Andrés Fonseca
May 29, 2024

El Pregonero App Documentation

Introduction

In today's fast-paced world, staying informed and connected is more crucial than ever. Recognizing this need, ***El Pregonero*** aims to develop a comprehensive platform that seamlessly integrates various aspects of entertainment and connectivity. This is achieved through offering a diverse range of content, including movies, news, sports highlights, and a user directory. Providing users with up-to-date news and entertainment on the go, the app also fosters community connection through location-based services.

Feature Description

Here's why out app is essential:

1. Diverse Content: ***El Pregonero*** provides users with a diverse range of content, including movies, news, sports highlights, and a user directory. This diversity caters to different interests and ensures that users can find something relevant and always engaging.

2. Stay Informed: With access to news articles and sports match highlights, users can stay up to date with the latest events and developments both locally and globally. Whether it's breaking news or updates on their favorite sports teams, users can rely on ***El Pregonero*** to keep them informed.

3. Entertainment on the Go: Whether waiting for a bus or taking a coffee break, users can enjoy movies and sports highlights right from their mobile devices. This on-the-go entertainment ensures that users can make the most of their free time, no matter where they are.

4. Community Connection: The user directory feature enhances community connection by allowing users to discover and connect with others nearby. Whether it's finding like-minded individuals or networking for professional opportunities, ***El Pregonero*** facilitates meaningful connections within the community.

5. Location-Based Services: By providing users with location-based services, such as finding nearby users and their locations, El Pregonero enhances convenience and utility. Users can easily locate nearby amenities, events, or even potential friends, making their daily lives more efficient and enjoyable.

User Flow

First Impression: News List

- Upon opening the app, users are greeted with the news list, providing them with the latest updates and headlines.
- At first glance, they see a program list where they can find their favorite movies and series.

Program Detail View

- If users tap on a program from the list, they are navigated to a detailed view where they can find more information about the selected program.

News List

- Users can also access the news list, which displays summaries of various news articles.
- If they tap on a news article, they are taken to a detailed view where they can read the full article.
- Additionally, users have the option to search through the news to find specific articles of interest.

News Sections

- Users can navigate to a section that lists all available news categories, allowing them to explore news topics based on their interests.

Sports Highlights

- Users can access the sports highlights section, where they find information about the latest football teams' matches.

Tapping on a match provides users with detailed information about the match, including scores, team details, and more.
Tab Bar Navigation

- In the tab bar at the bottom, users can find the "User List" option.
- Tapping on this option takes them to a user list screen, where they can see all the users registered on the app.
- Additionally, they find a button that, when tapped, navigates them to a map showing all users' locations.

User Location Map

- If users touch a specific user from the list, they are navigated to a map showing only that user's location, providing them with a more focused view.

Technical Overview

Technologies Used

- Swift (iOS native programming language)
- Alamofire (networking library)
- Kingfisher (image fetching library)
- Lotties (animation rendering library)

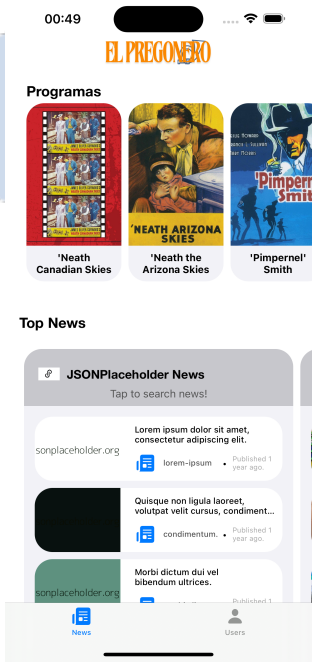
Implementation Details

- The navigation pattern used in this app is ***Coordination Navigation Pattern.***
- The architectural pattern used in this app is ***MVVM***
- This app uses Alamofire to implement a Networking Layer for all REST consumptions.
- This app uses mostly ***UICollectionViewCompositionalLayout*** for rendering data in the screens.

