[iOS] How-to: Integrate a Flow Journey into the Productized App

Introduction

Backbase Digital Sales Onboarding offers a series of Flow journeys and other common Flow components that, put together, allow a user to be onboarded with a new account. In this exercise, we will demonstrate how to incorporate these journeys into the existing productized Retail App so that users will be able to either log in to an existing account (as is in the existing version of the productized app), as well as sign up if they do not have an account.

Pre-requisites

Before setting up your `Podfile`, note that some of the Flow journey dependencies are intended as custom journey and should be imported from the US Onboarding reference app as Development Pods. If you need access to the reference repo, reach out to the Digital Sales team.

In addition, since the Flow app is not productized, you will need to include the necessary app configurations and routers from the reference app into your project.

Since we will be combining both the productized Retail App and the Digital Sales Onboarding journeys, your `Podfile` will need to include references to each of the necessary dependencies:

Podfile

```
# Backbase Retail App dependencies
pod 'RetailUSApp', '2.11.0'
pod 'DesignSystem'

# Digital Sales dependencies
pod 'FlowInteractionClient2'
pod 'FlowInteractionSDK'
pod 'FlowCommon'
pod 'FlowCommonUI'
pod 'FlowDesign'

# Digital Sales journeys
pod 'AddressJourney'
pod 'OTPJourney'
pod 'IDVerificationJourney'
pod 'StepNavigationJourney', '1.4.0'
pod 'ProductSelectionJourney'

pod 'WelcomeJourney',             path: './<path-to-flow-journeys>/Welcome/'
pod 'WalkthroughJourney',         path: './<path-to-flow-journeys>/Walkthrough/'
pod 'AboutYouJourney',            path: './<path-to-flow-journeys>/AboutYou/'
pod 'SSNJourney',                 path: './<path-to-flow-journeys>/SSN/'
pod 'CredentialsJourney',         path: './<path-to-flow-journeys>/Credentials/'
pod 'IdentityVerifiedJourney',    path: './<path-to-flow-journeys>/IdentityVerified/'
pod 'KYCJourney',                 path: './<path-to-flow-journeys>/KYC/'
pod 'DoneJourney',                path: './<path-to-flow-journeys>/Done/'
pod 'InReviewJourney',            path: './<path-to-flow-journeys>/InReview/'
pod 'DeclinedJourney',            path: './<path-to-flow-journeys>/Declined/'
pod 'CoApplicantWelcomeJourney',  path: './<path-to-flow-journeys>/CoApplicantWelcome/'
pod 'CoApplicantJourney',         path: './<path-to-flow-journeys>/CoApplicant'
```

```
pod 'TermsAndConditionsJourney',  path: './<path-to-flow-journeys>/TermsAndConditions'
pod 'LoadingJourney',             path: './<path-to-flow-journeys>/Loading/'
```

Setting up the Landing Journey

To demonstrate how we can provide users with two separate user actions, we will be implementing a custom journey called Landing. This journey will consist of a simple screen that contains two buttons, a `Sign Up` and a `Log In` button.

If the user already has an existing account, the `Log In` button will lead them to the normal Retail App authentication flow through Identity. Otherwise, the `Sign Up` button will route them to register for an account using the Digital Sales Onboarding Flow.

Creating the Landing module

For now, these configurations can be empty `struct`s. We will add the code to implement their functionality later.

Click here to expand...

```swift
import UIKit

struct Landing {
    static func build() -> UIViewController {
        return LandingViewController()
    }

    struct Configuration {
        var router = Landing.Router()
        var strings = Landing.Strings()
        var styles = Landing.Styles()
    }
}

extension Landing {
    struct Router {}
    struct Strings {}
    struct Styles {}
}
```

Setting up the view layer

First, we'll have to create a view layer that will serve as the landing screen. Note that we are binding our `LandingViewModel` in this view controller. For now, you can create an empty view model that contains the necessary stubs but does not perform any functionality yet:

Stubbed LandingViewModel

```swift
import UIKit

class LandingViewModel {
    func didTapLoginButton(navigationController: UINavigationController) {}
    func didTapSignUpButton(navigationController: UINavigationController) {}
}
```

Styling the view layer

When creating your own custom view, you should take advantage of the Backbase Mobile Design System to ensure a consistent look and feel across the app. Here, we will create a custom style, which we will use for our `Sign Up` and `Log In` buttons.

