September 29, 2015

Michael Mehr

AzuResults LLC

Mountain View, California, US - mailto:mike@azuresults.com

My Stamp Collection APP

A Swift Project

# Introduction

As part of my exploration of iOS programming which started early this year, I decided to update a web app I had developed to track my stamp collection. I decided I would port it to iOS to learn how to use the APIs, and specifically CoreData. The original app was written in PHP and ran on a WAMP stack on my Windows 7 development machine. But I had just bought a Mac Mini (2012 edition) and wanted to get things up and running. I took the Stanford iOS/Swift course available in iTunes, and was now armed and dangerous. So Swift 1.0 was available and 1.1 was coming out. How hard could it be?

Well, it’s been about half a year now, and while I haven’t been working on it solely, and I plan on returning to it over and over as my education proceeds, I now have a working model that I can share. I’ve posted it to Github for those who want to follow along. Unfortunately, its modular organization leaves much to be desired at present. The Xcode project is better organized than the files, so hopefully a screenshot or two will help with your understanding.

# Who I Am

My name is Mike Mehr. I’m a software engineer who’s had a good career (1978-2007) in embedded systems programming for the semiconductor test equipment industry. During that time, I also helped found a startup that was working on voice applications for PCs (back in 1984-86). Since 2007, I’ve been working on various projects as a contractor/consultant, mostly in C++ on Windows, but also in PHP for a website server project (WAMP stack). This year, I decided to advance my interest in mobile development, so I’ve been learning iOS since December 2014. I completed a couple small projects in Objective-C, and then dove wholeheartedly into Swift development. This means I’ve worked with (and cursed at) Swift 1.0, 1.1, 1.2, and now 2.0. I’m also interested in Android development and a friend wants to collaborate on a project there. But Apple development is enough to keep me busy for now!

Outside of my career life, I live in the San Francisco Bay Area about a block from Google world HQ, hoping the proximity effect will improve my coding. I play piano at home and tuba in some community bands, and love to read and watch movies, sci-fi and fantasy figuring prominently in my interests. One of my passions is my stamp collection, and so, welcome to my app.

You can contact me professionally at:

Mike Mehr, c/o AzuResults, LLC ([mike@azuresults.com](mailto:mike@azuresults.com) or on Twitter @mmehr2)

The code for this article can be seen here: <https://github.com/mmehr2/StampCollection>

DISCLAIMER: This code is presented as-is. It was designed for Swift 1 and iOS8 and has mostly been brought up to date with Swift 1.2. It is in process (as of the writing of this document) in being ported to Swift 2.0 and iOS9 (see my discussion of Core Data mods). However, it has not been optimized for tutorial purposes, let alone anyone’s usage except mine, and thus may even be hazardous to your mental health. I am working on cleaning it up, hopefully in time for the publication of this article, but if life gets in the way, well, you have been warned. I do apologize in advance.

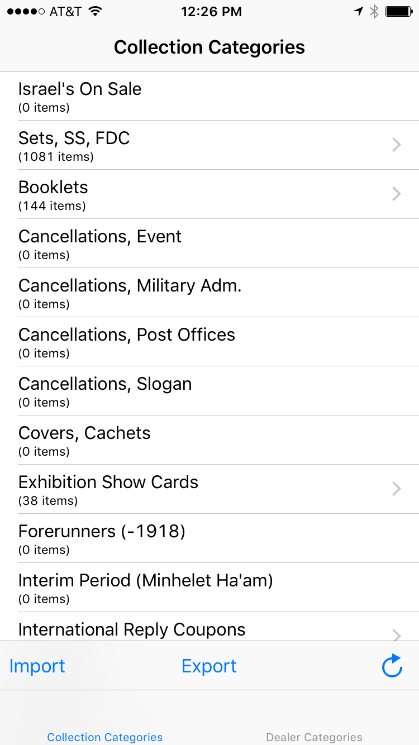
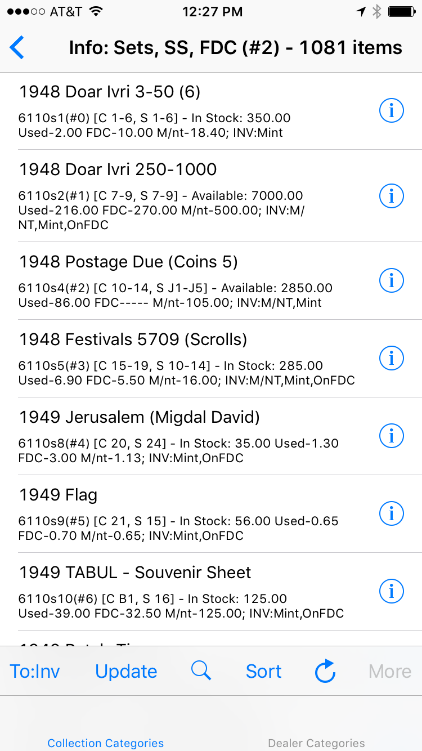
# Project goals

