which can be sorted and filtered - to this end I decided to write a instantiable class representing one product and attach code to it that would let it gather it's data for itself. Eventually, I wrote the following:

```
public class product
{
    private defaultDataSetTableAdapters.productsTableAdapter adapter = new
    defaultDataSetTableAdapters.productsTableAdapter();

    private productStruct ProductInfo = new productStruct();
    public productStruct productInfo
    {
        get
        {
            return ProductInfo;
        }
    }
    public int newStock { get; set; }

    public product(string iniName)
    {
        getData(iniName);
    }
    public product(DataRow row)
    {
        ProductInfo.productName = Convert.ToString(row[0]);
        ProductInfo.displayName = Convert.ToString(row[3]);
        ProductInfo.stock = Convert.ToInt32(row[1]);
        ProductInfo.price = Convert.ToDecimal(row[2]);
        ProductInfo.band = Convert.ToString(row[7]);
        ProductInfo.description = Convert.ToString(row[8]);
        ProductInfo.imagePath = Convert.ToString(row[6]);
        ProductInfo.type = Convert.ToString(row[4]);
    }

    public void saveStock()
    {
        adapter.updateStock(newStock, ProductInfo.productName);
        ProductInfo.stock = newStock;
        newStock = 0;
    }

    public void refresh()
    {
        getData(ProductInfo.productName);
    }

    private void getData(string searchName)
    {
        var rows = adapter.getDataTable(searchName);
        DataRow row = rows[0];
        ProductInfo.productName = searchName;
        ProductInfo.displayName = Convert.ToString(row[3]);
        ProductInfo.stock = Convert.ToInt32(row[1]);
        ProductInfo.price = Convert.ToDecimal(row[2]);
        ProductInfo.band = Convert.ToString(row[7]);
        ProductInfo.description = Convert.ToString(row[8]);
        ProductInfo.imagePath = Convert.ToString(row[6]);
        ProductInfo.type = Convert.ToString(row[4]);
    }
}
```

Properties and Variables A property in C# is an attribute of a class which allows outside code to read and write to objects within it. This can be used to make certain values read-only or write-only, or attach code to such events to control what conditions the object(s) can be accessed under. It is good practice to implement properties instead of tagging objects within as public because it ensures outside code cannot accidently or deliberately interfere with the objects within the class; while I will be the only contributor to this application's codebase this is not strictly necessary because I know how the class works and don't need to interfere with it, it remains good practice to implement. The properties and variables defined in lines 3 to 13 are as follows:

- **adapter** is the adaptor which the product will use to pull information from.

- **ProductInfo** a custom structure which holds all the information about the product in an uncomplicated, grouped object. It's structure is:

```
1    public struct productStruct
2    {
3        public string productName;
4        public string displayName;
5        public int stock;
6        public decimal price;
7        public string band;
8        public string description;
9        public string imagePath;
10       public string type;
11   }
12
```

  – productName is the camel case name of the product - camel case does not contain spaces and capitalizes the first letter of all words/components but the first. I chose this because it maximizes compatibility since it doesn't have any spaces, which can make string manipulation on it easier, and so that strange capitalizations won't allow duplicate products in the data base. Also, it is in keeping with my method of using camel case for all my variables and objects in my code.

  – displayName is the original name the employee who created the product set as the name.

  – stock is an integer value showing how much stock is left in the inventory.

  – price is a decimal detailing how much, in GBP, one unit of the product costs.

  – description is a written description of the product which is defined when the product is created.

  – imagePath is a file path to the image of the product. It should ideally be images/product-name.png.

  – type is the type of product it is, i.e. coaster or clock.

  An additional advantage is that using a struct like this allows me to assign a name to each field instead of having to check by index.

- **productInfo** is the property used to access the information in ProductInfo. It is read only.

- **newStock** This is a simple integer that can be read or written to which a later method will use to change the stock value of a product in the database.

Constructors In C#, constructors are declared with "public objectName(arguments)", without a type specifier and with the same name as the parent object. There are two contructors in this class - they are overloaded (i.e. the compiler chooses which one to use based on what parameters are given when the object is instantiated):

1. Beginning line 15, the base constructor takes a string holding the productName of the product to load into the new instance. It runs the getData() method using this, explained below.

41

2. Beginning line 19, the secondary constructor takes a full datarow which would typically be returned by a DataSet query and assigns all of the information in productStruct to the correct fields in that row.

Methods   There are three methods in the product class:

- **saveStock** uses adaptor to write the value of newStock to the database with the query updateStock, which consists of:

```
1    UPDATE products
2    SET stock = ?
3    WHERE (productName = ?)
4
```

  This changes the value of the stock field in all entries where productName matches what is given. saveStock also updates the stock value in ProductInfo and resets the value of newStock to 0.

- **refresh** runs getData again using the productName variable in ProductInfo.

- **getData** is a private method for use only within the class which runs the query getDataTable using adaptor. getDataTable contains:

```
1    SELECT productName, stock, price, displayName, productType, creator, [image], [
     band], description
2    FROM products
3    WHERE (productName = ?)
4
```

  This searches for productName in the table and returns the entire contents of the table when it finds a match. getData uses that information to populate ProductInfo using the appropriate data fields.

The product class itself is relatively barebones since most of the logic related to organising and presenting products is contained in later objects.

3.2.9 productPanel Class

I needed a way to show products consistently across all pages and manually writing the code for displaying products on every page that needed it would take more time and be less efficient than alternative solutions. To accomplish this, I wrote a class which inherits from the ASP.NET Panel, giving me the advantage of just being able to add the object as it is to forms so it will just render itself.

```
public class productPanel : System.Web.UI.WebControls.Panel
{
    public productStruct info;
    public productPanel(productStruct productInfo)
    {
        info = productInfo;

        this.CssClass = "productDisplayPanel";
        this.ID = info.productName + "Panel";

        Image displayedImage = new Image()
        {
            CssClass = "productImage",
            ID = info.productName + "ImageTag",
            ImageUrl = info.imagePath,
            Height = 400,
            Width = 400
        };
        displayedImage.Attributes.Add("runat", "server");
        this.Controls.Add(displayedImage);
        this.Controls.Add(new LiteralControl("<br />"));

        Label nameTag = new Label()
        {
            CssClass = "nameTag",
            Text = info.displayName,
            ID = info.productName + "NameTag"
        };
        nameTag.Attributes.Add("runat", "server");
        this.Controls.Add(nameTag);
        this.Controls.Add(new LiteralControl("<br />"));

        Label priceTag = new Label()
        {
            CssClass = "priceTag",
            Text = "  " + commonClasses.common.formatPrice(info.price),
            ID = info.productName + "PriceTag"
        };
        priceTag.Attributes.Add("runat", "server");
        this.Controls.Add(priceTag);

        Label stockTag = new Label()
        {
            CssClass = "stockTag",
            ID = info.productName + "StockTag"
        };
        if (info.stock == 0)
        {
            stockTag.Text = "Out of stock";
        }
        else
        {
            stockTag.Text = Convert.ToString(info.stock) + " in stock";
        }
        stockTag.Attributes.Add("runat", "server");
        this.Controls.Add(stockTag);
        this.Controls.Add(new LiteralControl("<br />"));

        Label descriptionTag = new Label()
        {
            CssClass = "descriptionTag",
```

```
62              Text = Convert.ToString(info.description),
63              ID = info.productName + "DescriptionTag"
64          };
65          descriptionTag.Attributes.Add("runat", "server");
66          this.Controls.Add(descriptionTag);
67          this.Controls.Add(new LiteralControl("<br />"));
68      }
69 }
70
```

Properties   The only property in this class is productInfo, which is just a productStruct initialized in the constructor.

Contructor   This class has only one constructor, which serves a double purpose as the only method in the class. The constructor initializes a set of controls which together make up the whole panel and contain all information the panel should display. First, the ID and CssClass of the panel itself using the dynamic referrer "this". For each individual control:

1. The ID and CssClass are set. The ID is set to be the name of the product the panel is being generated ofr appended with the name of the control (for the panel, an example ID would be "sampleClock3Panel"). The CSS class is simply the name of the object because all similar objects need the same CSS class to have the same styling applied to them.

2. Specific properties for each control are set - see below for these.

3. The control has the runat Attribute set to "server" using Attributes.Add() so that it runs server-side, not client-side, which would cause persistency problems.

4. The control is added to the panel's controls array using this.Controls.Add() - the controls array is an array in productPanel inherited from Panel which contains all the controls which are displayed within the panel.

5. An HTML
 (breakline, essentially a manual "return" key) is added by instantiating a LiteralControl in order to create some space between elements.

The individual controls are:

- **displayedImage** is the thumbnail (small cover image) for the product. It is set to display the image at the path at the imagePath property of the info productStruct and to be 500500 so all thumbnails are the same size.4

- **nameTag** is the label which will show the name of the product. It's Text property is set to the displayName property of info.

- **priceTag** displays the price of 1 unit of that product. It's Text property is set to the stored price of the product from info and prefixed with a "" character.

- **stockTag** is a label which just shows the integer value of how much of that product is left in the inventory. It's Text property is set to the stock value of info.

- **descriptionTag** is a label which will hold the description of the product. It's Text property is set to the description of the product using info.

Revision 29/04/18 There are some issues with styling using the current layout. I decided to create another panel to hold all the elements except the image so that I could apply styling to the positioning of the non-image elements together:

```
public product info { get; }
public productPanel(product productInfo)
{
    info = productInfo;

    this.CssClass = "productDisplayPanel";
    this.ID = info.productName + "Panel";

    Image displayedImage = new Image()
    {
        CssClass = "productImage",
        ID = info.productName + "ImageTag",
        ImageUrl = "~/images/" + info.imagePath,
        Height = 100,
        Width = 100
    };
    displayedImage.Attributes.Add("runat", "server");
    this.Controls.Add(displayedImage);

    Panel textWrapper = new Panel()
    {
        CssClass = "textWrapper",
        ID = info.productName + "TextWrapper"
    };
    textWrapper.Attributes.Add("runat", "server");

    Label nameTag = new Label()
    {
        CssClass = "nameTag",
        Text = info.displayName,
        ID = info.productName + "NameTag"
    };
    nameTag.Attributes.Add("runat", "server");
    textWrapper.Controls.Add(nameTag);

    Label priceTag = new Label()
    {
        CssClass = "priceTag",
        Text = "   " + commonClasses.common.formatPrice(info.price),
        ID = info.productName + "PriceTag"
    };
    priceTag.Attributes.Add("runat", "server");
    textWrapper.Controls.Add(priceTag);

    Label stockTag = new Label()
    {
        CssClass = "stockTag",
        ID = info.productName + "StockTag"
    };
    if (info.stock == 0)
    {
        stockTag.Text = "Out of stock";
    }
    else
    {
        stockTag.Text = Convert.ToString(info.stock) + " in stock";
    }
    stockTag.Attributes.Add("runat", "server");
    textWrapper.Controls.Add(stockTag);

    Label descriptionTag = new Label()
    {
        CssClass = "descriptionTag",
        Text = Convert.ToString(info.description),
```

```
65          ID = info.productName + "DescriptionTag"
66      };
67      descriptionTag.Attributes.Add("runat", "server");
68      textWrapper.Controls.Add(descriptionTag);
69
70      this.Controls.Add(textWrapper);
71 }
72
```

3.2.10 productList Class

This class contains all the logic for sorting, filtering and otherwise comparing products. Since the class is extremely large, I have split it up into groups of similar functions; each section has an accompanying code block, all of which are in order. I have omitted the class declaration to make the beginning and end sections easier to read, all of the following is encapsulated in ordinary "class productList " tags.

```
1   private defaultDataSetTableAdapters.productsTableAdapter adapter = new
        defaultDataSetTableAdapters.productsTableAdapter();
2   private product[] masterList;
3   private product[] WorkingList;
4   public product[] list
5   {
6     get
7     {
8       return WorkingList;
9     }
10  }
11
```

- **adapter** is, as before, the adapter the productList will use to interface with the database.

- **masterList** is one of two arrays of type product; this one stores the whole contents of the database table when the class is initialized. The class won't change the contents of this once it is initialized or it is specifically instructed to.

- **workingList** is the "primary" product array; it is copied from the master list upon initialization then all filtering and sorting is applied to it. The advantage of this method is that when the sort or filter needs to be reset, the database does not have to accessed twice which would make the application slower.

- **list** is the property allowing outside access to the working list. The master list is private with no property to prevent outside code from interfering with the contents of it because if it did, productList might not function correctly.

Basic Methods and Constructors   The class has a large amount of constructor overloads to account for all possible situations which serve to make it more adaptible. It can initialize using an array of products, an array of the names of products, a DataTable of products and, by default, from all the products in the table. Many of these features may prove to be redundant later and I may not use some so it is unlikely all these construction methods will survive into the finished project.

```
1   public void resetWorkingList()
2   {
3       WorkingList = masterList;
4   }
5
6   public Panel[] generateControls()
7   {
8       Panel[] result = new Panel[WorkingList.Length];
9       for (int i = 0; i < WorkingList.Length; i++)
10      {
11          result[i] = new productPanel(WorkingList[i].productInfo);
12      }
13      return result;
14  }
15
16  public void setWorkingList(product[] array)
17  {
18      WorkingList = array;
19  }
20
```

```
21 public productList()
22 {
23     var data = adapter.GetData();
24     int i = 0;
25     masterList = new product[data.Rows.Count];
26     foreach (DataRow row in data.Rows)
27     {
28         masterList[i] = new product(row);
29         i++;
30     }
31     WorkingList = masterList;
32 }
33 public productList(string[] productNames)
34 {
35     int i = 0;
36     foreach (string str in productNames)
37     {
38         masterList[i] = new product(str);
39         i++;
40     }
41 }
42 public productList(DataTable data)
43 {
44     int i = 0;
45     foreach (DataRow row in data.Rows)
46     {
47         masterList[i] = new product(row);
48         i++;
49     }
50     WorkingList = masterList;
51 }
52 public productList(product[] products)
53 {
54     masterList = products;
55     WorkingList = masterList;
56 }
57
```

The constructors are, in the order they are laid out in the code:

1. The first constructor pulls the whole contents of the database into the class. It uses the adaptor to get a DataTable (declared with the keyword "var" here, which sets the type to the appropriate one to store what it's being set to). The master list is initialized as a blank array with the same length as there are rows in the datatable, so there are exactly the right amount of elements to store all currently registered products.

