

# iOS Technical Case

Rafael Neuwirth Swierczynski

<https://github.com/rafaelSwi/JambleiOSTechnicalCase>

## HOW TO RUN IT

Since no external dependencies were used, theoretically no additional setup would be necessary beyond cloning the project and running it. For informational purposes, follows exact configuration I used:

**OS:** macOS Sequoia 15.7.3

**Computer:** Macbook Pro M1 2021 16GB

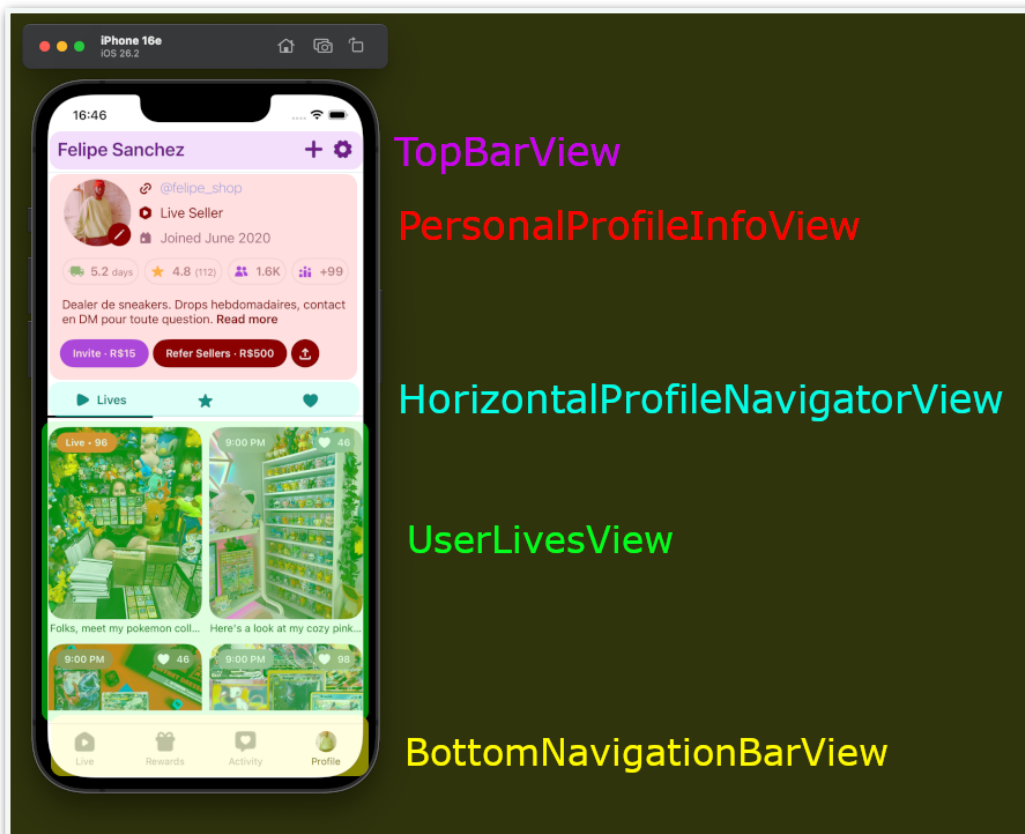
