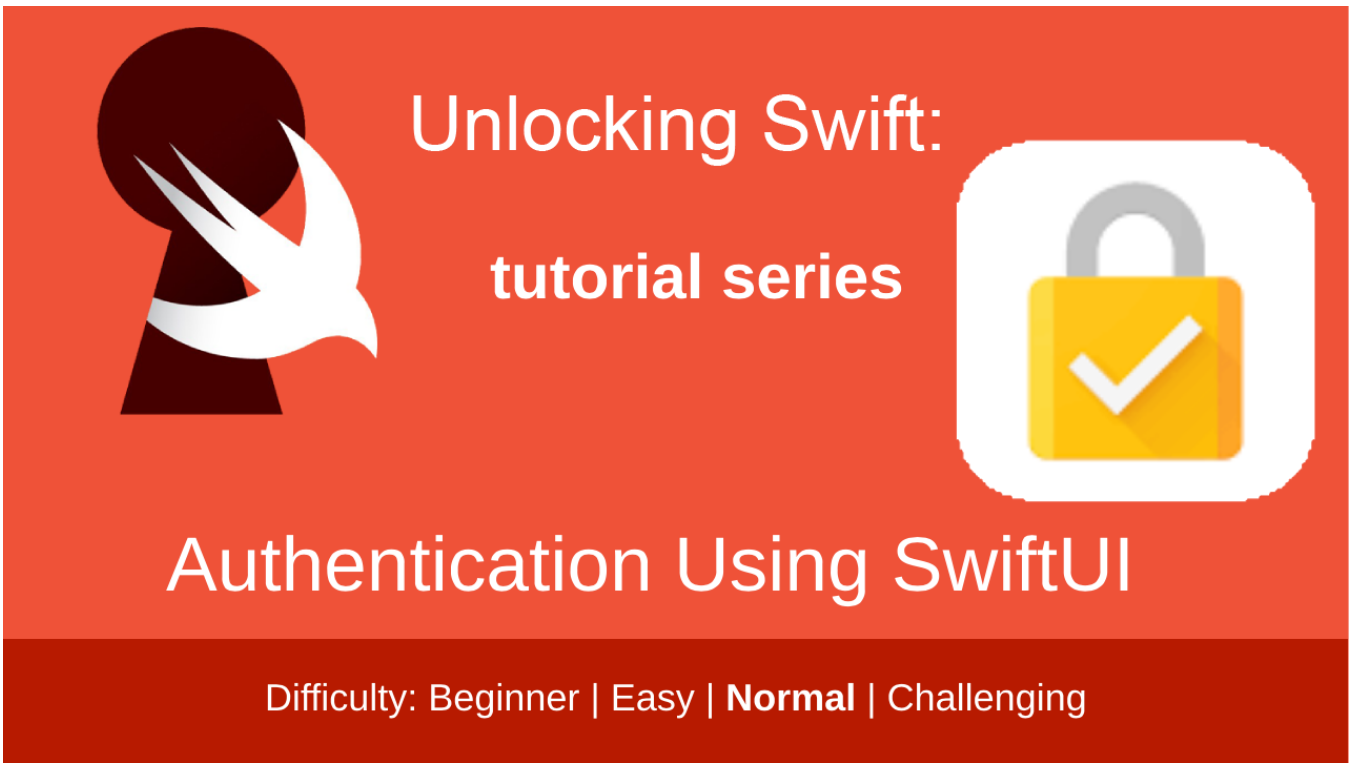
Steven Curtis

13.5K Followers About Follow

Authentication Using SwiftUI

Who are you, again?

