**Criterion C: Development**

Techniques used to create this project:

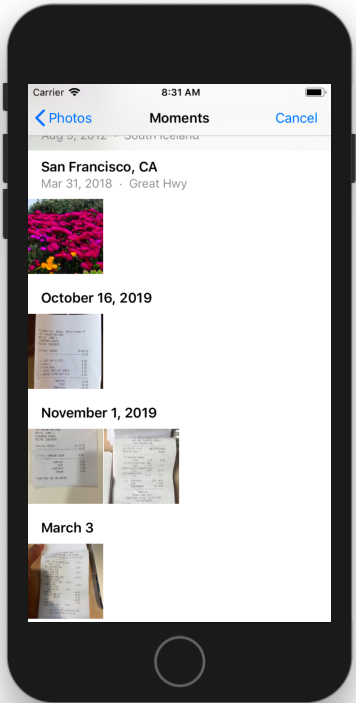
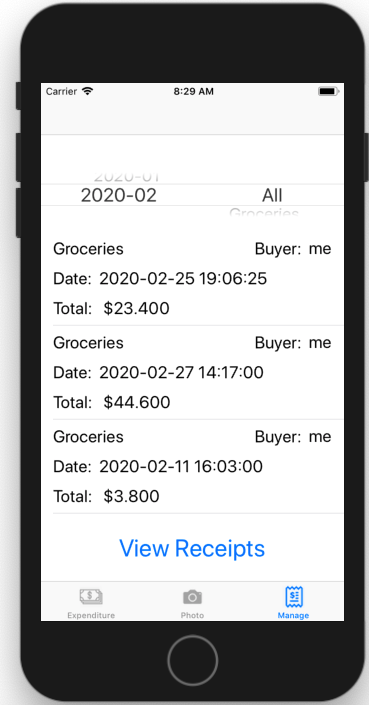
* Graphical user interface using Swift UI
* Use of libraries
  + Alamofire
  + Libraries in Apple’s IOS framework
* Use of Firebase API
  + Using Firebase Authentication
  + Using a NoSQL database (Realtime Database)
* Use of REST API (Tabscanner) to do OCR
  + Communicating asynchronously with escaping closures
* Encapsulation
* Static Polymorphism
* Modularity
* Validation

*\*The criteria numbers in this document refer to the success criteria in Criterion A: Planning*

*\*\*The full code for the classes that are referred to in this document can be found in the code listing in Appendix D*

**Graphical User Interfaces**

I used the user-friendly Swift UI and the storyboard to create view controllers with UIButtons, UIViews, and more. This satisfies criterion (3a) as the buttons and images have styles consistent with those of most other apps on iPhones, improving usability. This is shown for the ‘choose photo’ screen, and the ‘all receipts’ screen below.



The tab bar images are obvious, and the text below them adds detail. By allowing the user to choose a photo from their photo library or take one with the camera, I have satisfied criterion (5a).

**Use of libraries**

Alamofire

I installed Alamofire using CocoaPods to begin using the framework. Alamofire allows me to communicate with the Tabscanner API to do the OCR conversion of a receipt image and therefore satisfy criterion (5c). The library allows me to use predefined functions as a black box, reducing development time. Below are screenshots from my NetworkingClient class where I POST and GET the data from the API using the Alamofire functions.

***POST:***





***GET:***

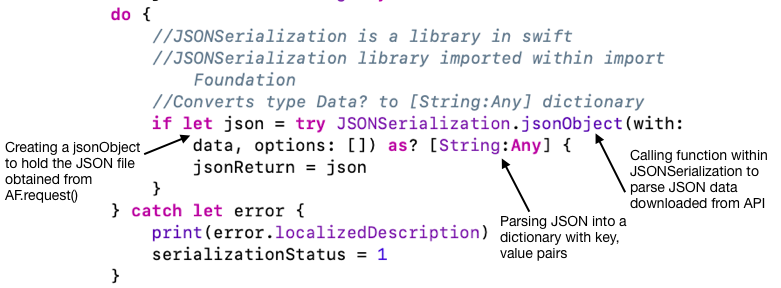


As required by the API, I did the POST function using multipart/form-data as my encoding type.

I then used the Alamofire **.request()** function to GET the OCR converted receipt from Tabscanner as a JSON file, which I then parse to display the data included in the receipt.

