## ***Criterion C: Development[[1]](#footnote-0)***

\**A client account is hard coded into the program with a username: “Samuel” and password: “example”*

# **Technique #1:**

Database Entry/Exit (Check-in/Check-out) login and creating new Log, Day, and Month entities based on the timestamp of the login.

Libraries

| * Apple’s CoreData framework[[2]](#footnote-1)   + @NSManagedObject subclasses to connect with the Core Data sqlite database.   + NSManagedObjectContext to save into the database. * SwiftUI framework[[3]](#footnote-2)   + FetchedResults<*Result*> Object to fetch data results |
| --- |

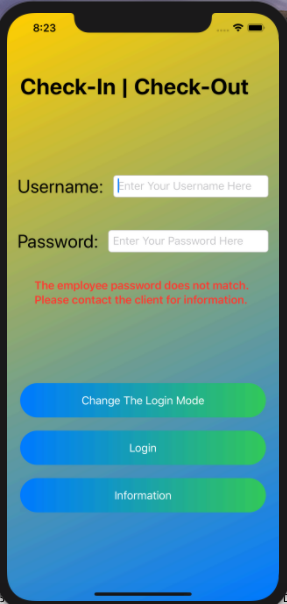
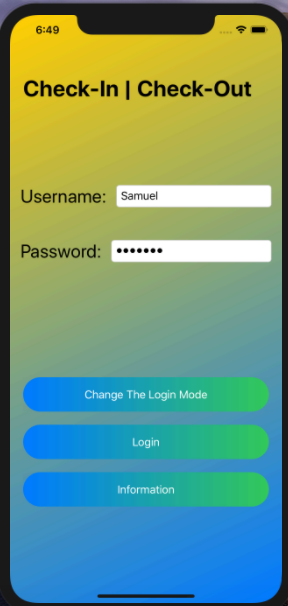
*\*For this technique, refer to Criterion B algorithm: Checking-in or out*

Thinking Procedurally:

1. With a device on-site (on the workplace’s premises), a user logs in for check-in/check-out securely.
2. Credentials are verified through a *credentialHandler*.
   1. Errors are handled and displayed through an enumeration of *credentialPossibilities*.
3. If credentials are valid, a *logHandler* adds a new log with the timestamp of the current time into the database.
4. If the log is the first of its day, a Day entity is created. If the day is the first of its month, a Month entity is created. Otherwise, the Log will just be assigned to its corresponding Day entity.

## 

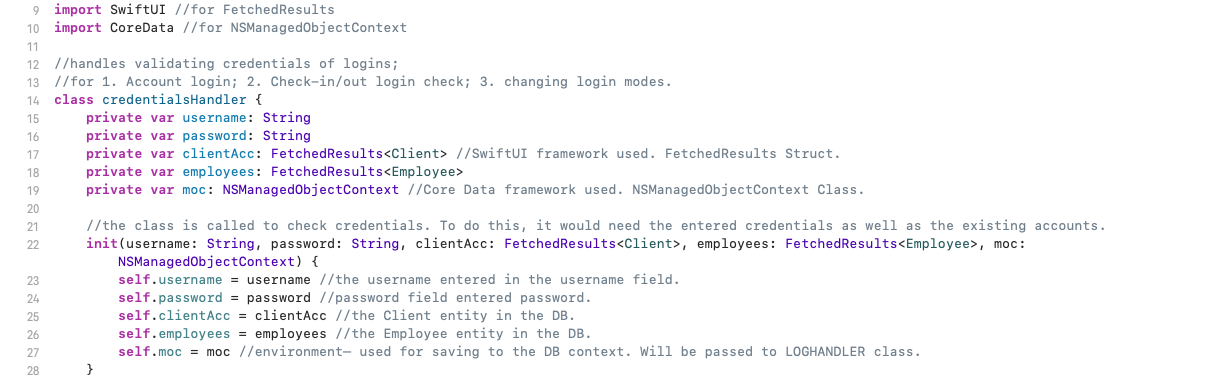
## A and B. Checking-in/out and verification



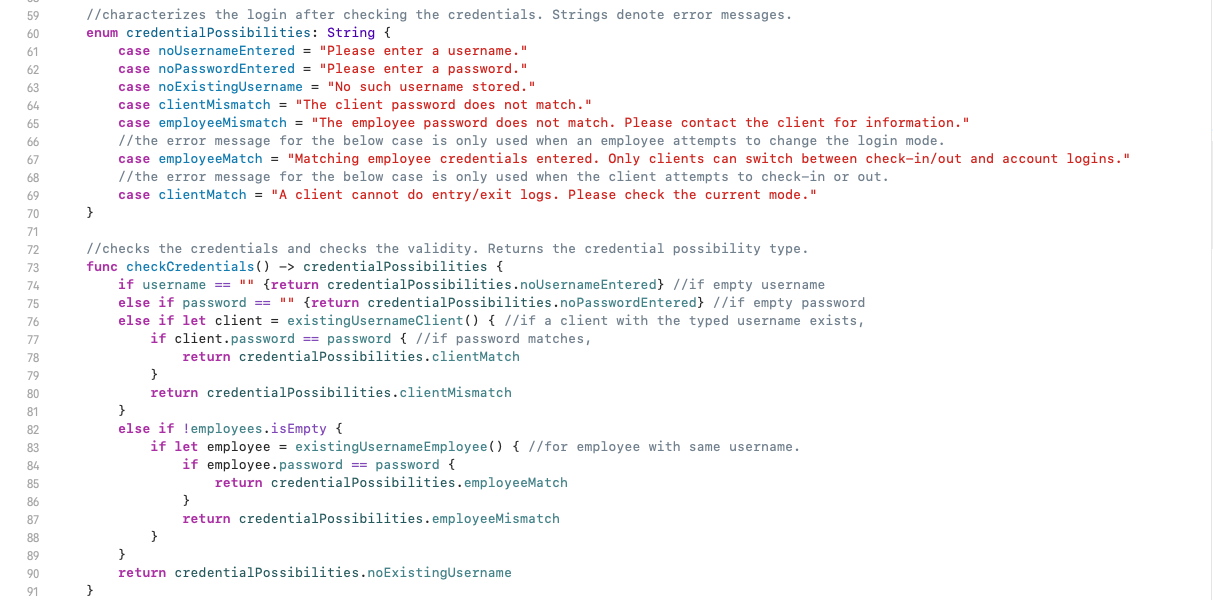
*Typed username and password (left).*

*Displayed error message of invalid employee credentials (right).*

Whenever a user attempts to login or change the login mode, credentials have to be checked.

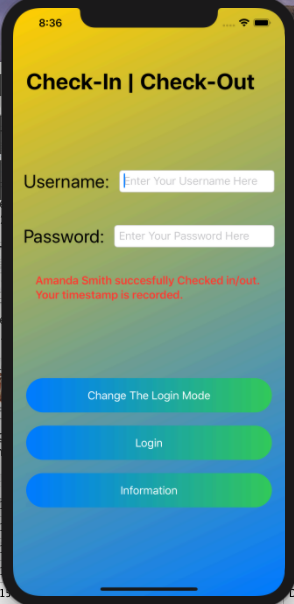


The *credentialsHandler* class handles the checking of entered credentials and returns errors if any.

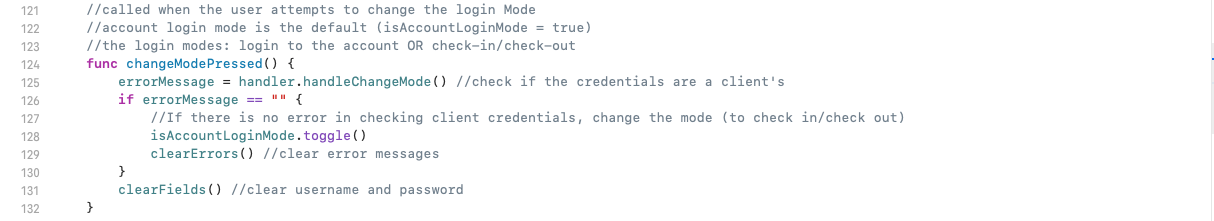


The *credentialsHandler* class has a *checkCredentials* method– which checks for if users with typed credentials exist. The class uses the enum *credentialPossibilities* to allow for thinking abstractly about the validity of the credentials, while String value pairs are used to allow return of error messages. This can be easily understood by a third party maintaining the program.

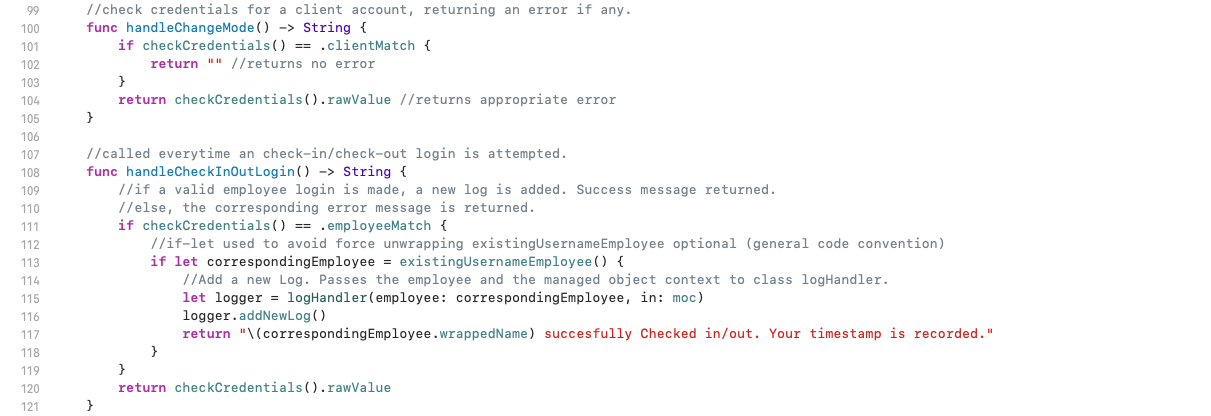
Checking credentials is used for user verification, which firstly allows secure and personalized logins (1rs success criteria). Secondly, it allows for personalized check-ins/outs (part 1 of 6th success criteria). Thirdly, it can be used to ensure that client verification is needed to change modes and activate check-in/out (part 2 of 6th success criteria).

