iOS Application for Inventory in Small Retail Stores

By Andrea Savage

Software Engineering

College of Engineering

California Polytechnic State University

December 17, 2016

Abstract

Currently, small retail stores with low technology budgets such as those right here in San Luis Obispo are struggling to integrate new technologies into their companies. This mobile application built for iOS with a Firebase backend is seeking to remove their barriers to entry. I built this application to give small retail stores a customizable application that allows them to display products electronically to customers and maintain accurate inventory both in one place. The construction of this application hinged around three major design decisions: UI design of the color management views, organization of the database, and accessing the database with optimized data queries. Anticipated future improvements are the addition of a tabbed controller for product detail pages, a section store map for clothing locations, and custom store sales events.

Contents

1 Introduction

As shopping moves online, small retail stores are struggling to utilize new technologies to retain customers and better manage inventory. Through my senior project, I want to provide small retail businesses a way to expedite receiving inventory and advertise their products electronically.

1.1 Problem

In cities like San Luis Obispo, there are multiple small retail stores with only a few employees and a small supply chain. These stores usually record inventory by hand, which can easily become a very lengthy and error-prone process. They also do not have websites, mobile applications, or other means of electronic advertising. This limits the number of customers these stores can reach. Currently, large retail stores employ entire technical departments to design their mobile apps and online stores. Smaller stores often hire expensive designers to create websites or use generic store inventory technologies. Stores that do not have the technical budget or are unwilling to devote time and energy resources to making the switch are barred from entry into the modern marketplace.

1.2 Background

These small retail stores record their inventory counts on paper only when they first receive the products. One such store is Lulu Luxe, a boutique in downtown SLO that has around 15 employees. Lulu Luxe receives their product shipments at the owner's home and never even has store inventory records. The store solely keeps inventory of their most expensive jeans. Although the store tracks these expensive items, the counts are infrequent and product loss is still quite common. Lulu Luxe and similar stores use inventory methods that limit the accuracy of the store's supply chain and damage their overall profit. Stores that use more accurate inventory counting methods experience less product loss by being more aware of both product theft and product misplacement. Inventory counts through a mobile app provides constant availabilty of the counts and culpability to the logged in employees who record the product movement.

1.3 Solution

To address the problem, I wanted to create a mobile application that encompasses both the organized inventory and customer outreach needs. The app would provide

a way for a store to scan their inventory into a database and a separate interface for customers to view the scanned inventory. The inventory management will provide a faster, more accurate inventory process. The customer interaction will allow stores to retain customers and advertise new items on a new platform. The store will have readily available updated inventory counts with newly received inventory added and checked out items removed. Providing a customizable, store-specific application will make it easy for stores to maintain brand recognition with their customers.

I believe that a mobile application is a good way for stores to ease into more advanced technologies without a large budget. Using mobile phones does not require the store to purchase any new technologies such as barcode scanners for their employees. Also, mobile applications appeal more than websites to the under 25 year old customer base that stores are often looking to reconnect with.

1.4 Predicted Design Decisions

Going into this project, I foresaw that the most important design decisions I would make would all be user interface (UI) based. User interface design is an important aspect of a successful application and I thought that creating simple-to-use layouts would be the most difficult part of this project. I predicted that there would be five main use cases, each requiring one view controller design: scanning in inventory as an employee, removing sold inventory as an employee, viewing inventory as an employee, viewing inventory as a customer, and searching inventory as a customer. After reassessing the required app capabilities, I added a third user type called Manager to the preexisting Employee and Customer user types. Adding a user type created eight more necessary use cases and view designs. I also anticipated that making a stable application would require consideration of user interactions and the usability of view designs. Though designing view layouts became the most time-consuming difficulty, designing my database was also an important design decision I solved.

2 Key Design Decisions

A large portion of my time for this project consisted of designing different aspects of my application. Three of the decisions that most shaped my project were the design of the UI for adding and editing colors, the layout of my database, and the ways I accessed that database.

2.1 Editing Colors UI Design

2.1.1 Permissions for User Types

The main user interface (UI) design choice that I made was about the design and view layout for adding and editing product color options. The first half of the decision was which user types (Manager, Employee, and Customer) could add and edit the colors. I eventually chose that the adding new colors permission would only be given to store managers. They could utilize this ability when creating a new product or by selecting 'Edit Product Colors' from their homepage. Employees can view the color options and modify the inventory counts for each product color, but not modify those options. By only allowing managers this control, updating store inventory product orders will be a simple task. There are so many options and slight name variations of colors in the clothing world that I believed the ability should be kept to a small number of people. Also, adding or removing product colors without plans to repurchase the option does not happen very often at these smaller stores. When it does happen, store managers are the ones who make the new product orders and I anticipate it will be easy for them to enter the change to the mobile app at same time as placing the new order.

