My Baseball Collection App

By: Nicolas Parra

nparra@calpoly.edu

Advisor: Sara Bahrami

Computer Science Department
College of Engineering
California Polytechnic University
March 2019
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table of Contents</td>
<td>2</td>
</tr>
<tr>
<td>Abstract</td>
<td>3</td>
</tr>
<tr>
<td>Introduction</td>
<td>3</td>
</tr>
<tr>
<td>Related Works</td>
<td>3</td>
</tr>
<tr>
<td>Audience</td>
<td>4</td>
</tr>
<tr>
<td>High Level Use Cases</td>
<td>4</td>
</tr>
<tr>
<td>Functional Requirements</td>
<td>4</td>
</tr>
<tr>
<td>Nonfunctional Requirements</td>
<td>5</td>
</tr>
<tr>
<td>Constraints</td>
<td>5</td>
</tr>
<tr>
<td>Wireframes</td>
<td>6</td>
</tr>
<tr>
<td><strong>Software Architecture Diagram</strong></td>
<td>8</td>
</tr>
<tr>
<td>Class Diagram</td>
<td>9</td>
</tr>
<tr>
<td><strong>State Diagram</strong></td>
<td>12</td>
</tr>
<tr>
<td>Sequence Diagrams</td>
<td>13</td>
</tr>
<tr>
<td>Database Structure</td>
<td>15</td>
</tr>
<tr>
<td>Design</td>
<td>15</td>
</tr>
<tr>
<td>Security Considerations</td>
<td>16</td>
</tr>
<tr>
<td>Scheduling and Development Cycle</td>
<td>17</td>
</tr>
<tr>
<td>Testing</td>
<td>17</td>
</tr>
<tr>
<td>Stretch Goals and Future Development</td>
<td>18</td>
</tr>
<tr>
<td>References</td>
<td>18</td>
</tr>
</tbody>
</table>
Abstract

My Baseball Collection is an iOS application that aims to simplify the management and expansion of physical baseball trading card collections. The app allows users to digitize their baseball card collection by uploading images of cards they possess, creating a wishlist of cards they are seeking, and viewing the collections and wishlists of other users. This project seeks to provide quality of life improvements to those within the baseball card trading community and to further facilitate communication and trading in an online world.

Introduction

Baseball card collecting and trading is about as old as baseball itself, with roots tracing back to the late 19th century, and while the world around it has undergone major changes since then, the hobby itself has remained essentially unchanged since its conception. Baseball cards originated as methods of advertising baseball equipment, tobacco, and other related products, and quickly attracted a large following of collectors and traders [1]. Each card typically consists of a decorative player, team, and year on the front, with a variety of information on the back, including player biographies, baseball trivia, and advertisements.

Like all physical collections, managing baseball cards can quickly become a tedious task when the size of the collection grows into the hundreds. Although I am not personally a collector, my discussions with hobbyists and the sentiments of online baseball card communities clearly echo the need for a lightweight, portable, and socially-integrated tool for assisting trading and managing baseball card collections. As such, a mobile application presents the perfect solution to this problem by allowing users to easily take pictures of their collection on the go, manage a list of sought-after cards, and easily share their collections and wishlists with other users both in-person and online.

Related Works

This project takes inspiration from other existing products which have sought to address similar problems.

One such product is “Baseball Card Collection” which acts as a database for collectors to enter their collection information [2]. While the core concept heavily aligns with my project’s, the implementation does not adequately address the issues raised previously. Primarily, this application is only compatible with Windows desktop computers and is entirely offline, which prevents collectors from communicating on-the-go both in-person and online. Furthermore, the application does not support card images, and implements an outdated and uninteresting user interface.

Another similar product is “MLB BUNT Baseball Card Trader” which allows users to collect, purchase, and trade digital cards [3]. The app is available on iOS and Android and has an intuitive and interesting user interface. This addresses the need for a portable, social, and online product, but does support physical collections, as all the cards are completely digital with no
real-world analogues. The app also does not allow users to store a wishlist, which makes keeping track of desired cards difficult.

