ESCUELA SUPERIOR DE ARTE Y TECNOLOGÍA

Mobile Programming 2023-2024

ANDROID AND IOS COMPARATIVE DOCUMENTATION

Subject Name: Mobile Programming
Module: 3º VGP
Professor: Rubén Blanco García
Year: 2023/2024
Author: Ignacio Sempere Santonja

Index

1.- iOS development

1.1.- iOS introduction

iOS is the proprietary mobile operating system developed by Apple Inc. for its range of mobile devices, including the iPhone, iPad, and iPod Touch. Since its debut in 2007, iOS has evolved significantly, becoming one of the most influential and widely used operating systems in the world. It is renowned for its intuitive user interface, robust security features, and seamless integration with the broader Apple ecosystem, which includes macOS, watchOS, and tvOS.

The iOS ecosystem is designed to provide a cohesive and synchronized experience across all Apple devices. This integration is facilitated through services like iCloud, which allows for the seamless synchronization of photos, documents, and settings across devices. Additionally, features such as Handoff, Continuity, and AirDrop enable users to start tasks on one device and finish them on another, enhancing productivity and convenience.

Over the years, Apple has released numerous versions of iOS, each bringing new features, improvements, and enhancements. Below is a list of all the major iOS versions released to date.

- **iPhone OS 1 (2007):** It included basic features like the Phone, Mail, Safari, and iPod apps.
- **iPhone OS 2 (2008):** It brought the App Store, allowing third-party app development and distribution.
- **iPhone OS 3 (2009):** It introduced features like cut, copy, paste, and MMS.
- **iOS 4 (2010):** It included multitasking, FaceTime, and the introduction of the iBooks app.
- **iOS 5 (2011):** It brought iCloud, iMessage, Notification Center, and Siri, Apple's voice assistant.
- **iOS 6 (2012):** It introduced Apple Maps, Passbook, and Facebook integration.
- **iOS 7 (2013):** It featured a complete redesign of the user interface, AirDrop, and Control Center.
- **iOS 8 (2014):** It brought HealthKit, HomeKit, and Apple Pay.
- **iOS 9 (2015):** It introduced Proactive Assistant, transit directions in Maps, and improved multitasking on iPad.
- **iOS 10 (2016):** It included a redesigned Lock screen, new features for Messages, and Siri SDK for developers.
- **iOS 11 (2017):** It brought a new Files app, ARKit for augmented reality, and major updates for iPad.
- **iOS 12 (2018):** It focused on performance improvements, Screen Time, and Group FaceTime.
- **iOS 13 (2019):** It introduced Dark Mode, enhanced privacy features, and a revamped Photos app.
- **iOS 14 (2020):** It brought home screen widgets, App Library, and new privacy features.

- **iOS 15 (2021):** It included improvements to FaceTime, Focus modes, and redesigned notifications.
- **iOS 16 (2022):** It introduced a customizable Lock screen, Live Activities, and new Messages features.
- **iOS 17 (2023):** It brought improvements to user experience, new features for Messages and Mail, and enhanced machine learning capabilities.

Each iteration of iOS has aimed to enhance the user experience, add new functionalities, and ensure that devices remain secure and efficient. The continual evolution of iOS underscores Apple's commitment to innovation and excellence, making it a cornerstone of the modern digital experience.

1.2.- Storyboard

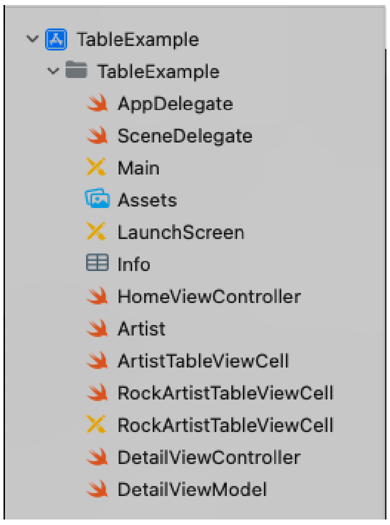
The same app that was developed with Android Studio, that consisted of a login form, a scrollable list with several items, a detail screen for each of the items and a contact form, has now been replicated with the XCode IDE. The app has been replicated both with the Storyboard and the Swift UI methods.

XCode Storyboard is a visual tool used to design and build the UI of Apple apps. It allows drag-and-dropping different elements on a view to better construct its design. It uses the MVC pattern to communicate between the different classes and implements navigation controllers to ease the flow of the application.

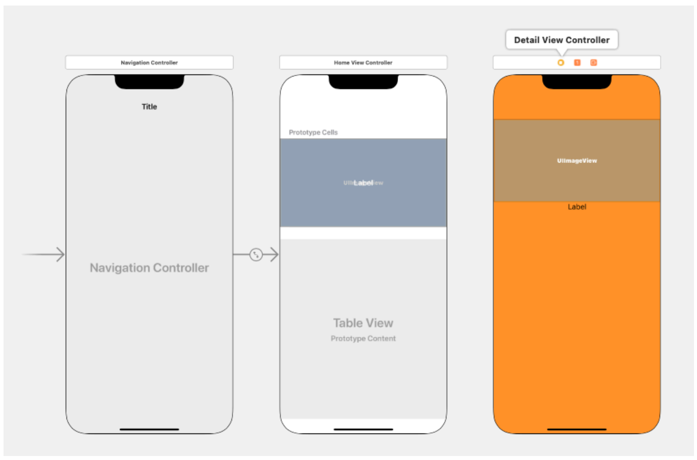
This version of the app only counts with a table view with all the items and a detail view to see the data of each item.

1.2.1.- Project structure

This is the structure of the Storyboard project.



