[iOS] How to: Build Custom Journey from existing ClientApi

Introduction

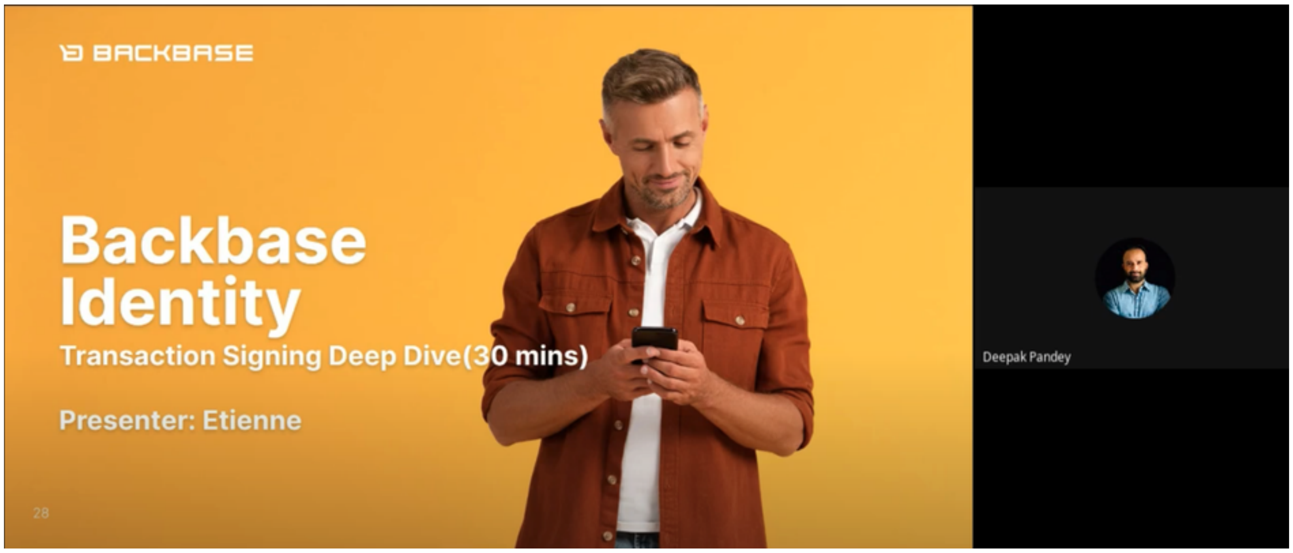
In the Journey architecture, it is important to distinguish each part of a journey into multiple layers to achieve a logical separation of concerns. In this exercise, we will demonstrate how to create a custom journey by building your own view controller and view model while retrieving model data using an existing ClientApi.

For our purposes, we will demonstrate how to use the User Profile Manager client. For more details on this API client, refer to this documentation page.

Understanding the User Profile Manager

You may notice there are two distinct API clients, User Manager and User Profile Manager. It is important to know that the User Profile Manager was designed specifically for bank employees and not end users, hence why it is not integrated in the Retail Banking mobile apps out of the box.

To understand these differences further, refer to this workshop on Backbase Identity:



Backbase Identity Workshop

Prerequisites

Ensure that you have access to the `UserManagerClient2` module. This should belong in the `RetailUSApp` dependency.

To build a custom journey properly, you should also import `DesignSystem` to take advantage of the Backbase Mobile Design System features.

```
pod 'RetailUSApp'
pod 'DesignSystem'
```

Setting up the `UserProfileManagementAPI`

First create a helper class that will provide the instance of the `UserProfileManagementAPIProtocol` based on which components have already been registered.

In this helper class, you are expecting to return an instance of a class that conforms to `UserProfileManagementAPIProtocol`, where the concrete implementation exists as `UserProfileManagementAPI`.

We will use this `UserProfileManagementClientManager` class later to provide the instance of the API client in the use case.

UserProfileManagementClientManager

```swift
final class UserProfileManagementClientManager {
    lazy var client: UserProfileManagementAPIProtocol = {
        guard let serverURL: URL = URL(string: Backbase.configuration().backbase.serverURL) else {
            fatalError("Invalid or no serverURL found in the SDK configuration.")
        }

        let newServerURL = serverURL
            .appendingPathComponent("api")
            .appendingPathComponent("user-manager")

        if let dbsClient = Backbase.registered(client: UserProfileManagementAPI.self),
           let client = dbsClient as? UserProfileManagementAPI {
            return client
        } else if let client = Resolver.optional(UserProfileManagementAPIProtocol.self) {
            return client
        } else {
            let client = UserProfileManagementAPI()
            client.baseURL = newServerURL
            if let dataProvider = Resolver.optional(DBSDataProvider.self) {
                client.dataProvider = dataProvider
                return client
            } else {
                try? Backbase.register(client: client)
                guard let dbsClient = Backbase.registered(client: UserProfileManagementAPI.self),
                      let client = dbsClient as? UserProfileManagementAPI else {
                    fatalError("Failed to retrieve Cards client")
                }
                return client
            }
        }
    }()
}
```

Building the use case

The use case acts as the connecting piece between the API client and the view model. In other words, the use case calls the API clients to supply the relevant data to the view model, which solely interacts with the view.