*Successful Check-in/out (left) and error message displayed when an employee attempts to change mode (right). Colors characterized by login mode (to Check-on/out or to log to account).*

\*The user can press the “Change The Login Mode” button (*refer to Criterion B mockups for details*) to switch between logging into user accounts and checking-in or out.\*

**

When the button is pressed, *changeModePressed* is called. *clearErrors* and *clearFields* interact with the GUI and allows the user to see error messages if any.



The check-in/out login mode is intended for on-site check-in/out. To ensure this, the default login mode is an account mode, and changing modes are handled by the *changeModePressed* and *handlerChangeMode* method, which will only change modes if there is a client match. This ensures that employees cannot check-in/out with their own devices (2nd part of the 6th success criteria).

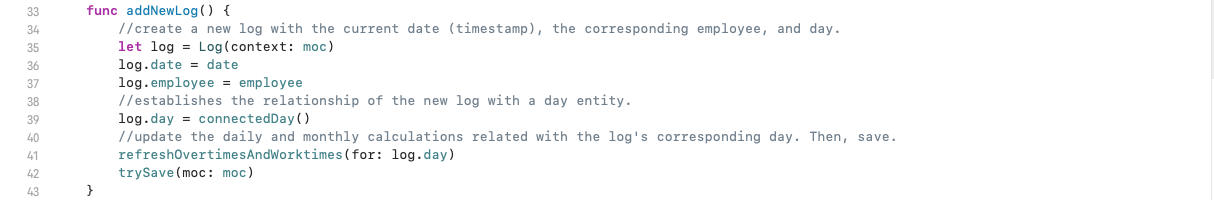
*handleCheckInOut* will check for verification on checking-in or out. Both *handleChangeMode* and *handleCheckInOut* use *checkCredentials* as they both need to validate credentials, evidence of thinking abstractly.

If a check-in/out is valid, a *logHandler* class handles the adding of a timestamp log.

## C. Timestamp of log and storage

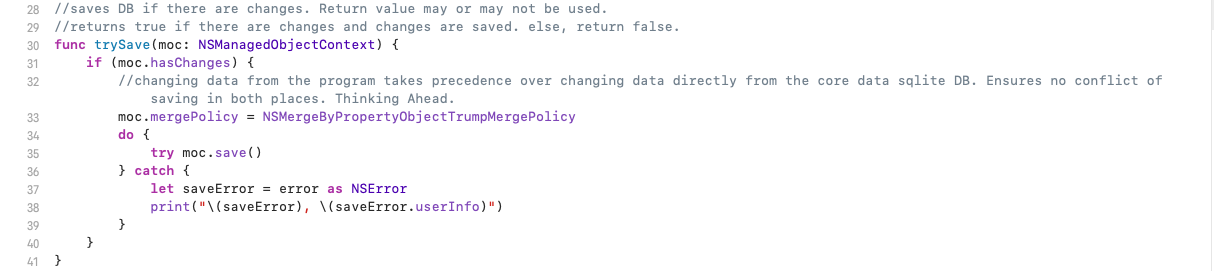
## 

\**UTC timezone used to cater to the client’s needs (refer to Appendix B)*.

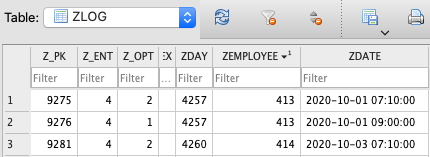


*logHandler* handles log changes. The *addNewLog* of the class allows a check-in/out timestamp to be recorded (line 35~39) and saved into the database through the *trySave* method (1rst part of the 6th success criteria). The Log’s relationship to a Day is done through the *connectedDay* method.





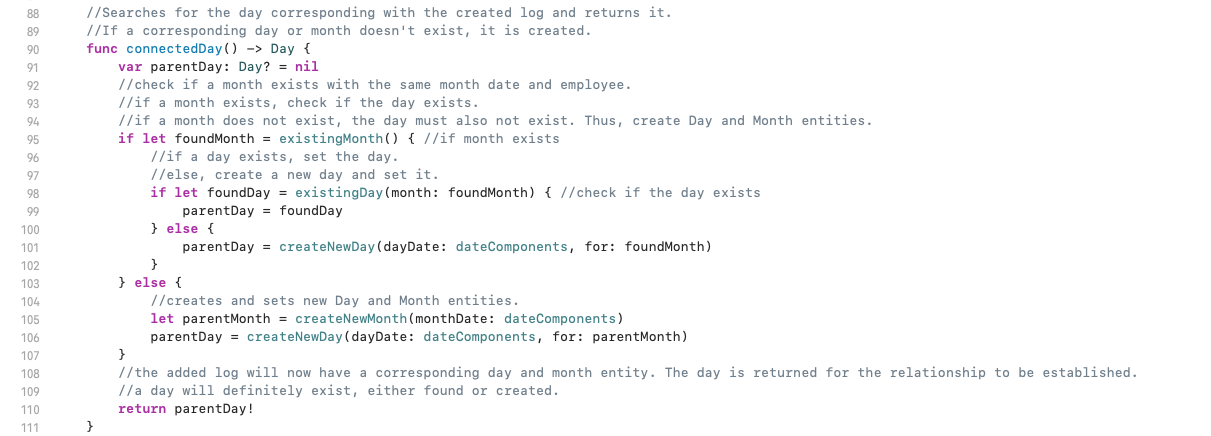
An *NSManagedObject* subclass of a Log entity is created beforehand with *@NSManaged* variables– used to connect to the database. This allows *addNewLog* to create a Log in the database with few lines of code. *trySave* is used to save into the database, evidence of storage mechanism, and adds a Log with the appropriate information:



*Log table in the sqlite Core Data database. ZDay and ZEmployee are foreign keys.*

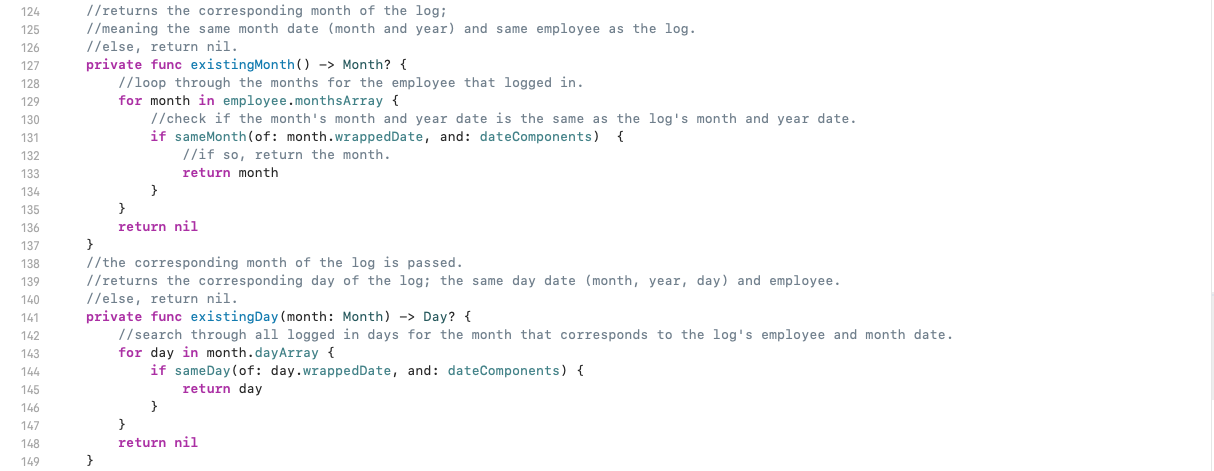
## **D. Setting entity relationships and creating new Day and Month entities.**

When adding a Log, its relationship with a Day is created through the *connectedDay* method (line 39 of *addNewLog*).

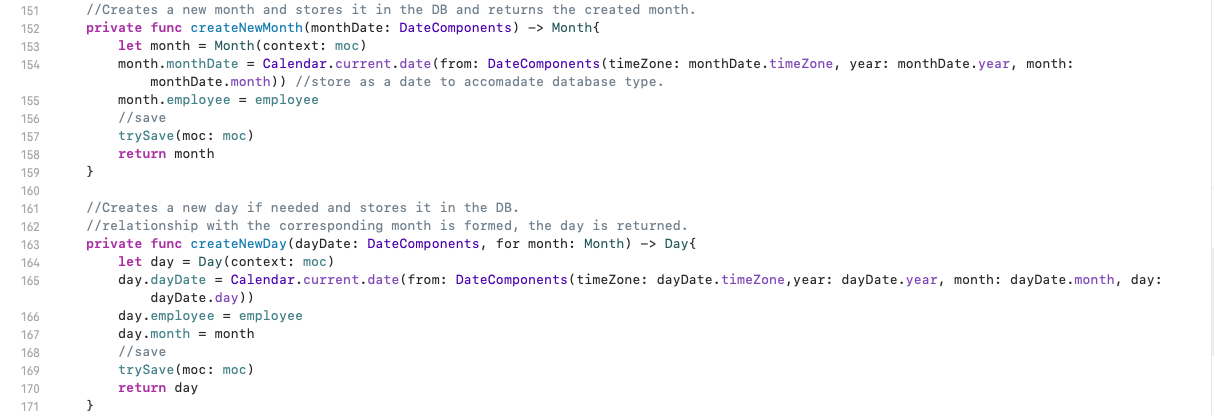


The *connectedDay* method finds the Day that should relate to the check-in/out Log while concurrently creating Month and Day entities that do not exist, hidden abstractly from the user. A nested if-let structure (95-103) is based on the logic that a Day should be checked after a Month is checked.