- TableExample
  - TableExample
    - AppDelegate
    - SceneDelegate
    - Main
    - Assets
    - LaunchScreen
    - Info
    - HomeViewController
    - Artist
    - ArtistTableViewCell
    - RockArtistTableViewCell
    - RockArtistTableViewCell
    - DetailViewController
    - DetailViewModel



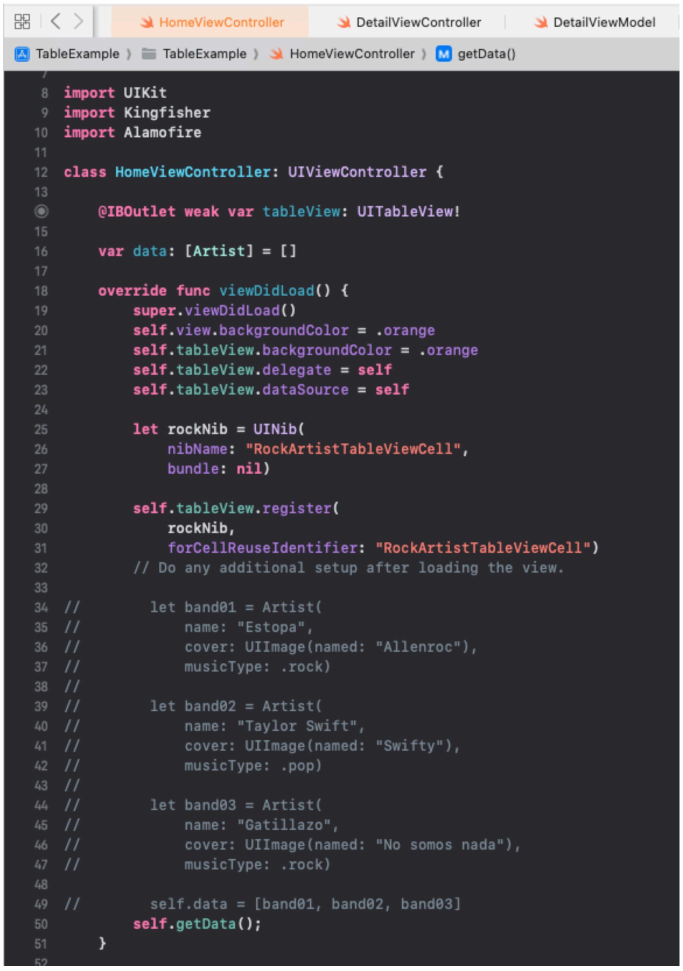
As it can be seen on the left image, the project includes very few classes being this the Artist class, its cell for the table view, the detail view controller, the detail view model, a specific cell for rock artists and the home view controller.

On the right image it can be seen the navigation flow of the app, starting this on the launch screen and accessing directly to the home view.

1.2.2.- Project implementation

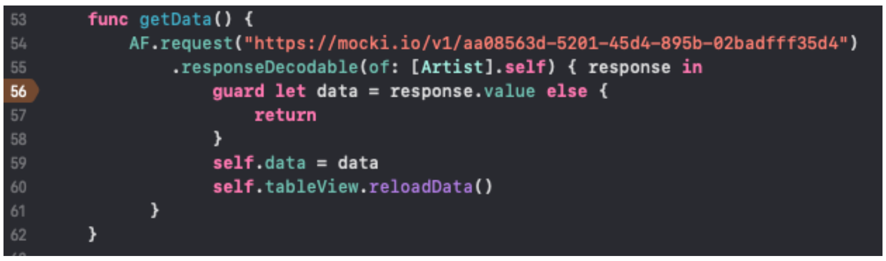
Following there are some screenshots showing the code of each class of the project.

The home view controller



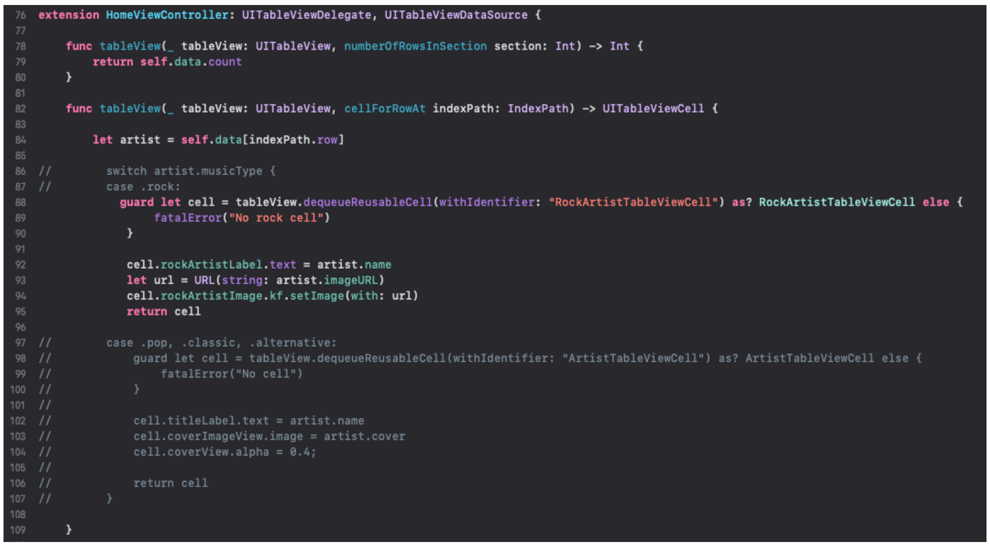
```swift
import UIKit
import Kingfisher
import Alamofire

class HomeViewController: UIViewController {

    @IBOutlet weak var tableView: UITableView!

    var data: [Artist] = []

    override func viewDidLoad() {
        super.viewDidLoad()
        self.view.backgroundColor = .orange
        self.tableView.backgroundColor = .orange
        self.tableView.delegate = self
        self.tableView.dataSource = self

        let rockNib = UINib(
            nibName: "RockArtistTableViewCell",
            bundle: nil)

        self.tableView.register(
            rockNib,
            forCellReuseIdentifier: "RockArtistTableViewCell")
        // Do any additional setup after loading the view.

//        let band01 = Artist(
//            name: "Estopa",
//            cover: UIImage(named: "Allenroc"),
//            musicType: .rock)
//
//        let band02 = Artist(
//            name: "Taylor Swift",
//            cover: UIImage(named: "Swifty"),
//            musicType: .pop)
//
//        let band03 = Artist(
//            name: "Gatillazo",
//            cover: UIImage(named: "No somos nada"),
//            musicType: .rock)

//        self.data = [band01, band02, band03]
        self.getData();
    }
```

Here we declare a table view. As it can be seen on the commented code, we first created the table items by hardcoding them.



