KeyLime

A Senior Project Report

Presented to

John Bellardo

Faculty of California Polytechnic State University

In Partial Fulfillment

of the Requirements for the Degree

Bachelor of Science in Computer Engineering

By

Josh Magera

on Project in Group with Eli Parker and Matt Orgill

June 2019

Abstract

New freshmen arrive at Cal Poly every year, experience Week of Welcome, and, if they haven't been to Firestone Grill within the first week, they can consider themselves an anomaly. But how long until those freshmen find the amazing sandwiches and breakfast burritos served at Gus's Grocery or hear about the free burger promo at Sylvester's? The goal of this senior project was to create an app, KeyLime, that makes it easy for college students to find new eateries and fresh deals that are local, affordable, and tasty. KeyLime aims to target college students and create a space for restaurants both new and old to be discovered. The goal of bridging the gap in communication between college students and restaurants is accomplished by allowing easy access to current available deals, and providing simple restaurant reviews from a similarly aged audience. The UI is split into tabs based on specific functionality, and users have means of interacting by declaring favorite restaurants and deals that they have used. The Discover tab is focused on finding new and trending restaurants by applying multiple filters. The Deals tab allows the user to find deals for favorite restaurants and those with approaching expiration dates. The Search tab allows users to search for specific restaurants or types of restaurant in the area. Finally, the Settings tab allows the user to manage and view account information while also allowing admin user access to their restaurant's deals. These main areas of functionality describe at a high level the app that will be expounded upon in this report in both design decisions required and implementation difficulties encountered.

Introduction

As most of our team was inexperienced with iOS development from the start, a good deal of research was required both before starting the project at all. Throughout the first quarter of the project, I was taking the Mobile Application Development class in iOS at Cal Poly, and my groupmates had taken the same class in Kotlin the quarter before. So, the learning curve was steep from the beginning, and we split preliminary research up by breaking down early design decisions into database choice, restaurant data sources and high level competition scouting to gather good ideas for app feel and functionality.

The design decisions that we anticipated facing before starting work on the project at all were rather large. We had little idea of what kind of UI and high level app design would be effective in connecting with college students, so Matt was assigned to research competition, flow, and feel of the app. At the same time, we needed a way to store our own data describing restaurants and deals so we could create a unique experience that didn't simply pull directly from an already established source. Eli researched databases that would be the most useful for this purpose, considering cost, integration with iOS, and speed among other things. The last major design decision we anticipated facing right from the start was finding means of procuring some existing data about restaurants, their locations, names, and any information we could get about food type

and pricing. I researched this area, much like Eli considering cost, integration difficulty, and amount and quality of information available. These areas of research and the design decisions that were made final will be expounded upon below.

As a group of three, splitting responsibilities, specific task assignment, and source control were extremely important. At a high level, Eli was assigned to managing users and account information, creating users, storing them in the database, and designing users' interaction with the data. Matt took the responsibility of designing UI and navigation for the flow and experience of the app as well as implementing filtering functionality that would allow the users to best connect with the data. Finally, I oversaw restaurant and deal data formatting, organization, and storage while also managing admin accounts for restaurant owners and creating functionality to allow adding, viewing, and changing deals for their restaurants. Assignment of specific tasks beyond these broad areas of responsibility were given on the basis of workload and availability of critical areas of the source code, since some areas that needed work were in the same file and required team coordination for completion. In terms of source control, we used Git through GitHub, creating our own branches off the master branch for each individual task. We communicated between each other what areas of the code that we needed to work on to make sure the same files were not touched by the others.

Requirements

Though the requirements for KeyLime were not imposed upon our senior project group by an outside source, there was a set amount of functionality that we wanted to implement as a rough, early-stage, proof-of-concept app. These requirements were set before the project was begun and were realized throughout the winter and spring quarter project as outlined below.

The full app design begins with a logo designed by Eli and presents the opportunity to login as an existing user, login as an admin account (protected by a passcode only supplied by us), or create a new account. Facebook login is included in this part of the app to allow easy account creation and login. The general format for the rest of the app experience followed a tabbed design.

