How to: Integrate a Native Widget into a Journey Based Application

Introduction

With Backbase moving to Journey based Productized apps, the native widgets are no longer introducing new features or regular upgrades by the product teams. To help our existing customers who are starting to migrate from Widget based apps to Journeys, we created this how-to guide with working sample code to explain how a developer could reuse a widget from the customer app (Custom/OOTB) into Journey based application as it is.

Pre-requisites

Before starting, make sure to have the following components:

1. US/Universal Productized application
2. Custom/OOTB Widget and Client
3. Widget views including (Layout, strings & assets)
4. Any other components that the widget/client relies on (EX: DTOs, helper classes, utils, etc)

Integration

US/Universal productized apps by default come with the setup of repo.backbase.com , you only need to make sure you have `mvnUser` `mvnPass` defined in **gradle.properties** (for Android) or **.netrc** (for iOS)

Android

In project-level **build.gradle**

```
repositories {
    jcenter()
    maven {
       url 'https://repo.backbase.com/android3'
       credentials {
          username = "yourUsername"
          password = "yourPassword"
       }
    }
}
```

In **app/build.gradle** , Start by adding the dependencies of the Widget/Client, in case OOTB is being used, and add any other related dependencies.

```
    implementation "com.backbase.android.widgets:card-management-widget:9.0.2@aar"
    implementation "com.backbase.android.clients:card-management-client:9.0.2@aar"

    implementation "com.backbase.android.common:common-utils:9.0.2@aar"
    implementation "com.backbase.android.common:common-ui:4.3.0@aar"
```

iOS

```
iOS - .netrc
machine repo.backbase.com
login <USERNAME>
password <PASSWORD>
```

In your **Podfile**, add the following line for `CardManagementWidget`. This will grant you access to both the `CardManagementWidget` and `CardManagementClient` modules:

```
pod 'CardManagementWidget'
```

Files & Assets

It's important to copy all the classes and assets that are being used by the widget in order to work

1. WidgetViews
2. Helper classes
3. DataProviders
4. Utils
5. DTOs
6. Layout files
7. other assets

Widget/Client Registration

Android

In the Application class, under **onInitialized(),** register the relevant Renderers and Clients along with DataProviders.

```
override suspend fun onInitialized() {
    BBRenderer.registerRenderer(CardManagementWidget::class.java)
    Backbase.requireInstance().registerClient(CardManagementClient(
        CustomAssetsFileDBSDataProvider(this),URI("backbase/cards-presentation-service"),GsonResponseBodyParser()
    ))
}
```

iOS

For our purposes, we'll be using a local version of the `CardManagementClient` so we can use the local model instead of querying from a server. To do so, create an extension of the `CardManagementClient`.

Implementing a local Client

```
import Foundation
import CardManagementClient

extension CardManagementClient {
    convenience init(dataProvider: DataProvider) throws {
        self.init()

        self.baseURL = dataProvider.baseUrl ?? URL(string: "http://localhost")!
        self.dataProvider = dataProvider
    }

    /// Use this method to create a local client
    static var local: CardManagementClient {
        let dataProvider = DataProvider(providerType: .local(path: "assets/backbase/localdata", launchArgumentNa
        do {
            return try CardManagementClient(dataProvider: dataProvider)
        } catch (let error) {
            fatalError(error.localizedDescription)
        }
    }
}
```

Registering the Renderers and Clients

Now that we have created our own `CardManagementClient` implementation, inside your `AppDelegate` file, register the relevant Renderers and Clients.

```
private func registerWidgets() {
    do {
        try BBRendererFactory.register(renderer: CardManagementWidget.self)
    } catch {
        Backbase.logWarning(self, message: "Unable to register widget \(String(describing: CardManagementWidget.s
    }
}
```

```
private func registerClients() {
    try? Backbase.register(client: CardManagementClient.local)
}
```

With these two functions set up, we can call them in the `init()` method of the `AppDelegate`.

```
override init() {
    super.init { sdk, design in
        return { appConfig in
            ...
    }
    registerWidgets()
    registerClients()
}
```

Model

Setting up the `config.json`

In your `config.json` file, include a value for `backbase.localModelPath` that points to a local JSON file for your model:

```
"backbase": {
    "localModelPath": "$(contextRoot)/localModel.json",
},
```

Android

Make sure to add the widgets to the model on CXP (in this example we are using the localModel.json)

In the Activity class, implement `ModelListener<Model>` interface

```
open class MyAppActivity : UsAppActivity(), ModelListener<Model>{
```

```
    override fun onModelReady(model: Model) {
        Log.d("TAG Activity", "onModelReady: Model Loaded")
    }

    override fun onError(e: Response) {
        BBLogger.error("TAG activity", "Model loaded with error: $error")
    }
```

Loading the model

In the onCreate method, load the model either from the server or locally

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    backbase.getModel(this, ModelSource.LOCAL)
}
```

iOS

You can load the model by building an implementation of the `ModelDelegate` protocol.