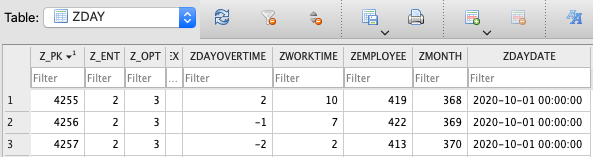
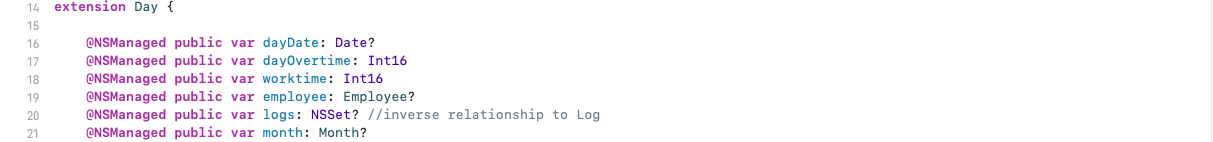
The method ensures that relationships between appropriate Month, Day, and Log entities are always established after a check-in/out is made. These relationships are necessary to allow employees (or the client viewing an employee) to view their month, day, and log information (8th success criteria).



*existingMonth* and *existingDay* loop through months and days respectively to find the corresponding entity. This is done through the *sameDay* and *sameMonth* method, which cross-checks a Day/Month and a Log’s month/day date and employee (*refer to Criterion B ERDs on Foreign Keys*).







*\*ZEmployee refers to the Primary Key of the Employee*

The *createNewDay* and *createNewMonth* methods create a new entity in the *NSManagedObjectContext* (moc) by accessing the variables in the @obj(Day) NSManagedObject class and save it to the database– storage mechanism. The dates of “Month” and “Day” entities (refer to Database Screenshots) only store the month and day information of a log respectively, so these entities avoid duplication. These methods are needed to ensure that necessary Day and Month information of employees are stored (8th success criteria)

Procedures C~D, are performed when a check-in/out is performed and not seen by the user– evidence of thinking abstractly.

# 

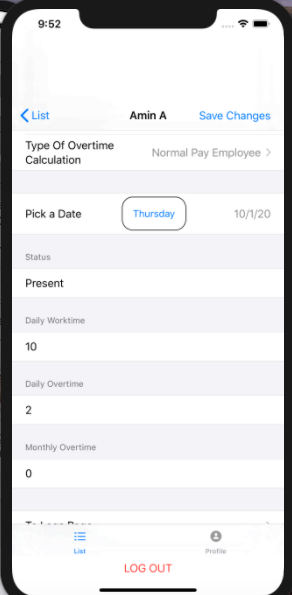
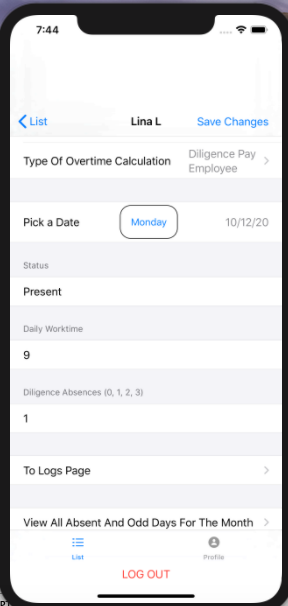
# **Technique #2:**

Calculating for day worktime and overtime, and monthly overtime and diligence absences, by tracking absences and manipulating dates.

Thinking Procedurally:

1. When a log is added or changed, the daily worktime and, for normal pay employees, daily overtime, is calculated and capped if needed.
2. Absent days or days with odd numbers of logs are recorded.
3. For diligence pay employees, diligence absences are recorded and capped if needed. For normal pay employees, daily and weekly overtime and absences are used to calculate and cap, if needed, monthly overtime. (*refer to Appendix B and Success Criteria #8, 9 for calculations and caps of different information*)

## GUI

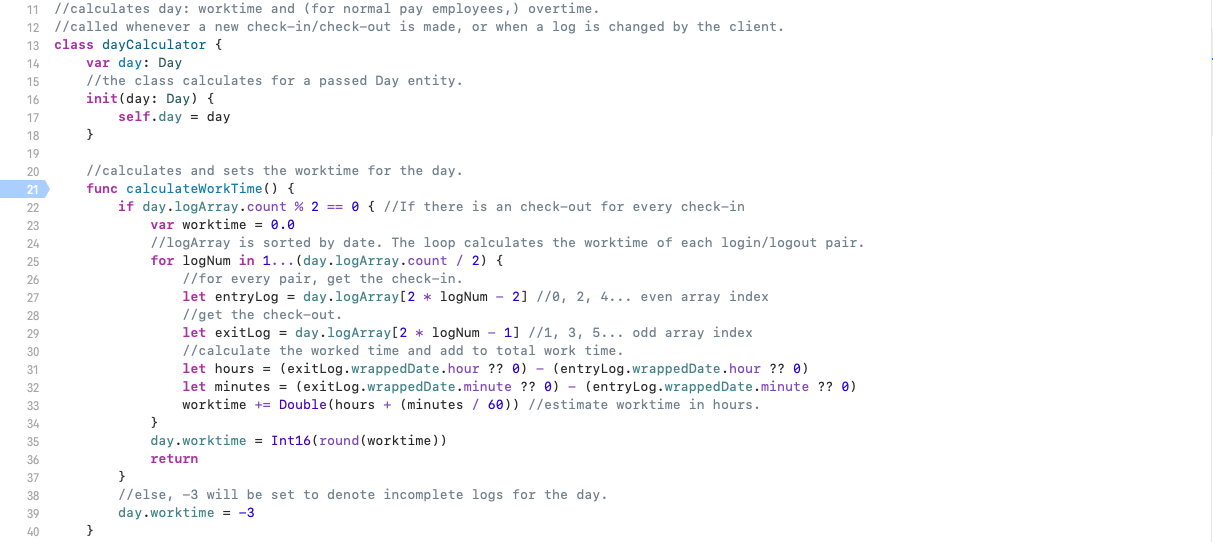


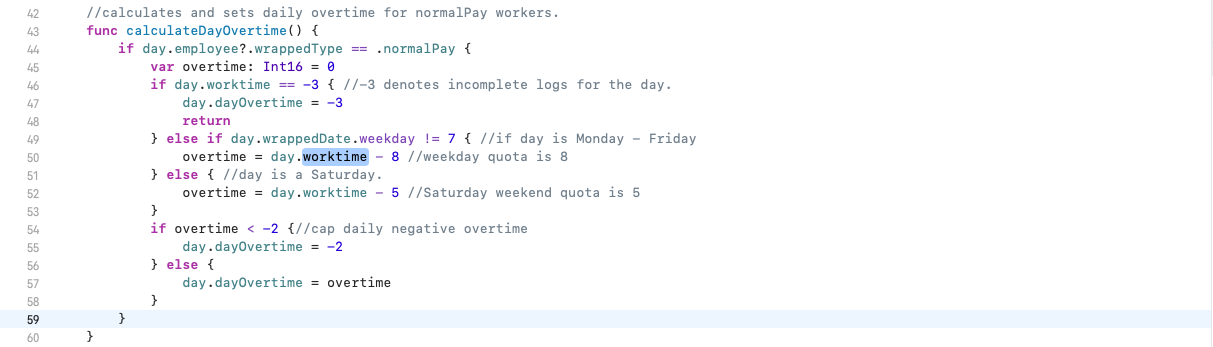
*Part of employee information view for a Diligence Pay Employee (right) and a Normal Pay Employee (right). Date picker is collapsed (refer to Appendix E for complete UI).*

The GUI is part of the employee information page.

## A. Calculating daily worktime and overtime

*For this calculation, refer to Criterion B algorithm flowchart:* Calculating For Day Work Time And Overtime



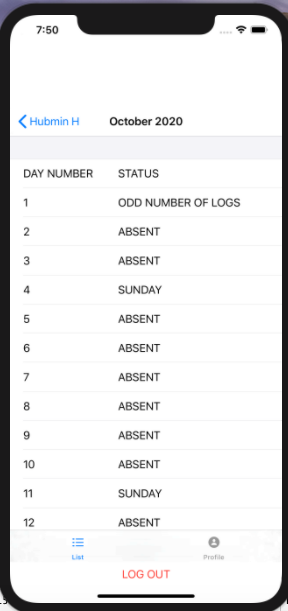


The *dayCalculator* handles calculation for a specific Day. It uses *calculateWorkTime* to calculate the daily work time of all employees. This is done by manipulating an array of the check-in/outs for the day, and determining which log is entry and exit (26~33, Success criteria #8a) is evidence of logical thinking. *calculateDayOvertime* calculates the daily overtime for normal pay employees (#8b). Capping the overtime at -2 (#9a) and using -3 (*refer to criterion B*) to denote incomplete/odd logs are evidence of thinking ahead. These dictionaries and calculations are kept abstract from the user as per the client’s needs (*refer to Appendix B– additional details*).

By using these abstract methods, changes can be made and the calculations can be maintained by third parties if the method of calculation ever changes.

