Swift Enums and Implementing Elegant Login

Sujay Busam
CS 59

June 7, 2016

In a language such as C, enumerated types, or enums, are often viewed essentially as a glorified set of integer mappings that facilitate type safety. For instance, an enum for the cardinal directions in C would have North, South, East, and West mapped to integers from 0 to 3. In Swift, however, enums are much more powerful. Each enumeration case in Swift is a fully-fledged value in itself. In this project, we explore some of the sophistication Swift enums provide and tie all of it together by implementing elegant login cases for a small iOS sample application. The relevant code samples and the Xcode project for the login implementation can be found at https://github.com/SujayBusam/swift-enums

Raw Values

C enums assign related names to a set of integer values, as mentioned above. In Swift, raw values are optional, and when provided, can be of type string, character, integer, or float. All the subsequent code examples will be keeping in the theme of my example login app and will build up to it. Here is an example of raw values as used in Swift.

```
enum LoginProvider: String {
    case Email = "email"
    case Facebook = "facebook"
    case Twitter = "twitter"
    case Google = "google"
}
```

This allows us to do something interesting and initialize an enum from a raw value, as shown below:

```
let provider = LoginProvider.init(rawValue: "email")
```

Associated Values

Associated values are an extension of raw values that provide more sophistication to Swift enums. Swift allows for the storage of other *associated value types* alongside each enum case value. Here is an example using the previous LoginProvider example:

```
enum LoginProvider {
    case Email(String, String)
    case Facebook
    case Twitter
    case Google
}

let provider = LoginProvider.Email("foo@bar.com", "password")
```

This allows for elegant flexibility. Instead of having each case store the same value, we can have different associated values for each case. Furthermore, associated types are not predetermined at compile time like raw values. Instead they can be passed in as parameters when initialized, as shown in the constructor above. This makes sense in our login context since logging in through Facebook, Twitter, or Google through a mobile app would not require an inputted email and password but the Email case would.

We can go even further with associated values. Since associated values can be of *any type*, we can create a struct that is used as an associated value and provides attributes and validation:

```
struct LoginUser {
    let email: String
    let password: String

    func isValid() -> Bool {
        // Implement custom email / password validation here
    }
}
```

Then this LoginUser struct can be used as a parameter for initializing the Email case:

```
enum LoginProvider {
    case Email(LoginUser)
    case Facebook
    case Twitter
    case Google
}

let user = LoginUser(email: "foo@bar.com", password: "password")
let provider = LoginProvider.Email(user)
```

All of this proves to be very useful in simplifying and abstracting code when dealing with login cases. Observe the following example:

```
switch provider {
  case let .Email(user) where user.isValid():
      // Login logic here for valid user
      break
  case let .Email(user) where !user.isValid():
      // Error handling here for invalid user
      break
}
```

This allows for a high degree of specificity in our switch. We can separate the logic based on the associated value of the Email LoginProvider.

Enum Functions

The fact that Swift enums can have functions shows that they are first-class value types in Swift. Consider this function added to our LoginProvider enum:

```
enum LoginProvider {
    case Email(LoginUser)
    case Facebook
    case Twitter
    case Google


    func login() {
        switch self {
        case let .Email(user) where user.isValid():
            // Login
            break
        case let .Facebook:
            // Facebook login logic
            break
        }
        // Other cases ...
    }
}
```

And then to use the function, we can simply do:

