# *Criterion C: Development*[1]

**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]
    - @NSManagedObject subclasses to connect with the Core Data sqlite database.
    - NSManagedObjectContext to save into the database.
- SwiftUI framework[3]
    - FetchedResults<*Result*> Object to fetch data results

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

Thinking Procedurally:

A. With a device on-site (on the workplace's premises), a user logs in for check-in/check-out securely.

B. Credentials are verified through a *credentialHandler*.

    a. Errors are handled and displayed through an enumeration of *credentialPossibilities*.

C. If credentials are valid, a *logHandler* adds a new log with the timestamp of the current time into the database.

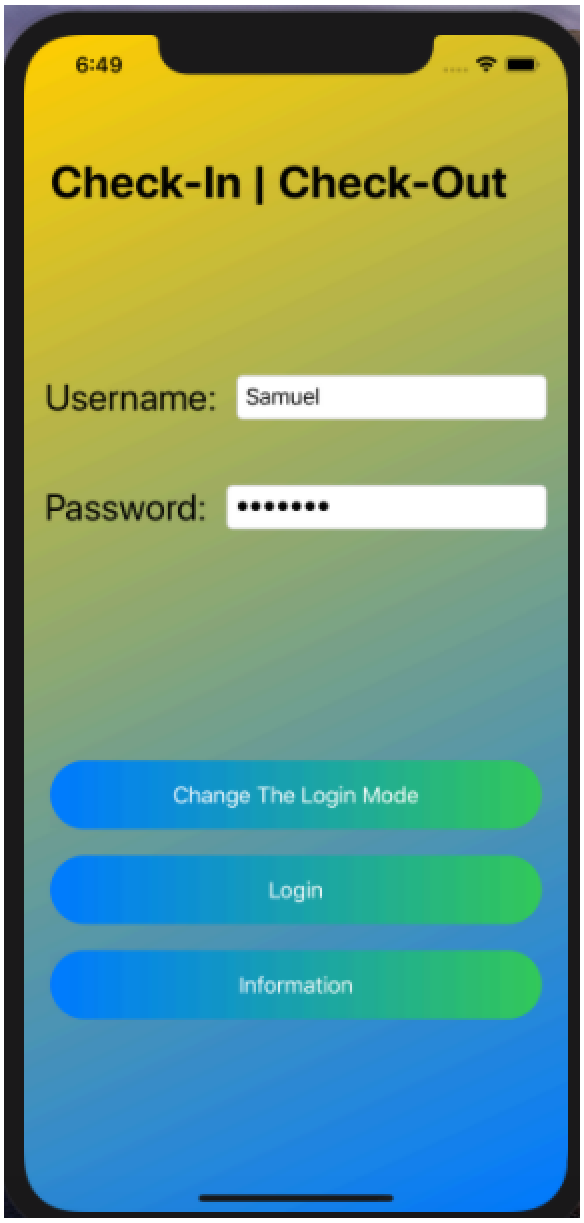
D. If the log is the first of its day, a Day entity is created. If the day is the first of its month, a Month entity is created. Otherwise, the Log will just be assigned to its corresponding Day entity.

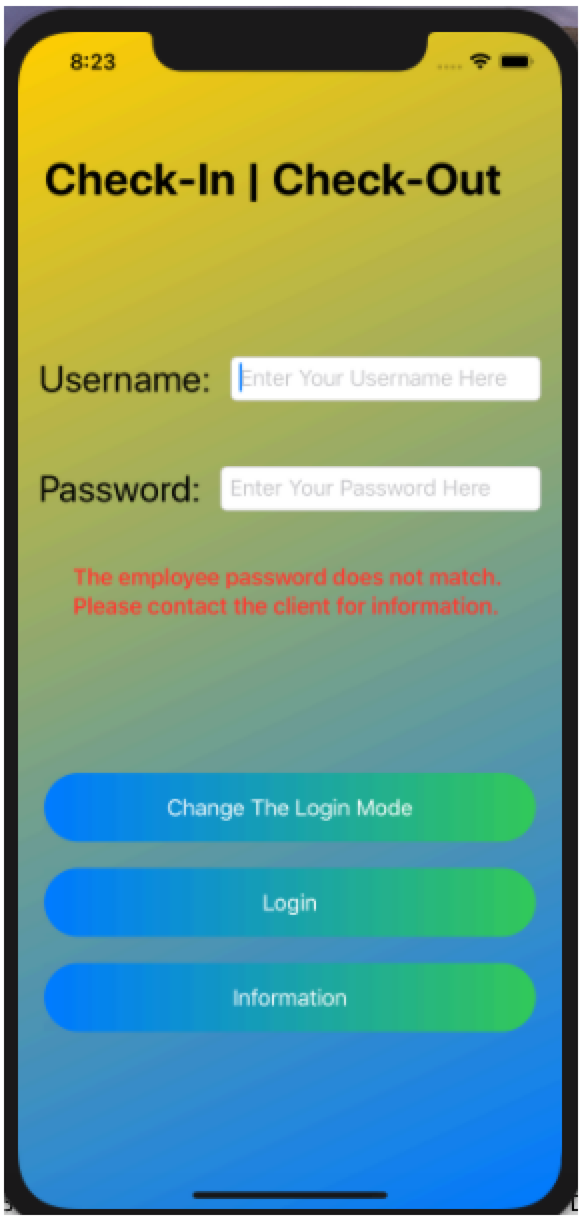
[1] Swift language and syntax was learned from and is similar to ("100 Days of SwiftUI – Hacking with Swift")

[2]("Apple Developer Documentation | Core Data")

[3]("Apple Developer Documentation | SwiftUI")

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

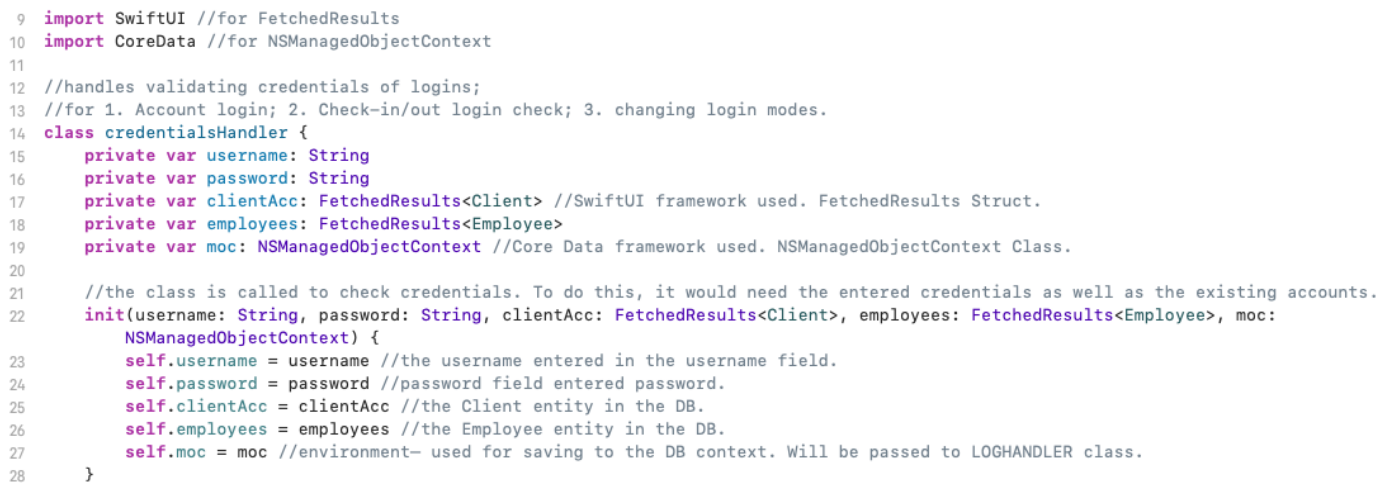




The employee password does not match.
Please contact the client for information.

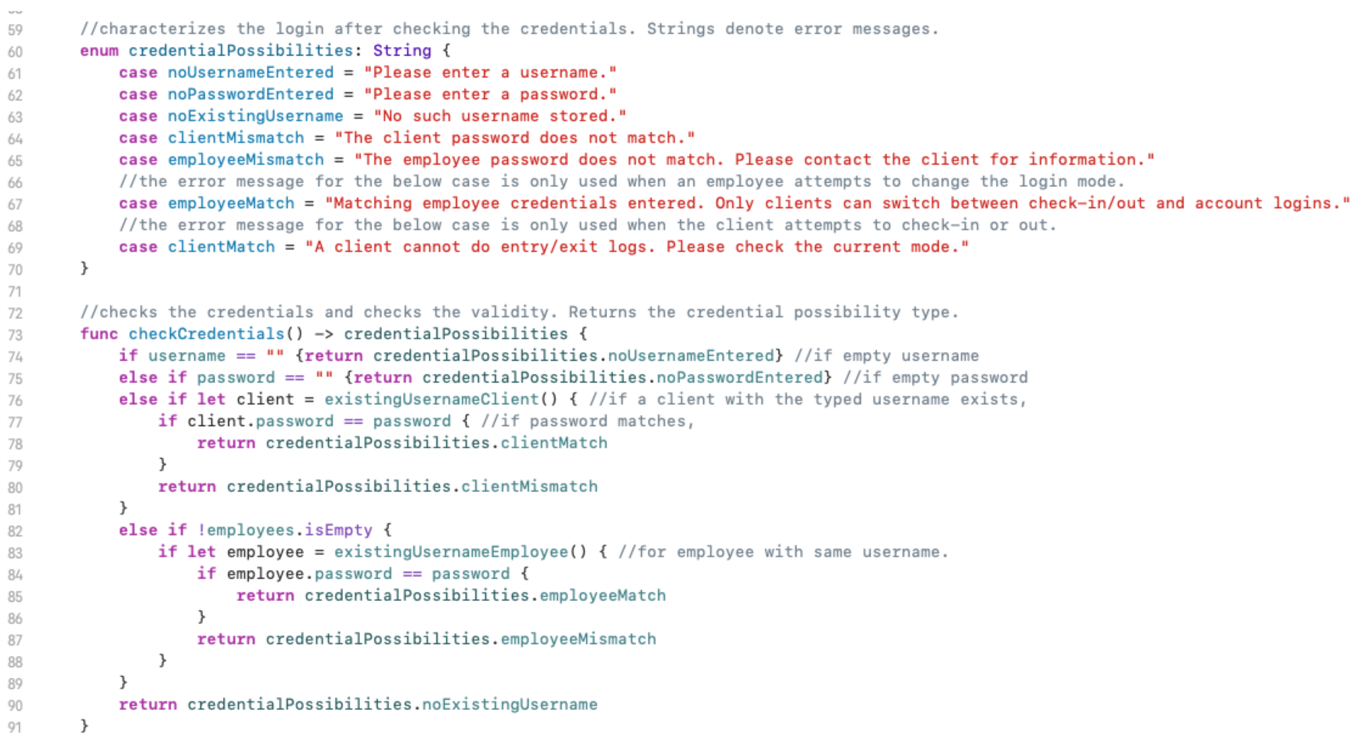
*Typed username and password (left).*
*Displayed error message of invalid employee credentials (right).*

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



```swift
import SwiftUI //for FetchedResults
import CoreData //for NSManagedObjectContext

//handles validating credentials of logins;
//for 1. Account login; 2. Check-in/out login check; 3. changing login modes.
class credentialsHandler {
    private var username: String
    private var password: String
    private var clientAcc: FetchedResults<Client> //SwiftUI framework used. FetchedResults Struct.
    private var employees: FetchedResults<Employee>
    private var moc: NSManagedObjectContext //Core Data framework used. NSManagedObjectContext Class.

    //the class is called to check credentials. To do this, it would need the entered credentials as well as the existing accounts.
    init(username: String, password: String, clientAcc: FetchedResults<Client>, employees: FetchedResults<Employee>, moc:
        NSManagedObjectContext) {
        self.username = username //the username entered in the username field.
        self.password = password //password field entered password.
        self.clientAcc = clientAcc //the Client entity in the DB.
        self.employees = employees //the Employee entity in the DB.
        self.moc = moc //environment- used for saving to the DB context. Will be passed to LOGHANDLER class.
    }
```

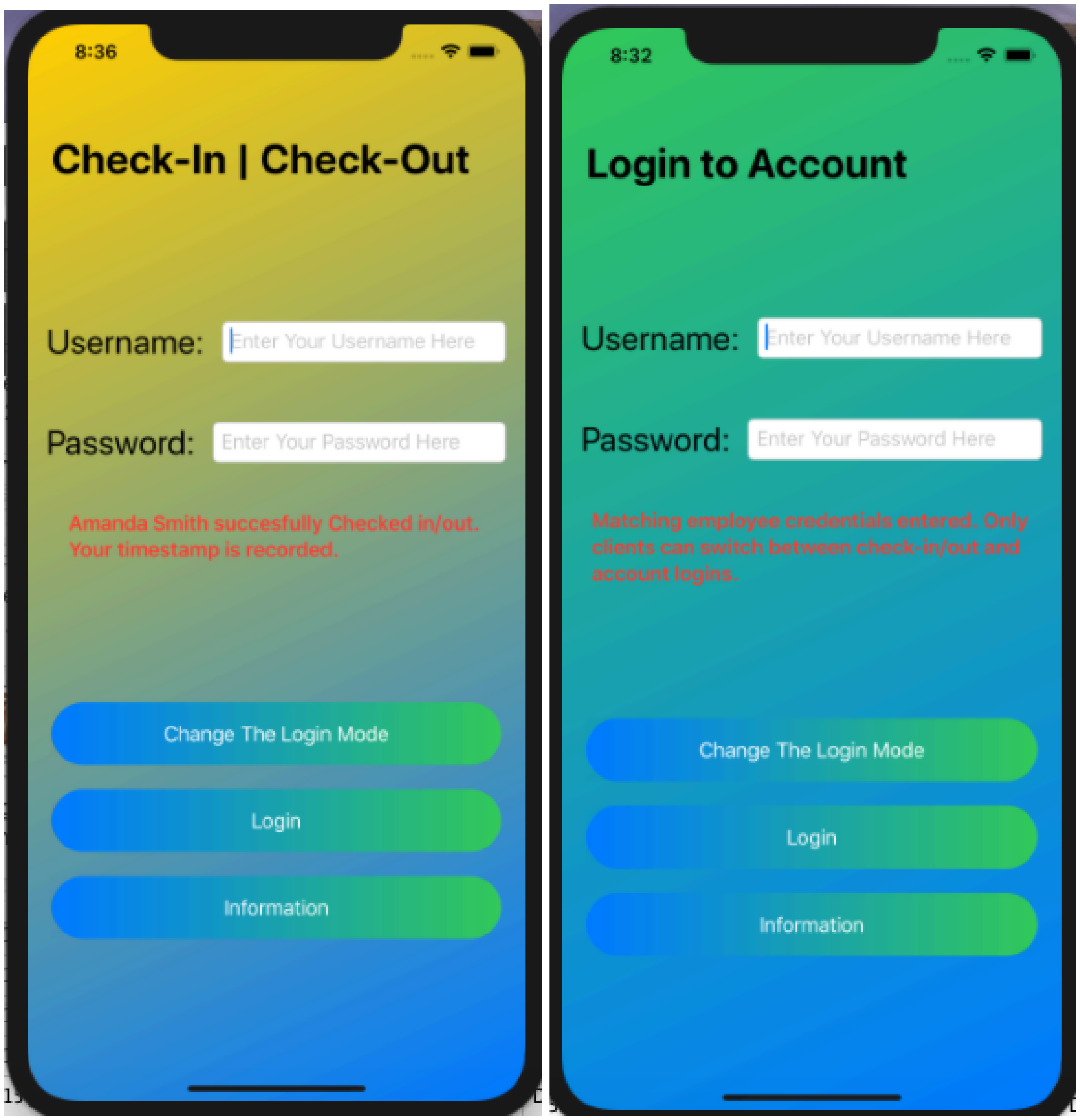
The *credentialsHandler* class handles the checking of entered credentials and returns errors if any.



```swift
    //characterizes the login after checking the credentials. Strings denote error messages.
    enum credentialPossibilities: String {
        case noUsernameEntered = "Please enter a username."
        case noPasswordEntered = "Please enter a password."
        case noExistingUsername = "No such username stored."
        case clientMismatch = "The client password does not match."
        case employeeMismatch = "The employee password does not match. Please contact the client for information."
        //the error message for the below case is only used when an employee attempts to change the login mode.
        case employeeMatch = "Matching employee credentials entered. Only clients can switch between check-in/out and account logins."
        //the error message for the below case is only used when the client attempts to check-in or out.
        case clientMatch = "A client cannot do entry/exit logs. Please check the current mode."
    }

    //checks the credentials and checks the validity. Returns the credential possibility type.
    func checkCredentials() -> credentialPossibilities {
        if username == "" {return credentialPossibilities.noUsernameEntered} //if empty username
        else if password == "" {return credentialPossibilities.noPasswordEntered} //if empty password
        else if let client = existingUsernameClient() { //if a client with the typed username exists,
            if client.password == password { //if password matches,
                return credentialPossibilities.clientMatch
            }
            return credentialPossibilities.clientMismatch
        }
        else if !employees.isEmpty {
            if let employee = existingUsernameEmployee() { //for employee with same username.
                if employee.password == password {
                    return credentialPossibilities.employeeMatch
                }
                return credentialPossibilities.employeeMismatch
            }
        }
        return credentialPossibilities.noExistingUsername
    }
```

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 (1$^{rs}$ 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).

