CashTag iOS Application Final Report

Kevin J. Tsui

June 6th, 2016

Overview

CashTag is an iOS application running specifically on the iPhone that simplifies management of cash purchases. The app stores user-inputted cash transactions and allows for quick and simple management for the transactions. Each transaction carries metadata which allows for specific queries to made including location-based queries and time-based queries to name a few standard methods. The goal of this application is

to make cash purchase management as simple as can be while also be informative and easy-to-read even with vast amounts of transactions.

Preface

Due to certain health issues that landed me in the hospital for several weeks and that this is my 3rd Iteration of CashTag after scrapping the project twice before, my timeframe for completing this project was significantly reduced. However, a considerable amount of work has been achieved to reach this point of CashTag. More details about both my health hindrances and project pivots will be explained in the "Difficulties" sections.

Technologies Used

- Platform: iOS 9.2 platform
- IDE: Xcode 7
- Libraries:
    - CoreData (Apple-API for data persistence)
    - OrderedDictionary.swift (by Lukas Kubanek)

Swift Basics (Need-to-Knows)

This section will discuss some basic Swift principles that will help explain what is going on behind the scenes in Swift.

Swift is an object-oriented language with quirks that make it rather difficult to get accustomed to. Some of the methodologies employed are foreign to most languages but are designed specifically to make iOS applications run more smoothly and effectively. As a result, the learning curve as a developer can be challenging but necessary.

Callbacks

One of the main functionalities to understand about Swift is that there is no main function that the developer can code. Everything is programmed and executed through **callbacks** which is orchestrated by the compiler. There are different ways in which to invoke callbacks including built-in functions provided by the iOS platform that get called when a specific event happens and closures which allow for something to be run asynchronously. For the vast majority of situations, we use these built-in callbacks functions and add our logic to the functions in order to develop the app. There does exist a main function; however, only the compiler has access to it. It is in this compiler-driven main function where there is a loop constantly running and waiting for events to take place to initiate these callback functions. This all happens on the main thread. As stated earlier, things can be done asynchronously which is a part of the platform's concurrency. Apple has a high emphasis on optimization, so things are run asynchronous on multiple threads as often as possible without the developer even having to worry about it.

Protocols and Delegation
Another important design functionality to know about Swift is that it is a single-inheritance language. However, it uses a design pattern called “Delegation” to replicate multi-inheritance. Protocols, Apple’s equivalent to Java Interfaces, are used as the primary method of supporting this multi-inheritance behavior. A class can “delegate” responsibilities to another class if it follows the guidelines of the protocol. This helps to make the code more portable and future-proof. A well-crafted example can be found in the Swift Language Reference Guide under the Protocol section.

(Scroll to the section of the document labeled “Delegation”)
https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/Protocols.html

Swift Data Structures
Something to note about the language is that it only supports three main built-in data structures: Arrays, Dictionaries, and Sets. Therefore, in order to have a data structure that behaves like a stack, we could use an array to replicate that behavior as an example.

Storyboard and UI Design
While most things can be done programmatically, Apple does provide basic UI design with Xcode’s **Storyboard**. This interface builder allows for a basic visual representation of what each screen in the app will look like. However, this isn’t necessarily a direct correlation to what the end-product will look like since customizations can be done programmatically. Xcode’s storyboard is constrained to a feature called **Autolayout**. Autolayout allows a single UI design to be compatible and portable across all Apple device platforms. This means instead of having to create separate UI designs for an iPhone and iPad, one could design the UI once and have it compatible for both. This is accomplished with vector mathematics. However, the tradeoff with this high portability feature is that it makes UI design much more convoluted and time consuming. This means accomplishing simple tasks such as placing a button in the middle of the screen aren’t simple anymore since the middle on one device could be 200 pixels from the edge of the screen while the middle on another device could be 2000 pixels from the edge of the screen.