I spent weeks agonizing over how I would deal with the possible data when I was originally developing the web app. This was probably about five years ago, and yet I found myself revisiting the plan again as I worked on the app. Was this the right way to do things? As always, it depended on what I wanted, and my goals for the project do keep evolving. These are my current goals for the project.

1. I wanted to track the replacement value of the collection, not the catalog value
2. I wanted it to reflect the ever-growing complexity of my collecting interests
3. I wanted it to work offline as much as possible, so I could refer to it at stamp shows without needing to rely on possibly slow or inaccessible data connections.
4. I wanted to allow for easy updates as new stamps were issued that I was interested in
5. I wanted to be able to back up my data and restore it from backups
6. I eventually wanted the app to work on my desktop as well, or at least on a tablet, but the portability of the iPhone version was a very desirable feature. So while cloud synchronization may eventually come to the project, I wasn’t as concerned with this up front.
7. While most of my collecting interests are in Modern Israel (since 1948), I could not assume all the items in the collection were from Israel, mainly due to joint issues with other countries (Israel has been doing this several times a year since the mid 1990’s).
8. I have no plans for commercializing this app, so while I wanted to follow best practices as I understood them, I would personally allow a certain lack of optimization if it led to a learning opportunity for me. Since I’m still learning about UI/UX issues, I put all that off into a later version.
9. I realized that my lack of understanding of the CocoaTouch framework or Swift might limit me, so it was best to plan for code refactoring, redesign, and liberal use of comments, in case I couldn’t later figure out what I had done.
10. My goals were to get the functionality I desired, not to limit it to just my code (third party libraries and frameworks are okay), or just to Swift (Objective-C being the most likely alternative).
11. I had already done quite a bit of work entering data for part of my collection (the basic stamps and first day covers, in about 25 albums), so the app would need to import data from the web app at least once. This sounded a lot like the backup/restore feature, so as long as I used the same CSV file format, I could implement the restore function first and bootstrap the web data into my app. This will work as long as all the data used by the app is persisted into the CSV files (or into other files that I would create backups for in addition).
12. Since I had bulk data management features baked in from the start, I wanted to continue to develop this version of the app (call it Phase One) until I reached adequate functionality with a rudimentary UI (so-called CRUD features), and then I would start to develop my plans for Phase Two, in which a better UI would guide the data entry and management processes which I had already debugged.

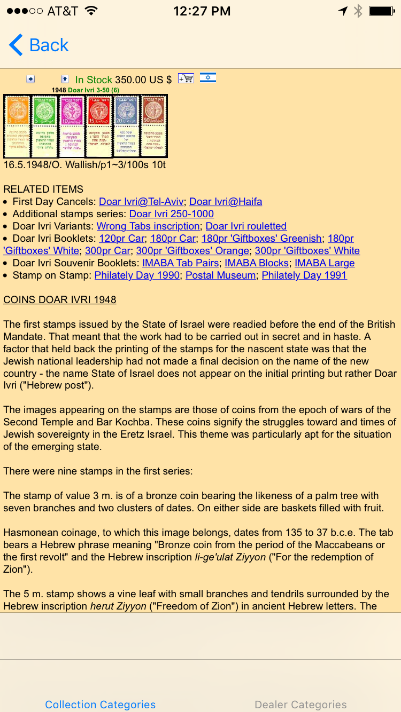
# Action shots of the current app (Phase One, September, 2015, iPhone 6)

Here are some screen shots of this early version of the app doing its thing. Not very impressive as a UI project (yet), but it has some pretty excellent data going on inside.