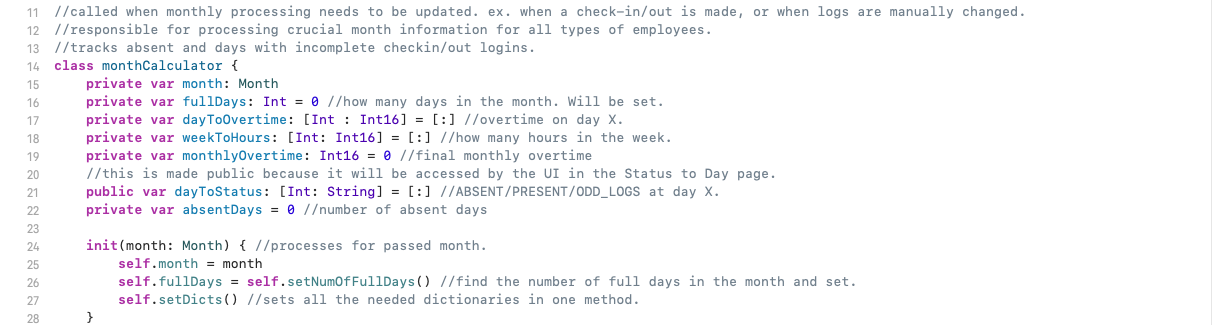
## B. Monthly absences, weekly overtime, odd logs

*Refer to Criterion B flowchart algorithm:* Processing for month information



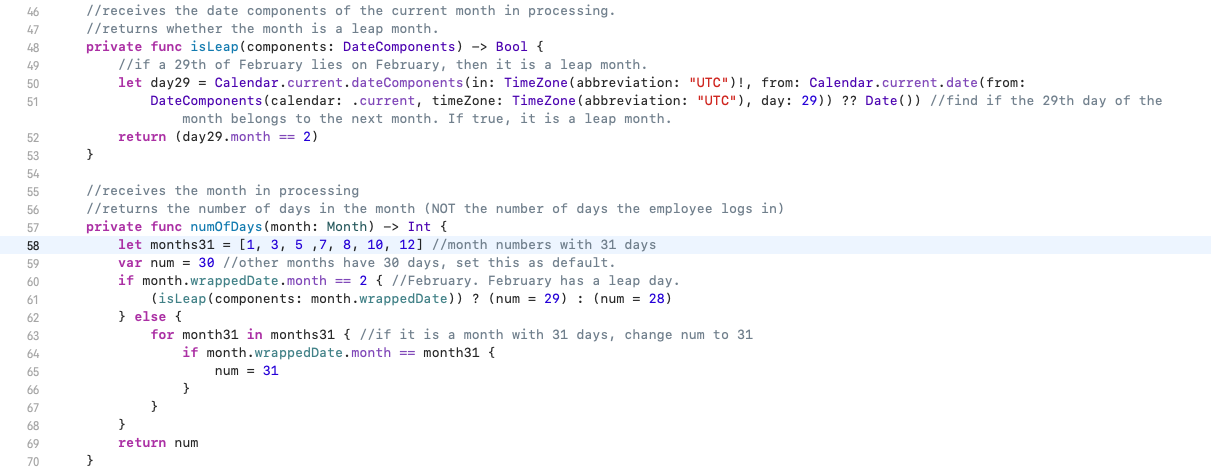
*UI view of the list of days and status. View is scrollable and continues until the end of the month– October (refer to Appendix E for continued view).*

Determining the status of days in a month and calculations happen through the *monthCalculator* class:

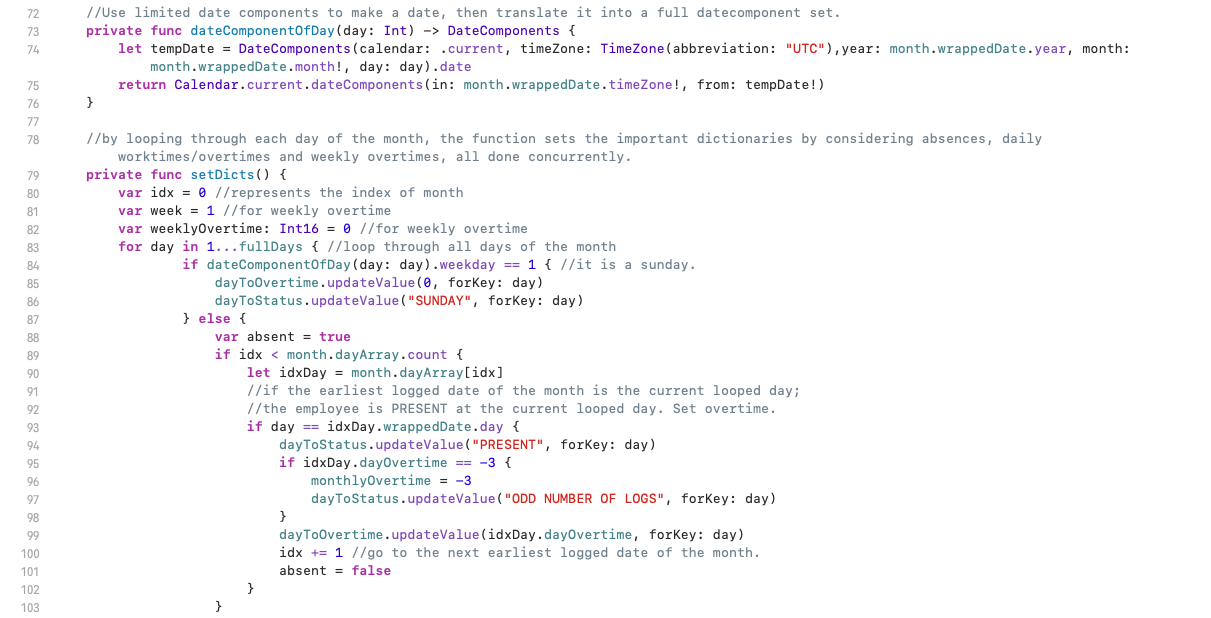


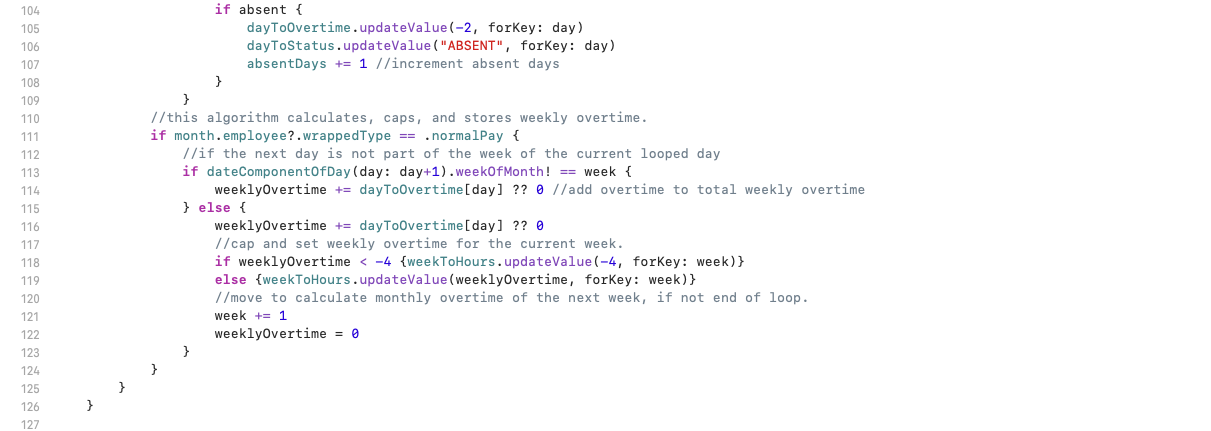
Abstract data types– Dictionaries– are used in this class to relate the day number to certain information. Dictionaries are not directly stored in the database but rather are used to simplify the calculation process and are third-party friendly.

The dictionaries allow the processing of monthly overtimes and diligence absences (8th success criteria). It also allows employees to view their status for each day of the month.



The *isLeap* and *numOfDays* methods are used to determine the full days of the month. This is needed to loop through each day of the month and determine necessary month information, evidence of thinking ahead and procedurally. This is used to loop through all the days of the month, allowing for absences to be tracked.





In one loop through all days of the month, most month information is found and dictionaries are set, evidence of thinking concurrently. For each day, the status of the day, daily and total weekly overtime, absences, and odd logs are all considered and found. One loop is done over multiple loops, making the calculation process more efficient.

This is done abstractly. It will set the dictionaries and allow the client to view their month information and their statuses of each day of the month.

Complex logical thinking of code above:

* Determining present/absent days (line 86~100)
  + Starting with idx = 0 and traversing through the Month entity’s dayArray along with the for loop. This is a complex manipulation of sets.



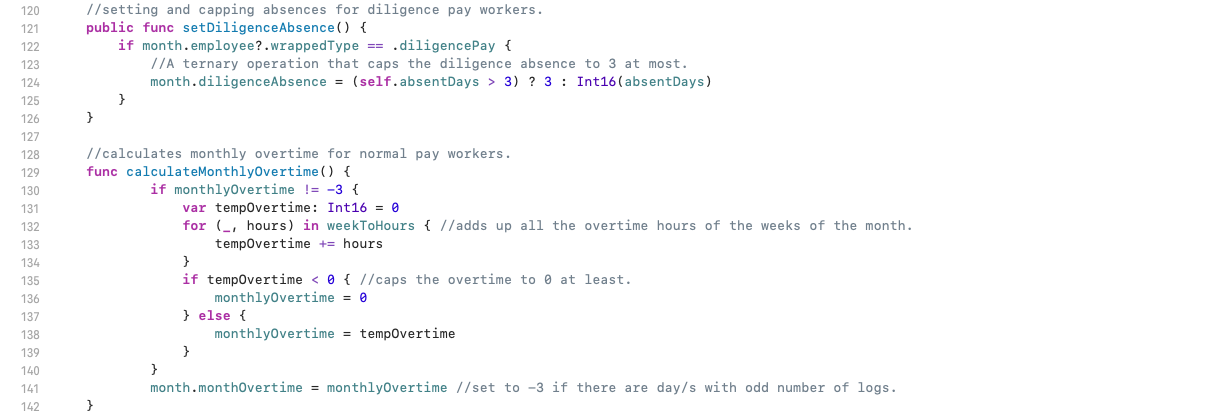
* + The *dayArray* is sorted starting from the earliest date, evidence of thinking ahead.
  + The looped day will be checked with the current front of the dayArray through a nested IF structure.
  + When they are the same, it means that the employee was present on that day. The idx increments, meaning that as the loop continues, it will check with the next earliest date.
  + Based on the logic that all days in the dayArray will be a subset of the loop of all days. This algorithm ensures that all present days are found.
  + Will be used for success criteria #8c.
* Determining weekly overtime (line 108~122, *setDicts*)
  + Using week as a count;
  + Add daily overtime to weekly overtime.
  + If the looped day is the last of its week, the weekly overtime is stored (capped, if needed– 9th success criteria) and the next looped day will affect the weekly overtime of the next day.
  + This calculates and caps weekly overtime along with just one concurrent loop through individual days.

