JOURNAL OF BAKU ENGINEERING UNIVERSITY- MECHANICAL AND INDUSTRIAL ENGINEERING
2025. Volume 9, Number 1 Pages 17-28

UOT: 539
DOI: https://doi.org/10.30546/09085.2025.01.011

THE ORGANIZATION OF CATERING SERVICES BASED ON INFORMATION TECHNOLOGY

Z. A. DJAFAROV
Azerbaijan Technical University, H.Javid ave 25, AZ 1073 Baku, Azerbaijan.
zafar.cafarov@aztu.edu.az

M. R. ABBASZADA
Azerbaijan Technical University, H.Javid ave 25, AZ 1073 Baku, Azerbaijan.
abbaszademurad3@gmail.com

ARTICLE INFO
Article history:
Received: 2025-03-05
Received in revised form: 2025-03-05
Accepted: 2025-04-02
Available online
Keywords:
Xcode, Swift, Apple, Food, Order, Navigation Controller, Bar button items, Root changing, User Defaults, File Manager, JSON
JEL Classification: L81, O33, L86

ABSTRACT
The article is dedicated to the use of information technologies in the process of ordering food through a mobile application. Here, users can get information about certain food items, learn their prices, and add their favorite dishes to the cart. Additionally, users must go through login and registration processes to use the application. The profile screen will display the details associated with the user's login. On the Food screen, users can complete the process by pressing the button on the dish they want to add to the cart. This application also provides users with fast delivery and payment options. Meals are divided into various categories, making the selection process easier. Each dish comes with detailed descriptions and reviews. Users can save their favorite dishes and reorder them later. The notification system keeps users informed about the status of their orders. The mobile application uses modern technologies to offer an intuitive and user-friendly interface. Ordering food is easy, fast, and secure. These features make the application stand out.

1. Introduction

Xcode—Xcode is a powerful integrated development environment (IDE) designed by Apple for the macOS platform. It is used to develop applications for iOS, macOS, watchOS, and tvOS. Xcode is intended for both beginner and experienced developers and is one of the most essential tools within the Apple ecosystem.

Key features of Xcode include:

• **Programming language support** – Supports languages such as Swift, Objective-C, Ruby, Python, C++, and C;

• **Simulators** – Allows testing applications on virtual devices for iPhone, iPad, Apple Watch, and Apple TV;

• **Interface Builder** – Works with storyboard and XIB formats;

• **Debugging** – An essential tool for identifying and fixing errors and bugs in real-time.

Xcode is the most optimal environment for developing Apple applications. It can be downloaded for free from the Mac App Store.

Xcode is an indispensable tool for development in the Apple ecosystem, offering powerful features that meet the needs of modern developers.

II. DESCRIPTION OF THE PROCESSING SYSTEM

First, to start the project, we need to download Xcode to our system. After the download process is complete, we open Xcode, click on "Create a new project," and select the "Single App" option. At this point, Xcode generates several functions and classes in the background to enable us to use the simulator and implement the code we will write. The stages of application development can be outlined as follows:

A. Configuring the login and registration screens.

B. Creating a custom class for the main (home) screen and using a collection view.

C. Creating helper classes.

D. Utilizing a JSON file.

E. Preparing the cart and profile screens using a navigation item.

A. Configuration of login and register screens

There are two methods for creating a login screen: using **storyboard** or implementing it programmatically. However, in this article, I will focus on extensive use of storyboard and XIB files. First, we add a view controller to the storyboard. To link this controller to a file, we create a separate view controller class. To establish the connection between the storyboard and the class, we navigate to the **Identity Inspector** section in the storyboard and assign the name of the created class to our screen. The same process can also be applied to the register screen. The register screen will require basic user information, such as the user's name, email, email password, and mobile number. Once the user provides this information, they can navigate to the login screen, where the data entered in the register screen will also be visible. This process is called **data transformation** and can be implemented using closures. Alternatively, it can also be achieved by creating a separate protocol and using a delegate. The information entered by the user during registration will be stored in a separate file, which will be used to verify the accuracy of the login details. If the entered login data does not match the information in the existing file, an error alert will appear on the screen. If the data is correct, the user will be redirected to the main (home) screen.To achieve this, the following UI elements will be used:

UITextField: For entering text inputs like username, email, and password.

- **UIButton**: For actions such as login and register.
- **UIView**: For creating custom animations.

To incorporate animations, one of the most optimal methods is using **Lottie files**. On the Lottie Files website, you can search for and find custom animations that suit your app's needs and integrate them into your code. To use Lottie in Xcode, you can download the Lottie package manager from GitHub. Once the Lottie library is integrated into your project, you can utilize it to enhance your application with custom animations. *(Fig. 1)*

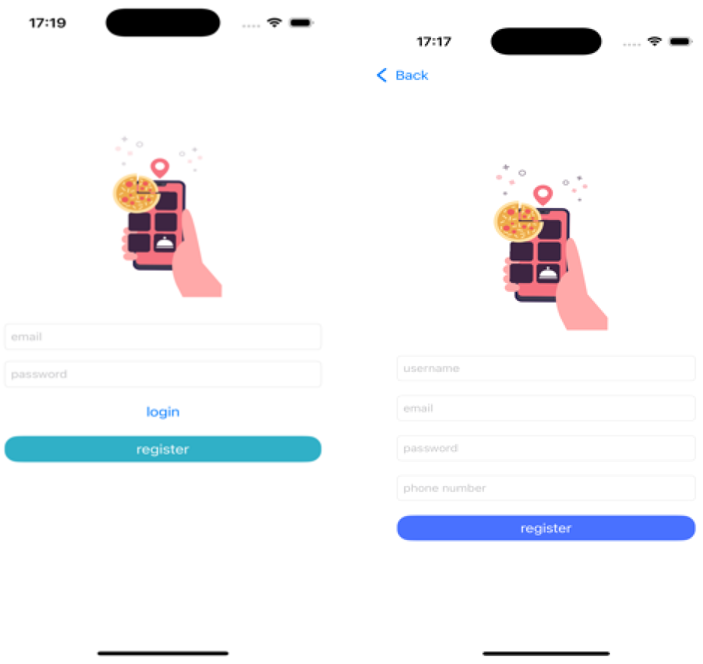
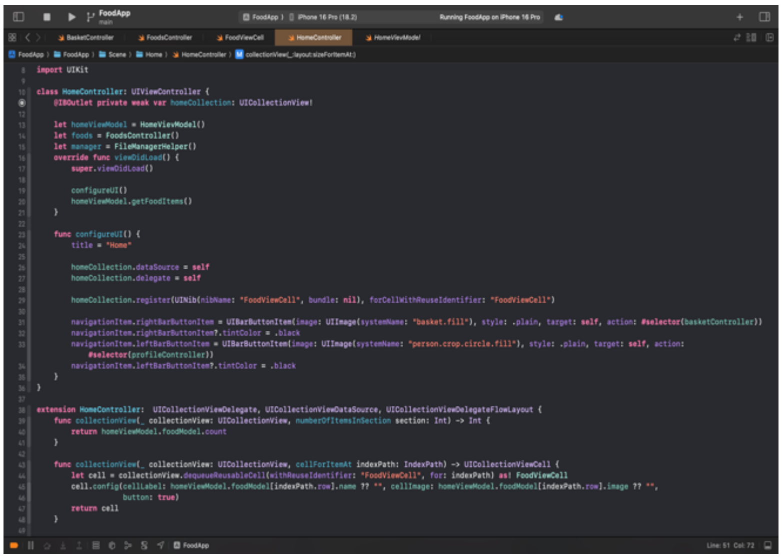