```
class RetailAppModelDelegate: NSObject, ModelDelegate {
    static let shared = RetailAppModelDelegate()

    func loadModel() {
        DispatchQueue.main.async { [weak self] in
            guard let strongSelf = self else { return }
            Backbase.model(strongSelf, order: [.file])
        }
    }

    func modelDidLoad(_ model: Model) {
        Backbase.logInfo(self, message: "Model loaded")
    }

    func modelDidFail(with error: Error) {
```

```
        Backbase.logWarning(self, message: "Failed to load model: \(error.localizedDescription)")
    }
}
```

Once you have this class set up, we can use it in our `AppDelegate` to load the model when we have successfully logged into the app using the app router from the Retail App.

```
override init() {
    super.init { sdk, design in
        return { appConfig in
            if let didUpdateState = appConfig.router.didUpdateState {
                appConfig.router.didUpdateState = { appState in
                    if appState == .loggedInEnrolled {
                        RetailAppModelDelegate.shared.loadModel()
                    }
                    return didUpdateState(appState)
                }
            }
        }
    }
}
```

Navigation setup

Android

In Activity class, implement `NavigationListener` interface to handle the navigation events

Note: If your project is using the Navigation System, make sure to replace the NavType with the relevant ones, check the documentation.

```
override fun onNavigationEvent(navigationEvent: NavigationEvent) {
    val pageId: String = navigationEvent.targetPageId
    if (currentPageId != pageId){
        currentPageId = pageId
    } else {
        return
    }
    when {
        BBAuthenticatorPresenter.isAuthenticatorShowEvent(navigationEvent) || BBAuthenticatorPresenter.isAuthent
            return
        }
        navigationEvent.relationship == NavType.PARENT -> {
            supportFragmentManager.popBackStack()
        }
        navigationEvent.relationship == NavType.ROOT || navigationEvent.relationship == NavType.ROOT_ANCESTOR ->
                supportFragmentManager.popBackStack(null ,POP_BACK_STACK_INCLUSIVE)
                replaceFragment(pageId)
            }
        else -> {
            replaceFragment(pageId)
        }
    }
}
```

replaceFragment() is used to do the actual navigation by replacing the current fragment in the stack to the target page using the PageFragment as the base.

```
fun replaceFragment(pageId: String) {
    val pageFragment: PageFragment = PageFragment.newInstance(pageId)
    val fragmentTransaction: FragmentTransaction = supportFragmentManager.beginTransaction()
    fragmentTransaction.addToBackStack(pageFragment.getTag())
    fragmentTransaction.replace(R.id.appContent, pageFragment).commit()
}
```

iOS

First, let's create our own `NavigationListener` class that can listen for any incoming `Notifications` and decide on the correct navigation action to take based on the JSON payload.

Implementing a `NavigationListener`

RetailAppNavigationListener

```
import Foundation
import Backbase

class RetailAppNavigationListener {
    static let shared = RetailAppNavigationListener()

    @objc
    func receiveNavigationNotification(_ notification: NSNotification) {
        guard
            let origin = notification.userInfo?["origin"] as? String,
            let target = notification.userInfo?["target"] as? String,
            let relationship = notification.userInfo?["relationship"] as? BBNavigationFlowRelationship
            else { return }

        let payload = notification.userInfo?["payload"] as? NavigationPayload

        switch relationship {
        case .root:
            handleRootNavigation(source: origin, target: target, payload: payload)
        case .child:
            handleChildNavigation(source: origin, target: target, payload: payload)
        case .parent:
            handleParentNavigation(source: origin, target: target, payload: payload)
        case .rootAncestor:
            handleRootAncestorNavigation(source: origin, target: target, payload: payload)
        default:
            break
        }
    }

    // MARK: - Navigation methods

    private func handleRootNavigation(source: String, target: String, payload: NavigationPayload?) {
        guard
            let model = Backbase.currentModel(),
            let renderable = model.item(byId: target)
            else { return }

        let controller = RenderingViewController(renderable: renderable, payload: payload)
        if let tabBarController = UIApplication.shared.keyWindow?.rootViewController as? UITabBarController,
           let navigationController = tabBarController.selectedViewController as? UINavigationController {
```