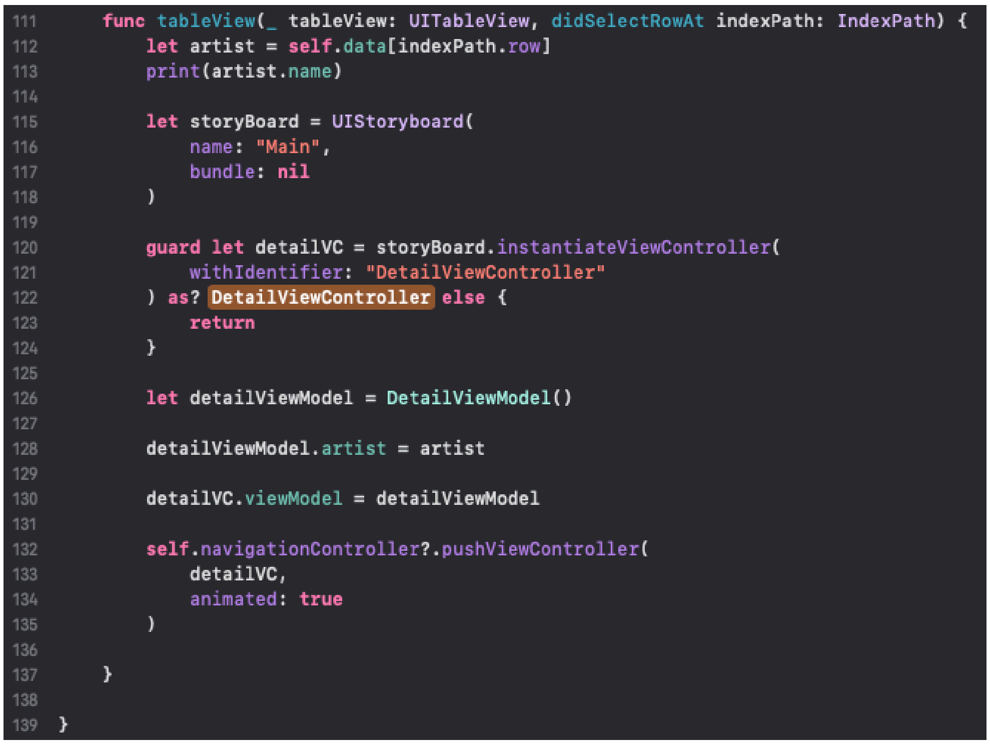
```swift
    func getData() {
        AF.request("https://mocki.io/v1/aa08563d-5201-45d4-895b-02badfff35d4")
            .responseDecodable(of: [Artist].self) { response in
                guard let data = response.value else {
                    return
                }
                self.data = data
                self.tableView.reloadData()
            }
    }
```

Here, by using the Alamofire library, we read a json file from an external api and load its content to the table view. Unfortunately, this api was temporary and its content is no longer accessible, so I don't have any screenshot of the result.



```swift
extension HomeViewController: UITableViewDelegate, UITableViewDataSource {

    func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        return self.data.count
    }

    func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {

        let artist = self.data[indexPath.row]

//        switch artist.musicType {
//        case .rock:
          guard let cell = tableView.dequeueReusableCell(withIdentifier: "RockArtistTableViewCell") as? RockArtistTableViewCell else {
              fatalError("No rock cell")
           }

           cell.rockArtistLabel.text = artist.name
           let url = URL(string: artist.imageURL)
           cell.rockArtistImage.kf.setImage(with: url)
           return cell

//        case .pop, .classic, .alternative:
//            guard let cell = tableView.dequeueReusableCell(withIdentifier: "ArtistTableViewCell") as? ArtistTableViewCell else {
//                fatalError("No cell")
//            }
//
//            cell.titleLabel.text = artist.name
//            cell.coverImageView.image = artist.cover
//            cell.coverView.alpha = 0.4;
//
//            return cell
//        }

    }
```

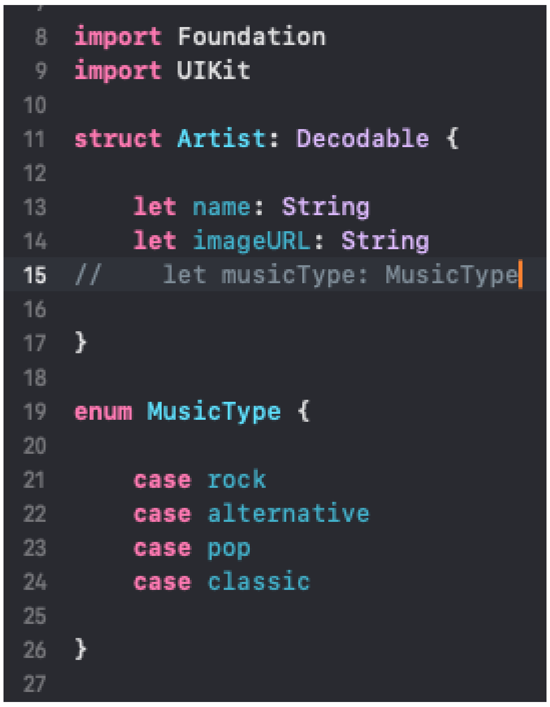
Here we fill each cell of the table with the received data.



```swift
    func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {
        let artist = self.data[indexPath.row]
        print(artist.name)

        let storyBoard = UIStoryboard(
            name: "Main",
            bundle: nil
        )

        guard let detailVC = storyBoard.instantiateViewController(
            withIdentifier: "DetailViewController"
        ) as? DetailViewController else {
            return
        }

        let detailViewModel = DetailViewModel()

        detailViewModel.artist = artist

        detailVC.viewModel = detailViewModel

        self.navigationController?.pushViewController(
            detailVC,
            animated: true
        )

    }

}
```

And here we link an event to each table item so that when it is clicked it opens the detail view with the corresponding data for each cell.