2. Uses an array of strings containing the names of the products that should be put into the master list. This uses the constructor for the product class that takes a string containing the name of the product to initialize and generates a product for each entry in the string. Letting each product initialize itself could prove to be a slow method as a database call must be made for each product which could create massive latency issues but I will review in testing.

3. Uses a premade DataTable passed to it on initialization to initialize each product in the list. Works similarly to #1 but doesn't create it's own table. Having this constructor lets me run LINQ queries on the table in other objects to filter it, which can sometimes be more powerful than just using SQL and moreover can be checked and debugged inside Visual Studio effectively.

Revision 24/04/18 I have changed some of the constructors prior to testing because upon reading back through, some clearly won't work as they don't set the working list or actually initialize the master list at all. I have added masterList initialization to the second and third constructors, and WorkingList intialization to the second constructor:

```
1 public productList()
2 {
```

```
3        var data = adapter.GetData();
4        int i = 0;
5        masterList = new product[data.Rows.Count];
6        foreach (DataRow row in data.Rows)
7        {
8            masterList[i] = new product(row);
9            i++;
10       }
11       WorkingList = masterList;
12   }
13   public productList(string[] productNames)
14   {
15       int i = 0;
16       masterList = new product[productNames.Length];
17       foreach (string str in productNames)
18       {
19           masterList[i] = new product(str);
20           i++;
21       }
22       WorkingList = masterList;
23   }
24   public productList(DataTable data)
25   {
26       int i = 0;
27       masterList = new product[data.Rows.Count];
28       foreach (DataRow row in data.Rows)
29       {
30           masterList[i] = new product(row);
31           i++;
32       }
33       WorkingList = masterList;
34   }
35   public productList(product[] products)
36   {
37       masterList = products;
38       WorkingList = masterList;
39   }
40
```

There are several basic methods which are included above the constructors, detailed here:

- **resetWorkingList** just discards the current workingList and sets it to the contents of the master list. It is there for when sorting and filtering need to be cleared and removed.

- **generateControls** generates an array of productPanels, one for each product in the working list and returns it.

- **setWorkingList** exists purely so that outside classes can set the contents of the workingList. This is largely just a contingency for the rare occasion where I may need to set the WorkingList outside the class but I suspect I will not need it and so it will also likely not make it into the final application.

Filtering and Sorting Unfortunately, the following code is extremely long and so this section will be more difficult to follow. The class contains extensive logic for filtering and sorting the contents of the workingList, primarily for the use of the storefront page (the code has been kept within the class without any web based dependencies in the event that I need to be able to use this logic on any other page). The sheer length of this code is mostly due to a limitation of C# which doesn't allow generics to filter what types they accept; if this wasn't the case I would be able to make a highly abstracted, generic quicksort function which all these methods could use but as it is I can't dictate what types the function would accept.

All the logic is presented to outside code using only two methods, simply "sort" and "filter". These methods use a switch to find what type of sort to apply and what field to apply it to; this keeps the outside appearance simple and interaction with the class understandable, letting the code within the class organise the implementation of quicksort and filtering.

```
public void sort(bool ascending, string sortType)
{
    switch (sortType)
    {
        case "price":
            WorkingList = sortPrice(WorkingList, ascending);
            break;
        case "stock":
            WorkingList = sortStock(WorkingList, ascending);
            break;
        case "name":
            WorkingList = sortName(WorkingList, ascending);
            break;
        case "band":
            WorkingList = sortBand(WorkingList, ascending);
            break;
        default:
            throw new System.ArgumentException("The sort type must be one of the
    specified values.");
    }
}

private static product[] sortPrice(product[] input, bool ascending)
{
    if (input.Length <= 1)
    {
        return input;
    }
    else if (input.Length == 2)
    {
        if ((input[0].productInfo.price > input[1].productInfo.price && ascending) || (
    input[0].productInfo.price < input[1].productInfo.price && !ascending))
        {
            product temp = input[0];
            input[0] = input[1];
            input[1] = temp;
        }
    }

    product[] subArray;
    List<product> subList = new List<product>();
    product[] superArray;
    List<product> superList = new List<product>();
    int pivotIndex = Convert.ToInt32(Math.Ceiling(Convert.ToDouble(input.Length) / 2)) -
     1;

    for (int i = 0; i < input.Length; i++)
    {
        if (i == pivotIndex)
        {
            continue;
```

```
49              }
50          else if (input[i].productInfo.price <= input[pivotIndex].productInfo.price)
51          {
52              if (ascending)
53              {
54                  subList.Add(input[i]);
55              }
56              else
57              {
58                  superList.Add(input[i]);
59              }
60          }
61          else if (input[i].productInfo.price > input[pivotIndex].productInfo.price)
62          {
63              if (ascending)
64              {
65                  superList.Add(input[i]);
66              }
67              else
68              {
69                  subList.Add(input[i]);
70              }
71          }
72      }
73
74      subArray = subList.ToArray();
75      superArray = superList.ToArray();
76      subArray = sortPrice(subArray, ascending);
77      superArray = sortPrice(superArray, ascending);
78
79      product[] result;
80      result = commonClasses.common.appendArray(subArray, input[pivotIndex]);
81      result = commonClasses.common.appendArray(result, superArray);
82
83      return result;
84 }
85
86 private static product[] sortStock(product[] input, bool ascending)
87 {
88      if (input.Length <= 1)
89      {
90          return input;
91      }
92      else if (input.Length == 2)
93      {
94          if ((input[0].productInfo.stock > input[1].productInfo.stock && ascending) || (
      input[0].productInfo.stock < input[1].productInfo.stock && !ascending))
95          {
96              product temp = input[0];
97              input[0] = input[1];
98              input[1] = temp;
99          }
100         return input;
101     }
102
103     product[] subArray;
104     List<product> subList = new List<product>();
105     product[] superArray;
106     List<product> superList = new List<product>();
107     int pivotIndex = Convert.ToInt32(Math.Ceiling(Convert.ToDouble(input.Length) / 2)) -
       1;
108
109     for (int i = 0; i < input.Length; i++)
110     {
111         if (i == pivotIndex)
112         {
113             continue;
114         }
```

51

```
115         else if (input[i].productInfo.stock <= input[pivotIndex].productInfo.stock)
116         {
117             if (ascending)
118             {
119                 subList.Add(input[i]);
120             }
121             else
122             {
123                 superList.Add(input[i]);
124             }
125         }
126         else if (input[i].productInfo.stock > input[pivotIndex].productInfo.stock)
127         {
128             if (ascending)
129             {
130                 superList.Add(input[i]);
131             }
132             else
133             {
134                 subList.Add(input[i]);
135             }
136         }
137     }
138
139     subArray = subList.ToArray();
140     superArray = superList.ToArray();
141     subArray = sortStock(subArray, ascending);
142     superArray = sortStock(superArray, ascending);
143
144     product[] result;
145     result = commonClasses.common.appendArray(subArray, input[pivotIndex]);
146     result = commonClasses.common.appendArray(result, superArray);
147
148     return result;
149 }
150
151 private static product[] sortBand(product[] input, bool ascending)
152 {
153     if (input.Length <= 1)
154     {
155         return input;
156     }
157     else if (input.Length == 2)
158     {
159         if ((input[0].productInfo.band.CompareTo(input[1].productInfo.band) > 0 &&
    ascending) || (input[0].productInfo.band.CompareTo(input[1].productInfo.band) < 0 &&
     !ascending))
160         {
161             product temp = input[0];
162             input[0] = input[1];
163             input[1] = temp;
164         }
165         return input;
166     }
167
168     product[] subArray;
169     List<product> subList = new List<product>();
170     product[] superArray;
171     List<product> superList = new List<product>();
172     int pivotIndex = Convert.ToInt32(Math.Ceiling(Convert.ToDouble(input.Length) / 2)) -
     1;
173
174     for (int i = 0; i < input.Length; i++)
175     {
176         if (i == pivotIndex)
177         {
178             continue;
179         }
```

```
180          else if (input[i].productInfo.band.CompareTo(input[pivotIndex].productInfo.band)
         <= 0)
181          {
182              if (ascending)
183              {
184                  subList.Add(input[i]);
185              }
186              else
187              {
188                  superList.Add(input[i]);
189              }
190          }
191          else if (input[i].productInfo.band.CompareTo(input[pivotIndex].productInfo.band)
         > 0)
192          {
193              if (ascending)
194              {
195                  superList.Add(input[i]);
196              }
197              else
198              {
199                  subList.Add(input[i]);
200              }
201          }
202      }
203
204      subArray = subList.ToArray();
205      superArray = superList.ToArray();
206      subArray = sortName(subArray, ascending);
207      superArray = sortName(superArray, ascending);
208
209      product[] result;
210      result = commonClasses.common.appendArray(subArray, input[pivotIndex]);
211      result = commonClasses.common.appendArray(result, superArray);
212
213      return result;
214 }
215
216 private static product[] sortName(product[] input, bool ascending)
217 {
218      if (input.Length <= 1)
219      {
220          return input;
221      }
222      else if (input.Length == 2)
223      {
224          if ((input[0].productInfo.displayName.CompareTo(input[1].productInfo.displayName
         ) > 0 && ascending) || (input[0].productInfo.displayName.CompareTo(input[1].
         productInfo.displayName) < 0 && !ascending))
225          {
226              product temp = input[0];
227              input[0] = input[1];
228              input[1] = temp;
229          }
230          return input;
231      }
232
233      product[] subArray;
234      List<product> subList = new List<product>();
235      product[] superArray;
236      List<product> superList = new List<product>();
237      int pivotIndex = Convert.ToInt32(Math.Ceiling(Convert.ToDouble(input.Length) / 2)) -
          1;
238
239      for (int i = 0; i < input.Length; i++)
240      {
241          if (i == pivotIndex)
242          {
```

```
243                continue;
244            }
245            else if (input[i].productInfo.displayName.CompareTo(input[pivotIndex].
       productInfo.displayName) <= 0)
246            {
247                if (ascending)
248                {
249                    subList.Add(input[i]);
250                }
251                else
252                {
253                    superList.Add(input[i]);
254                }
255            }
256            else if (input[i].productInfo.displayName.CompareTo(input[pivotIndex].
       productInfo.displayName) > 0)
257            {
258                if (ascending)
259                {
260                    superList.Add(input[i]);
261                }
262                else
263                {
264                    subList.Add(input[i]);
265                }
266            }
267        }
268
269        subArray = subList.ToArray();
270        superArray = superList.ToArray();
271        subArray = sortName(subArray, ascending);
272        superArray = sortName(superArray, ascending);
273
274        product[] result;
275        result = commonClasses.common.appendArray(subArray, input[pivotIndex]);
276        result = commonClasses.common.appendArray(result, superArray);
277
278        return result;
279 }
280
```

I am going to explain the central method and genericize the four sort methods since they are identical, but compare different fields. It should be noted that in order to compare strings, I have used the string.CompareTo() method, which returns 1 if the string provided comes before the string the method is called from in the alphabet.

- **sort** is the method which manages all the other methods. It takes a boolean which dictates whether to sort by ascending or descending (true for ascending and false for descending) and a string which represent the sort type. The string would do well to be replaced by an enumerator which would mean only a handful of options could be picked and enforced, as opposed to my current solution which just throws an exception if the contents of the string is not recognised as a sort type. The body of this method is a switch which determines what to sort by, and calls the appropriate method to sort the working list.

- **Individual Sorts** These are the four methods which actually contain the quicksort logic:
    - sortPrice
    - sortStock
    - sortBand
    - sortName

  Each method takes two parameters, the ascending/descending bool and the product[] list to be sorted. The main method calls them using the bool that it itself was passed, and points them at

the working list. Each one contains an almost line-by-line accurate implementation of the quicksort algorithm I outlined in the algorithms section; with the addition of a check whether or not to sort by ascending inserted into the main comparison logic. In other words, I have added an extra "else if" to check which list to add the current element to depending on the ascending bool.

Filtering   The code for filtering is much shorter as all that is needed is a for loop that goes through each element and checks if it needs removing or not. As such, there is only one filter function which, similarly to the sort() method, contains a switch which dictates what to filter by.

```
1  public void filter(string field, string value, bool whitelist)
2  {
3      List<product> filteredList = new List<product>();
4      int filteredIndex = 0;
5      switch (field)
6      {
7          case "productName":
8              for (int i = 0; i < masterList.Length; i++)
9              {
10                 if (whitelist)
11                 {
12                     if (masterList[i].productInfo.displayName.ToUpper() == value.ToUpper
   ())
13                     {
14                         filteredList.Add(masterList[i]);
15                         filteredIndex++;
16                     }
17                 }
18                 else
19                 {
20                     if (masterList[i].productInfo.displayName.ToUpper() != value.ToUpper
   ())
21                     {
22                         filteredList.Add(masterList[i]);
23                         filteredIndex++;
24                     }
25                 }
26             }
27             WorkingList = filteredList.ToArray();
28             break;
29         case "band":
30             for (int i = 0; i < masterList.Length; i++)
31             {
32                 if (whitelist)
33                 {
34                     if (masterList[i].productInfo.band.ToUpper() == value.ToUpper())
35                     {
36                         filteredList.Add(masterList[i]);
37                         filteredIndex++;
38                     }
39                 }
40                 else
41                 {
42                     if (masterList[i].productInfo.band.ToUpper() != value.ToUpper())
43                     {
44                         filteredList.Add(masterList[i]);
45                         filteredIndex++;
46                     }
47                 }
48             }
49             WorkingList = filteredList.ToArray();
50             break;
51
52         case "type":
53             for (int i = 0; i < masterList.Length; i++)
54             {
55                 if (whitelist)
```

```
56                {
57                    if (masterList[i].productInfo.type.ToUpper() == value.ToUpper())
58                    {
59                        filteredList.Add(masterList[i]);
60                        filteredIndex++;
61                    }
62                }
63                else
64                {
65                    if (masterList[i].productInfo.type.ToUpper() != value.ToUpper())
66                    {
67                        filteredList.Add(masterList[i]);
68                        filteredIndex++;
69                    }
70                }
71            }
72            WorkingList = filteredList.ToArray();
73            break;
74        default:
75            throw new ArgumentException("The filter field must be one of the specified
    values");
76        }
77 }
78
```

Within this code, there are multiple if statements apparently needlessly repeated since they all do essentially the same thing, just to different fields. This is indeed an inefficient solution but there doesn't appear to be any alternative. Revisions will probably follow.