Libraries and Classes in Apple’s IOS

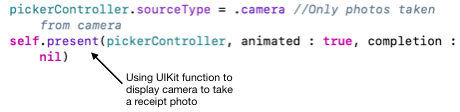
Apple’s IOS comes with many libraries included as part of the UIKit framework and the Foundation framework. In my project, I used the UIKit framework to manage user interactions with **UITextFields**, **UIImages**, **UITableviews** and more. In the Foundation framework, a **NSJSONSerialization** library is imported, which I used to parse the JSON file received from Tabscanner. Below is a screenshot of the code where I used the **JSONSerialization** library.



This helps satisfy criterion (2a) as I can then do various calculations using the dictionary and display the data to the user.

I also used classes such as **UINavigationController**, **UIImagePickerController**, **UIPickerView**, and **UITableView** to navigate between screens,choose receipt photos, and display data effectively and simply satisfying criteria (3a & 3b). Below is the sample code where I used the **UIPickerController** to satisfy criterion (5a).

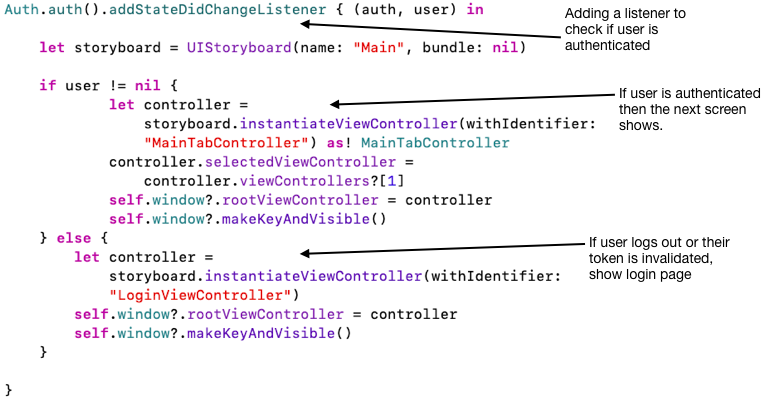




**Use of Firebase**

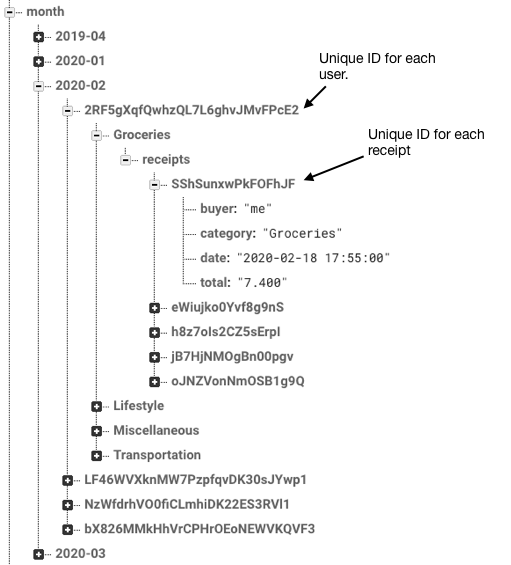
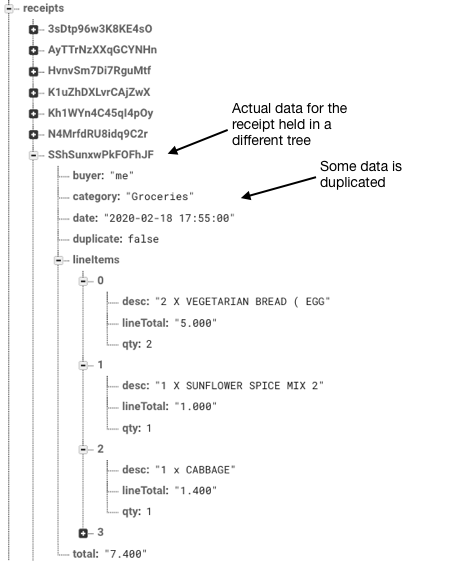
Firebase Authentication

I used the Firebase API in my project for authenticating users and then storing the receipt data in the realtime database. Below is a screenshot of my code where I used Firebase authentication. The user enters their email and password into email and password text fields and then either signs up or logs in. The user data is sent to firebase, which then attempts to create a new user or authenticate an existing one. If the user logs out, or their log-in token becomes invalid, the user is sent back to the login page. This authentication satisfies criterion (1b), as the user’s expenditure details can be kept private.