Fig 1. Login and register screens

B. Creating a Custom Class for the Main (Home) Screen and Using Collection View

A **collection view** is used on Apple devices to display multiple images or data items either side-by-side or top-to-bottom. Examples of collection views can be found in market applications or apps for selling clothing or goods. For the home screen, we assign a collection view to the controller in the storyboard. However, we create a **custom class** for the cells and define the cell size and the data to be displayed within that custom class. In addition to the custom class, we also create a **XIB file** for the cell. This XIB file will be used across two screens. One of the advantages of using XIB files is that if the same cell is required on multiple screens, creating it once is sufficient. After creating the cell class, we register it to the collection view on the home screen and call certain protocols to configure it. *(Fig. 2)*


FoodApp
main
FoodApp › iPhone 16 Pro (18.2)
Running FoodApp on iPhone 16 Pro
BasketController
FoodsController
FoodViewCell
HomeController
HomeViewModel
FoodApp › FoodApp › Scene › Home › HomeController › collectionView(_:layout:sizeForItemAt:)
FoodApp
Line: 51 Col: 72

```swift
import UIKit

class HomeController: UIViewController {
    @IBOutlet private weak var homeCollection: UICollectionView!

    let homeViewModel = HomeViewModel()
    let foods = FoodsController()
    let manager = FileManagerHelper()
    override func viewDidLoad() {
        super.viewDidLoad()

        configureUI()
        homeViewModel.getFoodItems()
    }

    func configureUI() {
        title = "Home"

        homeCollection.dataSource = self
        homeCollection.delegate = self

        homeCollection.register(UINib(nibName: "FoodViewCell", bundle: nil), forCellWithReuseIdentifier: "FoodViewCell")

        navigationItem.rightBarButtonItem = UIBarButtonItem(image: UIImage(systemName: "basket.fill"), style: .plain, target: self, action: #selector(basketController))
        navigationItem.rightBarButtonItem?.tintColor = .black
        navigationItem.leftBarButtonItem = UIBarButtonItem(image: UIImage(systemName: "person.crop.circle.fill"), style: .plain, target: self, action:
            #selector(profileController))
        navigationItem.leftBarButtonItem?.tintColor = .black
    }
}

extension HomeController:  UICollectionViewDelegate, UICollectionViewDataSource, UICollectionViewDelegateFlowLayout {
    func collectionView(_ collectionView: UICollectionView, numberOfItemsInSection section: Int) -> Int {
        return homeViewModel.foodModel.count
    }

    func collectionView(_ collectionView: UICollectionView, cellForItemAt indexPath: IndexPath) -> UICollectionViewCell {
        let cell = collectionView.dequeueReusableCell(withReuseIdentifier: "FoodViewCell", for: indexPath) as! FoodViewCell
        cell.config(cellLabel: homeViewModel.foodModel[indexPath.row].name ?? "", cellImage: homeViewModel.foodModel[indexPath.row].image ?? "",
                    button: true)
        return cell
    }

```

Fig 2. Home controller code description

To keep the code clean and organized in this controller, the following steps can be taken:

• Create a separate function to handle the logic inside **viewDidLoad** and call that function within **viewDidLoad**.

• To avoid cluttering the HomeController class, write an **extension** for it.

• In the extension, call the necessary protocols required for the HomeController. These protocols were mentioned above. Once we conform to these protocols in the extension, some functionalities will need to be integrated into the code. These functionalities are numberOfItemsInSection and cellForItemAt. With the help of these functions, we can display data on the screen and handle navigation between screens.

• Navigation between screens is configured using the didSelectItemAt function. This function allows us to display data and details on the food screen based on the cell clicked on the Home screen.

The Home screen consists of 6 cells, and the **navigation bar** contains 2 items. *(Fig. 3)*

