Faculty of Arts and Sciences

Department of Computer Science

CMPS 297N – Mobile Development

Fall 2020, Professor Rami Farran

Project Report

For the Group Term Project:

To-Do List (iOS)

Team Members:

Adam Helal

Farah Maria Majdalani

Mohammed Kaimouz

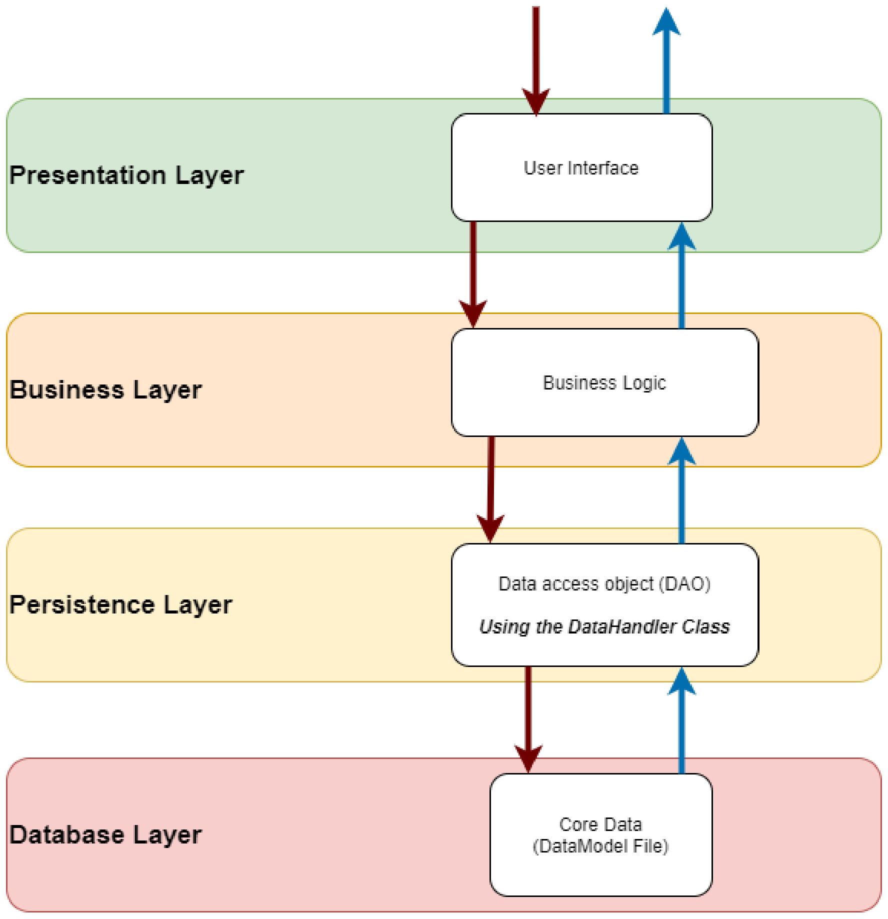
Youssef Itani

High-Level Description:

With the To-Do List app, experience a simple, minimalistic , and intuitive application developed solely for iOS devices. Our application was developed with productivity and efficiency as main goals, focusing mainly on adding and completing tasks, rather than wasting precious time on the unnecessary details.

Users will be able to create their own account using their emails. In addition, each user will be able to create multiple lists that allow better organization. Each list contains detailed tasks to boost your workflow efficiency.

Software Architecture:



Software Design-Pattern:

The software design-pattern that we used while implementing our application is the **Model-View Controller design pattern (MVC).**

Model:

We used the Core Data Model in order to store all of the data that are required in order for our application to function as it is supposed.

Entities: User, Task, List

View:

The view is basically the screen that is presented to the user. The screen displays the data the user has provided as input as part of views, which are ordered by constraints. As per the nature of iOS development, each screen has a corresponding controller.

Controller:

Each controller is bound to a view, and implements the functionality with logic. In addition, the controller makes sure that there is a clear channel of communication between the view and the model (Perform CRUD operations)

Components:

1. **loginViewController:**
    - Responsible for the logic input field constraints
    - Communicates with the database for authentication
2. **signUpViewController**
    - Responsible for the logic input field constraints
    - Communicates with the database in order to add the data
3. **ListTableViewController**
    - Communicates with the database in order to display the user’s lists.
    - Communicates with the database in order to add or delete lists
4. **TaskBarViewController**
    - Parent controller of the TaskTableViewController and the CompletedTaskTableViewController
5. **TaskTableViewController**
    - Communicates with the database in order to perform CRUD operations.
    - Provides the user with functionality to add,edit,delete and complete tasks.
6. **CompletedTaskTableViewController**
    - Communicates with the database in order to display/delete the completed tasks.

7. **AddTaskViewController**
    - Allows the user to create tasks by adding the necessary details, implementing the logic and adding it to the database
8. **TaskViewController**
    - Displays the current selected task and allows the user to edit.
    - Communicates with the database to retrieve/edit the information selected task.
9. **TaskTableViewCell and ListTableViewCell**
    - Format the cells that we can see in their respective table view controllers
10. **DataHandler**
    - A single generic class that handles operations on the persistence container in the core data, which helped in having the code to be much more reusable.

Project Roles

Initially, the teams were divided into Frontend/Backend teams. However, later on due to the constraints (core data was not learned yet), the work was divided such that :

- The frontend team would develop prototype 1 (all screens and full app logic/functionality without core data).
- The backend team would then take this prototype and integrate it with core data (add to all needed screens and functions) for the final deliverable (prototype 2).

Frontend Team:

- **Farah Maria Majdalani**:Designed the add/edit task page with it's full logic/functionality and connected it to the list of tasks page. Also designed the login & sign up pages.
- **Adam Helal**: Designed the list of lists("Your lists" page) & list of tasks/completed tasks page and implemented the full logic/functionality for it. Additionally, implemented full logic/functionality for the login & signup page.

Note: Frontend teammate roles were interchangeable (helped each other out) as too fix some bugs/implement features as wanted.

Backend Team:

- **Mohamad Kaimouz & Youssef Itani**: Designed the schema & relationships between the entities supporting the app. Designed the DataHandler generic class for core data operations. Managed a restricted access of only registered users to the app. Integrated CRUD operations with Core Data on all the ***components*** offered by the front-end team and changed the code accordingly.

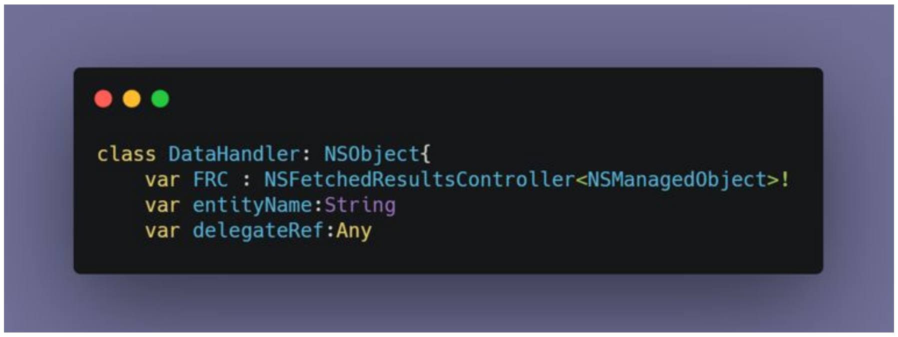
  Note: Due to hardships related to running a smooth virtual Mac OS machine as our main framework for developing an XCode app, we opted for the pair programming technique used in agile software development.
  Our team had to rent a capable virtual machine available on the web and we used screen sharing to complement the work of the front-end team.

This added layer of difficulty led to much longer development hours but at the same time a great assistance and collaboration in the development of each class that enabled the persistence of data in the app!

Documentation:

DataHandler Class

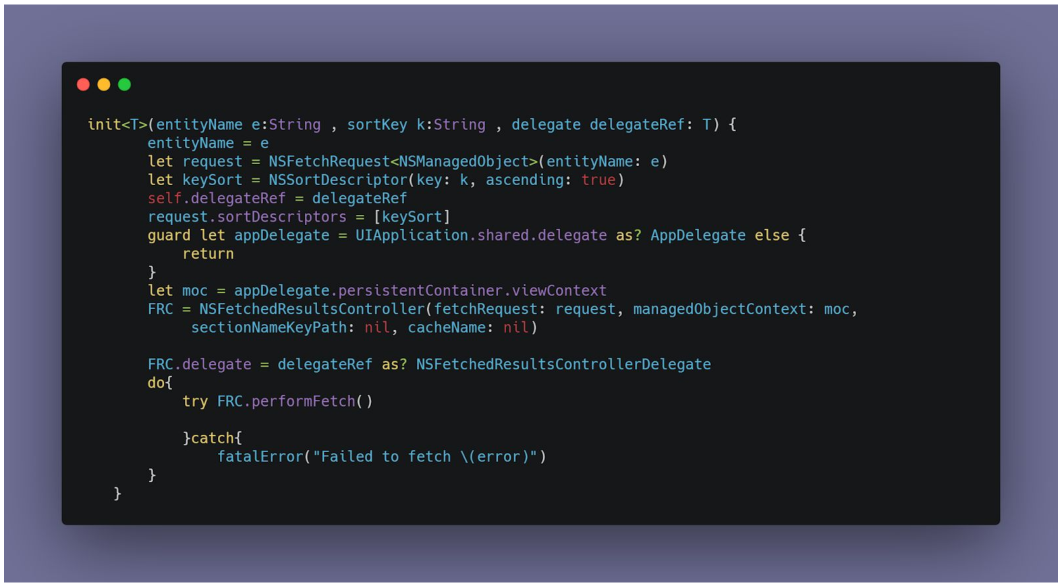
The DataHandler class is our generic class for the Core Data logic. We handle all CRUD operations on the Core Data through this class.



```swift
class DataHandler: NSObject{
    var FRC : NSFetchedResultsController<NSManagedObject>!
    var entityName:String
    var delegateRef:Any
```

- We set up ***three*** fields for easier reference inside the class and from outside the class. One of which is the **Fetched Results Controller**.