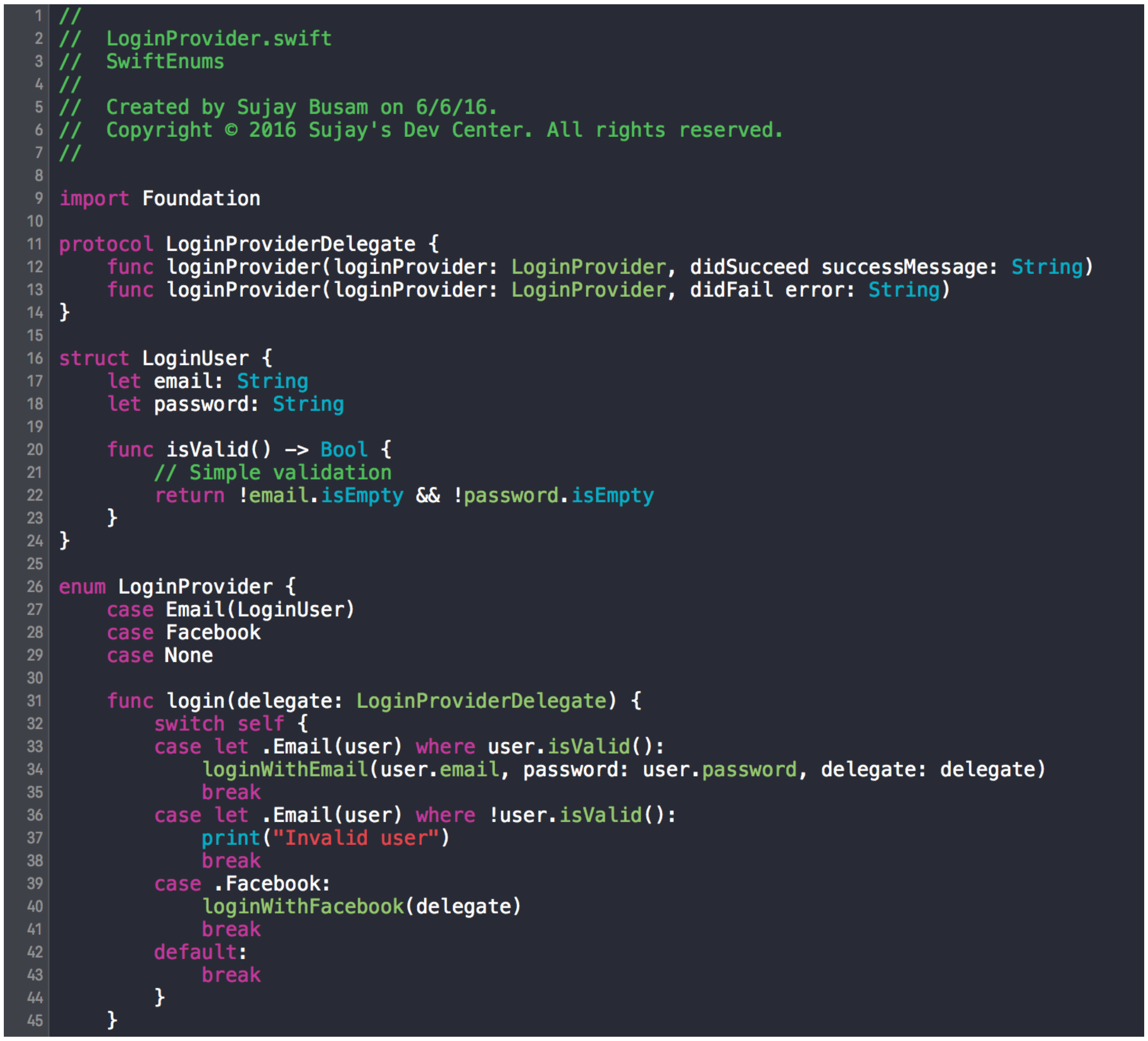
```
let fbProvider = LoginProvider.Facebook
provider.login()
```

Implementing Login With Enums

Now we tie everything together into a small iOS application that demonstrates the elegance and sophistication of Swift enums by demonstrating simplified login. The Xcode project is in the Github repo linked at the beginning of this report. The code is discussed here, and screenshots are provided.

