Using Async/Await in SwiftUI: Smooth UI Updates with Asynchronous Data Fetching

Tech Savvy Scribe · Follow
3 min read · May 4

In this article, we will explore how to use Swift's Async/Await feature in SwiftUI, focusing on smooth UI updates with asynchronous data fetching. We will start with basic examples and progress towards more advanced use cases, demonstrating how Async/Await can improve the user experience of your SwiftUI apps.

Example 1: Basic Async Data Fetching

Let's start with a simple example of fetching an integer value asynchronously and displaying it in a SwiftUI view.

Model

```
class DataFetcher: ObservableObject {
    @Published var number: Int?

    func fetchNumber() async {
        await Task.sleep(1_000_000_000) // Sleep for 1 second
        DispatchQueue.main.async {
            self.number = Int.random(in: 1...100)
        }
    }
}
```

SwiftUI View

```
struct ContentView: View {
    @StateObject var dataFetcher = DataFetcher()

    var body: some View {
        VStack {
            if let number = dataFetcher.number {
                Text("Random number: \(number)")
            } else {
                Text("Loading...")
            }
            Button("Fetch Number") {
                Task {
                    await dataFetcher.fetchNumber()
                }
            }
        }
    }
}
```

Example 2: Fetching Data from a Remote API

In this example, we will fetch data from a remote API and display the results in a SwiftUI List.

Model

```
struct Post: Codable, Identifiable {
    let id: Int
    let title: String
    let body: String
}

class PostFetcher: ObservableObject {
    @Published var posts: [Post] = []

    func fetchPosts() async {
        let url = URL(string: "https://jsonplaceholder.typicode.com/posts")!
        do {
            let (data, _) = try await URLSession.shared.data(from: url)
            let fetchedPosts = try JSONDecoder().decode([Post].self, from: data
            DispatchQueue.main.async {
                self.posts = fetchedPosts
            }
        } catch {
            print("Error fetching posts: \(error)")
        }
    }
}
```

SwiftUI View

```
struct ContentView: View {
    @StateObject var postFetcher = PostFetcher()

    var body: some View {
        NavigationView {
            List(postFetcher.posts) { post in
                VStack(alignment: .leading) {
                    Text(post.title)
                        .font(.headline)
                    Text(post.body)
                        .font(.subheadline)
                }
            }
            .navigationTitle("Posts")
            .task {
                await postFetcher.fetchPosts()
            }
        }
    }
}
```

Example 3: Error Handling and Loading State

In this example, we will add error handling and a loading state to our previous example.

Model

```
enum DataError: Error {
    case networkError(Error)
    case decodingError(Error)
}

class PostFetcher: ObservableObject {
    @Published var posts: [Post] = []
```

```
    @Published var error: DataError?
    @Published var isLoading: Bool = false

    func fetchPosts() async {
        isLoading = true
        let url = URL(string: "https://jsonplaceholder.typicode.com/posts")!
        do {
            let (data, _) = try await URLSession.shared.data(from: url)
            let fetchedPosts = try JSONDecoder().decode([Post].self, from: data
            DispatchQueue.main.async {
                self.posts = fetchedPosts
                self.isLoading = false
            }
        } catch {
            print("Error fetching posts: \(error)")
            DispatchQueue.main.async {
            self.isLoading = false
            if let decodingError = error as? DecodingError {
              self.error = .decodingError(decodingError)
            } else {
              self.error = .networkError(error)
            }
        }
    }
  }
}
```

SwiftUI View

```
var postFetcher = PostFetcher()

var body: some View {
    NavigationView {
        Group {
            if postFetcher.isLoading {
                ProgressView()
            } else if let error = postFetcher.error {
                VStack {
```

```
                    Text("Error:")
                        .font(.headline)
                    Text(errorDescription(error))
                        .font(.body)
                        .multilineTextAlignment(.center)
                }
            } else {
                List(postFetcher.posts) { post in
                    VStack(alignment: .leading) {
                        Text(post.title)
                            .font(.headline)
                        Text(post.body)
                            .font(.subheadline)
                    }
                }
            }
        }
        .navigationTitle("Posts")
        .task {
            await postFetcher.fetchPosts()
        }
    }
}

private func errorDescription(_ error: DataError) -> String {
    switch error {
    case .networkError(let networkError):
        return "Network error: \(networkError.localizedDescription)"
    case .decodingError(let decodingError):
        return "Decoding error: \(decodingError.localizedDescription)"
    }
}
```

Conclusion

Using Async/Await in SwiftUI enables us to create smooth UI updates when fetching asynchronous data. By with SwiftUI's reactive nature, we can build responsive and user-friendly applications that handle asynchronous data fetching, loading states, and error handling with ease.

By following the examples provided in this article, you can improve the user experience of your SwiftUI apps and write more maintainable and readable code.

> *If you enjoyed the article and would like to show your support, be sure to:*
>
> 👏 *Applaud for the story (50 claps) to help this article get featured*
>
> 👉 *Follow me on* **Medium**
>
> *Check out more content on my* ***Medium profile***

Swift | Swift Async Await | Swiftui Async Data | Swiftui | IOS App Development