Audience

This app is intended for use by baseball card collectors and traders, and as a supplement to social media and other technologies. It allows hobbyists to keep track of their own collection, while easily browsing the collections of others in the community in order to facilitate discussion and trading. The app can also be used by those looking to build a collection, by allowing them to keep a shareable wishlist of cards they are looking to obtain.

High Level Use Cases

Functional Requirements

- The user can create and login to their unique account
• The user can upload and store saved photos of cards
• The user can use the built-in camera to take and save photos of cards
• The user can view all saved cards and sort by name, year, and team
• The user can search and filter saved cards by player name
• The user can create, edit, and delete sets of cards
• The user can delete cards in each set
• The user can edit card information
• The user can add, edit, and remove card entries from the wishlist
• The user can search and sort existing wishlist entries
• The user can view friends’ card collections and wishlists

Nonfunctional Requirements

• An internet or cellular connection is required
• Database queries should be asynchronous when applicable, allowing the user to do other tasks until the query completes
• Network communications should be able to adapt to poor cellular network conditions
• Database queries should be isolated to allow seamless backend service switching if needed
• The application UI should scale and function properly on all device sizes
• Due to the large amount of image fetching, the application must utilize image compression and caching techniques
• The application should implement account security measures, such as password encryption and secure key storage
• The application must function on iPhone 6 models or newer
• The application must function on iOS version 11 or newer

Constraints

The app is currently heavily reliant on network connections and cannot be run offline, as all data is stored remotely on the database. As such, it can perform poorly and provide a frustrating user experience over a slow network. In the future, this problem could be remediated by allowing the user to opt-in to storing images and metadata locally, so that network connections would only be required when data is changed, or a friend’s data is requested.

Account security also faces some constraints, as there is currently no method for a user to recover a lost password. A password recovery system would likely require some web or email client, which was slightly outside the scope of this project. Nonetheless like the previous constraint, this could be addressed moving forward as the project expands.
Wireframes

My Baseball Collection

Sign in

Email

Password

Sign in

Forgot password?

Sign-in with Gmail

Sign-in with Facebook

Kirk Gibson

Front Preview:

Back Preview:

Enter Name

Enter Team

Enter Year

Enter Additional Information (Optional)

Add to Set Dodgers 1989

Add a Card

Name

Year

Add Date

Kirk Gibson

Team: Los Angeles Dodgers

Year: 1989

Additional Information: Unopened, Willing to sell
Class Diagram

A subsection of the class diagram depicting relationships for account-related classes

Other classes (No Relationships shown)

Models - Data storage with minimum/no logic

**CachedCardImage**
- `+ id: String`
- `+ imgF: Data`
- `+ imgB: Data`
- `+ BNll: Bool`

**Friend**
- `+ UID: Uint`
- `+ Login: String`
- `+ FullName: String`
- `+ Status: FriendStatus`

**CachedCardImageList**
- `+ cards: [CachedCardImage]`
- `+ cap: Int`
- `+ iter: Int`

- `+ getByld(): CachedCardImage`
- `+ getByIndex(): CachedCardImage`
- `+ setImgF(): Bool`
- `+ setImgB(): Bool`
- `+ clearImgB(): Bool`
- `+ addCard()`

**Set**
- `+ Name: String`
- `+ CreationDate: Date`
- `+ ModifyDate: Date`

**Card**
- `+ Name: String`
- `+ Team: String`
- `+ Year: String`
- `+ AdditionalInfo: String`
- `+ ImageFront: Ullmage`
- `+ ImageBack: Ullmage`
- `+ BNll: Bool`
- `+ CreationDate: Date`
- `+ ModifyDate: Date`