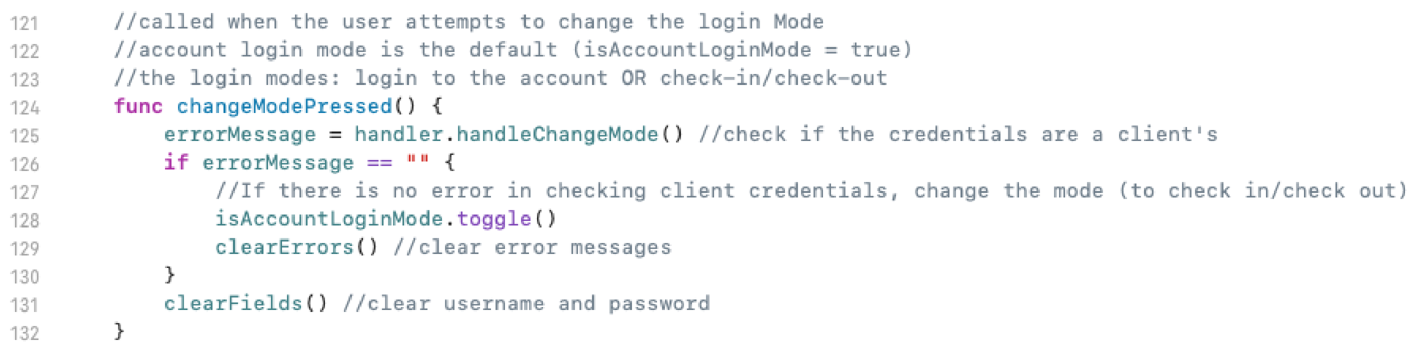


Amanda Smith succesfully Checked in/out. Your timestamp is recorded.

Matching employee credentials entered. Only clients can switch between check-in/out and account logins.

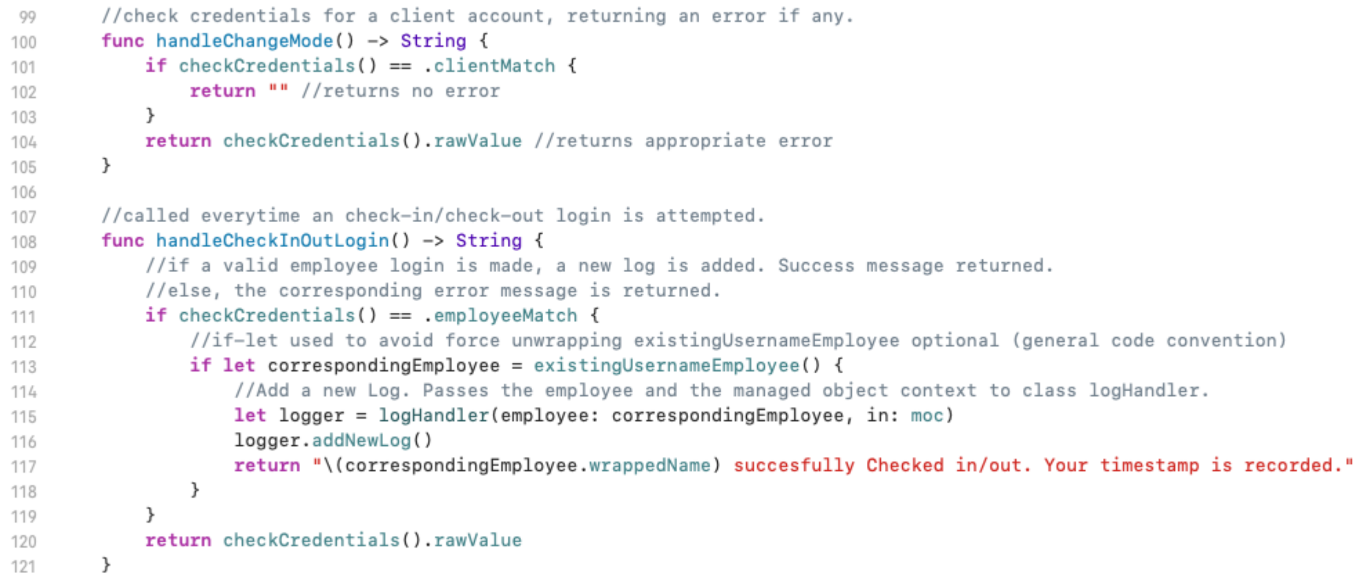
*Successful Check-in/out (left) and error message displayed when an employee attempts to change mode (right). Colors characterized by login mode (to Check-on/out or to log to account).*

*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.*



```swift
//called when the user attempts to change the login Mode
//account login mode is the default (isAccountLoginMode = true)
//the login modes: login to the account OR check-in/check-out
func changeModePressed() {
    errorMessage = handler.handleChangeMode() //check if the credentials are a client's
    if errorMessage == "" {
        //If there is no error in checking client credentials, change the mode (to check in/check out)
        isAccountLoginMode.toggle()
        clearErrors() //clear error messages
    }
    clearFields() //clear username and password
}
```

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.



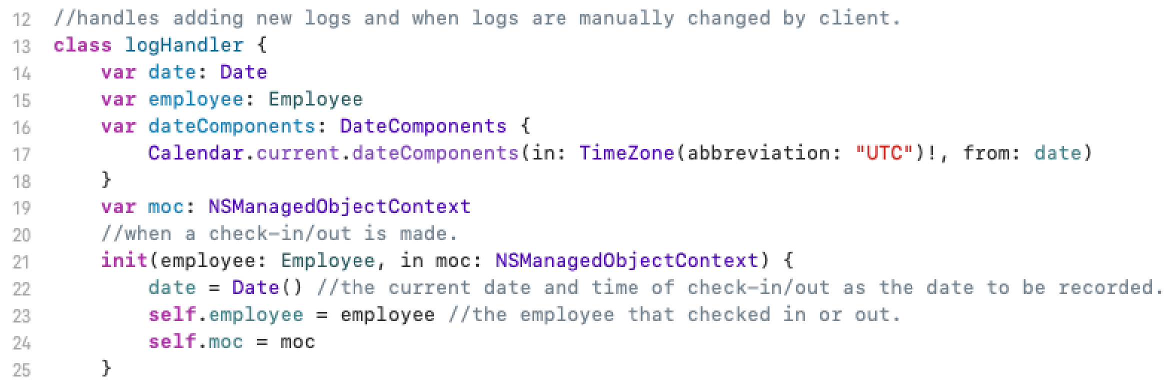
```swift
//check credentials for a client account, returning an error if any.
func handleChangeMode() -> String {
    if checkCredentials() == .clientMatch {
        return "" //returns no error
    }
    return checkCredentials().rawValue //returns appropriate error
}

//called everytime an check-in/check-out login is attempted.
func handleCheckInOutLogin() -> String {
    //if a valid employee login is made, a new log is added. Success message returned.
    //else, the corresponding error message is returned.
    if checkCredentials() == .employeeMatch {
        //if-let used to avoid force unwrapping existingUsernameEmployee optional (general code convention)
        if let correspondingEmployee = existingUsernameEmployee() {
            //Add a new Log. Passes the employee and the managed object context to class logHandler.
            let logger = logHandler(employee: correspondingEmployee, in: moc)
            logger.addNewLog()
            return "\(correspondingEmployee.wrappedName) succesfully Checked in/out. Your timestamp is recorded."
        }
    }
    return checkCredentials().rawValue
}
```

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.

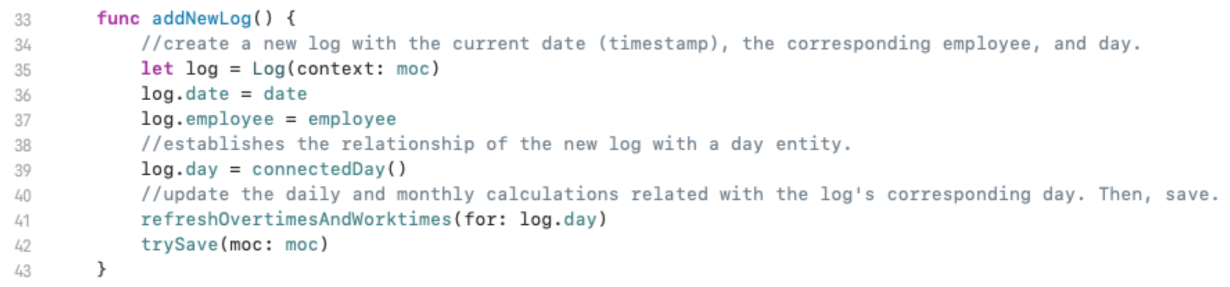
If a check-in/out is valid, a *logHandler* class handles the adding of a timestamp log.

## C. Timestamp of log and storage



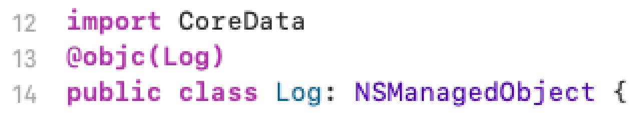
```swift
//handles adding new logs and when logs are manually changed by client.
class logHandler {
    var date: Date
    var employee: Employee
    var dateComponents: DateComponents {
        Calendar.current.dateComponents(in: TimeZone(abbreviation: "UTC")!, from: date)
    }
    var moc: NSManagedObjectContext
    //when a check-in/out is made.
    init(employee: Employee, in moc: NSManagedObjectContext) {
        date = Date() //the current date and time of check-in/out as the date to be recorded.
        self.employee = employee //the employee that checked in or out.
        self.moc = moc
    }
```

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

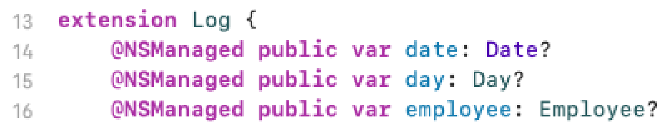


```swift
33      func addNewLog() {
34          //create a new log with the current date (timestamp), the corresponding employee, and day.
35          let log = Log(context: moc)
36          log.date = date
37          log.employee = employee
38          //establishes the relationship of the new log with a day entity.
39          log.day = connectedDay()
40          //update the daily and monthly calculations related with the log's corresponding day. Then, save.
41          refreshOvertimesAndWorktimes(for: log.day)
42          trySave(moc: moc)
43      }
```

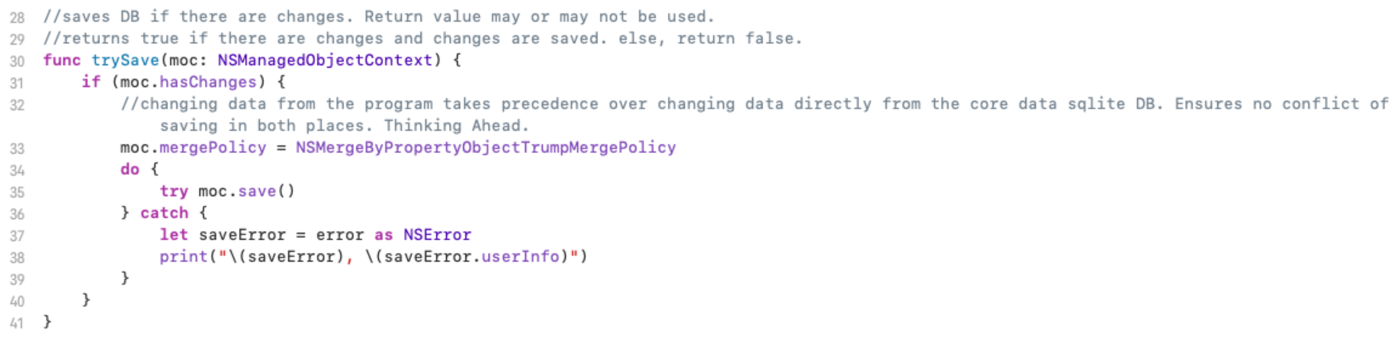
*logHandler* handles log changes. The *addNewLog* of the class allows a check-in/out timestamp to be recorded (line 35~39) and saved into the database through the *trySave* method (1rst part of the 6th success criteria). The Log's relationship to a Day is done through the *connectedDay* method.



```swift
import CoreData
@objc(Log)
public class Log: NSManagedObject {
```



```swift
extension Log {
    @NSManaged public var date: Date?
    @NSManaged public var day: Day?
    @NSManaged public var employee: Employee?
```



```swift
//saves DB if there are changes. Return value may or may not be used.
//returns true if there are changes and changes are saved. else, return false.
func trySave(moc: NSManagedObjectContext) {
    if (moc.hasChanges) {
        //changing data from the program takes precedence over changing data directly from the core data sqlite DB. Ensures no conflict of
          saving in both places. Thinking Ahead.
        moc.mergePolicy = NSMergeByPropertyObjectTrumpMergePolicy
        do {
            try moc.save()
        } catch {
            let saveError = error as NSError
            print("\(saveError), \(saveError.userInfo)")
        }
    }
}
```

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:

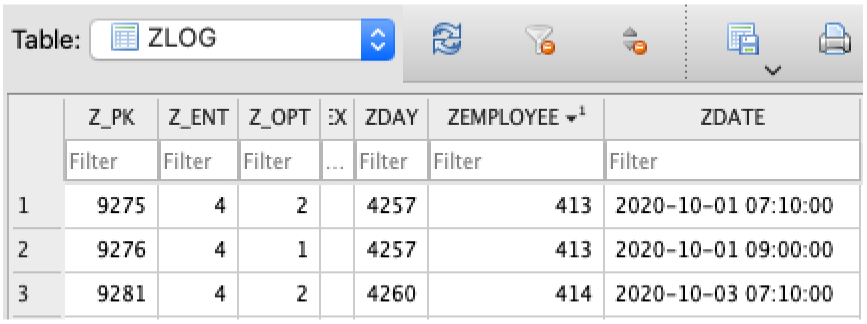


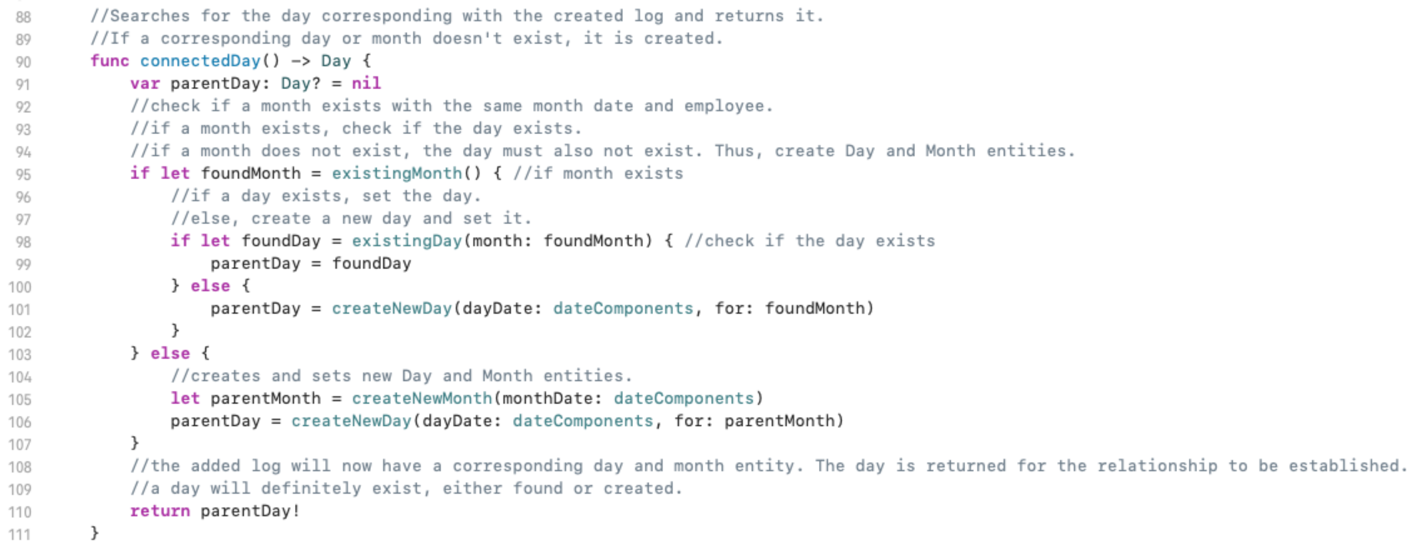
Table: ZLOG

| | Z_PK | Z_ENT | Z_OPT | EX | ZDAY | ZEMPLOYEE | ZDATE |
|---|---|---|---|---|---|---|---|
| | Filter | Filter | Filter | ... | Filter | Filter | Filter |
| 1 | 9275 | 4 | 2 | | 4257 | 413 | 2020-10-01 07:10:00 |
| 2 | 9276 | 4 | 1 | | 4257 | 413 | 2020-10-01 09:00:00 |
| 3 | 9281 | 4 | 2 | | 4260 | 414 | 2020-10-03 07:10:00 |