**Xcode:** 26.2

**Simulator iOS:** 26.2

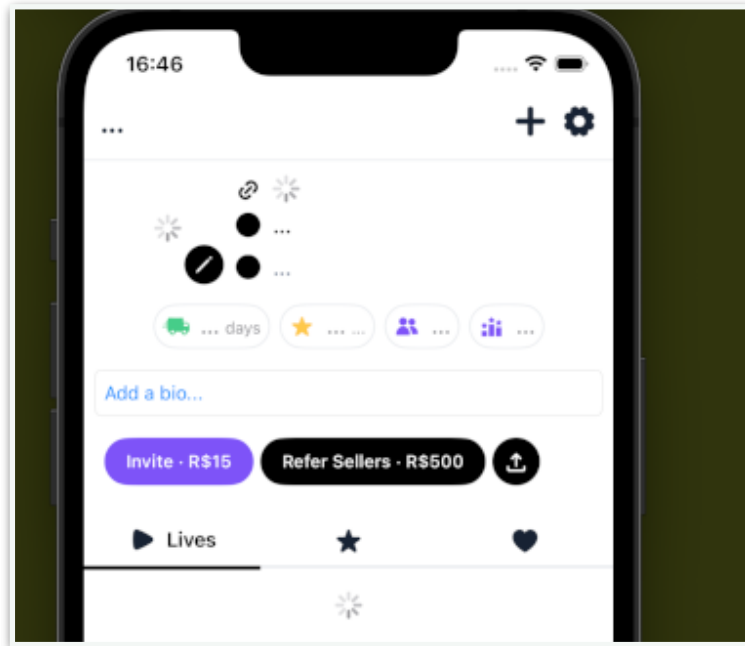
**Simulator Device:** iPhone 16e

## HOW IT WORKS

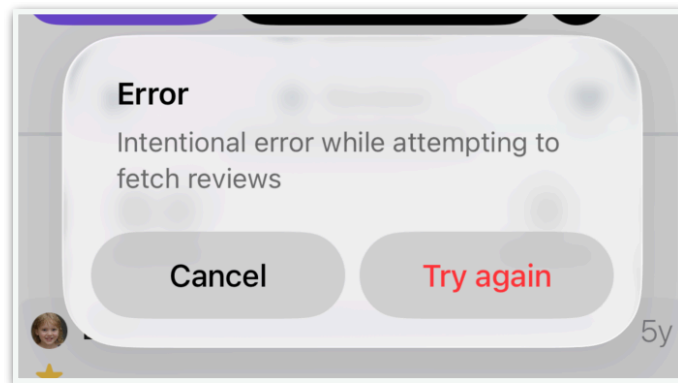
The UI is separated into several different components.



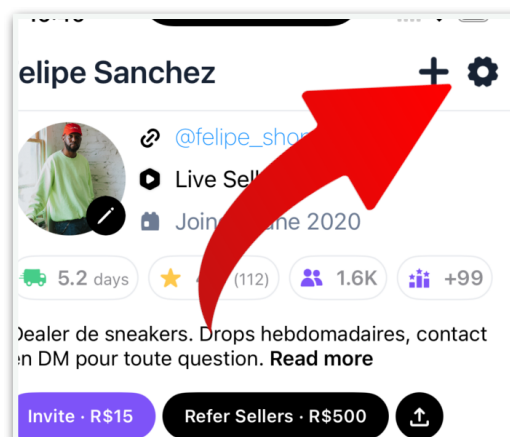
The UI components have placeholders while they wait for the data to load, so by default a 1-second delay is simulated, and during this 1 second you can observe the loading status of the views.