Originally, I decided that both employees and managers would be able to add and edit the colors. This helps with efficiency because employees will not require manager assistance when scanning in the morning inventory deliveries. The managers will also not have to plan ahead to enter the updates before employees require them. As I mentioned above, this could cause confusion about naming, where multiple employees add the same new color but with slightly different names. However, I mainly decided against this option because it provides employees with an option they don't really need and over complicates their homepage.

Instead of providing managers with an 'Edit Product Colors' button, I could provide a special manager-only feature to the product viewing screens that allows them to edit the colors from there. That way it would be obvious which product the user intends to modify. Providing the action in this format also maintains UI consistency for users and does not require new view layout designs. Almost every other view screen of this app is based around product selection and emphasizes what product the user is viewing or modifying. Still, the 'Edit Product Colors' action is meant for updating a product only when their product inventory changes permanently. Store managers place the product orders for different color options, so the changes they need to make are already ordered by product and then by color. Having a specific menu option on the homepage for editing a product color is more intuitive and user-friendly because it specifically states the action's purpose.

2.1.2 Page Layout Design

After making the permissions decision, I had to select the most user-friendly way to format the editing view. In my UI design, the user selects a product first, either by creating a new product or in the 'Edit Product Colors' button from the homepage. Then, a view opens that allows the user to enter a color name, color type, and select a color from the color picker. The color type of a new color is the encompassing color (red, orange, black, etc.) that the new color falls into and is required for future color ordering. My full view for editing these colors is shown below:

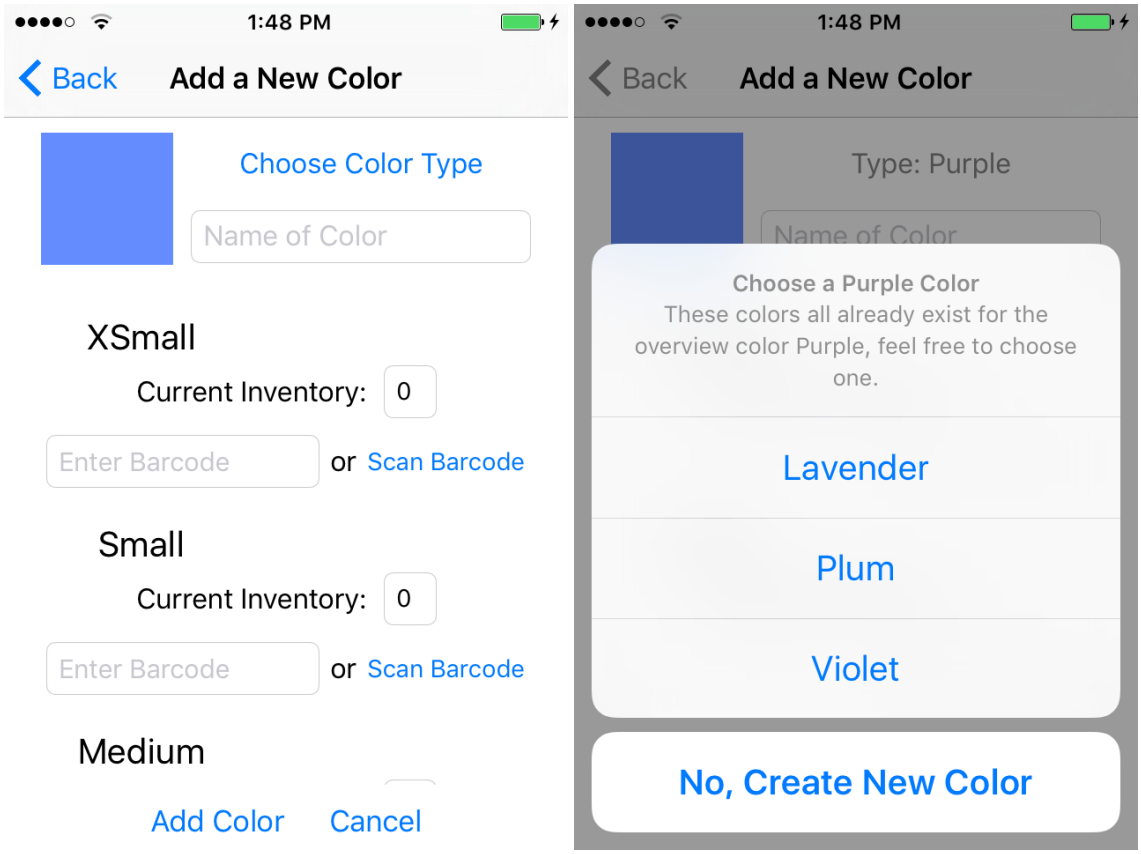

Figure 1: Color Editing Screenshots

To prevent users from having to reenter preexisting product colors, this view design provides a complete list of already created colors of the color type in a popup view after a color type is selected. The color type is at the top of page on the right

hand side, where the user often looks first. This helps to amplify the chances that the user will select a color type and view their current options before starting the process of creating a new one. After determining the color properties for a selected or created color, the user then enters the current inventory counts of this new color for each size (xsmall, small, medium, large, and xlarge). The user is able to scan or type in the applicable barcodes for each size, if they are known, for easier scanning by employees later.