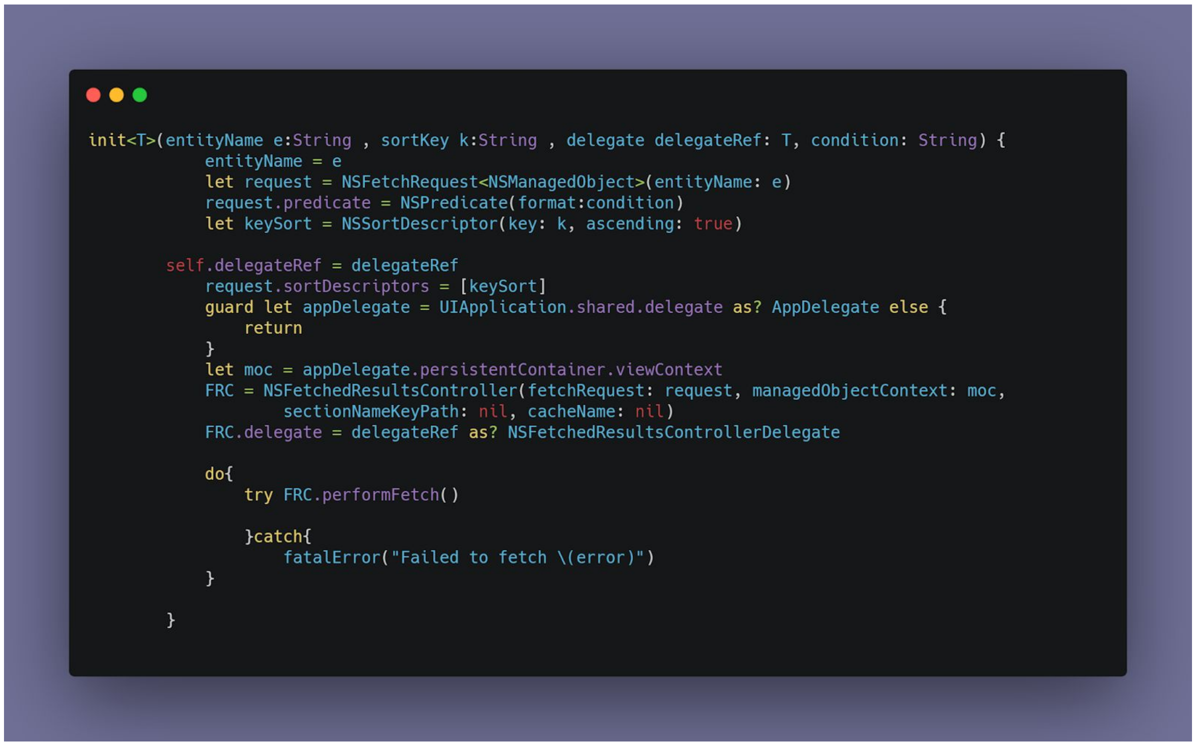
As for the initializers, we wrote two initializers to set up our DataHandler class.



```swift
init<T>(entityName e:String , sortKey k:String , delegate delegateRef: T) {
        entityName = e
        let request = NSFetchRequest<NSManagedObject>(entityName: e)
        let keySort = NSSortDescriptor(key: k, ascending: true)
        self.delegateRef = delegateRef
        request.sortDescriptors = [keySort]
        guard let appDelegate = UIApplication.shared.delegate as? AppDelegate else {
            return
        }
        let moc = appDelegate.persistentContainer.viewContext
        FRC = NSFetchedResultsController(fetchRequest: request, managedObjectContext: moc,
             sectionNameKeyPath: nil, cacheName: nil)

        FRC.delegate = delegateRef as? NSFetchedResultsControllerDelegate
        do{
            try FRC.performFetch()

            }catch{
                fatalError("Failed to fetch \(error)")
        }
    }
```

- This is the first initializer, the general initializer. We pass to it three parameters:
    - **entityName**: The name of the entity (table) in the core data model, to set the value of the class field
    - **sortKey**: The key according to which is used to sort the resultant data
    - **delegate**: To set the class field for future reference
- Inside the initializer we start setting up our request by providing all the required fields to perform the fetch

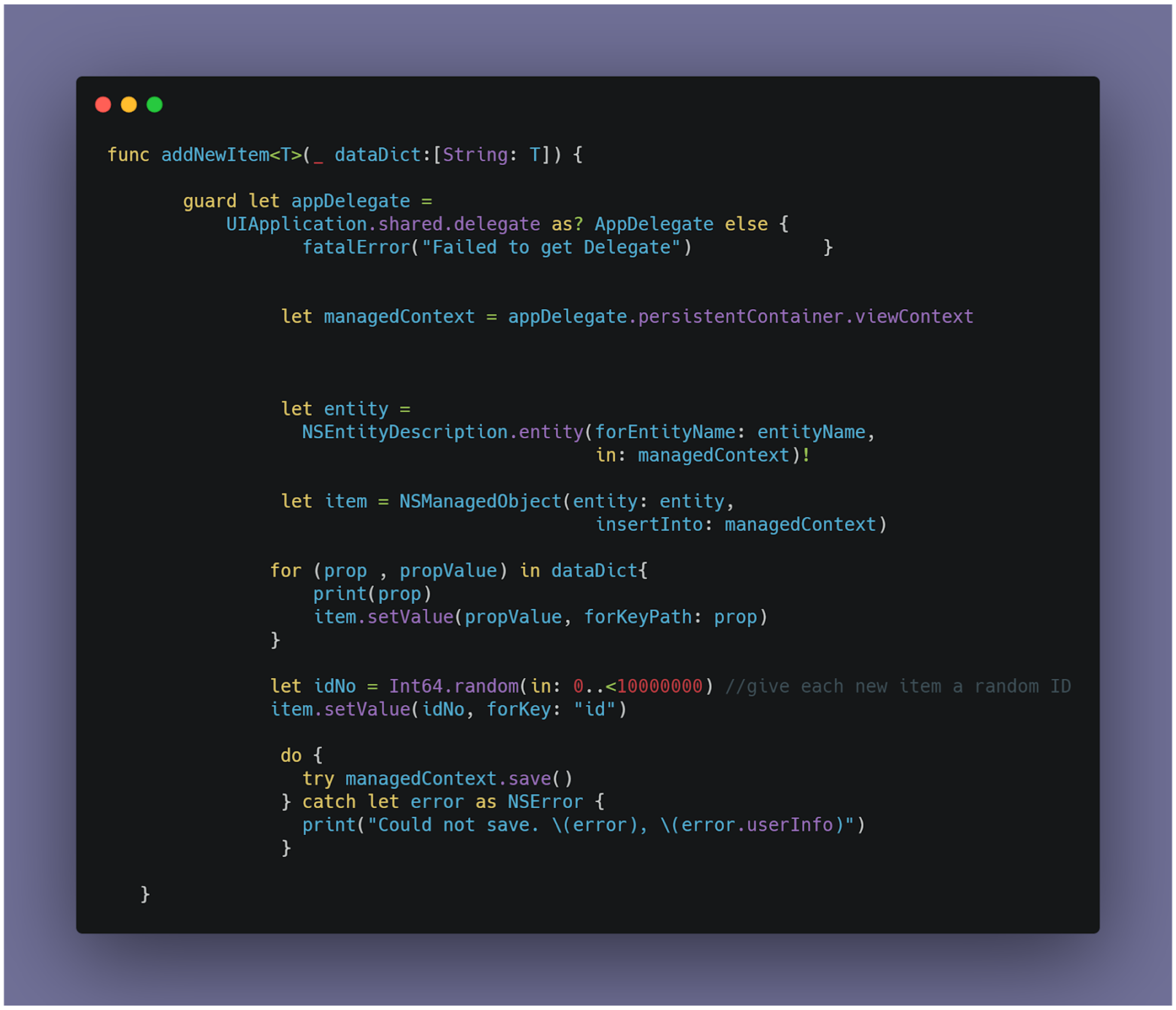
As for the second initializer:



```
init<T>(entityName e:String , sortKey k:String , delegate delegateRef: T, condition: String) {
            entityName = e
            let request = NSFetchRequest<NSManagedObject>(entityName: e)
            request.predicate = NSPredicate(format:condition)
            let keySort = NSSortDescriptor(key: k, ascending: true)

        self.delegateRef = delegateRef
            request.sortDescriptors = [keySort]
            guard let appDelegate = UIApplication.shared.delegate as? AppDelegate else {
                return
            }
            let moc = appDelegate.persistentContainer.viewContext
            FRC = NSFetchedResultsController(fetchRequest: request, managedObjectContext: moc,
                    sectionNameKeyPath: nil, cacheName: nil)
            FRC.delegate = delegateRef as? NSFetchedResultsControllerDelegate

            do{
                try FRC.performFetch()

                }catch{
                    fatalError("Failed to fetch \(error)")
            }

        }
```

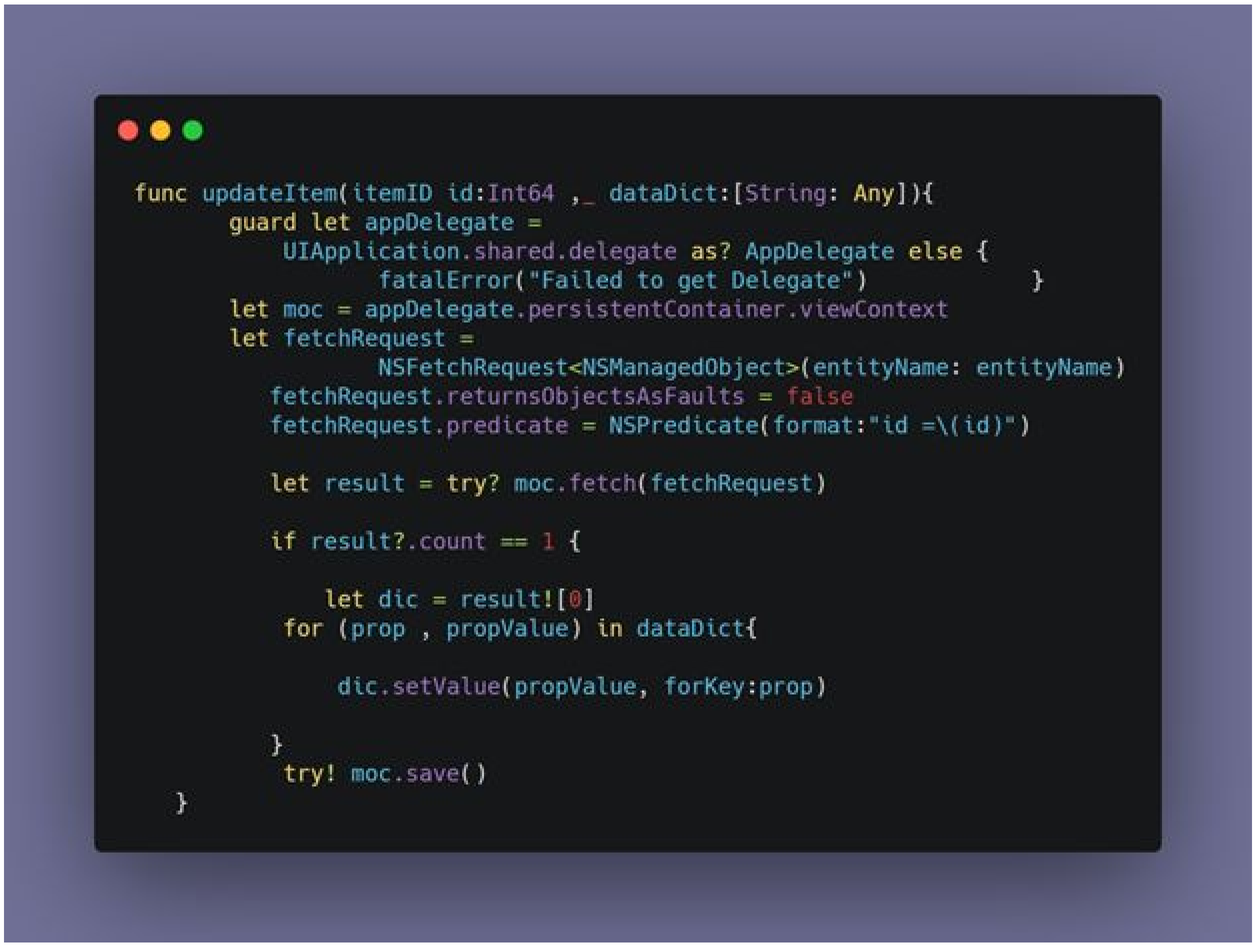
- This initializer is very similar in format to the first one except it requires one more parameter:
    - **condition**: A string that represents the condition to pass to the request object
- In addition to the regular request setup we did in the first initializer, here we are able to add a condition to the request which filters the results we get back from Core Data