Setup and Usage

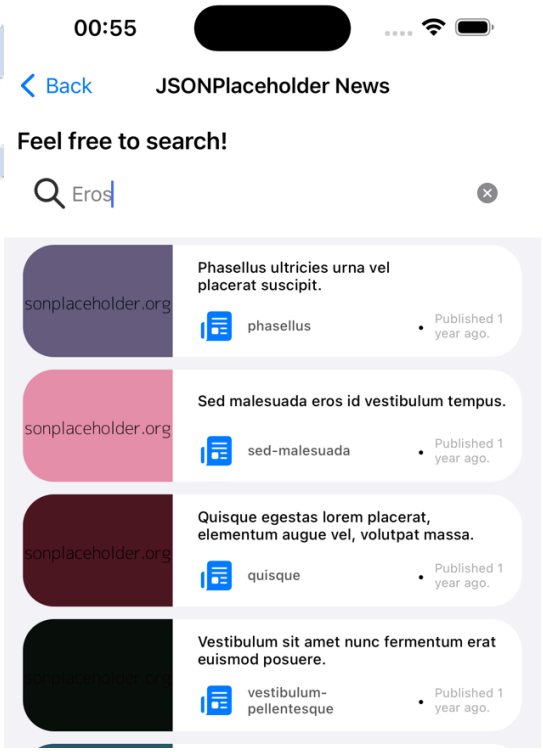
For End Users

1. You first encounter with news list screen:

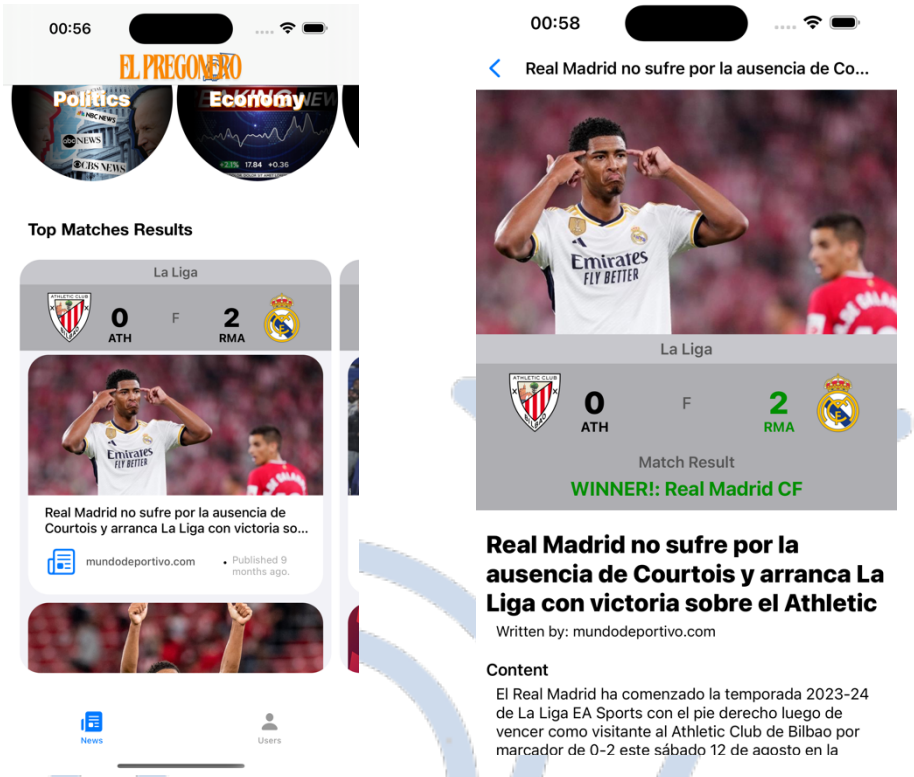


2. You can find a program detail screen if you tap a program:

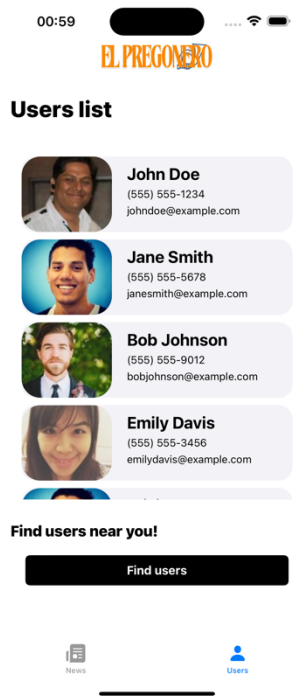
3. If you tap on first container you can search your news:



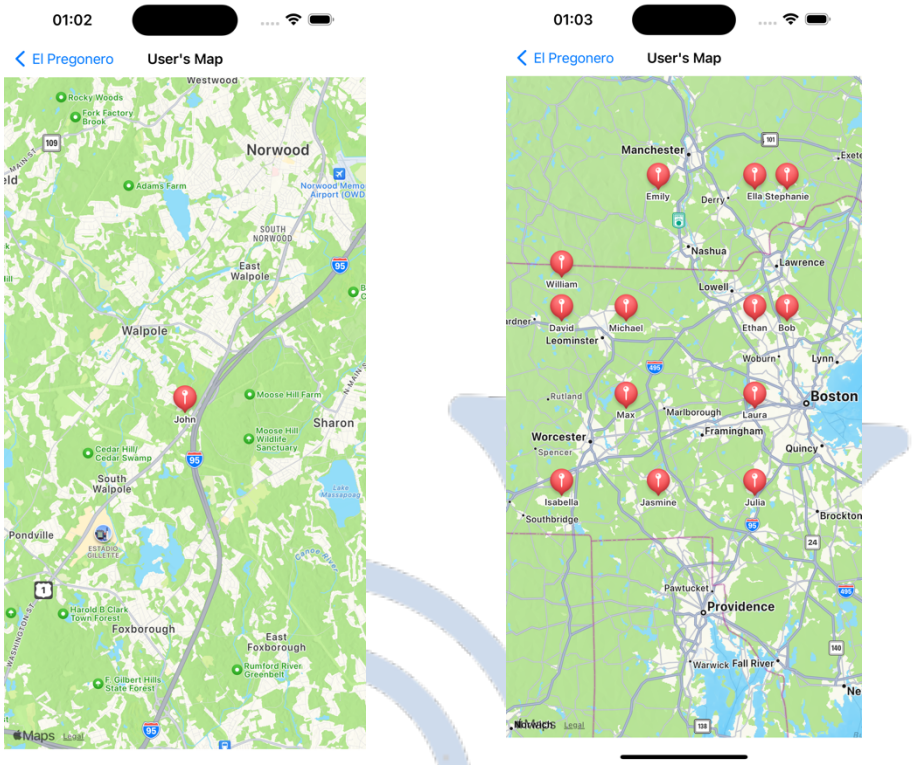
4. You can also find your favorite matches results which also include a detail screen:



5. On the tab bar, you can also find. *Users* option what will present you a user's list:



6. You can choose whether to show a single user's location or all of them:



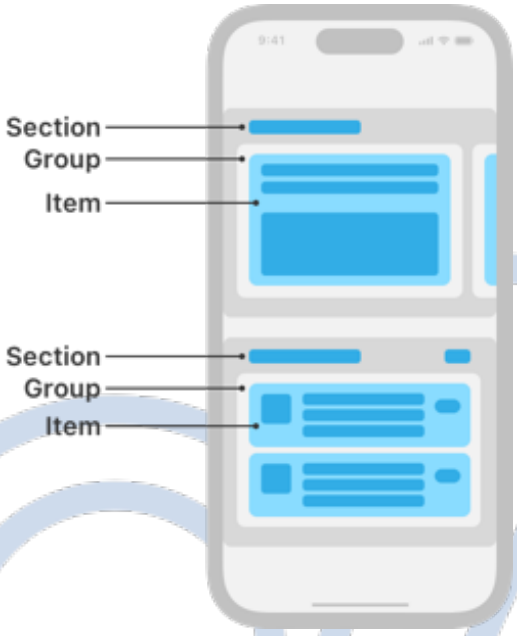
For Developers

Install required pods:

```
pod 'Alamofire'
pod 'Kingfisher'
pod 'lottie-ios'
```

UICollectionVIewCompositionalLayout

This feature mostly uses *CompositionalLayout* to set all the objects contained on the views. Let's look at apple's definition on why using it is helpful:



A compositional layout is a type of collection view layout. It's designed to be composable, flexible, and fast, letting you build any kind of visual arrangement for your content by combining — or compositing — each smaller component into a full layout.

A compositional layout is composed of one or more sections that break up the layout into distinct visual groupings. Each section is composed of groups of individual items, the smallest unit of data you want to present. A group might lay out its items in a horizontal row, a vertical column, or a custom arrangement.

Retrieved from Apple's Developer Documentation.

To set up and use it in your development environment, follow these steps:

1. Add a 'collectionViewContainer' view in the storyboard for the CollectionView that we will add programmatically and connect an *IBOutlet* to it:

```
1  @IBOutlet weak var collectionContainerView: UIView!
```

2. Create a UICollectionView object. We'll set it up soon:

```swift
private lazy var collectionView: UICollectionView? = {
}()
```

3. Use this general layout setup function, which will help you defining each object requirement on your layout:

```swift
func createBasicListLayout() -> UICollectionViewLayout {
    let itemSize = NSCollectionLayoutSize(widthDimension: .fractionalWidth(1.0),
                                          heightDimension: .fractionalHeight(1.0))
    let item = NSCollectionLayoutItem(layoutSize: itemSize)

    let groupSize = NSCollectionLayoutSize(widthDimension: .fractionalWidth(1.0),
                                           heightDimension: .absolute(44))
    let group = NSCollectionLayoutGroup.horizontal(layoutSize: groupSize,
                                                   subitems: [item])

    let section = NSCollectionLayoutSection(group: group)


    let layout = UICollectionViewCompositionalLayout(section: section)
    return layout
}
```

This will define the size on each object inside your layout.

Retrieved from Apple's Developer Documentation.

4. Adjust your collection view like so:

```swift
private lazy var collectionView: UICollectionView? = {
    let layout = createBasicListLayout()
    let collectionView = UICollectionView(frame: .zero, collectionViewLayout: layout)
    return collectionView
}()
```

5. Setup your collection view delegates in order to instantiate and configurate cells. Your collection view will follow the layout set up on your createLayout function. Your cells will follow the size set up on the *item* part.