1A.  1B. 

1A – shot of main screen, collection tab

1B – shot of first screen of data in category 2 (Sets, SS, FDC)

1C.  1D. 

1C – shot of first item detail screen (Doar Ivri 3-50)

1D – shot of web info for that item (detail from vendor site)

# Data modeling

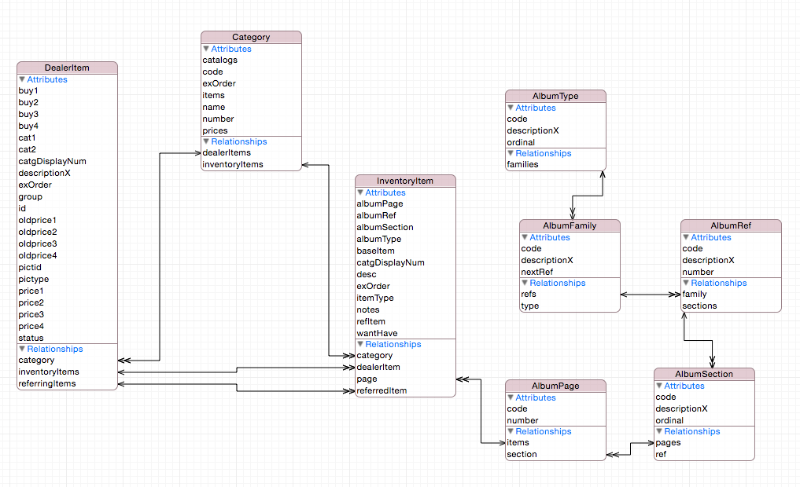
I eventually decided to stick with the basic data design I had used in the web app, but to add or modify it as I wished after I had the basic data imported. This design consisted of two layers.

The first layer (which I have called the Information or Dealer layer) is composed of data taken from dealer websites that include identifying characteristics of each item, and its replacement cost as being sold by that dealer. Since I have a favorite dealer website that has an amazingly comprehensive database, I would use this data as my initial “universe of possibility”.

The second layer of data would consist of the items actually intended for my collection, which I called the Inventory layer. This would include data about item condition, where it was in my collection physically (album and page number), and a reference to the item in the Info layer that it represented. Duplicates would be allowed, and a flag would specify if the item was just on my want list (not in my possession yet).

For those items that I had in my collection that were not being sold by any dealer, I developed a method of inventing my own data for Layer 1 that would represent the particular description and either a generic valuation or some indicator that it wasn’t yet valued. I also had to deal with items which were too new to appear in the dealer database but eventually would so appear.

There are many categories of items in my collection, and I chose to go with the category system developed by the primary dealer website, to make it easier to manage. Some of these categories were not of interest to me. And one category of item came from an entirely different dealer website.

Fig.2 Current CoreData model (Phase 1 on left (3 classes), Phase 2 on right).

# Code design

I organized the code files inside the Xcode project in groups according to their role in the project. This roughly corresponds to the MVC suggested pattern, but also adopts some ideas of MVVC that I was reading about at the time.

My threading strategy for background work was to write everything in easiest fashion (synchronous calls, no explicit use of threading), and at the highest levels of work, implement those calls on a background thread, and transfer back to the main UI thread in its completion handler. This worked for all the use cases I have done so far (bulk data import/export, sort, search).

## Overall Group Organization

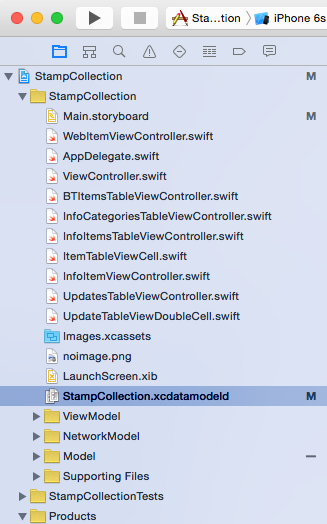


Figure 3A. Project files (overall)

In my scheme, the Storyboard and the View Controllers implementing it live at this level.

## View Model

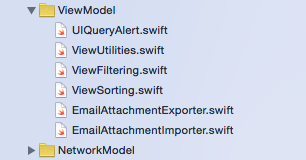


Figure 3B. Project files (view model)

The VM area is for code that manipulates the views and utilities used directly by the VC’s. This includes the sorting and searching utilities that I use to query CoreData.