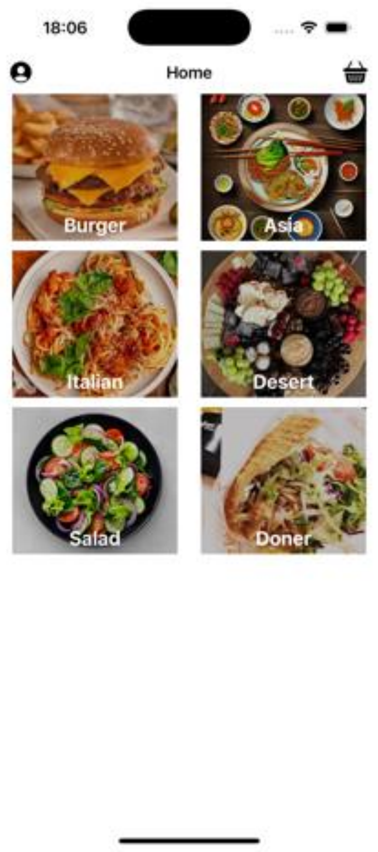
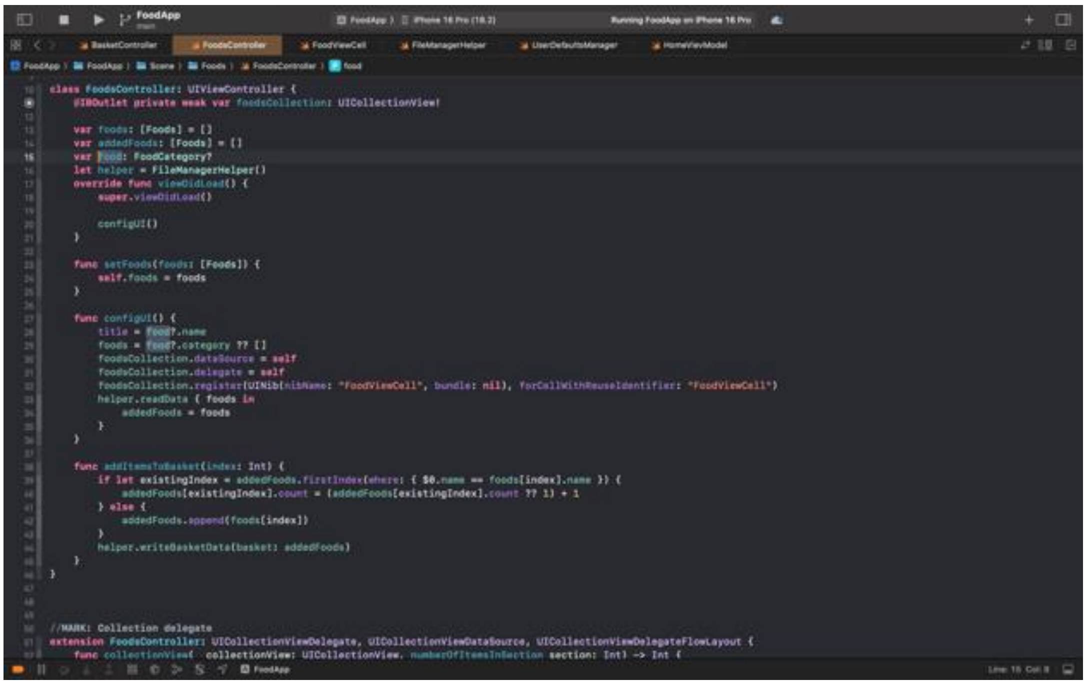


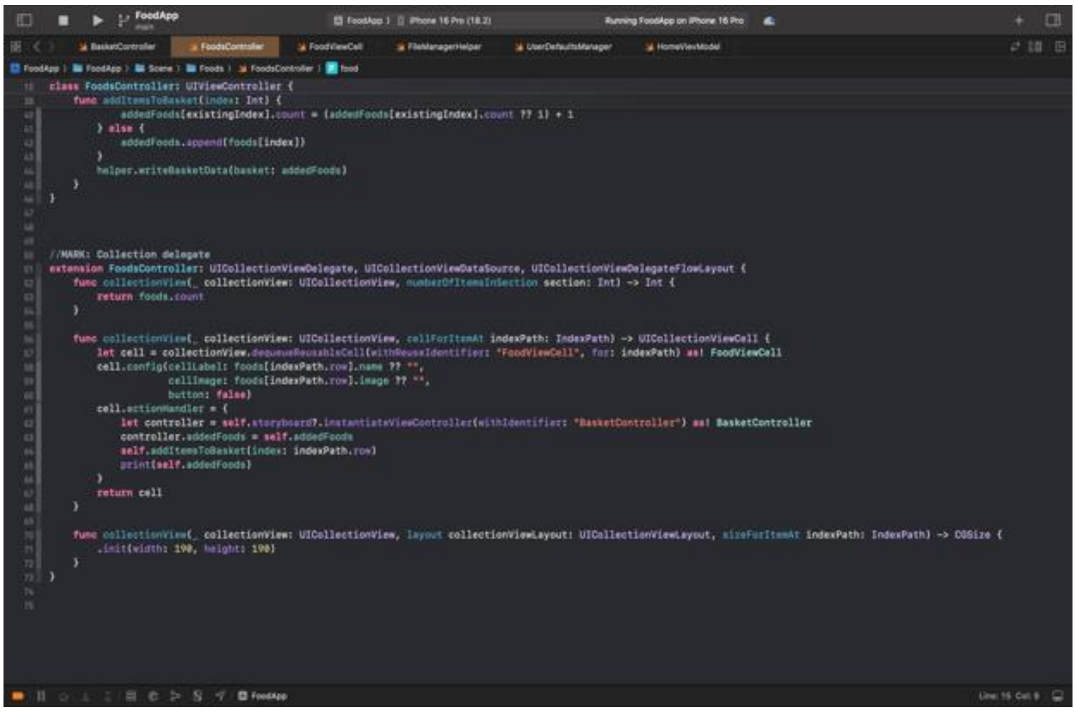
Fig 3. Home controller

When a user clicks on a cell in this view, the corresponding food items will be displayed, and with each click on the button within the food item, that food will be added to the basket screen. In this controller, we will use the same **XIB file** that was used in the Home screen. The only difference is that this XIB file will be registered with the collection view in the **FoodController**. Additionally, as mentioned above, we will conform to the necessary protocols via an extension. This setup will ensure proper functionality for the food selection and basket actions. *(Fig. 4)*



```swift
class FoodsController: UIViewController {
    @IBOutlet private weak var foodsCollection: UICollectionView!

    var foods: [Foods] = []
    var addedFoods: [Foods] = []
    var food: FoodCategory?
    let helper = FileManagerHelper()
    override func viewDidLoad() {
        super.viewDidLoad()

        configUI()
    }

    func setFoods(foods: [Foods]) {
        self.foods = foods
    }

    func configUI() {
        title = food?.name
        foods = food?.category ?? []
        foodsCollection.dataSource = self
        foodsCollection.delegate = self
        foodsCollection.register(UINib(nibName: "FoodViewCell", bundle: nil), forCellWithReuseIdentifier: "FoodViewCell")
        helper.readData { foods in
            addedFoods = foods
        }
    }

    func addItemsToBasket(index: Int) {
        if let existingIndex = addedFoods.firstIndex(where: { $0.name == foods[index].name }) {
            addedFoods[existingIndex].count = (addedFoods[existingIndex].count ?? 1) + 1
        } else {
            addedFoods.append(foods[index])
        }
        helper.writeBasketData(basket: addedFoods)
    }
}


//MARK: Collection delegate
extension FoodsController: UICollectionViewDelegate, UICollectionViewDataSource, UICollectionViewDelegateFlowLayout {
    func collectionView(_ collectionView: UICollectionView, numberOfItemsInSection section: Int) -> Int {
```