The Discover tab contains a table view that presents trending and filtered restaurants, which can be viewed in more detail in a separate view and favorited in the table view or that restaurant view. The detailed restaurant view allows the viewing of deals for that specific restaurant in a horizontal slider (collection view) and accessing those in more detail in the deal view. The filter for the discover view allows filtering by price, rating, distance, food type, and favorited status.

The Deals tab contains a horizontal slider for deals of the user's favorite restaurants along with a vertically scrolling table view for discovering new deals that are expiring

soon. Choosing any of these deals allows the user to view the more detailed deal view and indicate whether they have used the deal or not. This indication is meant for restaurants that want a deal to be only used once, in which case they would require a user to show that the deal is unused on the app and then select it as used. Once marked as used, a deal cannot be changed back to unused status for the guarantee to the restaurant that an unmarked deal has indeed not been used.

The Search tab allows a user to enter a restaurant name or food type into an auto-predicting search bar and find what they search for on a map. This functionality uses basic Apple MapKit search features, but allows the user to select a restaurant and view a callout for the restaurant with the data in our database including rating, price, distance, and image. The individual restaurant callout also contains a button that allows the user to jump to the detailed restaurant view.

The Settings tab allows users to view basic information of their account. This includes their user ids along with restaurants that they have rated and restaurants that they have favorited. Selecting one of these also allows the user to jump to the detailed restaurant view. Logout functionality is also provided here and admin users are able to view a button that takes them to a view of their restaurant including a collection of their deals with functionality for adding, modifying and deleting these deals.

Background

As mentioned above, I took the Mobile Application Development class in iOS at Cal Poly with Professor Randy Scovil. Most of my background experience with similar projects and technologies came from that class. These areas of experience included working with Apple's UIKit and MapKit, Firebase, and Firestore.

Working with UIKit consisted of a large part of designing the KeyLime app, since every view uses features as simple as UILabels up to UITableViews and UIPickerViews. Design with these basic iOS items was very similar to many of the labs in CSC 436, which were meant to give an overview of many different tools available within the iOS framework like table views, collection views, buttons, text fields, and more. These labs also provided the basis of my knowledge on using MapKit by building search queries and manipulating map annotations at a simple level, though much more research was required during the project to customize annotation callout views create predictive text suggestions among other things. Some experience with Firebase and Firestore was gained in these labs also, providing tools for creating classes that were easily ported to and from dictionaries in a format that integrated with the database in a clean and efficient way.

My final project for Mobile App Development was very similar in a lot of ways and was the main source that I could draw on for much of the implementation for this app. This

final project was a rock climbing iOS app called BetaShark that allowed users to find gyms, climbing routes within the gyms, and techniques for climbing those routes. BetaShark's data was designed to be completely crowd-sourced, so users could add gyms, routes, and betas (techniques) on their own. This functionality was similar to the ability that KeyLime required to allow admin accounts to add, delete, and modify deals in their restaurant. Since the theoretical production-level data of BetaShark would be too large and ever-changing (routes change constantly in gyms) to manage from an admin perspective, the crowd-sourced data of BetaShark was controlled by the users themselves. This was done by allowing users to up-vote and down-vote gyms, routes, and betas, each of which would be deleted if the down-votes became greater than the up-votes. This functionality is similar to the ability to favorite restaurants and mark deals as used in KeyLime, and the use of Firestore to track users' interactions with the data was common between the two. Finally, the use of table views to display large amounts of similarly structured data is common to both KeyLime and BetaShark. Included in BetaShark was the ability to search and filter gyms, routes, and betas in table views in a similar way to the vision for restaurant discovery in KeyLime. However, KeyLime's filtering aspirations were definitely more ambitious than the very simple sort by name or difficulty provided in BetaShark.