Additional:

If you want to add a header and/or a footer to your section, do as following:

1. Set up your header or footer layout inside your createLayout function:

```swift
let header = NSCollectionLayoutBoundarySupplementaryItem(
    layoutSize: NSCollectionLayoutSize(widthDimension: .fractionalWidth(1.0),
    heightDimension: .absolute(50)),
    elementKind: UICollectionView.elementKindSectionHeader,
    alignment: .topLeading)
    section.boundarySupplementaryItems = [header]
```

```swift
let footer = NSCollectionLayoutBoundarySupplementaryItem(
    layoutSize: NSCollectionLayoutSize(widthDimension: .fractionalWidth(1.0),
    heightDimension: .absolute(1)),
    elementKind: UICollectionView.elementKindSectionFooter,
    alignment: .bottom)
    section.boundarySupplementaryItems = [footer]
```

2. Register them on your collection view setup:

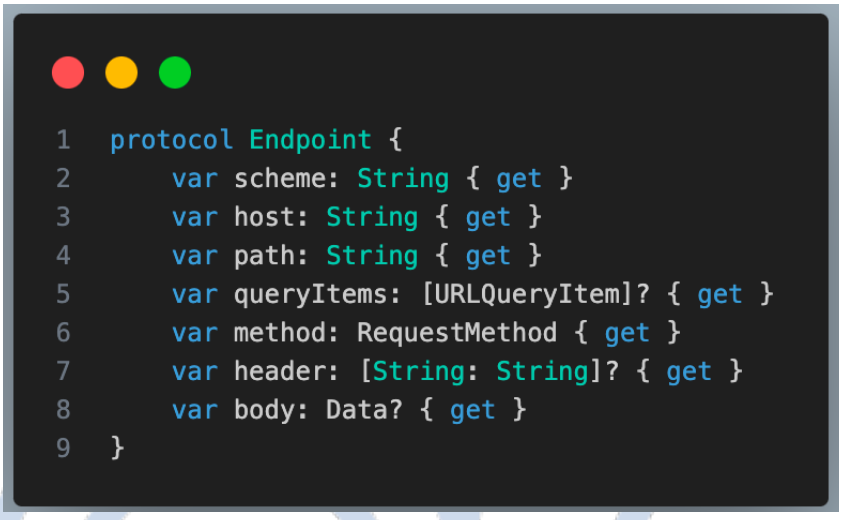
```swift
collectionView.register(HeaderView.self,
forSupplementaryViewOfKind: UICollectionView.elementKindSectionHeader,
withReuseIdentifier: "header")
collectionView.register(FooterView.self,
forSupplementaryViewOfKind: UICollectionView.elementKindSectionFooter,
withReuseIdentifier: "footer")

```

Networking Layer

Firstly, we set up base layer's files. They are the core of out networking layer.

1. Define an *Endpoint* protocol:



```swift
protocol Endpoint {
    var scheme: String { get }
    var host: String { get }
    var path: String { get }
    var queryItems: [URLQueryItem]? { get }
    var method: RequestMethod { get }
    var header: [String: String]? { get }
    var body: Data? { get }
}
```

This is the core of all our future endpoints. It makes sure you configure (override) each *URL* part (host, path, headers, etc…)

2. Extend said protocol to override host by default:

```swift
extension Endpoint {
    var scheme: String {
        return "https"
    }
}
```

Here we define scheme by default because it will be rarely changed, since we mostly use TLS.

3. Define *RequestError* enum to catch all possible errors within our request:

```swift
enum RequestError: Error {
    case decode
    case invalidURL
    case noResponse
    case unauthorized
    case unexpectedStatusCode
    case unknown
    case networkError(Error)
}
```

4. Define *RequestMethod* enum that includes the most common request types:

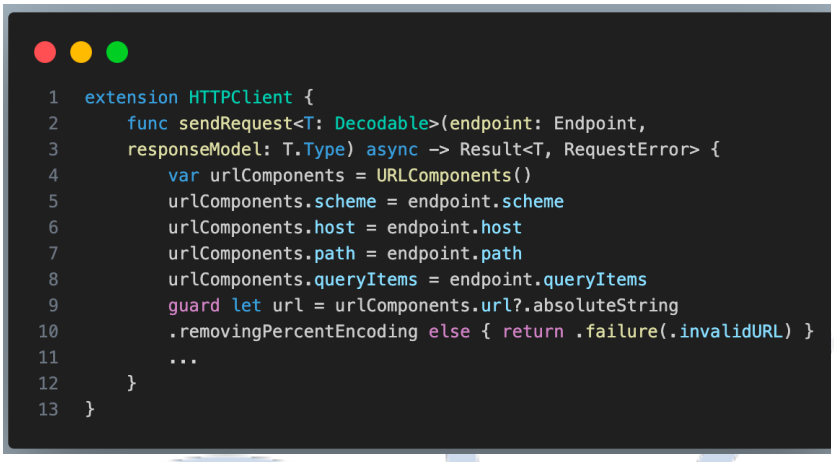
```swift
enum RequestMethod: String {
    case delete = "DELETE"
    case get = "GET"
    case patch = "PATCH"
    case post = "POST"
    case put = "PUT"
}
```

5. Define a *HTTPClient* protocol which will be our app root request performer:

```swift
protocol HTTPClient {
    func sendRequest<T: Decodable>(endpoint: Endpoint,
    responseModel: T.Type) async -> Result<T, RequestError>
}
```

Every consumption we make will conform this protocol, so it uses the same function in every one of them, instead of defining an individual consumption every time.