An alternative to this format is providing a popup list of colors with an 'Add New Color' option as the top button. The popup would display each color as a colored swatch next to the assigned color name. A mockup of this design is shown in the figure below.

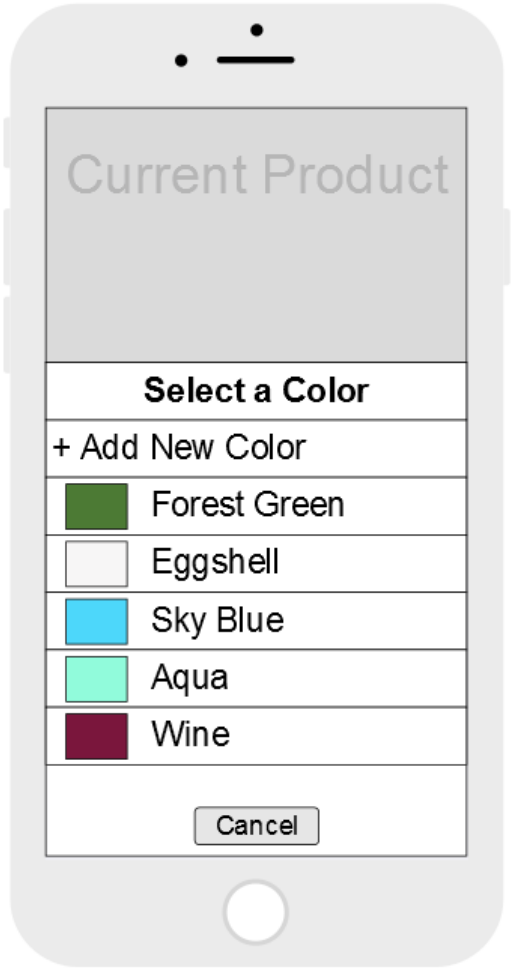

Figure 2: Color Popup View Mockup

The add color button would open yet another popup view in the form of an alert view. This screen would allow the same three main features of a color: the name, type, and RGB color picker values, to be input by the user. This additional popup view is shown in the below figure.

Figure 3: Color Adding View Mockup

The two possible actions, adding a new color and selecting a color, are split into definitive views. This makes the role of the user at each step more intuitive and simpler. Providing all of the color options in a selectable list in this form makes the already created color options obvious to the user and highlights the simplicity of

reusing them. This format requires plenty of popup action sheets and alert views. Though they split views into smaller tasks, multiple popup menus can confuse the user by not being added to the navigation stack even though multiple new subviews are open. This model also does not include the important combination of color and sizes. In this view, adding the size inventory counts is separate from the new color, and having product inventory counts for each color and size combination makes it easier for customers to discover if a store has what they want.

Another alternative to modify these colors would be to provide a list of all the options as small colored squares. The user would first select a color and then be able to add or delete products with the respective product sizes from that color.

Figure 4: Color Swatches Alternative View Mockup

Consistent display formats are important because they create a more intuitive overall user experience. This design maintains the same format for displaying colors as the other product detail views in this app. However, unless the store has a very small amount of product colors, the overall color options would overwhelm the user. Using the above design also allows managers to edit all aspects of a specific color at one time. Though useful in edge cases, this violates the user-friendly goal of consistent display formats. Here, users select a color and then a related product while in other views, colors are displayed as a sub-aspect of each product. This format is also ordered in reverse of the database organization, where colors are listed under each product, which means that this would require additional data access queries.

2.2 Database Organization

To have shared and persistent data for all users of my application, I decided to use a cloud-hosted NoSQL database that stores data in JSON format called Firebase [1]. One of the first design decisions that I made for this project was how to format my data and organize it in my database. I ended up organizing my data with five top-level keys as shown in the diagram on the following page.