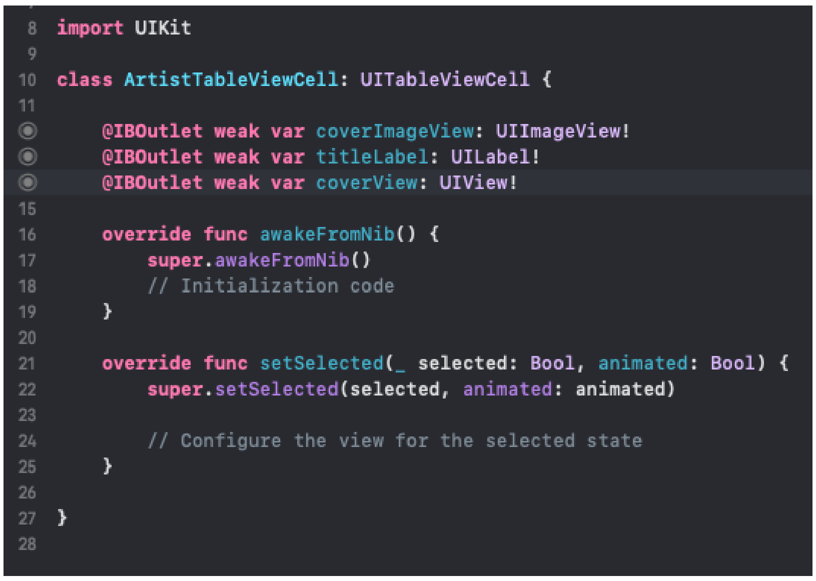
The artist class



```swift
import Foundation
import UIKit

struct Artist: Decodable {

    let name: String
    let imageURL: String
//    let musicType: MusicType

}

enum MusicType {

    case rock
    case alternative
    case pop
    case classic

}

```

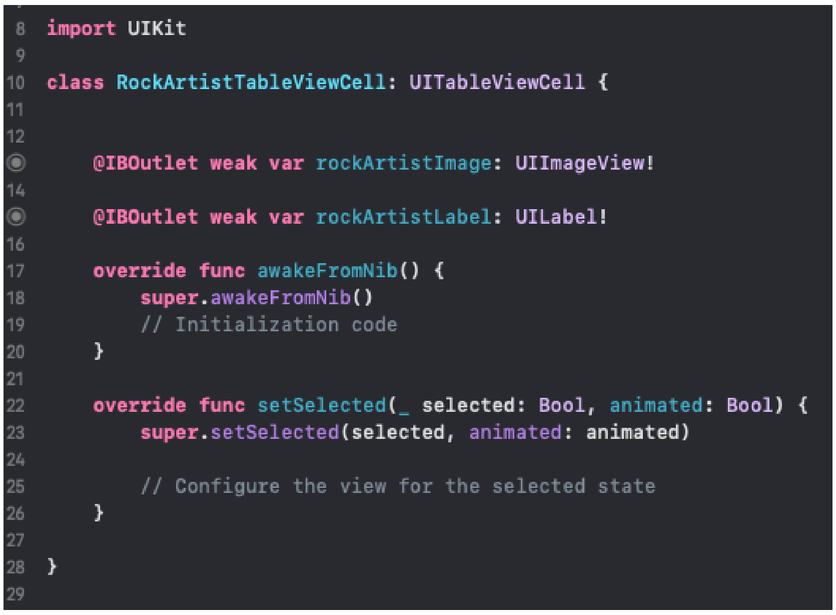
It has to be decodable so that it can be parsed from the json file.

The artist table view cell class with its components



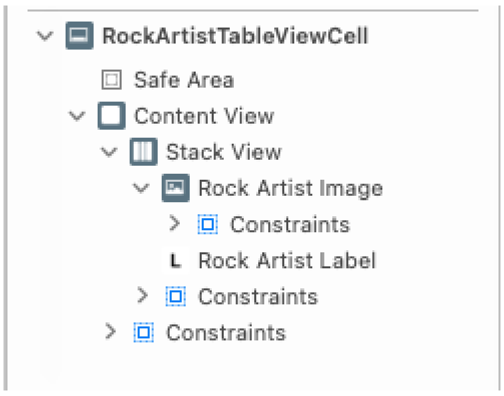
```swift
import UIKit

class ArtistTableViewCell: UITableViewCell {

    @IBOutlet weak var coverImageView: UIImageView!
    @IBOutlet weak var titleLabel: UILabel!
    @IBOutlet weak var coverView: UIView!

    override func awakeFromNib() {
        super.awakeFromNib()
        // Initialization code
    }

    override func setSelected(_ selected: Bool, animated: Bool) {
        super.setSelected(selected, animated: animated)

        // Configure the view for the selected state
    }

}

```

The rock artist table view cell with its components

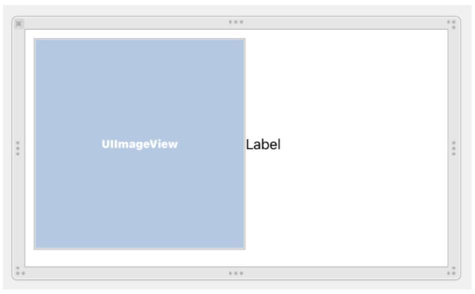


```swift
import UIKit

class RockArtistTableViewCell: UITableViewCell {


    @IBOutlet weak var rockArtistImage: UIImageView!

    @IBOutlet weak var rockArtistLabel: UILabel!

    override func awakeFromNib() {
        super.awakeFromNib()
        // Initialization code
    }

    override func setSelected(_ selected: Bool, animated: Bool) {
        super.setSelected(selected, animated: animated)

        // Configure the view for the selected state
    }

}
```

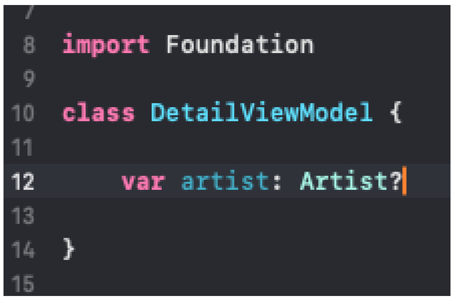
The design for the rock artist table view cell



```
RockArtistTableViewCell
    Safe Area
    Content View
        Stack View
            Rock Artist Image
                Constraints
            Rock Artist Label
            Constraints
        Constraints
```