Though my final project for Mobile App Development had many overlapping areas of functionality with KeyLime, the implementation was not necessarily the same in both projects. First of all, I didn't implement all of these features in KeyLime because of the group nature of the project. So, I was able to offer advice and help when the parts that overlapped with my experience were being worked on, and sometimes a similar approach was used. But many times a different approach was used for more efficient design. The features that I did implement did not necessarily follow directly the design that I used in BetaShark either because I got to discuss good design with my groupmates to improve on the design. Areas where the two projects did not overlap were plentiful as well and included Facebook login, individual detail views, an emphasis on intuitive UI design, MapKit specifics of customized callouts and linking with Firestore data, and more.

Individual Contributions and Roadblocks

Since this app was a group effort and Matt's idea originally, it is important to recognize what work I contributed to the project specifically so I don't take credit for more than is my due. My focus on the project was originally intended to be on restaurant and deal data along with management of admin account functionality. However, by the end of the project, we as a group, while taking a definite greater ownership of our specific responsibilities, collaborated on much of the work and didn't hold tight boundaries against working in each other's area of responsibility. In addition, there were a number of difficulties that I had in working on the app that contributed to the final product in solving them.

As I had been in charge of researching sources of data for restaurants, I developed a test app early on that became useful in adding restaurants to our database. This separate app design simply contains a MapView and a search field with predictive text suggestions (material that was easily carried over into the Search tab of the main app). This was used to search for restaurants and add them to our database with many useful values from the MapItem type returned from a simple query along with base values for the restaurant type that we thought would be useful for later implementation. This app had to be updated later on when it was necessary to add fields to our restaurant class, but, otherwise, it retained its utility.

At a higher design level, I had a hand in designing the fields for the main classes necessary for organization of the data. The main classes that had to be created included restaurant, deal, and user classes. As the project moved ahead, the need was realized for fields that were not anticipated before, so these classes had to be updated and maintained. An example of this maintenance included adding a restaurant rating array to keep track of the rating that a user had given a specific restaurant so that loading the detailed restaurant view would recall the rating previously given.

Another area of my involvement was being solely in charge of the Search tab. I dealt with MapKit and location management, allowing the user to search different restaurants to see what is nearby. Selecting a certain annotation on the MapView pulls up a custom callout view that displays the restaurant information from our database that corresponds with the Apple Map search item for that restaurant. One decision that I had to make here was how to manage restaurants that were not in our database. I ended up pulling up the callout view with empty restaurant information and signifying that the restaurant had not partnered with our app and therefore was unavailable.

A roadblock that I encountered in creating the Search tab was trying many different ways of customizing the annotation callout views, because there were a number of options documented online, but none of them showed any visual result when I tried to implement them. Only an empty callout view was displayed until I found that the size of the view was being reset to zero, so that creating constraints on the height and width of the view caused it to finally work. A different bug that I encountered later on occurred when marking a restaurant as a favorite after opening the detailed restaurant view from the Search tab. In returning to the Search tab and then reopening the restaurant detail view, the restaurant would no longer be marked as a favorite. I found that this was because, though the database was being updated with the user's interaction, the local copy of the user data was not being updated and, therefore, was passing inconsistent information to the restaurant view. Despite being a confusing issue, this turned out to be a quick fix by simply making sure that the local user was updated at the same time as the database.

Another area of work that I had to research and nail down early on was in managing source control for the storyboards. Git tracks the changes to storyboard files, but since

they are basically big XML files, changes by multiple people to one storyboard cause conflicts that can only be resolved in a binary way by taking the whole of one of the files. In other words, merging even slight changes to different areas of a storyboard was impossible with Git. I found that options for fixing this problem included not using storyboards at all, or splitting into multiple separate storyboards and never touching the same storyboard at the same time as another group member. I opted for the latter option, since I thought that storyboards were a useful tool. So, I refactored our single storyboard into multiple storyboards for each tab and for the most important and highly modified single views like the restaurant detail view.