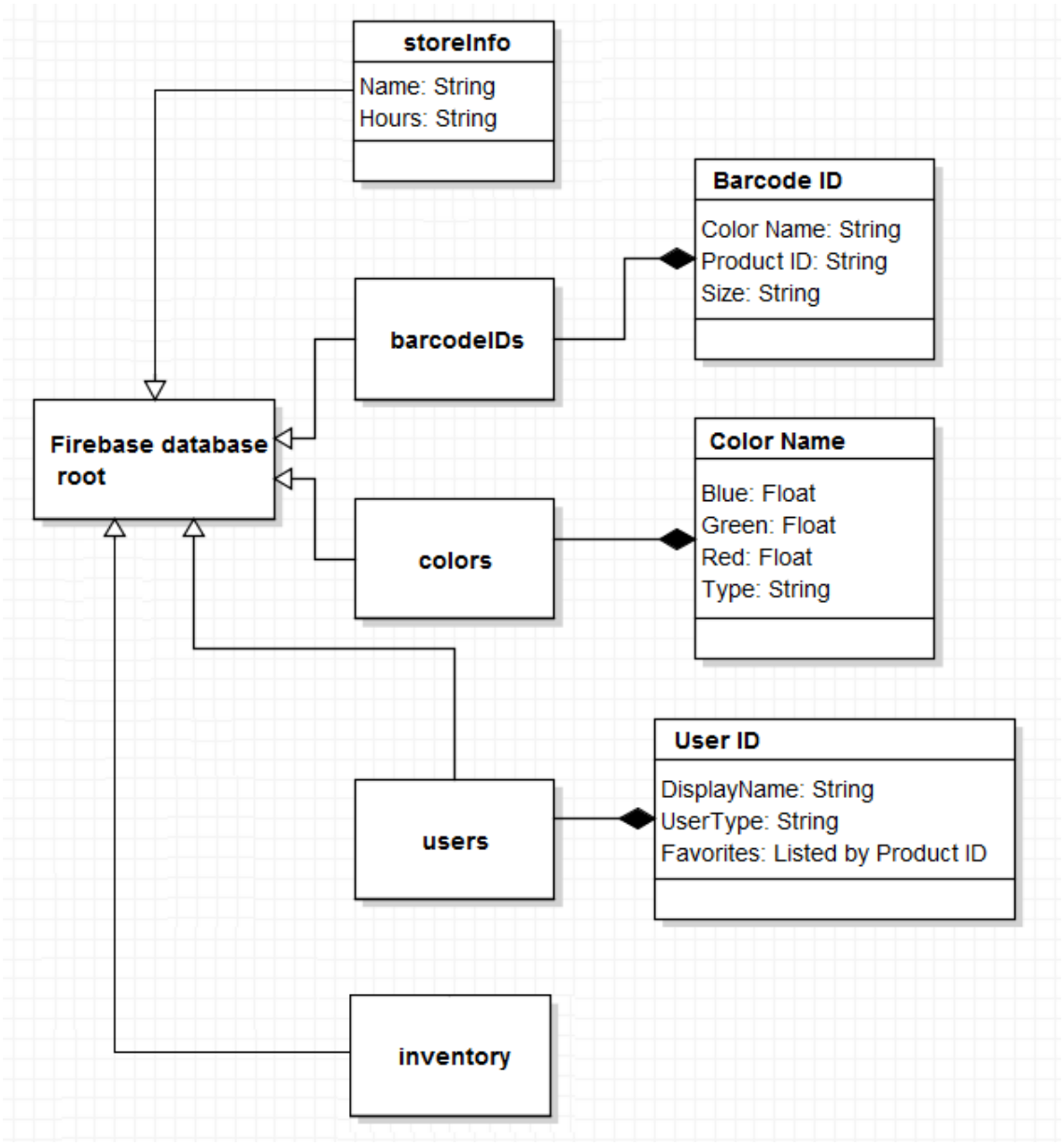

Figure 5: Database Overview UML Diagram

StoreInfo is an object that holds all of the information about a store that is provided to the customers and can be modified by the store managers. The barcode IDs are listed under barcodeIDs with the ID as each object's key. They have three properties, the product ID, name of the color, and size that the ID belongs to. The

color options of a product are displayed to users in my product views as small colored squares. These squares require red, green, and blue (RGB) values to have the correct background color. Colors can have a wide variety of names and so each color's RGB values is listed under colors by its assigned name. Colors also have a type field, which specifies what encompassing basic color (i.e. Red, Orange, Black, Gray) the color falls under. The database also has a list of all the users that have accounts listed under users by an unique Firebase-assigned user ID. Users have a modifiable display name, a user type (Customer, Employee, or Manager), and a list of favorite products that they have saved. My database design also has all of the products the store sells listed under inventory. The format of a product in my database is shown below.