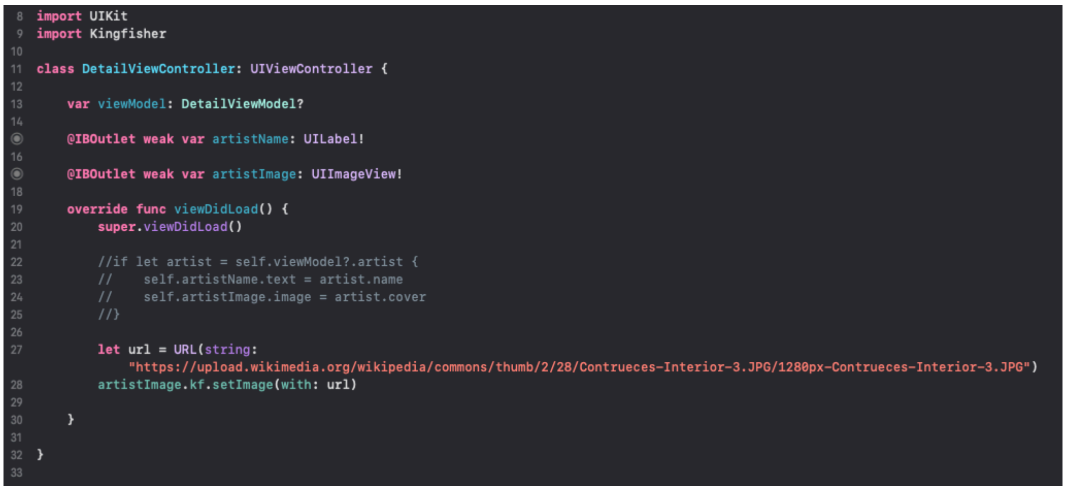


The detail view model which only includes one optional artist



```swift
import Foundation

class DetailViewModel {

    var artist: Artist?

}
```

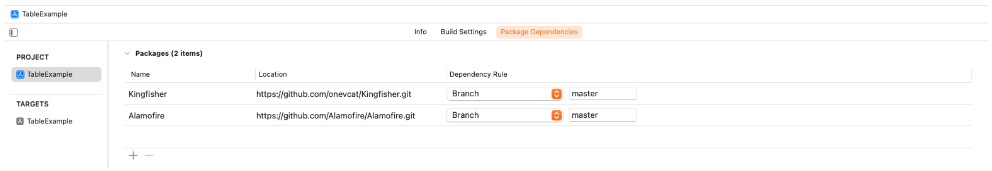
The detail view controller



```swift
import UIKit
import Kingfisher

class DetailViewController: UIViewController {

    var viewModel: DetailViewModel?

    @IBOutlet weak var artistName: UILabel!

    @IBOutlet weak var artistImage: UIImageView!

    override func viewDidLoad() {
        super.viewDidLoad()

        //if let artist = self.viewModel?.artist {
        //    self.artistName.text = artist.name
        //    self.artistImage.image = artist.cover
        //}

        let url = URL(string:
            "https://upload.wikimedia.org/wikipedia/commons/thumb/2/28/Contrueces-Interior-3.JPG/1280px-Contrueces-Interior-3.JPG")
        artistImage.kf.setImage(with: url)

    }

}
```

Here with the help of the Kingfisher library we can parse an URL to an image and store it as the cover of the artist.

And finally, the packages used on the project.



Packages (2 items)

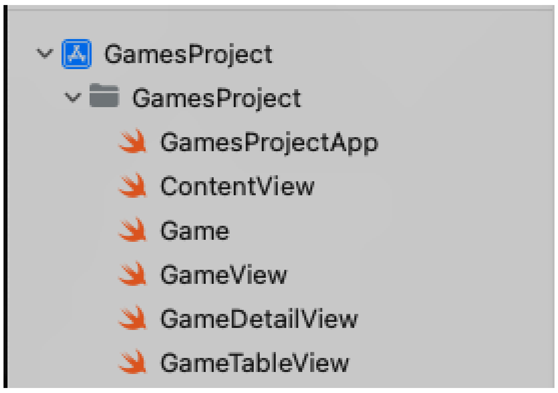
Name	Location	Dependency Rule	
Kingfisher	https://github.com/onevcat/Kingfisher.git	Branch	master
Alamofire	https://github.com/Alamofire/Alamofire.git	Branch	master

1.3.- Swift UI

As mentioned before, the app has been replicated using both Storyboards and Swift UI. SwiftUI is a modern framework for building user interfaces across all Apple platforms, including iOS, macOS, watchOS, and tvOS. It provides a declarative syntax that simplifies the process of designing and developing UIs.

1.3.1.- Project structure

This is the structure of the Swift UI project.



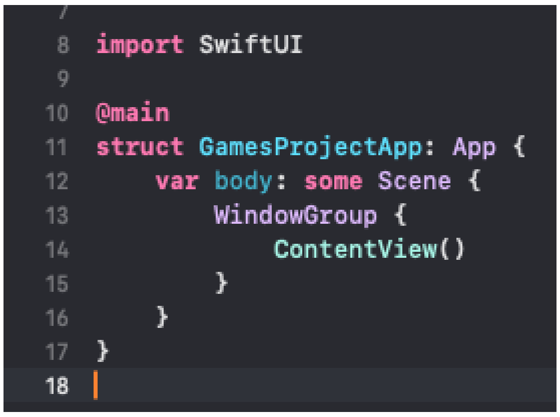
```
GamesProject
  GamesProject
    GamesProjectApp
    ContentView
    Game
    GameView
    GameDetailView
    GameTableView
```

As it can be seen, it includes the same classes as the Storyboard project, having games instead of artists. Another difference is that here there aren't any controllers and the structure stays simpler and cleaner.

1.3.2.- Project implementation

Following there are some screenshots showing the code of each class of the project.