Data Persistence
The main way to handle persisting data in iOS is using the standard built-in CoreData package. This includes something called an **NSManagedObjectContext** which serves as a type of disk storage database on the iOS device. One can write data to the context and retrieve later even if the app closes down. Data requests to this quasi-database differ than standard queries. One must create a “fetch request” and specify what it should be looking for in the database which is

based on the specified “entity description”. The data returned will be in an array of a generic type which the user can cast to a certain type and perform whatever actions they please.

MVC Design pattern .

Swift’s main design pattern of choice in developing applications is known as the “Model-View-Controller” design pattern or “MVC” for short. The reason behind this is to ensure that important parts of the application could be portable and stable during events like platform updates or changes to XCode or the changes in project environments.

The Model

The Model’s main objective to hold all of the application’s business logic. This means that the core of what is actually happening in the application belongs to the Model. Theoretically, by coding the core logic into the model, one could potentially run the entire application in any environment whether it being the command line to the iPhone screen itself.

The View

The View is where all user-viewable related objects and items are handled. This means the navigation bar in the top of screen, “back” button, textfields, labels, maps, etc. all belong under the view. Swift helps abstract this portion of MVC by keeping these objects in XCode’s “Storyboard”. Storyboard is the iOS application user interface builder.

The Controller

The Controller is the bridge between the Model and View. This allows communication to flow between the other 2 sections which otherwise have no means of communication. This design pattern allows for circumstances such as a change to the model to have no effect and no way of breaking the functionality of the View and vice versa.

CashTag MVC Implementation .

The following will outline the implementation of the MVC design pattern in CashTag. Only the main components of the application will be outlined, all extraneous and less pertinent classes will be ignored.

Transaction [Model]

The main object of the system that holds information about a certain cash transaction.

- **Date**: The date of the transaction
- **PayType**: Details what type of method was used to pay the transaction (e.g. Credit, Debit, Cash, etc.)
- **Title**: From whom you purchased the item from in the transaction (e.g. Target, Costco, Firestone, Chipotle, Ralph’s, Apple Store, etc.)
- **Cost**: The amount of money paid for the transaction

- **Location**: Where the transaction took place (e.g. “Europe”, “San Luis Obispo”, “Orange County”, etc.)

OrderDictionary [Model]
This is a custom data structure that allows for key-value coding functionality that is seen with a standard dictionary data structure with the ability to hold the order of insertion. Therefore, this allows the ability to acquire values through the data structure’s index (like an array) as well as the key (like a dictionary). This is necessary to keep the list of transactions in the home scene ordered by date.

TransactionTableViewScene (Storyboard) [View]
The app’s home page where the user will see all the transactions in a list on-screen. This scene (created in storyboard) is the actual visual representation of the work being done by its controller.

TransactionTableViewController [Controller]
The controller of the app’s home page where all the work is done to provide the necessary information to the view. This includes establishing a connection to the NSManagedObjectContext, acquiring the data, making any changes to the context, populating data, passing information to-and-from other controllers, and many more functions.

TransactionDetailViewScene (Storyboard) [View]
This is the scene where the user will be able to input data into a transactions whether it is adding data for a new transaction or editing data from an existing. This is also created in Storyboard.

TransactionDetailViewController [Controller]
This controller will take the information from the UI elements in the view (textfields, buttons, etc.) and interpret the data. This will usually mean take the information storing it into a transaction that will eventually get stored into the context.

SearchTableViewScene (Storyboard) [View]
Here will list all of the transactions based on the search preferences inputted by the user.

SearchTableViewController [Controller]

The job of this controller is to take the search inputs and find all transactions that abide by the search. This can be done dynamically in realtime using the search bar or with search filters.

SearchDetailViewScene (Storyboard) **[View]**
This scene is almost identical to the TransactionDetailViewScene. This is so the user can edit transactions from the search table directly.

SearchDetailViewController **[Controller]**
Here the controller will look up the appropriate transaction that has been selected from the search results in the context and populate the UI elements in the view with the existing data. Changes will be saved if any are made.

SearchFilterViewScene (Storyboard) **[View]**
The user will be able to input their search preferences in this scene. This information inputted will be passed to its corresponding controller.