## 

## C. Monthly overtime and diligence absence

*Refer to Criterion B flowchart algorithm:* Monthly Overtime

(*refer to Technique #2 GUI for the GUI display of monthly overtime and diligence absence*)



The *setDiligenceAbsence* method sets the diligence absence and caps it if necessary. The *calculateMonthlyOvertime* method adds up capped weekly overtime to calculate monthly overtime and will cap if necessary (success criteria 8c, 9c, 9d).

# Technique #3:

Converting necessary employee information for a month into JSON data and email as an attached file.

*For this technique, refer to Criterion B flowchart algorithm:* Sending Json Information as an Email To Client

Libraries/Objects:

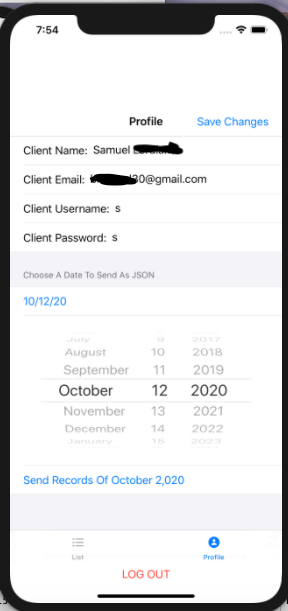
| * MessageUI framework[[4]](#footnote-3)   + MFMailComposerViewControllerDelegate protocol: monitors the email sheet.   + MFMailComposeViewController object: sets information for the email.   + mailComposeController:didFinishWithResult:error: instance method: dismiss the view after sending email. * Encodable protocol and JSONEncoder object: allows objects to be encoded to JSON. |
| --- |

**This technique satisfies the 10th success criteria.**

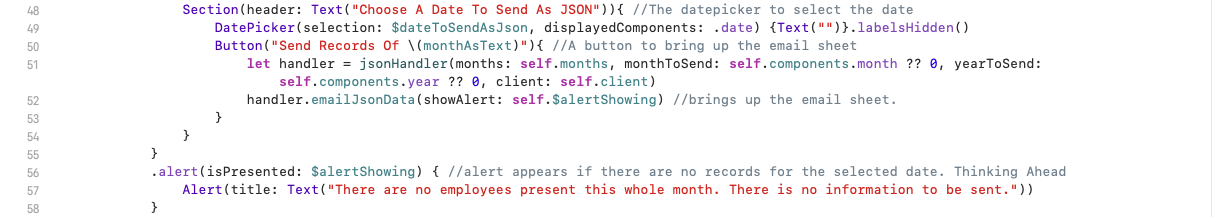
Thinking Procedurally:

1. The client chooses the month date (month and year) that he wants.
2. For each month entity with the selected month date (since one month entity belongs to one employee only), wrap necessary information (username, name, extra wage type, monthly overtime, diligence absence) into an Encodable object.
3. Convert an array of the object into JSON data.
4. Send as an attached JSON file in an email.

## A. GUI and filter months

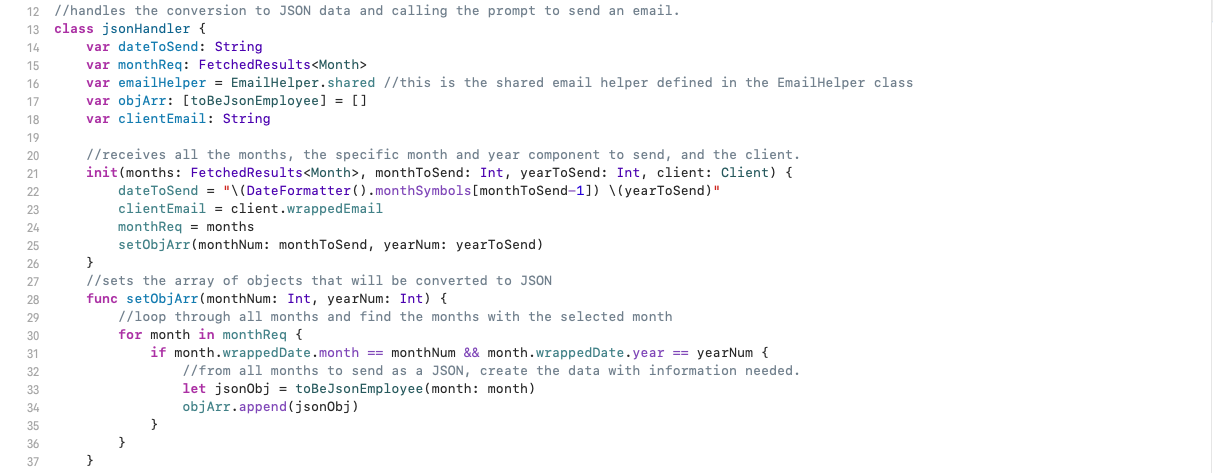


*Client profile information and a date picker to choose which month information to send.*



*GUI code. Refer to Appendix E for full code.*

The client can select a date through a datepicker and request for a send, which will call *handler*– a *jsonHandler*. An alert will be prompted if there is no information to send.



The *jsonHandler* sets the object array through the *setObjArr* method, which loops through the months. Months with the selected month date (year and month component) are converted into *toBeJsonEmployee* objects. This ensures that all Month entities with the selected date will be part of the object array, evidence of thinking logically.

The method allows Mr. Samuel to be able to send information of Employees on the certain month that he selects. If the data that the client needs changes in the future, this class can be edited.

## B. The encodable object

This object will be encoded to JSON and contains only the necessary information the client wants (*refer to Appendix B– Additional Details*) and is not seen by the client directly, evidence of thinking abstractly.

Creating this struct allows the information to be encoded into JSON, ready to be sent as an email.