```
            navigationController.pushViewController(controller, animated: true)
        } else {
            UIApplication.shared.keyWindow?.rootViewController = UINavigationController(rootViewController:
        }
    }

    private func handleChildNavigation(source: String, target: String, payload: NavigationPayload?) {
        guard
            let tabBarController = UIApplication.shared.keyWindow?.rootViewController as? UITabBarController
            let navigationController = tabBarController.selectedViewController as? UINavigationController,
            let model = Backbase.currentModel(),
            let renderable = model.item(byId: target)
            else { return }

        navigationController.pushViewController(RenderingViewController(renderable: renderable, payload: pay
    }

    private func handleParentNavigation(source: String, target: String, payload: NavigationPayload?) {
        guard
            let tabBarController = UIApplication.shared.keyWindow?.rootViewController as? UITabBarController
            let navigationController = tabBarController.selectedViewController as? UINavigationController
            else { return }

        navigationController.popViewController(animated: true)
    }

    private func handleRootAncestorNavigation(source: String, target: String, payload: NavigationPayload?) {
        guard
            let tabBarController = UIApplication.shared.keyWindow?.rootViewController as? UITabBarController
            let navigationController = tabBarController.selectedViewController as? UINavigationController
            else { return }

        navigationController.popToRootViewController(animated: true)
    }
}

typealias NavigationPayload = [String: Any]
```

Registering the `NavigationListener`

We can then instantiate and register this navigation listener with the Mobile SDK inside the `init()` method of our `AppDelegate`.

```
private func registerNavigationListener() {
    let selector = #selector(RetailAppNavigationListener.receiveNavigationNotification(_:))
    Backbase.register(navigationEventListener: RetailAppNavigationListener.shared, selector: selector)
}
```

```
override init() {
    super.init { sdk, design in
        return { appConfig in
            ...
        }
    }
    registerNavigationListener()
}
```

Now the app is ready for any navigation events that may be triggered.

Triggering Navigation

The best place to trigger the navigation in our app would be from the MoreMenu as it provides us with the flexibility to add any type of action, in our case it would be NavigationEvent.

To do that, we need to configure the MoreMenu to add a section to the default MoreMenu items.

Android

```
override fun createApplicationConfiguration() = UsApplicationConfiguration {
    //Other configurations

    journeyConfigurations = UsJourneyConfigurations {
        moreConfigurationDefinition = {
            DefaultUsMoreConfiguration {
                sections = mutableListOf<MenuSection>().apply {
                    add(DefaultUsManageMoreMenuSection())
                    add(DefaultUsContactUsMoreMenuSection())
                    add(DefaultUsSettingsMoreMenuSection())
                    add(DefaultUsSecurityMoreMenuSection())
                    add(
                        MenuSection(
                            items = listOf(cardsItem),
                            title = DeferredText.Constant("Cards")
                        )
                    )
                }
            }
        }
    }
}

private val cardsItem = MenuItem(
    title = DeferredText.Constant("Cards Management"),
    subtitle = DeferredText.Constant("cards"),
    icon = DeferredDrawable.Resource(R.drawable.backbase_ic_add_box),
) {
    Backbase.getInstance()!!.publishEvent("bb.action.cards.open", JSONObject())
    OnActionComplete.DoNothing
}
```

Note: make sure the event name used for navigation is defined on the target page in the Model

iOS

We can extend the More Menu configuration with our own custom function that creates a new section for this navigation event as well as the publish action that will trigger this event.

```
extension More {
    static func configure(_ configuration: inout More.Configuration) {
        let cardMenuItem = More.MenuItem(title: .init(value: "Cards"),
                                         icon: nil,
                                         action: cardMenuItemAction(navigationController:))
        let cardsSection = More.MenuSection(title: .init(value: "Cards Management"),
```

```
                                        items: [cardMenuItem])

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

    private static func cardMenuItemAction(navigationController: UINavigationController) {
        if let renderable: Renderable = Backbase.currentModel()?.app() {
            Backbase.publish(event: "bb.action.cards.open",
                             object: renderable,
                             payload: nil)
        }
    }
}
```

With this extension, we can call it inside the `init()` method of our `AppDelegate` to add this configuration to our More Menu.

```
override init() {
    super.init { sdk, design in
        return { appConfig in
            More.configure(&appConfig.more)
        }
    }
}
```