6. Extend this protocol to setup *sendRequest()* function



```
extension HTTPClient {
    func sendRequest<T: Decodable>(endpoint: Endpoint,
    responseModel: T.Type) async -> Result<T, RequestError> {
        var urlComponents = URLComponents()
        urlComponents.scheme = endpoint.scheme
        urlComponents.host = endpoint.host
        urlComponents.path = endpoint.path
        urlComponents.queryItems = endpoint.queryItems
        guard let url = urlComponents.url?.absoluteString
        .removingPercentEncoding else { return .failure(.invalidURL) }
        ...
    }
}
```

Set up the urlComponents with the ones you modified in your Endpoint Class *(we will see this in a bit).*

Build the URL with said components.

7. Build and perform the URLRequest with said *Endpoint* setups:

```
var urlRequest = try URLRequest(url: url.asURL())
    urlRequest.httpMethod = endpoint.method.rawValue
    urlRequest.allHTTPHeaderFields = endpoint.header

    if let body = endpoint.body {
     urlRequest.httpBody = body
    }

let request = AF.request(urlRequest).serializingDecodable(responseModel)
```

8. Use the request's response to handle statusCodes behavior:

```swift
guard let response = await request.response.response else {
    return .failure(.noResponse)
}

switch response.statusCode {
    case 200...299:
    guard let decodedResponse = try? await request.value else {
        return .failure(.decode)
    }

    return .success(decodedResponse)
    case 401:
        return .failure(.unauthorized)
    default:
        print("Response status code: \(response.statusCode)")
        return .failure(.unexpectedStatusCode)
}
```

Once we have our base setup, we can already start understanding how the requests will work and how to perform them. We will need to setup 3 objects to make our requests: ***Endpoint***, ***Model*** and ***Services***.

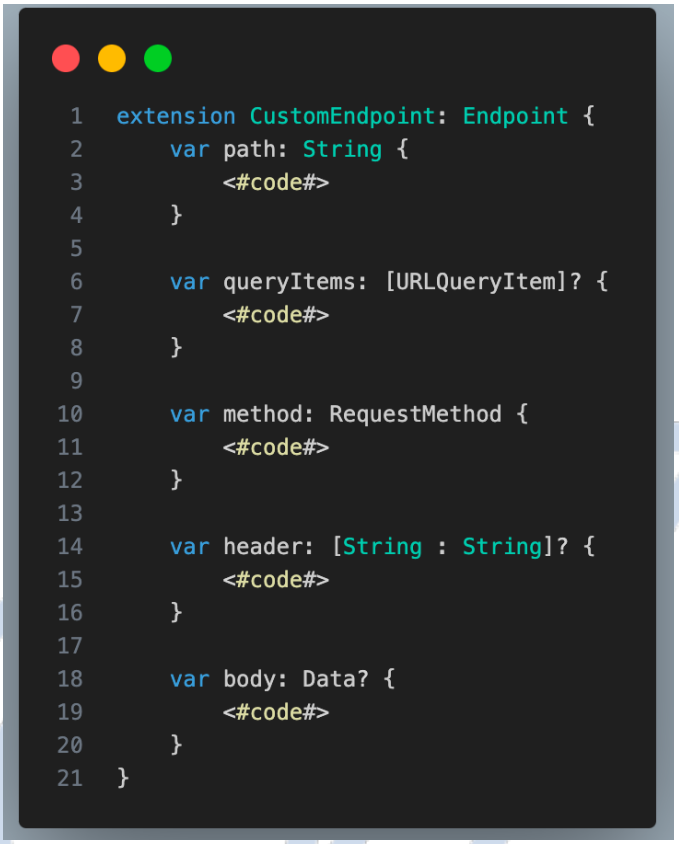
Let's break it down:

Endpoint

1. Define an endpoint inside an enum:

```swift
enum CustomEndpoint {
    case uniqueEndpoint
}
```

2. Extend this enum and make it conform our *Endpoint* protocol:



```
extension CustomEndpoint: Endpoint {
    var path: String {
        <#code#>
    }

    var queryItems: [URLQueryItem]? {
        <#code#>
    }

    var method: RequestMethod {
        <#code#>
    }

    var header: [String : String]? {
        <#code#>
    }

    var body: Data? {
        <#code#>
    }
}
```

As soon as we do this, we will be asked to add the missing stubs. Here we will override each URL value as we need, depending in our request. Let's look at a simple request.