## Network Model

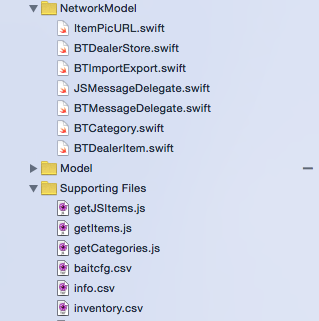


Figure 3C. Project files (networking)

This is where the code that loads the data from the dealer sites lives. This includes the Web Scraping code (in Javascript and Swift); see explanation below.

## Model

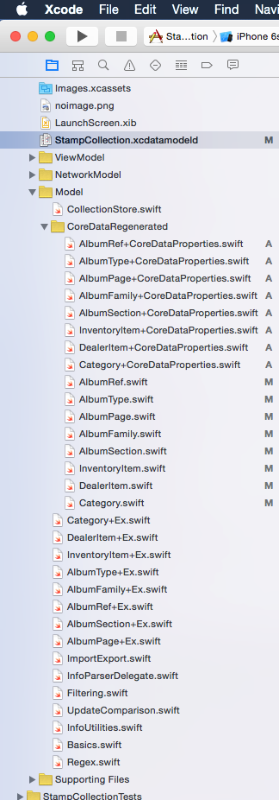


Figure 3D. Project files (model)

This is where the code for the abstract model of the collection lives. It includes Collection Store, which implements the CoreData stack and adapts it for the needs of the program, and also the auto-generated class files that CoreData updates. Due to the design change between Swift 1 and 2 here, there are three sets of files (plain, +Ex, and +CoreDataProperties). As you can see, the files added by Swift 2 are marked ‘A’; files that I had to modify which were originally generated by CoreData under Swift 1 are marked ‘M’. See explanation below.

The abstract model also includes utility classes that implement basic data management features, such as import/export using the 3rd party library for managing .CSV files. Finally, there are general utility classes and functions, including some Swift support for regular expressions that I borrowed.

# Third-party frameworks

I used a few imported Objective C frameworks and classes so far to enable various features I had no desire (or ability) to write on my own. This included the following:

1. CHCSVParser class by Dave DeLong (CSV file import/export)
2. SSZipArchive framework by Sam Soffes (ZIP files for email attachment)
3. TMCache framework by Tumblr (for caching picture files)

The CSV parser was a simple drop-in file that just worked. I highly recommend it due to its comprehensive nature and how it “just works out of the box”. I only had to make one tweak for my app. I needed data fields that contained a space to be properly quoted using double-quote marks when written out to the CSV file. This turned out to be a simple matter of adding the space character to the character set variable named “\_illegalCharacters” in the CHCSVParser.m file. This is a real tribute to the author’s design that I could figure this out from his code.

The ZipArchive framework was a bit harder to figure out, but I persevered, and thanks to help from StackOverflow posts and Google searches, I was able to get it working. Some of the trickiness was getting the project properties set right, importing the right versions of the zip dynamic library libz.dylib, and getting the referenced minizip project included properly. But that all eventually worked as well. Now it only crashes on return from the MFMail API call. (Any ideas on why? Please contact me.)

My use of TMCache is a work in progress, designed to allow caching of .JPG pic files. I wanted a persistent cache that behaved the right way when memory ran out, but also allowed me to not care about where the files were in the OS. I toyed with using CoreData for these files too, but in the end, there were many warnings on StackOverflow about consequences when backing up and restoring CoreData projects that used its persistence features on large files. Currently, the cache is not implemented in my code, so it only downloads pics from the internet if there is a connection, and iOS is free to flush them away if it needs space. I may keep this behavior and only implement the persistent caching on the Mac version. For now, consider this to be redundant code, optional in Phase One.

# Web scraping

Apple provides extensive capabilities for injecting Javascript code into downloaded websites as part of the WKWebKit framework (new in iOS 8). I decided to use this to implement my live data update feature. The original PHP code was based on a more convoluted form of this in which I copied and pasted the web screens into TXT files and then attempted to reconstruct the data from that. It never really worked right, but was good enough that I could patch and tweak it to get it working properly. I hoped that I could improve on that by getting more accurate data directly from the web page, and I was right.

