

Nutrition.



Inspiration and Purpose:

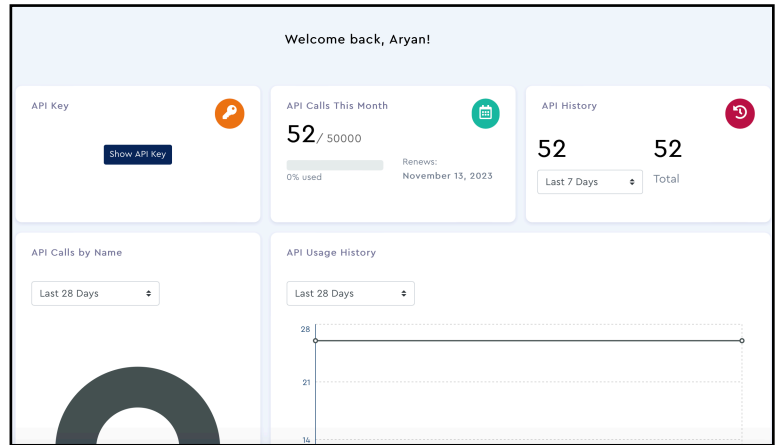
The Nutrition Viewer app is designed to help users view detailed nutritional information about various foods, including details such as calories, serving size, fat content, saturated fat content, protein content, sodium, potassium, cholesterol, carbohydrates, fiber, and sugar. The app also allows users to save their favourite foods and view them later in a dedicated list.

Users, such as, Gym trainers and fitness enthusiasts can quickly access detailed nutritional information for a wide range of foods. This feature is essential for designing tailored diet plans for clients and personal fitness goals.

The app provides accurate data on calories, macronutrients (such as protein, carbohydrates, and fats), as well as micronutrients (including sodium, potassium, and cholesterol). This precision aids trainers and fitness enthusiasts in creating nutrition plans that align with specific fitness goals.

Functionality:

- I. **Data Retrieval:** The app fetches food data from the NinjaAPI, which provides a database of nutritional information for various foods.



Img 1: API Dashboard

The ObservableObject protocol is a crucial component within the Combine framework, is used by the app within classes and models to manage state. These classes store data, while the @ObservedObject property wrapper is integrated within views to maintain instances of observable objects. Moreover, the @Published property wrapper is used to indicate properties that should trigger change notifications. A simple placement of @Published before a property ensures that it automatically updates SwiftUI views that are observing changes.

URLSession:

URLSession is a key element in the process. It is the tool employed to download data from and upload data to specific endpoints indicated by URLs. Multiple URLSession instances can be created in an application, and each instance handles a group of interrelated data-transfer tasks. Additionally, URLSession includes a shared session singleton, tailored for managing basic requests.

```
class Api : ObservableObject{
    @Published var foods = [Food]()

    func loadData(query: String, completion: @escaping ([Food]) -> ()) {
        let query = query.addingPercentEncoding(withAllowedCharacters: .urlQueryAllowed)
        let url = URL(string: "https://api.api-ninjas.com/v1/nutrition?query=" + query!)!
        var request = URLRequest(url: url)
        request.setValue("4CmqgsKqgzM2E1WXkSLYZg==PNV0c5TeX467XS7u", forHTTPHeaderField: "X-API-Key")
        URLSession.shared.dataTask(with: request) { data, response, error in
            if let data = data {
                if let foods = try? JSONDecoder().decode([Food].self, from: data) {
                    DispatchQueue.main.async {
                        completion(foods)
                    }
                } else {
                    print("Failed to decode JSON response")
                }
            } else {
                print("No data received from the API")
            }
        }.resume()
    }
}
```

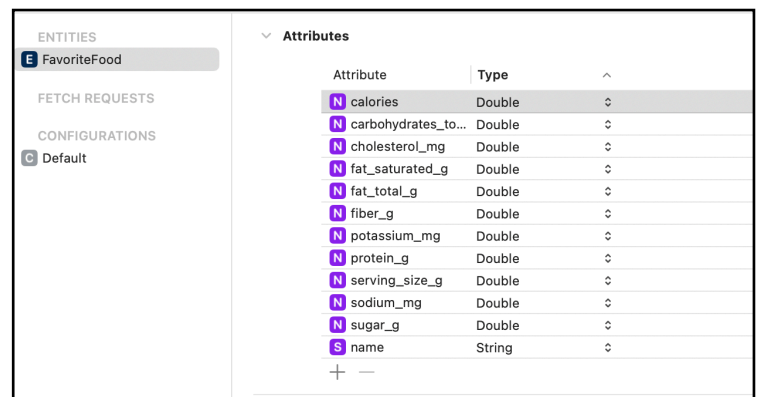
Img 2: ObservableObjects, URLSession and Dispatch queue

DispatchQueue:

DispatchQueue, as a manager of task execution, can work either serially or concurrently, whether on the main thread of the app or on a background thread. This mechanism is employed to asynchronously send signals for UI updates after receiving nutrition data from the server. The introduction of asynchronicity is crucial because network requests, such as HTTP API calls,

may take an unpredictable amount of time. It is essential to avoid blocking the application while waiting for data.