Steven Curtis Jul 16, 2021 · 4 min read ★

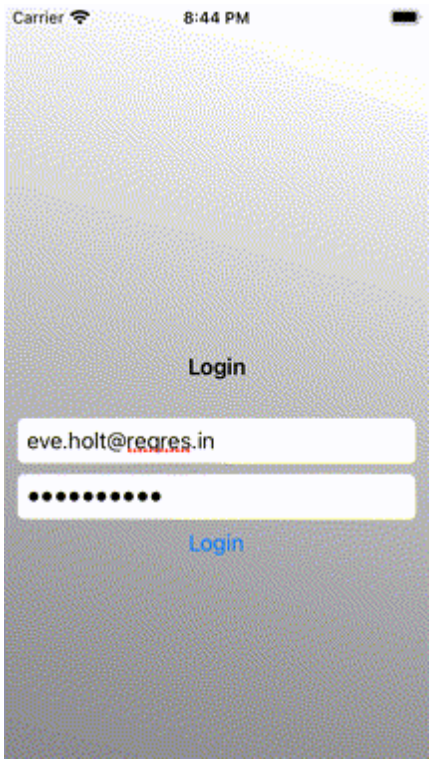


Before we start

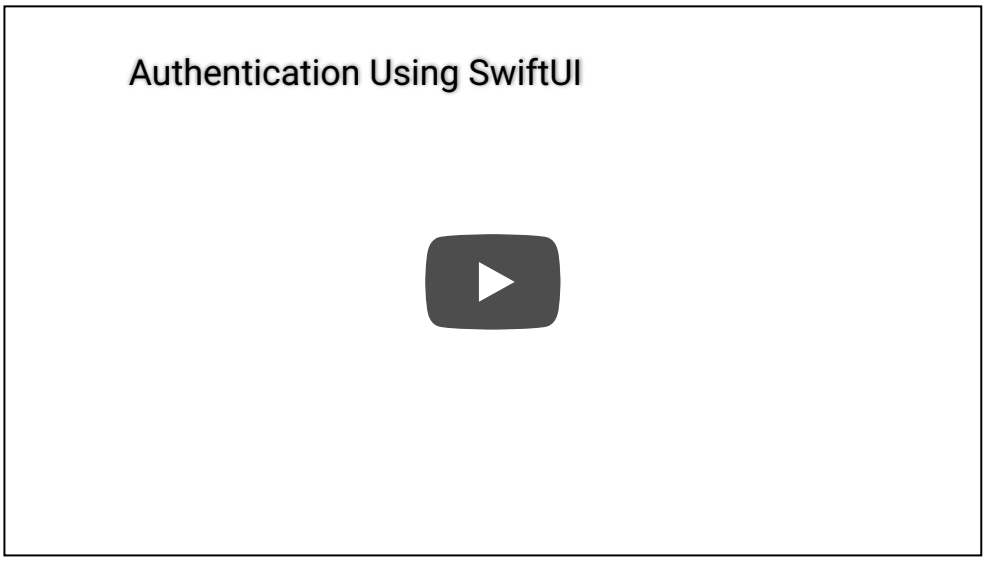
Difficulty: Beginner | Easy | **Normal** | Challenging

The opportunity

The completed project



The supporting video



Prerequisites

Keywords and Terminology

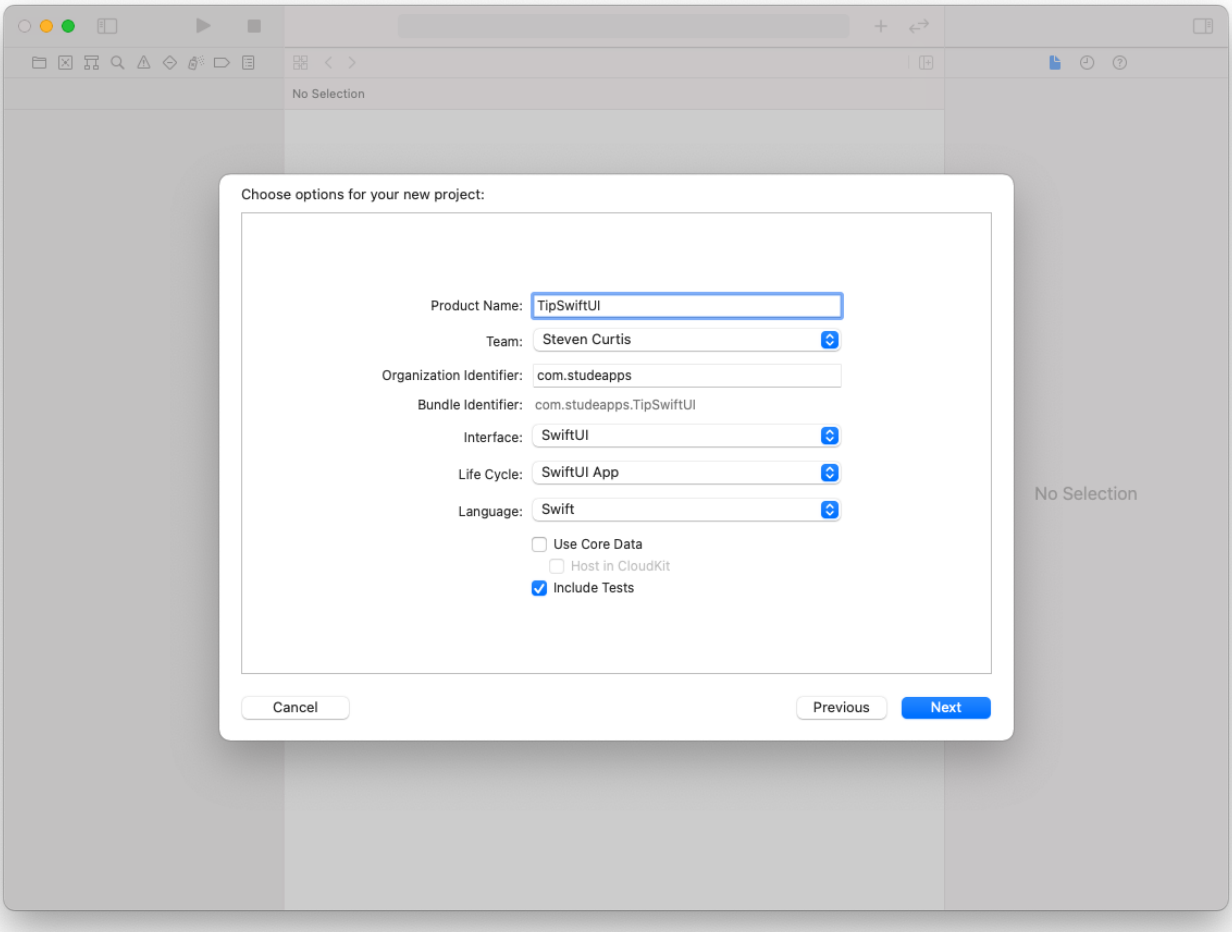
ObservableObject: A type of object with a publisher that emits before the object has changed