SearchFilterViewController **[Controller]**
This will acquire the search preferences from the UI elements in the view and fetch the results which will be given to the SearchTableViewController to display in its table.

How It Works

This section will very briefly detail how the application will accomplish the core of its duties. The following information is a considerable simplification of what is actually going on in the code but is meant to give a higher-level overview of what is happening and how.

CashTag has one main object type which is the “Transaction”. A transaction object contains properties (also known as instance variables in Java) each containing metadata such as location, date, title, cost, etc. In order to achieve data persistence, we must store these transaction objects into an NSManagedObjectContext which only accepts NSManagedObjects. Therefore, Transaction objects will be subclassed from NSManagedObjects.

Apple provides a built-in protocol for tables inside Xcode (UITableViewController) which is what is being used by CashTag to display transactions in listed format. We must implement the various different functions in order to have the table working as intended. For example, there are callback functions under the UITableViewController that account for what explicitly will show up in a singular row of the table, what actions can be done in a singular row, how many sections the table will be divided into, and several more.

Data gets passed back-and-forth between different screens (known as scenes) in the application using “segues”. Segues allow the developer to decide where the screen will navigation to as well as prepared the incoming scene to handle the incoming data. A classic example of this in CashTag is passing a transaction’s data from its creation in TransactionDetailViewController back to the home page TransactionTableViewController where the controller will insert it into the context.
The user can search for specific transactions based on search preferences inputted by the user. While on the search page, the user can access a separate scene where they can detail their search preferences. Then the filters will be applied and the search page will returned a filtered list of transactions based on the metadata that they were looking for.

Design Decisions

This section will explain briefly some of the decisions I made and why I made them when implementing CashTag.

Creating a solid foundation in which to build the application upon had to be extremely well-thought out. As a result it took multiple iterations before I found a sustainable foundation in which I could build feature upon.

OrderedDictionary
Rather than using a standard dictionary or array data structure, I realized I needed an OrderedDictionary since I needed a way to access values not only through keys but through indexing. OrderedDictionaries allow for this since it combines the functionality of a dictionary and array.

An instance of why I would need this is trying to group transactions by month and year. With this in mind, if a user added a transaction in Mar 2016, then one in Feb 2016, and another in Jan 2016, we would expect a grouping ordered like “Mar 2016”, “Feb 2016”, and the “Jan 2016”. However, with a dictionary since there is no order index stored, the order would be alphabetized showing up as “Feb 2016”, “Jan 2016”, “Mar 2016”.

FetchedResultsController
I used a FetchedResultsController as my means of accessing data from the managed object context because the standard fetch request doesn’t allow for more complicated queries. Since I needed a way to replicate a standard SQL “Group by” aggregation, I learned that I was able to replicate this type of request using a FetchedResultsController.

With this controller I was able to fetch transactions grouped by their date. And since this controller was built to work along the UITableViewController, it also automatically created custom sections in the View’s table each sorted and grouped by the transactions’ dates.

Functionality over design
The application may look rather barebones in terms of design and that is because I elected to focus heavily on the functionality of the application given my reduced timeframe. Also since UI design takes up a considerable amount of time, I found it would not be time well spent trying to conform to all of the Storyboard's autolayout constraints.

Current Progress .

As of this moment, the core functionality of the application has been implemented and is the most stable it has ever been due to revised foundation.

The functionality of the app includes…(not exhaustive)

1. Inserting/Editing/Deleting a transaction from the application and having the data persist
2. Customized and in-depth data retrieval and display in the transactions table view scene
3. Monthly total balance of transactions
4. Dynamic table section headers (grouped by date) that automatically correct/update their balanced with each change/update to transactions
5. Multi-functional search capabilities
    a. Dynamic search that searches and displays results in real-time as the user types in the search bar
    b. In-depth search based on preferences which takes in user-specified preferences and returns a filtered list of transactions
6. Ability to edit/update transactions directly from the search page and have the updates applied in real-time
7. One-Tap-Add - tapping the "+1" button on a transaction creates a new copy of that transaction with the date set to the moment you pressed the button. Useful for recurring transactions.