In the Settings tab, I worked on creating an admin view that was only accessible by admin users. I did this by creating a button that only appears if the current user account is flagged as an admin account. The view accessible by pressing that button shows the user their restaurant information and allows them to change the image linked with the restaurant. The Admin View also allows the admin user to view the deals they have added to their restaurant and modify, add to, or delete them at will. These deals are viewable in a side-scrolling collection view. Also within the Settings tab, I implemented a popup table view that displays the current user's favorited or rated restaurants in a simple and easily accessible way. One difficulty that I did have with this was in creating the view with a transparent background so that it looked like a popup. To do this, I had to segue to the list view modally, but this caused loss of the navigation controller context of the previous view. When viewing the detailed restaurant view resulting from choosing one of the restaurants in the list, the user was not able to return to the list view or the Settings tab at all. After trying different approaches, I was able to start a new navigation controller before the list view and set the Clears Graphics Content setting to false on the navigation controller and the new view to allow the transparent background.

The Deals tab was another area where I had most of the design initiative, being that deals data fell under my responsibility. I created the framework for searching deals, which basically consists of a horizontally scrolling collection view near the top of the screen to display deals posted by favorite restaurants and a vertically scrolling table view that displays new deals that are nearest to expiring. While Matt did the actual filtering of the deals themselves, I created the layout and base functionality needed to pull the deals from the database and display them as needed.

Finally, I added functionality for displaying the images linked with restaurants and deals through the whole of the app. This included making sure images were loaded in the Discover tab table view, the restaurant detail view, the Deal tab table view and collection view, the deal detail view, and all other places where deal collection views were viewable. The functionality for loading the images stayed mostly the same, but implementing it seamlessly into all the different contexts in which it was needed took a while.

These were the main areas of the app that I worked on, though there were other smaller tasks and bugs that needed addressing along the way that diverted time and effort away from the major tasks. It is important to note that there were many important design decisions that had to be made to accomplish these tasks effectively which have not been addressed here. A more in-depth look at these design decisions and the implementation stemming from the is provided below.

Key Design Decisions

The design decisions that went into creating this app were focused on both high-level technology choices and implementation-level programming choices. It is important to note that our group had already chosen iOS from the start as our target platform, since it is prevalent in the app market, I was anticipating gaining some experience with it, and Matt and Eli wanted to learn. Along with this choice came the decision to use Xcode as a development environment because of its design for iOS development. We also chose Git for source control, since our group had the most experience with it and Xcode had a built in tool at our disposal. Other choices of technologies used to create the app were mainly driven by early research, which was split into the categories of database, user interface design and competition, and restaurant data source. Each of the team members took one of these areas of research and came up with the best decision based on the circumstances in which we were creating our app.

After some research by Eli, Firestore was chosen for data storage and organization, while Firebase Authentication and Storage would be used for user management and image storage respectively. This choice stemmed partly from prior experience with Firebase from the iOS class I was taking and partly from the technology's easy integration with iOS that seemed perfect for our needs over another technology like SQLite. Also, the NoSQL structure of the database was perfect for our needs, since we mainly just wanted storage for collections of restaurants, deals, and users and didn't need the structure of a relational database. Matt researched apps with functionality similar to the budding KeyLime and drew up some target designs for the UI while brainstorming ideas for user experience choices we had to make. One result of his research and some group brainstorming was the design for the Deals tab including a side scrolling section for deals by favorite restaurants in tandem with the general vertically scrolling section for other new deals. Another result was the choice to implement ratings as short string descriptions in order to diverge from Yelp's design and make ratings simple, easy and intuitive. My research resulted in choosing Apple Maps to provide restaurant data, since most online resources maintained that Apple Maps and Google Maps would be comparable in utility. Google required an external SDK and pricing for large amounts of queries, while Apple Maps is free, already integrated into the iOS development environment of Xcode that we used, well documented, and provided easy means of customizing popup views for restaurants in the search view. I also looked into alternatives like Foursquare and Yelp API, but the prices and

information limitations set by companies with similar products were not advantageous in the early stages of development.