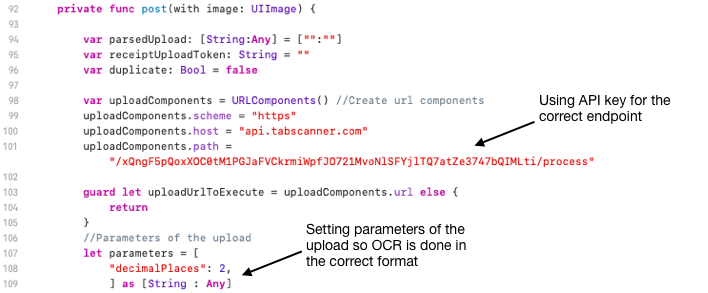
Firebase Realtime Database

To save data of previous OCR converted receipts, I used the Firebase Realtime Database. This meant that a user could not only select a previous month and view the expenditure on that month, but also could see the purchases made on that month in a specific category. In the Realtime Database, I stored the data on receipts and the purchases within them, and used unique identifiers so a specific receipt could be located. Thus, I satisfied criteria (4a & 4b). For efficient querying, I used a denormalized structure (as shown in the screenshots below) by utilising unique ids for each receipt and each user. Thus, I could separate the data into different trees, which avoids having many layers of nested data. This improves read and write times as when data is fetched at a specific path, all the child nodes are also retrieved. However, this also meant that some data was duplicated. Nevertheless, read and write time was improved as if the user only wanted to know brief data of the receipt (i.e. **buyer**, **category**, **date**, & **total**), they did not also have to download the **lineItems**.



**Use of REST API (Tabscanner) to do OCR**

To satisfy criterion (5c), I used a REST API named Tabscanner to do the OCR of the receipt photos. In order to do so, I created a **URLSession** with the Tabscanner API to POST the receipt image, and then GET the data using the endpoints specified by the API.





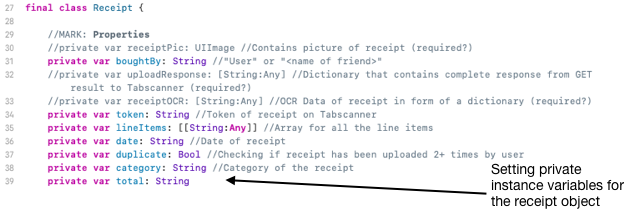


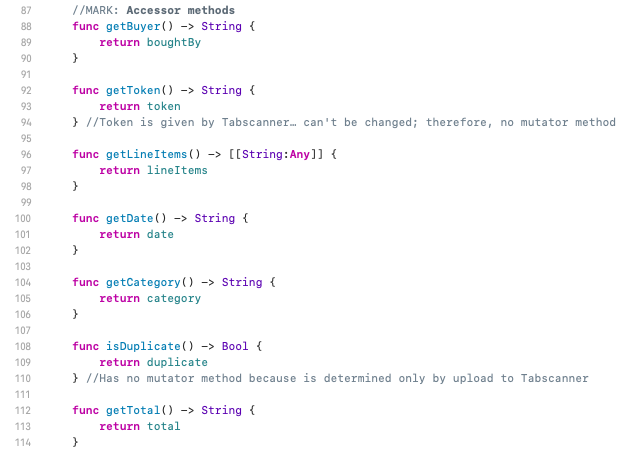
Communicating Asynchronously with Escaping Closures

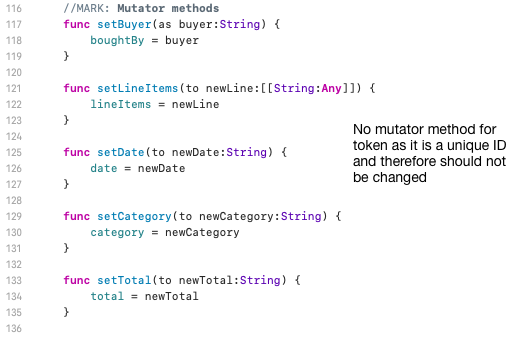
In the code screenshots above, I used escaping closures because the POST and GET requests were asynchronous. For example the code below line 162 is within the NetworkingClient’s **.getOCRData()** function’s closure, and so is only run when the JSON file has been received by the user’s device. This prevents **nil** values being shown and used.

**Encapsulation**

Encapsulation was used to hide the implementation of certain data and methods, and ensure certain variables weren’t changed. In the screenshots below, I used accessor and mutator methods to prevent private variables within a receipt object such as the **token** (that is a unique ID for each receipt) to be edited. This facilitates code reuse and maintainability.







**Static Polymorphism**

Static polymorphism was implemented in many of my classes as I had to set the instance variables of those classes from many different classes. For example, my **FirebaseHandler** class has two static functions to download the receipts which are shown in the **ManageViewControllerClass**.