- II. **User Input:** Users can input a food item's name in the search bar, and the app sends a request to the NinjaAPI to retrieve relevant nutritional data.
- III. **Advanced JSON Parsing:** The app uses advanced JSON parsing techniques (Img 2) to process the API response and extract specific nutritional details.
- IV. **Favourites:** Users can click on the heart button to save their favourite foods. The heart button turns red when clicked, indicating the food is a favourite.
- V. **Core Data:** The app utilises Core Data to store the details of the user's favourite foods in a local database, allowing them to access their favourite foods at any time.



The screenshot displays the CoreData local database interface. On the left, under 'ENTITIES', 'FavoriteFood' is selected. Below it, 'FETCH REQUESTS' and 'CONFIGURATIONS' are listed, with 'Default' selected under configurations. On the right, the 'Attributes' section is expanded, showing a table of attributes for the 'FavoriteFood' entity.

Attribute	Type	
calories	Double	↕
carbohydrates_to...	Double	↕
cholesterol_mg	Double	↕
fat_saturated_g	Double	↕
fat_total_g	Double	↕
fiber_g	Double	↕
potassium_mg	Double	↕
protein_g	Double	↕
serving_size_g	Double	↕
sodium_mg	Double	↕
sugar_g	Double	↕
name	String	↕

Img 3: CoreData local database

User Experience and interaction:

- I. The app features an intuitive and user-friendly interface with a search bar for entering food names. If the users enters a name which is not food, the app will not return any result.
- II. Users can interact with the displayed food details, and clicking on the heart button allows them to save their favourite foods.
- III. A dedicated "Favourites" tab allows users to view their saved favourite foods.
- IV. The app uses a clear and visually appealing design to present food details to users. Contrasting and appealing colours are used to enhance the user experience.

Error Handling and Reporting:

- I. The app handles errors gracefully and provides feedback to the user in case of network issues or problems with the API request.
- II. Error messages are displayed to inform the user about issues, and the app logs errors for debugging and support purposes.
- III. Do, catch blocks, fatal errors and test cases are used extensively for error handling and reporting.

```
if context.hasChanges {
    do {
        try context.save()
    } catch {
        let nerror = error as NSError
        fatalError("Unresolved error \(nerror), \(nerror.userInfo)")
    }
}
```

Img 4: Fatal error handling

```
do {
    try managedObjectContext.save()
} catch {
    print("Error saving favorite food: \(error)")
}
```

Img 5: error handling

```
final class NutritionViewerUITests: XCTestCase {

    override func setUpWithError() throws {
        // Put setup code here. This method is called before the invocation of each test method in the class.

        // In UI tests it is usually best to stop immediately when a failure occurs.
        continueAfterFailure = false

        // In UI tests it's important to set the initial state - such as interface orientation - required for your tests before they run. The setUp method is a good place to do this.
    }

    override func tearDownWithError() throws {
        // Put teardown code here. This method is called after the invocation of each test method in the class.
    }
}
```

Img 6: XCT error handling

Testing and Debugging:

- I. The app has undergone rigorous testing through XCTest for unit testing and UI testing using XCTest for user interface testing.
- II. Debugging has been performed to identify and resolve issues, ensuring a smooth user experience.

```
import XCTest

final class NutritionViewerUITestsLaunchTests: XCTestCase {

    override class var runsForEachTargetApplicationUIConfiguration: Bool {
        true
    }

    override func setUpWithError() throws {
        continueAfterFailure = false
    }

    func testLaunch() throws {
        let app = XCUIApplication()
        app.launch()

        // Insert steps here to perform after app launch but before taking a screenshot,
        // such as logging into a test account or navigating somewhere in the app

        let attachment = XCTAttachment(screenshot: app.screenshot())
        attachment.name = "Launch Screen"
        attachment.lifetime = .keepAlways
        add(attachment)
    }
}
```

Img 7: XCT UI Tests

Overview of Features:

- I. Search for food items by name.
- II. View detailed nutritional information for various foods.
- III. Save favourite foods by clicking on the heart button.
- IV. Access a list of saved favourite foods.
- V. Intuitive and user-friendly interface.
- VI. Real-time data retrieval from the NinjaAPI.
- VII. Data storage using Core Data for offline access.



Img 8: Nutrition app logo

The Nutrition Viewer app is a valuable tool for individuals interested in monitoring and exploring nutritional details of different foods, making informed dietary choices, and keeping track of their favourite foods.