Adding a new item to the entity



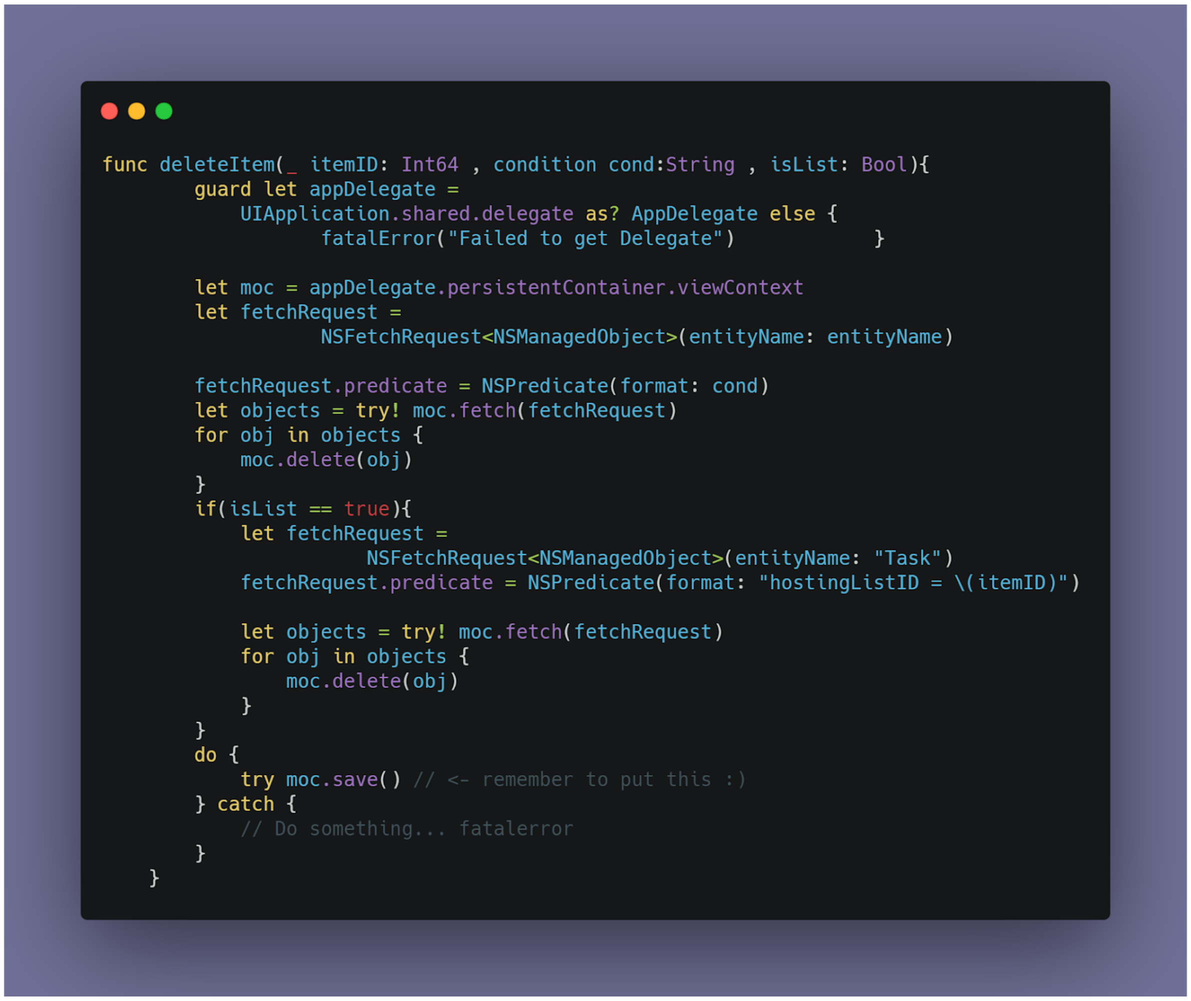
```
func addNewItem<T>(_ dataDict:[String: T]) {

       guard let appDelegate =
           UIApplication.shared.delegate as? AppDelegate else {
                  fatalError("Failed to get Delegate")            }


                let managedContext = appDelegate.persistentContainer.viewContext



                let entity =
                  NSEntityDescription.entity(forEntityName: entityName,
                                             in: managedContext)!

                let item = NSManagedObject(entity: entity,
                                             insertInto: managedContext)

               for (prop , propValue) in dataDict{
                   print(prop)
                   item.setValue(propValue, forKeyPath: prop)
               }

               let idNo = Int64.random(in: 0..<10000000) //give each new item a random ID
               item.setValue(idNo, forKey: "id")

                do {
                  try managedContext.save()
                } catch let error as NSError {
                  print("Could not save. \(error), \(error.userInfo)")
                }

   }
```

- This function here allows you to add a new item to your table in the Core Data model.
    - **dataDict**: is a dictionary of properties and their values, which can be of any type.
- A normal Core Data insert operation where we are getting the keys and values from the dictionary we passed and setting these values inside the loop. We also get the entity we are adding to from the class field we created.
- A simple check exists to see if the entity we are adding to is not a user as our user model does not have an id field.



```
func updateItem(itemID id:Int64 ,_ dataDict:[String: Any]){
       guard let appDelegate =
           UIApplication.shared.delegate as? AppDelegate else {
                  fatalError("Failed to get Delegate")            }
       let moc = appDelegate.persistentContainer.viewContext
       let fetchRequest =
                  NSFetchRequest<NSManagedObject>(entityName: entityName)
          fetchRequest.returnsObjectsAsFaults = false
          fetchRequest.predicate = NSPredicate(format:"id =\(id)")

          let result = try? moc.fetch(fetchRequest)

          if result?.count == 1 {

              let dic = result![0]
           for (prop , propValue) in dataDict{

               dic.setValue(propValue, forKey:prop)

          }
           try! moc.save()
   }
```

- This function handles the update operation. It has two parameters:
    - **itemID**: the ID value of the item you want to update
    - **dataDict**: a dictionary of the properties you want to update with the updated value
- This function works in a similar way to the addNewitem except that it only updates specified properties for an item with the specified ID. A simple check exists to check if the provided ID returns a valid item.

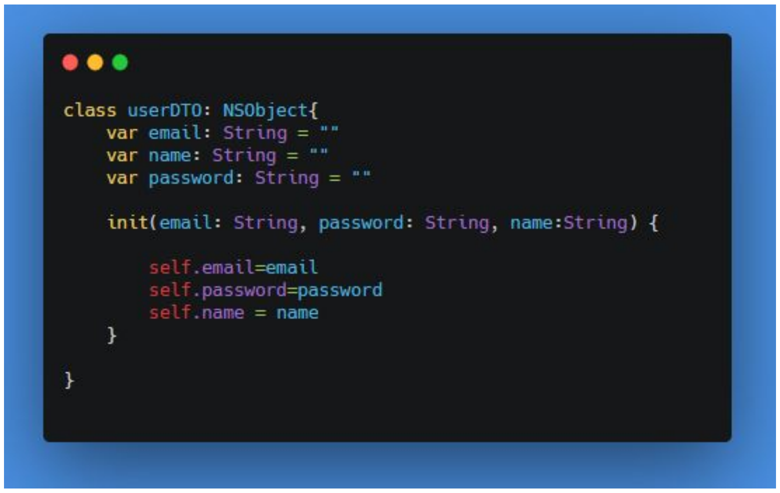


```
func deleteItem(_ itemID: Int64 , condition cond:String , isList: Bool){
        guard let appDelegate =
            UIApplication.shared.delegate as? AppDelegate else {
                   fatalError("Failed to get Delegate")           }

        let moc = appDelegate.persistentContainer.viewContext
        let fetchRequest =
                   NSFetchRequest<NSManagedObject>(entityName: entityName)

        fetchRequest.predicate = NSPredicate(format: cond)
        let objects = try! moc.fetch(fetchRequest)
        for obj in objects {
            moc.delete(obj)
        }
        if(isList == true){
            let fetchRequest =
                       NSFetchRequest<NSManagedObject>(entityName: "Task")
            fetchRequest.predicate = NSPredicate(format: "hostingListID = \(itemID)")

            let objects = try! moc.fetch(fetchRequest)
            for obj in objects {
                moc.delete(obj)
            }
        }
        do {
            try moc.save() // <- remember to put this :)
        } catch {
            // Do something... fatalerror
        }
    }
```

- This function handles the delete operation on the Core Data and takes three parameters:
    - **itemID**: The id of the list you are deleting
    - **condition**: A string that represents the condition to filter the request
    - **isList**: A Boolean value that reflects if the item to be deleted is a list
- In this function we perform a normal delete operation. We pass a condition that filters the results that will be deleted (this condition could be the id on an item). We then delete all the returned results.
- In case isList is set to “true” we then need to remove the list item with all the tasks under it (related to it).

User Data Transfer Object

The following snippet is the data transfer object class for the **User** entity.

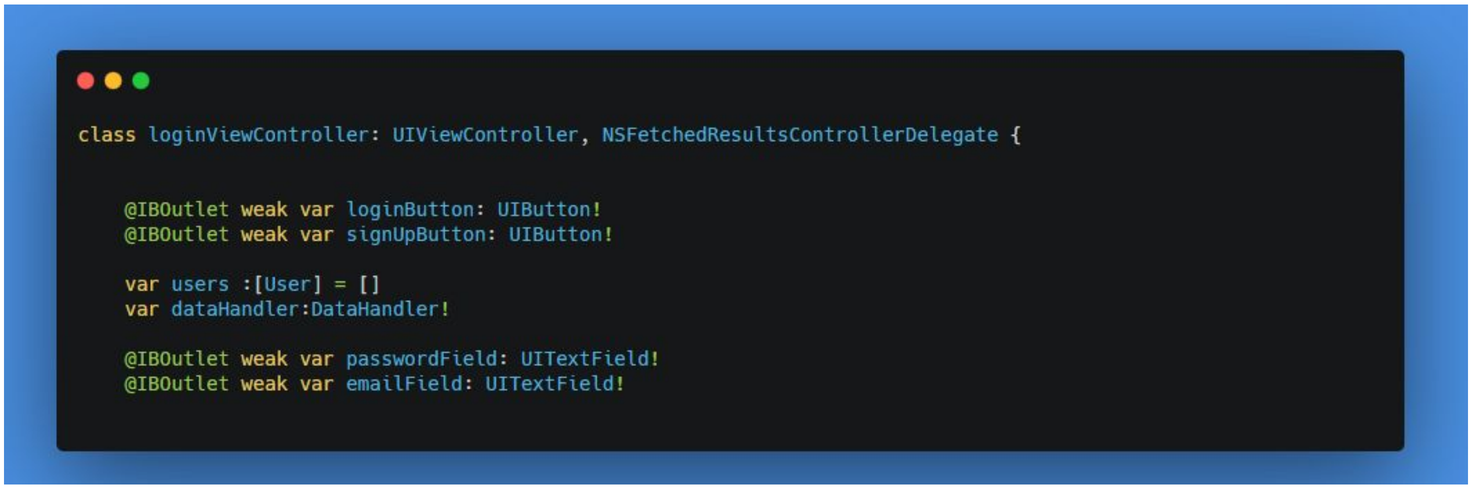


```swift
class userDTO: NSObject{
    var email: String = ""
    var name: String = ""
    var password: String = ""

    init(email: String, password: String, name:String) {

        self.email=email
        self.password=password
        self.name = name
    }

}
```