There were two sites that I wanted to gather data from. The primary one was based in Israel, with English data), and had almost all the categories I needed already well organized. It provided a web page for each one that had listings of the individual items in each category. Each listing had a unique ID string, a more lengthy description, availability status, up to two catalog reference numbers, and then from one to four price fields (depending on the category). The second website only represented one category of item I was interested in, but it turned out that in gathering its data, I was able to use the same methods I developed for the primary website.

In order to develop this code, I had to make a few assumptions. The main one was that the design of the website wasn’t going to change dramatically over the useful life of my app. As I went along, I also learned some limitations to this approach, in that I was at the mercy of the developer’s internal changes to their data. But since they had already committed to being a reference catalog of sorts themselves, they already planned on listing many items even if they were not for sale by them, just to have a complete listing available for other stamp collectors. (I believe stamp collecting is still very popular in Israel, and the popularity of the website will assure that it is well maintained for some years to come.) Luckily for me, the website’s main language is still English, so I didn’t have to do much translation from the Hebrew, or indeed deal with anything related to bidirectional text or user interfaces.

So how do you get Javascript code to gather your data and put it back into your Swift app? After studying the documentation on the WebKit framework, and more importantly, reading Mattt Thompson’s excellent article on the technology here (http://nshipster.com/wkwebkit/), I felt I could dive in. Initially, I studied the source code for the web pages at the site, to determine an approach. They used frames a lot, and the frame I was interested in was only one of many. So initially I relied on a feature of the script injection that would allow my code to be injected into all frames of the site, not just the main one. (I later discovered the URL that allowed me to view just what I wanted without the frames.)

There would be two scripts needed. One would get the main category titles and information, and one would get the individual items for a particular category. These would interpret the Javascript DOM for each web page and copy the data I wanted into an object that the script would pass back to the app via a message to the delegate object. Thus I would need two separate types of messages. The WebKit classes automatically translate the Javascript data object sent as the message body into Objective-C data objects, using just NSNumber, NSString, NSDictionary, NSArray, NSDate, and NSNull.

The data for each page of interest resembles a grid like a spreadsheet. HTML table elements are used to great advantage, and this makes the data easy to parse. The columns are labeled (see data property “headers”), and the line items can be collected into arrays of objects, where each line object can have data properties that are named as in the headers, or to save data space, the redundant headers can be eliminated. Various other data properties can be used to pass other descriptive info if needed.

Let’s dig into the details a bit deeper. My class that manages the web data transformation process is named BTMessageDelegate. It contains a hidden WKWebView object that communicates with the website at a given URL and runs an injected Javascript file to generate a message back to the app. Its job is to load the individual data items for one particular category from the website. There will be one of these created to manage each page that we want to grab data from. I later generalized it so that the class would also handle the main category web page, since this was just a special case of the problem and shared much of the solution.

Here is the climactic moment in the getItems.js file, lines 239-40, when the data is packaged up and sent back to the app:

var result = { "dataCount": (tables.length > 0? dataItemCount: -1), "notes": bulletedList, "headers": headers, "items": items };

webkit.messageHandlers.getItems.postMessage(result);

This code constructs the message body from parsed variables and sends it back to the app. The message name is “getItems” and the results body has properties “dataCount”, “notes”, “headers”, and “items”.

For comparison, here is the message returned from getCategories.js:

var result = { "tableRows": categories, "tableCols": headers };

webkit.messageHandlers.getCategories.postMessage(result);

It has “tableRows” and “tableCols” and its name is “getCategories”. But the usage scenario is the same.

It’s easy enough to write handlers for each message in the class, using the WKMessageHandler protocol. The more interesting problem is how to handle the separation of concerns regarding persistence versus network communications. I chose to use another protocol layer called BTMessageProtocol. Methods of the protocol would allow it to post notifications to its client when individual message items were parsed, so that each could be saved to persistent classes individually. A slight complication is that there is some information which belongs in the Category class that is only available on the individual item detail page (parsed by the getItems script). So the protocol has an extra method to notify the client that this information has been loaded from the page.