**WishlistEntry**
- `+ Name: String`
- `+ Team: String`
- `+ Year: String`
- `+ AdditionalInfo: String`
- `+ Image: Ullmage`
- `+ BNll: Bool`
- `+ CreationDate: Date`
- `+ ModifyDate: Date`
Views - UI logic and reactive binding only, no business logic

CardViewController
- displayedImage: UIImageView
- nameLabel: UILabel
- teamLabel: UILabel
- yearLabel: UILabel
- infoView: UITextView
- flipButton: UIButton
- editButton: UIButton
- mv: CardMV

CollectionViewController
- collectionView: UICollectionView
- addSetButton: UIButton
- searchBar: UISearchBar
- sortButton: UIButton
- friendsButton: UIButton
- mv: CollectionMV
  + collectionView()
  + filterContentForSearchText()
  + sortBy()
  + dataDidChange()

SetViewController
- setV: UICollectionView
- editNameButton: UIButton
- addCardButton: UIButton
- searchBar: UISearchBar
- sortButton: UIButton
- mv: SetMV
  + collectionView()
  + sortBy()
  + dataDidChange()
  + setCellImage()

WishlistEntryViewController
- entryImage: UIImageView
- nameLabel: UILabel
- teamLabel: UILabel
- yearLabel: UILabel
- infoView: UITextView
- editButton: UIButton
- mv: WishlistMV
  + updateImageUI()

WishlistViewController
- wishlistV: UITableView
- searchBar: UISearchBar
- sortButton: UIButton
- friendsButton: UIButton
- addButton: UIButton
- mv: WishlistMV
  + tableView()
  + swipeCellActivated()
  + filterContextForSearchText()
  + sortBy()
  + dataDidChange()

ModelViews - Execution logic, manipulates data from Models to change Views

CardMV
- name: String
- team: String
- year: String
- info: String
- imgFront: Data
- imgBack: Data
- set: Int
- bNil: Bool
  + executeChangeFrontImage()
  + executeChangeBackImage()
  + executeChangeSet()
  + executeConfirm()
  + executeFlipCard()
  + executeDeleteCard()

CollectionViewMV
  + executeAddSet()
  + executeDeleteSet()
  + executeViewFriends()
  + executeSort()

LoginMV
- usernameString: String
- passwordString: String
- createUsernameString: String
- createPasswordString: String
  + executeLogin()
  + executeCreateAccount()

WishlistEntryMV
- name: String
- team: String
- year: Int
- info: String
- img: Data
- bNil: Bool
  + executeChangeImage()
  + executeConfirm()
  + executeDeleteEntry()
  + executeViewFriends()
  + executeSort()

SetMV
  + executeEditName()
### Supporting Classes - Database and Business Logic

Only primary functions shown for brevity

<table>
<thead>
<tr>
<th>QBWishlistInterface</th>
<th>QBCardInterface</th>
<th>QBSetInterface</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ instance: QBWishlistInterface</td>
<td>+ instance: QBCardInterface</td>
<td>+ instance: QBSetInterface</td>
</tr>
<tr>
<td>- currentUserEntries: [QBCCOCustomObject]</td>
<td>- currentCards: [QBCCOCustomObject]</td>
<td>- currentUserSets: [QBCCOCustomObject]</td>
</tr>
<tr>
<td>- selectedEntry: Int</td>
<td>- cachedCardImages: CachedCardImageList</td>
<td>- selectedSet: Int</td>
</tr>
<tr>
<td>+ getUserWishlist()</td>
<td>+ getCurrentSetCards(): String</td>
<td>+ getUserSets()</td>
</tr>
<tr>
<td>+ getSelectedEntryInfo(): WishlistEntry</td>
<td>+ getSelectedCardInfo(): Card</td>
<td>+ getSelectedSetInfo()</td>
</tr>
<tr>
<td>+ editSelectedEntry()</td>
<td>+ editCard()</td>
<td>+ addSet()</td>
</tr>
<tr>
<td>+ deleteCurrentEntry()</td>
<td>+ addCard()</td>
<td>+ renameSet()</td>
</tr>
<tr>
<td>+ deleteCurrentCard()</td>
<td>+ deleteCurrentSet()</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Business</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>+ instance: Business</td>
<td></td>
</tr>
</tbody>
</table>
Sequence Diagrams