Fig 4. FoodController class



```swift
class FoodsController: UIViewController {
    func addItemsToBasket(index: Int) {
            addedFoods[existingIndex].count = (addedFoods[existingIndex].count ?? 1) + 1
        } else {
            addedFoods.append(foods[index])
        }
        helper.writeBasketData(basket: addedFoods)
    }
}


//MARK: Collection delegate
extension FoodsController: UICollectionViewDelegate, UICollectionViewDataSource, UICollectionViewDelegateFlowLayout {
    func collectionView(_ collectionView: UICollectionView, numberOfItemsInSection section: Int) -> Int {
        return foods.count
    }

    func collectionView(_ collectionView: UICollectionView, cellForItemAt indexPath: IndexPath) -> UICollectionViewCell {
        let cell = collectionView.dequeueReusableCell(withReuseIdentifier: "FoodViewCell", for: indexPath) as! FoodViewCell
        cell.config(cellLabel: foods[indexPath.row].name ?? "",
                    cellImage: foods[indexPath.row].image ?? "",
                    button: false)
        cell.actionHandler = {
            let controller = self.storyboard?.instantiateViewController(withIdentifier: "BasketController") as! BasketController
            controller.addedFoods = self.addedFoods
            self.addItemsToBasket(index: indexPath.row)
            print(self.addedFoods)
        }
        return cell
    }

    func collectionView(_ collectionView: UICollectionView, layout collectionViewLayout: UICollectionViewLayout, sizeForItemAt indexPath: IndexPath) -> CGSize {
        .init(width: 190, height: 190)
    }
}
```

Fig 4. FoodController extension

In the **FoodController**, we add an "Add" button to the cell, allowing the user to add the desired food items to the basket screen by pressing the button. *(Fig. 5)*

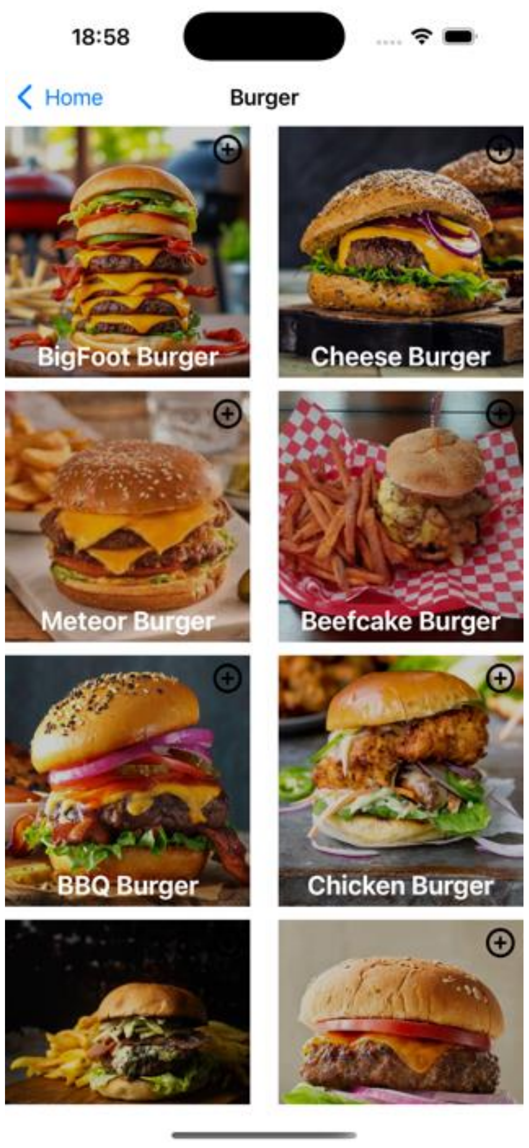


Fig 5. Food Controller screen

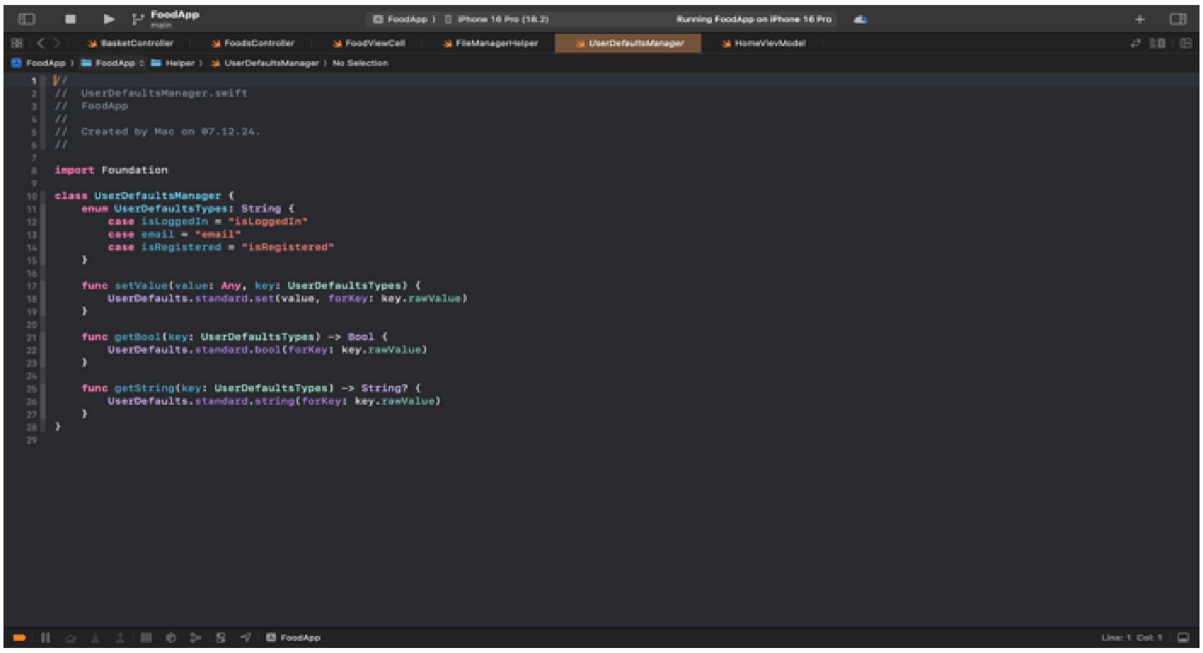
Creating helper classes

In this project, there are **2 helper classes** and **3 struct models** used. The struct models are **Foods**, **FoodCategory**, and **User**, and they are utilized for data transfer between screens and displaying data on the UI. The helper classes are **FileManagerHelper** and **UserDefaultsManager**. These classes serve different purposes.