The class that brings this all together is called BTDealerStore. It contains all the protocol objects and can load them asynchronously in parallel, or one at a time in series. This is governed by the use of the enum named LoadStoreStyle. Options are provided for loading just the categories, just the items, or everything. I originally wanted to limit the use of the web (for example, at a stamp show where hotel data access might not be good), so I could just reload the category list, or the data for one particular category at a time, or the entire batch, for when reception would be good.

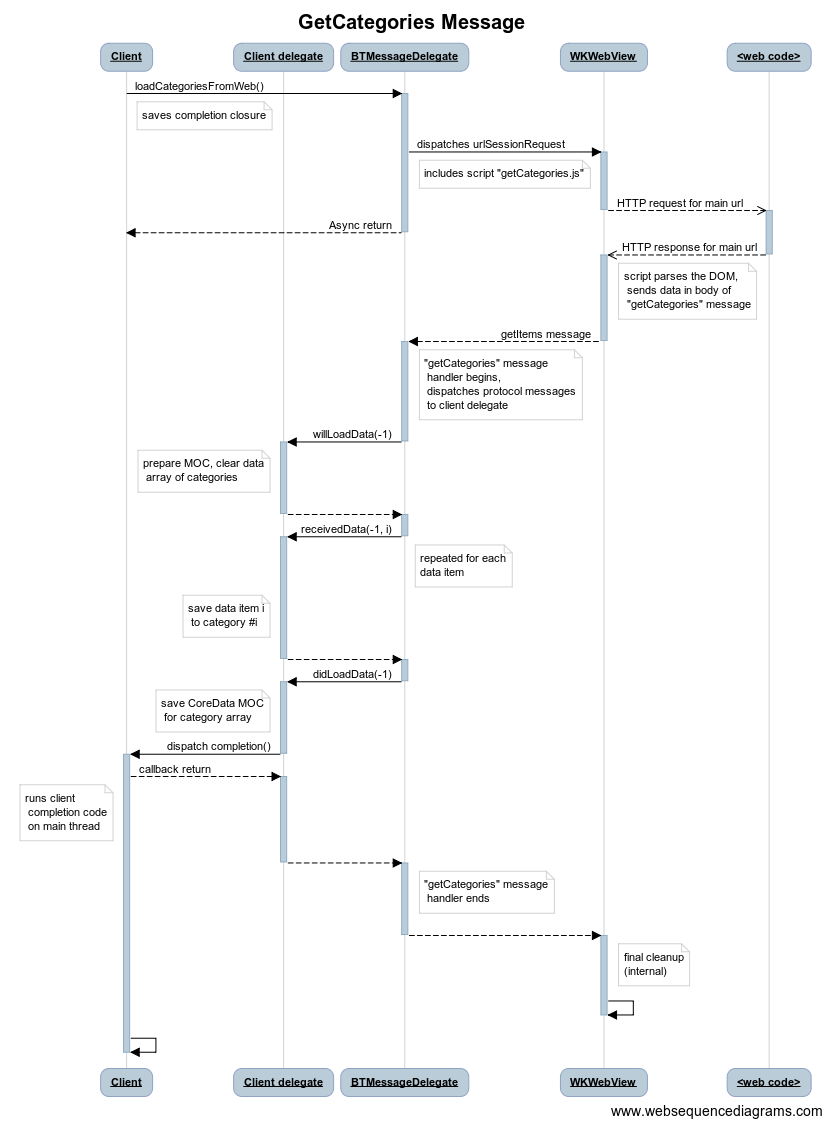
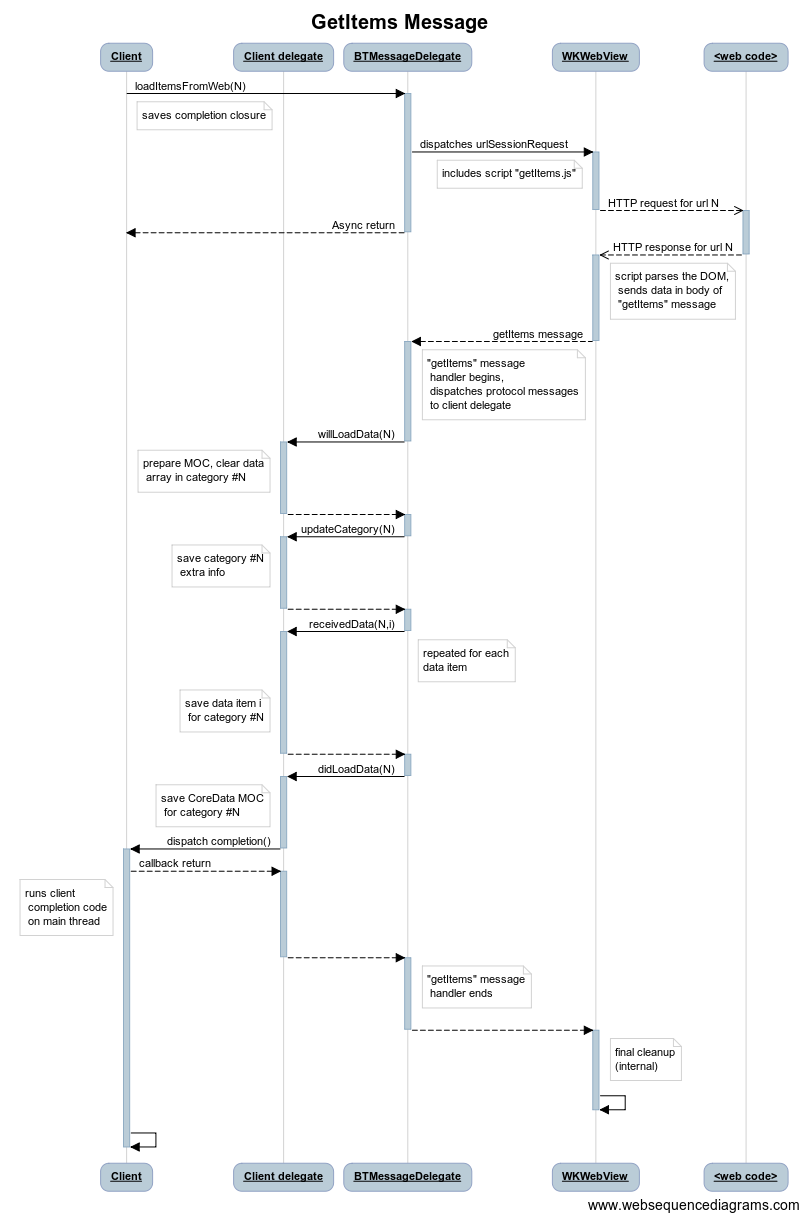
Figure X. Message diagram: getCategories