The **filter** method takes two strings, one for the field and one for the value to search for, and a boolean which dictates whether it it whitelisting or blacklisting (true for whitelist, false for blacklist). It initializes a List class of type product and uses a switch to determine what code to execute, depending on the value of the field string passed to the method. Within each case, a for loop runs which goes through element by element; within that for loop is an if statement checking whether or not we are whitelisting and within each of those code which adds the current value to the earlier initialized list depending on whether it matches the search or it doesn't (according to the whitelist/blacklist). At the end of the for loop, the working list is set to the output of List.ToArray() which is explained earlier.

```
1 public void removeOutOfStock()
2 {
3     List<product> filteredList = new List<product>();
4     int filteredIndex = 0;
5     for (int i = 0; i < masterList.Length; i++)
6     {
7         if (masterList[i].productInfo.stock > 0)
8         {
9             filteredList.Add(masterList[i]);
10            filteredIndex++;
11        }
12    }
13    WorkingList = filteredList.ToArray();
14 }
15 }
16
```

This last piece of code is still designed for filtering but exists as it's own method because it is more convenient. The method simply makes a new list and adds all the objects from the masterlist which are not out of stock using a for loop, converts it to an array and sets that to be the working list.

Revision 18/04/18 I have made multiple changes to the product handling code since writing this annotation.

1. I have completely removed productStruct as it served little purpose but to be a slave to product. All code which references productStruct can easily be either deleted or modified by just removing the ".productInfo" from "product.productInfo.field". Instead, the product class now just contains all the information as individual properties, the code initializing a productStruct in product has been replaced with this:

```
1  public readonly string productName;
2  public readonly string displayName;
3  private int Stock;
4  public int stock { get { return Stock; } }
5  public readonly decimal price;
6  public readonly string band;
7  public readonly string description;
8  public readonly string imagePath;
9  public readonly string type;
10 public int newStock { get; set; }
11
```

These are all the needed variables for the product just as they were in productStruct but on their own. Stock needs to be written to in some code within the class so I have created the variable "Stock" alongside the property "stock" which allows outside code to view it but not edit while still retaining that functionality for inside code.

2. In the "sort" and "filter" methods, I have implemented what I described above; the switches now rely on an enum instead of a string which means the default case can be removed. The relevant code:

```
1  public enum sortType { name,band,stock,price }
2  public enum filterType { name,band,type }
3
4  public void sort(bool ascending, sortType type)
5  {
6      switch (type)
7      {
8          case sortType.price:
9              WorkingList = sortPrice(WorkingList, ascending);
10             break;
11         case sortType.stock:
12             WorkingList = sortStock(WorkingList, ascending);
13             break;
14         case sortType.name:
15             WorkingList = sortName(WorkingList, ascending);
16             break;
17         case sortType.band:
18             WorkingList = sortBand(WorkingList, ascending);
19             break;
20     }
21 }
22
23 public void filter(filterType field, string value, bool whitelist)
24 {
25     List<product> filteredList = new List<product>();
26     int filteredIndex = 0;
27     switch (field)
28     {
29         case filterType.name:
30             for (int i = 0; i < masterList.Length; i++)
31             {
32                 if (whitelist)
33                 {
34                     if (masterList[i].displayName.ToUpper() == value.ToUpper())
35                     {
36                         filteredList.Add(masterList[i]);
```

```
37                              filteredIndex++;
38                          }
39                      }
40                      else
41                      {
42                          if (masterList[i].displayName.ToUpper() != value.ToUpper())
43                          {
44                              filteredList.Add(masterList[i]);
45                              filteredIndex++;
46                          }
47                      }
48                  }
49                  break;
50          case filterType.band:
51              for (int i = 0; i < masterList.Length; i++)
52              {
53                  if (whitelist)
54                  {
55                      if (masterList[i].band.ToUpper() == value.ToUpper())
56                      {
57                          filteredList.Add(masterList[i]);
58                          filteredIndex++;
59                      }
60                  }
61                  else
62                  {
63                      if (masterList[i].band.ToUpper() != value.ToUpper())
64                      {
65                          filteredList.Add(masterList[i]);
66                          filteredIndex++;
67                      }
68                  }
69              }
70              break;
71
72          case filterType.type:
73              for (int i = 0; i < masterList.Length; i++)
74              {
75                  if (whitelist)
76                  {
77                      if (masterList[i].type.ToUpper() == value.ToUpper())
78                      {
79                          filteredList.Add(masterList[i]);
80                          filteredIndex++;
81                      }
82                  }
83                  else
84                  {
85                      if (masterList[i].type.ToUpper() != value.ToUpper())
86                      {
87                          filteredList.Add(masterList[i]);
88                          filteredIndex++;
89                      }
90                  }
91              }
92              break;
93      }
94      WorkingList = filteredList.ToArray();
95  }
96
```

3. I refactored the "filter" method. I moved the for loop out of the switch to shorten the code and moved the statement that sent the final list to the working list outside also. Furthermore, I changed the filter to get it's source information from the working list instead of the master list. This is good because it means multiple filters can be applied, without resetting the last one as it would have done before. The new code:

```
1  public void filter(filterType field, string value, bool whitelist)
2  {
3      List<product> filteredList = new List<product>();
4      int filteredIndex = 0;
5      for (int i = 0; i < WorkingList.Length; i++)
6      {
7          switch (field)
8          {
9              case filterType.name:
10                 if (whitelist)
11                 {
12                     if (WorkingList[i].displayName.ToUpper() == value.ToUpper())
13                     {
14                         filteredList.Add(WorkingList[i]);
15                         filteredIndex++;
16                     }
17                 }
18                 else
19                 {
20                     if (WorkingList[i].displayName.ToUpper() != value.ToUpper())
21                     {
22                         filteredList.Add(WorkingList[i]);
23                         filteredIndex++;
24                     }
25                 }
26                 break;
27             case filterType.band:
28                 if (whitelist)
29                 {
30                     if (WorkingList[i].band.ToUpper() == value.ToUpper())
31                     {
32                         filteredList.Add(WorkingList[i]);
33                         filteredIndex++;
34                     }
35                 }
36                 else
37                 {
38                     if (WorkingList[i].band.ToUpper() != value.ToUpper())
39                     {
40                         filteredList.Add(WorkingList[i]);
41                         filteredIndex++;
42                     }
43                 }
44                 break;
45             case filterType.type:
46                 if (whitelist)
47                 {
48                     if (WorkingList[i].type.ToUpper() == value.ToUpper())
49                     {
50                         filteredList.Add(WorkingList[i]);
51                         filteredIndex++;
52                     }
53                 }
54                 else
55                 {
56                     if (WorkingList[i].type.ToUpper() != value.ToUpper())
57                     {
58                         filteredList.Add(WorkingList[i]);
59                         filteredIndex++;
60                     }
61                 }
62                 break;
63         }
64     }
65     WorkingList = filteredList.ToArray();
66 }
67
```

4. (Added after the above) I have discovered a way to refer to a property of a class in C#. This is *indispensable* because it means I can completely refactor the sorting and filtering code. Note that I reverted to using the strings and switches method, I found I needed it this way in another page. The sorting methods can now only be 95 lines without losing any functionality:

```
1  public void sort(bool ascending, string type)
2  {
3      Func<product, IComparable> referrer = null;
4      switch (type)
5      {
6          case "price":
7              referrer = x => x.price;
8              break;
9          case "stock":
10             referrer = x => x.stock;
11             break;
12         case "name":
13             referrer = x => x.displayName;
14             break;
15         case "band":
16             referrer = x => x.band;
17             break;
18     }
19     WorkingList = quicksort(WorkingList);
20
21
22     product[] quicksort(product[] input)
23     {
24         if (input.Length <= 1)
25         {
26             return input;
27         }
28         else if (input.Length == 2)
29         {
30             if (referrer(input[0]).CompareTo(referrer(input[1])) > 0 && ascending
    || (referrer(input[0]).CompareTo(referrer(input[1])) < 0) && !ascending)
31             {
32                 product temp = input[0];
33                 input[0] = input[1];
34                 input[1] = temp;
35             }
36         }
37
38         product[] subArray;
39         List<product> subList = new List<product>();
40         product[] superArray;
41         List<product> superList = new List<product>();
42         int pivotIndex = Convert.ToInt32(Math.Ceiling(Convert.ToDouble(input.Length
    ) / 2)) - 1;
43
44         for (int i = 0; i < input.Length; i++)
45         {
46             if (i == pivotIndex)
47             {
48                 continue;
49             }
50             else if (referrer(input[0]).CompareTo(referrer(input[1])) < 0)
51             {
52                 if (ascending)
53                 {
54                     subList.Add(input[i]);
55                 }
56                 else
57                 {
58                     superList.Add(input[i]);
59                 }
60             }
61             else if (referrer(input[0]).CompareTo(referrer(input[1])) > 0)
```

```
62              {
63                  if (ascending)
64                  {
65                      superList.Add(input[i]);
66                  }
67                  else
68                  {
69                      subList.Add(input[i]);
70                  }
71              }
72          }
73
74          subArray = subList.ToArray();
75          superArray = superList.ToArray();
76          subArray = quicksort(subArray);
77          superArray = quicksort(superArray);
78
79          product[] result;
80          result = commonClasses.common.appendArray(subArray, input[pivotIndex]);
81          result = commonClasses.common.appendArray(result, superArray);
82
83          return result;
84      }
85 }
86
```

There are many changes in the above code. The first is that I have compacted all the sorting methods into one method, then nested it inside the "sort" method. The advantage of nesting is that the method no longer needs to be called with ascending because it already has access to it from it's parent method. **Func<product, IComparable>** is the major breakthrough in this revision - this allows me to set a dynamic reference to any property (of a type which inherits from the interface IComparable) of product that I want. I can set it to any property of product, then calling "referrer(product1).CompareTo(product2)" will return an integer; if it is 0 the two are equal, if above 0 product1's property is bigger than product2's and if below 0 it is smaller. Replacing the appropriate lines in the new quicksort function with this but replacing product1 and product2 with the items being compared makes the function perfectly generic and a good example of abstracted code.

```
1 public void filter(string field, string value, bool whitelist)
2 {
3      List<product> filteredList = new List<product>();
4      Func<product, string> referrer = null;
5      int filteredIndex = 0;
6      switch (field)
7      {
8          case "productName":
9              referrer = x => x.productName;
10             break;
11         case "band":
12             referrer = x => x.band;
13             break;
14         case "type":
15             referrer = x => x.type;
16             break;
17     }
18     for (int i = 0; i < WorkingList.Length; i++)
19     {
20         if (whitelist)
21         {
22             if (referrer(WorkingList[i]).ToUpper() == value.ToUpper())
23             {
24                 filteredList.Add(WorkingList[i]);
25                 filteredIndex++;
26             }
27         }
28         else
```

```
29          {
30              if (referrer(WorkingList[i]).ToUpper() != value.ToUpper())
31              {
32                  filteredList.Add(WorkingList[i]);
33                  filteredIndex++;
34              }
35          }
36      }
37      WorkingList = filteredList.ToArray();
38 }
39
```

I have overhauled the filtering logic as well to use a similar method with **Func<product, string>** which does the same as above but will only operate on properties that are of type string, which I can do because the filter class only needs to operate on strings.

3.2.11 Store Front Page

The store front page is the heart of my application. The C# code is as follows:

```csharp
1  productList productsDisplayList;
2  Panel[] panels;
3  bool initialized = false;
4
5  protected void Page_Load(object sender, EventArgs e)
6  {
7      if (initialized)
8      {
9          populatePage();
10     }
11     else
12     {
13         productsDisplayList = new productList();
14         populatePage();
15         initialized = true;
16     }
17 }
18
19 private void populatePage()
20 {
21     panels = productsDisplayList.generateControls();
22     productsListPanel.Controls.Clear();
23     for (int i = 0; i < panels.Length; i++)
24     {
25         Button detailsButton = new Button()
26         {
27             CssClass = "detailsButton",
28             Text = "View",
29             ID = productsDisplayList.list[i].productName + "DetailsLinkButton",
30             CommandArgument = productsDisplayList.list[i].productName
31         };
32         detailsButton.Click += new EventHandler(detailsButton_Click);
33         detailsButton.Attributes.Add("runat", "server");
34         panels[i].Controls.Add(detailsButton);
35         productsListPanel.Controls.Add(panels[i]);
36     }
37 }
38
39 protected void detailsButton_Click(object sender, EventArgs e)
40 {
41     Button btn = (Button)sender;
42     Session["productRedirectName"] = btn.CommandArgument;
43     Server.Transfer("~/productView.aspx", false);
44 }
45
46 protected void startSortButton_Click(object sender, EventArgs e)
47 {
48     bool ascending;
49     if (sortType.SelectedIndex == 0)
50     {
51         ascending = true;
52     }
53     else
54     {
55         ascending = false;
56     }
57     productsDisplayList.sort(ascending, sortField.SelectedValue);
58     populatePage();
59 }
60
61 protected void searchButton_Click(object sender, EventArgs e)
62 {
63     string searchType = searchFieldDropdown.SelectedValue;
64     string searchText = searchBox.Text;
65     bool whitelist = Convert.ToBoolean(whitelistSelect.SelectedValue);
```

```
66        productsDisplayList.filter(searchType, searchBox.Text, whitelist);
67        populatePage();
68 }
69
70 protected void coasterClockButton_Click(object sender, EventArgs e)
71 {
72        string filterMode = coastersOrClocks.SelectedValue;
73        productsDisplayList.filter("stock", filterMode, true);
74        populatePage();
75 }
76
77 protected void resetFilter_Click(object sender, EventArgs e)
78 {
79        productsDisplayList.resetWorkingList();
80        populatePage();
81 }
82
```

The Page   The page contains a panel full of various controls for filtering and sorting the contents of the page and an empty panel in the content called productsDisplayPanel. It inherits from Customer.Master.

The Code   The variables on this page are:

- **productsDisplayList** is the local productList instance. It stores all the products which were in the database when the page was first loaded and contains all the logic and code which was detailed above.

- **panels** is the array which will hold all the productPanels the page is currently operating on. This does not store all the panels currently rendered on the page, as I cannot find a way to do that in C# and ASP.NET. The array that contains those is this.Controls[] which also holds the rest of the controls on the page.