• **UserDefaultsManager** is used mainly for handling small-scale data, such as checking whether the user is logged in or not.

• **FileManagerHelper** is used for storing the user's data in a file when they register.

However, there's an important nuance here: both **UserDefaults** and **FileManager** store data inside the application, meaning that if the app is deleted, all the stored data will be wiped out. In such cases, using **Keychain** is more appropriate because it stores data on the phone itself, not within the app. Keychain is better suited for storing larger data and handling transformations. The best approach for using **UserDefaults** is to create a dedicated class for it. The class mentioned above is designed to handle this. Inside the class, **enums** and functions are used, with the enum simplifying the creation of instances. This makes the code more readable and easier for other developers to understand. This class will be used mainly in the **SceneDelegate** and **Profile screen**. In the **SceneDelegate**, we check if the user is logged in using **UserDefaults**. If they are, this information is saved in **UserDefaults**. When the user clicks the login button, the information will be saved in memory. At this point, the root screen of the app changes. The **root** determines which screen the user will see when they open the app for the first time or the second time. Logically, if the user is already logged in or registered, these screens will not be shown again. Therefore, when the user clicks the login button, the root will change to the **Home screen**. From then on, whenever the user enters the app, they will be greeted with the Home screen. *(Fig. 6)*

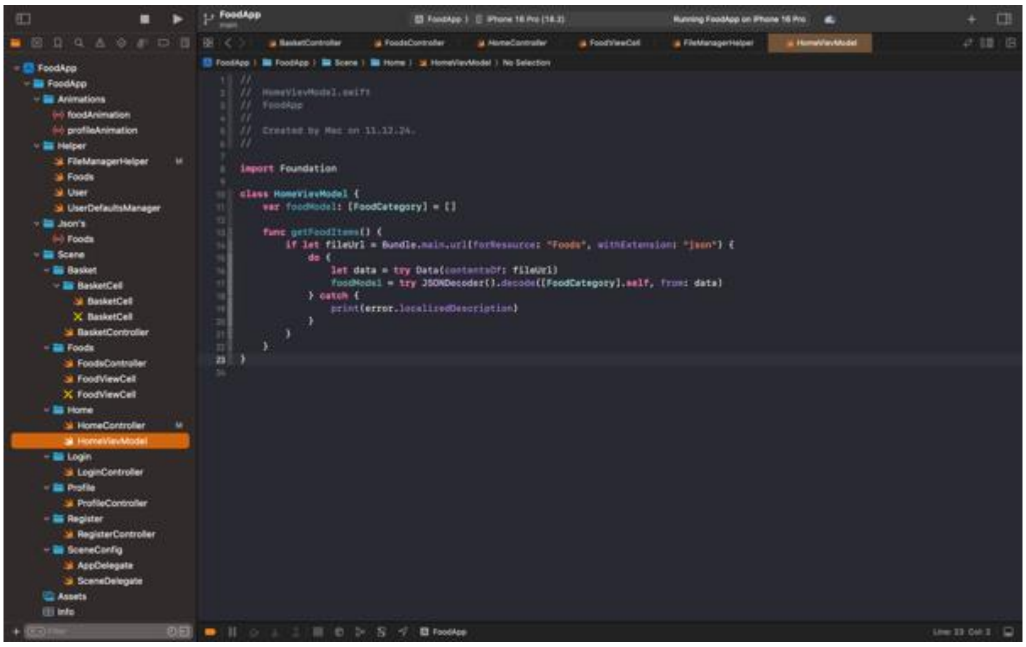


```
//
//  UserDefaultsManager.swift
//  FoodApp
//
//  Created by Mac on 07.12.24.
//

import Foundation

class UserDefaultsManager {
    enum UserDefaultsTypes: String {
        case isLoggedIn = "isLoggedIn"
        case email = "email"
        case isRegistered = "isRegistered"
    }

    func setValue(value: Any, key: UserDefaultsTypes) {
        UserDefaults.standard.set(value, forKey: key.rawValue)
    }

    func getBool(key: UserDefaultsTypes) -> Bool {
        UserDefaults.standard.bool(forKey: key.rawValue)
    }

    func getString(key: UserDefaultsTypes) -> String? {
        UserDefaults.standard.string(forKey: key.rawValue)
    }
}
```

Fig 6. UserDefaultsManager class

Using JSON file

JSON files are typically used when fetching data from APIs. However, in this current project, you can also use a JSON file to display data, but you will have to manually input the data into the JSON file. The data inside the JSON file must match the structure of the struct models so that the data can be transferred from the JSON file to the struct model. If they do not match, no data will be displayed on the screen, and an error message will be shown in the console. There is a common method to transfer data from JSON to the struct model, which is widely used. *(Fig. 7)*