Features

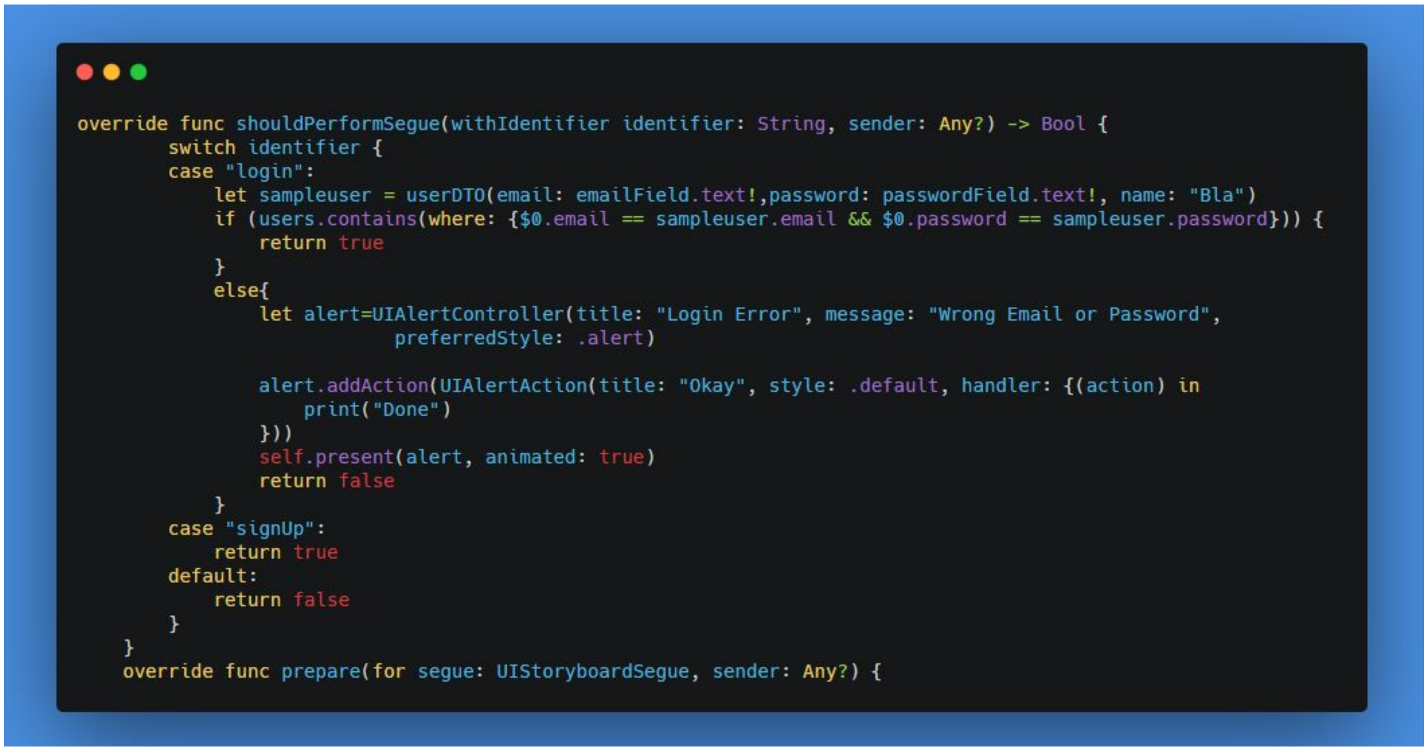
Logging In

LoginViewController



```swift
class loginViewController: UIViewController, NSFetchedResultsControllerDelegate {


    @IBOutlet weak var loginButton: UIButton!
    @IBOutlet weak var signUpButton: UIButton!

    var users :[User] = []
    var dataHandler:DataHandler!

    @IBOutlet weak var passwordField: UITextField!
    @IBOutlet weak var emailField: UITextField!
```

- Fields Important to set up our login screen

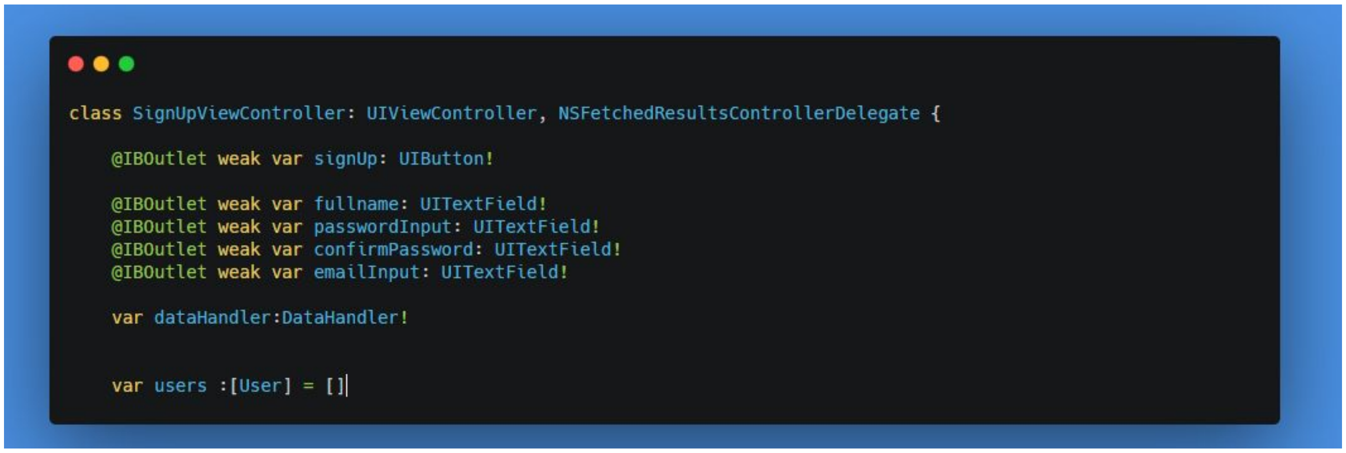


```
override func shouldPerformSegue(withIdentifier identifier: String, sender: Any?) -> Bool {
        switch identifier {
        case "login":
            let sampleuser = userDTO(email: emailField.text!,password: passwordField.text!, name: "Bla")
            if (users.contains(where: {$0.email == sampleuser.email && $0.password == sampleuser.password})) {
                return true
            }
            else{
                let alert=UIAlertController(title: "Login Error", message: "Wrong Email or Password",
                            preferredStyle: .alert)

                alert.addAction(UIAlertAction(title: "Okay", style: .default, handler: {(action) in
                    print("Done")
                }))
                self.present(alert, animated: true)
                return false
            }
        case "signUp":
            return true
        default:
            return false
        }
    }
    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
```

- The segue from the login screen is bound to the ListTableViewController where the list of lists are displayed. In the **shouldPerformSegue** function we perform important checks to see if the provided user is registered in our users table. If this check fails a pop up with a login error message shows up

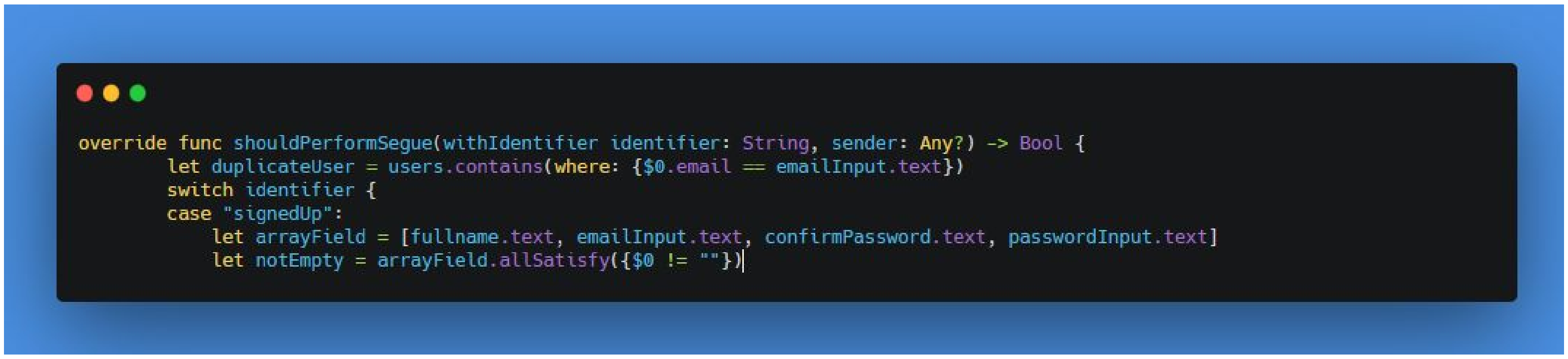
Signing Up

SignUpViewController



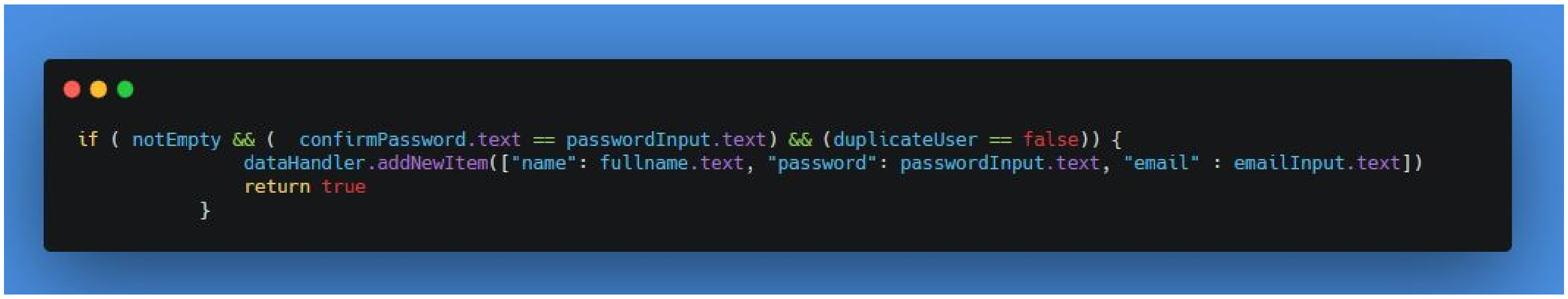
```
class SignUpViewController: UIViewController, NSFetchedResultsControllerDelegate {

    @IBOutlet weak var signUp: UIButton!

    @IBOutlet weak var fullname: UITextField!
    @IBOutlet weak var passwordInput: UITextField!
    @IBOutlet weak var confirmPassword: UITextField!
    @IBOutlet weak var emailInput: UITextField!

    var dataHandler:DataHandler!


    var users :[User] = []
```

- This class handles the logic behind the signing up screen. The initializer sets up the users array with all the users found in our Core Data