Logging In

User

LoginView Controller

LoginMV

QBUUser Interface

Select 'Log In' - User

Login View Controller

executeLogin() - LoginMV

callLogin() - QBUUser Interface

getFriendsList() - QBUUser Interface

Segue to Homepage

Home Page is Displayed

Viewing Friends' Accounts

User

Friendslist Controller

FriendslistMV

QBUUser Interface

Select Friends

Friends List is Populated

Select a Friend

Friend Collection is Displayed

showFriendsList() - FriendslistMV

getFriendsList() - QBUUser Interface

Return Friends

tableView(index) - FriendslistMV

getUserSets() - QBUUser Interface

Return Sets

Return Sets
Database Structure

This project is heavily dependent on a QuickBlox cloud database to store account, collection, wishlist, and friend information. QuickBlox offers a non-relational backend database that stores records of custom objects and accounts. This application makes use of five tables: Card, Set, WishlistEntry, ProfilePicture, and Friend. Each table consists of all user-generated objects of that type, with each object containing the specific data. For example, the Card table consists of objects with fields: ObjectId, UserId, SetId, Name, Team, Year, AdditionalInfo, FrontImage, and BackImage. The application also implements the built-in QuickBlox account API which tracks usernames, passwords, and unique IDs for each user.

The database is queried via network connection every time user-generated content needs to be fetched, created, or updated. To cut down on excessive and slow network communications, an image caching system is implemented for card images, wishlist images, and user profile images. The caching system stores temporary local copies of each image after it is downloaded, so that it can be fetched immediately on any subsequent use. After storing 64 images in the cache, future downloads will replace the oldest to prevent excessive device memory use. This system offers the best tradeoff between performance and memory usage and allows the cache size to be easily adjusted according to user feedback.

The application also uses image compression to reduce file sizes, thereby further reducing network communication times. iPhone camera images tend to be 1 to 3 megabytes, which can result in long download times when fetching dozens of images simultaneously. Furthermore, wireless connections are notoriously unstable, which may further negatively impact performance. Taking these constraints into account, the application applies a 50% compression to all uploaded images. While this results in a loss of image quality, baseball cards tend to have large illustrations and text sizes, which limit the negative impact of compression.

Design

This project closely follows a reactive programming style, which allows data changes to dynamically propagate through the code by updating results whenever operands change. Reactive programming can be closely compared with the Observer design pattern, but offers more fine-grained data control, as well as less code in implementation. The reactive style works best with interactive user interfaces, which makes it a perfect fit for this project’s requirements. For example, when creating a new card, the name, team, and year fields are required before the card can be saved. As the user types in each field, values are automatically updated and the ‘confirm’ button is enabled or disabled as requirements are met. To implement this style, the project makes heavy use of a framework called ‘Bond’, which allows interactive binding of UI elements to backend logic variables.

At a higher level, the project implements the Model-View-ViewModel (MVVM) software architecture pattern. This pattern decouples the UI, Logic, and Data portions of the application and is closely related to the Model-View-Controller (MVC) software architecture pattern. MVVM places the most emphasis on ViewModel classes, which are responsible for the application’s internal computational logic as well as much of the UI logic. View classes create
UI bindings with the ViewModels, and any data that the program needs is stored in Model classes. This architecture works extremely well with the reactive pattern discussed earlier, as each View class needs only bind its various UI components to a ViewModel class. Then transitions between Views, database queries, and data manipulation can all be handled by the various ViewModel classes. While the architecture ideally fully decouples UI logic from the View classes, its implementation often results in too many responsibilities for the ViewModel classes. Thus, my implementation takes a more balanced approach by assigning certain UI logic such as screen transitions to the ViewModels, while other simpler UI logic is handled by Views.