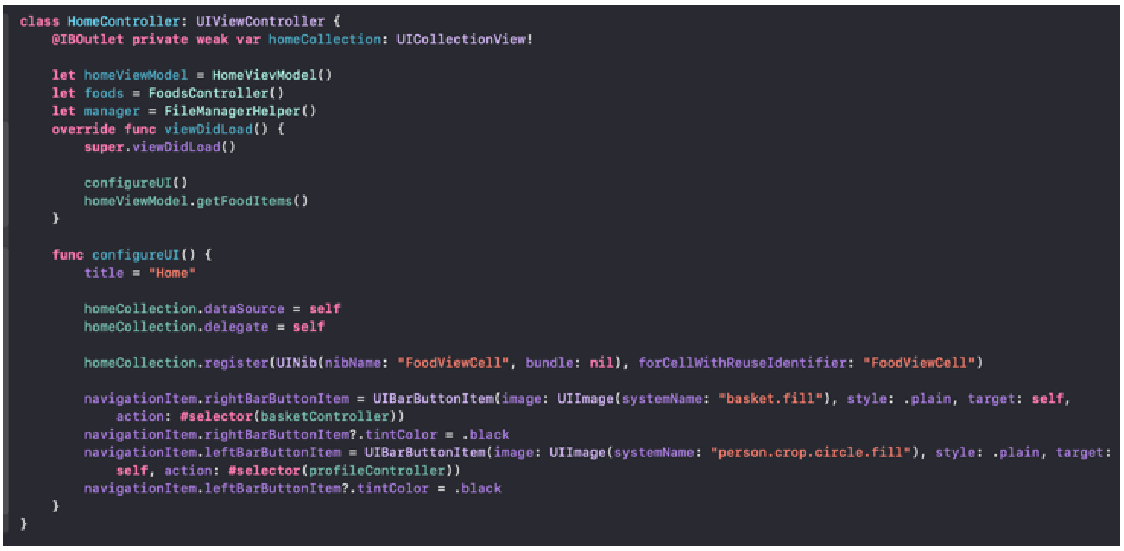
```swift
//
//  HomeViewModel.swift
//  FoodApp
//
//  Created by Mac on 11.12.24.
//

import Foundation

class HomeViewModel {
    var foodModel: [FoodCategory] = []

    func getFoodItems() {
        if let fileUrl = Bundle.main.url(forResource: "Foods", withExtension: "json") {
            do {
                let data = try Data(contentsOf: fileUrl)
                foodModel = try JSONDecoder().decode([FoodCategory].self, from: data)
            } catch {
                print(error.localizedDescription)
            }
        }
    }
}
```

Fig 7. Reading from JSON file

As seen, data is read from the JSON file using the **JSONDecoder** function. We assign the decoded struct to the instance we created. For this, the instance must be of a variable type. If it were a **let** constant, the compiler would throw an error.

Creating the basket and profile screens using navigation items

If you want to navigate to other screens from the home screen using the navigation item, you first need to assign a button to the navigation item in the code. When the user clicks on a button, the navigation controller will ensure the transition between screens and display the content of the respective screen. For navigation items, system-provided images are used to make the user interface more visually appealing. The basket button will be on the right side, and the profile button will be on the left. Thus, by assigning Objective-C-style selector functionality to each button, we define which screen will be shown when the user clicks on each button. This is how the navigation item is provided in the code. (Fig 8).

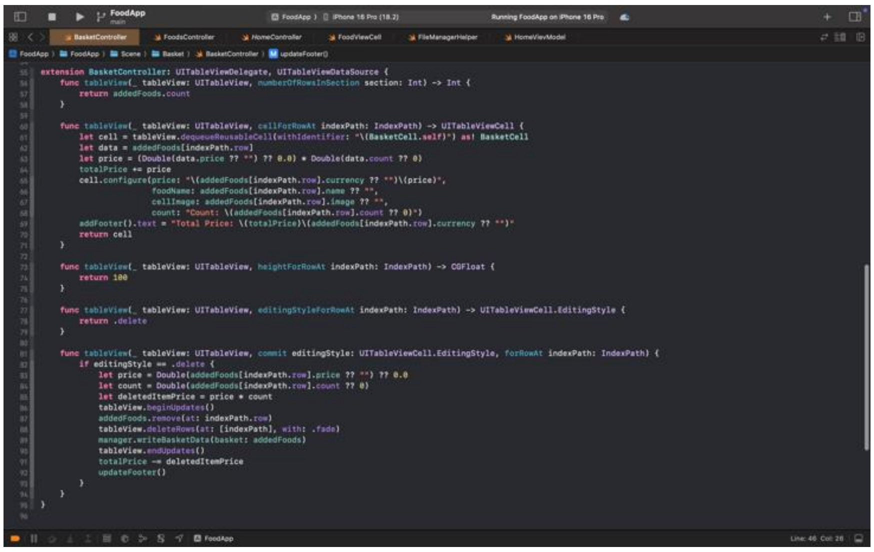


```swift
class HomeController: UIViewController {
    @IBOutlet private weak var homeCollection: UICollectionView!

    let homeViewModel = HomeViewModel()
    let foods = FoodsController()
    let manager = FileManagerHelper()
    override func viewDidLoad() {
        super.viewDidLoad()

        configureUI()
        homeViewModel.getFoodItems()
    }

    func configureUI() {
        title = "Home"

        homeCollection.dataSource = self
        homeCollection.delegate = self

        homeCollection.register(UINib(nibName: "FoodViewCell", bundle: nil), forCellWithReuseIdentifier: "FoodViewCell")

        navigationItem.rightBarButtonItem = UIBarButtonItem(image: UIImage(systemName: "basket.fill"), style: .plain, target: self,
            action: #selector(basketController))
        navigationItem.rightBarButtonItem?.tintColor = .black
        navigationItem.leftBarButtonItem = UIBarButtonItem(image: UIImage(systemName: "person.crop.circle.fill"), style: .plain, target:
            self, action: #selector(profileController))
        navigationItem.leftBarButtonItem?.tintColor = .black
    }
}
```

Fig 8. Navigation Items

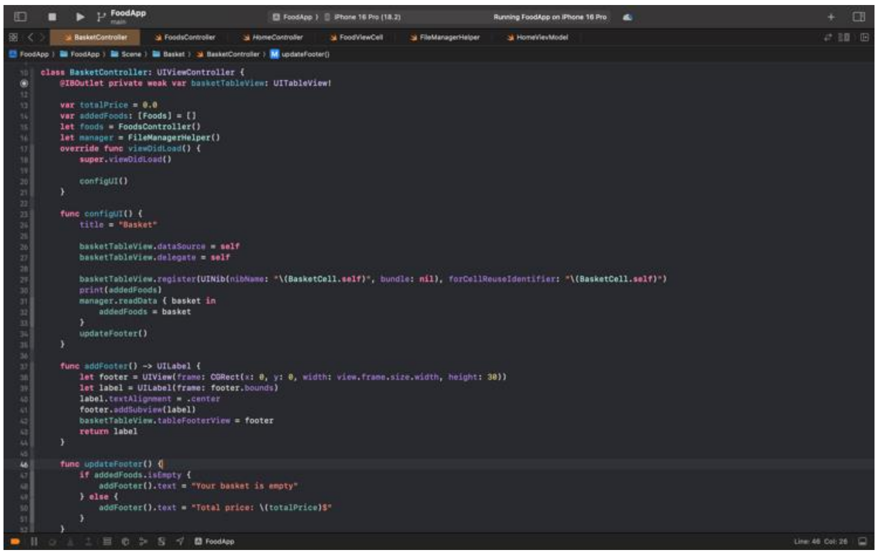
Thus, each button has its own functionality.