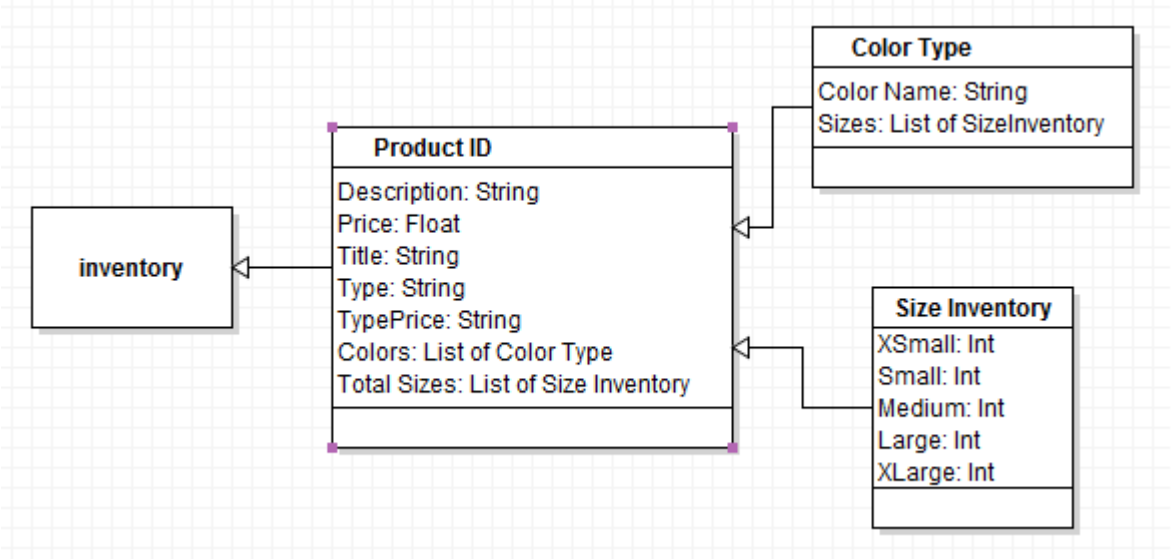

Figure 6: Database Inventory UML Diagram

Products are indexed by their unique product ID or universal product code (UPC). Products have a title, price, type, description, colors, and sizes. The only missing data is a product image, which is stored in a separate aspect of Firebase called Firebase Storage, which holds large data files. Currently, my app supports five clothing sizes, xsmall, small, medium, large, and xlarge, so they are specific child keys of the total size list. Products also have a list of colors where each color has a size list of the inventory counts for just that color. This leads to more data to be stored, but makes providing accurate inventory counts to the customer simpler.

2.2.1 Alternative One

I originally had the barcode ID values listed under their respective products. This format keeps all of the data about one product confined to one location which requires a single database query for providing that information to a user. This is especially important when users may want all of the current barcodes for a product to be displayed. However, this layout overly complicates the data by nesting it into the products. NoSQL databases are meant for a flatter data hierarchy. If the data is instead split into separate paths, also called denormalization, it can be efficiently downloaded in separate calls, as it is needed [1]. This refers to the fact that in databases such as Firebase, returning only certain levels of data is impossible and nested data will be returned unnecessarily whenever a product is accessed. Also, my organization addresses displaying current barcodes for a product by the barcodeID list. A list of barcodes is easily requested by querying Firebase for a list of products with the child key Product equal to the searched product's value.

2.2.2 Alternative Two

Listing the color RGB values separately instead of adding the three additional RGB values to where each product already has its color options listed seems repetitive. With all of the color data listed underneath the products, a second database call for the color information is unnecessary. This has the same problem as barcode IDs in that this nested data, though slight, will be returned whenever a product is accessed. This method also risks repetition because if multiple products use the same color, then the RGB values will be repeated and more memory will be used. Not having an explicit color list also makes displaying preexisting color options to the user more difficult.

2.2.3 Alternative Three

I could also index products by their title rather than their product ID. Users are more likely to input a title to search for a product rather than a product ID. Using the title allows for the product ID to be optional, so users can enter the ID later or not at all if the store does not use IDs. Database indexes must be unique and exist for every item. Products will always have title names and requiring them to be unique will help stores with product recognition. The issue is that titles are of variable length and extremely long titles would be difficult to store and for users to remember. Plus, enforcing title uniqueness requires extra code while product IDs are already ensured to be unique by the product manufacturers.

2.2.4 Alternative Four

Finally, I chose to divide the products by product color and allocate each color an inventory count list of the five sizes. Instead, I could have chosen for the inventory counts to be stored by size first and then color. This would simplify finding all of the product colors available for a user's preferred size. Ordering color-first instead simplifies filtering for all of the inventory available of a certain colored product. Employees enter newly arrived inventory using the second ordering, so it will be more commonly used. Also, the number of product colors is variable while the five sizes are static. Placing the likely larger division of data closer to the root of the database reflects the flattened hierarchy that is preferential for Firebase.

2.3 Accessing the Database

After the database was laid out, the next step was to access the data. Firebase has API calls that make listening for data changes and accessing data values fairly simple, but accessing the data in more complex ways required a design plan. My app has a menu on the customer homepage for users to filter all of the products available at the store. Users can filter by product ID or by any combination of:

- product titles
- product prices
- product types
- product colors

I had to find a way to narrow down the products shown on the homepage based on the user's input while also respecting the memory limits of the phone and filtering time lengths. After much deliberation and testing, I ended up using a slightly complex order of search values that combines both client-side and server-side searching.

For optimization on the client device, I expedited the filter loading time by continuing to page the data by only loading 15 products at a time and by using asynchronous threads to allow the user to still interact with the view. For the server-side, one important thing I learned was that there is an indexOn command you can set as a rule for your Firebase database that states which child keys the database should base organization on for better optimized search queries. I used this rule to optimize searching for products by their Title, Price, Type, and special TypePrice keys:

inventory: .indexOn: ['Title', 'Price', 'Type', 'TypePrice']

My final approach for presenting product results when the user presses the 'Filter' button in the search menu follows the decision tree below.