- **initialized** is a boolean value which just states if the page has already been loaded for the first time by the user. It is used to ensure that the database is not access more than it has to be by ensuring that productsDisplayList is not initialized again every time the page refreshes. This in turn ensures that productsDisplayList is consistent across page refreshes which is important to retain the sort and filter.

The methods within this page are as follows:

- **Page_Load** here just checks if initialized is true. If it is, it will just run populatePage() and if not it will instantiate a new instance of productsDisplayList, then run populatePage() and set initialized to true so this iteration does not trigger twice in a row.

- **populatePage** generates all the needed controls and adds them to the Controls[] property of productsListPanel. It starts by setting panels[] to the output of productsDisplayList.generateControls() which is detailed in the productsList documentation so panels is now just an array of productPanels. It clears out the already existing controls in productsListPanel to make way for the new ones it is generating. It runs a for loop which goes through each panel in panel because for this page only, we need a button to be generated with each panel which will take the user to the individual product view page; within the loop a button (detailsButton) is initialized. The appropriate properties of the button are set:

    - CssClass is set to "detailsButton" to ensure CSS styling is applied to it.
    - Text is set to "View" to clearly illustrate to the user what the button does.
    - ID is set to the name of the product appended with "DetailsLinkButton". This might seem a little useless, but ASP.NET will throw exceptions if similar objects on the page have the same or no ID.

– CommandArgument is set to the name of the product the panel is linked to. This determines what parameter is passed to the function that runs when the button is clicked, so that the function of each detailsButton can be unique.

It instantiates a new EventHandler, which will catch when the button is clicked and execute a method, and points it towards the method detailsButton_Click. It adds the attribute "runat:server" to ensure the control functions as expected with other controls then adds the whole button to the Controls[] list of the whole productPanel that the for loop is currently examining. Finally, it adds the whole productPanel to productsDisplayPanel.

- **detailsButton_Click** is a standard click event method. It takes the sender which triggered the event and the arguments of the event contained in the EventArgs instance "e". The method gets the button instance which sent it by filtering sender. It sets the "productRedirectName" session to the command argument of the button (which as you can see above is the name of the product that button is linked to). It then redirects the user to the productView page, which uses the session value to determine what product to show.

- **startSortButton_Click** is another standard click event. startSort is a button in the panel containing all the sorting controls which applies the selected sort. The method gets whether the user wants ascending or descending sort by checking which value of the dropdown box sortType is selected. It then runs productsDisplayList.sort and passes it the selected value of sortField (which dictates what property to sort by) and the ascending bool it obtained earlier. When the sort is done running on the list, it runs populatePage to refresh the page and show all the products in their new, sorted form.

- **searchButton_Click** is another button's click event. It pulls the field to search into searchType from searchFieldDropdown and the searchText from searchBox (note that this does not need to be sanitized as it never runs on the database). It determines whether or not the user wants a whitelisted search or a blacklisted search from the selected value of whitelistSelect. It uses the filter() method of productsDisplayList to filter the list, then regenerates all the controls on the page using populatePage.

- **resetFilter_Click** is the final method for this page. It is the button click event of the final button on the page which simply runs resetWorkingList on productsDisplayList to clear all sorts and filters, then runs populatePage to update the actual webpage.

3.2.12 Individual Product Page

This is the page the user sees when they click the button labelled "View" on the store front page. It contains a larger version of the image of the product and a description.

```
1  product currentProduct;
2  defaultDataSetTableAdapters.cartsTableAdapter cartsAdaptor = new
       defaultDataSetTableAdapters.cartsTableAdapter();
3
4  protected void Page_Load(object sender, EventArgs e)
5  {
6      if (Convert.ToString(Session["productRedirectName"]) == null || Convert.ToString(
       Session["productRedirectName"]) == "")
7      {
8          Server.Transfer("~/products.aspx", false);
9      }
10     currentProduct = new product(Convert.ToString(Session["productRedirectName"]));
11     productPanel currentPanel = new productPanel(currentProduct);
12     currentPanel.CssClass = "fullProductPanel";
13     product.Controls.Add(toAdd);
14 }
15
16 protected void cartButton_Click(object sender, EventArgs e)
17 {
18     if (Convert.ToString(Session["userType"]) != "customer")
19     {
20         returnLabel.Text = "You must log in to a customer account to have a cart!";
21     }
22     else
23     {
24         if (amountToAdd.Text != "0")
25         {
26             if (cartsAdaptor.checkExisting(Convert.ToString(Session["currentUser"]),
       currentProduct.productName) == null)
27             {
28                 cartsAdaptor.insertProduct(Convert.ToString(Session["currentUser"]),
       currentProduct.productName, Convert.ToInt32(amountToAdd.Text));
29             }
30             else
31             {
32                 cartsAdaptor.updateAmount(Convert.ToInt32(amountToAdd.Text), Convert.
       ToString(Session["currentUser"]), currentProduct.productName);
33             }
34             returnLabel.Text = amountToAdd.Text + " " + currentProduct.productName + "s
       added to your cart!";
35         }
36         else
37         {
38             returnLabel.Text = "You can't add 0 of something to your cart!";
39         }
40     }
41 }
42
```

The page is contains a panel called "product" to drop the productPanel into, a textbox for them to enter the amount of the product they want to add to their cart and a button to actually add it to their cart. The variables on the page are:

- **currentProduct** is the instance of product the page is basing itself off.

- **cartsAdaptor** is the adaptor for the carts table, which the page needs because it needs to add products to the carts table.

The methods on the page are:

- **Page_Load** contains code to initialize all the controls which need to be on the page for it to display the product effectively. It first checks if the session "productRedirectName" has actually

been assigned a value by the store front page, and if not, it redirects the user to the store front page because it can't display a without being instructed which one to show. It uses the value of that session to instantiate a new product with the name of the product passed from the session to it's constructor. It instantiates a productPanel using the product class and calls it "currentPanel". It sets the CssClass property of the panel to "fullProductPanel" so that different styling is applied to it than those on the store front page. Finally, it adds the panel to the panel on the page.

- **cartButton_Click** is the click event for the only button on the page, the "add to cart" button. The method initially checks if the user that is currently logged in is a customer, as these are the only accounts that are allowed to have a cart. It checks if the amount of the product they're trying to add is 0 and denies them if it is (no check is needed to ensure that the contents of the amountToAdd text box is actually a number because the "TextMode" property of the box is set to "number" so the page will not allow the user to type anything but numbers into it). It then uses the adaptor's query checkExisting to check if the customer has already added that product to their cart. checkExisting runs:

```
1 SELECT COUNT(*) AS Expr1, [user], productName
2 FROM carts
3 GROUP BY [user], productName
4 HAVING ([user] = ?) AND (productName = ?)
5
```

If there isn't that product in the user's cart, then the query insertProduct is used:

```
1 INSERT INTO `carts` (`user`, `productName`, `amount`)
2 VALUES (?, ?, ?)
3
```

in order to add it with the amount the user specified. If the product is already in their cart, the query updateStock is run:

```
1 UPDATE carts
2 SET amount = amount + ?
3 WHERE ([user] = ?) AND (productName = ?)
4
```

At all possible terminations of this process, the user is updated with the status of the operation using returnLabel.

Revision 29/04/18 The huge set of nested if statements in the method "cartButton_Click" is both difficult to understand and unnecessary. I have overhauled them to be in series and just end the method when a failing condition is met.

```
1  if (Convert.ToString(Session["userType"]) != "customer")
2  {
3      returnLabel.Text = "You must log in to a customer account to have a cart!";
4      return;
5  }
6  if (amountToAdd.Text == "0")
7  {
8      returnLabel.Text = "You can't add 0 of something to your cart!";
9      return;
10 }
11
12 if (cartsAdaptor.checkExisting(Convert.ToString(Session["currentUser"]), currentProduct.
       productName) == null)
13 {
14     cartsAdaptor.insertProduct(Convert.ToString(Session["currentUser"]), currentProduct.
       productName, Convert.ToInt32(amountToAdd.Text));
15 }
16 else
17 {
18     cartsAdaptor.updateAmount(Convert.ToInt32(amountToAdd.Text), Convert.ToString(
       Session["currentUser"]), currentProduct.productName);
```

```
19  }
20  returnLabel.Text = amountToAdd.Text + " " + currentProduct.productName + "s added to
        your cart!";
21
```

3.2.13 Customer Registration

This page is univesally accessible through a link on the customer log in page, as I mentioned in the documentation for that.

```
1  private defaultDataSetTableAdapters.customersTableAdapter customersQueryTable = new
        defaultDataSetTableAdapters.customersTableAdapter();
2  protected void Page_Load(object sender, EventArgs e)
3  {
4
5  }
6
7  protected void registerButton_Click(object sender, EventArgs e)
8  {
9      if (passwordBox.Text == confirmPasswordBox.Text)
10     {
11         int dump;
12         if (int.TryParse(phoneNumberBox.Text, out dump))
13         {
14             if (customSecurity.sanitizeCheck(new string[] { usernameBox.Text,
    passwordBox.Text, address1Box.Text, address2Box.Text, cityBox.Text, countryDropdown.
    SelectedValue, postcodeBox.Text, phoneNumberBox.Text, forenameBox.Text, surnameBox.
    Text }))
15             {
16                 try
17                 {
18                     customersQueryTable.newCustomer(usernameBox.Text, customSecurity.
    generateMD5(passwordBox.Text), address1Box.Text, address2Box.Text, cityBox.Text,
    countryDropdown.SelectedValue, postcodeBox.Text, phoneNumberBox.Text, forenameBox.
    Text, surnameBox.Text);
19                     customLogging.newEntry("Someone registered the user " + usernameBox.
    Text);
20                     returnLabel.Text = "User " + usernameBox.Text + " created";
21                 }
22                 catch (Exception except)
23                 {
24                     returnLabel.Text = "Registration failed, reason: " + except.Message;
25                 }
26             }
27             else
28             {
29                 returnLabel.Text = customSecurity.sanitizeErrorMessage;
30             }
31         }
32         else
33         {
34             phoneNumReturn.Text = "The phone number must be digits only!";
35         }
36     }
37     else
38     {
39         confirmPasswordBox.Text = "";
40         passwordBoxReturn.Text = "The passwords don't match!";
41     }
42 }
43
44 protected void confirmPasswordBox_TextChanged(object sender, EventArgs e)
45 {
46     passwordsMatchCheck();
47 }
48
49 protected void passwordBox_TextChanged(object sender, EventArgs e)
50 {
51     passwordsMatchCheck();
52 }
53
54 private void passwordsMatchCheck()
55 {
```

```
56    if (passwordBox.Text != confirmPasswordBox.Text)
57    {
58        passwordBoxReturn.Text = "The passwords don't match!";
59    }
60 }
61
```

The Page On the page there are text boxes for:

- Username

- Password

- Confimation of Password (to reduce the chances the user types the wrong password in since both boxes are set to not show their contents)

- First Name

- Surname

- Address Line 1

- Address Line 2

- Town/City

- Post Code

- Phone Number

Additionally, there is a dropdown box for the user to pick a country from, two labels to show messages to the user and a button to submit the details ggiven.

The Code The only variable on the page is **customersQueryTable**, which serves the same purpose as adaptor on other pages. It is the DataSet adaptor for the customers table in the database and manage all queries to be run on that table.
The methods on the page are:

- **Page_Load** serves no purpose in this page because nothing needs to happen on the page when it loads.

- **registerButton_Click** performs a series of checks on the contents of the boxes before inserting the information into the database using a set of if statements. The checks it performs are:

  - Whether the password box and confirm password boxes' contents match. If they don't, the process is cancelled and the relevant return label shows an appropriate message.
  - If the phone number actually parses as an integer. This is necessary because I can't set the text mode of the phone number text box to "number", because it will then remove 0s from the beginning and other related things.
  - A sanitize check with customSecurity.sanitizeCheck() to ensure all boxes are free of SQL sensitive characters. All these boxes' contents will be inserted into the database, so this is necessary to prevent SQL injection.

  Once all these checks are complete, a try-catch statement is used to run the necessary query on the database. I enclosed the database query like this because when I wrote this I wasn't sure whether or not to expect any errors but I will likely revise it. The query run is:

```
1 INSERT INTO customers (username, [password], addressLine1, adressLine2, city,
     country, postCode, phoneNumber, forename, surname)
2 VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
3
```

  This simply adds values to all the new required fields. The creation is logged, and the return label is appropriately updated with a message.