*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.**

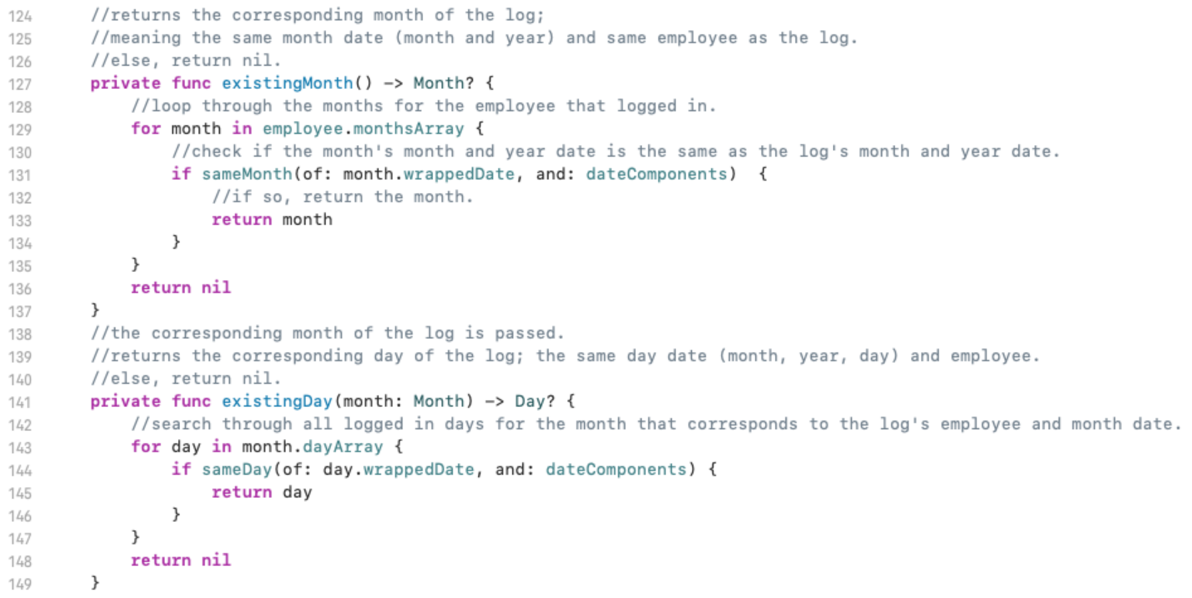
When adding a Log, its relationship with a Day is created through the *connectedDay* method (line 39 of *addNewLog*).



```
88      //Searches for the day corresponding with the created log and returns it.
89      //If a corresponding day or month doesn't exist, it is created.
90      func connectedDay() -> Day {
91          var parentDay: Day? = nil
92          //check if a month exists with the same month date and employee.
93          //if a month exists, check if the day exists.
94          //if a month does not exist, the day must also not exist. Thus, create Day and Month entities.
95          if let foundMonth = existingMonth() { //if month exists
96              //if a day exists, set the day.
97              //else, create a new day and set it.
98              if let foundDay = existingDay(month: foundMonth) { //check if the day exists
99                  parentDay = foundDay
100             } else {
101                 parentDay = createNewDay(dayDate: dateComponents, for: foundMonth)
102             }
103         } else {
104             //creates and sets new Day and Month entities.
105             let parentMonth = createNewMonth(monthDate: dateComponents)
106             parentDay = createNewDay(dayDate: dateComponents, for: parentMonth)
107         }
108         //the added log will now have a corresponding day and month entity. The day is returned for the relationship to be established.
109         //a day will definitely exist, either found or created.
110         return parentDay!
111     }
```

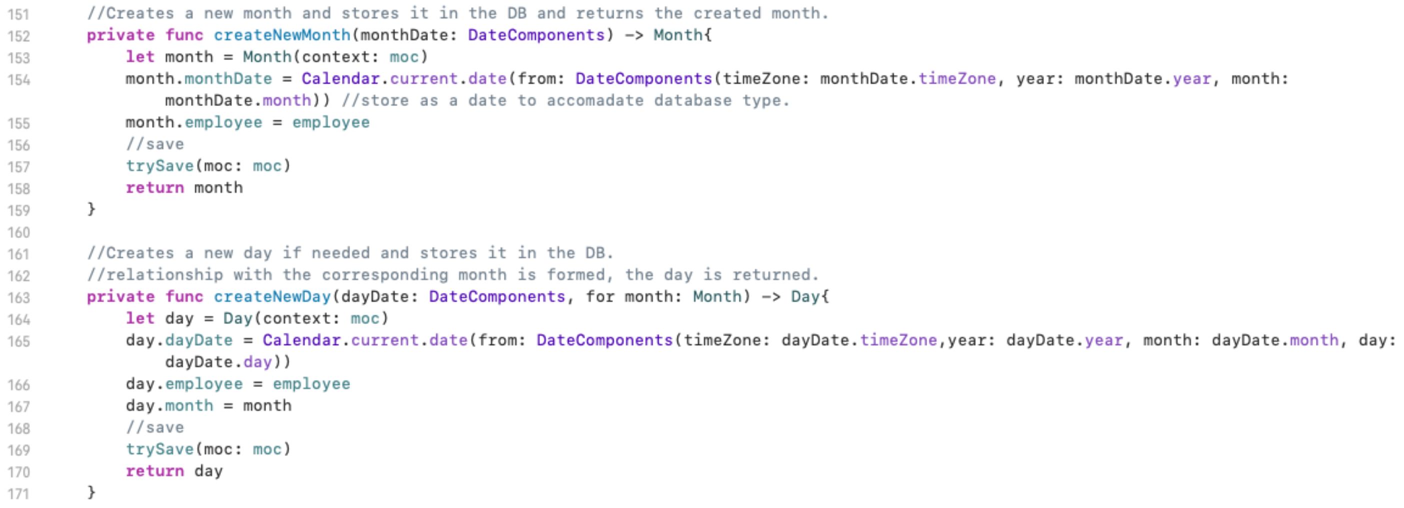
The *connectedDay* method finds the Day that should relate to the check-in/out Log while concurrently creating Month and Day entities that do not exist, hidden abstractly from the user. A nested if-let structure (95-103) is based on the logic that a Day should be checked after a Month is checked.

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).

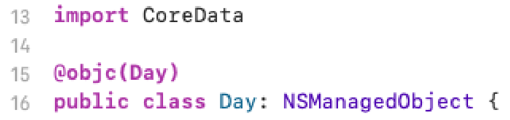


```swift
    //returns the corresponding month of the log;
    //meaning the same month date (month and year) and same employee as the log.
    //else, return nil.
    private func existingMonth() -> Month? {
        //loop through the months for the employee that logged in.
        for month in employee.monthsArray {
            //check if the month's month and year date is the same as the log's month and year date.
            if sameMonth(of: month.wrappedDate, and: dateComponents)  {
                //if so, return the month.
                return month
            }
        }
        return nil
    }
    //the corresponding month of the log is passed.
    //returns the corresponding day of the log; the same day date (month, year, day) and employee.
    //else, return nil.
    private func existingDay(month: Month) -> Day? {
        //search through all logged in days for the month that corresponds to the log's employee and month date.
        for day in month.dayArray {
            if sameDay(of: day.wrappedDate, and: dateComponents) {
                return day
            }
        }
        return nil
    }
```

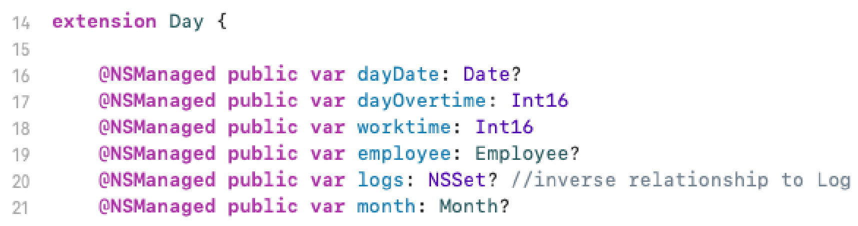
*existingMonth* and *existingDay* loop through months and days respectively to find the corresponding entity. This is done through the *sameDay* and *sameMonth* method, which cross-checks a Day/Month and a Log's month/day date and employee (*refer to Criterion B ERDs on Foreign Keys*).



```swift
    //Creates a new month and stores it in the DB and returns the created month.
    private func createNewMonth(monthDate: DateComponents) -> Month{
        let month = Month(context: moc)
        month.monthDate = Calendar.current.date(from: DateComponents(timeZone: monthDate.timeZone, year: monthDate.year, month:
            monthDate.month)) //store as a date to accomadate database type.
        month.employee = employee
        //save
        trySave(moc: moc)
        return month
    }

    //Creates a new day if needed and stores it in the DB.
    //relationship with the corresponding month is formed, the day is returned.
    private func createNewDay(dayDate: DateComponents, for month: Month) -> Day{
        let day = Day(context: moc)
        day.dayDate = Calendar.current.date(from: DateComponents(timeZone: dayDate.timeZone,year: dayDate.year, month: dayDate.month, day:
            dayDate.day))
        day.employee = employee
        day.month = month
        //save
        trySave(moc: moc)
        return day
    }
```



```swift
import CoreData

@objc(Day)
public class Day: NSManagedObject {
```



```swift
extension Day {

    @NSManaged public var dayDate: Date?
    @NSManaged public var dayOvertime: Int16
    @NSManaged public var worktime: Int16
    @NSManaged public var employee: Employee?
    @NSManaged public var logs: NSSet? //inverse relationship to Log
    @NSManaged public var month: Month?
```

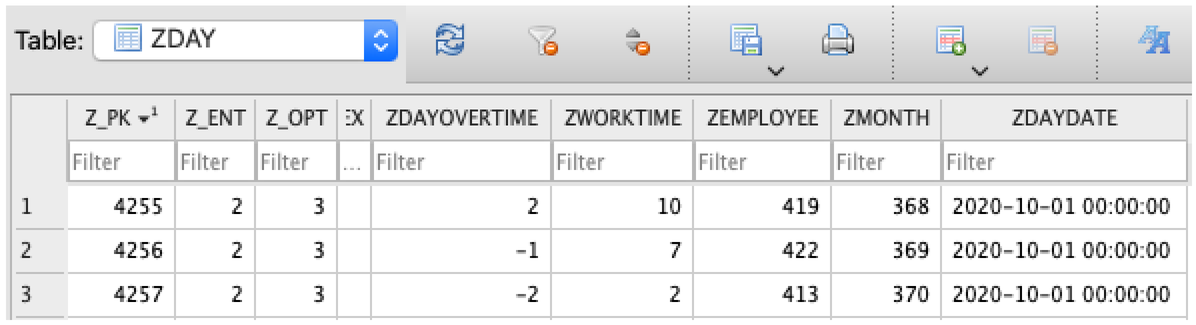


Table: ZDAY

| | Z_PK | Z_ENT | Z_OPT | EX | ZDAYOVERTIME | ZWORKTIME | ZEMPLOYEE | ZMONTH | ZDAYDATE |
|---|---|---|---|---|---|---|---|---|---|
| | Filter | Filter | Filter | ... | Filter | Filter | Filter | Filter | Filter |
| 1 | 4255 | 2 | 3 | | 2 | 10 | 419 | 368 | 2020-10-01 00:00:00 |
| 2 | 4256 | 2 | 3 | | -1 | 7 | 422 | 369 | 2020-10-01 00:00:00 |
| 3 | 4257 | 2 | 3 | | -2 | 2 | 413 | 370 | 2020-10-01 00:00:00 |

**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.

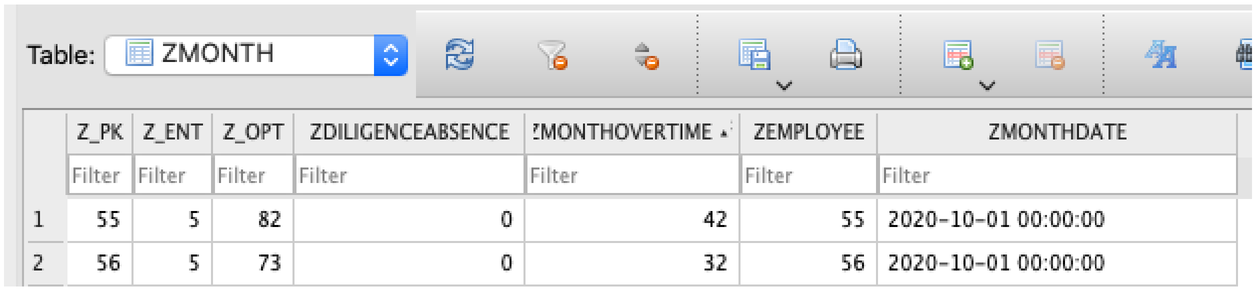


Table: ZMONTH

| | Z_PK | Z_ENT | Z_OPT | ZDILIGENCEABSENCE | ZMONTHOVERTIME | ZEMPLOYEE | ZMONTHDATE |
|---|---|---|---|---|---|---|---|
| | Filter | Filter | Filter | Filter | Filter | Filter | Filter |
| 1 | 55 | 5 | 82 | 0 | 42 | 55 | 2020-10-01 00:00:00 |
| 2 | 56 | 5 | 73 | 0 | 32 | 56 | 2020-10-01 00:00:00 |

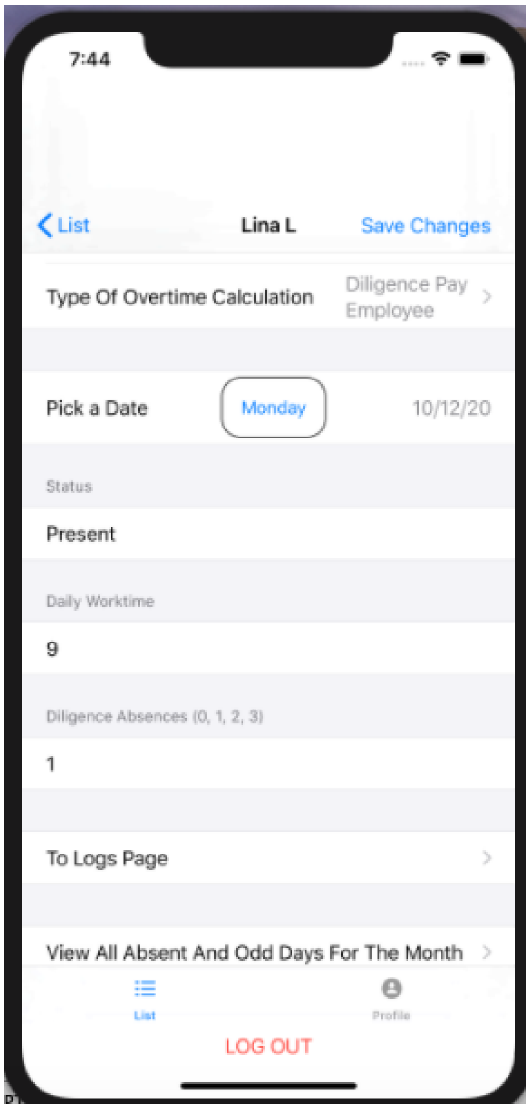
## Technique #2:

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

Thinking Procedurally:

A. When a log is added or changed, the daily worktime and, for normal pay employees, daily overtime, is calculated and capped if needed.
B. Absent days or days with odd numbers of logs are recorded.
C. 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**



7:44

List Lina L Save Changes

Type Of Overtime Calculation: Diligence Pay Employee

Pick a Date: Monday 10/12/20

Status

Present

Daily Worktime

9

Diligence Absences (0, 1, 2, 3)

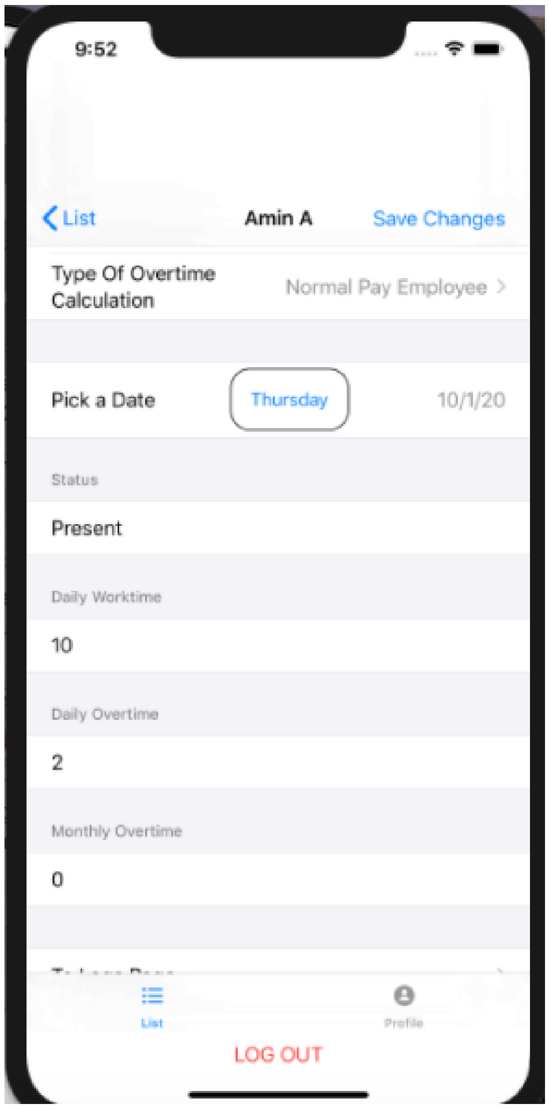
1

To Logs Page

View All Absent And Odd Days For The Month

List Profile

LOG OUT



9:52

List Amin A Save Changes

Type Of Overtime Calculation: Normal Pay Employee

Pick a Date: Thursday 10/1/20

Status

Present

Daily Worktime

10

Daily Overtime

2

Monthly Overtime

0

List Profile

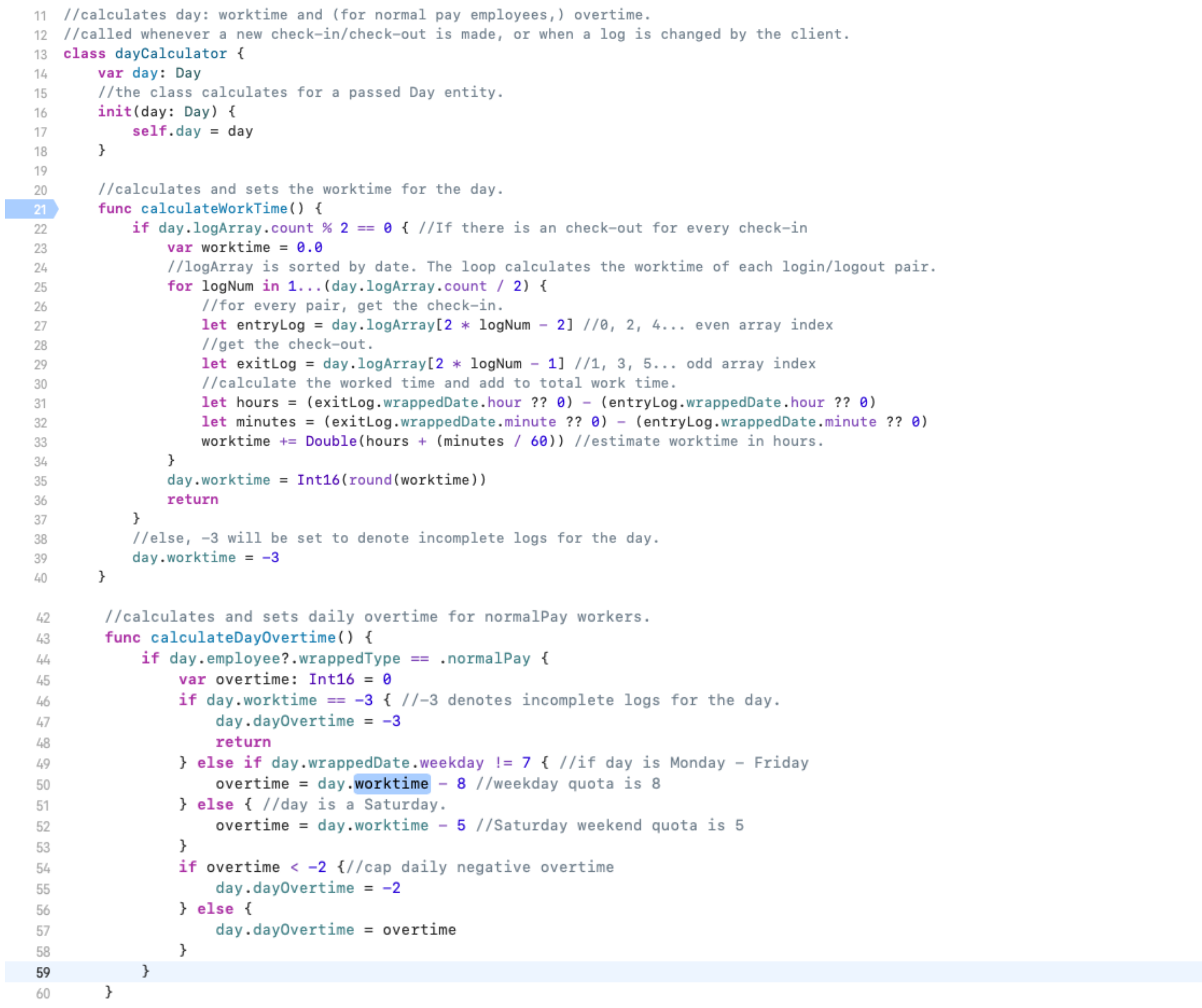
LOG OUT

*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



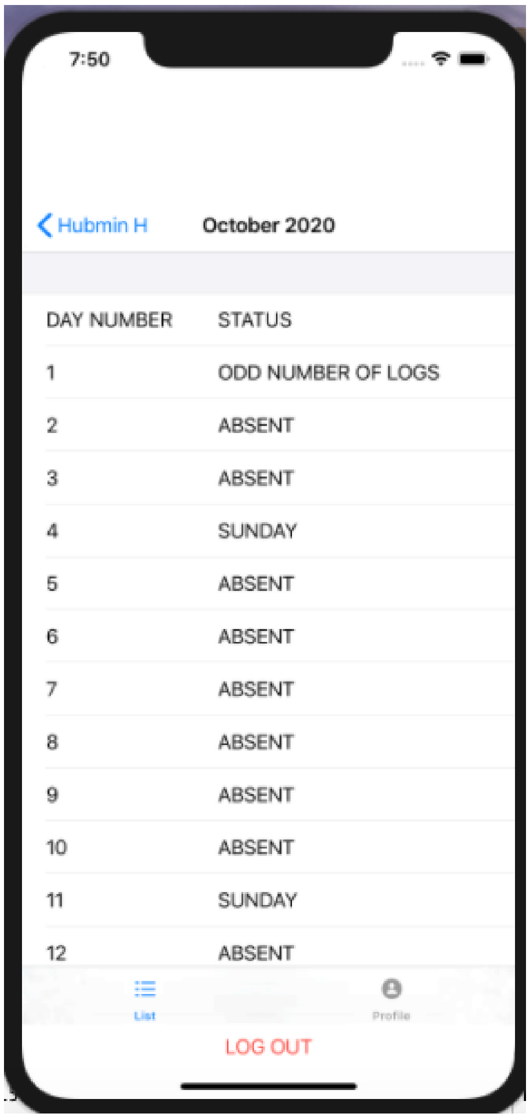
```
11  //calculates day: worktime and (for normal pay employees,) overtime.
12  //called whenever a new check-in/check-out is made, or when a log is changed by the client.
13  class dayCalculator {
14      var day: Day
15      //the class calculates for a passed Day entity.
16      init(day: Day) {
17          self.day = day
18      }
19
20      //calculates and sets the worktime for the day.
21      func calculateWorkTime() {
22          if day.logArray.count % 2 == 0 { //If there is an check-out for every check-in
23              var worktime = 0.0
24              //logArray is sorted by date. The loop calculates the worktime of each login/logout pair.
25              for logNum in 1...(day.logArray.count / 2) {
26                  //for every pair, get the check-in.
27                  let entryLog = day.logArray[2 * logNum - 2] //0, 2, 4... even array index
28                  //get the check-out.
29                  let exitLog = day.logArray[2 * logNum - 1] //1, 3, 5... odd array index
30                  //calculate the worked time and add to total work time.
31                  let hours = (exitLog.wrappedDate.hour ?? 0) - (entryLog.wrappedDate.hour ?? 0)
32                  let minutes = (exitLog.wrappedDate.minute ?? 0) - (entryLog.wrappedDate.minute ?? 0)
33                  worktime += Double(hours + (minutes / 60)) //estimate worktime in hours.
34              }
35              day.worktime = Int16(round(worktime))
36              return
37          }
38          //else, -3 will be set to denote incomplete logs for the day.
39          day.worktime = -3
40      }

42      //calculates and sets daily overtime for normalPay workers.
43      func calculateDayOvertime() {
44          if day.employee?.wrappedType == .normalPay {
45              var overtime: Int16 = 0
46              if day.worktime == -3 { //-3 denotes incomplete logs for the day.
47                  day.dayOvertime = -3
48                  return
49              } else if day.wrappedDate.weekday != 7 { //if day is Monday - Friday
50                  overtime = day.worktime - 8 //weekday quota is 8
51              } else { //day is a Saturday.
52                  overtime = day.worktime - 5 //Saturday weekend quota is 5
53              }
54              if overtime < -2 {//cap daily negative overtime
55                  day.dayOvertime = -2
56              } else {
57                  day.dayOvertime = overtime
58              }
59          }
60      }
```

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

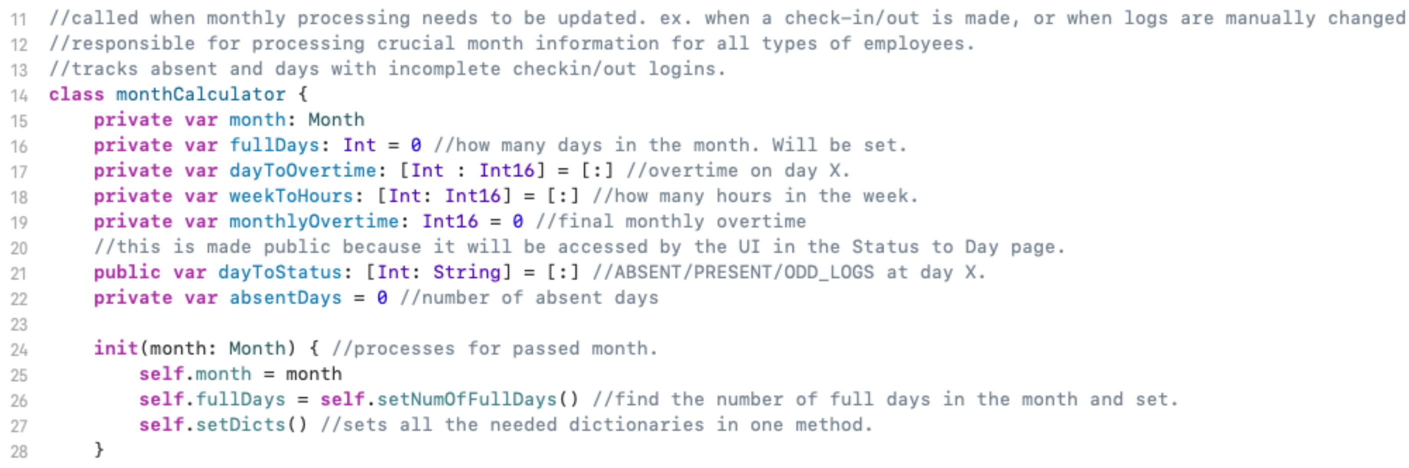
*Refer to Criterion B flowchart algorithm:* Processing for month information



| DAY NUMBER | STATUS |
|---|---|
| 1 | ODD NUMBER OF LOGS |
| 2 | ABSENT |
| 3 | ABSENT |
| 4 | SUNDAY |
| 5 | ABSENT |
| 6 | ABSENT |
| 7 | ABSENT |
| 8 | ABSENT |
| 9 | ABSENT |
| 10 | ABSENT |
| 11 | SUNDAY |
| 12 | ABSENT |

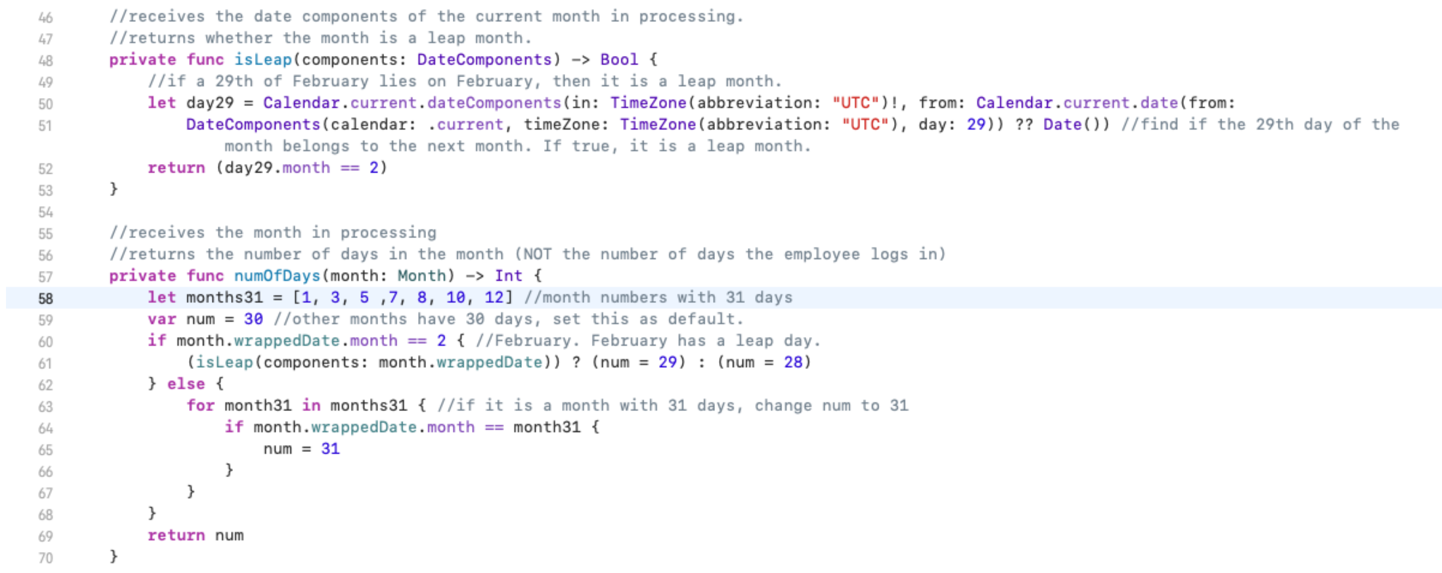
*UI view of the list of days and status. View is scrollable and continues until the end of the month– October (refer to Appendix E for continued view).*