In the **Basket screen**, a **TableView** is used, and for the table, a custom class and xib file are created. In the xib file, the design is structured according to the data to be displayed on the screen. Inside the cell, **UIImageView** and **UILabels** are used. Three main labels will be used here: one will display the name of the added food, one will show the quantity of the food, and the other will show the total price of the food based on the quantity. Some mathematical functionalities are also applied here. Below the **TableView**, a **TableFooter** is used. The TableFooter will remain fixed at the bottom of the table, even as the table is scrolled down. The footer will calculate the total amount of the added foods. (Fig 9).



```swift
extension BasketController: UITableViewDelegate, UITableViewDataSource {
    func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        return addedFoods.count
    }

    func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
        let cell = tableView.dequeueReusableCell(withIdentifier: "\(BasketCell.self)") as! BasketCell
        let data = addedFoods[indexPath.row]
        let price = (Double(data.price ?? "") ?? 0.0) * Double(data.count ?? 0)
        totalPrice += price
        cell.configure(price: "\(addedFoods[indexPath.row].currency ?? "")\(price)",
                       foodName: addedFoods[indexPath.row].name ?? "",
                       cellImage: addedFoods[indexPath.row].image ?? "",
                       count: "Count: \(addedFoods[indexPath.row].count ?? 0)")
        addFooter().text = "Total Price: \(totalPrice)\(addedFoods[indexPath.row].currency ?? "")"
        return cell
    }

    func tableView(_ tableView: UITableView, heightForRowAt indexPath: IndexPath) -> CGFloat {
        return 100
    }

    func tableView(_ tableView: UITableView, editingStyleForRowAt indexPath: IndexPath) -> UITableViewCell.EditingStyle {
        return .delete
    }

    func tableView(_ tableView: UITableView, commit editingStyle: UITableViewCell.EditingStyle, forRowAt indexPath: IndexPath) {
        if editingStyle == .delete {
            let price = Double(addedFoods[indexPath.row].price ?? "") ?? 0.0
            let count = Double(addedFoods[indexPath.row].count ?? 0)
            let deletedItemPrice = price * count
            tableView.beginUpdates()
            addedFoods.remove(at: indexPath.row)
            tableView.deleteRows(at: [indexPath], with: .fade)
            manager.writeBasketData(basket: addedFoods)
            tableView.endUpdates()
            totalPrice -= deletedItemPrice
            updateFooter()
        }
    }
}
```

Fig 9. Calculation of the total quantity and price of the food.



```swift
class BasketController: UIViewController {
    @IBOutlet private weak var basketTableView: UITableView!

    var totalPrice = 0.0
    var addedFoods: [Foods] = []
    let foods = FoodsController()
    let manager = FileManagerHelper()
    override func viewDidLoad() {
        super.viewDidLoad()

        configUI()
    }

    func configUI() {
        title = "Basket"

        basketTableView.dataSource = self
        basketTableView.delegate = self

        basketTableView.register(UINib(nibName: "\(BasketCell.self)", bundle: nil), forCellReuseIdentifier: "\(BasketCell.self)")
        print(addedFoods)
        manager.readData { basket in
            addedFoods = basket
        }
        updateFooter()
    }

    func addFooter() -> UILabel {
        let footer = UIView(frame: CGRect(x: 0, y: 0, width: view.frame.size.width, height: 30))
        let label = UILabel(frame: footer.bounds)
        label.textAlignment = .center
        footer.addSubview(label)
        basketTableView.tableFooterView = footer
        return label
    }

    func updateFooter() {
        if addedFoods.isEmpty {
            addFooter().text = "Your basket is empty"
        } else {
            addFooter().text = "Total price: \(totalPrice)$"
        }
    }
```

Fig 9. Configuration of basket screen

As seen here, the register function is also used for the TableView. By registering, we specify that the xib file will be inside that TableView. The user can delete the foods they have added to the basket at any time. When this happens, the price of the deleted item will be subtracted from the total amount. If we express the calculation of the foods in a mathematical formula, it would look like this:

count = addedFoods.count

price = addedFoods.price * count

deletedItemPrice = price * count

totalPrice = totalPrice – deletedItemPrice

it can be shown as follows. (Fig. 10).

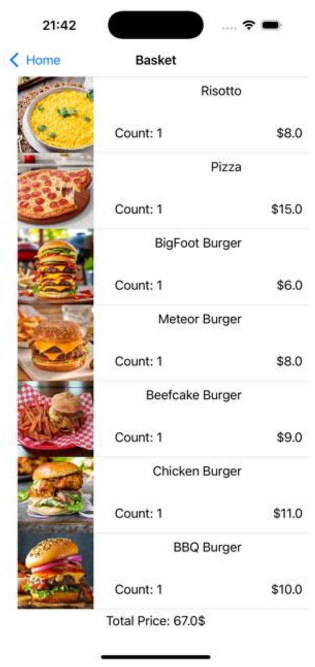
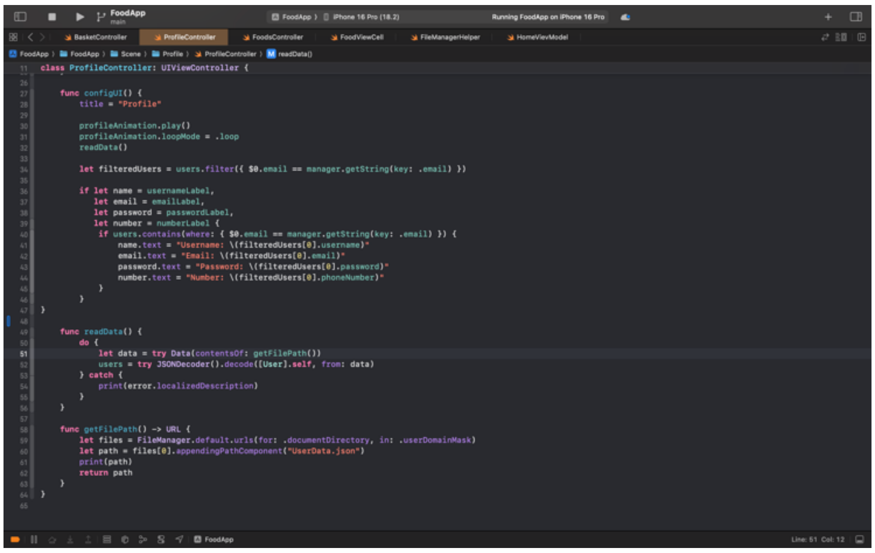