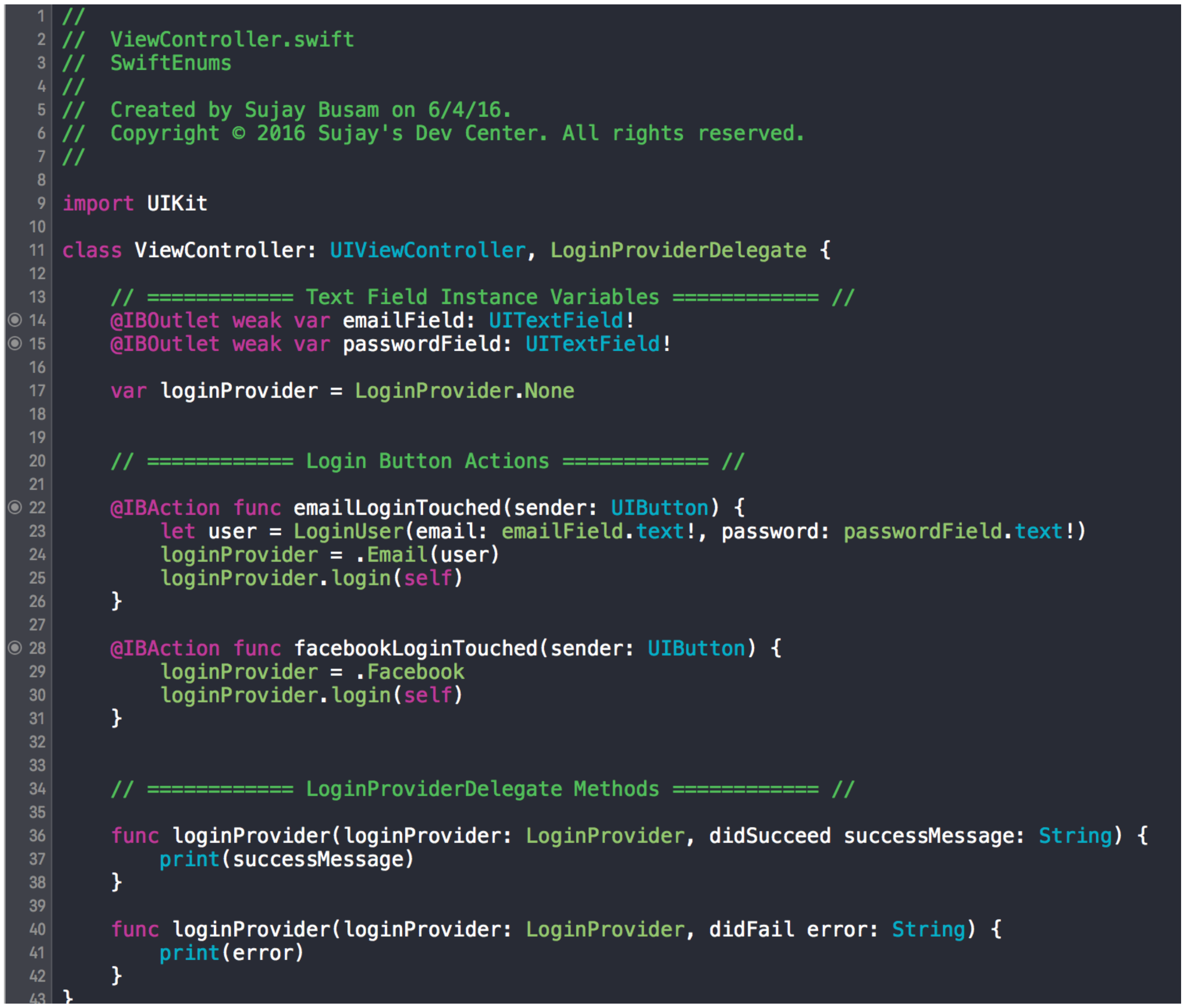
The application demonstrates a login flow that greatly simplifies code and provides abstraction, and can be reused in almost any application. This is the power of Swift enums.