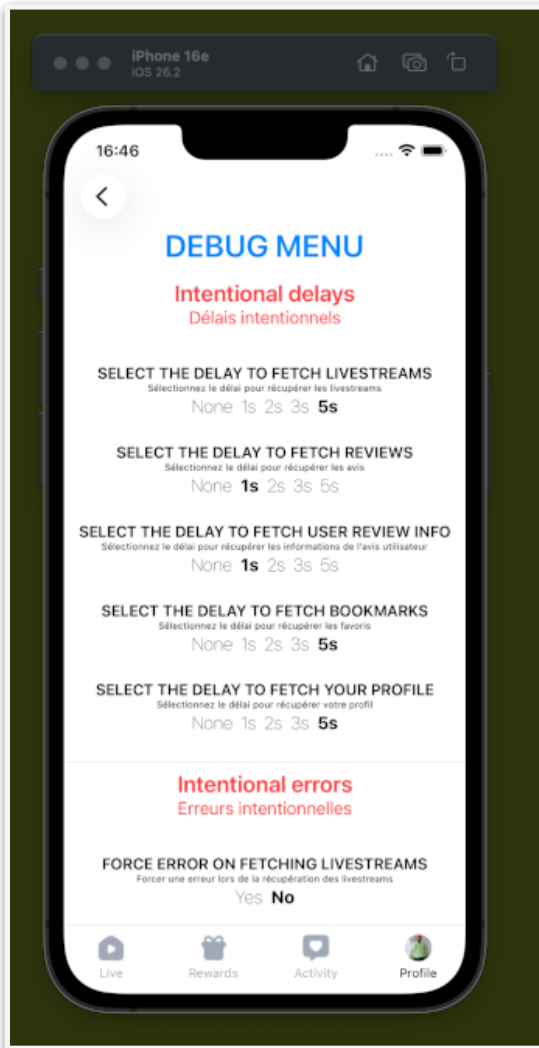


Errors can also occur, so if an error happens, a message will be displayed on the screen with an option to try again.



It's quite unlikely you'll encounter a genuine error since these are all functions that pull data locally and simulate a delay, but if you want to induce this behavior, a Debug Menu has been created.



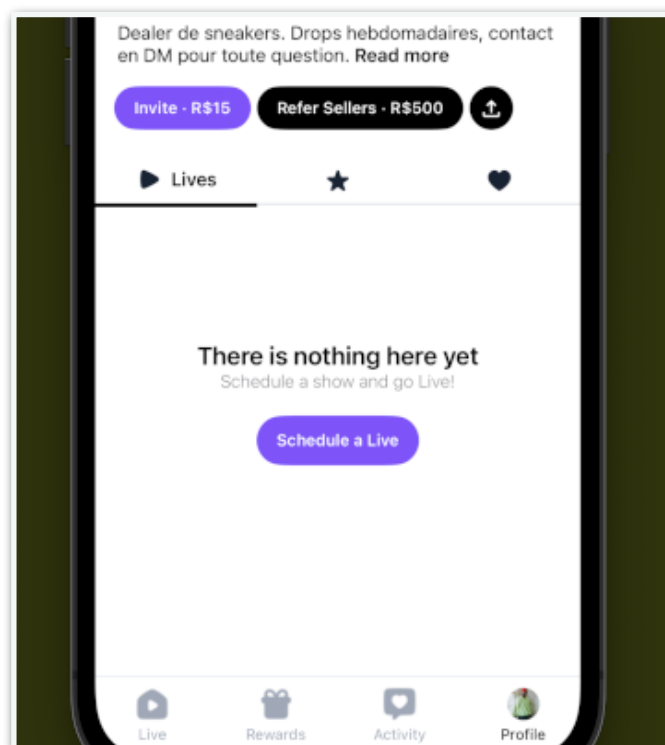


It's difficult to say that the Debug Menu is somewhat pretty, but it's functional for what it promises to be. In it, you can simulate the delay for each fake fetch, whether an error will occur, and also simulate empty search results.

When you select any option, you will automatically be taken out of the debug menu to test the newly selected option.

*The changed options are not saved locally.*

This is what the "Lives" tab looks like if there are no results.