Other design choices involved actual programming decisions instead of overarching choice of tools for the whole project. These decisions are mostly in the interest of balancing efficiency in time, computation, speed, and flow of the UI, especially when it was sometimes clear that one or more had to be sacrificed in favor of another. There were many of these decisions necessary to build the app as a whole, but the focus here is on the design decisions that I had a heavy or sole role in making.

One of the main programming decisions took place in designing the classes used for organizing restaurant and deal information. Using data structures to organize data, represent it in a compact and easily accessible form, and facilitate the transfer of data through the program is standard, but I was able to use some of Swift's features to enhance the ease and effectiveness of the structures.

First, I created a dictionary computed property in both the restaurant and deal classes that ports the data fields of the class to a dictionary with fields labeled the same way as our database fields for that data type. This made pushing data to the database quick and easy because the Firestore update API call takes a dictionary as a parameter. Instead of manipulating the data in place for update calls, the conversion to a compatible type was built into the class. This kind of operation was used frequently for updating user interactions with the data (favorite restaurants, used deals, and admin manipulation of deals) to the database and was therefore important to handle in a convenient and clean way.

Another design feature in the classes was a convenience initializer that supplemented the designated initializer for both the restaurant and deal classes. This initializer took a dictionary as a parameter and mapped the expected values from the fields named in the database to the designated initializer to create a new instance of the class. This made pulling information from Firebase extremely simple because the result of a database query is always a document object that contains a Data attribute in the form of a dictionary. Porting the information in these documents to class instances was an extremely frequent task to be performed, and direct initialization with a convenience initializer made the code clean and efficient.

Also in the class design, I implemented a computed property for the rating string and price string displayed for restaurants. Ratings were implemented as double values (0-5) and prices were implemented as integers so that the math would be simple to compute in updating these values. However, the display of the both to the users took the form of short, descriptive strings per our high level design plan. The display of these specific strings was first implemented in multiple places in the code, but I consolidated them into the restaurant class itself by creating a computed property that output a different string based on the value of the restaurant's rating or price value. This meant that the strings

themselves only appeared once in the code in order to achieve easy access straight from the restaurant class itself and simple manipulation in the future if Matt's research showed better popularity for different string representations.

Another example of a programming design decision was in the loading of images, which was my responsibility throughout the entire app. One decision that had to be made was changing the image field in the restaurant and deal class to a string, since the images were stored in Firebase Storage with our only point of access being a URL request. Also, since loading an image from an API call is computationally expensive and time intensive, keeping the amount of calls to a minimum is of the highest priority after making sure that the user has a fluid experience. Both of these needs were met by using target optional variables to preserve an image locally once it was downloaded the first time. This occurs in the table views in both the Discover tab and the Deals tab by creating an optional value within the cells to store the image once loaded. This allows the reuse of already loaded cells to preserve the image data when a cell is scrolled back into view. A similar technique is used in the detail views for both restaurants and deals. These views provide an optional image variable to which the selected cell can pass its image data on. The detail view then reuses the image data instead of reloading the image if the target optional was non-nil, but reloads the image if the cell isn't able to pass the image data for any reason. These were ways of cutting down on delay for the user in waiting for images to load especially since Firebase queries kick off an asynchronous thread that takes a while to execute. At the same time, these techniques improved the efficiency of each query and cut down on computational and even financial cost since Firebase charges per query past a certain threshold.

Future Work

The goal of this project was to make a prototype app for a working business idea and set it in motion to some small degree. However, we did not quite reach the level of implementation that would allow a full release of the app and kickoff of a small business. Below are some of the areas that need future work for a full-on implementation and wide release of KeyLime.

The first and most noticeable need for future development is the UI design. If the app were to be released on the App Store, the goal of marketing to college students would depend heavily on the intuitive design, flow, and feel of the app, since younger audiences tend to have high value on and specific tastes for these things. Since this project took place over a relatively short amount of time, our group's main goal was to produce functional software, ignoring at the start the look and feel of the app. Toward the end of the project, we were able to clean up the UI to some degree, but it would still need a good amount of work to push to release level. Also, our team consisted of computer engineering students only, so our lack of familiarity in designing effective user

experiences played a role in this as well. Enlisting the help of graphic design members could be a valuable next step.