For this example, we will build a simple function that retrieves the user profile from the API and passes it into a given completion handler.

```swift
import UserManagerClient2

protocol UserProfileUseCase {
    func getUserProfile(completion: @escaping (UserProfile?) -> Void)
}
```

Using this protocol, create a class that implements it. Notice that, when we initialize the use case, we are calling our `UserProfileManagementClientManager` to provide the registered instance of the `UserProfileManagementClient`.

We then call `getOwnUserProfileCall()` from the API client to retrieve data for the `UserProfile` we wish to pass to the view layer.

UserProfileUseCaseImpl

```swift
import Resolver
import Backbase
import UserManagerClient2

class UserProfileUseCaseImpl: UserProfileUseCase {
    private let userProfileApi: UserProfileManagementAPIProtocol?

    convenience init() {
        let userProfileApi = UserProfileManagementClientManager().client
        self.init(userProfileApi: userProfileApi)
    }

    init(userProfileApi: UserProfileManagementAPIProtocol) {
        self.userProfileApi = userProfileApi
    }

    func getUserProfile(completion: @escaping (UserProfile?) -> Void) {
        guard let call = try? userProfileApi?.getOwnUserProfileCall() else { return }
        call.execute { result in
            switch result {
            case let .success(response):
                completion(response.body)
            case let .failure(error):
                Backbase.logError(self, message: "Failed to get user profile: \(error)")
            }
        }
    }
}
```

Building the journey

As with other journeys, you will need to build your own view and controller, which will retrieve data from the view model. Following the convention set forth in the out-of-the-box journeys, this MVVM architecture uses RxSwift to bind the views with their corresponding model data.

Implementing the view model

The view model will contain a reference to our `UserProfileUseCase`. Using this use case, the view model will receive the `UserProfile` object if the call is successful. Whenever this object is received, the view model will create an observable DTO of the relevant data we wish to display and notify any view that has subscribed to it.

```swift
import RxSwift
import RxCocoa

class UserProfileViewModel {
    private let userProfileUseCase: UserProfileUseCase = UserProfileUseCaseImpl()

    var userProfile: BehaviorRelay<GenericUserProfile?> = .init(value: nil)

    func getUserProfile() {
        userProfileUseCase.getUserProfile(completion: { [weak self] userProfile in
            let genericProfile: GenericUserProfile = .init(fullName: userProfile?.fullName ?? "N/A",
                                                           phoneAddress: userProfile?.phoneAddresses?.first?.num
                                                           electronicAddress: userProfile?.electronicAddresses?.
```

```
                                                            postalAddress: userProfile?.postalAddresses?.first?.a
            self?.userProfile.accept(genericProfile)
        })
    }
}
```

Implementing the view controller

Inside the view controller, initialize the `UserProfileViewModel`.

Then create a simple `UILabel` and, using RxSwift, bind it to the user profile observable from the view model. We can assign the text of this label to display the fields of the user profile retrieved from the API.

UserProfileViewController

```swift
import UIKit
import RxSwift

class UserProfileViewController: UIViewController {
    private var viewModel: UserProfileViewModel = .init()

    private let disposeBag: DisposeBag = DisposeBag()

    private lazy var profileLabel: UILabel = {
        var label = UILabel()
        label.translatesAutoresizingMaskIntoConstraints = false
        label.numberOfLines = 0
        return label
    }()

    override func loadView() {
        super.loadView()
        view.addSubview(profileLabel)
        profileLabel.centerXAnchor.constraint(equalTo: view.centerXAnchor).isActive = true
        profileLabel.centerYAnchor.constraint(equalTo: view.centerYAnchor).isActive = true
        profileLabel.heightAnchor.constraint(greaterThanOrEqualToConstant: 100).isActive = true
        profileLabel.widthAnchor.constraint(greaterThanOrEqualToConstant: 100).isActive = true
    }

    override func viewDidLoad() {
        super.viewDidLoad()
        viewModel.getUserProfile()
        viewModel.userProfile.bind(onNext: { userProfile in
            DispatchQueue.main.async { [weak self] in
                guard let userProfile = userProfile else { return }
                self?.profileLabel.text = """
                    Full Name: \(userProfile.fullName)
                    Phone Address: \(userProfile.phoneAddress)
                    Electronic Address: \(userProfile.electronicAddress)
                    Postal Address: \(userProfile.postalAddress)
                """
            }
        }).disposed(by: disposeBag)
    }
}
```

With the view controller, view model, and use case implemented, you now have all the pieces that make up a journey. Since our custom use case references and calls the generated `UserProfileManagementClient`, this describes the relationship between our custom journey and an existing API client.