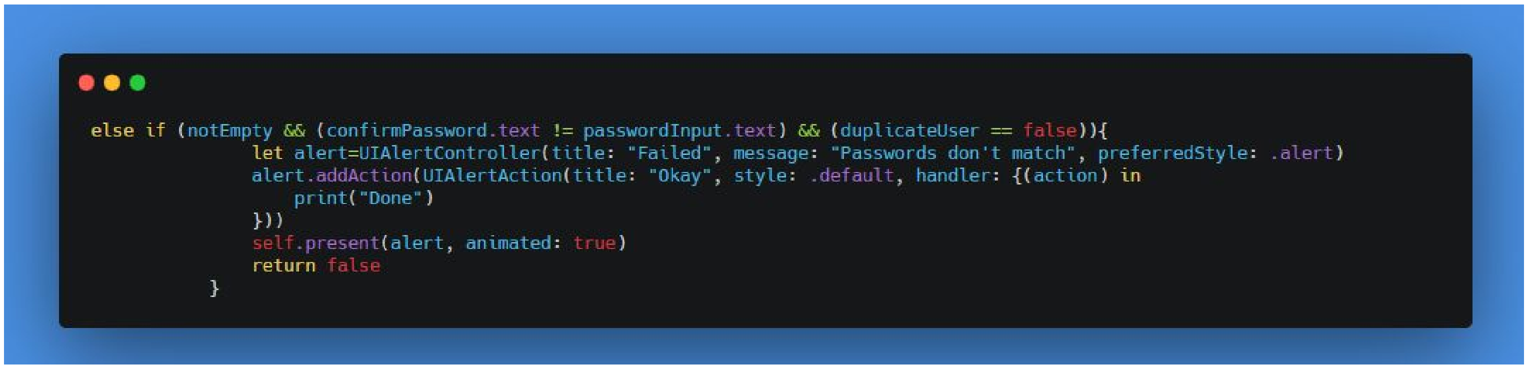
```
override func shouldPerformSegue(withIdentifier identifier: String, sender: Any?) -> Bool {
        let duplicateUser = users.contains(where: {$0.email == emailInput.text})
        switch identifier {
        case "signedUp":
            let arrayField = [fullname.text, emailInput.text, confirmPassword.text, passwordInput.text]
            let notEmpty = arrayField.allSatisfy({$0 != ""})
```

- This function checks if appropriate information were provided in the text fields. The duplicateUser field is a boolean field that checks if the provided email was registered before



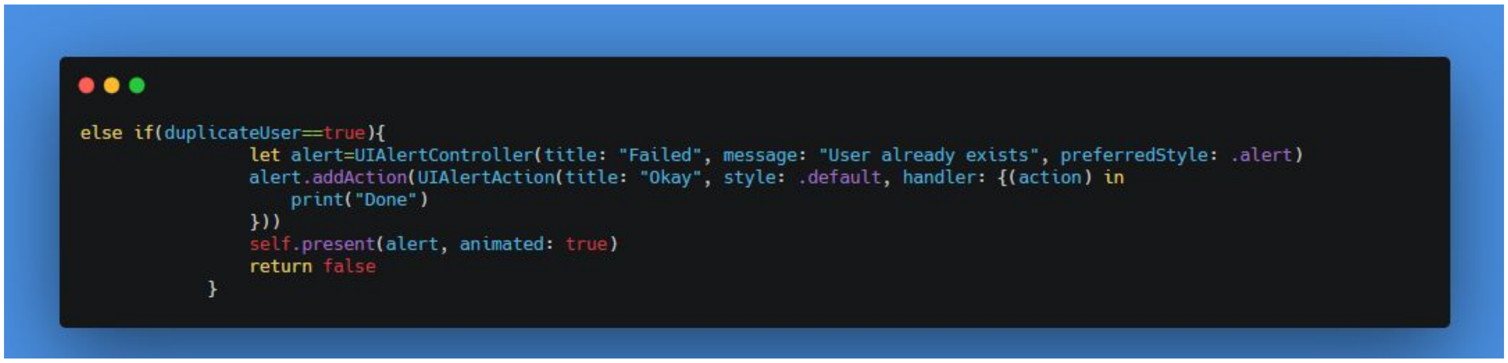
```
if ( notEmpty && (  confirmPassword.text == passwordInput.text) && (duplicateUser == false)) {
               dataHandler.addNewItem(["name": fullname.text, "password": passwordInput.text, "email" : emailInput.text])
               return true
           }
```

- This first check checks if all the fields are not empty and the text provided in both the passwordInput field and the confirmPassword field match. It also checks if the user provided was not registered before. If these checks pass we use our dataHandler class to register this user and add it to our users table



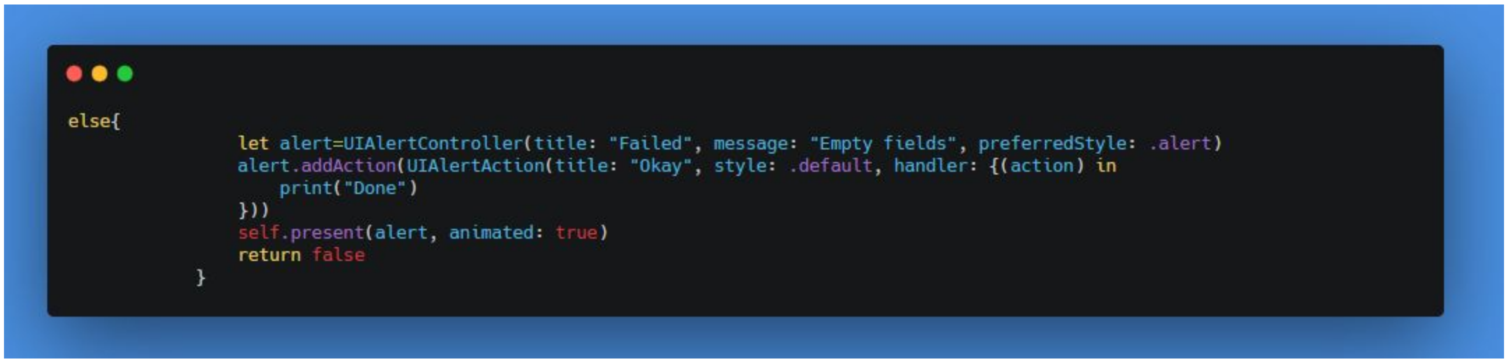
```
else if (notEmpty && (confirmPassword.text != passwordInput.text) && (duplicateUser == false)){
               let alert=UIAlertController(title: "Failed", message: "Passwords don't match", preferredStyle: .alert)
               alert.addAction(UIAlertAction(title: "Okay", style: .default, handler: {(action) in
                   print("Done")
               }))
               self.present(alert, animated: true)
               return false
           }
```

- We check if there is the two passwords don't match and provide a pop up as a feedback if they don't



```
else if(duplicateUser==true){
            let alert=UIAlertController(title: "Failed", message: "User already exists", preferredStyle: .alert)
            alert.addAction(UIAlertAction(title: "Okay", style: .default, handler: {(action) in
                print("Done")
            }))
            self.present(alert, animated: true)
            return false
        }
```

- If the user is registered before a pop up feedback will show up with a descriptive message telling you so.

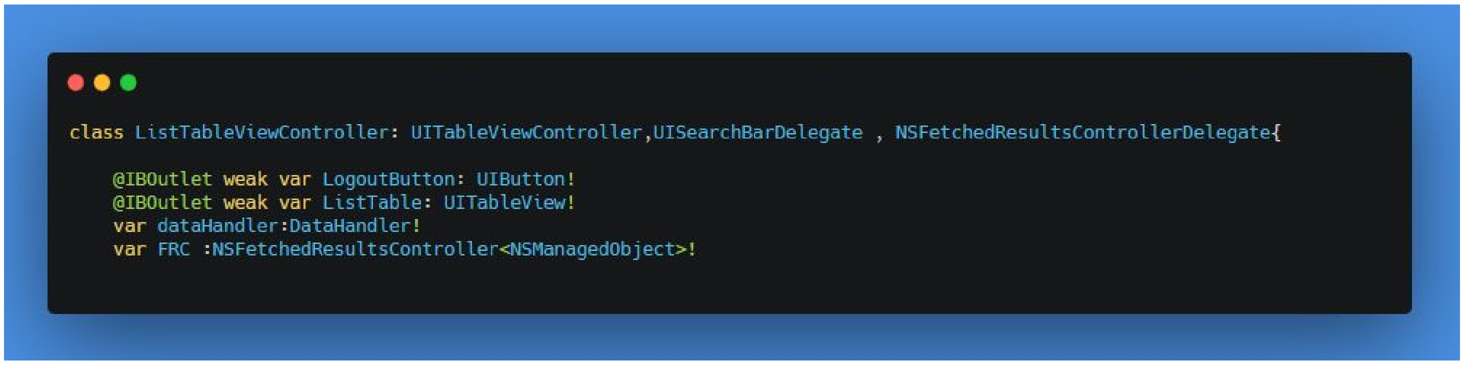


```
else{
            let alert=UIAlertController(title: "Failed", message: "Empty fields", preferredStyle: .alert)
            alert.addAction(UIAlertAction(title: "Okay", style: .default, handler: {(action) in
                print("Done")
            }))
            self.present(alert, animated: true)
            return false
        }
```

- In the case of any empty fields a different pop up message will show telling you so as all fields are required to sign up