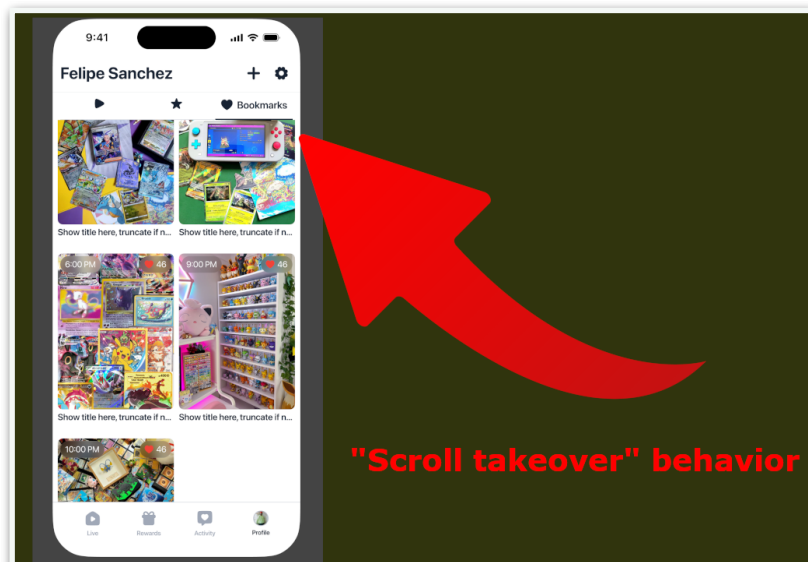
Leaving aside the fact that all asynchronous calls are fake and the fetch functions are extremely simple, the project's file structure was separated as follows:

```
└─ Project/
   └─ Extensions/
      └─ ExampleExtension.swift
   └─ Miscellaneous/
      └─ DebugSettings.swift
   └─ Models/
      └─ GenericUser.swift
   └─ Services/
      └─ UserService.swift
   └─ UI/
      └─ ExampleScreen/
         └─ Components/
            └─ ExampleHeader/
               └─ ExampleHeaderView.swift
               └─ ExampleHeaderViewModel.swift
         └─ ExampleScreenView.swift
         └─ ExampleScreenViewModel.swift
```

There are many features in the app that don't work, such as the feature to change your profile picture. Only what was requested in the PDF was implemented, feel free to test it.

## WHAT WAS LEFT UNDONE

Regarding what was mentioned in the PDF, everything was implemented. However, there was one aspect shown in the Figma design that I was not able to replicate without relying on external dependencies.



The swipe behavior where the view expands and takes over the screen was the only aspect I was unable to fully replicate while still meeting all other requirements. I did find a way to reproduce this behavior by removing the TabBar, however, doing so prevented horizontal swipe navigation between tabs. Since horizontal swiping was explicitly listed as a requirement in the PDF, I choose to not use the "Scroll takeover" behavior.

Aside from that, all other requirements from both the Figma design and the PDF were fulfilled. I sincerely hope the code will not be reviewed too deeply in terms of polish, as it was written under a tight deadline in order to faithfully replicate the described behavior within two days.

## WHAT WAS DONE BY AI

If we're talking in terms of generated code, only very specific functions like "timeAgo" which basically returns a string showing how much time has passed since a specific date.

```
// IA made this
func timeAgo(from date: Date) -> String {
    let now = Date()
    let calendar = Calendar.current
    let components = calendar.dateComponents([.year, .month, .weekOfYear, .day, .hour, .minute, .second], from: date, to: now)

    if let year = components.year, year > 0 {
        return "\(year)y"
    } else if let month = components.month, month > 0 {
        return "\(month)mo"
    } else if let week = components.weekOfYear, week > 0 {
        return "\(week)w"
    } else if let day = components.day, day > 0 {
        return "\(day)d"
    } else if let hour = components.hour, hour > 0 {
        return "\(hour)h"
    } else if let minute = components.minute, minute > 0 {
        return "\(minute)m"
    } else if let second = components.second, second > 0 {
        return "\(second)s"
    } else {
        return "now"
    }
}
```

This is the kind of thing I'd prefer to use AI for

Regarding some of the fake fetch functions and View/ViewModel implementations, nothing was generated. I did use AI to clarify a few doubts and to help with things I didn't immediately recall how to implement (for example, adding a translucent background to the livestream overlay), but nothing beyond that.

# ARE THE VIEWS REUSABLE?

Since I don't know the actual models returned by the backend, the views were built based on the fake models I created. For example, I did not know how the backend provides a user's average rating, so I assumed it would be a Float and implemented the UI accordingly.