Providing the build function

To access this journey, provide an entry point that conforms to the structure of other out-of-the-box journeys by implementing a static function as follows:

```swift
import UIKit

struct UserProfileJourney {
    static func build(navigationController: UINavigationController) -> UIViewController {
        return UserProfileViewController()
    }
}
```

Using this build function, we can call it from any access point, either from the bottom tab menu or from the More Menu. For our purposes, we shall trigger this navigation from the More Menu.

Configuring the More Menu

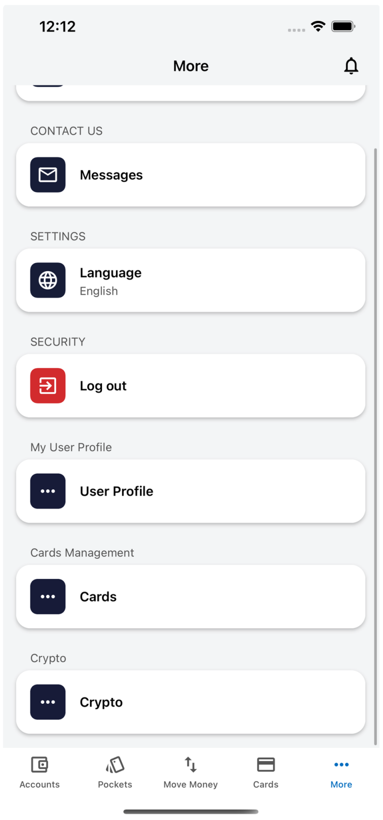
```swift
import RetailMoreJourney
import BackbaseDesignSystem
import RetailJourneyCommon
import Backbase
import Resolver

extension More {
    static func configure(_ configuration: inout More.Configuration) {
        let userProfileMenuItem = More.MenuItem(title: .init(value: "User Profile"),
                                    icon: nil,
                                    action: userProfileMenuItemAction(navigationController:))

        let userProfileSection = More.MenuSection(title: .init(value: "My User Profile"),
                                               items: [userProfileMenuItem])

        let deferredSections = configuration.menu.deferredSections
        configuration.menu.deferredSections = {
            guard let deferredSections = deferredSections else { return [] }
            var sections = deferredSections()
            sections.append(userProfileSection)
            return sections
        }
    }

    private static func userProfileMenuItemAction(navigationController: UINavigationController) {
        let viewController = UserProfileJourney.build(navigationController: navigationController)
        viewController.title = LocalizedString(deferredValue: "User Profile").value
        navigationController.pushViewController(viewController, animated: true)
    }
}
```



Accessing your custom journey from the More Menu

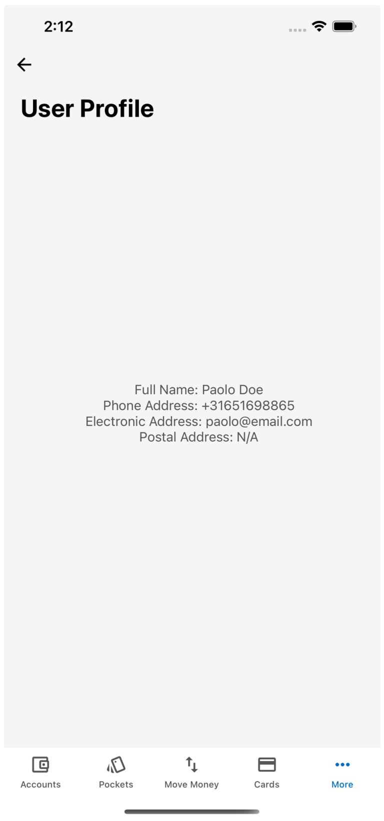
Utilizing the Mobile Design System

When implementing a custom journey, you should take advantage of Backbase's Mobile Design System. This allows you to make the journey look and feel consistent across the other screens of the app. In addition, it allows you to reuse design components without having to rewrite those configurations.

To do this, let's revisit our view controller. Import `BackbaseDesignSystem`. This library will give you access to all the shared styling components that have already been defined.

In the initialization of our `profileLabel`, we will use an existing style called `summaryStackViewDefaultLabel` on line 10 and apply it to the label before it is returned.

```swift
 1  import BackbaseDesignSystem
 2  
 3  class UserProfileViewController: UIViewController {
 4      ...
 5  
 6      private lazy var profileLabel: UILabel = {
 7          var label = UILabel()
 8          label.translatesAutoresizingMaskIntoConstraints = false
 9          label.numberOfLines = 0
10          DesignSystem.shared.styles.summaryStackViewDefaultLabel(label)
11          return label
12      }()
13  
14      ...
15  }
```



Full Name: Paolo Doe
Phone Address: +31651698865
Electronic Address: paolo@email.com
Postal Address: N/A

Stylized label using the Mobile Design System