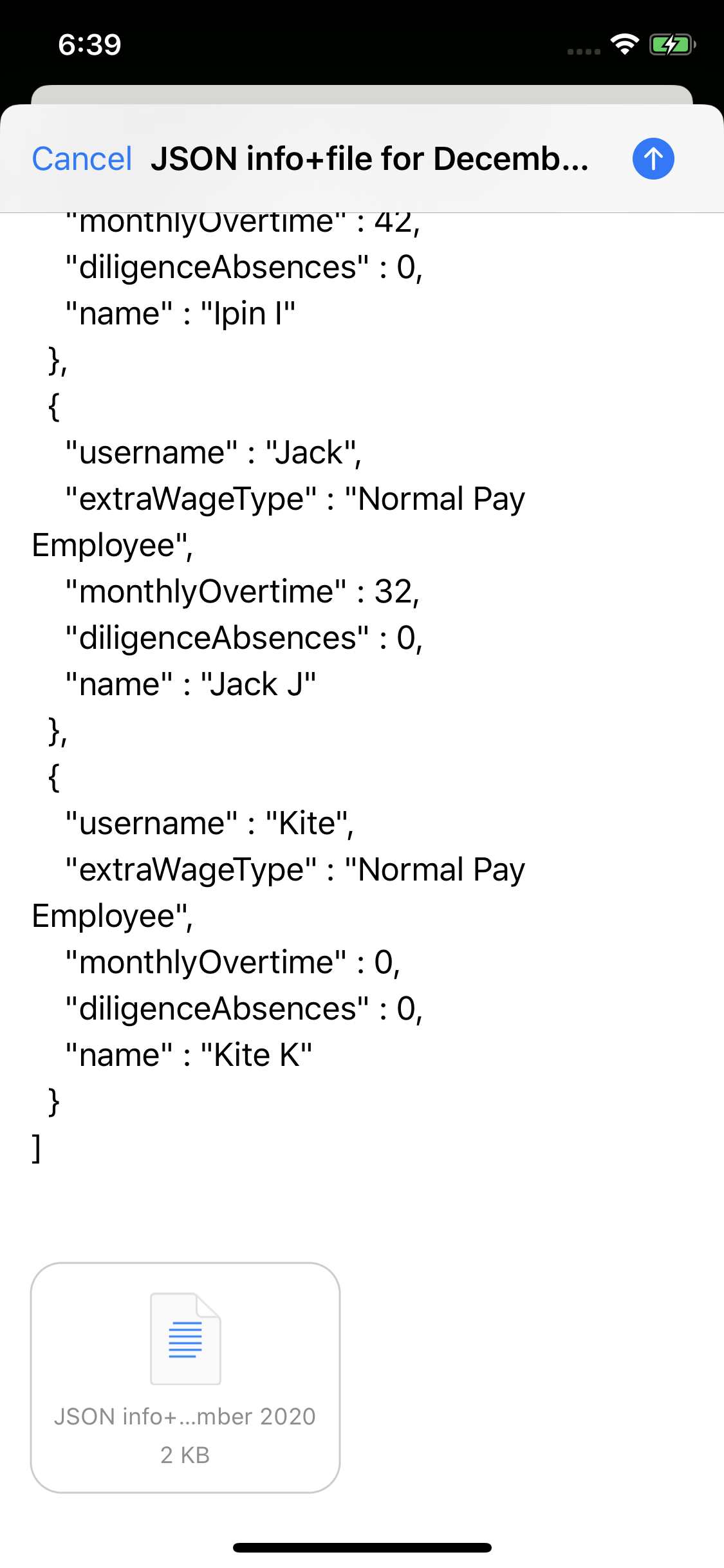
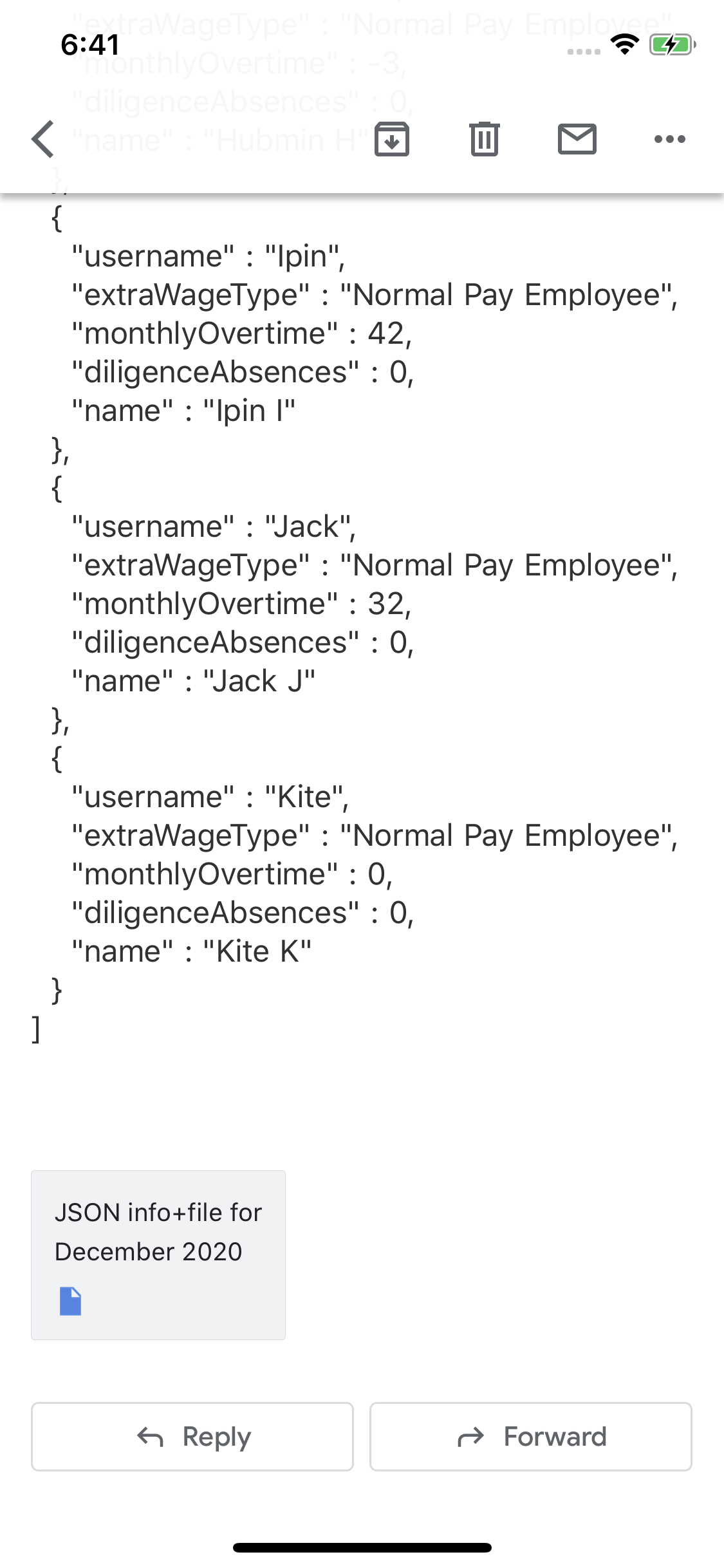
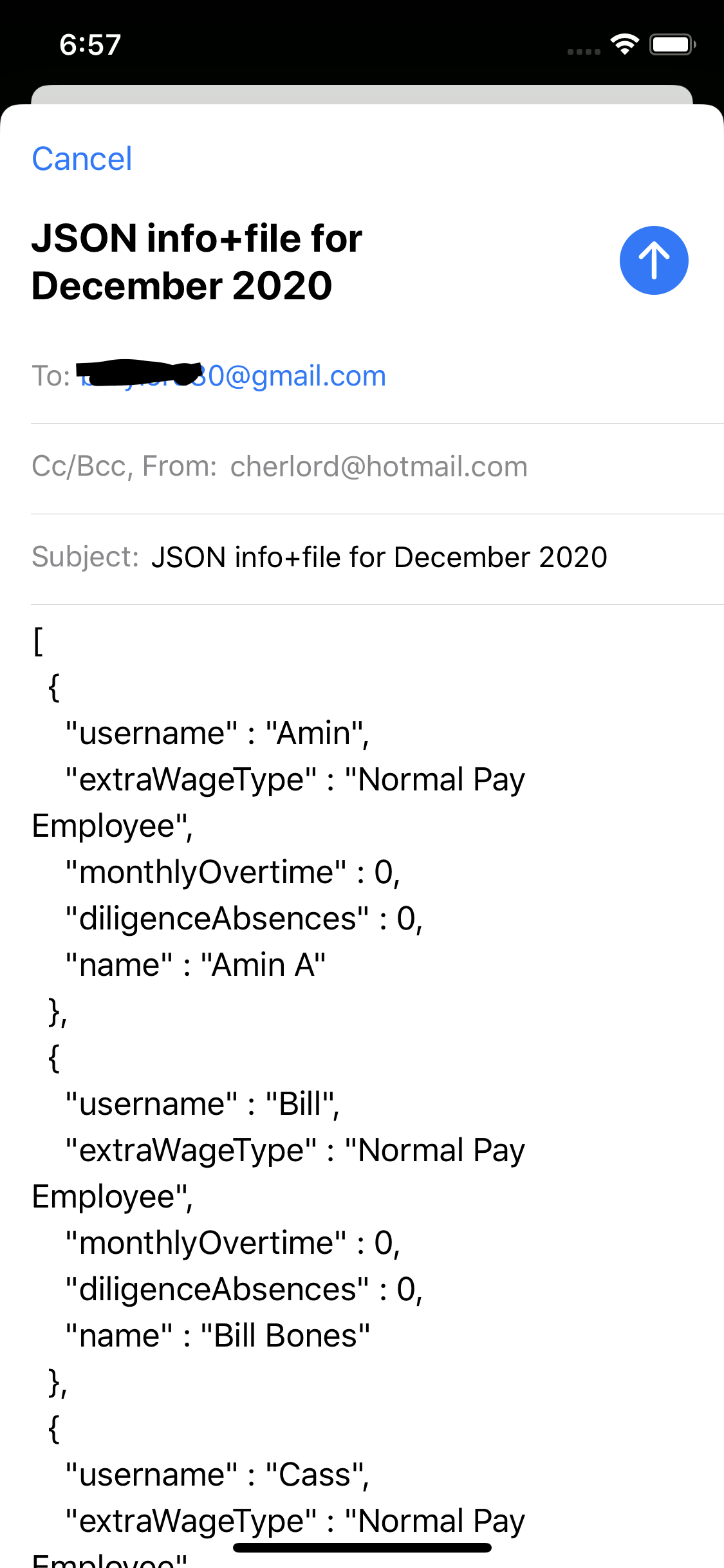
## C. Encoding to JSON

The *emailJsonData* method tries to encode the object array into a Json file as well as the string json form, printing an error if failed.

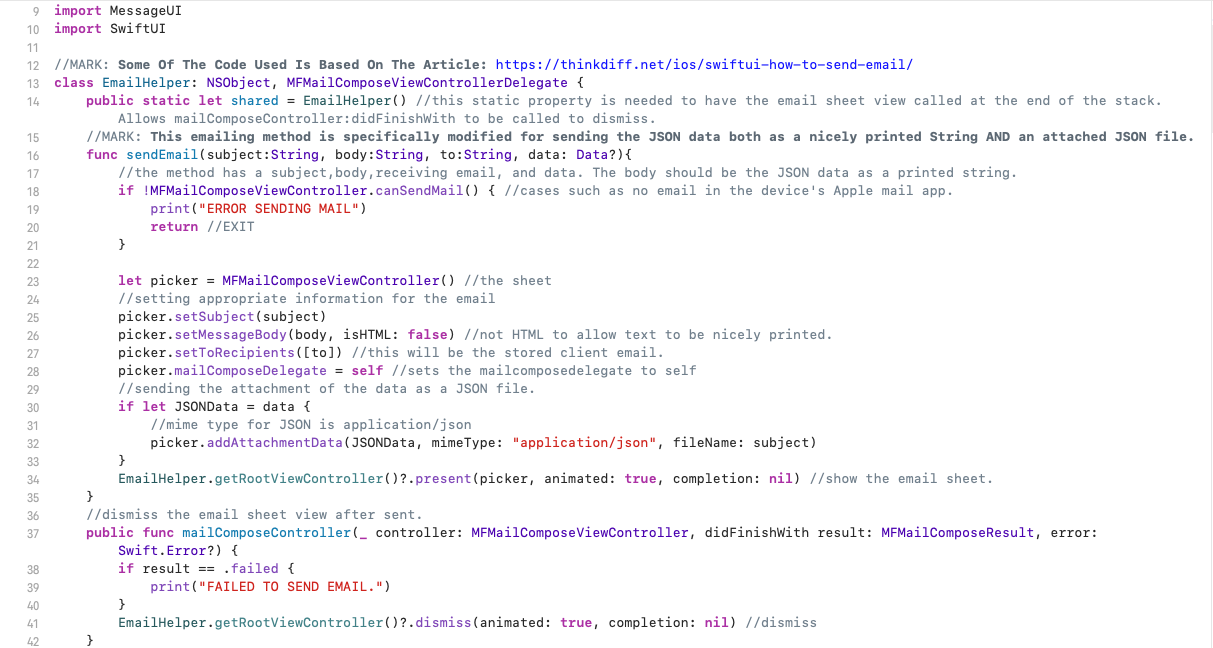
It then calls *emailHelper*, which is an *EmailHelper*.