SwiftUI: A simple way to build user interfaces across Apple platforms

Setting up the project

This project does have much to do with my SwiftUI tip calculator project, and in fact the setup is the same

We can create a new SwiftUI project, and here we are using the SwiftUI template to do so:



Create a new SwiftUI App, of course change the product name though!

We also take the same approach to the initializing the ViewModel and View:

```swift
@main
struct SwiftUIAuthenticationApp: App {
    var body: some Scene {
        WindowGroup {
            let nm = NetworkManager(session: URLSession.shared)
            let viewModel = LoginViewModel(networkManager: nm)
            LoginView(viewModel: viewModel)
        }
    }
}
```

init.swift hosted with ❤️ by GitHub view raw

which then opens the door to being able to login, using `jsonplaceholder.typicode.com` and their `/api/login` login.

The credentials

In order to log into this endpoint, we can use the fixed default username and password:

`eve.holt@reqres.in`

and

`cityslicka`

in order to gain access to the application. No problem!

The Technique

I'm going to use `@Published` properties in for the username, password `String` values and the loading and login `Boolean` values.

The reason for this is that I like to think that the `@Published` property wrapper is at the heart of `SwiftUI`, the easiest to understand as it enables objects to be able to announce when changes occur. In this case, the properties (previously explained) update the model which inherits from `ObservableObject` so it can make it clear that the property has changed — to our Swift view in the `UIViewController`.

Once we have met the requisites for loading the "MenuView" in this case, we will use a `NavigationLink` to do just that.

our network call to the endpoint with the user-entered username and password. If the username and password are correct, the endpoint will return a successful response that mean we then flick our login boolean to true (more on this later). The error handling is out of scope here, but I will make a loading indicator give the user some indication that the application is loading. Oh, and don't forget these actions need to happen on the main thread as we are going away to make that network call on a background thread.

So what does that all look like? I've got this separated out into different files and classes (ok, structs as this is `SwiftUI`):

Where we go to: MenuView

We are going to move to a new view from the initial login. This isn't a "big deal", of course I wouldn't make this too tricky. So, in that spirit, I've made us go to a view that simply shows "Hello, World!" to the end user. Simplistic, and a little boring. That is here:

```
1  struct MenuView: View {
2      var body: some View {
3          Text(/*@START_MENU_TOKEN@*/"Hello, World!"/*@END_MENU_TOKEN@*/)
4      }
5  }
6
7  struct MenuView_Previews: PreviewProvider {
8      static var previews: some View {
9          MenuView()
10     }
11 }
```

MenuView.swift hosted with ❤️ by GitHub view raw

Driving the Logic: The ViewModel

The ViewModel has those published properties, and makes the network call to attempt to log us in.