Revision 05/05/18 I decided the horrendous set of nested if statements could easily be reorganised and that the try-catch statement could be removed since the code isn't likely to throw exceptions. **registerButton_Click** now contains:

```
1  if (passwordBox.Text != confirmPasswordBox.Text)
2  {
3      confirmPasswordBox.Text = "";
4      passwordBoxReturn.Text = "The passwords don't match!";
5      return;
6  }
7
8  int dump;
9  if (!int.TryParse(phoneNumberBox.Text, out dump))
10 {
11     phoneNumReturn.Text = "The phone number must be digits only!";
12     return;
13 }
14
15 if (!customSecurity.sanitizeCheck(new string[] { usernameBox.Text, passwordBox.Text,
       address1Box.Text, address2Box.Text, cityBox.Text, countryDropdown.SelectedValue,
       postcodeBox.Text, phoneNumberBox.Text, forenameBox.Text, surnameBox.Text }))
16 {
17     returnLabel.Text = customSecurity.sanitizeErrorMessage;
18     return;
19 }
20
21 customersQueryTable.newCustomer(usernameBox.Text, customSecurity.generateMD5(passwordBox
       .Text), address1Box.Text, address2Box.Text, cityBox.Text, countryDropdown.
       SelectedValue, postcodeBox.Text, phoneNumberBox.Text, forenameBox.Text, surnameBox.
       Text);
22 customLogging.newEntry("Someone registered the user " + usernameBox.Text);
23 returnLabel.Text = "User " + usernameBox.Text + " created";
24
```

3.2.14 Cart Page

This page shows the user what is in their cart, and gives them the option to edit it.

```
 1  struct cartItem
 2  {
 3      public product product;
 4      public int amount;
 5  }
 6
 7  defaultDataSetTableAdapters.cartsTableAdapter cartsTableAdapter = new
        defaultDataSetTableAdapters.cartsTableAdapter();
 8  cartItem[] cartArray;
 9
10  protected void Page_Load(object sender, EventArgs e)
11  {
12      populateList();
13      if (cartArray.Length == 0)
14      {
15          productsListPanel.Controls.Add
16              (
17                  new Label()
18                  {
19                      Text = "Your cart is empty!",
20                      CssClass = "cartEmptyMessage"
21                  }
22              );
23          purchaseButton.Visible = false;
24      }
25      else
26      {
27          populatePage();
28          purchaseButton.Visible = true;
29      }
30  }
31
32  private void populateList()
33  {
34      using (DataTable temp = cartsTableAdapter.getCart(Convert.ToString(Session["
        currentUser"])))
35      {
36          cartArray = new cartItem[temp.Rows.Count];
37          int i = 0;
38          foreach (DataRow row in temp.Rows)
39          {
40              cartArray[i].product = new product(Convert.ToString(row[2]));
41              cartArray[i].amount = Convert.ToInt32(row[3]);
42              i++;
43          }
44      }
45  }
46
47  private void populatePage()
48  {
49      foreach(cartItem current in cartArray)
50      {
51          productPanel panel = new productPanel(current.product);
52          panel.Controls.Remove(panel.Controls[3]);
53          panel.Controls.Remove(panel.Controls[3]);
54
55          TextBox amountBox = new TextBox()
56          {
57              CssClass = "amountBox",
58              ID = current.product.productName + "_CartAmountBox",
59              Text = Convert.ToString(current.amount),
60              TextMode = TextBoxMode.Number,
61              AutoPostBack = true
62          };
63          amountBox.TextChanged += new EventHandler(amountBox_TextChanged);
```

```
64          panel.Controls.Add(amountBox);
65
66          Button removeButton = new Button()
67          {
68              CssClass = "removeButton",
69              ID = current.product.productName + "_CartRemoveButton",
70              Text = "Remove",
71              CommandArgument = current.product.productName
72          };
73          removeButton.Click += new EventHandler(removeButton_Click);
74          removeButton.Attributes.Add("runat", "server");
75          panel.Controls.Add(removeButton);
76
77          panel.CssClass = "cartProductDisplayPanel";
78          productsListPanel.Controls.Add(panel);
79      }
80 }
81
82 protected void removeButton_Click(object sender, EventArgs e)
83 {
84      Button btn = (Button)sender;
85      int i = 0;
86      int index = 0;
87      foreach (cartItem current in cartArray)
88      {
89          if (current.product.productName == productName)
90          {
91              index = i;
92          }
93          i++;
94      }
95      var temp = new List<cartItem>(cartArray);
96      temp.RemoveAt(index);
97      cartArray = temp.ToArray();
98      populatePage();
99 }
100
101 protected void amountBox_TextChanged(object sender, EventArgs e)
102 {
103      TextBox box = (TextBox)sender;
104      string productName = box.ID.Split('_')[0];
105      int i = 0;
106      int index = 0;
107      foreach(cartItem current in cartArray)
108      {
109          if (current.product.productName == productName)
110          {
111              index = i;
112              break;
113          }
114          i++;
115      }
116      cartArray[index].amount = Convert.ToInt32(box.Text);
117 }
118
119 protected void purchaseButton_Click(object sender, EventArgs e)
120 {
121      makeOrder();
122 }
123
124 protected void makeOrder()
125 {
126      var productsAdaptor = new defaultDataSetTableAdapters.productsTableAdapter();
127      var ordersAdaptor = new defaultDataSetTableAdapters.ordersTableAdapter();
128      if (Convert.ToString(Session["userType"]) != "customer")
129      {
130          returnLabel.Text = "You're not a registered customer so you can't make an order!
     ";
```

```
131          return ;
132      }
133      foreach ( cartItem current in cartArray )
134      {
135          int currentStock = Convert . ToInt32 ( productsAdaptor . getStock ( current . product .
        productName ) ) ;
136          if ( currentStock < current . amount )
137          {
138              returnLabel . Text = ”Sorry , we only have ” + currentStock + ” of ” + current .
        product . productName + ” in stock !”;
139              return ;
140          }
141      }
142      commonClasses . customLogging . newEntry (”Customer ” + Convert . ToString ( Session [”
        currentUser”]) + ” checked out”) ;
143      foreach ( cartItem current in cartArray )
144      {
145          ordersAdaptor . newOrder ( DateTime . Now , current . product . price ∗ current . amount ,
        current . amount , Convert . ToString ( Session [”currentUser”]) , current . product .
        productName ) ;
146          productsAdaptor . updateStock ( Convert . ToInt32 ( productsAdaptor . getStock ( current .
        product . productName )) − current . amount , current . product . productName ) ;
147      }
148      cartsTableAdapter . deleteCart ( Convert . ToString ( Session [”CurrentUser”]) ) ;
149      populateList () ;
150      populatePage () ;
151      returnLabel . Text = ”Thanks for your order !”;
152 }
153
```

The Page   The page contains a panel to place all the programmatically generated panels into, a button for the user to purchase the contents of their cart and a label for returning messages to the user.

The Code   The variables and properties in this page's C# code are:

- **cartItem** is a customer structure which I have declared to help me store items in the cart for the purposes of the page. Since productList doesn't contain functionality for multiple amounts of an item within the list and it's general design is that of showing the products in stock, not a seperate collection, I made this singularly for the purposes of this page. It stores an instance of "product" and a integer representing the amount of that product. It is intended to be used to create an array.

- **cartsTableAdapter** is just an instance of the adaptor to hold queries which will need to be executed on the "carts" table in the database.

- **cartArray** is the array of type cartItem which will serve as the local list of all the items in the user's cart.

The methods are:

- **Page_Load** first runs populateList() to pull the relevant contents of the carts table into cartArray. It then checks if the cartArray object is empty (ie. it's length is 0, meaning the user has no items in their cart). If it is, a label is generated then added to the page to inform the user that there are no products in their cart. If it isn't, populatePage() is run to generate all the elements which the page needs and the purchase button (which is invisible by default) is set to be visible.

- **populateList** uses the query getCart in cartsTableAdapter to get a complete list of all the records in the table. The query is a simple get all:

```
1 SELECT ID , [ user ] , productName , amount FROM carts
2
```

74

Once the method has the table, it initializes cartArray using the amount of rows in the table as the amount of elements to initialize, and an integer to serve as the index of the foreach loop it is about to run. A for each loop is run which goes through each row in the DataTable and sets the next element in cartArray to match the correct columns for the name and the amount, using the name to initialize a new product.

- **populatePage** is designed to create all the elements to be displayed on the page. All the logic in the method is enclosed in a for each loop which goes through all the elements in cartArray and generates an augmented productPanel for them. The code begins by intializing a new instance of productPanel which generates most of the needed elements by itself. Immediately after initialization, the code removes the stockTag and descriptionTag because they are both not needed in the design of this page. This is accomplished by removing the control at position 3 in the controls array of the productPanel instance twice - since the controls are adjacent, the descriptionTag falls into the place that the stockTag just occupied. Then, a new TextBox is created to let the user change the amount of the product they have in their cart and the relevant properties set:

  - CssClass is set to amountBox so that styling can be applied
  - ID is set to the product's name plus "_CartAmountBox" as this is used later
  - Text is set to the amount registered of the product because this textbox should start with displaying the amount of the product which is in the cart
  - TextMode is set to be "number" because this will prevent users entering anything but digits, which is what we want for this box
  - AutoPostback is set to true; postback forces a page refresh and allows the textbox to trigger event handlers when the contents is changed and the return key is pressed

  The new TextBox has a new EventHandler assigned to it which will trigger **amountBox_TextChanged** when the text is changed. The control is added to the controls of the current productPanel.
  A new button is instantiated to let the user remove something from their cart with the following properties:

  - CssClass is set to "amountBox" to allow styling
  - ID is set to the name of the current product plus "_CartRemoveButton" so that the control can be referred to later on
  - Text is set to remove to indicate the button's purpose to the user
  - The CommandArgument of the button is set to the name of the product so that the button functions per product

  An EventHandler is added to the button's click hook and assigned the method "removeButton_Click". The attribute "runat:server" is added to the button so that it functions correctly within ASP. The button is added to the panel's control list. Finally, the panel itself has it's Css-Class changed to "cartProductDisplayPanel" so that individual styling can be applied to it with no effect from the styling applied to productPanel by default and it is added to the panel on the page which is set aside for it.

- **removeButton_Click** runs when the remove button is clicked. It gets the button which ran the method by filtering the sender by type "Button" and gets the name of the product whose remove button was clicked from the ID of the button (Ideally this would be taken from the command argument but at this time that doesn't seem to work) by splitting it using String.Split() at the "_" character and taking the first part. It uses a foreach loop with an externally managed index "i" to set the integer "index" to be the index of that panel within the array holding all the augmented productPanels by matching the names of each to the string it obtained from the button's ID. It converts the cartArray to a list temporarily so that removing items is much easier, and removes the element at the previously found index. The list is converted back to an array and set to cartArray, and populatePage() is run to show the new information on the page.

- **amountBox_TextChanged** runs when the value inside amountBox is changed and the return key is pressed. As before, the method obtains the sender and the name of the product in question from the attributes of the sender by splitting the ID. Again, a for each loop is run with a manual "i" to find the index of the product in cartArray. Once these pieces of information are found, the .amount property of the correct element in cartArray is changed to the user's new desired amount.

- **purchaseButton_Click** effectively just runs makeOrder. This is in it's own method because this is a dummy - if the project were to actually be deployed, then much different code would go here.

- **makeOrder** begins by opening two new adaptors because access to the products table and the orders table are needed for the customer to actually purchase their items. The method checks that the logged in user is actually a customer by checking that the "userType" session is "customer" and returns an appropriate message explaining that they cannot make an order if they aren't. Then, a foreach loop is run to check that there is actually enough of each product in the cart in stock to satisfy the customer's order - an integer is obtained from the output of the getStock query:

```
1  SELECT stock
2  FROM products
3  WHERE (productName = ?)
4
```

If the amount of the product the user is trying to buy is more than the amount of the product available, the operation is cancelled and the user is notified with the return label. Once the above checks are complete, the purchase can be "made". First, the purchase is logged - the names of the products bought/their amounts are not needed in the log since they are stored in the orders table. Then, for each item in the cart, two queries are run - the newOrder query on the orders table:

```
1  INSERT INTO `orders` (`datePlaced`, `spent`, `productAmount`, `customer`, `product
      `)
2  VALUES (?, ?, ?, ?, ?)
3
```

Which creates a new order with the correct information which is calculated on the spot. The other query is the updateStock query for the products table so that the correct amount of stock is removed to represent the customer's purchase:

```
1  UPDATE products
2  SET stock = ?
3  WHERE (productName = ?)
4
```

Once these queries have finished for each product, the deleteCart query is run once on the cart table:

```
1  DELETE FROM carts
2  WHERE ([user] = ?)
3
```

This just deletes all the cart entries for the user, since they have just purchased them all. Finally, populateList is run to refresh the local list (to be blank), populatePage is run to clear the page of all the elements that were on it and the return label is set to display a thank you message.

Revision 07/05/18   Upon re reading this code, I realize now that the amount box doesn't update the amount of the product in the carts table so if the page is refreshed, the new value will be lost. I created the following query and called it updateAmount:

```
1  UPDATE carts
2  SET amount = amount + ?
3  WHERE ([user] = ?) AND (productName = ?)
4
```

And added it to the **amountBox_TextChanged** method, at the end:

```csharp
TextBox box = (TextBox)sender;
string productName = box.ID.Split('_')[0];
int newAmount = Convert.ToInt32(box.Text);
int i = 0;
int index = 0;
foreach(cartItem current in cartArray)
{
    if (current.product.productName == productName)
    {
        index = i;
        break;
    }
    i++;
}
cartArray[index].amount = newAmount;
cartsTableAdapter.updateAmount(newAmount, Convert.ToString(Session["currentUser"]),
    productName);
```

3.2.15 Employee Registration

The employee registration page is different to the customer registration page because the only people authorized to register a new administrator are administrators. Employees may register other, normal employees but not administrators. It inherits from the Employee.Master file, so it is only accessible to employees by default. This, however, is not enough and so an additional check needs to be put in place for this page only, in Page_Load.

```csharp
protected void Page_Load(object sender, EventArgs e)
{
    if (Convert.ToString(Session["userIsAdmin"]) == "False")
    {
        adminCheckBox.Visible = false;
    }
    else
    {
        adminCheckBox.Visible = true;
    }
}

protected void newUser_Click(object sender, EventArgs e)
{
    if (submittedPasswordBox.Text == submittedConfirmPasswordBox.Text)
    {
        if (customSecurity.sanitizeCheck(new string[] { submittedUsernameBox.Text,
    submittedPasswordBox.Text, forenameBox.Text, surnameBox.Text }))
        {
            if (submittedUsernameBox.Text != "master" || submittedUsernameBox.Text != "
    Market")
            {
                using (defaultDataSetTableAdapters.employeesTableAdapter
    employeeQueryTable = new defaultDataSetTableAdapters.employeesTableAdapter())
                {
                    try
                    {
                        employeeQueryTable.newEmployee(submittedUsernameBox.Text,
    customSecurity.generateMD5(submittedPasswordBox.Text), adminCheckBox.Checked,
    forenameBox.Text, surnameBox.Text);
                        registerReturn.Text = "New employee created";
                        customLogging.newEntry("Employee " + submittedUsernameBox.Text +
     " was created");
                    }
                    catch (Exception except)
                    {
                        registerReturn.Text = "Database operation failed with error " +
    except.Message;
                    }
                }
            }
            else
            {
                registerReturn.Text = "That is a reserved username!";
            }
        }
        else
        {
            registerReturn.Text = customSecurity.sanitizeErrorMessage;
        }
    }
    else
    {
        registerReturn.Text = "The passwords don't match!";
        submittedConfirmPasswordBox.Text = "";
    }
}
```

The Page The page contains text boxes for:

- The desired Username

- The desired password and another box to confirm it

- The employee's forename

- The employee's surname

Additionally, there is a checkbox to specify whether or not the new user should be an administrator, a button to submit the form and a label to show the user the result of the operation.

The Code There are no additional properties on the page. The methods are:

- **Page_Load** contains a similar check to the employee check present on Employee.Master, but it checks the "userIsAdmin" session instead of the "userType" session. If the user is an admin, the administrator checkbox is made visible, and invisible if they are not.

