E. Zigmond, L. (Matt) Jiang, N. Gowravaram
Computer Science 50
7 December 2014

Budget Buddy Design

FRAMEWORK

Budget Buddy, like any good iOS application, fundamentally relies on a Model View Controller structure, which will be discussed here. The app is based on a data model implemented using Apple's Core Data framework. The data model consists of two different types of entities. The first is the receiptInfo type which stores basic information: the total amount (a double), the date of the transaction (NSDate), the type of the expense (a string), and the payee (a string), as well as a reference to the matching receiptDetails entity. The receiptDetails entity contains more detailed information that is not usually needed: the category of the expense, the payment method (both strings), and the image data for the receipt, stored as binary data in the NSData type, as well as a reference to the matching receiptInfo entity.

The primary reason to separate every individual receipt into two parts, the info and the details, was that most of the basic information is very small and quick to access, but the image data is needed only infrequently and is much larger than all of the other data in the receipt datum. Thus, leveraging the ability to define Core Data relationships in the model allowed us to save memory and speed up requests to the model by only asking for the large image data when absolutely necessary.

The data in the model is presented to the user in a hierarchy of views in the app. The "highest" view in the hierarchy is a Tab View that defines two distinct tabs, each containing a separate navigation controllers. The first of these tabs is the log tab. The log navigation controller has as its root a table view that displays a list of all of the receipts saved by the user, in chronological order. From this table, the user can either choose to add a receipt, which transitions to a form for entering information, or can click on a table cell to see the details. From the detail view, the receipt can be edited using the same form for adding receipts. Even though the control needed is different for the actions of editing versus adding a receipt, the view needed is precisely the same.

Here, the use of the navigation view controller as a parent to the various subviews was key because the navigation view controller allows us to create a first in first out stack of views. Thus, when you are leaving the receipt information entry form, the view is popped off of the stack of views and you return to the previous view, whether that is the root table view or the detail view. Thus, the views are agnostic about how the user got there, and the navigation controller keeps track of the stack.

In the other segment of the tab view is the summary view navigation controller. This part of the hierarchy is simpler and contains as its root a table view of all of the months for which the user has entered receipt data. The table only allows the user to see months for which information is available, and clicking on any of the rows in the table leads to a summary view containing a pie chart (displayed using the Core Plot framework) and other summary information for the time period. Again, the navigation view controller defines a stack of views so that the user can return to the summary table from the pie chart view.

In our design of the controllers supporting these many different views, we attempted to minimize the amount of data from the model that needed to be loaded into memory at a given time, in consideration of the small memory capacity of the iPhone, and we tried to minimize calls to the model to minimize load times. However, because of our inexperience with the Core Data framework and the limited time we had to explore solutions, some of the interactions between the model and the controllers could be improved upon.

We hoped to implement the table view using Apple's NSFetchedResultsManager, which allows one to load information from a model into a table view as an "as-needed" basis – that is, only when the user is at a point in the table when the data should be displayed. However, we had difficulty understanding the proper way to implemented the NSFetchedResultsManager and had to come up with our own solution.

Our current alternative has to load all of the information into memory at a time, which could certainly present a problem as the data set grows large. Fortunately, our good model design means that the large image data does not have to be loaded, but this approach still leaves much to be desired. Further, Apple's FetchedResultsManager has support for automatic sectioning for table views, which we had to implement by creating NSDictionary objects of all of the receipt data with the section titles as keys to arrays of receipts.

IMPLEMENTATION

This section will discuss some of the finer points of the implementation that may not be immediately evident from the comments in the source code.

The add receipt view controller attempts to automatically find the total on a receipt using the Tesseract Optical Character Recognition framework. We are very excited about this feature – when it works! We found that the recognition requires very good lighting and cropping, and even then, our string parsing algorithm only searches for sections of the receipt containing dollar signs, which does not work if a dollar sign is missed or if the receipt does not have dollar signs in the amounts.

Given more time, we would also like to extend the usefulness of the image processing to get more data like the date of the transaction or possibly the payment method. However, given the complexity of implementing partial matching considering how error-prone the text taken from the receipts tends to be, we decided to focus elsewhere.

One of the features of the app that was most interesting to implement was our custom text field that attempts to offer autocompletion suggestions to the user based on past entries stored in the data model. This is fascinating example, we believe, of a great interaction between the view (the text field itself) the controller (which passes user input to the model as a string match query), and the data model.

In order to improve the way that the autocomplete text field functions, we have also considered, but have not had the chance to implement, changes to the data model to optimize searching. It might speed up the text matching if the text to search, in this case the payees, were stored as separate entities from the rest of the receipt data. We also hope to continue thinking of ways to deal with different capitalizations of data that is clearly intended to be the same, e.g. CVS vs. CvS when it is unclear to the model which capitalization is "correct."

CONCLUSION

This concludes the design documentation for our application Budget Buddy. Any further questions may be sent via e-mail to the creators of the application, whose contact information is below.

Ezra Zigmond
- ezigmond@college.harvard.edu

Linghua (Matt) Jiang
- linghuajiang@college.harvard.edu

Nihal Gowravaram
- nihalgowravaram@college.harvard.edu