Figure Y. Message diagram: getItems (all items for single category N)

# Persistence

My data model has quite a few classes, but the main three that deal with the data layers are the Category, DealerItem, and InventoryItem classes. The other classes all have to do with inventory location, and are not essential to the data model used for the Phase One app. DealerItem is the basic Layer 1 data item, and Category is its aggregator (via the dealerItems relationship). InventoryItem implements Layer 2, and Category also aggregates those via inventoryItems.

## Core Data mods for iOS 9

CoreData has evolved significantly in iOS 9, as far as the needs of this program at least. The previous design was that CoreData would generate model classes for you, and you would have to add any extra behavior you wanted via extensions (which made it hard to add storage properties of your own). But the new design assumes that you own the classes and their inheritance hierarchies, including properties, but persistent properties are added by CoreData in an extension file using the extended file name “+CoreDataProperties”. This file is the only code that gets rewritten by CoreData at model generation time, allowing you to keep it segregated from your code. Unfortunately for me, there seems to be a bug in the current implementation such that all persistent properties are made optional, even though you deselect that checkbox in the data model. So I still have to edit the file after regeneration; hopefully, Apple will fix this, although I can make changes by hand. Still, this would be a problem in a product situation.

For example, take the small class AlbumType. In the Old Way, CoreData generated the following file, named AlbumType.swift:

class AlbumType: NSManagedObject {

@NSManaged var code: String

@NSManaged var ordinal: Int16

@NSManaged var descriptionX: String

@NSManaged var families: NSOrderedSet

}

Then I needed to add all the extra behavior in the file AlbumType+Ex.swift, via an extension:

extension AlbumType {

…

}

But once I converted to Swift 2, I needed to use the AlbumType+CoreDataProperties.swift file that was generated by CoreData. This file looks exactly like the original AlbumType.swift, except the class definition line is replaced by another extension line like above.