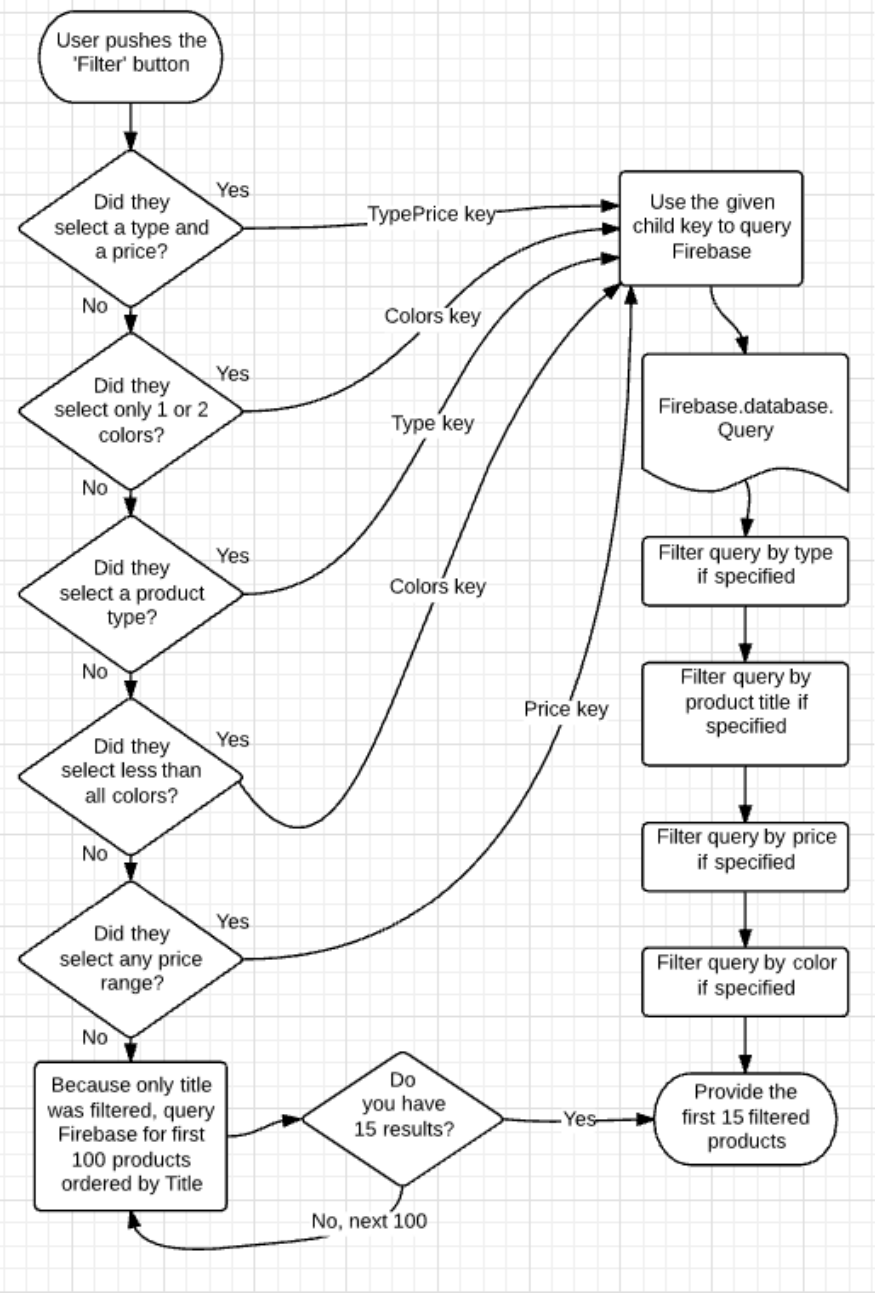

Figure 7: Filtering Decision Tree Diagram

The algorithm first checks if the user specified both a clothing type and a price range that is smaller than the maximum price. If they have, Firebase is queried for all products that fit the type and price requirements by checking the special 'TypePrice' field of the products. The 'TypePrice' field is just a key whose value for a product is the product type followed by a space and then the product price. This adds repetitive data to each product, but expedites queries by allowing two values to be searched at once. This is extremely important for databases such as this one because there is no way to simulate complex SQL queries such as joins even though it is often necessary to compare multiple fields of an object when accessing the database.

If both of those fields are not specified, the method checks for a small amount of colors, then any specified product type, any specified color preferences, and finally any price range. These requirements are ordered by the criteria most likely to eliminate a large portion of the products. When a criterion is matched, Firebase is queried for all matching products. The resulting Firebase.database.Query object is interpreted as a snapshot and the list of children is filtered by any remaining filter variables. The first 15 of the filtered product results are displayed to the user. The client-side filtering first attempts type, then title, then price, and lastly color. This is ordered so that any client-side color filtering has the least amount of products possible to search. Comparing two arrays of colors is inefficient and avoided as much as possible.