Landing.Styles

```
import BackbaseDesignSystem

extension Landing {
    struct Styles {
        public lazy var primaryButton: Style<Button> = { button in
            DesignSystem.shared.styles.primaryButton(button)
            button.normalBackgroundColor = DesignSystem.shared.colors.surfacePrimary.default
            button.highlightedBackgroundColor = DesignSystem.shared.colors.surfaceSecondary.default
            button.setTitleColor(DesignSystem.shared.colors.primary.default, for: .normal)
        }
    }
}
```

We will also want to utilize the `LocalizedString` API to assign our button strings in the conventional way.

Landing.Strings

```
import RetailJourneyCommon

extension Landing {
    struct Strings {
        var loginButtonTitle = LocalizedString(deferredValue: "Log In")
        var signUpButtonTitle = LocalizedString(deferredValue: "Sign Up")
    }
}
```

Using the pieces above, you can now use the following sample code to implement the `LandingViewController` , which should look as follows:

LandingViewController

```
import UIKit
import BackbaseDesignSystem
import RetailJourneyCommon
import RxSwift
import RxCocoa

class LandingViewController: UIViewController {
    private let viewModel = LandingViewModel()
    private let disposeBag = DisposeBag()
    private var configuration = Landing.Configuration()

    private lazy var signUpButton: Button = {
        let button = Button()
        button.translatesAutoresizingMaskIntoConstraints = false
        button.setTitle(configuration.strings.signUpButtonTitle.value, for: .normal)
        return button
    }()

    private lazy var loginButton: Button = {
        let button = Button()
        button.translatesAutoresizingMaskIntoConstraints = false
        button.setTitle(configuration.strings.loginButtonTitle.value, for: .normal)
        return button
    }()

    override func loadView() {
```

```
        view = UIView()
        view.backgroundColor = DesignSystem.shared.colors.primary.default
        view.addSubview(signUpButton)
        view.addSubview(loginButton)
        setLayoutConstraints()
        applyStyles()
    }

    override func viewDidLoad() {
        super.viewDidLoad()
        bindViewModel()
    }

    private func setLayoutConstraints() {
        signUpButton.centerXAnchor.constraint(equalTo: view.centerXAnchor).isActive = true
        signUpButton.heightAnchor.constraint(equalToConstant: 50).isActive = true
        signUpButton.widthAnchor.constraint(equalToConstant: 200).isActive = true
        signUpButton.centerYAnchor.constraint(equalTo: view.centerYAnchor, constant: 200).isActive = true

        loginButton.centerXAnchor.constraint(equalTo: view.centerXAnchor).isActive = true
        loginButton.heightAnchor.constraint(equalToConstant: 50).isActive = true
        loginButton.widthAnchor.constraint(equalToConstant: 200).isActive = true
        loginButton.topAnchor.constraint(equalTo: signUpButton.bottomAnchor, constant: 20).isActive = true
    }

    private func applyStyles() {
        configuration.styles.primaryButton(signUpButton)
        configuration.styles.primaryButton(loginButton)
    }

    private func bindViewModel() {
        signUpButton.rx.tap.bind { [weak self] in
            guard let self = self,
                  let navigationController = self.navigationController else { return }
            self.viewModel.didTapSignUpButton(navigationController: navigationController)
        }.disposed(by: disposeBag)

        loginButton.rx.tap.bind { [weak self] in
            guard let self = self,
                  let navigationController = self.navigationController else { return }
            self.viewModel.didTapLoginButton(navigationController: navigationController)
        }.disposed(by: disposeBag)
    }
}
```

Inserting the Landing screen into your app

The Retail App checks the user's logged in/biometrics enrolled state to determine which screens to show based on the user's current session. Knowing this, we can override the `.notLoggedInNotEnrolled` state to display the Landing screen when the app is first launched.

Overriding the app state

```
import UIKit
import RetailUSApp
import RetailAppCommon

extension RetailUSAppState {
    var stateViewController: UIViewController {
```

```
        if self == .notLoggedInNotEnrolled {
            return UINavigationController(rootViewController: Landing.build())
        } else {
            let screenBuilder = RetailUSAppRouter.screenBuilder(for: self.entryPoint)
            return RetailUSAppRouter.viewController(for: screenBuilder)
        }
    }
}
```

To take advantage of our custom `stateViewController` property, we can call it in a custom router, which we will use to override at the app level.