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



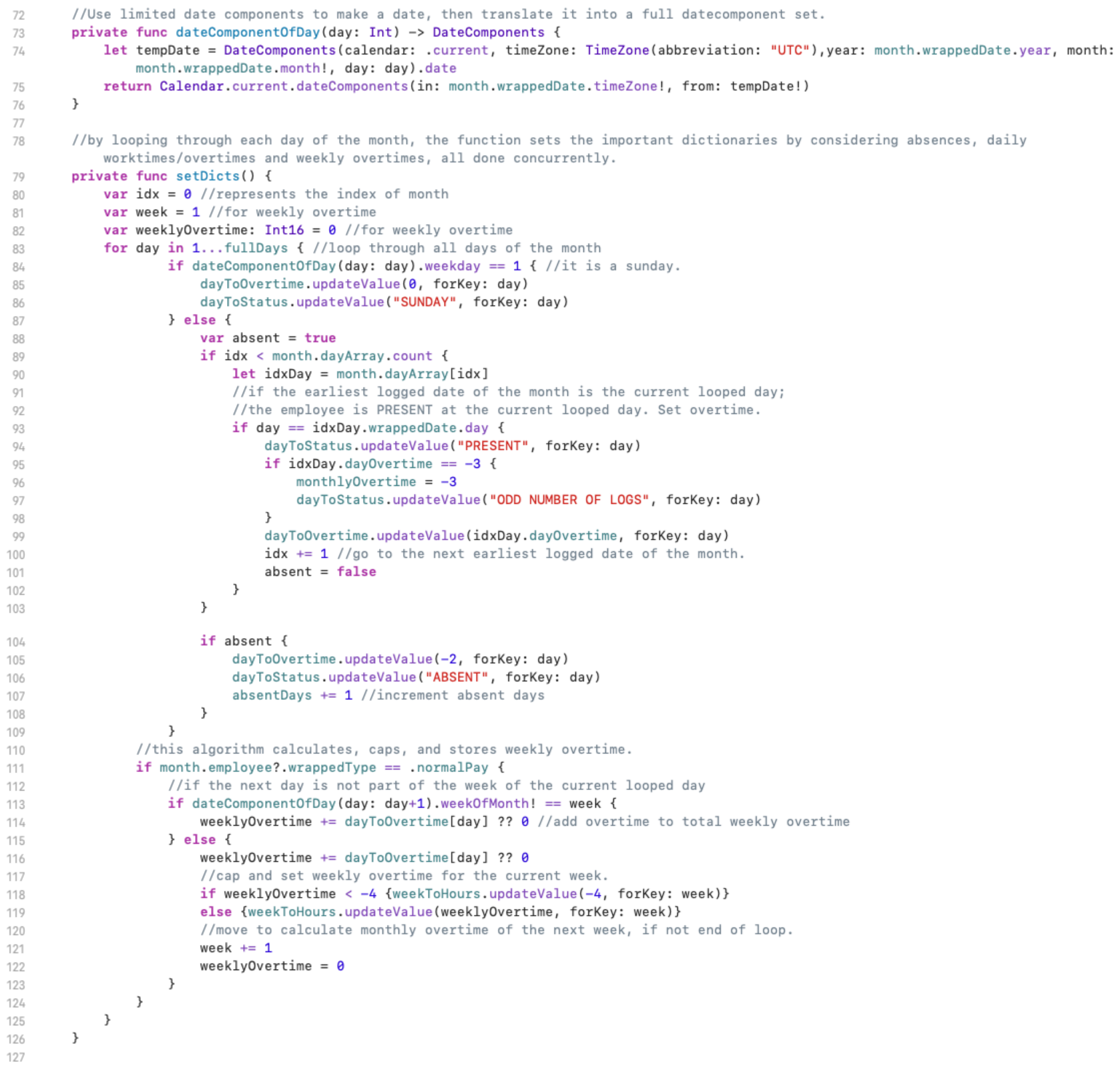
```swift
//called when monthly processing needs to be updated. ex. when a check-in/out is made, or when logs are manually changed
//responsible for processing crucial month information for all types of employees.
//tracks absent and days with incomplete checkin/out logins.
class monthCalculator {
    private var month: Month
    private var fullDays: Int = 0 //how many days in the month. Will be set.
    private var dayToOvertime: [Int : Int16] = [:] //overtime on day X.
    private var weekToHours: [Int: Int16] = [:] //how many hours in the week.
    private var monthlyOvertime: Int16 = 0 //final monthly overtime
    //this is made public because it will be accessed by the UI in the Status to Day page.
    public var dayToStatus: [Int: String] = [:] //ABSENT/PRESENT/ODD_LOGS at day X.
    private var absentDays = 0 //number of absent days

    init(month: Month) { //processes for passed month.
        self.month = month
        self.fullDays = self.setNumOfFullDays() //find the number of full days in the month and set.
        self.setDicts() //sets all the needed dictionaries in one method.
    }
```

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.



```swift
    //receives the date components of the current month in processing.
    //returns whether the month is a leap month.
    private func isLeap(components: DateComponents) -> Bool {
        //if a 29th of February lies on February, then it is a leap month.
        let day29 = Calendar.current.dateComponents(in: TimeZone(abbreviation: "UTC")!, from: Calendar.current.date(from:
            DateComponents(calendar: .current, timeZone: TimeZone(abbreviation: "UTC"), day: 29)) ?? Date()) //find if the 29th day of the
                month belongs to the next month. If true, it is a leap month.
        return (day29.month == 2)
    }

    //receives the month in processing
    //returns the number of days in the month (NOT the number of days the employee logs in)
    private func numOfDays(month: Month) -> Int {
        let months31 = [1, 3, 5 ,7, 8, 10, 12] //month numbers with 31 days
        var num = 30 //other months have 30 days, set this as default.
        if month.wrappedDate.month == 2 { //February. February has a leap day.
            (isLeap(components: month.wrappedDate)) ? (num = 29) : (num = 28)
        } else {
            for month31 in months31 { //if it is a month with 31 days, change num to 31
                if month.wrappedDate.month == month31 {
                    num = 31
                }
            }
        }
        return num
    }
```

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.



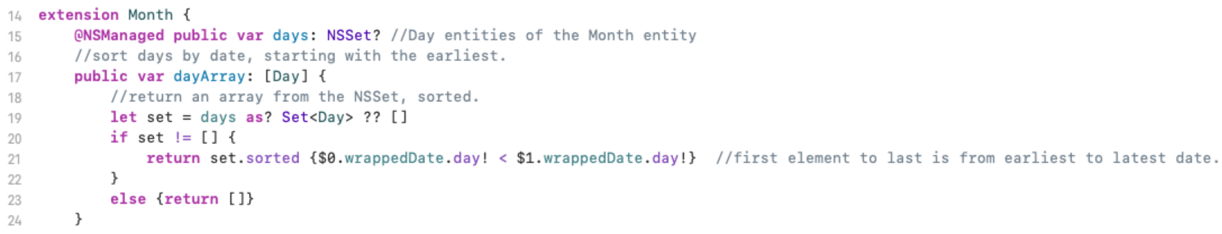
```swift
//Use limited date components to make a date, then translate it into a full datecomponent set.
private func dateComponentOfDay(day: Int) -> DateComponents {
    let tempDate = DateComponents(calendar: .current, timeZone: TimeZone(abbreviation: "UTC"),year: month.wrappedDate.year, month: month.wrappedDate.month!, day: day).date
    return Calendar.current.dateComponents(in: month.wrappedDate.timeZone!, from: tempDate!)
}

//by looping through each day of the month, the function sets the important dictionaries by considering absences, daily worktimes/overtimes and weekly overtimes, all done concurrently.
private func setDicts() {
    var idx = 0 //represents the index of month
    var week = 1 //for weekly overtime
    var weeklyOvertime: Int16 = 0 //for weekly overtime
    for day in 1...fullDays { //loop through all days of the month
            if dateComponentOfDay(day: day).weekday == 1 { //it is a sunday.
                dayToOvertime.updateValue(0, forKey: day)
                dayToStatus.updateValue("SUNDAY", forKey: day)
            } else {
                var absent = true
                if idx < month.dayArray.count {
                    let idxDay = month.dayArray[idx]
                    //if the earliest logged date of the month is the current looped day;
                    //the employee is PRESENT at the current looped day. Set overtime.
                    if day == idxDay.wrappedDate.day {
                        dayToStatus.updateValue("PRESENT", forKey: day)
                        if idxDay.dayOvertime == -3 {
                            monthlyOvertime = -3
                            dayToStatus.updateValue("ODD NUMBER OF LOGS", forKey: day)
                        }
                        dayToOvertime.updateValue(idxDay.dayOvertime, forKey: day)
                        idx += 1 //go to the next earliest logged date of the month.
                        absent = false
                    }
                }

                if absent {
                    dayToOvertime.updateValue(-2, forKey: day)
                    dayToStatus.updateValue("ABSENT", forKey: day)
                    absentDays += 1 //increment absent days
                }
            }
        //this algorithm calculates, caps, and stores weekly overtime.
        if month.employee?.wrappedType == .normalPay {
            //if the next day is not part of the week of the current looped day
            if dateComponentOfDay(day: day+1).weekOfMonth! == week {
                weeklyOvertime += dayToOvertime[day] ?? 0 //add overtime to total weekly overtime
            } else {
                weeklyOvertime += dayToOvertime[day] ?? 0
                //cap and set weekly overtime for the current week.
                if weeklyOvertime < -4 {weekToHours.updateValue(-4, forKey: week)}
                else {weekToHours.updateValue(weeklyOvertime, forKey: week)}
                //move to calculate monthly overtime of the next week, if not end of loop.
                week += 1
                weeklyOvertime = 0
            }
        }
    }
}
```

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.



```swift
extension Month {
    @NSManaged public var days: NSSet? //Day entities of the Month entity
    //sort days by date, starting with the earliest.
    public var dayArray: [Day] {
        //return an array from the NSSet, sorted.
        let set = days as? Set<Day> ?? []
        if set != [] {
            return set.sorted {$0.wrappedDate.day! < $1.wrappedDate.day!}  //first element to last is from earliest to latest date.
        }
        else {return []}
    }
```

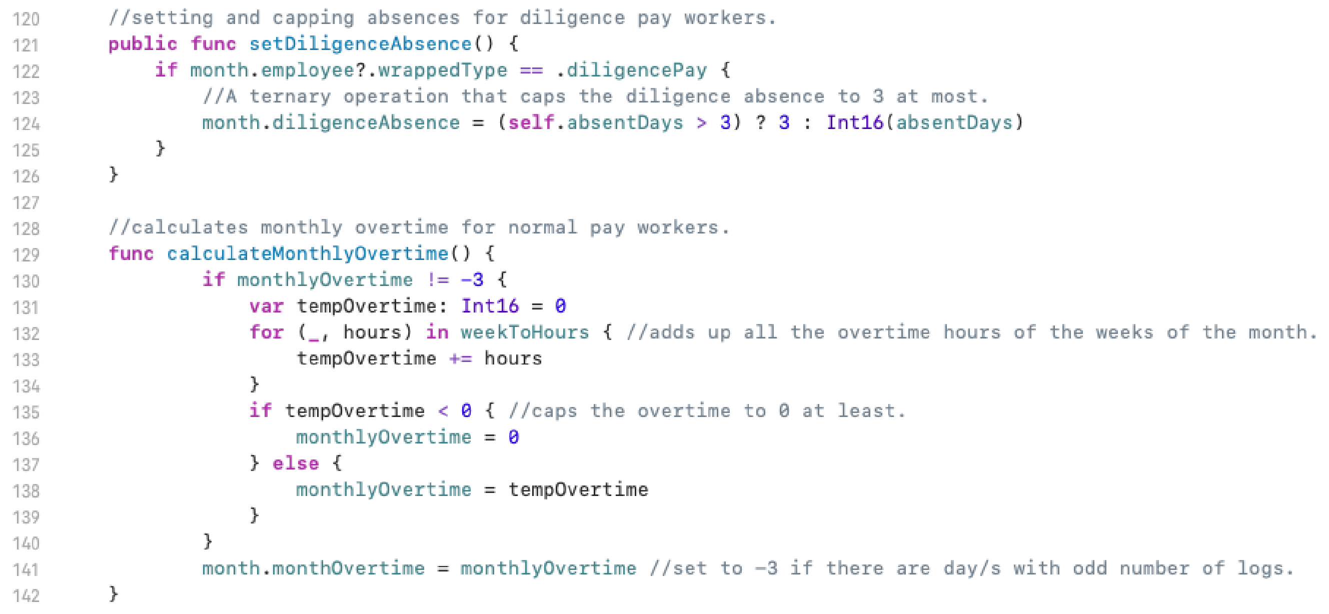
    - The *dayArray* is sorted starting from the earliest date, evidence of thinking ahead.
    - The looped day will be checked with the current front of the dayArray through a nested IF structure.
    - When they are the same, it means that the employee was present on that day. The idx increments, meaning 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:* <u>Monthly Overtime</u>
(*refer to Technique #2 GUI for the GUI display of monthly overtime and diligence absence*)



```swift
//setting and capping absences for diligence pay workers.
public func setDiligenceAbsence() {
    if month.employee?.wrappedType == .diligencePay {
        //A ternary operation that caps the diligence absence to 3 at most.
        month.diligenceAbsence = (self.absentDays > 3) ? 3 : Int16(absentDays)
    }
}

//calculates monthly overtime for normal pay workers.
func calculateMonthlyOvertime() {
        if monthlyOvertime != -3 {
            var tempOvertime: Int16 = 0
            for (_, hours) in weekToHours { //adds up all the overtime hours of the weeks of the month.
                tempOvertime += hours
            }
            if tempOvertime < 0 { //caps the overtime to 0 at least.
                monthlyOvertime = 0
            } else {
                monthlyOvertime = tempOvertime
            }
        }
        month.monthOvertime = monthlyOvertime //set to -3 if there are day/s with odd number of logs.
}
```

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]
  - 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.

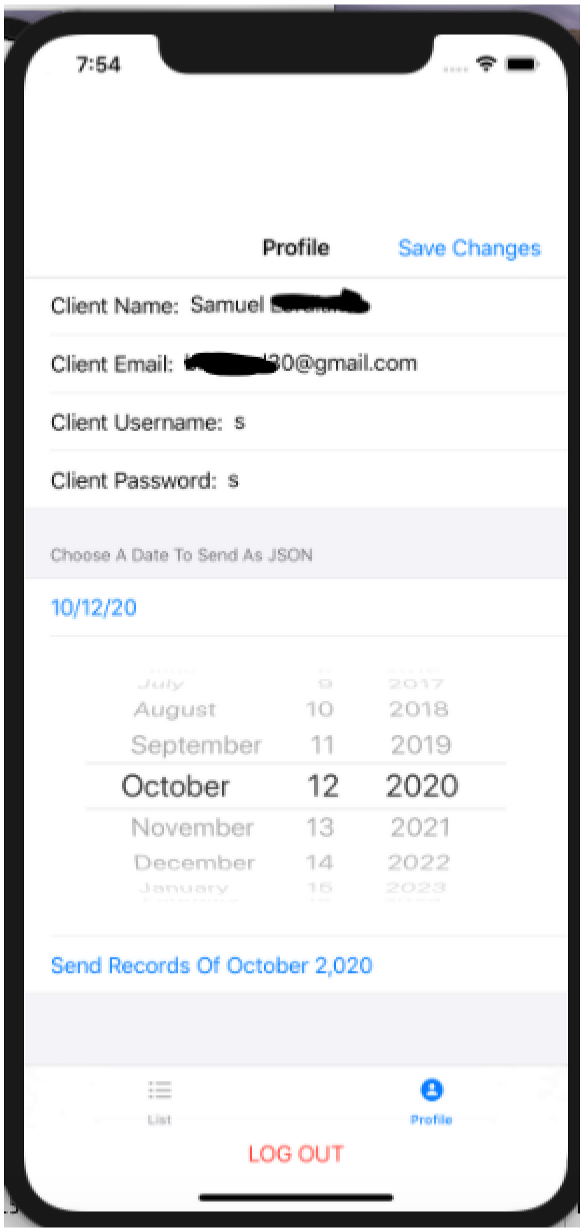
**This technique satisfies the 10th success criteria.**

Thinking Procedurally:

A. The client chooses the month date (month and year) that he wants.
B. 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.
C. Convert an array of the object into JSON data.
D. Send as an attached JSON file in an email.