First we start with LoginProvider.swift:



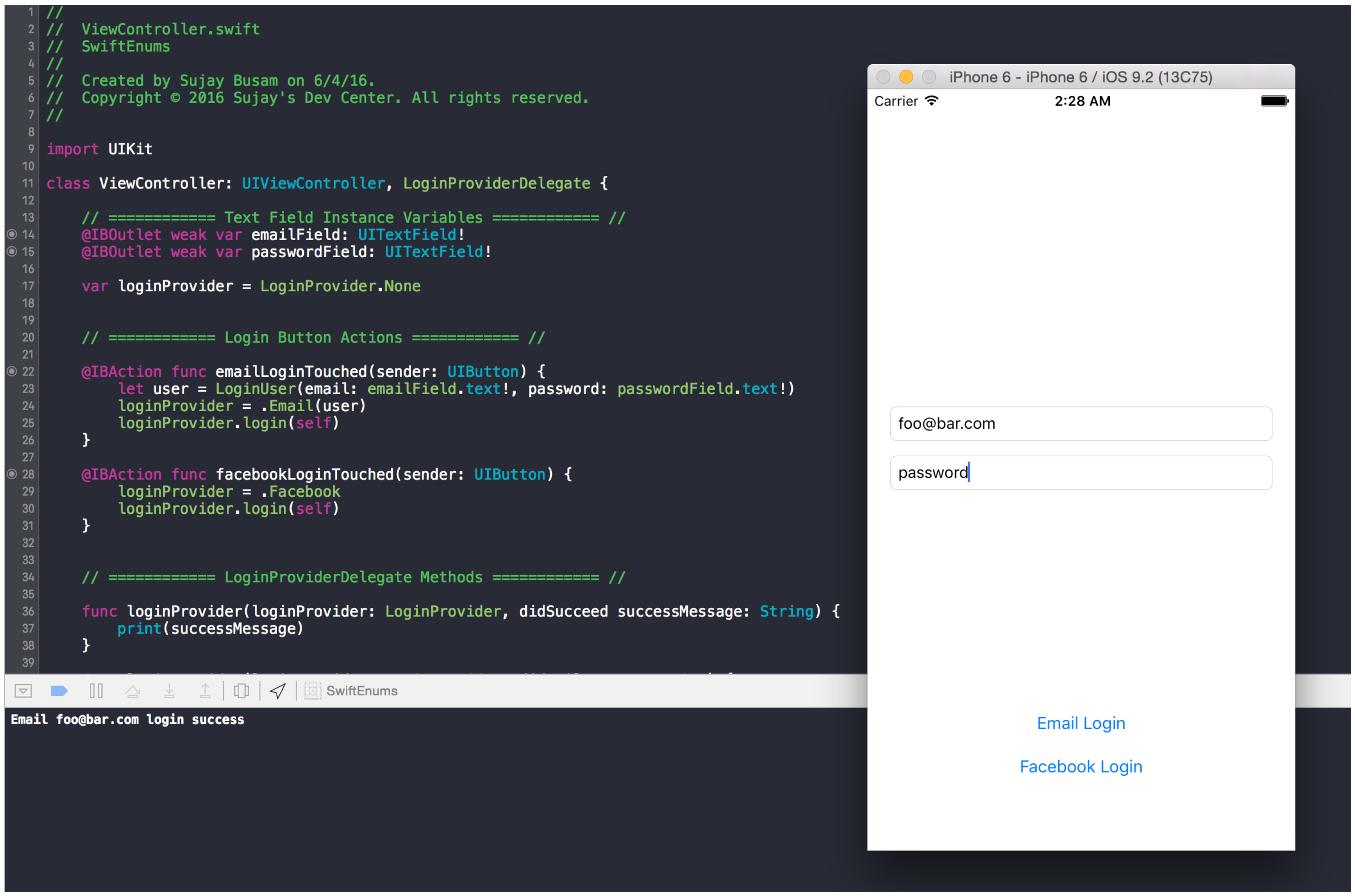
```swift
//
//  LoginProvider.swift
//  SwiftEnums
//
//  Created by Sujay Busam on 6/6/16.
//  Copyright © 2016 Sujay's Dev Center. All rights reserved.
//

import Foundation

protocol LoginProviderDelegate {
    func loginProvider(loginProvider: LoginProvider, didSucceed successMessage: String)
    func loginProvider(loginProvider: LoginProvider, didFail error: String)
}

struct LoginUser {
    let email: String
    let password: String

    func isValid() -> Bool {
        // Simple validation
        return !email.isEmpty && !password.isEmpty
    }
}

enum LoginProvider {
    case Email(LoginUser)
    case Facebook
    case None

    func login(delegate: LoginProviderDelegate) {
        switch self {
        case let .Email(user) where user.isValid():
            loginWithEmail(user.email, password: user.password, delegate: delegate)
            break
        case let .Email(user) where !user.isValid():
            print("Invalid user")
            break
        case .Facebook:
            loginWithFacebook(delegate)
            break
        default:
            break
        }
    }
```