This custom router simply takes the state and returns the correct view controller that is supposed to be displayed for that user state. Since we are using our custom `stateViewController`, it will correctly route to the Landing journey if the user is not logged in and not enrolled in biometrics.

Creating your custom app state router

```
import RetailUSApp
import RetailAppCommon
import UIKit

extension RetailUSAppRouter {
    static func didUpdateAppState(state: RetailUSAppState) -> AppWindowUpdater {
        return { window in
            DispatchQueue.main.async {
                UIView.transition(with: window, duration: 0.25, options: .transitionCrossDissolve, animation
                    let oldState = UIView.areAnimationsEnabled
                    UIView.setAnimationsEnabled(false)
                    window.rootViewController = state.stateViewController
                    UIView.setAnimationsEnabled(oldState)
                })
            }
        }
    }
}

```

Finally, we can override the `didUpdateState` router in our `AppDelegate` with our custom router.

Overriding your app-level router

```
override init() {
    super.init { sdk, design in
        ...
        return { appConfig in
            ...
            appConfig.router.didUpdateState = RetailUSAppRouter.didUpdateAppState(state:)
        }
    }
}
```

At this point, your app should now start with the Landing journey. This replaces the default Retail App entry point into the Identity Authentication journey.

Navigating out of the Landing Journey

Preparing to enter Digital Sales Flow

Let's break down what we want to accomplish in the `Sign Up` flow. For the Digital Sales Onboarding flow to be successful, you need to set up the following items:

- Accessing the values in the correct configuration file
- Registering your Flow dependencies (journeys)
- Registering your Authentication client

Setting up the Flow configuration file

Because Digital Sales may run on a separate environment, you will want to have a separate configuration file to contain the references to the correct server URLs and other miscellaneous values that are necessary to start the onboarding process.

Here is a sample configuration file:

flowConfig.json

```
{
    "development": {
        "debugEnable": true
    },
    "backbase": {
        "experience": "onboarding-us-ios",
        "serverURL": <your-server-url>,
        "version": "6.2.7",
        "navigationType": "graph",
        "identity":{
            "baseURL": <your-identity-server-url>,
            "clientId":"mobile-client",
            "realm":"us-customers",
            "applicationKey":"alphaflow"
        }
    },
    "security": {
        "sslPinning": {
            "checkChain": true
        }
    }
}
```

We will also need to create a configuration that pulls the data correctly from this configuration file. As such, you will need to implement a `BackbaseConfiguration` as follows:

Flow BackbaseConfiguration

```
struct BackbaseConfiguration {
    let configPath: String
    let forceDecryption: Bool

    init(
        configPath: String = Variants.configuration["FLOW_CONFIG_PATH"] as? String ?? "assets/backbase/conf/
        forceDecryption: Bool = false
    ) {
        self.configPath = configPath
        self.forceDecryption = forceDecryption
```

```
    }
}
```

With these pieces, along with the code you ported over from the US Onboarding reference app, you can create a class that provides the necessary setup functions to prepare the Digital Sales Flow journeys. Later on, we will call this `SignUp` class to perform the setup before routing into the Onboarding flow.