[4](“Apple Developer Documentation | MessageUI”)

### A. GUI and filter months


7:54

Profile Save Changes

Client Name: Samuel [redacted]

Client Email: [redacted]0@gmail.com

Client Username: s

Client Password: s

Choose A Date To Send As JSON

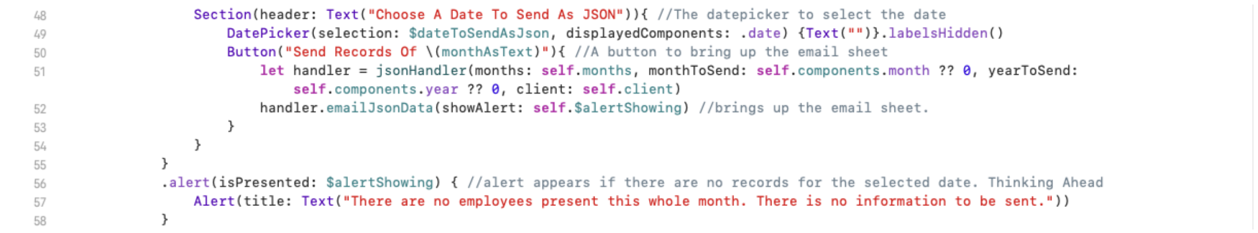
10/12/20

| | | |
|---|---|---|
| July | 9 | 2017 |
| August | 10 | 2018 |
| September | 11 | 2019 |
| October | 12 | 2020 |
| November | 13 | 2021 |
| December | 14 | 2022 |
| January | 15 | 2023 |

Send Records Of October 2,020

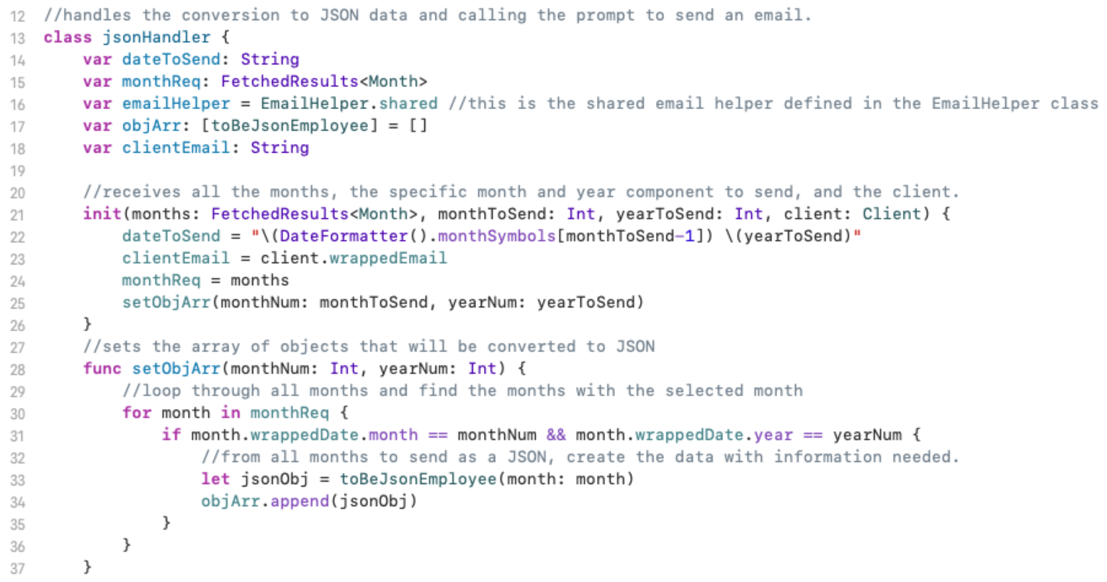
LOG OUT

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


```
                Section(header: Text("Choose A Date To Send As JSON")){ //The datepicker to select the date
                    DatePicker(selection: $dateToSendAsJson, displayedComponents: .date) {Text("")}.labelsHidden()
                    Button("Send Records Of \(monthAsText)"){ //A button to bring up the email sheet
                        let handler = jsonHandler(months: self.months, monthToSend: self.components.month ?? 0, yearToSend:
                            self.components.year ?? 0, client: self.client)
                        handler.emailJsonData(showAlert: self.$alertShowing) //brings up the email sheet.
                    }
                }
            }
            .alert(isPresented: $alertShowing) { //alert appears if there are no records for the selected date. Thinking Ahead
                Alert(title: Text("There are no employees present this whole month. There is no information to be sent."))
            }
```

*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.

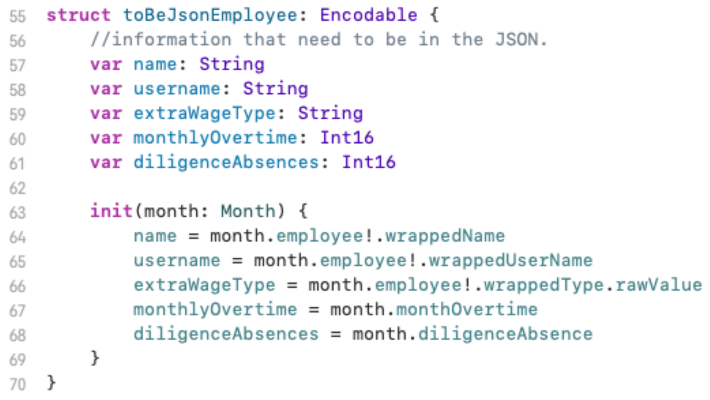


```swift
//handles the conversion to JSON data and calling the prompt to send an email.
class jsonHandler {
    var dateToSend: String
    var monthReq: FetchedResults<Month>
    var emailHelper = EmailHelper.shared //this is the shared email helper defined in the EmailHelper class
    var objArr: [toBeJsonEmployee] = []
    var clientEmail: String

    //receives all the months, the specific month and year component to send, and the client.
    init(months: FetchedResults<Month>, monthToSend: Int, yearToSend: Int, client: Client) {
        dateToSend = "\(DateFormatter().monthSymbols[monthToSend-1]) \(yearToSend)"
        clientEmail = client.wrappedEmail
        monthReq = months
        setObjArr(monthNum: monthToSend, yearNum: yearToSend)
    }
    //sets the array of objects that will be converted to JSON
    func setObjArr(monthNum: Int, yearNum: Int) {
        //loop through all months and find the months with the selected month
        for month in monthReq {
            if month.wrappedDate.month == monthNum && month.wrappedDate.year == yearNum {
                //from all months to send as a JSON, create the data with information needed.
                let jsonObj = toBeJsonEmployee(month: month)
                objArr.append(jsonObj)
            }
        }
    }
```

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. ***redacted*** 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

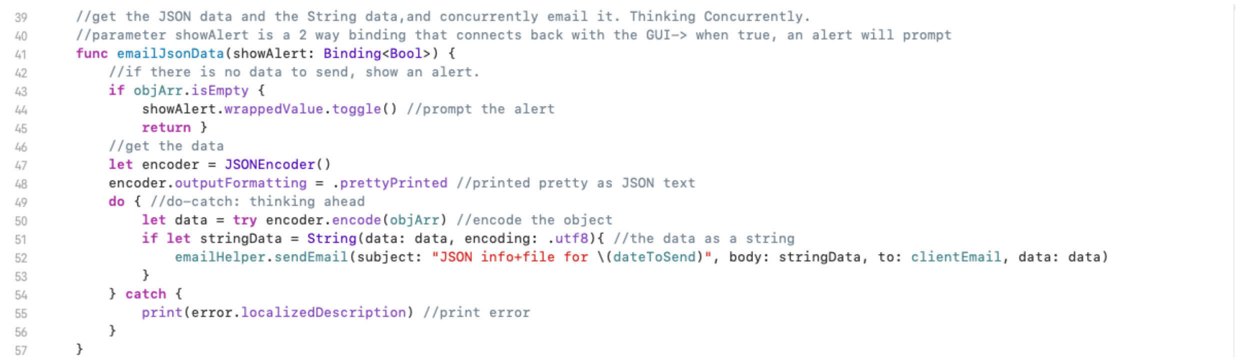


```swift
struct toBeJsonEmployee: Encodable {
    //information that need to be in the JSON.
    var name: String
    var username: String
    var extraWageType: String
    var monthlyOvertime: Int16
    var diligenceAbsences: Int16

    init(month: Month) {
        name = month.employee!.wrappedName
        username = month.employee!.wrappedUserName
        extraWageType = month.employee!.wrappedType.rawValue
        monthlyOvertime = month.monthOvertime
        diligenceAbsences = month.diligenceAbsence
    }
}
```

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

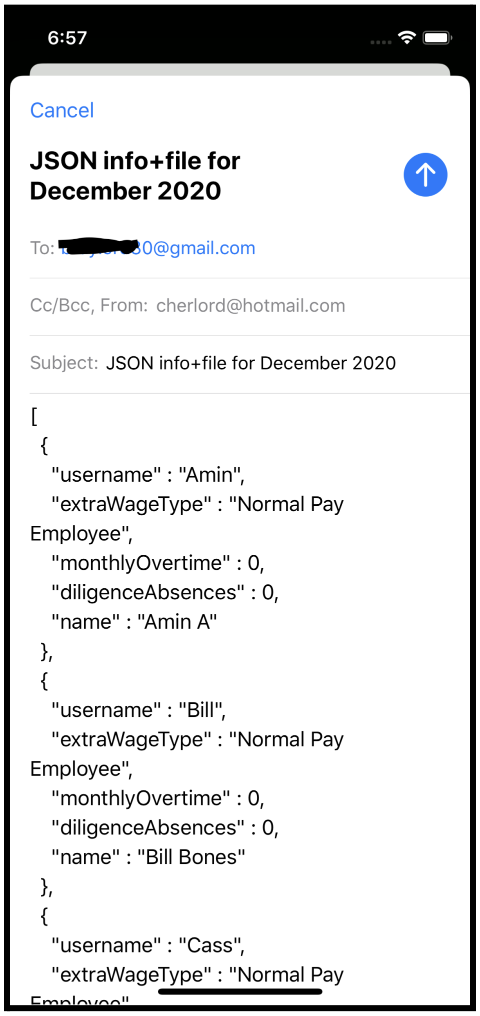


```swift
//get the JSON data and the String data,and concurrently email it. Thinking Concurrently.
//parameter showAlert is a 2 way binding that connects back with the GUI-> when true, an alert will prompt
func emailJsonData(showAlert: Binding<Bool>) {
    //if there is no data to send, show an alert.
    if objArr.isEmpty {
        showAlert.wrappedValue.toggle() //prompt the alert
        return }
    //get the data
    let encoder = JSONEncoder()
    encoder.outputFormatting = .prettyPrinted //printed pretty as JSON text
    do { //do-catch: thinking ahead
        let data = try encoder.encode(objArr) //encode the object
        if let stringData = String(data: data, encoding: .utf8){ //the data as a string
            emailHelper.sendEmail(subject: "JSON info+file for \(dateToSend)", body: stringData, to: clientEmail, data: data)
        }
    } catch {
        print(error.localizedDescription) //print error
    }
}
```

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]



6:57

Cancel

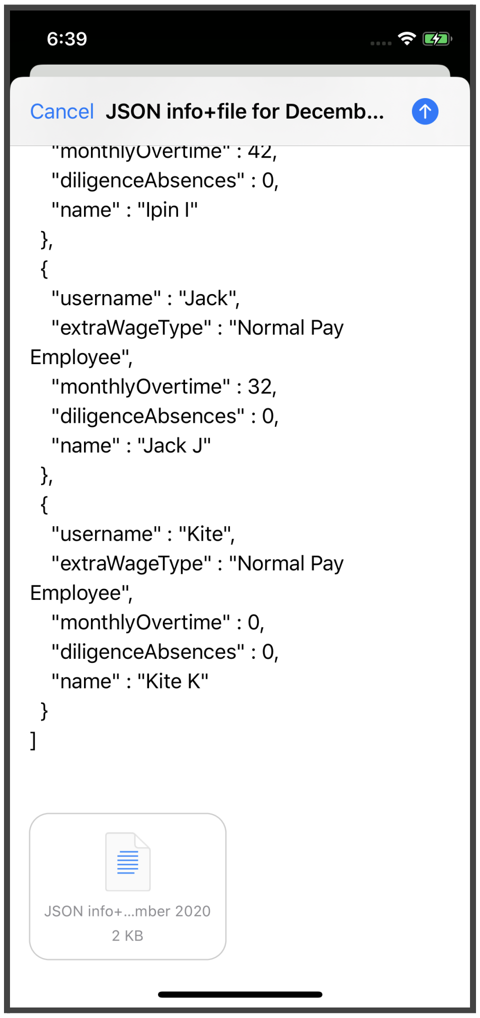
**JSON info+file for December 2020**

To: [redacted]0@gmail.com

Cc/Bcc, From: cherlord@hotmail.com

Subject: JSON info+file for December 2020

```
[
 {
   "username" : "Amin",
   "extraWageType" : "Normal Pay
Employee",
   "monthlyOvertime" : 0,
   "diligenceAbsences" : 0,
   "name" : "Amin A"
 },
 {
   "username" : "Bill",
   "extraWageType" : "Normal Pay
Employee",
   "monthlyOvertime" : 0,
   "diligenceAbsences" : 0,
   "name" : "Bill Bones"
 },
 {
   "username" : "Cass",
   "extraWageType" : "Normal Pay
```

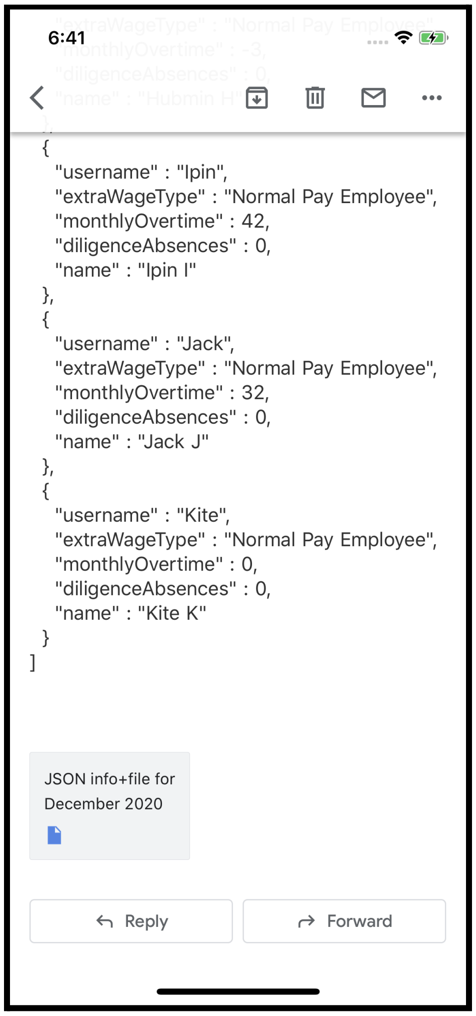


6:39

Cancel **JSON info+file for Decemb...**

```
   "monthlyOvertime" : 42,
   "diligenceAbsences" : 0,
   "name" : "Ipin I"
 },
 {
   "username" : "Jack",
   "extraWageType" : "Normal Pay
Employee",
   "monthlyOvertime" : 32,
   "diligenceAbsences" : 0,
   "name" : "Jack J"
 },
 {
   "username" : "Kite",
   "extraWageType" : "Normal Pay
Employee",
   "monthlyOvertime" : 0,
   "diligenceAbsences" : 0,
   "name" : "Kite K"
 }
]
```