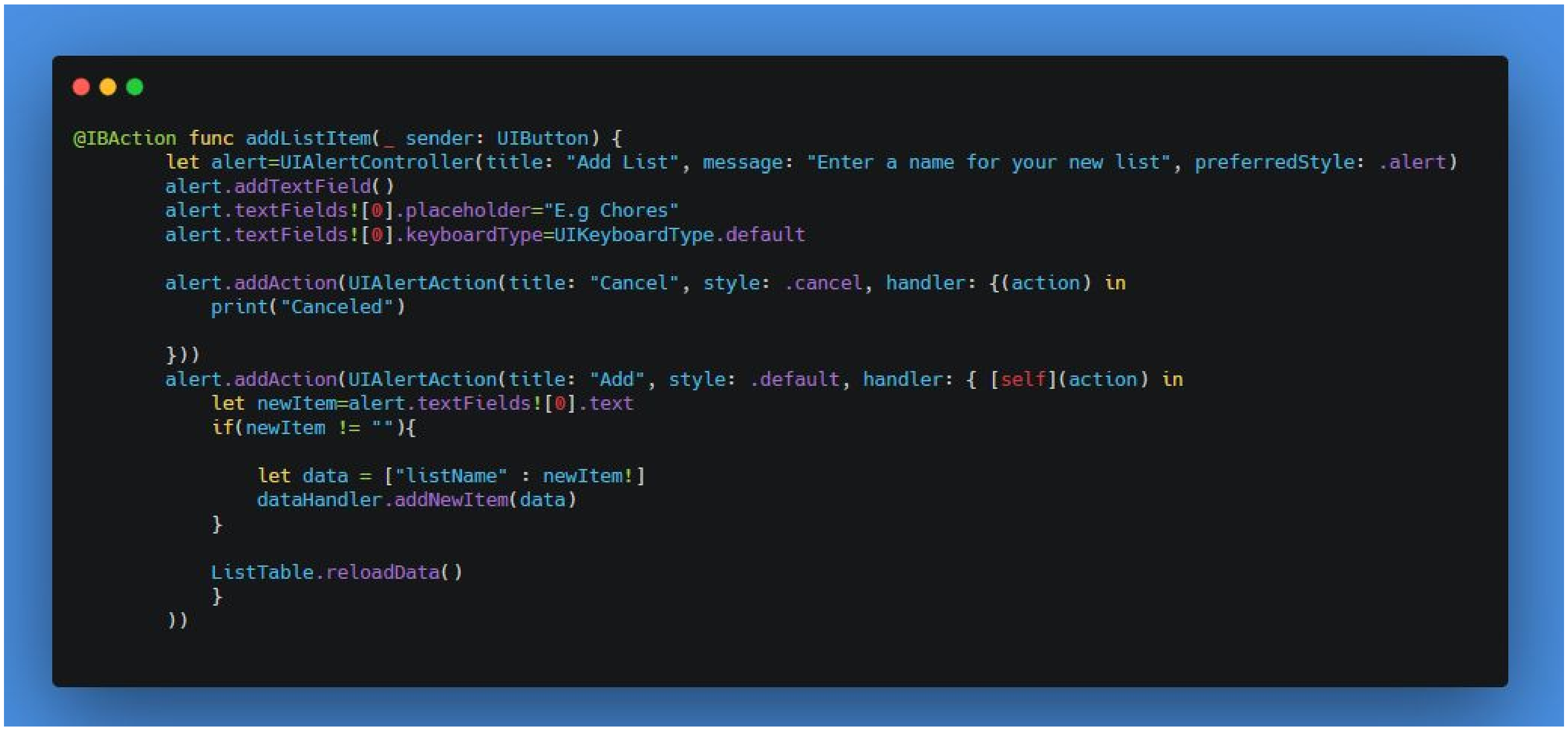
List of Lists

ListTableViewController



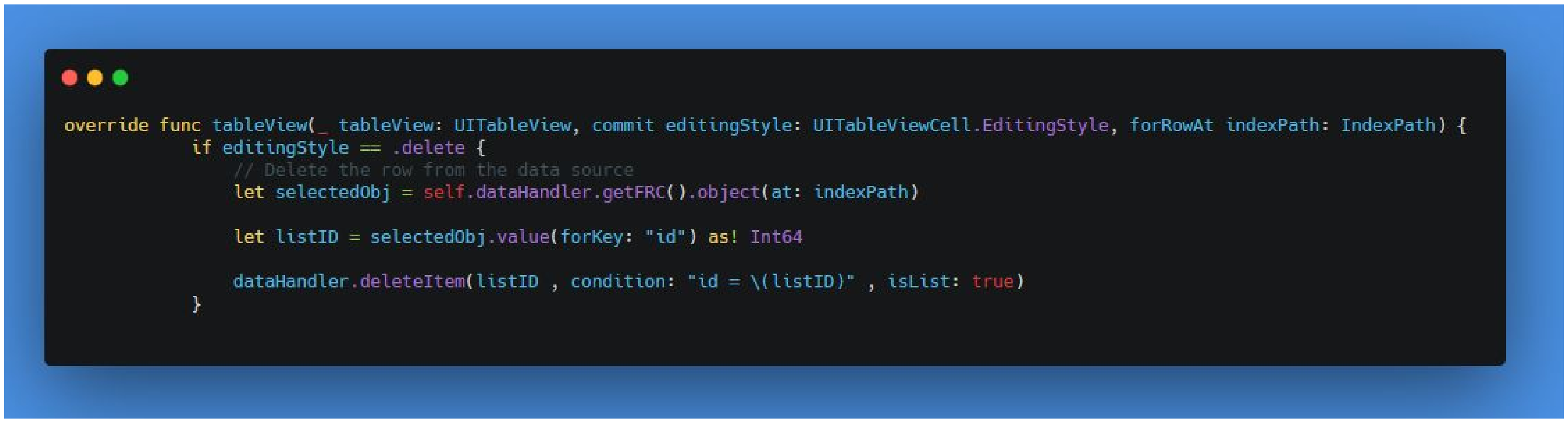
```
class ListTableViewController: UITableViewController,UISearchBarDelegate , NSFetchedResultsControllerDelegate{

    @IBOutlet weak var LogoutButton: UIButton!
    @IBOutlet weak var ListTable: UITableView!
    var dataHandler:DataHandler!
    var FRC :NSFetchedResultsController<NSManagedObject>!
```

- This class handles the logic behind our list of lists where the tasks are hosted



```swift
@IBAction func addListItem(_ sender: UIButton) {
        let alert=UIAlertController(title: "Add List", message: "Enter a name for your new list", preferredStyle: .alert)
        alert.addTextField()
        alert.textFields![0].placeholder="E.g Chores"
        alert.textFields![0].keyboardType=UIKeyboardType.default

        alert.addAction(UIAlertAction(title: "Cancel", style: .cancel, handler: {(action) in
            print("Canceled")

        }))
        alert.addAction(UIAlertAction(title: "Add", style: .default, handler: { [self](action) in
            let newItem=alert.textFields![0].text
            if(newItem != ""){

                let data = ["listName" : newItem!]
                dataHandler.addNewItem(data)
            }

            ListTable.reloadData()
            }
        ))
```

- You can add list items from a pop up once the “+” button is pressed. It is worth mentioning that you won’t be able to add a list with an empty title as we handle this edge case
- The addListItem function takes the information from the text field in the pop up and prepare a proper dictionary with the required properties to pass to our addNewItem in the DataHandler class

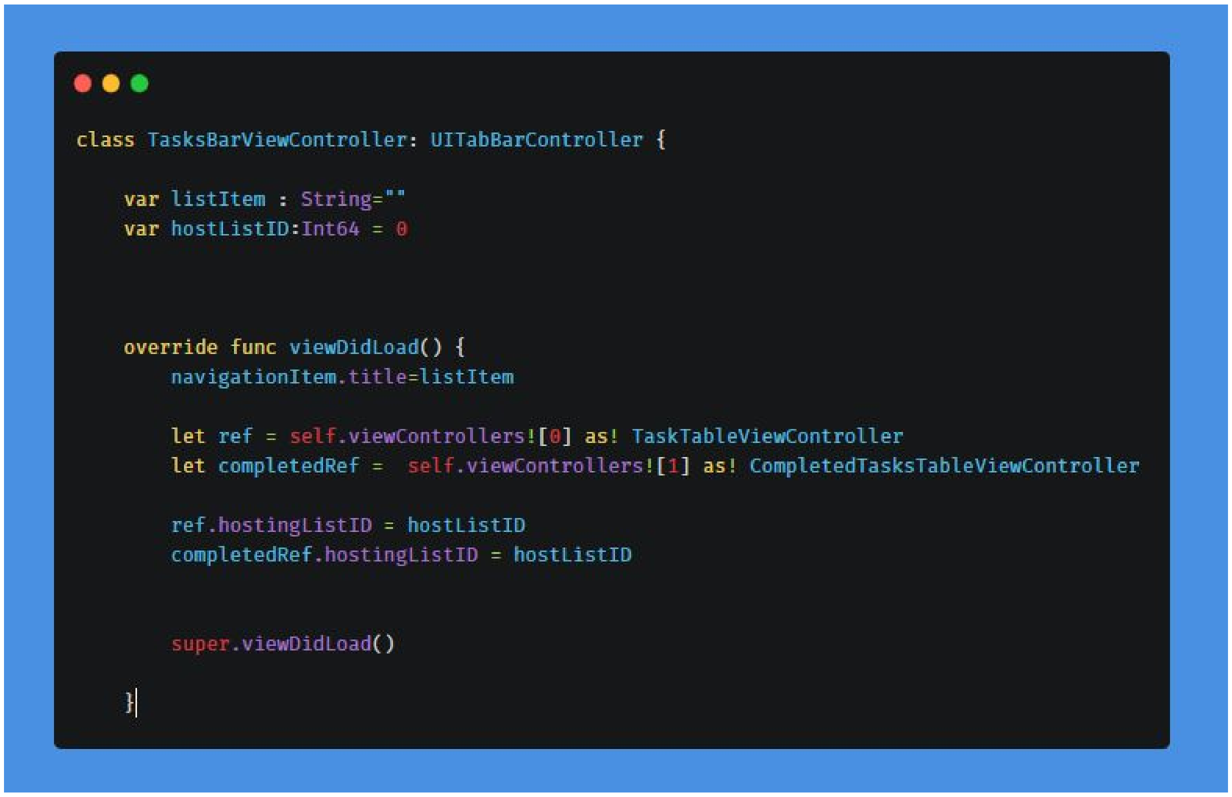


```swift
override func tableView(_ tableView: UITableView, commit editingStyle: UITableViewCell.EditingStyle, forRowAt indexPath: IndexPath) {
            if editingStyle == .delete {
                // Delete the row from the data source
                let selectedObj = self.dataHandler.getFRC().object(at: indexPath)

                let listID = selectedObj.value(forKey: "id") as! Int64

                dataHandler.deleteItem(listID , condition: "id = \(listID)" , isList: true)
            }
```

- In this snippet we add a custom functionality to swiping left on certain list item which will delete that item

Tasks

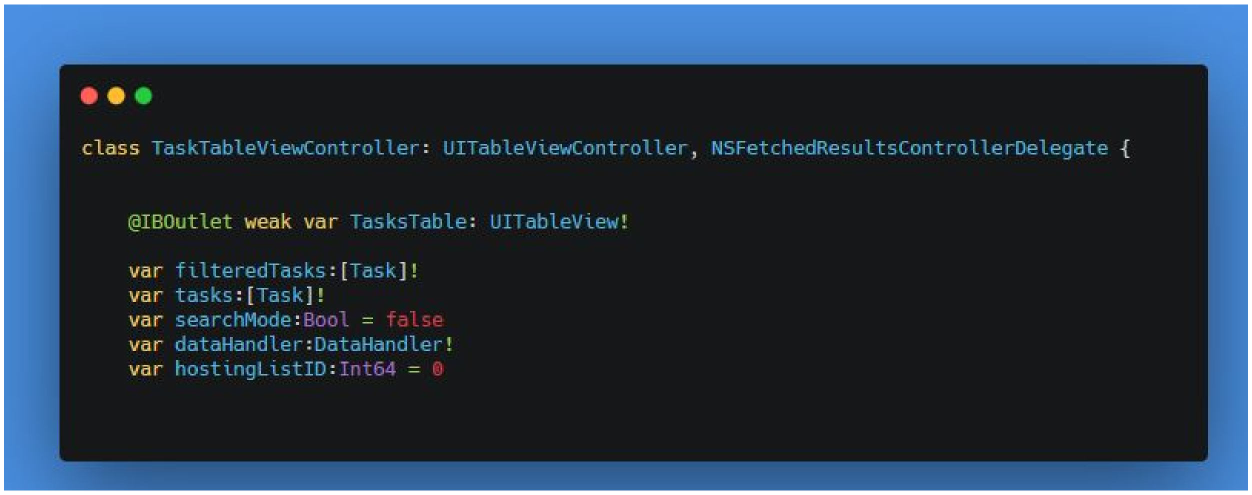
TaskBarViewController



```
class TasksBarViewController: UITabBarController {

    var listItem : String=""
    var hostListID:Int64 = 0



    override func viewDidLoad() {
        navigationItem.title=listItem

        let ref = self.viewControllers![0] as! TaskTableViewController
        let completedRef =  self.viewControllers![1] as! CompletedTasksTableViewController

        ref.hostingListID = hostListID
        completedRef.hostingListID = hostListID


        super.viewDidLoad()

    }
```