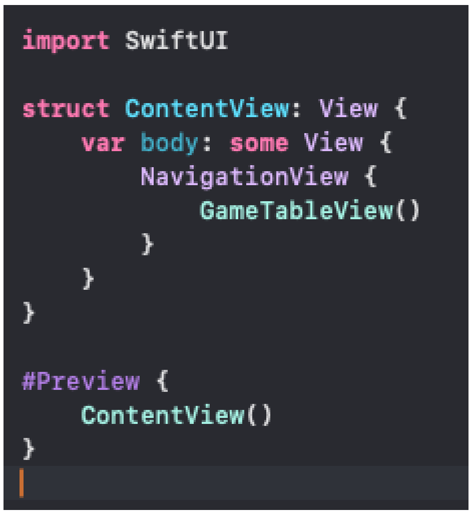
The GamesProjectApp, the main class of the project.



```swift
import SwiftUI

@main
struct GamesProjectApp: App {
    var body: some Scene {
        WindowGroup {
            ContentView()
        }
    }
}
```

It loads the content of the content view function.

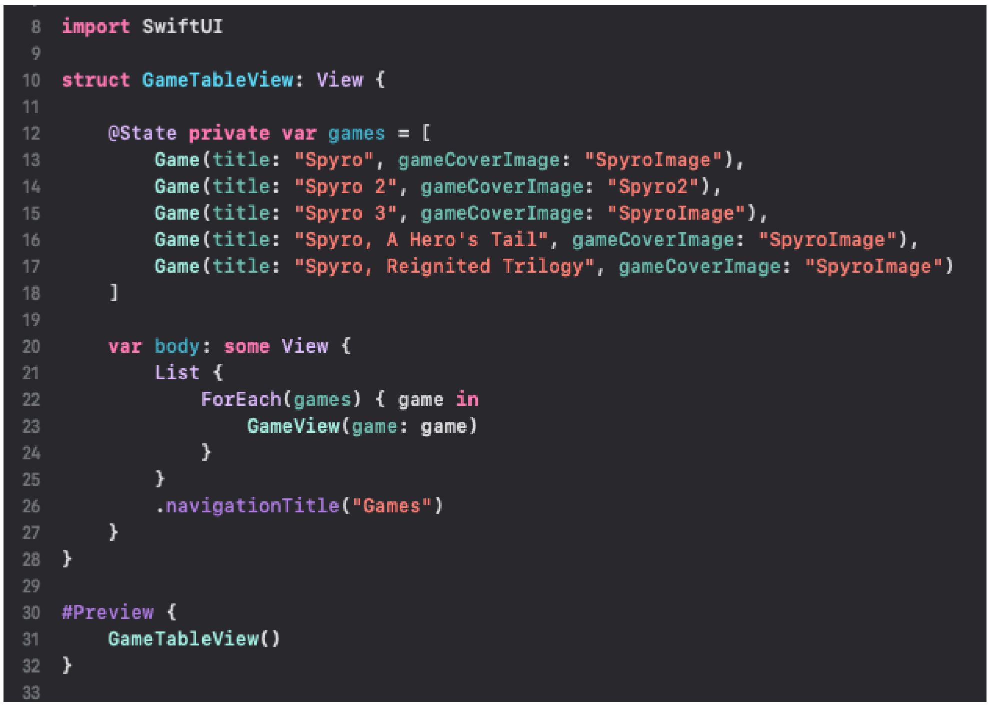
The content view class.



```swift
import SwiftUI

struct ContentView: View {
    var body: some View {
        NavigationView {
            GameTableView()
        }
    }
}

#Preview {
    ContentView()
}
```

It loads the content of the GameTableView class. Also we can preview the content of this class with the Preview macro.

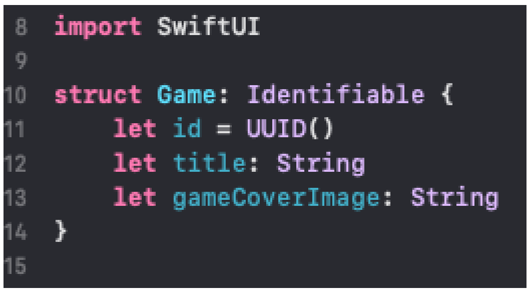
The game table view class.



```swift
import SwiftUI

struct GameTableView: View {

    @State private var games = [
        Game(title: "Spyro", gameCoverImage: "SpyroImage"),
        Game(title: "Spyro 2", gameCoverImage: "Spyro2"),
        Game(title: "Spyro 3", gameCoverImage: "SpyroImage"),
        Game(title: "Spyro, A Hero's Tail", gameCoverImage: "SpyroImage"),
        Game(title: "Spyro, Reignited Trilogy", gameCoverImage: "SpyroImage")
    ]

    var body: some View {
        List {
            ForEach(games) { game in
                GameView(game: game)
            }
        }
        .navigationTitle("Games")
    }
}

#Preview {
    GameTableView()
}
```

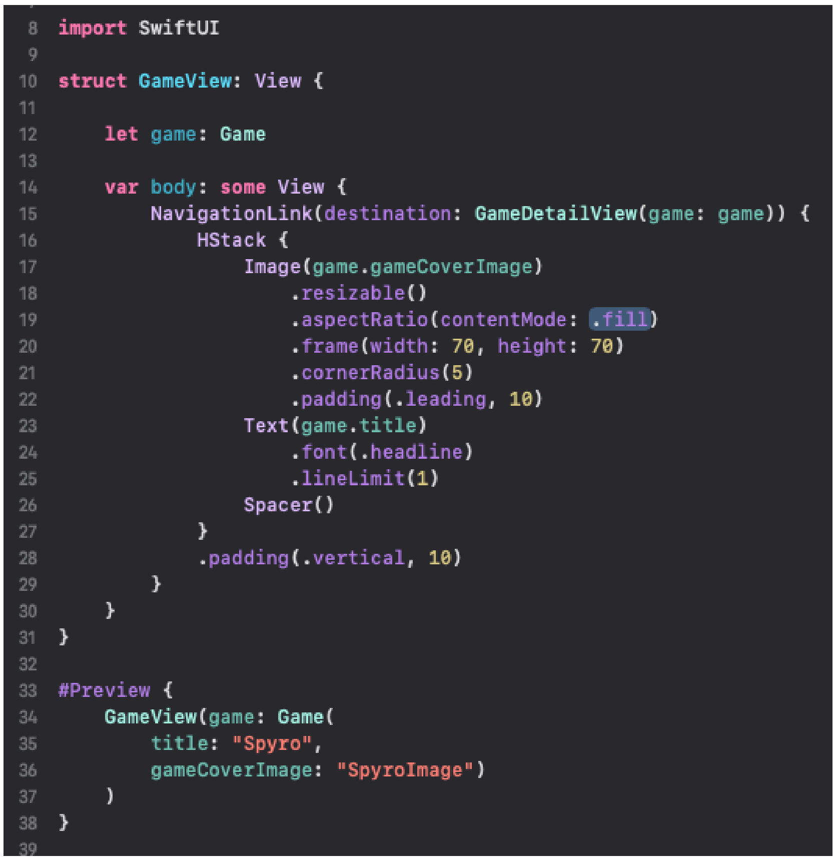
Here we are creating several Game items within an array that will be loaded on a list. Unlike the Storyboard project, we are hardcoding all the content of the table.