JSON info+...mber 2020
2 KB



6:41

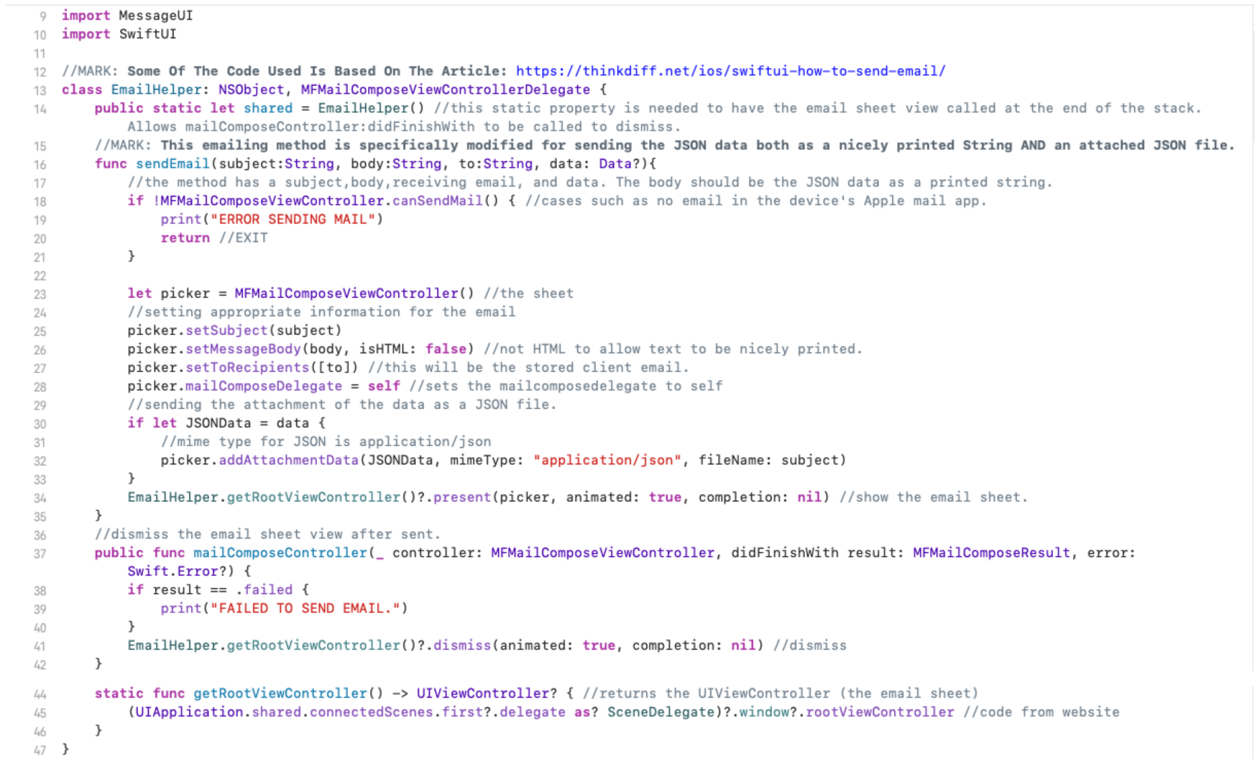
```
 {
   "username" : "Ipin",
   "extraWageType" : "Normal Pay Employee",
   "monthlyOvertime" : 42,
   "diligenceAbsences" : 0,
   "name" : "Ipin I"
 },
 {
   "username" : "Jack",
   "extraWageType" : "Normal Pay Employee",
   "monthlyOvertime" : 32,
   "diligenceAbsences" : 0,
   "name" : "Jack J"
 },
 {
   "username" : "Kite",
   "extraWageType" : "Normal Pay Employee",
   "monthlyOvertime" : 0,
   "diligenceAbsences" : 0,
   "name" : "Kite K"
 }
]
```

JSON info+file for December 2020

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

[5] https://thinkdiff.net/ios/swiftui-how-to-send-email/ citation

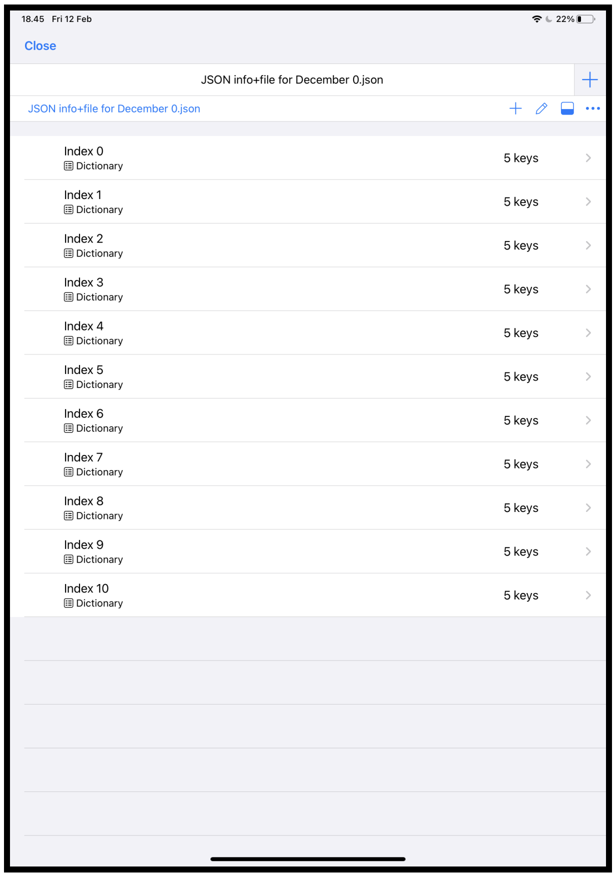
*Images taken as iPhone screenshots (the app must be run on an iPhone for the email function to work).*



```
import MessageUI
import SwiftUI

//MARK: Some Of The Code Used Is Based On The Article: https://thinkdiff.net/ios/swiftui-how-to-send-email/
class EmailHelper: NSObject, MFMailComposeViewControllerDelegate {
    public static let shared = EmailHelper() //this static property is needed to have the email sheet view called at the end of the stack.
        Allows mailComposeController:didFinishWith to be called to dismiss.
    //MARK: This emailing method is specifically modified for sending the JSON data both as a nicely printed String AND an attached JSON file.
    func sendEmail(subject:String, body:String, to:String, data: Data?){
        //the method has a subject,body,receiving email, and data. The body should be the JSON data as a printed string.
        if !MFMailComposeViewController.canSendMail() { //cases such as no email in the device's Apple mail app.
            print("ERROR SENDING MAIL")
            return //EXIT
        }

        let picker = MFMailComposeViewController() //the sheet
        //setting appropriate information for the email
        picker.setSubject(subject)
        picker.setMessageBody(body, isHTML: false) //not HTML to allow text to be nicely printed.
        picker.setToRecipients([to]) //this will be the stored client email.
        picker.mailComposeDelegate = self //sets the mailcomposedelegate to self
        //sending the attachment of the data as a JSON file.
        if let JSONData = data {
            //mime type for JSON is application/json
            picker.addAttachmentData(JSONData, mimeType: "application/json", fileName: subject)
        }
        EmailHelper.getRootViewController()?.present(picker, animated: true, completion: nil) //show the email sheet.
    }
    //dismiss the email sheet view after sent.
    public func mailComposeController(_ controller: MFMailComposeViewController, didFinishWith result: MFMailComposeResult, error:
        Swift.Error?) {
        if result == .failed {
            print("FAILED TO SEND EMAIL.")
        }
        EmailHelper.getRootViewController()?.dismiss(animated: true, completion: nil) //dismiss
    }

    static func getRootViewController() -> UIViewController? { //returns the UIViewController (the email sheet)
        (UIApplication.shared.connectedScenes.first?.delegate as? SceneDelegate)?.window?.rootViewController //code from website
    }
}
```

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. ***redacted*** can (*refer to Appendix B– Additional Details*) can use his JSON app to view the information easily:

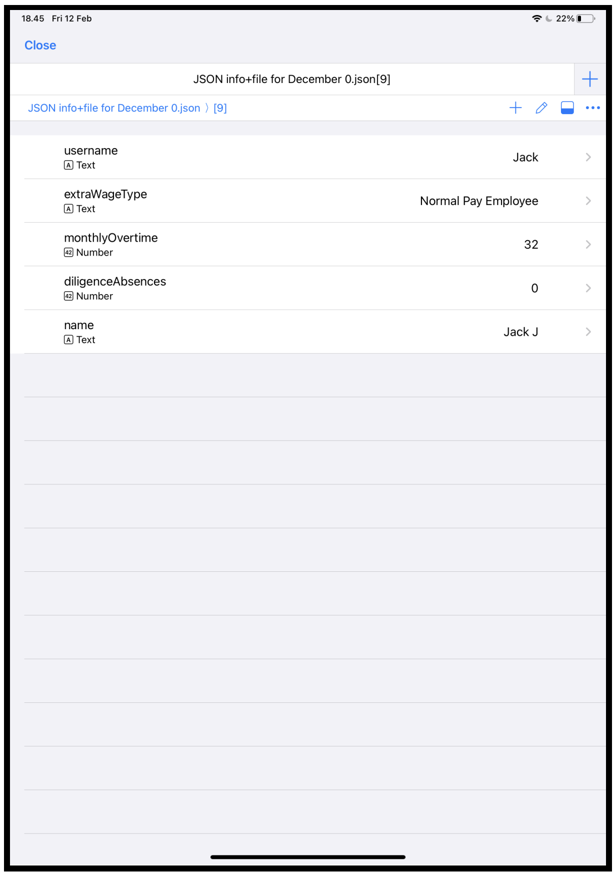

18.45 Fri 12 Feb

Close

JSON info+file for December 0.json

JSON info+file for December 0.json

| | |
|---|---|
| Index 0 Dictionary | 5 keys |
| Index 1 Dictionary | 5 keys |
| Index 2 Dictionary | 5 keys |
| Index 3 Dictionary | 5 keys |
| Index 4 Dictionary | 5 keys |
| Index 5 Dictionary | 5 keys |
| Index 6 Dictionary | 5 keys |
| Index 7 Dictionary | 5 keys |
| Index 8 Dictionary | 5 keys |
| Index 9 Dictionary | 5 keys |
| Index 10 Dictionary | 5 keys |


18.45 Fri 12 Feb

Close

JSON info+file for December 0.json[9]

JSON info+file for December 0.json 〉 [9]

| | |
|---|---|
| username Text | Jack |
| extraWageType Text | Normal Pay Employee |
| monthlyOvertime Number | 32 |
| diligenceAbsences Number | 0 |
| name Text | Jack J |

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

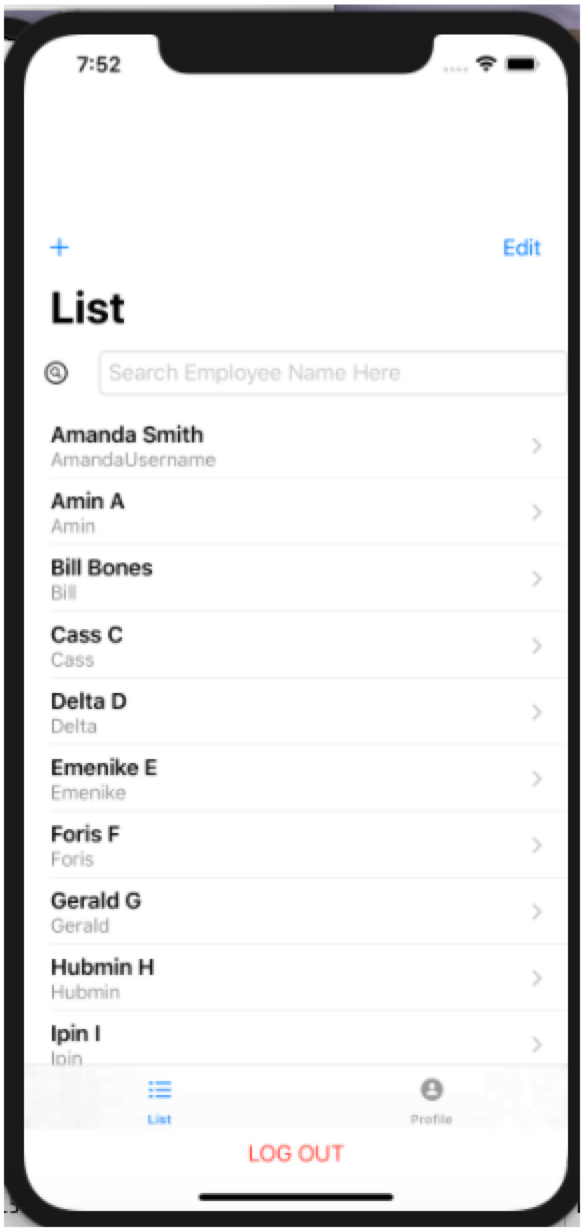
After he sends, the sheet will be dismissed.