Sign Up module

```
import Backbase
import BackbaseIdentity
import Resolver

class SignUp {
    private lazy var flowConfig = App.Configuration(backbase: sdk)
    private let backbaseConfig: BackbaseConfiguration
    private let sdk: BackbaseSDK

    init() {
        do {
            backbaseConfig = BackbaseConfiguration()
            sdk = try BackbaseSDK(configuration: backbaseConfig)
        } catch {
            fatalError(error.localizedDescription)
        }
    }

    func setup() {
        registerDependencies()
        registerAuthClient()
    }

    private func registerDependencies() {
        Resolver.register { self.flowConfig }
        Resolver.register { self.flowConfig.design }
        Resolver.register { self.flowConfig.welcome }
        Resolver.register { self.flowConfig.walkthrough }
        Resolver.register { self.flowConfig.stepNavigation }
        Resolver.register { self.flowConfig.otp }
        Resolver.register { self.flowConfig.credentials }
        Resolver.register { self.flowConfig.identityVerified }
        Resolver.register { self.flowConfig.kyc }
        Resolver.register { self.flowConfig.done }
        Resolver.register { self.flowConfig.inReview }
        Resolver.register { self.flowConfig.address }
        Resolver.register { self.flowConfig.aboutYou }
        Resolver.register { self.flowConfig.ssn }
        Resolver.register { self.flowConfig.idVerification }
        Resolver.register { self.flowConfig.declined }
        Resolver.register { self.flowConfig.loading }
        Resolver.register { self.flowConfig.coApplicant }
        Resolver.register { self.flowConfig.coApplicantWelcome }
        Resolver.register { self.flowConfig.productSelection }
        Resolver.register { self.flowConfig.termsAndConditions }
    }

    private func registerAuthClient() {
        let clientSecret = ""
```

```
        let identityAuthClient = BBIDAuthClient(clientSecret: clientSecret)
        do {
            try identityAuthClient.add(BBIDDeviceAuthenticator())
        } catch {
            print(error)
        }
        Backbase.register(authClient: identityAuthClient)
    }
}
```

Creating the router

With the code we've written that should execute for both the `Log In` and `Sign Up` flows, we can now implement the router events for their respective buttons.

Note that in the `onboardUser` event, we are taking advantage of the Flow navigation structure that is defined in the US Onboarding reference app, which you should have copied into your project.

LandingRouter

```
extension Landing {
    struct Router {
        var logIn: (UINavigationController) -> Void = { navigationController in
            let registerScreenBuilder = Register.build()
            navigationController.pushViewController(registerScreenBuilder(navigationController), animated: t
        }

        var onboardUser: (UINavigationController) -> Void = { navigationController in
            SignUp().setup()
            var appConfig = Resolver.resolve(App.Configuration.self)
            appConfig.navigation = navigationController
            let router = FlowStartRouter(rootViewController: navigationController)
            router.showNextStep(withConfig: StepConfiguration(stepName: "loading"))
        }
    }
}
```

Implementing the view model

Let's return to our Landing journey. In the view model, we will take advantage of the router we've created and create functions for each route that is triggered. Return to the stubbed view model you created earlier and call the correct router events in their respective functions.

That is, for the `didTapLoginButton` view model function, call the router event for the code you wish to execute to lead to the Identity Authentication journey. Apply the same process for the `Sign Up` button as well.

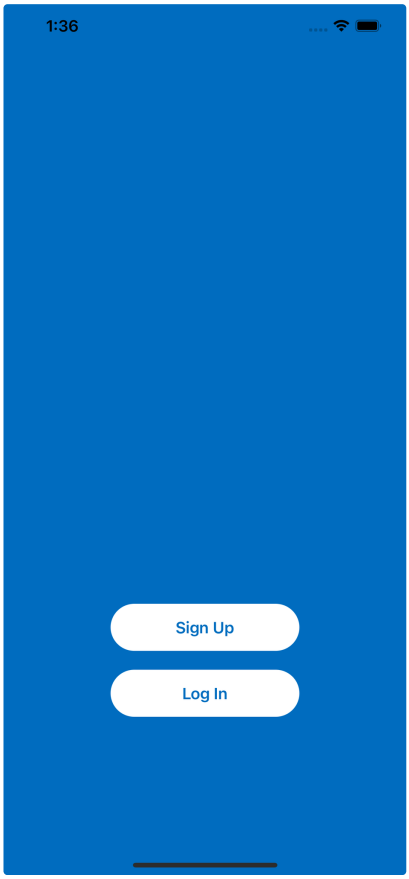
LandingViewModel

```
import UIKit

class LandingViewModel {
    let configuration = Landing.Configuration()

    func didTapLoginButton(navigationController: UINavigationController) {
        configuration.router.logIn(navigationController)
    }

    func didTapSignUpButton(navigationController: UINavigationController) {
        configuration.router.onboardUser(navigationController)
```

```
    }
}
```

Conclusion

Following these implementation steps, you should now have a version of the Retail App that incorporates both the Digital Sales Flow and the Identity Authentication Journey as options for users launching the app. The Landing journey screen should look something like this, with buttons that route to their respective modules.



Landing Journey