To make this work, I simply commented out the contents of the original class (all the lines starting with @NSManaged), and then the original class was extended both by CoreData and me. At some point, it would be wise to move the extensions (or part of them anyway) back into the original class design, which is now under my control, but sometimes it is also good to have some behavior in extensions as well, since it helps organize the concepts better.

## Collection Store class

This class is the master model class, as it provides access to the stamp collection data for use by the UI classes (view controllers and views). It implements its persistence features via CoreData, and also implements the protocols which are used by the import/export subsystem. It also provides a comprehensive, extensible interface for sorting and searching the database, creating subsets that go with subcategories, and allowing selection of data subsets selected by particular date ranges and/or keyword searches. The data update feature is called from here as well.

CollectionStore implements a simple management scheme for thread-safe use of CoreData ManagedObjectContext objects. This allows the use of an integer ContextToken in place of passing the actual MOC around everywhere, such as the import/export protocols and classes. Basically, I assign a new MOC for use whenever a data operation is done on a background thread. When a section of code needs to use the persistent model data, it can pass its context token in and the associated MOC is retrieved internally. I keep the object reference around in a dictionary until it is no longer needed, after which point it will be reclaimed by ARC. I also dedicated token 0 for use by the main UI thread, so that the same MOC is returned for use by all UI functions. Refer to use of the private array variable “mocsForThreads” in the section just after the CoreData stack implementation (lines 79-164 of the current version of CollectionStore.swift).

# Data updates

There are many problems to be solved when we have existing data that we want to update from the live web connection. We rely on the basic design of the dealer websites remaining consistent from month to month, but also we rely on data changes being traceable. By this, I mean that if an object of a certain description has its ID string modified for some reason (and the dealer site I use does this regularly), the code needs to be able to match things up. I do this mostly by matching the description data, but realizing that a perfect match isn’t always possible. Therefore, these are always provided as suggestions for the user to review and commit manually (possibly in batches).

Usually, the website data has additions of new items, and when existing items change, it is usually the price or availability that is changing. These are easy to detect and automate, but they are included in the review cycle just to be safe.

The real problem is when the ID or description fields change, and this is what the code is designed to deal with. The user is presented with lists of suggested updates, and is allowed to commit them directly or to make changes. (I wanted to have the opportunity to override the algorithm’s decisions. In many cases this worked fine, but there are problems, which I had planned to code around eventually as I discovered the failure modes.)

Refer to the updateCategory() function of CollectionStore.swift, and all the code in UpdateComparison.swift.

# Sorting and searching

This subsystem is a sophisticated, extensible way of using CoreData filtering and sorting (NSPredicate and NSSortDescriptor and related classes). I had initially implemented ways of selecting years or year ranges of the collection in the web app version, and wanted to continue to allow this. I had also implemented a version of keyword search that ended up being very useful in finding items in the collection that matched some category I hadn’t planned for, such as all the New Year issues, or all the issues referring to a particular topic. I wanted a way to specify these kinds of search criteria (and more) without having to deal with predicates or descriptors all over the code. This turned out to be an ideal task for Swift enums with associated values.

The idea is to use enums in the UI code (View and ViewModel), and then translate them into predicates and sort descriptors for use with the Model. There were various glitches along the way, and my plan to ideally use CoreData for all filtering and searching turned out to be impractical for various reasons including my lack of familiarity with the classes, so I implemented a two-step plan where some of the basic data would be filtered by CoreData, and then the returned arrays of NSManagedObjects would be further filtered by Swift functional calls (filter()). I could mix and match the implementation of this, so some search types were pure CoreData predicates, some were pure in-memory Swift filtering, and some worked best with a hybrid approach.

In addition, there were complications in that some types of filtering would work only for Layer 1 (Info), or only for Layer 2 (Inventory), and some were fine for both types of data. This was all worked out in the implementation files, mostly found in ViewFiltering.swift.

The sorting subsystem was originally designed as a two-level affair, with some sorting done with NSSortDescriptors in CoreData, and some in-memory sorting using Swift functional style (sorted()). The functional code can be found in ViewSorting.swift. However, after some failures to get sort descriptors to work for my needs, I decided to go fully with the Swift in-memory sorting model. The code that makes this decision is in the table view controller in InfoItemsTableViewController.swift, cleverly disguised in the function addSortAction().