Fig 10. Basket screen

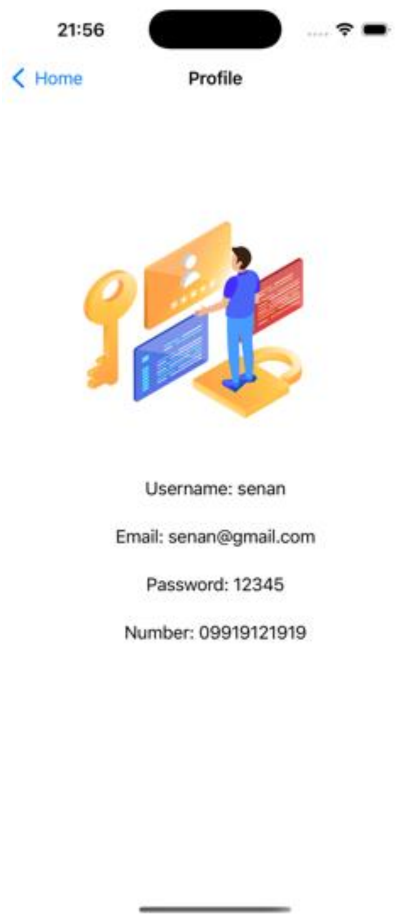
In the profile screen, the data associated with the email the user logged in with will be displayed. To achieve this, when the user clicks the login button on the login screen, we use **UserDefaults**. We store the logged-in email in the app's memory using **UserDefaults**. Thus, in the profile screen, the data corresponding to the logged-in email will be displayed. This data will be shown on the screen using a filter function. (Fig 11).



```
class ProfileController: UIViewController {

    func configUI() {
        title = "Profile"

        profileAnimation.play()
        profileAnimation.loopMode = .loop
        readData()

        let filteredUsers = users.filter({ $0.email == manager.getString(key: .email) })

        if let name = usernameLabel,
           let email = emailLabel,
           let password = passwordLabel,
           let number = numberLabel {
            if users.contains(where: { $0.email == manager.getString(key: .email) }) {
                name.text = "Username: \(filteredUsers[0].username)"
                email.text = "Email: \(filteredUsers[0].email)"
                password.text = "Password: \(filteredUsers[0].password)"
                number.text = "Number: \(filteredUsers[0].phoneNumber)"
            }
        }
}

    func readData() {
        do {
            let data = try Data(contentsOf: getFilePath())
            users = try JSONDecoder().decode([User].self, from: data)
        } catch {
            print(error.localizedDescription)
        }
    }

    func getFilePath() -> URL {
        let files = FileManager.default.urls(for: .documentDirectory, in: .userDomainMask)
        let path = files[0].appendingPathComponent("UserData.json")
        print(path)
        return path
    }
}
```

Fig 11. Configuration of profile screen

With the filter function, we filter from the existing instance and assign the filtered result to a new instance. During the filtering process, we retrieve the data by matching the email from **UserDefaults**. This way, the user will be able to see the account details of the last login. (Fig 12).



Username: senan

Email: senan@gmail.com

Password: 12345

Number: 09919121919

Fig 12. Profile screen

III. PERFORMANCE IMPROVEMENT

The article presents an analysis of the development of a mobile application for the food ordering process. While the proposed system is functionally appropriate, the following suggestions could be made to further improve performance and user experience:

• **Use of Database Instead of User Defaults and FileManager**: Transition from using User Defaults and FileManager to a more robust database solution like Core Data or SQLite to handle large volumes of data more efficiently.

• **Increase Asynchronous Processes**: Enhance the app's responsiveness by utilizing asynchronous processes for tasks such as network requests, image loading, and database operations, ensuring smoother performance and a better user experience.

• **Optimize Memory Usage**: Use memory efficiently by implementing strategies like data caching and minimizing memory leaks, which can improve the overall performance of the app.

• **Improve Network Performance**: Optimize network operations by implementing techniques like data compression, background fetching, and error handling to reduce latency and enhance the speed of data transfers.

• **Testing and Monitoring**: Incorporate extensive testing (unit tests, UI tests) and continuous monitoring to identify potential issues early, ensuring the app works seamlessly across different devices and network conditions.

This article not only addresses the technical aspects of app development but also focuses on solutions aimed at enhancing user experience. Each suggested feature highlights effective methods used in iOS programming for better performance and user satisfaction.

Conclusion and discussions

Using FileManager and UserDefaults as a small-scale database is aimed at simplifying the user's experience and making the food ordering process more accessible. This ensures that user data is neither lost nor altered unintentionally. By leveraging FileManager, a secure environment is created, which is crucial for safely storing and accessing user data. This approach ensures that the information is protected and easily retrievable when needed, contributing to a more reliable and secure app experience.

REFERENCES

1. 1.Khruleva N.D. Features of programming in the Swift language. International journal of Professional Sceince No 11-2021, p 115-119.
2. Scott, Susan V. and Zachariadis, Markos (2012) Origins and development of SWIFT, 1973–2009. Business History, 54 (3). pp. 462-482. ISSN 0007-6791.
3. Martin, R. C. (2008). Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall.
4. Tian, Y., Liu, X., & Zhang, X. (2021). "Performance Optimization Techniques for Mobile Applications: A Survey." Journal of Software Engineering and Applications, 14(3), 156-170.
5. Gupta, S., & Singh, R. (2020). "Cloud-Based Food Delivery Applications: Challenges and Future Trends." International Journal of Computer Applications, 182(45), 12-18.
6. Hossain, M. T., & Hossain, M. S. (2022). "Design and Performance Analysis of a Scalable Food Ordering System Using Microservices." IEEE Access, 10, 12345-12360.
7. Data Structures & Algorithms in Swift Implementing Practical Data Structures with Swift By the raywenderlich Tutorial Team Kelvin Lau & Vincent Ngo
8. Data Structures and Algorithms in Swift - Implementing practical data structures with Swift 4
9. Swift Programming: The Big Nerd Ranch Guide by Matthew Mathias and John Gallagher
10. Learn Swift by examples Beginner level SIMPLE INTRODUCTION SERIES