For *Shows* endpoint, we can set it up like so:

3. Override host:

```
var host: String {
    switch self {
    case .shows:
        return "streaming-availability.p.rapidapi.com"
    }
}
```

4. Override path:

```swift
var path: String {
    switch self {
    case .shows:
        return "/shows/search/filters"
    }
}
```

5. Override queryItems:

```swift
var queryItems: [URLQueryItem]? {
    switch self {
    case .shows:
        return [URLQueryItem(name: "country", value: "us"),
        URLQueryItem(name: "cursor", value: "240855:'Neath Canadian Skies")]
    }
}
```

6. Override method:

```swift
var method: RequestMethod {
    switch self {
    case .shows:
        return .get
    }
}
```

This is simple; you just change the request method needed for that specific request.

7. Override header:

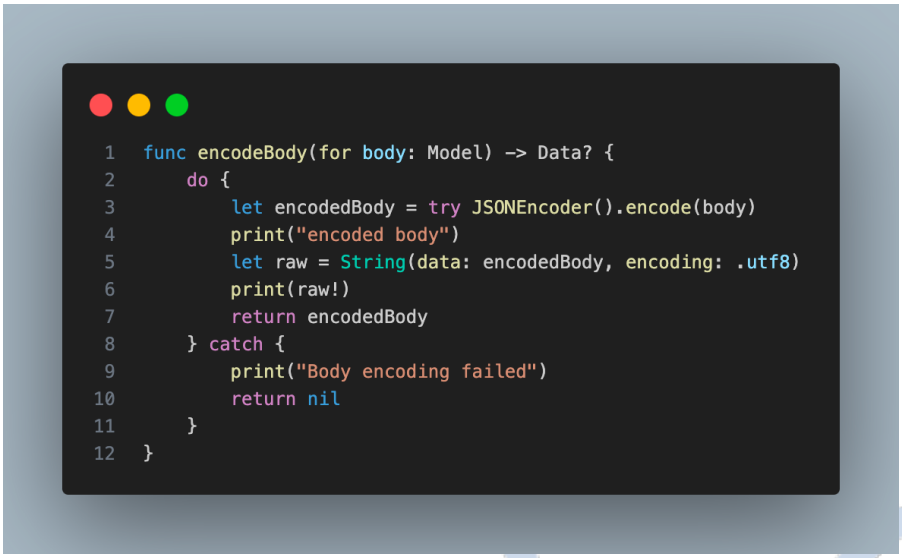
```swift
var header: [String : String]? {
    switch self {
    case .shows:
        let token = "63ff6c1525msh99771ce8dac8c28p17daa8j[redacted]"
        return [
            "x-rapidapi-key": token,
            "x-rapidapi-host" : self.host
        ]
    }
}
```

Here is how we can setup our headers in an endpoint if needed.

8. Override body:

```swift
var body: Data? {
    switch self {
    case .shows:
        return nil
    }
}
```

In case the request type is .get, if we have a .post request we must define a body to deliver. Since we serialize body objects as *Data* objects, this is how we can encode our body (following a body model):



```swift
func encodeBody(for body: Model) -> Data? {
    do {
        let encodedBody = try JSONEncoder().encode(body)
        print("encoded body")
        let raw = String(data: encodedBody, encoding: .utf8)
        print(raw!)
        return encodedBody
    } catch {
        print("Body encoding failed")
        return nil
    }
}
```

Model

1. Define the model our request response should be aligned with:

```swift
struct User {
  let id: String
  let firstName: String
  let lastName: String
  let email: String?
}
```

Services

1. Define a *Serviceable* protocol:

```swift
protocol ShowsServiceable {
    func getShows() async -> Result<Shows,
    RequestError>
}
```

2. Define a *Service* class that conforms our *Serviceable* and *HTTPClient* protocols:

```
class ShowsServices: HTTPClient, ShowsServiceable {
    func getShows() async -> Result<Shows, RequestError> {
        return await sendRequest(endpoint: ShowsEndpoint.shows,
        responseModel: Shows.self)
    }
}
```

Since we made it conform *HTTPClient* we need to setup and send a request (remember this function inside the client will perform the request with the given data in the parameters).

This will take our defined *Endpoint* and *Model* and will send the request. Now, we have setup out request completely. How can we retrieve the response data for future usage? Let's meet our ***ViewModel***.