Another area for future work in line with preparing for wide use of KeyLime is preparing the software for a large user load. The app has up to this point only been tested among our group with at most three or four running instances at any one time. This testing ensured functionality at a base level, but, if the app were to be released, we would need to ensure reliability for a large user base. Some considerations that would need to be taken are large amounts of database queries and their cost, and coordinating similar actions by multiple users. The first would need to be considered heavily because limiting the number of queries from the app was not taken into account too much past matters of computational efficiency. Large amounts of the restaurant dataset are still queried many times (never unnecessarily), so it would be necessary to do some research on how best to limit those queries to ask for less data unless prompted by the user for more data. The second consideration is in the interest of data consistency for user interaction with restaurants. If ten users updated their rating for the same restaurant all at the same time, the consistency of the restaurant rating with their user data is uncertain because of lack of proper testing with these kinds of loads. This type of testing, along with more intensive testing in pretty much all areas of the software would be beneficial before creating a release version.

Reflections

In looking back on the project, the changes that I would make exist mainly in wasted time due to incomplete planning early on, encountering bugs, and lack of knowledge about the chosen technologies before starting implementation. It is very possible that many of the things that fall into these areas of improvement were unforeseeable and unavoidable by reasonable means given the circumstances of the project and lack of prior experience with certain requirements of the project. However, some were avoidable had a better approach been taken.

One way that a lot of time was wasted and difficulties were encountered was through Git issues. If we had known from the start the issues Git has with source control for storyboard files, I think that we would have considered other options more carefully or at least started out implementing separate storyboards in a well-planned way. Instead, it took a while to blunder through a number of horrific merge conflicts until we actually realized the issue and fixed it. In the same vein, Git gave us a lot of issues with the pod files used to import Firebase and Facebook Login. It once again took us a good amount of time manually reinstalling pods every time we pulled from Git before being able to coordinate a single install with all the pod files to be pushed to Git.

Another issue that could have been avoided was in inadequate planning of our class fields and data organization at the very start of the project. The design of the restaurant,

deal, and user classes changed four or five times during development, creating a domino effect through the code of changes that had to be made to account for added or removed fields. We had tried to plan out the exact fields needed in each class from the beginning of the project, but found later that we didn't think through the entire app design. Things like tracking the specific ratings that the users had given every restaurant that they rated were overlooked at the start. In this example, an array field had to be added and many initializer calls for the restaurant class throughout many parts of the code had to be updated to account for the new field. Since these changes stretched through most of the code, the whole group had to consolidate everything being worked on at the time into one Git branch, and then a single person had to make the changes and push them so all team members would have them without creating merge conflicts later on. This was a process that had to happen multiple times, which slowed down the whole group a bit. The issue could have been avoided by planning out very specifically all features that were going to be included in the app from the start and building the class structures from that information.

Overall, despite the parts of the development process that didn't go as smoothly as we hoped, the majority of the actual implementation was thought through before implementing and didn't lead to regret of the design decisions we made.

Conclusion

KeyLime is an app for convenience and discovery in its beginning stages. The goal of this senior project was to create a proof-of-concept, working software that focused on implementation and functionality. Though the design itself is not finalized and clean enough for a final public release, the core features of the app are up and running reliably. These core areas of functionality are those that were outlined in the beginning stages of the project and therefore show the success and completion of what our team set out to create. There is more work required to finalize the software in terms of testing for a large user base as the underlying business idea would necessitate. The future work required to create a better UI that targets college students more effectively would require either a new branch of learning for part of our team or an addition of specialized personnel. This area of improvement is, therefore, out of the scope of this senior project, although it definitely a means of further development of KeyLime as a product. In looking forward to future work, it is also useful to look back and learn from the process and mistakes that carried this project as far as it has come. Building on and applying skills learned in the past four years of study at Cal Poly in researching new technologies, planning their implementation, using good programming practices, and innovating all the while has grown me to greater readiness in entering the professional programming field.