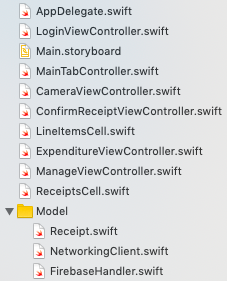
Used when the category that the user wants to get the receipts for is “All”



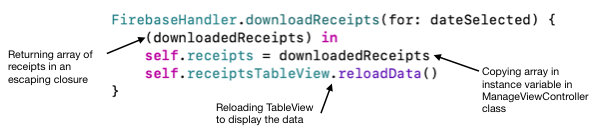
Used when the user wants to get receipts from a specific category.

**Modularity**

I used modularity by breaking my program down into multiple view controllers, which each had custom classes that allowed me to keep the functionality for each screen contained within that view controller’s class file.



I used abstract classes such as **NetworkingClient** and **FirebaseHandler**, improving maintainability as the upload and download functions were kept organized in these static classes. Furthermore, it reduced code repetition as multiple View Controllers needed the data from my database, so they just called the static functions.

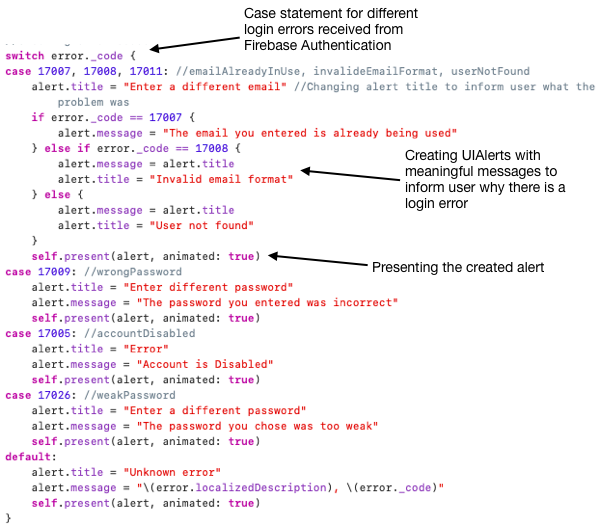


**Validation & Verification**

Validation

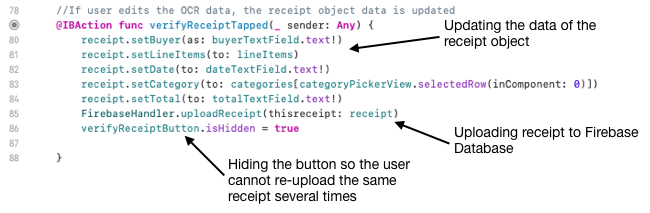
Validation whenever data was being manually entered by a user to ensure the data was of the correct format. For example, in the login screen I was able to connect to the Firebase Authentication API in order to check if the emails were given in the proper format (@…com), and that the login details were correct. I also validated that the user had actually typed something into the text fields when logging in (shown in the screenshot below).





Verification

The accuracy of the OCR would depend on the lighting, photo size, camera quality, and more, so I used a manual double verification check so that my user could confirm its accuracy. I presented the OCR data to my user so they could correct the data before it was saved to the database. To do this, I created another view controller that was instantiated any time a receipt object was created.



**Word Count: 1143**

Works Cited

Advent, Brian. "iOS Swift Tutorial: Working with Camera & Photo Library - UIImagePickerController 17/24" *YouTube*, 17 Dec. 2016, www.youtube.com/watch?v=4CbcMZOSmEk.

Atanasov, Dejan. "Alamofire 4 ? Multipart File Upload with Swift 3." *Medium*, 2 Aug. 2018, medium.com/swift2go/alamofire-4-multipart-file-upload-with-swift-3-174df1ef84c1.

Beaulieu, Dan. "How Do I Setup a Second Component with a UIPickerView." *Stack Overflow*, 12 May 2015, stackoverflow.com/questions/29617835/how-do-i-setup-a-second-component-with-a-uipickerview. Accessed 5 Mar. 2020.

"Firebase API Reference." *Firebase*, Google, firebase.google.com/docs/reference. Accessed 13 June 2019.

"PhotoPicker Discovery Error: Error Domain=PlugInKit Code=13." *Stack Overflow*, stackoverflow.com/questions/44465904/photopicker-discovery-error-error-domain-pluginkit-code-13.

"UIKit Documentation." *Apple Developer*, Apple, developer.apple.com/documentation/uikit. Accessed 11 May 2019.

*World's Most Advanced Receipt OCR With Cloud Receipt API*, Tabscanner, tabscanner.com/. Accessed 17 Aug. 2019.