If none of the query criteria are met, the user has only specified a product title search string. Searching a non-case sensitive string by query is impossible, so instead the first 100 products ordered by Title are received from the database and parsed for the correct title string. If at least 15 products are not found, Firebase is queried for the next 100 products. This continues until 15 products are found or there are no more products. This is an extremely inefficient filtering method which is why it is the last resort of a series of criteria.

2.3.1 Only Server-Side Filtering

Rather than a combined filtering approach, I could have chosen to filter all of the products through queries to the database. This approach would be quite fast because Firebase optimizes their structures for fast queries, especially when using the indexOn rule. This also requires only simple API calls as all of the work is done server-side. This would often be the best solution for working with large amounts of data, but this method does not allow joined or multiple key queries except for tricks like my 'TypePrice' field. Users can filter by four different variables, which requires checking

four different product child keys, so this is not a feasible option.

I also realized that this method caused problems with my product title filtering option. Providing users with a text field to search in implies that the entered text does not need to perfectly match the upper and lower case and can be anywhere in the searched field. I wanted to fulfill that user assumption, but the query refinement options for Firebase do not allow for comparing anything but the beginning of the value. The queryStartingAtValue and queryEndingAtValue method calls are also case sensitive, which further complicates the matter.

2.3.2 Only Client-Side Filtering

To avoid all of those issues, I could instead filter all of the products client-side. This solves both of the problems I was having by allowing product title searches to be more inclusive and products to be filtered based on multiple values. Product titles can be searched by the user-specified string using Regex and case issues can be fixed by converting all strings to only lowercase characters. The products can be narrowed down by one variable at a time or by complex list filtering statements. However, loading all of the products to the phone for filtering would take way too many resources. The loading time would be astronomical if even possible, and loading would potentially take more memory than the app is allotted. I could also remove the filtering by product title option from my filtering menu. This would have less of a need for complex filtering organization and filtering with only three variables would take a shorter processing time on average. I decided against choosing this because product titles are the most memorable part of a product for customers and users expect the filtering menu to allow for a free-text search field.

3 Future Work

3.1 Tabbed Product Details

Currently when users select a product from the list on the customer home screen, the product details open in a new view on top of the navigation stack. Instead of opening in a new view, the product details would appear in a new tab of the homepage. This tabbed view format allows for a better overall user experience. Users, especially customers, prefer easy to navigate apps and this flattened view hierarchy will simplify use. Users can more easily switch between products and will always be able to access home with only one click. Being able to have several product

details views open as well as the home view lets customers 'save' products for later perusal without adding them to their favorites list.

3.2 Clothing Map

A future expansion to this project could include a map of the inside of the store split into numbered sections. Each product could be given a section number, so customers visiting the store would find it easier to locate the products they liked on the app. Store managers would also be able to use this feature to create new store layouts for the employees to set up. Stores often change their layouts which causes confusion for employees both with moving the products and finding the products after they have moved. Besides helping customers, this addition could help employees find products when they are requested. My app currently provides a 'Store Info' view for customers to see the store's name and business hours. The map could be displayed in this section as well as on each product's details page with the respective section highlighted.

3.3 Store Sales

My current application does not have very many features that help stores retain and attract customers. To further expand this goal, I would like to add a sales feature to the app. The feature would allow store managers to create special sales events on the app that reflect in-store sales. A manager would be able to set a starting and ending date for the sale, allowing them to schedule events in advance. The sales event could notify customers of the event and allow the store title banners to be modified with special sales graphics. Managers could also enter adjusted product prices, specifying mark-off rates for different product groups so employees would not have to change each product individually.

4 Reflections

Most of my first quarter of my senior project went to learning iOS development skills while the second quarter was devoted more to actually developing the product. If I were to restart this project with the domain-specific knowledge I have now, I would do several things differently.

4.1 Different Technology Choices

4.1.1 Use a Mac

This may seem obvious to many people, but using the hardware designed to work with software makes your life easier. I believed that running Xcode in an OSX virtual machine, or VM, on my PC would save time and money because I wouldn't have to buy a new computer. Due in part to the fact that my computer is more than two years old, this was not the best choice. Creating the VM on Windows with VirtualBox was pretty simple, but did not give me access to features that in hindsight would have helped my productivity. Two examples of this are touchpad scrolling and CocoaPods. Even though my touchpad supported horizontal and vertical scrolling, I could not get this to work with the VM, so I went without this convenient feature. Another useful feature that I had trouble getting to work was CocoaPods. CocoaPods is a dependency manager for iOS projects that allows a developer to easily integrate third-party features and tools into their project. I eventually succeeded in using CocoaPods to use a third party color picker tool [4], but I first struggled with integrating many other potential tools manually.

4.1.2 No iOS7 Support

I originally supported iOS version 7 and above in my application. This happened because the phone that I borrowed to program on was an iPhone 4, which is only supported through iOS version 7.1. Near the end of the quarter, I ended up switching to a slightly newer phone that allowed me to upgrade to only supporting iOS 8 and above.