## Technique #4: GUI of Lists

Displaying and manipulating lists

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

### A. List of employees



7:52

+ Edit

List

Search Employee Name Here

Amanda Smith
AmandaUsername

Amin A
Amin

Bill Bones
Bill

Cass C
Cass

Delta D
Delta

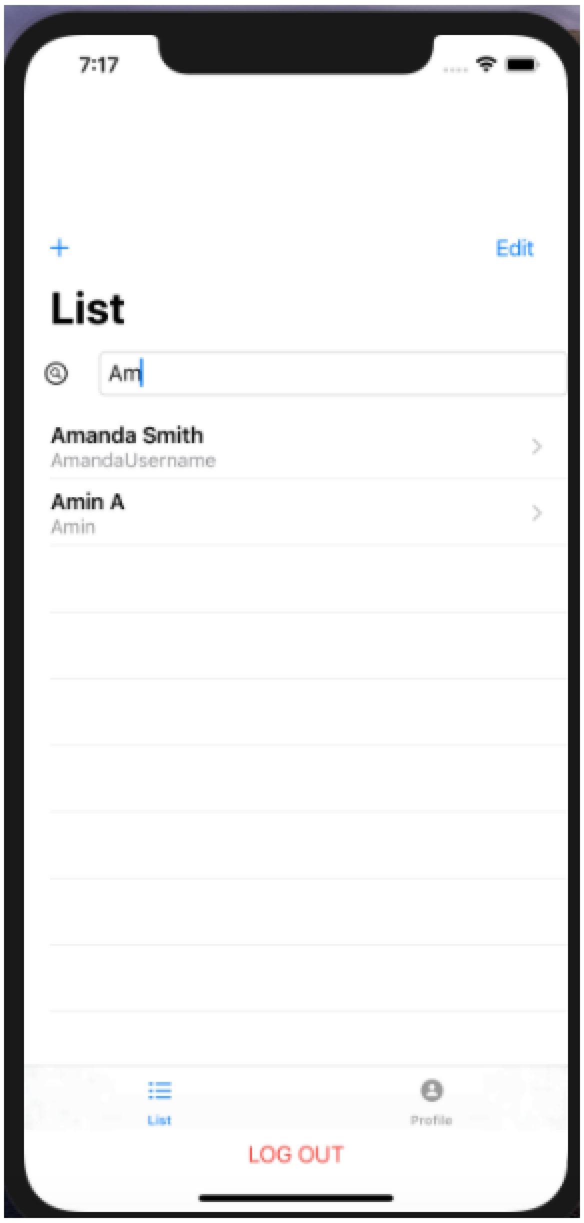
Emenike E
Emenike

Foris F
Foris

Gerald G
Gerald

Hubmin H
Hubmin

Ipin I
Ipin



7:17

+ Edit

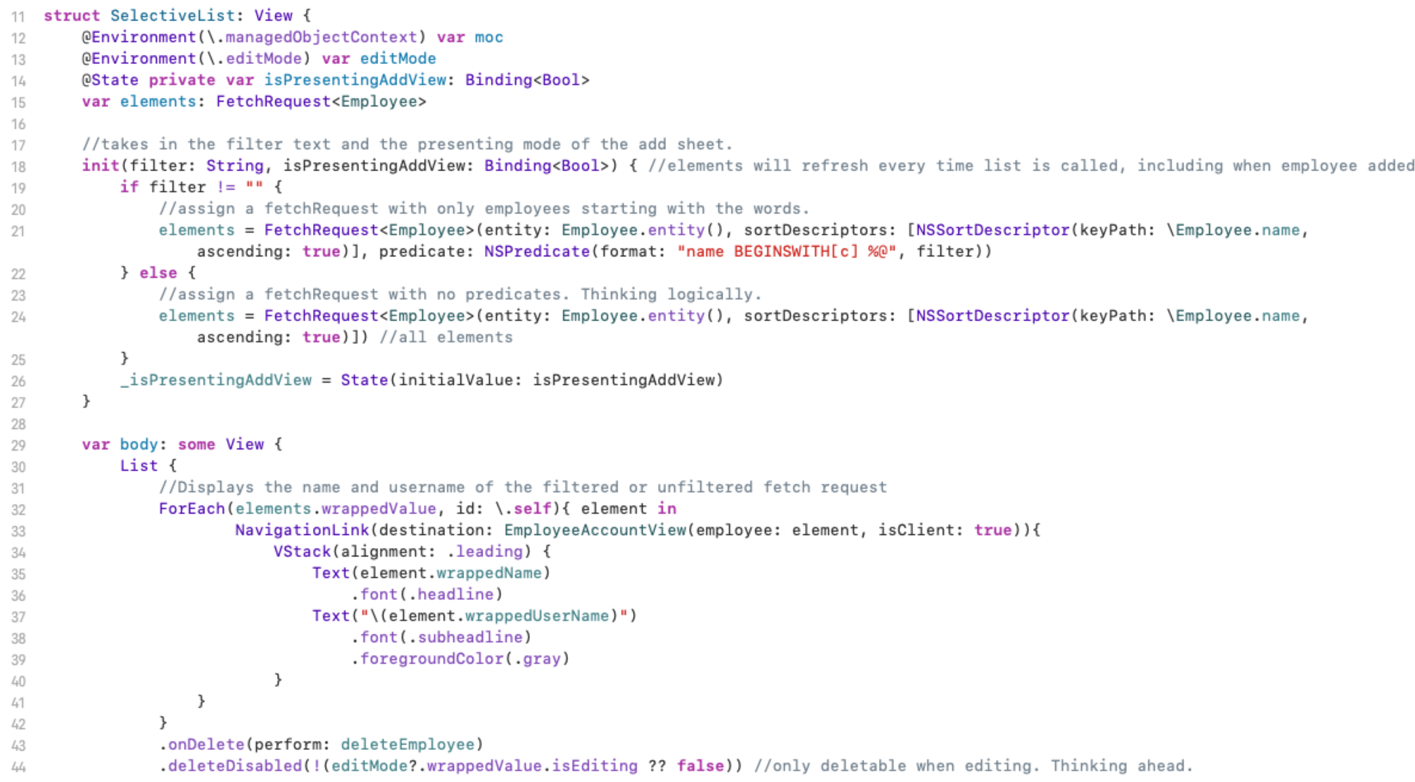
List

Am

Amanda Smith
AmandaUsername

Amin A
Amin

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

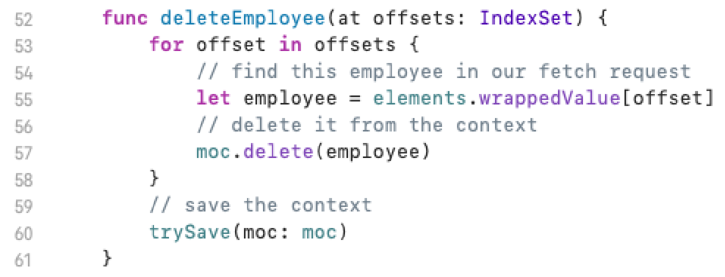


```
11  struct SelectiveList: View {
12      @Environment(\.managedObjectContext) var moc
13      @Environment(\.editMode) var editMode
14      @State private var isPresentingAddView: Binding<Bool>
15      var elements: FetchRequest<Employee>
16  
17      //takes in the filter text and the presenting mode of the add sheet.
18      init(filter: String, isPresentingAddView: Binding<Bool>) { //elements will refresh every time list is called, including when employee added
19          if filter != "" {
20              //assign a fetchRequest with only employees starting with the words.
21              elements = FetchRequest<Employee>(entity: Employee.entity(), sortDescriptors: [NSSortDescriptor(keyPath: \Employee.name,
                    ascending: true)], predicate: NSPredicate(format: "name BEGINSWITH[c] %@", filter))
22          } else {
23              //assign a fetchRequest with no predicates. Thinking logically.
24              elements = FetchRequest<Employee>(entity: Employee.entity(), sortDescriptors: [NSSortDescriptor(keyPath: \Employee.name,
                    ascending: true)]) //all elements
25          }
26          _isPresentingAddView = State(initialValue: isPresentingAddView)
27      }
28  
29      var body: some View {
30          List {
31              //Displays the name and username of the filtered or unfiltered fetch request
32              ForEach(elements.wrappedValue, id: \.self){ element in
33                      NavigationLink(destination: EmployeeAccountView(employee: element, isClient: true)){
34                          VStack(alignment: .leading) {
35                              Text(element.wrappedName)
36                                  .font(.headline)
37                              Text("\(element.wrappedUserName)")
38                                  .font(.subheadline)
39                                  .foregroundColor(.gray)
40                          }
41                  }
42              }
43              .onDelete(perform: deleteEmployee)
44              .deleteDisabled(!(editMode?.wrappedValue.isEditing ?? false)) //only deletable when editing. Thinking ahead.
```

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).

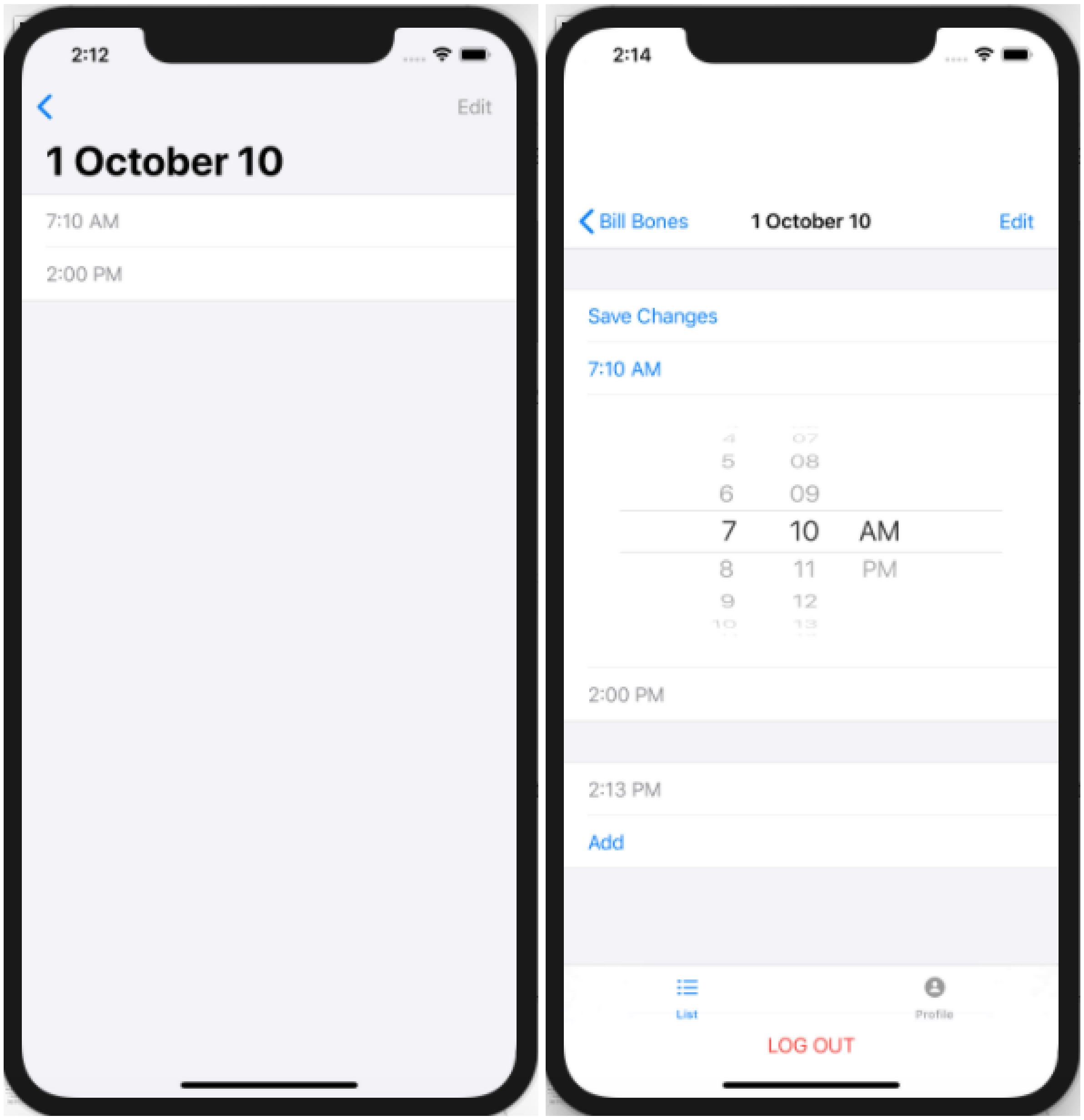


```
    func deleteEmployee(at offsets: IndexSet) {
        for offset in offsets {
            // find this employee in our fetch request
            let employee = elements.wrappedValue[offset]
            // delete it from the context
            moc.delete(employee)
        }
        // save the context
        trySave(moc: moc)
    }
```

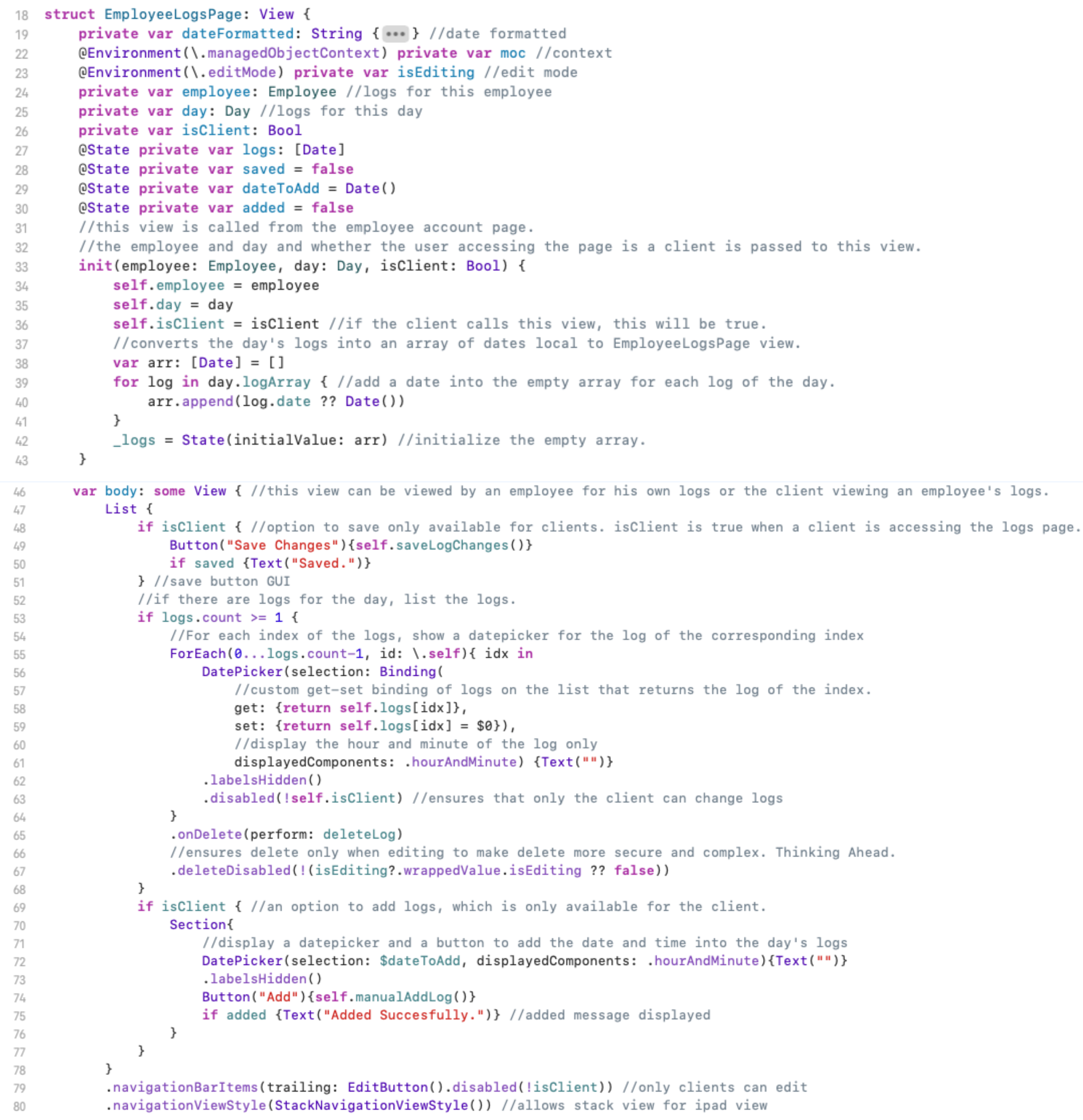
(*refer to Appendix E for complete codes*)

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).*



```
18 struct EmployeeLogsPage: View {
19     private var dateFormatted: String {•••} //date formatted
22     @Environment(\.managedObjectContext) private var moc //context
23     @Environment(\.editMode) private var isEditing //edit mode
24     private var employee: Employee //logs for this employee
25     private var day: Day //logs for this day
26     private var isClient: Bool
27     @State private var logs: [Date]
28     @State private var saved = false
29     @State private var dateToAdd = Date()
30     @State private var added = false
31     //this view is called from the employee account page.
32     //the employee and day and whether the user accessing the page is a client is passed to this view.
33     init(employee: Employee, day: Day, isClient: Bool) {
34         self.employee = employee
35         self.day = day
36         self.isClient = isClient //if the client calls this view, this will be true.
37         //converts the day's logs into an array of dates local to EmployeeLogsPage view.
38         var arr: [Date] = []
39         for log in day.logArray { //add a date into the empty array for each log of the day.
40             arr.append(log.date ?? Date())
41         }
42         _logs = State(initialValue: arr) //initialize the empty array.
43     }

46     var body: some View { //this view can be viewed by an employee for his own logs or the client viewing an employee's logs.
47         List {
48             if isClient { //option to save only available for clients. isClient is true when a client is accessing the logs page.
49                 Button("Save Changes"){self.saveLogChanges()}
50                 if saved {Text("Saved.")}
51             } //save button GUI
52             //if there are logs for the day, list the logs.
53             if logs.count >= 1 {
54                 //For each index of the logs, show a datepicker for the log of the corresponding index
55                 ForEach(0...logs.count-1, id: \.self){ idx in
56                     DatePicker(selection: Binding(
57                         //custom get-set binding of logs on the list that returns the log of the index.
58                         get: {return self.logs[idx]},
59                         set: {return self.logs[idx] = $0}),
60                         //display the hour and minute of the log only
61                         displayedComponents: .hourAndMinute) {Text("")}
62                     .labelsHidden()
63                     .disabled(!self.isClient) //ensures that only the client can change logs
64                 }
65                 .onDelete(perform: deleteLog)
66                 //ensures delete only when editing to make delete more secure and complex. Thinking Ahead.
67                 .deleteDisabled(!(isEditing?.wrappedValue.isEditing ?? false))
68             }
69             if isClient { //an option to add logs, which is only available for the client.
70                 Section{
71                     //display a datepicker and a button to add the date and time into the day's logs
72                     DatePicker(selection: $dateToAdd, displayedComponents: .hourAndMinute){Text("")}
73                     .labelsHidden()
74                     Button("Add"){self.manualAddLog()}
75                     if added {Text("Added Succesfully.")} //added message displayed
76                 }
77             }
78         }
79         .navigationBarItems(trailing: EditButton().disabled(!isClient)) //only clients can edit
80         .navigationViewStyle(StackNavigationViewStyle()) //allows stack view for ipad view
```

*(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/.