View Model

1. Setup a ViewModel class and make a reference for our *Service* class:

```
class CustomViewModel {
    var customService: CustomService()

    func fetchData() {

    }
}
```

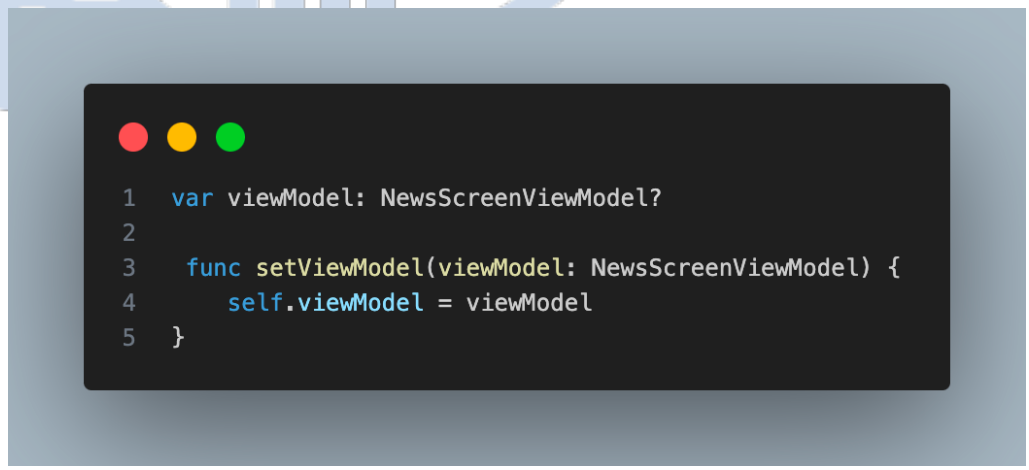
2. Since our request are asynchronous, we must use *Task* block to perform our requests:

```
func fetchData() {
        Task(priority: .userInitiated) {
            let result = await customService.getCustomData()
            switch result {
                case .success(let data):
                handleData(data)
                case .failure(let error):
                handleError(error)
            }
        }
    }
```

This will basically be the base of our consumptions. It performs the getData() function inside our service (remember in it the sendRequest() function makes the request) and then we can handle that request if it succeed or if it fails.

Here now you can store that data in an outside variable and use it anywhere else.

The common usage is inside a viewController. First, you should setup your ViewModel object.



```
var viewModel: NewsScreenViewModel?

 func setViewModel(viewModel: NewsScreenViewModel) {
    self.viewModel = viewModel
}
```

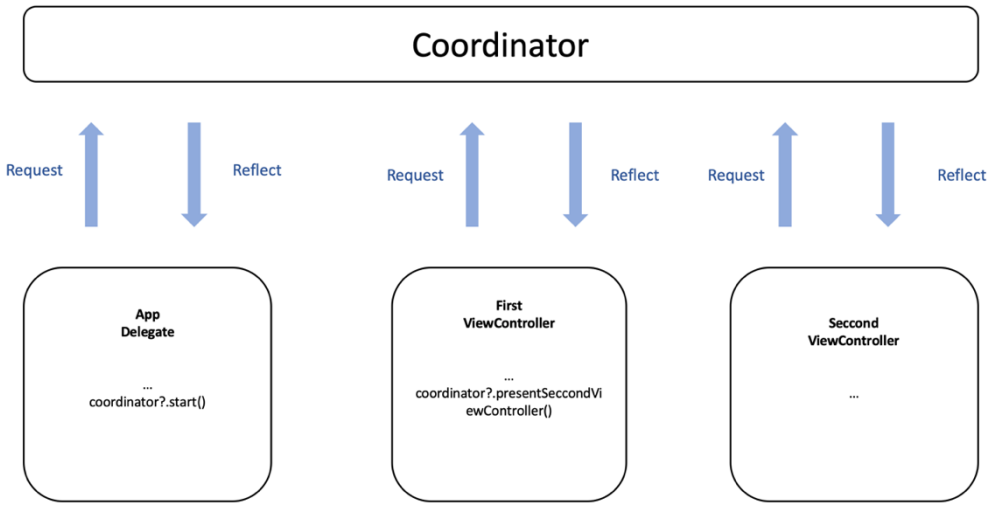
You should initialize this in the invoking coordinator (*Refer to Coordinator Pattern Documentation).*

Then, in the viewDidLoad, call the fetchData() function and adjust its behavior:

```
override func viewDidLoad() {
    super.viewDidLoad()
    loadLottie()
    fetchData {
        <#code#>
    } failure: {
        <#code#>
    }
}
```

Coordinator Pattern

A coordinator navigation pattern uses a *UINavigationController* object as root to present or remove screens from the navigation stack. Let's look at a definition:



Asset Source.

The Coordinator Pattern is a design pattern that aims to centralize and manage the navigation flow in an iOS application. It provides us a clean architecture by separating the responsibilities of view controllers, making them more focused on their specific tasks.

Retrieved from Medium's Developer Article.