The Game class with all its components.



```swift
import SwiftUI

struct Game: Identifiable {
    let id = UUID()
    let title: String
    let gameCoverImage: String
}

```

The GameView class

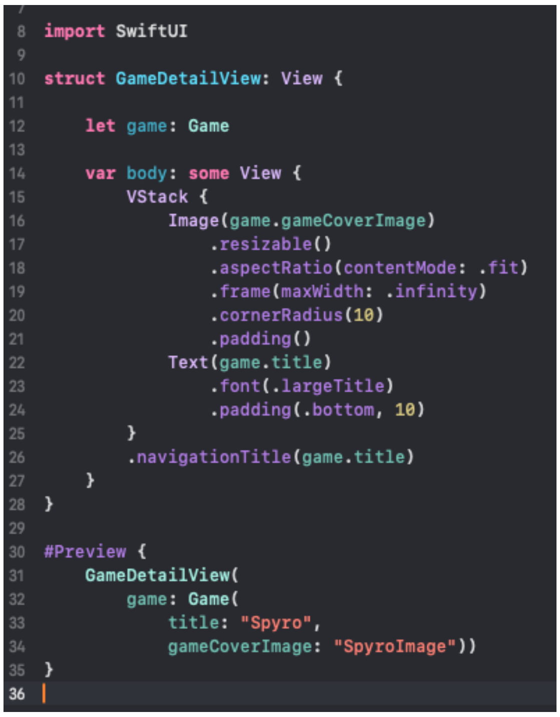


```swift
import SwiftUI

struct GameView: View {

    let game: Game

    var body: some View {
        NavigationLink(destination: GameDetailView(game: game)) {
            HStack {
                Image(game.gameCoverImage)
                    .resizable()
                    .aspectRatio(contentMode: .fill)
                    .frame(width: 70, height: 70)
                    .cornerRadius(5)
                    .padding(.leading, 10)
                Text(game.title)
                    .font(.headline)
                    .lineLimit(1)
                Spacer()
            }
            .padding(.vertical, 10)
        }
    }
}

#Preview {
    GameView(game: Game(
        title: "Spyro",
        gameCoverImage: "SpyroImage")
    )
}

```

Here we define how each item will be displayed on the table view.

As it can be seen, Swift UI allows the developer to rapidly design a view by declaring and appending the properties of each item. This makes the building of UI very rapid and easy.

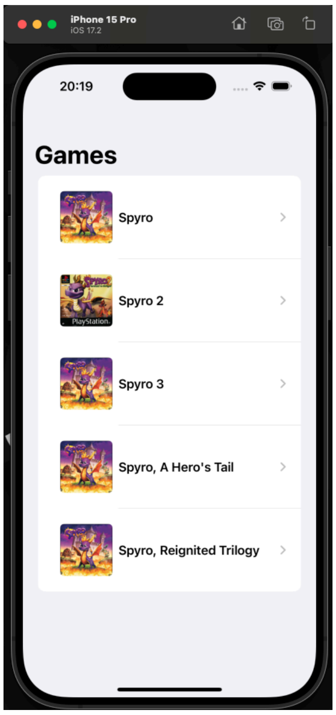
Finally, the GameDetailView class

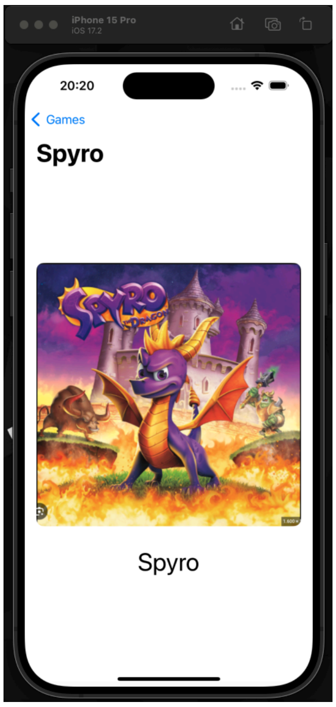


```swift
import SwiftUI

struct GameDetailView: View {

    let game: Game

    var body: some View {
        VStack {
            Image(game.gameCoverImage)
                .resizable()
                .aspectRatio(contentMode: .fit)
                .frame(maxWidth: .infinity)
                .cornerRadius(10)
                .padding()
            Text(game.title)
                .font(.largeTitle)
                .padding(.bottom, 10)
        }
        .navigationTitle(game.title)
    }
}

#Preview {
    GameDetailView(
        game: Game(
            title: "Spyro",
            gameCoverImage: "SpyroImage"))
}
```

Here we define how the data of each game will be displayed on the detail screen.

This is the final result of the app.





2.- Comparative

Android Studio and **XCode** are the primary IDEs for their respective platforms. Both tools let developers create, test, and implement mobile applications by providing extensive capabilities that are specific to their respective ecosystems. They differ greatly in terms of design, functionality, and supported programming languages, even if they have the same basic goal.