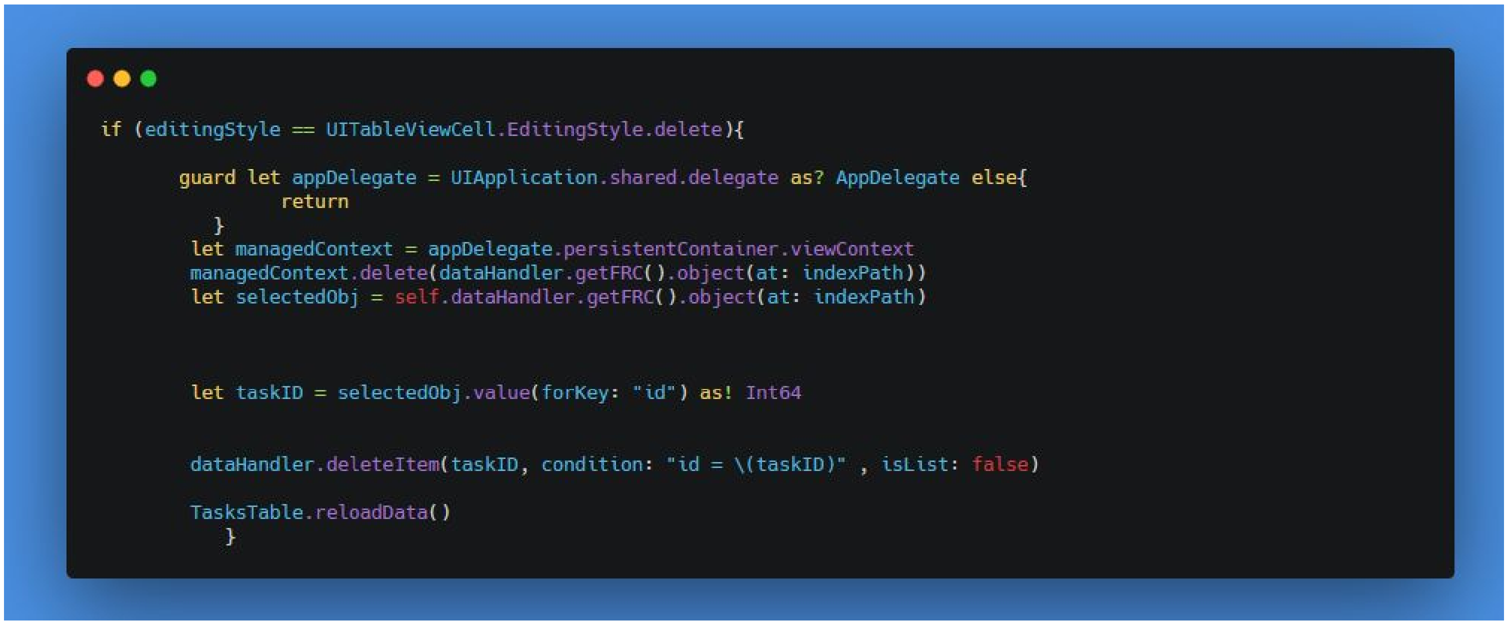
This class basically handles some housekeeping information about our task bar which includes as casting it's children views(views accessed by the tab) to their corresponding controller

TaskTableViewController

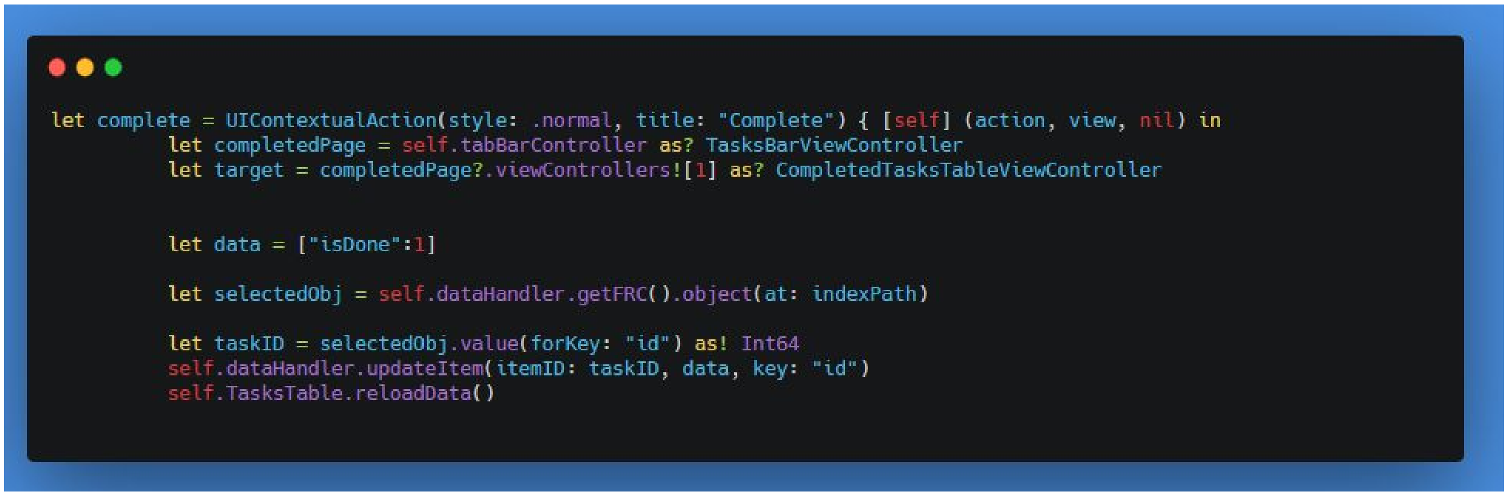


```
class TaskTableViewController: UITableViewController, NSFetchedResultsControllerDelegate {


    @IBOutlet weak var TasksTable: UITableView!

    var filteredTasks:[Task]!
    var tasks:[Task]!
    var searchMode:Bool = false
    var dataHandler:DataHandler!
    var hostingListID:Int64 = 0
```

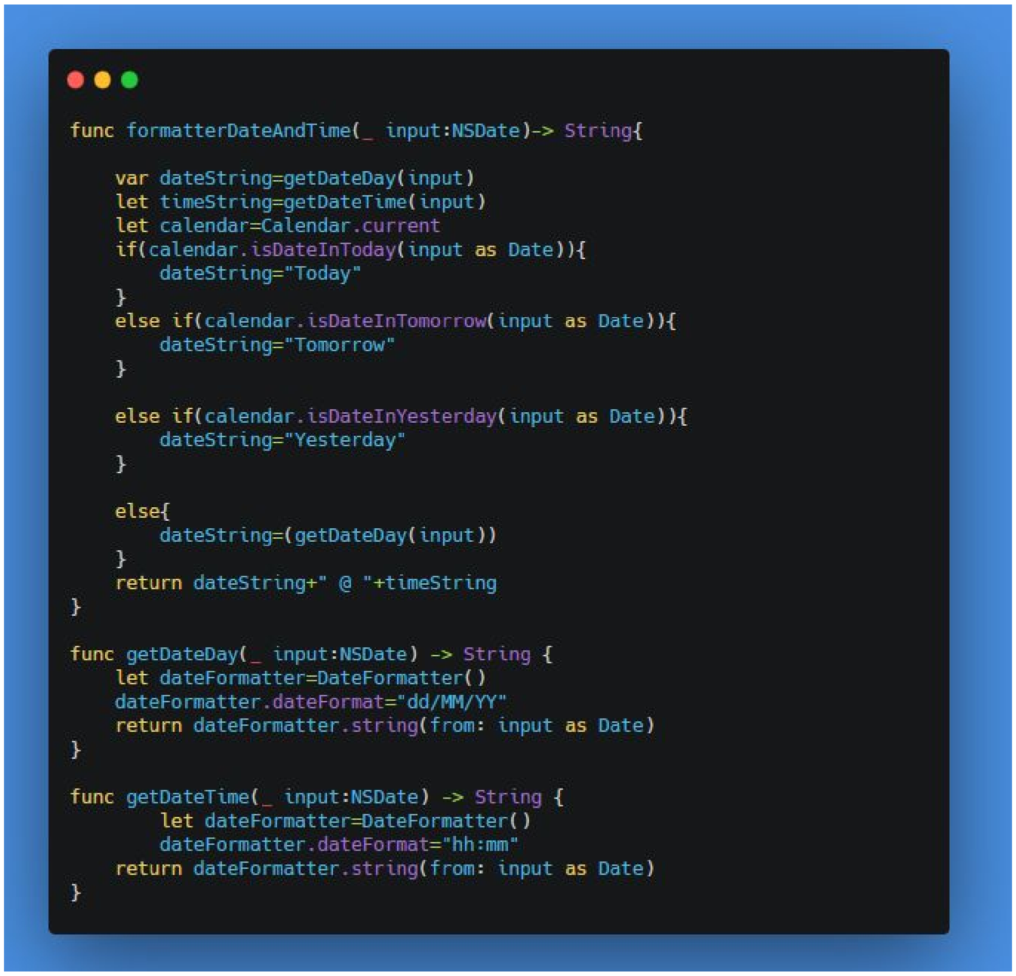
In this class we handle the logic behind the tasks inside a certain list. The hostingListID is an important field that saves the ID of the list where the currently present tasks reside.


```
if (editingStyle == UITableViewCell.EditingStyle.delete){

        guard let appDelegate = UIApplication.shared.delegate as? AppDelegate else{
                return
          }
         let managedContext = appDelegate.persistentContainer.viewContext
         managedContext.delete(dataHandler.getFRC().object(at: indexPath))
         let selectedObj = self.dataHandler.getFRC().object(at: indexPath)


         let taskID = selectedObj.value(forKey: "id") as! Int64


         dataHandler.deleteItem(taskID, condition: "id = \(taskID)" , isList: false)

         TasksTable.reloadData()
           }
```

- In this snippet of code, we add a custom swipe functionality that deletes the specified task if you swipe left


```
let complete = UIContextualAction(style: .normal, title: "Complete") { [self] (action, view, nil) in
            let completedPage = self.tabBarController as? TasksBarViewController
            let target = completedPage?.viewControllers![1] as? CompletedTasksTableViewController


            let data = ["isDone":1]

            let selectedObj = self.dataHandler.getFRC().object(at: indexPath)

            let taskID = selectedObj.value(forKey: "id") as! Int64
            self.dataHandler.updateItem(itemID: taskID, data, key: "id")
            self.TasksTable.reloadData()
```

- In this snippet of code, we add a custom swipe functionality that sets the task as completed after you swipe right. The logic above updates the property in the Core Data and consequently updates the tasks in the table and moves the completed task to the “Completed” tab.