Only supporting version 8 and above provides developers with more design options. IOS 8 takes the UIActionSheet and UIAlertView subcomponents and provides a simpler to use UIAlertController class. My application uses multiple action sheets and alert views in most of the view controllers. With the UIAlertController, I would have spent less time setting each of the subcomponents up and would be able to have less complex handlers for each view controller. Also, AVFoundation.framework is updated in iOS 8, with better camera and video controls. AVFoundation.framework is one of the most useful frameworks available for iOS development and is used both in my barcode scanner and my image selection view. This framework allows developers to "record, edit, and play audio and video" as well as react to objects detected in the video, such as barcodes [2].

If I had chosen to support only iOS 9, I would have access to an even bigger selection of auto layout options. One of the biggest improvements is stack views, which

allows developers to easily create a horizontal or vertical stack or linear grouping of views without having multiple tiers of nested views. Auto layout was one of my biggest struggles because such a powerful tool also has a steep learning curve. Stack views would have made my layouts simpler and easier to modify when designs altered.

4.2 Use Apple Specific Features

During the beginning of my project development, I had a hard time adjusting to Apple UI design standards. This meant that I spent a lot of time implementing my own features that iOS does not provide. Creating my own drop down menus, barcode scanner, and other features never took very long, but because I was using these non-standard objects, I had difficulty up keeping them. Any time I needed to update a view layout, modifying my custom objects took quite a bit of time. In hindsight, I should have just designed my view layouts with provided objects and added in custom features after all of my layouts were finalized.

4.3 No Size Classes

Auto layout supplies size classes as a way to have different layouts for different phone sizes and orientations. I started formatting all of my layouts using size classes and found that modifying any piece of a layout after employing size classes is extremely complicated. There are nine size classes, which are not as independent as they appear. Size classes are hierarchical, which means when the placement of a view in any size class is changed, the developer must then modify the sub and super size classes. This ends up with the developer needing to modify all nine size classes and likely having to start over some of the sub classes because the current constraints cannot conform to the new addition. In summary, size classes are a feature that allows applications to be adapted to many iPhone and iPad configurations, but should be added after development is done and all layouts are finalized.

4.4 Inherited View Controllers

Several of my view controllers have the same basic view components, error alert views, text fields with 'Done' buttons, and product detail views (an image view, a title field, a price field, a 'Choose Color' action sheet, a 'Choose Type' action sheet, a description text view, and a size inventory display). Even though each controller displayed, created, or edited these views differently, the underlying data required and user touch handling was the same. Realizing earlier that large chunks of my basic

view handling code could be recycled with inherited view controllers would have saved me development time. Originally designing my view controllers hierarchically would also have eliminated some of the modifications to finished views I had to employ later in development.

4.5 Alternative to Firebase

Finally, the platform I used for persistent and universal data storage and authentication is called Firebase. Firebase has a cloud-hosted NoSQL database that stores data in JSON format [1]. The user authentication and large data storage tools that Firebase provided were extremely helpful. The database was difficult to structure as is not SQL-based and difficult to use for filtered data queries. More information on this is in Section 2.3, Key Design Decisions: Accessing the Database. I wish that I had realized data filtering would be a problem and looked into more alternatives or used a technology I was already familiar with like MongoDB. I also would have saved myself database restructuring time if I had remembered earlier that universal product codes, or UPCs, are not individual for each item, instead one is assigned for each size and color of a product. Using a database format that I was more comfortable with would have made my multiple database restructures less time-consuming.

5 Conclusion

Developing this senior project has taught me a lot about mobile development. I have learned about database design, especially for databases that are not based on SQL. I have learned to design these databases for effective queries and about filtering data both client- and server-side. I spent a great deal of time improving my UI/UX design abilities. I expanded my knowledge of data persistence and the different options available to mobile developers. I have also learned a great deal of both OSX and iPhone skills and improved my Git habits of committing more often and committing useful messages to help later revisions. Overall, I wish that I could have continued development on my app for a little longer, but I am happy with the outcome of my senior project and am excited to use the skills I learned out in the real world.

My full source code is available at:
https://github.com/ajsavage/Senior-Project-Retail-Inventory

References

[1] "Working with lists of data in ios," Firebase, November 2016. [Online]. Available: https://firebase.google.com/docs/database/ios/lists-of-data

Google's Firebase documentation and reference guide.

[2] A. Inc, "Apple developer api reference," 2016. [Online]. Available: https://developer.apple.com/reference/avfoundation

Definition and Reference on the AVFoundation Framework for iOS.

[3] B. Johnson, "Uicollectionview tutorial," September 2016. [Online]. Available: https://www.raywenderlich.com/136159/uicollectionview-tutorial-getting-started

Extremely helpful tutorial and tutorial website for provided iOS tools and objects.

[4] J. Kasperi, "Swift hsv color picker," September 2016. [Online]. Available: https://cocoapods.org/pods/SwiftHSVColorPicker

Third-party color picker view subclass available on the CocoaPods website.