- **newUser_Click** triggers when the new user button is clicked. It contains a series of checks not dissimilar to those that the customer registration page performs. The first checks that the two boxes contain the same amount and notifies the user if they don't. The second uses customSecurity.sanitizeCheck to ensure there are no SQL-sensitive characters in the textboxes, and returns the standard error message if they are. The third checks specifically checks if the username of the account that is being created is trying to be set to "master" or "Market". Both of these are reserved for function within the program as master is the default administrator account which cannot be interfered with, and "Market" is the username I am using to reserve items for the market, detailed below. Finally, a new adaptor is opened to the employees table and the new account is created with newEmployee:

```
1 INSERT INTO employees (username, [password], admin, forename, surname)
2 VALUES (?, ?, ?, ?, ?)
3
```

  The return label shows an appropriate message and the creation of a new employee is logged. The last-described logic is enclosed in a try-catch statement should any database errors occur.

Revision 01/05/18 As before, the series of nested if statements in **newUser_Click** is an absurd solution here, and makes the code much more difficult to read. I have amended as below:

```
1 if (submittedPasswordBox.Text != submittedConfirmPasswordBox.Text)
2 {
3     registerReturn.Text = "The passwords don't match!";
4     submittedConfirmPasswordBox.Text = "";
5 }
6
7 if (!customSecurity.sanitizeCheck(new string[] { submittedUsernameBox.Text,
      submittedPasswordBox.Text, forenameBox.Text, surnameBox.Text }))
8 {
9     registerReturn.Text = customSecurity.sanitizeErrorMessage;
10 }
11
12 if (submittedUsernameBox.Text == "master" || submittedUsernameBox.Text == "Market")
13 {
14     registerReturn.Text = "That is a reserved username!";
15 }
16
17 using (defaultDataSetTableAdapters.employeesTableAdapter employeeQueryTable = new
      defaultDataSetTableAdapters.employeesTableAdapter())
18 {
19     try
20     {
```

```
21          employeeQueryTable.newEmployee(submittedUsernameBox.Text, customSecurity.
     generateMD5(submittedPasswordBox.Text), adminCheckBox.Checked, forenameBox.Text,
     surnameBox.Text);
22          registerReturn.Text = "New employee created";
23          customLogging.newEntry("Employee " + submittedUsernameBox.Text + " was created")
     ;
24      }
25      catch (Exception except)
26      {
27          registerReturn.Text = "Database operation failed with error " + except.Message;
28      }
29 }
30
```

3.2.16 User Management Page

This page allows employees to see and change information and credentials of users (both employees and customers), and administrators to delete them. It inherits from Employee.Master.

```
1  defaultDataSetTableAdapters.employeesTableAdapter employeeQueryTable = new
       defaultDataSetTableAdapters.employeesTableAdapter();
2  defaultDataSetTableAdapters.customersTableAdapter customerQueryTable = new
       defaultDataSetTableAdapters.customersTableAdapter();
3
4  protected void Page_Load(object sender, EventArgs e)
5  {
6      if (Convert.ToString(Session["userIsAdmin"]) == "False")
7      {
8          configControls.Visible = false;
9          employeesDisplayTable.Columns[4].Visible = false;
10         customersDisplayTable.Columns[3].Visible = false;
11     }
12     else
13     {
14         configControls.Visible = true;
15         employeesDisplayTable.Columns[4].Visible = true;
16         customersDisplayTable.Columns[3].Visible = true;
17     }
18 }
19
20 private string username;
21
22 protected void employeesDisplayTable_RowCommand(object sender, GridViewCommandEventArgs
       e)
23 {
24     username = employeesDisplayTable.Rows[Convert.ToInt32(e.CommandArgument)].Cells[0].
       Text;
25     switch (e.CommandName)
26     {
27         case "deleteUser":
28             if (username == "master")
29             {
30                 returnLabel.Text = "You can't delete the master admin account!";
31                 deletingUsersPersistent.deleting = false;
32                 deletingUsersPersistent.type = "";
33                 break;
34             }
35             deleteUser(username, "Employee");
36             break;
37         case "changeUserPassword":
38             changeUserPassword(username, "Employee");
39             break;
40     }
41 }
42
43 protected void customersDisplayTable_RowCommand(object sender, GridViewCommandEventArgs
       e)
44 {
45     username = customersDisplayTable.Rows[Convert.ToInt32(e.CommandArgument)].Cells[0].
       Text;
46     switch (e.CommandName)
47     {
48         case "deleteUser":
49             deleteUser(username, "Customer");
50             break;
51         case "changeUserPassword":
52             changeUserPassword(username, "Customer");
53             break;
54     }
55 }
56
57 static class deletingUsersPersistent
```

```
58 {
59      public static string username;
60      public static bool deleting;
61      public static string type;
62 }
63
64 protected void deleteUser(string username, string type)
65 {
66      GridView displayTable;
67      if (type == "Employee")
68      {
69          displayTable = employeesDisplayTable;
70      }
71      else if (type == "Customer")
72      {
73          displayTable = customersDisplayTable;
74      }
75      else
76      {
77          throw new Exception();
78      }
79
80      if ((deletingUsersPersistent.deleting) && (username == deletingUsersPersistent.
         username) && deletingUsersPersistent.type == type)
81      {
82          returnLabel.Text = type + " account " + deletingUsersPersistent.username + " was
          deleted";
83          deletingUsersPersistent.deleting = false;
84          deletingUsersPersistent.type = "";
85          if (type == "Employee")
86          {
87              employeeQueryTable.deleteUser(username);
88          }
89          else
90          {
91              customerQueryTable.deleteUser(username);
92          }
93          customLogging.newEntry(type + " " + username + " was deleted");
94          displayTable.DataBind();
95          System.Threading.Thread.Sleep(750);
96          returnLabel.Text = "";
97      }
98      else
99      {
100         returnLabel.Text = "Click again to delete - note that this cannot be undone!";
101         deletingUsersPersistent.deleting = true;
102         deletingUsersPersistent.username = username;
103         deletingUsersPersistent.type = type;
104     }
105 }
106
107 protected void changeUserPassword(string username, string type)
108 {
109     GridView displayTable;
110     if (type == "Employee")
111     {
112         displayTable = employeesDisplayTable;
113     }
114     else if (type == "Customer")
115     {
116         displayTable = employeesDisplayTable;
117     }
118     else
119     {
120         throw new Exception();
121     }
122
123     returnLabel.Text = "";
```

```
124        deletingUsersPersistent.deleting = false;
125        if (customSecurity.sanitizeCheck(new string[] { passwordBox.Text }))
126        {
127            if (passwordBox.Text != "" || confirmPassword.Text != "")
128            {
129                if (passwordBox.Text == confirmPassword.Text)
130                {
131                    if (type == "Employee")
132                    {
133                        employeeQueryTable.changePassword(customSecurity.generateMD5(
                           passwordBox.Text), username);
134                    }
135                    else
136                    {
137                        customerQueryTable.changePassword(customSecurity.generateMD5(
                           passwordBox.Text), username);
138                    }
139                    customLogging.newEntry(type + " " + username + "'s password changed");
140                    returnLabel.Text = type + " " + username + "'s password was changed to '
                       " + passwordBox.Text + "'.";
141                }
142                else
143                {
144                    returnLabel.Text = "The passwords do not match!";
145                    System.Threading.Thread.Sleep(750);
146                    returnLabel.Text = "";
147                }
148            }
149            else
150            {
151                returnLabel.Text = "You must fill both boxes!";
152                System.Threading.Thread.Sleep(750);
153                returnLabel.Text = "";
154            }
155        }
156        else
157        {
158            returnLabel.Text = customSecurity.sanitizeErrorMessage;
159        }
160 }
161
162 protected void registerRedirect_Click(object sender, EventArgs e)
163 {
164     Server.Transfer("~/registerEmployee.aspx", false);
165 }
166
167 protected void newCustomerButton_Click(object sender, EventArgs e)
168 {
169     Server.Transfer("~/registerCustomer.aspx", false);
170 }
171
```

The Page   On this page, there are two ASP.NET GridView elements. These are rendered tables which
show the contents of a databound source on the page. Many different types of databound sources exist
but the one I am using for both these GridViews is my database.

   The first GridView is linked to the employees table, and shows usernames, full names and whether
they are an administrator or not. Additionally, it has a column that contains two buttons in each row;
one is the delete button for that account and the other will change the password. These buttons have the
username of the employee row they are on bound to them as a command argument, and are linked with
an eventhandler automatically to the same method. The CommandName attribute of each is how they
much be differentiated between, they are set to "deleteUser" and "changeUserPassword" respectively.
Following this GridView is a button which simply redirects to the employee registration page.

The other GridView is linked to the customers table. It shows the customers' usernames and full names in addition an identical set of buttons to those described above, which are both bound to a different method. Underneath the GridView is a button which redirects to the customer registration page.

Below these controls are two text boxes where the administrator can enter the password they wish to change an account's to; the extra one is for confirmation. Finally, there is a label to return messages to the user. Ultimately, the page looks like this:

The Code The properties and variables on this page are:

- **employeeQueryTable** is an adaptor for the queries registered to the employee table.

- **customerQueryTable** is an adaptor for the queries registered to the customer table.

- **username** is a string to hold the username of the username in question between methods.

- **deletingUsersPersistent** is a static class which holds the information about users between executions of the method which carries it out. Since I wanted the additional security against user mistakes of making them click the button to delete twice, the method must be executed twice and be aware when it is being executed a second time in a row. Hence, I created a static class which cannot be instantiated, only referred to, to hold information that would inform that method. It holds the username and the type of the user in question and a bool that determines whether they are on their "second click".

The methods on the page are:

- **Page_Load** in this instance determines whether or not to show the controls for changing and deleting accounts. If the user is marked as an admin by the "userIsAdmin" session, then the columns containing the buttons and the elements at the bottom of the page (which are in a panel called "configControls") are set to be visible. If not, they are set to be invisible.

- **employeesDisplayTable_RowCommand** is called when a RowCommand is triggered on the GridView showing employees. This only occurs when one of the buttons within the table is clicked. First, the method gets the username of the user the command is called upon; it does this by taking the CommandArgument of the button and using it as the index of a row in the GridView, then finding the value stored in the first cell (the username cell). It then uses a switch to determine which button was clicked according to it's CommandName. If the user is trying to delete an employee, the methods checks with an if statement if they are trying to delete the master account by examining the username variable previously set - if they are, they are informed with the return label that they cannot and the deletingUsersPersistent class is reset to it's default state. If not, deleteUser() is run. Should the user be trying to change a password, the switch runs changeUserPassword with the correct parameters.

- **customersDisplayTable_RowCommand** is called when a RowCommand is triggered in the GridView corresponding to the customers table. It serves a similar purpose to "employeesDisplayTable_RowCommand", pulling the username of the customer in question into "username" in exactly the same way as it does in the employees GridView. A similar switch is run, but there is no check for sensitive usernames.

- **deleteUser** is called by both RowCommand methods when the user is trying to delete an account. It determines which type of user to delete with the parameters it is passed, which should only be "Employee" or "Customer" - if they aren't, a new exception is thrown.
  An if statement determines if the user is clicking the delete button for the first time or the second. This is checked by examining the values of the deleting, username and type properties of deletingUsersPersistent - if they match the current conditions, code is run to delete the account. If not, the return label shows a warning that the user is about to delete an account if they click a second time. The contents of deletingUsersPersistent are set to the current conditions. Should the user click a second time and start the account deletion process, the return label is updated with an appropriate message. The deletingUsersPersistent class is set to it's default state, and the relevant query is run. If an employee is being deleted, deleteUser is run from the correct adaptor:

```
1  DELETE FROM 'employees' WHERE (('username' = ?))
2
```

  If a customer is being deleted, deleteUser is run from the customer adaptor:

```
1  DELETE FROM 'customers' WHERE (('username' = ?))
2
```

  Both of these queries are run using the username in deletingUsersPersistent. The deletion is logged, the table is refreshed using DataBind(), a pause is initiated and the message is cleared.

- **changeUserPassword** runs when either RowCommand calls it. Similarly to deleteUser, it finds the correct GridView with the parameter it is passed. The contents of the return label is cleared, and deletingUsersPersistent.deleting is set to false so one cannot click to delete someone, click change password then click to delete that person again and actually delete them without warning. Then, a series of checks are performed:

  1. customSecurity.sanitizeCheck is called on both password boxes and the standard error displayed if the check is failed.

  2. Both password boxes are confirmed to actually contain text

  3. The password boxes' contents are confirmed to match

  The appropriate query is run. If an employee is the subject of the change, changePassword is run from the employees query table:

```
1  UPDATE employees
2  SET [password] = ?
3  WHERE (username = ?)
4
```

If a customer is the subject of the password change, changePassword is run from the customers tables adaptor:

```
1  UPDATE customers
2  SET [password] = ?
3  WHERE (username = ?)
4
```

The change is logged and the return label is updated with the appropriate text.

- **registerRedirect_Click** and **newCustomerButton_Click** redirect to the employee and customer registration pages respectively.

Revision 05/05/18 This code is extremely messy. I have made multiple changes:

1. I have made the "type" variable passed into deleteUser and changeUserPassword an enumerator, which means the two options are enforced and a switch can be used in each:

```
1  private enum userType { employee, customer }
2
```

And changed the contents of both mentioned methods appropriately, at the beginning where a reference to the appropriate GridView is set:

```
1  GridView displayTable;
2  switch (type)
3  {
4      case userType.customer:
5          displayTable = customersDisplayTable;
6          break;
7      case userType.employee:
8          displayTable = employeesDisplayTable;
9          break;
10     default:
11         throw new Exception();
12 }
13
```

Ideally, the default case would not be there but later code throws a build error citing that displayTable isn't assigned to (even though this is impossible) if it isn't.

2. Removed unnecessary sleep times. Initially I inserted these so I could clear error messages after a certain time but this is unnecessary and not in keeping with the rest of the application.