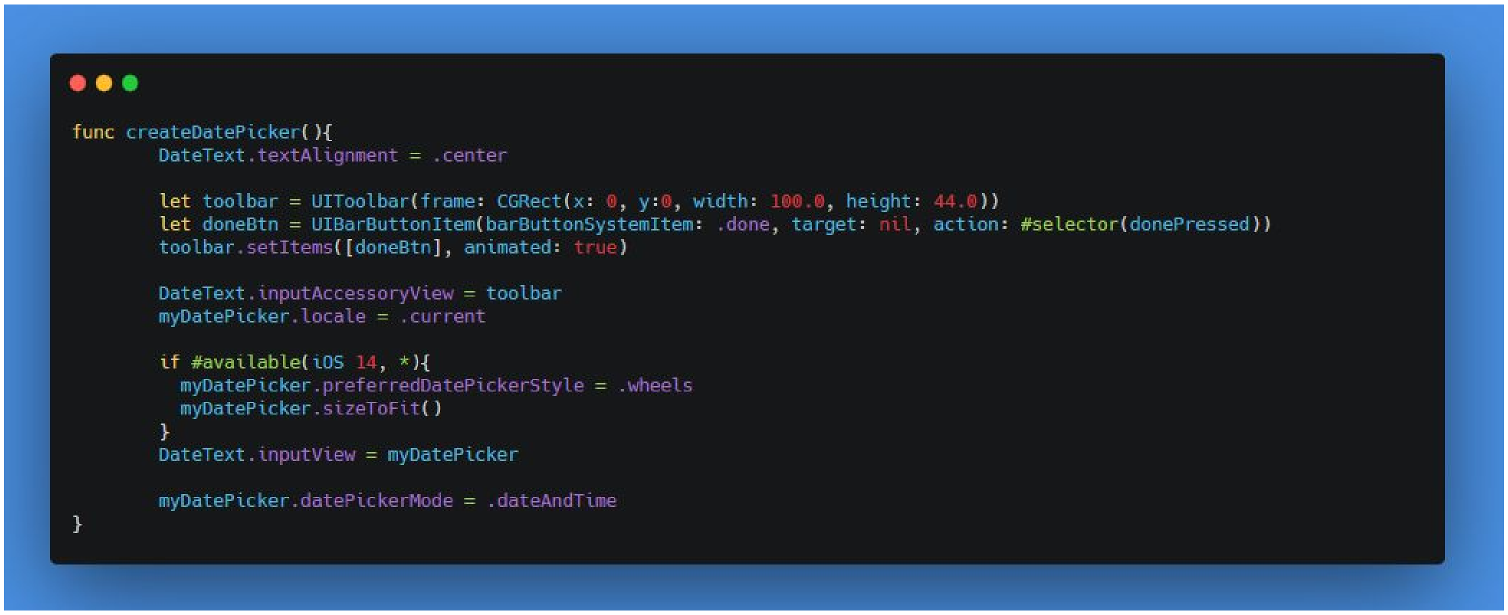
```swift
func formatterDateAndTime(_ input:NSDate)-> String{

    var dateString=getDateDay(input)
    let timeString=getDateTime(input)
    let calendar=Calendar.current
    if(calendar.isDateInToday(input as Date)){
        dateString="Today"
    }
    else if(calendar.isDateInTomorrow(input as Date)){
        dateString="Tomorrow"
    }

    else if(calendar.isDateInYesterday(input as Date)){
        dateString="Yesterday"
    }

    else{
        dateString=(getDateDay(input))
    }
    return dateString+" @ "+timeString
}

func getDateDay(_ input:NSDate) -> String {
    let dateFormatter=DateFormatter()
    dateFormatter.dateFormat="dd/MM/YY"
    return dateFormatter.string(from: input as Date)
}

func getDateTime(_ input:NSDate) -> String {
        let dateFormatter=DateFormatter()
        dateFormatter.dateFormat="hh:mm"
    return dateFormatter.string(from: input as Date)
}
```

- The formatter formats the due date in a humanized format in the tasks table (e.g. yesterday, today, tomorrow)

Add Tasks

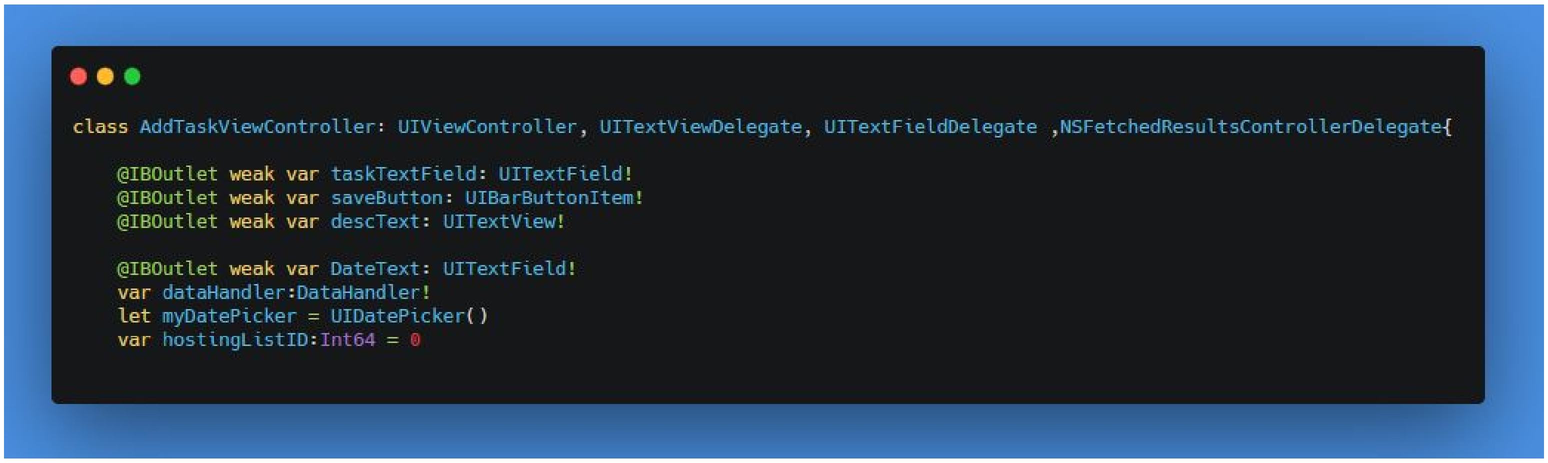
AddTaskViewController

This class handles the logic behind the Add task screen accessed when you click the “Add” button from the TaskTableView screen



```
func createDatePicker(){
        DateText.textAlignment = .center

        let toolbar = UIToolbar(frame: CGRect(x: 0, y:0, width: 100.0, height: 44.0))
        let doneBtn = UIBarButtonItem(barButtonSystemItem: .done, target: nil, action: #selector(donePressed))
        toolbar.setItems([doneBtn], animated: true)

        DateText.inputAccessoryView = toolbar
        myDatePicker.locale = .current

        if #available(iOS 14, *){
          myDatePicker.preferredDatePickerStyle = .wheels
          myDatePicker.sizeToFit()
        }
        DateText.inputView = myDatePicker

        myDatePicker.datePickerMode = .dateAndTime
}
```

- This function sets up a datePicker wheel where we set the default value to current date and time. This design choice was picked for feasibility of picking a date & time and better organization.



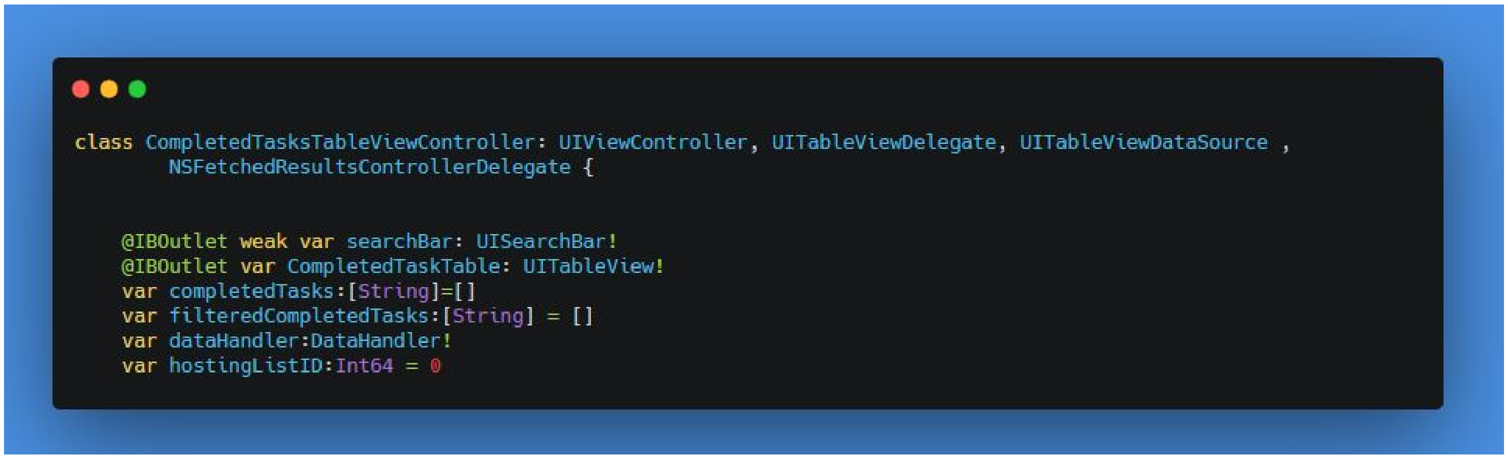
```
class AddTaskViewController: UIViewController, UITextViewDelegate, UITextFieldDelegate ,NSFetchedResultsControllerDelegate{

    @IBOutlet weak var taskTextField: UITextField!
    @IBOutlet weak var saveButton: UIBarButtonItem!
    @IBOutlet weak var descText: UITextView!

    @IBOutlet weak var DateText: UITextField!
    var dataHandler:DataHandler!
    let myDatePicker = UIDatePicker()
    var hostingListID:Int64 = 0
```

- These fields are used for formatting the screen properly & retrieving the required information to create a task
- the user will not be allowed to save the new task if the Task Title Field is empty

Completed Tasks

CompletedTasksTableViewController

This class handles the logic behind the “Completed” tab.



```
class CompletedTasksTableViewController: UIViewController, UITableViewDelegate, UITableViewDataSource ,
        NSFetchedResultsControllerDelegate {


    @IBOutlet weak var searchBar: UISearchBar!
    @IBOutlet var CompletedTaskTable: UITableView!
    var completedTasks:[String]=[]
    var filteredCompletedTasks:[String] = []
    var dataHandler:DataHandler!
    var hostingListID:Int64 = 0
```

- This screen provides you with all the tasks that you have completed under the list you entered from the ListViewController screen.
- Normal delete swipe functionality to clear deleted items along with updating the necessary tables

---

Future Components:

1. **Reminders**: The user can choose when to be notified about the certain tasks - when their deadline approaches.
2. **Extra Task Fields:** Newer fields to fill (allowing users to customize their tasks even more)
    - Example: Priority fields.
3. **Extra Sorting Options:** Allow the user to sort their task list by various options (such as date or priority) for their convenience.
4. **Add more visual settings:** Allow the user to choose different color themes(ex: dark mode).
5. **Implementing a database in the cloud:** Store tasks and lists for each user in a database in the cloud, so they could access their data anywhere and anytime, and to avoid losing their saved data in case something happens to their devices.
    - Example: *Firebase*

Improvements:

1. Integrate the search bar with Core Data. The issue we faced when implementing with Core Data is that the goal of putting the tasks into a list that could be edited by the search function was out of scope with the current implementation of Core Data.
2. The current implementation accesses the same container of lists & tasks for registered users. In the future, access to personalized lists & tasks based on the user logging in should be implemented.
3. Improve visual layout of rows. Rows were supposed to have a color scheme of alternating white and blue. However, when adding and editing, the color scheme would get mixed up , due to handling data outside the table view controller.
4. Add secure text (without the iOS auto-suggest overlay).
5. Allow the user to dismiss the keyboard if they tapped the enter key or anywhere on the screen. Issue faced was that a keyboard didn’t pop up for some simulators when developing the application so the issue went unnoticed.
6. Visual improvements by using more visually refined components(ex: task view cell would have a custom component to display due date rather than generic labels).