Security Considerations

Software security is a topic that I find very interesting and important to consider, especially when developing publicly available software. Security is often overlooked during the software design life cycle, and is in many cases added towards the end of development, or even after launch. Security at this stage is prone to poor implementation, or requires the rewriting of large chunks of code. To prevent this, security concerns were mapped and considered during every stage of development. The largest attack surface for this project is the login process, where sensitive user passwords and information is processed. Most importantly, improper handling of login credentials for a user that logs in with Facebook or Google could compromise the security of their linked account. Furthermore, the application needs to generate a hidden username and password for accounts created in this way in order to provide compatibility with the QuickBlox API. Thus, the project uses a framework called SwiftyRSA, which implements the RSA encryption scheme. A 2048 bit key is securely stored in the app and is used to deterministically encrypt and sign user credentials. This result is then used as a hidden password when creating a linked Facebook or Google account and can be correctly generated on any subsequent login.

Other possible attacks include injecting malicious code into user input fields, such as in a card’s description. The iOS operating system and Swift language handles the majority of input sanitization automatically, but the application implements additional checks such as length limits and type checking before input is uploaded to the database. Furthermore, the QuickBlox API implements a secure OAuth protocol which stores a local key to protect any network communications and stored data.

Because both of these methods require the use of keys, secure key storage needs to be considered as well. Secure key storage can be accomplished in one of two ways: by storing keys on some central server, or by storing keys locally. For the purposes of this project, local key storage offers easier implementation and avoids further dependency on an external server, making it the better option. Thus, the app uses a framework called Cocoapods-Keys to securely store obfuscated copies of each key. This makes it difficult for an attacker to download and disassemble the app in order to steal the encryption keys.

Scheduling and Development Cycle

Documentation Phase: September - December 2018
- Project Outline
- Project Requirements
● Use Cases
● Wireframing
● Diagrams

Implementation Phase: January - March 2019
● Code implementation
● Code testing
● Documentation Revision
● Agile Sprint Cycles
  ○ Sprint 1:
    ■ Database structure
    ■ Database fetching
  ○ Sprint 2:
    ■ Core collection and wishlist functionality
  ○ Sprint 3:
    ■ User accounts, Facebook, and Google integration
    ■ Friends list
    ■ Searching and sorting
  ○ Sprint 4:
    ■ UI
    ■ Device compatibility
    ■ Usage testing
  ○ Sprint 5:
    ■ Documentation finalization
    ■ Final Report
    ■ App Store publication

Testing

Project testing was carried out using the built-in Xcode testing suite, and was split into unit testing and integration testing. Each unit test executed a specific function or method call and verified its result. Integration testing built on the unit tests by verifying the testing interaction between various project modules. An example integration test is as follows:
This test cases logs in with the testcases account, adds a new set, renames the set, then deletes the set. After each step, a unit test is run with the XCTestCase call to verify that each individual operation completed correctly. Due to callback-style of the database API calls, a four second delay is inserted between each step to ensure that network communications have finished.

Using the built-in Xcode bots, integration testing occurred automatically once every 24 hours to ensure that new code additions and revisions didn’t break previous code. Each integration test measured the number of tests passed, total code coverage, and a measurement of which code had changed since the last test. Otherwise, UI and secondary functionality was tested manually, as it was implemented.

Stretch Goals and Future Development

I plan to continue development on this project post-launch on the App Store. Some future goals include implementing in-app messaging between users, the ability to view other users’ public collections without needing to be on their friends list, and further UI design and interactivity. Furthermore, I may eventually create a web component of the app which would allow synchronization between the app and web client, as well as easier sharing. Lastly, I do plan on porting the app onto the Android operating system and allowing cross-platform interaction between users on each system.

References