3. I have, as in other places, reorganised the horrendously eye watering set of nested if statements inside changeUserPassword into a sequence of if statements which will disqualify the operation if triggered:

```
1  GridView displayTable;
2  switch (type)
3  {
4      case userType.customer:
5          displayTable = customersDisplayTable;
6          break;
7      case userType.employee:
8          displayTable = employeesDisplayTable;
9          break;
10     default:
11         throw new Exception();
12 }
13
14 returnLabel.Text = "";
15 deletingUsersPersistent.deleting = false;
16
17 if (passwordBox.Text == "" || confirmPassword.Text == "")
18 {
```

```
19      returnLabel.Text = "You must fill both boxes!";
20      return;
21 }
22
23 if (passwordBox.Text != confirmPassword.Text)
24 {
25      returnLabel.Text = "The passwords do not match!";
26      return;
27 }
28
29 if (type == userType.employee)
30 {
31      employeeQueryTable.changePassword(customSecurity.generateMD5(passwordBox.Text),
          username);
32 }
33 else
34 {
35      customerQueryTable.changePassword(customSecurity.generateMD5(passwordBox.Text),
          username);
36 }
37 customLogging.newEntry(type + " " + username + "'s password changed");
38 returnLabel.Text = type + " " + username + "'s password was changed to '" +
        passwordBox.Text + "'.";
39
```

4. I have added "Market" to the reserved usernames check in deleteUser and updated the error message:

```
1 if (username == "master" || username == "Market")
2 {
3      returnLabel.Text = "You can't delete this account!";
4      deletingUsersPersistent.deleting = false;
5      deletingUsersPersistent.type = userType.customer;
6      break;
7 }
8
```

3.2.17 Product Management/Configuration

The product management page allows employees to create new products and edit the stock and prices of existing ones.

```csharp
private defaultDataSetTableAdapters.productsTableAdapter productQueryTable = new
    defaultDataSetTableAdapters.productsTableAdapter();

protected void Page_Load(object sender, EventArgs e)
{

}

protected void productsTable_RowCommand(object sender, GridViewCommandEventArgs e)
{
    if(e.CommandName != "deleteProduct")
    {
        return;
    }
    string productName = productsTable.Rows[Convert.ToInt32(e.CommandArgument)].Cells
    [0].Text;
    if ((deletingProductsPersistent.deleting) && (productName ==
    deletingProductsPersistent.product))
    {
        returnLabel.Text = "Product " + productName + " was deleted";
        System.IO.File.Delete(Server.MapPath("~/images/") + productName);
        using (var cartsAdaptor = new defaultDataSetTableAdapters.cartsTableAdapter())
        {
            cartsAdaptor.deleteProducts(productName);
        }
        using (var marketAdaptor = new defaultDataSetTableAdapters.marketTableAdapter())
        {
            marketAdaptor.deleteProducts(productName);
        }
        using (var ordersAdaptor = new defaultDataSetTableAdapters.ordersTableAdapter())
        {
            ordersAdaptor.removeProducts(productName);
        }
        productQueryTable.deleteProduct(productName);
        returnLabel.Text = "Product deleted";
        customLogging.newEntry("The product " + productName + " was deleted");
        productsTable.DataBind();
        System.Threading.Thread.Sleep(2000);
        returnLabel.Text = "";
    }
    else
    {
        returnLabel.Text = "Click again to delete - note that this cannot be undone!";
        deletingProductsPersistent.deleting = true;
        deletingProductsPersistent.product = productName;
    }
}

protected void productAddButton_Click(object sender, EventArgs e)
{
    do
    {
        string errorAppend = "";

        if (customSecurity.sanitizeCheck(new string[] { productNameBox.Text,
    productPrice.Text, bandBox.Text, descriptionBox.Text }) != true)
        {
            returnMessage.Text = customSecurity.sanitizeErrorMessage;
            break;
        }

        string displayName = productNameBox.Text;
        string productName = displayName;
```

```csharp
60          string imagePath;
61          TextInfo cultInfo = new CultureInfo("en-US", false).TextInfo;
62          productName = cultInfo.ToTitleCase(productName);
63          productName = productName.Replace(" ", "");
64          productName = char.ToLower(productName[0]) + productName.Substring(1);
65          if (productName.All(Char.IsLetterOrDigit) == false)
66          {
67              returnMessage.Text = "Please only use numbers and letters in the product
     name!";
68              break;
69          }
70
71          if (!priceValid(productPrice.Text))
72          {
73              returnMessage.Text = "Invalid price format";
74              break;
75          }
76          decimal price = Convert.ToDecimal(productPrice.Text);
77
78          if (imageUpload.HasFile)
79          {
80              if (imageUpload.FileName.Substring(imageUpload.FileName.IndexOf(".")) != "
     png")
81              {
82                  returnMessage.Text = "The image must be a PNG!";
83              }
84              try
85              {
86                  string fileName = imageUpload.FileName;
87                  imagePath = "~/images/" + fileName;
88                  imageUpload.SaveAs(Server.MapPath(imagePath));
89              }
90              catch (Exception except)
91              {
92                  returnMessage.Text = "File Upload failed with error " + except.Message +
     ", please contact a developer";
93                  return;
94              }
95          }
96          else
97          {
98              errorAppend = " with no image";
99              imagePath = "NONE";
100         }
101
102         returnMessage.Text = "Product created named " + productName + ", priced at    " +
     common.formatPrice(price) + " and displayed as " + displayName + errorAppend;
103         productQueryTable.newProduct(productName, 0, Convert.ToDecimal(price),
     displayName, typeDropdown.SelectedValue, Convert.ToString(Session["currentUser"]),
     imagePath, bandBox.Text, descriptionBox.Text);
104         customLogging.newEntry("The product " + productName + " was created");
105         productsTable.DataBind();
106     } while (false);
107 }
108
109 private static bool priceValid(string input)
110 {
111     string noPoint = input.Replace(".", "");
112     foreach (char c in noPoint)
113     {
114         if (!char.IsDigit(c))
115         {
116             return false;
117         }
118     }
119     try
120     {
121         string decimalPlaces = input.Substring(input.IndexOf('.') + 1);
```

```
122        if (decimalPlaces.Length > 2)
123        {
124            return false;
125        }
126        return true;
127    }
128    catch
129    {
130        return true;
131    }
132 }
133
134 static class deletingProductsPersistent
135 {
136     public static string product;
137     public static bool deleting;
138 }
139
```

The Page The page contains a single GridView, which is marked as editable. This means a small button appears at the side which, when clicked, allows the user to change the stock and price values of a product: Additionally, the page contains a set of controls forming a form for adding products. These

are mostly textboxes, with an image upload dialog and a dropdown to dictate whether the product is an image or a clock.

The Code The only properties on this page are:

- **productQueryTable**, which is a loval instance of the table adaptor for queries on the product table.

- **deletingProductsPersistent** is a static class which serves an identical purpose as the one in the user configuration page. It enables the "click twice to delete" functionality.

I have declared it in the root of the class because it is used in multiple methods. The methods in the code are as follows:

- **Page_Load** does not serve any purpose on this page.

- **productsTable_RowCommand**, like on the users configuration page, is called when a button is pressed inside a template field (ie. button column) in the GridView on the page (called productsTable). It starts with an if statement checking if the command name is not "deleteProduct" - this is needed because this table is set to be editable; when the edit button is pressed, this method is run as part of the function of the Edit button but because I have added custom code to it, it would run that code and delete the product if the button were pressed twice - this if statement prevents that from executing if the command is not labelled as a delete command. It does not need to check anything else about the RowCommand because it will only be executed intentionally when the user wants to delete a product.

  The method gets the name of the product in question using the CommandArgument of the RowCommand (which is set to be the index of the row of the table in the ASP code) as the index of a row, and getting the value of the first cell of that row. In this cell will be the product name, as it is laid out in the table. Before deletion commences, the deleting and product attributes of deletingProductsPersistent are checked to ensure this is the second click on the delete button. If they aren't as expected, then they are set to the current conditions, and an appropriate message is shown to the user. The method is ended.

  The first step of the deletion is to run System.IO.File.Delete to remove the image of the product from the directory it's in; the directory is obtained by using Server.MapPath to get the actual path of the " /images" folder and appending it with the name of the product so it points to a specific file. Three queries are then run to keep referential integrity of the database enforced:

  1. cartsAdaptor.deleteProducts removes all the instances of the product which is being deleted from user carts. This also ensures users cannot purchase products which no longer exist.

  ```
  1 DELETE FROM carts
  2 WHERE (productName = ?)
  3
  ```

  2. marketAdaptor.deleteProducts removes all instances of the product being deleted from the physical markets records. This step is not strictly necessary as the client should enforce this in the real world before deleting the product from the application, but it still enforces referential integrity. Since the client should do this, problems shouldn't arise with products which are still on sale being deleted.

  ```
  1 DELETE FROM market
  2 WHERE (productName = ?)
  3
  ```

  3. ordersAdaptor.removeProducts removes all instances of the product from the orders table. This is not a good solution to this problem: the integrity of the database's references must be enforced but deleting past orders sabotages data and will show inconsistencies in sales. Therefore, this needs revision which I will address later.

  ```
  1 DELETE FROM orders
  2 WHERE (product = ?)
  3
  ```

The final query productQueryTable.deleteProduct deletes the product itself from the products table:

```
1  DELETE FROM `products`
2  WHERE (( `productName` = ?))
3
```

The return label is updated appropriately and the deletion is logged. productsTable.DataBind is run to refresh the contents of productsTable to show the deletion.

- **productAddButton_Click** is run when the add button is clicked. It consists of a do loop which is ended with a "while (false)", meaning it will only be run once. I did this because it allows me to perform checks and break the loop if any of them fail. A string called errorAppend is initialized to contain any extensions to log messages. The first check is customSecurity.sanitizeCheck, as with all database-related operations in the application. If it is failed it, as usual, displays the standard error message and breaks the loop. The needed variables are declared (displayName, productName and imagePath). displayName and productName are both set to the contents of the product name textbox. All the characters in productName are check to be letters or digits with the IsLetterOrDigit method of the Char class. If any of them aren't, the user is notified and the operation broken. priceValid is run on the price box to ensure it is in the appropriate format for the price. The price variable is set to the value of the price box. The uploaded image is checked in multiple ways:

  1. If there isn't a file selected in the dialog, set the image path as being "NONE" and errorAppend is set to " with no image" so that the log will reflect this.
  2. If the image given isn't a PNG file, the page rejects the image and notifies the user.
  3. The image is attempted to be uploaded. This portion of the code is enclosed in a try - catch statement because I have little experience with image management in .NET and so am expecting errors during development.

  The image is uploaded by getting the file name of the image and appending it to " /images" so when it is displayed the control looks in the images folder in the root directory. Finally, the image is actually saved using the SaveAs method of the image upload dialog; the path it is given is generated by giving the image path variable to the method Server.MapPath which will return a full directory string for the image.
  The return label is updated to inform the user that the operation was successful, and a query is run to store the products and all it's information in the database:

```
1  INSERT INTO products
2  (productName, stock, price, displayName, productType, creator, [image], [band],
        description)
3  VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)
4
```

  The creation is logged and productsTable is refreshed with DataBind to reflect the creation.

- **priceValid** exists to check if the given number is valud as a price in GBP. A variable is created which holds the price without the decimal point to check if it is all digits, which is carried out for each character in the string using char.IsDigit(). If this check or any other is failed, "false" is returned. The digits left of "." are obtained using substring, and the length of the resulting string is checked. If there are more than 2, the check is failed. If there aren't any, an exception will be thrown which will be caught by the try catch statement and true will be returned. If all these checks are passed, true is returned.

Revision 21/04/18 I have made some changes to the code:

1. I have removed the do-while loop in productAddButton_Click and replaced all the break statements with return. This solution was essentially redundant as the whole code block was encompassed in a method which could just be broken with "return".

```
1  string errorAppend = "";
2
3  if (customSecurity.sanitizeCheck(new string[] { productNameBox.Text, productPrice.
       Text, bandBox.Text, descriptionBox.Text }) != true)
4  {
5      returnMessage.Text = customSecurity.sanitizeErrorMessage;
6      return;
7  }
8
9  string displayName = productNameBox.Text;
10 string productName = displayName;
11 string imagePath;
12 TextInfo cultInfo = new CultureInfo("en-US", false).TextInfo;
13 productName = cultInfo.ToTitleCase(productName);
14 productName = productName.Replace(" ", "");
15 productName = char.ToLower(productName[0]) + productName.Substring(1);
16 if (productName.All(Char.IsLetterOrDigit) == false)
17 {
18     returnMessage.Text = "Please only use numbers and letters in the product name!"
        ;
19     return;
20 }
21
22 if (!priceValid(productPrice.Text))
23 {
24     returnMessage.Text = "Invalid price format";
25     return;
26 }
27 decimal price = Convert.ToDecimal(productPrice.Text);
28
29 if (imageUpload.HasFile)
30 {
31     if (imageUpload.FileName.Substring(imageUpload.FileName.IndexOf(".")) != "png")
32     {
33         returnMessage.Text = "The image must be a PNG!";
34         return;
35     }
36     try
37     {
38         string fileName = imageUpload.FileName;
39         imagePath = "~/images/" + fileName;
40         imageUpload.SaveAs(Server.MapPath(imagePath));
41     }
42     catch (Exception except)
43     {
44         returnMessage.Text = "File Upload failed with error " + except.Message + ",
        please contact a developer";
45         return;
46     }
47 }
48 else
49 {
50     errorAppend = " with no image";
51     imagePath = "NONE";
52 }
53
54 returnMessage.Text = "Product created named " + productName + ", priced at   " +
       common.formatPrice(price) + " and displayed as " + displayName + errorAppend;
55 productQueryTable.newProduct(productName, 0, Convert.ToDecimal(price), displayName,
        typeDropdown.SelectedValue, Convert.ToString(Session["currentUser"]),
       imagePath, bandBox.Text, descriptionBox.Text);
56 customLogging.newEntry("The product " + productName + " was created");
57 productsTable.DataBind();
58
```

2. Replaced the foreach loop in priceValid with a .All function so that less iteration is needed:

```
1  if(!noPoint.All(Char.IsDigit))
```

```
2 {
3     return false;
4 }
5
```