Difficulties .

This section will discuss the difficulties I've had during the development of CashTag.

Learning Swift
One of the initial main difficulties of this entire project was learning the basic flow of developing an iOS application. There are so many different design patterns that Apple enforces quietly including delegation, segues, and callbacks. This makes research not only necessary but mandatory if one is to learn how to do something as simple as pass data from one scene to the next (as an example). There are still several things that I have yet to understand though I have learned a lot during this process.

Online tutorials aren't fully fleshed out solutions

One of the most prevalent issues from research and looking through tutorials is that the proposed solutions are usually workarounds and do the bare minimum to get a desired functionality rather than develop the code in a way that makes further in-depth development easy and straightforward. This alone has resulted in me redoing entire sections of the application because the proposed way breaks functionality or restricts further development. An example of this would be when the proposed solution only accounts for swipe-to-delete capabilities when I may want multiple swipe-to actions (edit, delete, details, etc.).

The other main issue from research is that a considerable amount of solutions are all in Objective-C which to a Swift native seems near hieroglyphic.

Dealing Storyboard's Autolayout
UI design under the constraints of Autolayout resulted in a tremendous amount of time just putting together what I have now. The luxury of having the UI be portable across all devices makes standard UI object placement extremely tedious and difficult to work with. For example, the screen may look visually appropriate in storyboard until you test the application on a device and find out your textfield is completely off-screen. Nuances such as these have made UI customization something that has been very difficult to do without hassle.

Establishing a stable foundation (leading to multiple pivots)
The application's boilerplate code ended up being the most difficult issue to overcome. Without a clear idea of what data structures, protocols, and implementation methods, creating the application was a shot in the dark and result in having to scrap the project completely on two occasions.

The first pivot came from using too many online tutorial workarounds (which I hadn't realized at the time were workarounds). I eventually came to the point where I wasn't able to find a way to incorporate changes to my transaction model with completely breaking the entire project.

The second pivot came poor choices of data structures and protocols. By opting to use the standard fetch request, I was restricted to a standard list of transactions with no way to customize how they were displayed in the transactions table. Also, I chose a list of lists to represent my transactions which made it impossible to move around transactions based on their month and year. Since both of these decisions were deeply rooted throughout the project, I scrapped it once again.

Now I am currently on my third iteration of CashTag and have so far chosen well in terms of implementation. I have switched to using an OrderedDictionary as my main data structure to hold the transactions and switched to using a FetchedResultsController as my main approach to querying the context. I've also made several other smart changes including subclassing my Transaction class under NSManagedObject so that I can directly store Transaction objects in my context.

Health Issues

During this whole senior project process, I suffered a lung injury twice. The first was a spontaneous collapse of part of my lung, and the second time was a relapse of that same lung. This caused me to be in-and-out of multiple hospitals and several thoracic (lung) surgeons over the course of several weeks. This greatly reduced my ability to work on this project for a considerable amount of time which made getting to where I am more difficult.

Future Considerations .

This section will detail what the next steps would be in further developing CashTag.

Dashboard

Adding a dashboard that holds various extraneous information and analytics would greatly enhance user experience. For example, seeing a list of one's most frequent/recent transactions would be useful. Graphs and stats related to the user's transactions would also be useful.

Data Migration

The main and largest issue of concern for the future is data migration. If there are changes made to the application that potentially alter some of the model objects (i.e. changing the schema of the model's database), the application will crash since there is no code to handle such changes.The only current solution to fixing the crash is to completely erase the app from the device. However, doing this will also erase all previous transaction history. Therefore, this is a serious issue that must be accounted for in the future.

Customizing the UI Design

Another issue would be how to customize the UI to look less standardized and potentially add more aesthetics to make user-experience more engaging. However, as previously stated, Xcode's storyboard is quite difficult to work with.