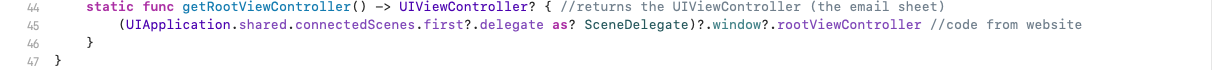
## 

## D. Send Email[[5]](#footnote-4)

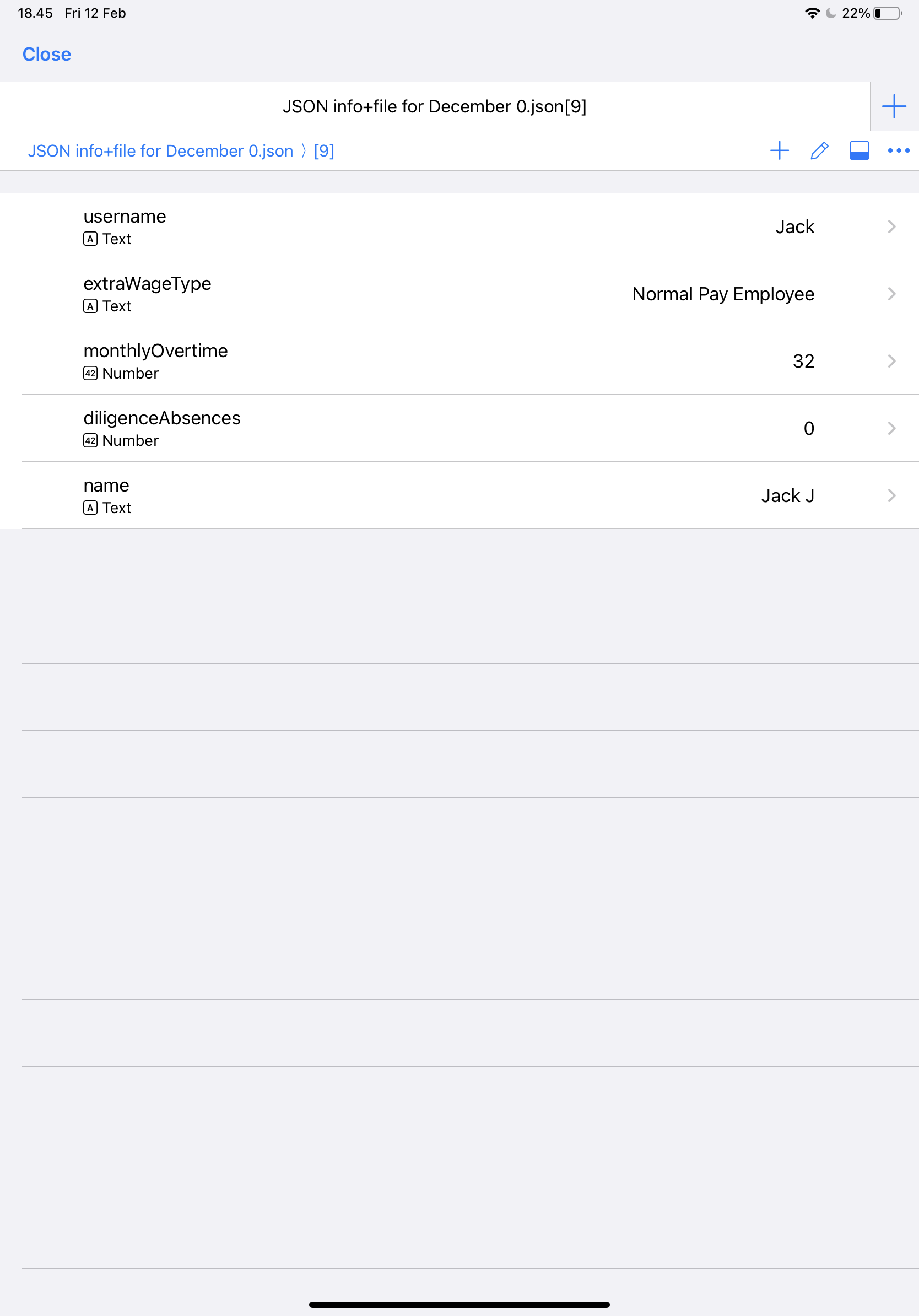
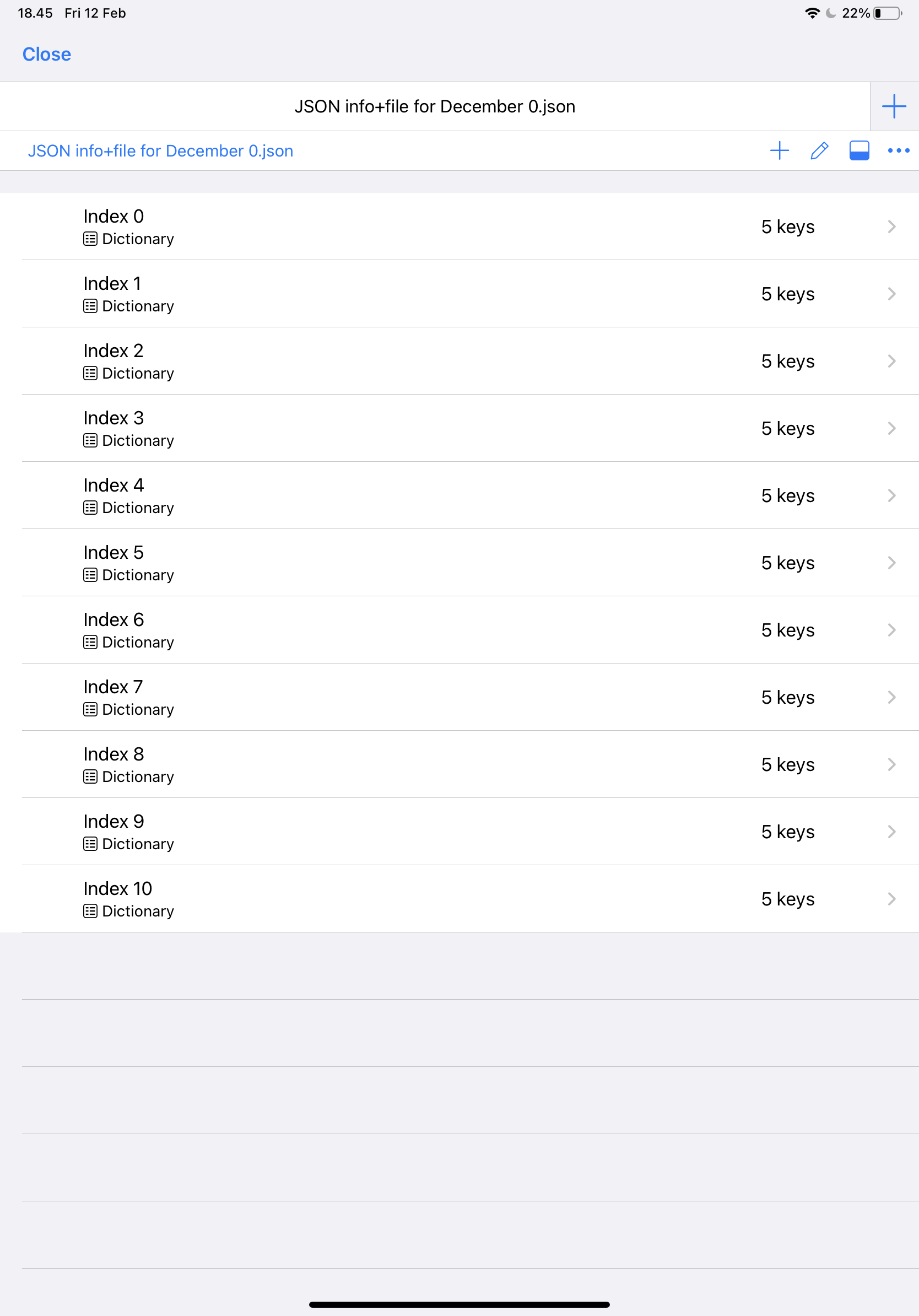


*Email sheet top view (left) and bottom view with attachment (middle), and receiving end of the email through gmail app (right).*

*Images taken as iPhone screenshots (the app must be run on an iPhone for the email function to work).*

**The *EmailHelper* sends an email with the Json data as an attached file through the *sendEmail* method and will resolve the view when the *mailComposerController* method. *jsonHandler* composes (HAS A) of an *EmailHelper* and *toBeJsonEmployee(s)*, evidence of thinking abstractly. This will bring up the email sheet as seen in the GUI screenshots above and will send the email when confirmed.

After this email is sent, Mr. Samuel can (*refer to Appendix B– Additional Details*) can use his JSON app to view the information easily:



*IPad screenshots of viewing the JSON file through a JSON viewing app (app “Jayson” was used), satisfying client’s needs.*

After he sends, the sheet will be dismissed.

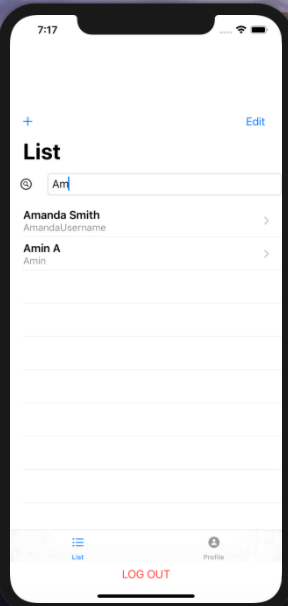
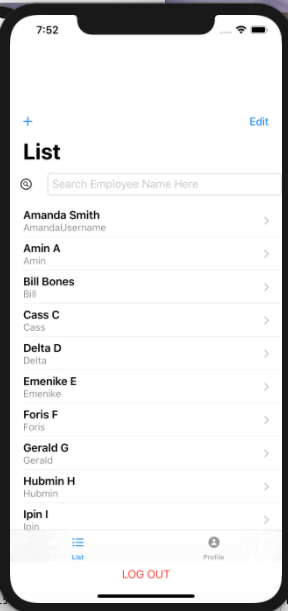
# 