3.2.18 Market

The market page is used by employees to manage what products they are taking to physical markets. The page inherits from Employee.Master and therefore is inaccessible by ordinary users.

```
1  defaultDataSetTableAdapters.marketTableAdapter adaptor = new defaultDataSetTableAdapters
       .marketTableAdapter();
2
3  public static marketItem[] takingItems = new marketItem[0];
4  public static productList productsList;
5
6  protected void Page_Load(object sender, EventArgs e)
7  {
8      if (Convert.ToInt32(Session["marketInitialized"]) == 1)
9      {
10         populatePage();
11     }
12     else
13     {
14         productsList = new productList();
15         takingItems = new marketItem[0];
16         populatePage();
17         Session["marketInitialized"] = 1;
18     }
19 }
20
21 private void populatePage()
22 {
23     populateMainPanel();
24     populateTakingPanel();
25 }
26
27 private void populateMainPanel()
28 {
29     var controlSet = productsList.generateControls();
30     productsListPanel.Controls.Clear();
31     int i = 0;
32     foreach (productPanel current in controlSet)
33     {
34         Button addToStall = new Button()
35         {
36             ID = productsList.list[i].productName + "AddToStall",
37             Text = "Add",
38             CommandArgument = productsList.list[i].productName
39         };
40         addToStall.Click += new EventHandler(addToStall_Click);
41         addToStall.Attributes.Add("runat", "server");
42         current.Controls.Add(addToStall);
43         productsListPanel.Controls.Add(current);
44         i++;
45     }
46 }
47
48 private void populateTakingPanel()
49 {
50     var data = adaptor.getStallItems(Convert.ToString(Session["currentUser"]));
51     List<marketItem> tempList = new List<marketItem>();
52     foreach (DataRow current in data)
53     {
54         marketItem tempItem = new marketItem();
55         tempItem.product = new product(Convert.ToString(current[1]));
56         tempItem.amount = Convert.ToInt32(current[2]);
57         tempList.Add(tempItem);
58     }
59     takingItems = common.appendArray(takingItems, tempList.ToArray());
60     generateTakingControls();
61 }
62
```

```
63  private void generateTakingControls()
64  {
65      takingPanel.Controls.Clear();
66      foreach (marketItem current in takingItems)
67      {
68          productPanel panel = new productPanel(current.product);
69          panel.ID = panel.ID + "Taking";
70          foreach (Control currentControl in panel.Controls)
71          {
72              if (currentControl.ID != "" && currentControl.ID != null)
73              {
74                  currentControl.ID = currentControl.ID + "Taking";
75              }
76          }
77          foreach (Control currentControl in panel.Controls[1].Controls)
78          {
79              if (currentControl.ID != "" && currentControl.ID != null)
80              {
81                  currentControl.ID = currentControl.ID + "Taking";
82              }
83          }
84          TextBox amountBox = new TextBox()
85          {
86              ID = current.product.productName + "_AmountBox",
87              CssClass = "marketAmountBox",
88              Text = Convert.ToString(current.amount)
89          };
90          amountBox.TextChanged += new EventHandler(amountBox_TextChanged);
91          amountBox.Attributes.Add("runat", "server");
92          panel.Controls.Add(amountBox);
93          takingPanel.Controls.Add(panel);
94      }
95  }
96
97  private void addToStall_Click(object sender, EventArgs e)
98  {
99      Button btn = (Button)sender;
100     bool existing = false;
101     foreach (marketItem current in takingItems)
102     {
103         if (current.product.productName == btn.CommandArgument)
104         {
105             existing = true;
106         }
107     }
108     if (existing)
109     {
110         returnLabel.Text = "You've already got that product in your taking list, please
        update it's amount or remove it!";
111     }
112     else
113     {
114         marketItem newItem = new marketItem();
115         newItem.amount = 1;
116         newItem.product = new product(btn.CommandArgument);
117         takingItems = common.appendArray(takingItems, newItem);
118         generateTakingControls();
119     }
120 }
121
122 private void amountBox_TextChanged(object sender, EventArgs e)
123 {
124     TextBox box = (TextBox)sender;
125     string productName = box.ID.Split('_')[0];
126     int index = 0;
127     foreach (marketItem current in takingItems)
128     {
129         if (current.product.productName == productName)
```

96

```csharp
130          {
131              break;
132          }
133          index++;
134      }
135      takingItems[index].amount = Convert.ToInt32(box.Text);
136  }
137
138  protected void applyButton_Click(object sender, EventArgs e)
139  {
140      defaultDataSetTableAdapters.productsTableAdapter productsAdaptor = new
          defaultDataSetTableAdapters.productsTableAdapter();
141      foreach (marketItem current in takingItems)
142      {
143          int currentStock = Convert.ToInt32(productsAdaptor.getStock(current.product.
          productName));
144          if (currentStock < current.amount)
145          {
146              returnLabel.Text = "Sorry, we only have " + currentStock + " of " + current.
          product.productName;
147              return;
148          }
149      }
150      adaptor.deleteStall(Convert.ToString(Session["currentUser"]));
151      foreach (marketItem current in takingItems)
152      {
153          adaptor.newStallItem(current.product.productName, current.amount, Convert.
          ToString(Session["currentUser"]));
154          productsAdaptor.updateStock(Convert.ToInt32(productsAdaptor.getStock(current.
          product.productName)) - current.amount, current.product.productName);
155      }
156      returnLabel.Text = "Changes successfully applied";
157      populatePage();
158  }
159
160  protected void endStallButton_Click(object sender, EventArgs e)
161  {
162      Server.Transfer("~/marketEnd.aspx", false);
163  }
164
165  public struct marketItem
166  {
167      public product product;
168      public int amount;
169  }
170
```

The Page   The page holds two panels for products. The first is for an exhaustive list of all the products in the database (productsListPanel), the second is for a list of all the products the employee has selected to take with them to the market (takingPanel). Furthermore, there are two buttons to apply the changes to the market and to end the stall and record sales made. As usual, a return label is included.

The Code   The properties and variables on this page are:

- **adaptor** is a instance of the market table query adaptor.

- **marketItem** is a custom structure designed to allow a list of products to be made with amounts attached. It is functionally identical to cartItem, having product "product" and integer "amount" properties.

- **takingItems** is an array of marketItem which represents all the items the user is taking with them on the physical market. It is initialized as an empty array so that AppendArray can be run on it.

- **productsList** is a vanilla instance of productList to display all the items in the database.

97

The methods in the code are:

- **Page_Load** here checks if the market has been initialized with the unique session "marketInitialized", stored as a 1 or a 0. If it has already been initialized, only populatePage is run to save processing power and time. If not, productsList is initialized as a vanilla productList and takingItems is initialized as a blank array of marketItems. populatePage is run and the session is set to "1" to indicate that the page has been initialized already.

- **populatePage** is essentially just a method to bind together populateMainPanel and populateTakingPanel. It simply runs them both.

- **populateMainPanel** fills the panel which should show all the available items in the first panel on the page. It creates a variable with the context-sensitive keyword "var" to hold all the panels generated by productsList.generateControls then clears the controls which are already in the panel on the page. A foreach loop is run with an external index tracked to run through all the productPanels in the controlSet variable described above. A new button is made, given a unique ID generated with the product's name, labelled with "Add" and given a CommandArgument of the product's name. A new EventHandler is initialized and assigned to the new button to run "addToStall_Click" when the button is clicked. The "runat:server" attribute is added so the button functions correctly and the button is added to the current productPanel. The current productPanel is added to the controls list of the panel waiting for it on the page with Controls.Add. This process is repeated for all productPanels in the array.

- **populateTakingPanel** fills the second panel with all the items the user is taking. It runs the getStallItems query into the variable data (a DataTable):

```
1 SELECT productName, amount
2 FROM market
3 WHERE (employee = ?)
4
```

To obtain all the items the current employee is taking to market. It creates a new list called "tempList" to hold all the items after they are processed. A foreach loop is run for each row in the DataTable (ie. each type of item the employee is taking) Within the loop, a new marketItem is instantiated, it's product property set to a new product instance generated using the name of the product gotten from the first cell in the row, it's amount set with the second cell in the row and it is added to the temporary list of marketItems. The contents of tempList are appended to the existing takingItems array, and generateTakingControls is run.

- **generateTakingControls** creates all the controls and productPanels needed to show in the bottom panel to display what the user is taking with them to market. It starts by clearing the existing controls out of the panel to start with a clean slate, and runs through each item in takingItems. A new productPanel is instantiated and "Taking" appended to it's ID. All of the controls within that productPanel have their IDs appended with this so that ASP.NET doesn't throw exceptions based on different things having the same IDs (which will happen otherwise since we are making new copies of productPanels to add to the bottom main panel which would otherwise be entirely identical (in terms of IDs) to those in the top). Foreach loops are used for this purpose. The first foreach loop runs through all the controls in the whole panel and appends all those whose IDs aren't blank with "Taking". The second foreach loop runs through all the controls in the textWrapper panel within the productPanel (control index 1) and does exactly the same thing.
A new textbox is created and it's ID generated with the name of the product appended with "_AmountBox"; the underscore is there so the the product the textbox refers to can be easily identified in "amountBox_TextChanged".

- **addToStall_Click** runs when the add to stall button is clicked. It gets the button which called it by filtering it's sender by type button. It creates a boolean to determine whether or not the product the user is clicking on is already in their taking list and assigns it false by default. A foreach loop runs through each marketItem in takingItems and sets existing to true if any of their

names match that of the product the user is trying to add. An if statement checks the boolean, cancels the operation and returns an error if the user is trying to add something they are already taking. If they are adding a new product, a new marketItem is initialized, it's amount set to 1 to start with and it's product set to a new instance of product generated with the correct name. takingItems is appended with the item, and generateTakingControls is run to reflect the addition.

- **amountBox_TextChanged** runs when the return key is pressed within one of the amount text boxes for the products the user is taking with them. It gets the sender by filtering sender by type TextBox and gets the name of the product in question using the text before the underscore in it's ID. It uses a foreach loop with an externally managed index to find the relevant item in takingItems by breaking when it finds the correct marketItem. The amount property of the marketItem in takingItems at the index of the foreach loop when it broke is changed appropriately.

- **applyButton_Click** runs when the apply button is clicked. It opens a new instance of the products table adaptor to run queries on that table from. A foreach loop runs through all the item types the user is taking and checks if the amounts are more than the available stock using getStock:

```
1 SELECT stock
2 FROM products
3 WHERE (productName = ?)
4
```

If any of them are trying to take more than available then an error message is shown and the operation is cancelled before any changes are made. If not, the database table is cleared of the stall in order to readd it. This is a terrible method of updating the table, but at the time this code was written I was pressed for time. Another foreach loop is run through all the items in takingItems and for each one two queries are run; newStallItem on the market table to add the item back:

```
1 INSERT INTO `market` (`productName`, `amount`, `employee`) VALUES (?, ?, ?)
2
```

And updateStock on the products table to subtract the amount the employee is taking from the current stock.

```
1 UPDATE products
2 SET stock = ?
3 WHERE (productName = ?)
4
```

Finally, the return label is updated with an appropriate message and populatePage is run to reflect the changes.

- **endStallButton_Click** runs whenm the button to end the stall is clicked. It redirects the user to marketEnd.aspx to manage the sales made.

Revision 01/05/18 I realised that the code checking the stock amount in applyButton_Click wasn't performing the correct task. As-is, it was checking the amout against the non-reserved stock which meant it wasn't accounting for the stock already reserved in the table. I have amended the contents of the first foreach loop in that method to the following:

```
1 int reserved = Convert.ToInt32(adaptor.getReserved(current.product.productName));
2 int currentStock = Convert.ToInt32(productsAdaptor.getStock(current.product.productName)
    );
3 if (currentStock + reserved < current.amount)
4 {
5     returnLabel.Text = "Sorry, we only have " + currentStock + " of " + current.product.
    productName;
6     return;
7 }
8
```

The query "getReserved" contains:

```sql
1 SELECT amount
2 FROM market
3 WHERE (productName = ?)
4
```
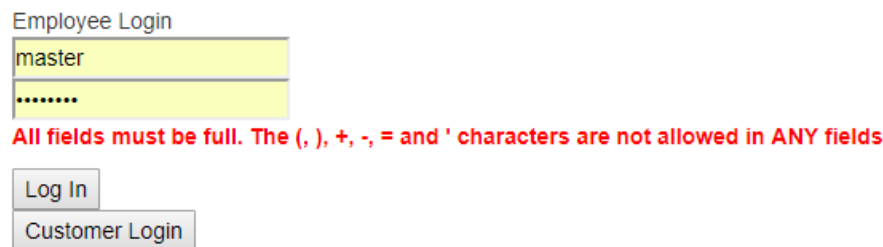
4 Testing

4.1 To Inform Development

4.1.1 Employee Login

This page is relatively simple, and so I am expecting few problems. The master account exists by default, as I manually added it and it is designed so that it cannot be removed or changed without manual intervention. Once the project is complete, I will remove, disable or change the account's password. The testing is as follows:

1. **SQL Injection**. I tried a common SQL injection: ''; DROP ALL TABLES; –'' which would ordinarily delete all the tables in my database. The application rejects the '';'' so no SQL injection will work here. Note that the yellow boxes indicate my browser has autofilled them according to my saved ''master'' and password combination. This does not compromise testing.
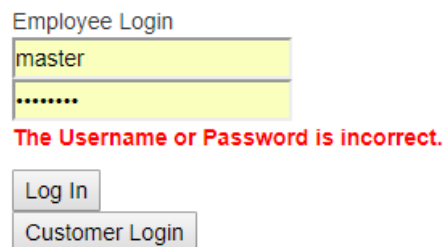
2. **Logging in with incorrect credentials**. I attempted a multitude of situations:

   (a) Using an existing username with a bad password.
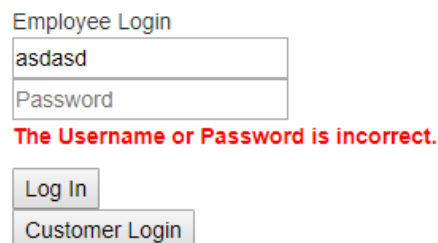
   (b) Using a fictional username and password. Note that the password box is blank here as it should be because I am using a different username so my browser does not attempt to correct me.

   (c) Leaving the password box blank with a known username.

(d) Leaving the password box blank with a fictional username.

(e) Leaving both boxes blank.

3. **Correct Credentials**. This works as expected, redirecting me to the employee homepage.

4.1.2 Customer Login

This uses practically identical code to the employee login page, so I have kept testing brief.

1. **SQL Injection**. I used the same string ”; DROP ALL TABLES; –” to test this.

2. **Incorrect Credentials**. I tried all three scenarios as shown in the testing of the employee login page, and got the same result each time:

3. **Correct Credentials**. I entered the sample customer credentials I created ("customer" and "password"). The site behaved as expected, and redirected me to the default page.

4.1.3 Cart Page

The cart page employs a lot of untested logic, and so requires relatively extensive testing.

4.2   To Inform Evaluation

5   Evaluation

5.1   Testing

5.2   Usability Features

5.3   Evaluation

I think I have

5.4 Maintenance