This file implements the Swift "delegate" which acts like a sort of interface, the LoginUser struct mentioned before, and the LoginProvider enum. Within the cases, there are helper functions called that handle the logic for logging in each case. Those can be viewed in the repo. The important aspect is that all of that is abstracted away. The helper functions then use the LoginProviderDelegate to call a success or failure callback. This allows for a clean, simple view controller as shown next in ViewController.swift:



```
//
//  ViewController.swift
//  SwiftEnums
//
//  Created by Sujay Busam on 6/4/16.
//  Copyright © 2016 Sujay's Dev Center. All rights reserved.
//

import UIKit

class ViewController: UIViewController, LoginProviderDelegate {

    // ============ Text Field Instance Variables ============ //
    @IBOutlet weak var emailField: UITextField!
    @IBOutlet weak var passwordField: UITextField!

    var loginProvider = LoginProvider.None


    // ============ Login Button Actions ============ //

    @IBAction func emailLoginTouched(sender: UIButton) {
        let user = LoginUser(email: emailField.text!, password: passwordField.text!)
        loginProvider = .Email(user)
        loginProvider.login(self)
    }

    @IBAction func facebookLoginTouched(sender: UIButton) {
        loginProvider = .Facebook
        loginProvider.login(self)
    }


    // ============ LoginProviderDelegate Methods ============ //

    func loginProvider(loginProvider: LoginProvider, didSucceed successMessage: String) {
        print(successMessage)
    }

    func loginProvider(loginProvider: LoginProvider, didFail error: String) {
        print(error)
    }
}
```

The actual view controller, where the front-end logic resides, is nice and compact because of this implementation. Everything is abstracted to the point where any login involves simply specifying the type and calling the login method on the enum. The delegate methods then take care of printing the success or error message to the console. Here is the application being run with the email login type:



```swift
//
//  ViewController.swift
//  SwiftEnums
//
//  Created by Sujay Busam on 6/4/16.
//  Copyright © 2016 Sujay's Dev Center. All rights reserved.
//

import UIKit

class ViewController: UIViewController, LoginProviderDelegate {

    // ============ Text Field Instance Variables ============ //
    @IBOutlet weak var emailField: UITextField!
    @IBOutlet weak var passwordField: UITextField!

    var loginProvider = LoginProvider.None


    // ============ Login Button Actions ============ //

    @IBAction func emailLoginTouched(sender: UIButton) {
        let user = LoginUser(email: emailField.text!, password: passwordField.text!)
        loginProvider = .Email(user)
        loginProvider.login(self)
    }

    @IBAction func facebookLoginTouched(sender: UIButton) {
        loginProvider = .Facebook
        loginProvider.login(self)
    }


    // ============ LoginProviderDelegate Methods ============ //

    func loginProvider(loginProvider: LoginProvider, didSucceed successMessage: String) {
        print(successMessage)
    }

```

```
Email foo@bar.com login success
```

This is a simple, abstracted pattern for login flow that can be used across all similar applications. It demonstrates just how sophisticated and powerful Swift enums can be.