Note how the properties are changed on the main queue to make sure that we aren't trying to update the UI from a background thread.

```
1  import Foundation
2  import NetworkLibrary
```

```swift
    @Published var username: String = "" // "eve.holt@reqres.in"
    @Published var password: String = "" // "cityslicka"
    @Published var loading: Bool = false
    @Published var login: Bool = false

    private var anyNetworkManager: AnyNetworkManager<URLSession>

    init<T: NetworkManagerProtocol>(networkManager: T) {
        self.anyNetworkManager = AnyNetworkManager(manager: networkManager)
    }

    func loginNetworkCall() {
        let data: [String : Any] = ["email": username, "password": password]
        self.loading = true
        self.login = false
        anyNetworkManager.fetch(url: API.login.url, method: .post(body: data), comple
            switch res {
            case .success(let data):
                let decoder = JSONDecoder()
                if let _ = try? decoder.decode(Login.self, from: data) {
                    DispatchQueue.main.async {
                        self?.login = true
                    }
                } else {
                    // potential here to update the view with an error
                }
            case .failure(let error):
                // potential here to update the view with an error
                print ("Error \(error)")
            }
            DispatchQueue.main.async {
                self?.loading = false
            }
        })
    }
}
```

LoginViewModel.swift hosted with ❤️ by GitHub view raw

Why didn't I add the error handling here? The truth is to keep this tutorial relatively easy to follow (which is also a compelling reason to keep the network library out of this too).

`ObservedObject` to allow us find out if there are any changes. This is no problem in the interpretation of the code below.

I thought while coding this that I might as well make a background layer with a gradient; what a laugh I thought. It turned out that it wasn't too difficult and this gives us little bit better interpretation of a "real" project that you might use in your Swift study or work.

You might also be interested to see that the navigation link does not have a label, we are going to programatically move to the `MenuView` when the login property is set to true in the view model.

```
NavigationLink(
    destination: MenuView(),
    isActive: self.$viewModel.login,
    label: {}
)
```

NavigationLink.swift hosted with ❤️ by GitHub view raw

which is then of course driven by the connected viewModel, which could look something like the following:

```
struct LoginView: View {
    @ObservedObject var viewModel: LoginViewModel

    init(viewModel: LoginViewModel) {
        self.viewModel = viewModel
    }

    var body: some View {
        NavigationView {
            ZStack{
                AppBackgroundView()
                    .zIndex(0)
                if self.viewModel.loading {
                    ProgressView()
                        .zIndex(1)
                }
                VStack{
                    LoginText()
```

```
                        .autocapitalization(.none)
                    SecureField("Password", text: self.$viewModel.password)
                        .textFieldStyle(RoundedBorderTextFieldStyle())
                        .padding(.horizontal, /*@START_MENU_TOKEN@*/10/*@END_MENU_TOK
                    Button("Login", action: loginAction)
                    NavigationLink(
                        destination: MenuView(),
                        isActive: self.$viewModel.login,
                        label: {}
                    )
                }
            }
        }
    }

    func loginAction() {
        viewModel.loginNetworkCall()
    }
}

struct LoginView_Previews: PreviewProvider {
    static var previews: some View {
        let nm = MockNetworkManager(session: URLSession.shared)
        LoginView(viewModel: LoginViewModel(networkManager: nm))
    }
}

struct LoginText : View {
    var body: some View {
        return Text("Login")
            .font(.headline)
            .fontWeight(.semibold)
            .padding(.bottom, 20)
    }
}

struct AppBackgroundView: View {
    var body: some View {
        ZStack {
            LinearGradient(gradient: Gradient(colors: [.clear, .secondary]), startPoi
        }
    }
}
```

LoginView.swift hosted with ❤️ by GitHub view raw

```swift
enum API {
    case login
    case posts

    var url: URL {
        var component = URLComponents()
        switch self {
        case .login:
            component.scheme = "https"
            component.host = "reqres.in"
            component.path = path
        case .posts:
            component.scheme = "https"
            component.host = "jsonplaceholder.typicode.com"
            component.path = path
        }
        return component.url!
    }
}

extension API {
    fileprivate var path: String {
        switch self {
        case .login:
            return "/api/login"
        case .posts:
            return "/posts"
        }
    }
}
```

extra.swift hosted with ❤️ by GitHub view raw

Conclusion

Yeah, you might want to start thinking about other ways of Binding these properties in `SwiftUI`. This makes me think about a way of describing that in another article, but still I think this is an interesting article that can help you pick up programming in `SwiftUI` as we go forwards in time.

I hope this article, as with all my articles have helped people out and I hope you enjoy your coding journey.