To set it up and use it in your environment, follow these steps:

1. In a new file, define a Coordinator protocol:

```swift
protocol Coordinator {
    var navigationController: UINavigationController? { get set }
    func start()
    func push(viewController: UIViewController, animated: Bool)
    func showNavigationBar(animated: Bool)
    func hideNavigationBar(animated: Bool)
}
```

   Define inside it all the desired functionality you want it to implement. For instance, the basic navigation functions (push, pop, show or hide navigation bar, enable or disable gesture recognizers)

2. Extend it and implement what each function should perform:

```swift
extension Coordinator {
    func push(viewController: UIViewController, animated: Bool) {
        self.navigationController?.pushViewController(viewController, animated: animated)
    }

    func showNavigationBar(animated: Bool = false) {
        self.navigationController?.navigationBar.topItem?.title = ""
        self.navigationController?.navigationBar.isHidden = false
    }

    func hideNavigationBar(animated: Bool = false) {
        self.navigationController?.navigationBar.isHidden = true
    }
}
```

   Remember this pattern is based on a navigation controller.

3. Define an additional protocol that refers to the base coordinator:

```swift
protocol Coordinating {
    var coordinator: Coordinator? { get set }
}
```

All our following coordinators will conform this protocol. They will use it to be based on our base coordinator.

4. Create your first custom screen coordinator:

```swift
class UsersListScreenCoordinator: Coordinating {
    var coordinator: (any Coordinator)?

    init(coordinator: Coordinator?) {
        self.coordinator = coordinator
    }

    func showNavigationBar(animated: Bool = false) {
        self.coordinator?.showNavigationBar(animated: animated)
    }

    func hideNavigationBar(animated: Bool = false) {
        self.coordinator?.hideNavigationBar(animated: animated)
    }

    func popToRootController(animated: Bool) {
        self.coordinator?.popToRootController(animated: animated)
    }

    func pop(animated: Bool) {
        self.coordinator?.pop(animated: animated)
    }

    func enableDragPopGesture() {
        self.coordinator?.enableDragPopGesture()
    }

    func disableDragPopGesture() {
        self.coordinator?.disableDragPopGesture()
    }

    func pushToUserMap(with data: User, showAllUsers: Bool = false) {
        if let detailScreen = UIStoryboard(name: UserListMapScreenVC.storyboard, bundle: nil)
            .instantiateViewController(withIdentifier: UserListMapScreenVC.identifier) as? UserListMapScreenVC {
            detailScreen.data = data
            detailScreen.showAllUsers = showAllUsers
            self.coordinator?.push(viewController: detailScreen, animated: true)
        }
    }
}
```

This is how all your coordinators should look like. Create a new coordinator class and make it conform Coordinating protocol. Then, since the class should conform the base coordinator's protocol, it will have the base functions you defined at the beginning (push, pop, hide or show navbar…). For instance, if you want to navigate to a following view, instantiate it and use the push() function.

If our detail screen had a coordinator and/or a view model, they should be initialized here. For instance (detailScreen.setViewModel())

This way the objects such as ViewModels or following Coordinators are previously initialized for later usage. As you can already notice, all coordinators must have prior initialization. Therefore, a base initialization should exist.

5. A main coordinator (or initial coordinator) should be defined:

```swift
class RootCoordinator: Coordinator {

    var navigationController: UINavigationController?

    init(navigationController: UINavigationController) {
        self.navigationController = navigationController
    }

    func start() {
        if let launchScreen = UIStoryboard(name: CustomLaunchScreenVC.storyboard, bundle: nil)
            .instantiateViewController(withIdentifier: CustomLaunchScreenVC.identifier) as? CustomLaunchScreenVC {
            var controller: UIViewController & Coordinating = launchScreen
            controller.coordinator = self
            navigationController?.setViewControllers([controller], animated: false)
        }
    }

    func push(viewController: UIViewController, animated: Bool) {
        self.navigationController?.pushViewController(viewController, animated: animated)
    }

    func pop(animated: Bool) {
        self.navigationController?.popViewController(animated: animated)
    }
}
```

In *El Pregonero App* case, this initial coordinator presents our *LaunchScreen*. It instantiates it and sets it as the first screen at the navigation stack. It should be initialized in the *SceneDelegate* in our app.

Conclusion

The implementation of the described functionality in *El Pregonero* is paramount for several reasons. Firstly, in today's fast-paced world, users crave seamless access to information and entertainment, making it crucial to offer a diverse range of content in one convenient platform. By integrating features such as news lists, program details, and sports highlights, we ensure that users stay informed and engaged, catering to their varied interests. Moreover, the inclusion of a user list and location-based services fosters community connection, allowing users to discover and interact with others nearby. Through careful design and execution, we've created a user flow that prioritizes user experience and accessibility, ultimately enhancing the overall value and utility of *El Pregonero*.

If you have any questions, encounter issues, or would like to provide feedback or suggestions for enhancements, please don't hesitate to reach us out at dafrprof@gmail.com. Your input is invaluable to us as we strive to deliver the best possible user experience.