Android Studio, developed by Google, is the official IDE for Android development. It is built on JetBrains' IntelliJ IDEA and provides a powerful and flexible environment for building Android apps. It supports Java and Kotlin as primary programming languages. Kotlin, endorsed by Google as the preferred language for Android development, offers modern syntax, safety features, and interoperability with Java, making it a robust choice for developers. Android Studio includes an array of tools such as a robust code editor with intelligent code completion, a flexible Gradle-based build system, and a suite of debugging and performance analysis tools. The integrated Android Emulator allows developers to simulate a wide range of devices and configurations, facilitating comprehensive testing. Additionally, Android Studio's Layout Editor provides a visual design interface for creating and refining user interfaces, leveraging both XML and a drag-and-drop interface.

XCode, developed by Apple, is the official IDE for iOS, macOS, watchOS, and tvOS development. It supports programming languages such as Swift and Objective-C. Swift, introduced by Apple in 2014, is the preferred language for iOS development due to its modern syntax, safety features, and performance advantages over Objective-C. XCode offers a rich set of features including a sophisticated code editor with syntax highlighting, autocompletion, and refactoring tools. It also provides powerful debugging tools, performance analyzers, and comprehensive testing capabilities, including unit tests and UI tests. XCode's Interface Builder is a standout feature, enabling developers to design and test user interfaces using a visual editor. With the introduction of SwiftUI, developers can now build UIs using a declarative syntax, which is integrated seamlessly into XCode, allowing for real-time previews and interactive design experiences.

The relationship between these IDEs and their respective programming languages is intrinsic and deeply integrated. Android Studio’s support for Kotlin and Java ensures that developers can leverage the extensive libraries and tools available in the Java ecosystem, while also benefiting from Kotlin's modern features. XCode’s support for Swift and Objective-C allows developers to build on a legacy of robust, high-performance code while taking advantage of Swift's contemporary language features and safety mechanisms.

In summary, Android Studio and XCode each provide a tailored environment optimized for their specific platforms. Android Studio offers flexibility and power through Kotlin and Java, robust build and testing tools, and a visual layout editor for designing user interfaces. XCode provides a seamless and integrated development experience for Apple platforms, with strong support for Swift, powerful debugging and testing tools, and advanced UI design capabilities through Interface Builder and SwiftUI. Both IDEs embody the design philosophies and ecosystem strengths of their respective platforms, enabling developers to create high-quality mobile applications.

2.1.- Development process

After having developed two similar applications in both Android Studio and XCode, I have been able to see quite a few similarities between both IDEs. For example, during the development of both applications, in the case of Android Studio, we have made use of the Kotlin and Jetpack Compose classes. On the other hand, in the case of XCode we have used Storyboards and Swift UI. Additionally, both IDEs have a typical window layout, with a menu at the top, a file explorer on the side, code in the center, and debug information at the bottom, which makes development easier if you're already familiarized with these types of programs.

On the one hand, both Kotlin and Storyboards have been much more complex since the syntax, the creation of variables and functions and the flow of function calls of the internal API were more difficult. This type of framework requires minimal programming knowledge to understand how to create and manage the different classes and resources necessary to make the application work. In terms of code maintainability, I consider that these frameworks make it difficult to improve and/or extend if a long time has passed since the last time it was worked on. To do this, it is necessary to structure the classes and code very well, as well as add comments to know what each part of the code does.

On the other hand, developing the application with Jetpack Compose and Swift UI has been much easier. After having developed the application with Kotlin and Storyboards, the development environments of Android and Apple have not been so unknown. Both frameworks offer a simple, fast and direct way to layout and configure both the interface and the functionality of the different views of the application. Additionally, both frameworks offer a preview of what is being programmed, which speeds up the development process since it is not necessary to launch the application to check how the design looks. This type of more visual and agile framework also allows better maintainability of the code, since, being simpler, it is easier to structure it and remember its operation after having not modified it for a while.

Both for the development of the application on Android and iOS, external libraries have been used. These libraries have been used to read and parse json files from an external database and to read and load images from a url. In the case of Android, Firebase has been used as an external database, RetrofitClient to read and parse json files and Glide to read and load images through a URL. On the other hand, in the case of iOS, Mocki.io has been used as an external api to store a json file, Alamofire to read and parse said json file and Kingfisher to read and upload images through a URL.

3.- Post mortem

Although we have had few classes and in some of them we have had technical problems that have made their development difficult, I consider that the final result is positive and that everything seen, mentioned and explained in class is relevant. We have been able to develop an application for both Android and iOS devices, although the latter one has been developed entirely in class. In addition, we have seen fundamental aspects that must be taken into account when developing and marketing an application, such as the minimum version of Android or iOS that the application will support, what permissions must be requested from the end user or designing an interface. user as simple and intuitive as possible.

In my case, as I had already taken a course in programming mobile applications with Android, the development of this part of the practice was not so difficult for me, I only needed a couple of classes to remember what the Android workflow was. However, I had never programmed with Jetpack Compose, which surprised me and I really liked its simplicity, both in the syntax and in the preview of what is being programmed. I consider it to be a language that allows for very easy and fast application prototyping.

On the other hand, I had never programmed anything for Apple devices, nor had I used the XCode IDE, nor did I know about Storyboards, nor had I worked with Swift UI. I think Storyboards are quite similar to developing with Kotlin in the sense that you need to have a basic knowledge of programming to understand what you are doing. You need to understand how the internal flow of function calls works to know how to load the different views of your application. Instead, Swift UI has been very similar to Jetpack Compose, I consider that they are two simpler, more agile and visual languages, which allow faster and easier prototyping of an application.

I was also surprised and really liked the fact that we have scheduled small demos for the Apple Vision Pro, since it is a totally new product on the market and there is a high demand from developers for this device. Clearly, with the content that we have seen in class we do not have a sufficient level to be able to work developing this type of applications, however, it does help us to mention it in the CV, which gives us a useful advantage. Furthermore, once we have seen the basic mechanisms for programming content in the Apple Vision Pro, I believe that we have the necessary tools to be able to investigate and learn on our own.

To conclude, it is true that there are many things that we have not been able to delve into and that have been explained quickly or have not been seen at all, but it is also true that many of us will probably not develop anything for mobile devices again throughout our lives. career. However, this subject has given us the necessary tools to program simple applications and do our own research in case we want to develop and/or market an application.