# **Technique #**4**:** GUI of Lists

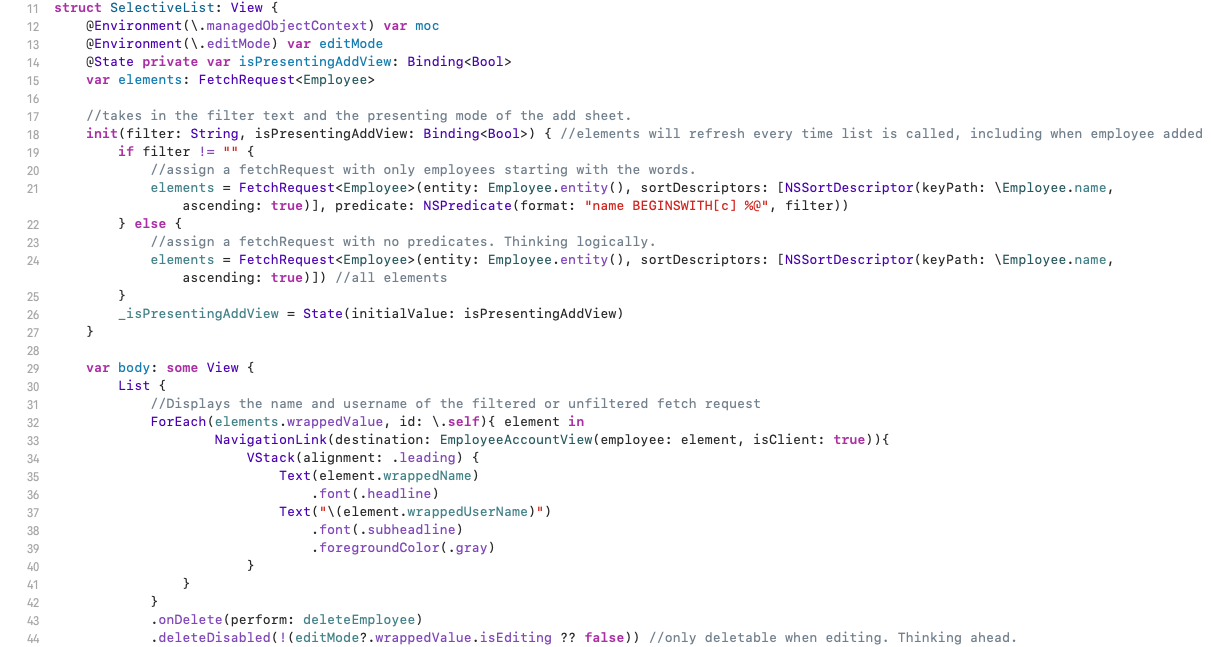
Displaying and manipulating lists

1. Sorting and searching Employee list– CRUD (adding, deleting).
2. Logs list (only clients can change).

## A. List of employees



*List of employee names unsearched (left) and searched (right).*



The *SelectiveList* will fetch employees and filter and sort based on the client’s search– thinking procedurally.

\*By using a ForEach of the filtered FetchRequest (line 32), the list will update when a name is searched or when an employee is added or deleted. Tapping on the employee name will direct the user to the employee page (line 33).\*

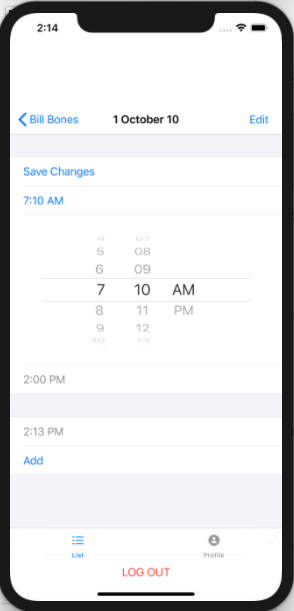
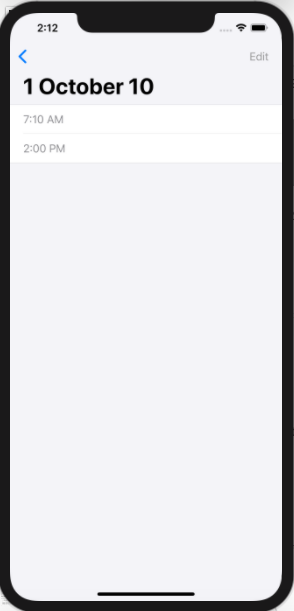
These codes allow Mr. Lordianto to view and search through a list of employees intuitively (3rd success criteria).



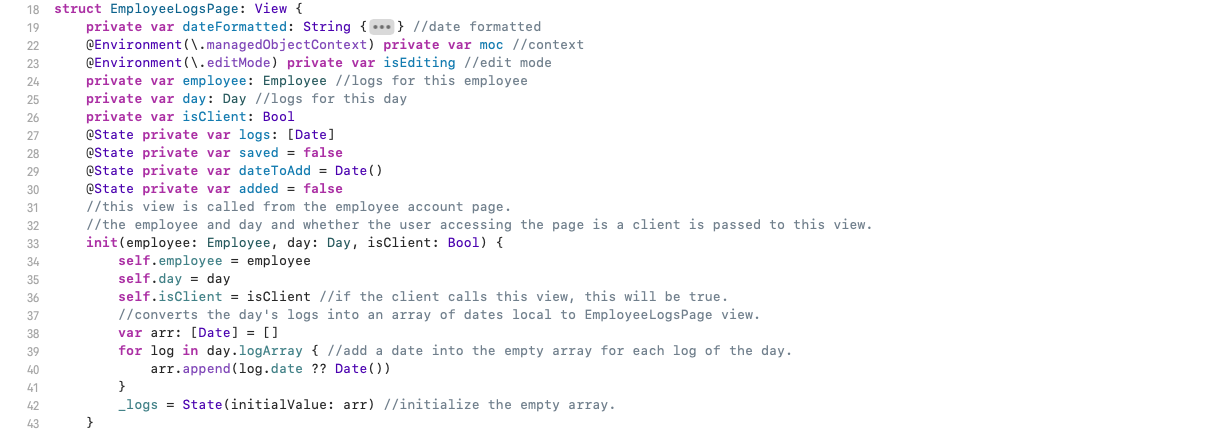
(*refer to Appendix E for complete codes*)

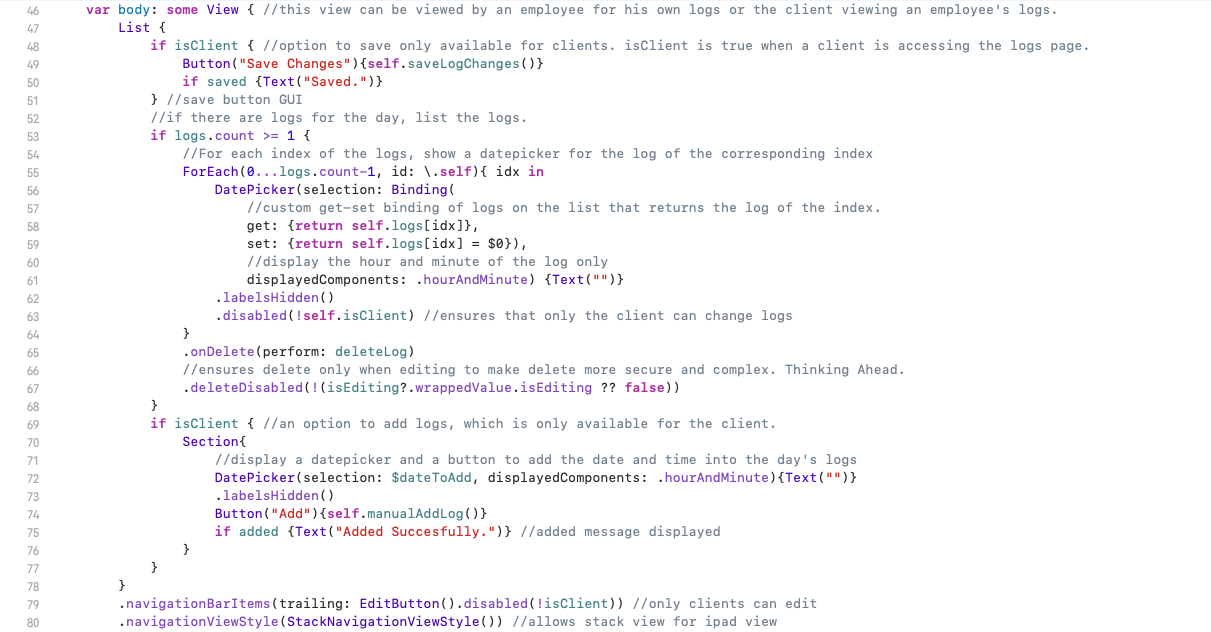
From the list, the client can add or delete an employee through *deleteEmployee* and the addView sheet, which will be saved in the context (also known as the database)– 4th success criteria.

## B. List of logs



*Logs Page for an employee’s view (left) and the client’s view–datepicker is selected (right).*





*(refer to Appendix E– List of Logs for complete codes*)

To display the list of logs, the logs of the day are converted into an array of dates private to the view (38~42). This is done so that the client can edit the dates concurrently in the list with @State while not changing the database logs before saving, evidence of thinking ahead and procedurally.

Using a boolean *isClient* variable, it is ensured that only the client can change, delete, and add logs (48, 69, 79). This allows the client to handle duplication or mistakes of employee check-in or outs, satisfying the 7th success criteria.

Word Count: 1188 words (without citations and references)

Works Cited

“100 Days of SwiftUI – Hacking with Swift.” *Www.hackingwithswift.com*, www.hackingwithswift.com/100/swiftui.

“Apple Developer Documentation | Core Data.” *Developer.apple.com*, developer.apple.com/documentation/coredata.

“Apple Developer Documentation | MessageUI.” *Developer.apple.com*, developer.apple.com/documentation/messageui. Accessed 22 Feb. 2021.

“Apple Developer Documentation | SwiftUI.” *Developer.apple.com*, developer.apple.com/documentation/swiftui/.

‌

1. Swift language and syntax was learned from and is similar to (“100 Days of SwiftUI – Hacking with Swift”) [↑](#footnote-ref-0)
2. (“Apple Developer Documentation | Core Data”) [↑](#footnote-ref-1)
3. (“Apple Developer Documentation | SwiftUI”) [↑](#footnote-ref-2)
4. (“Apple Developer Documentation | MessageUI”) [↑](#footnote-ref-3)
5. <https://thinkdiff.net/ios/swiftui-how-to-send-email/> citation [↑](#footnote-ref-4)