Westpac Mobile Assessment - Native iOS Currency Converter

19/02/2020
Bhuman Soni

Overview

This document contains the requirements specification to build a mobile currency converter app. While the app can be built for either Android or iOS, this document focuses on the technical requirements for a native iOS app. The rest of this document is structured as follows

- iOS requirements as mentioned in the assessment document
- Define the user story for the app
- A description of what the app will do
- Technical requirements i.e. what libraries or frameworks can be used to achieve this
- UI design samples for the minimal UI
- Some backend architecture design
    - Decisions to adopt this style
- Lastly, Questions i.e. if any aspect of building this is not clear

Technical task

The task we would like you to complete is to create an app that makes a call for international currency data (JSON) and use it to create a currency converter app that converts international currencies into Australian Dollars. Refer to Apple's Human Interface Guidelines or Android's material design as a resource to provide a good customer experience.

User story (Agile)

I am an Australian travelling to Europe and while there are many currency converter apps, I want one that converts international money to Australian dollars (AUD).

Understanding the user story

This needs to be a simple app with minimal interface and a clear call to action. It should be something that the user launches and instantly gets what they want i.e. currency conversion to AUD. At its core, It could be a simple app with four UI elements. An interface (dropdown) to select the international currency, textbox to add the value to be converted, an uneditable value

to show converted AUD and a button to trigger the conversion. An additional feature where the user can backtrack through their currency conversions. Given the volatility of exchange rates, the user may or may not need to be informed when the exchange rates were last updated. The ability to search through available currencies can be a better user experience. It would be added to this app should there be a time and budget for it.

Technical Task Requirements

As handed down in the assessment document

The app must be written in Swift or Kotlin
The app must be written using Xcode 9+ for iOS or Android Studio for Android.
Use of suitable design patterns, and a strict separation of concerns when developing a mobile application
Create a library containing all functionality that could be reused by 3rd party developers in any application that requires exchange rate calculations. 3rd parties using this library should not be tightly coupled to any specific technology and should be able to consume the library in any way they choose
Code should be commented sufficiently to allow auto generation of library API documentation
Use all the data the API provided

- *Asynchronous development principals when retrieving and displaying data originating from network calls*

UI interaction and data binding principals

- *Sound management of User Interface*
- *The application should be able to be re-branded (colours, fonts, assets) - 2 distinct branded targets/variants should be included in the project*
- *The app should be built with a universal UI.*

Correct use of the application life cycle, management of the UI thread
Incremental Commits of code and proper use/understanding of gitflow

- *Quick Overview - http://nvie.com/posts/a-successful-git-branching-model/*
- *In-depth Tutorial - https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow*

Unit tests/mocks to demonstrate the code is testable

Requirements breakdown

This aims to look at the native iOS app requirements, as examines the underlying technology used to achieve it.

User Interface

At the highest level would be the ***ConverterUIViewController***, which will house the four (or five) interface elements required for the user to convert foreign currency.

Then, Currency selection: ***currencySelectorPV*** (UIPickerView) that shows all the available currency conversion options. The currency options will contain the currency name with the currency symbol in brackets.

Currency input: ***foreignInputTF*** (UITextField) where the user can enter values and a placeholder that says “foreign currency value”. It can delegate events to ***ConverterUIViewContoller.***

Currency Output: ***auOutputTF*** (UITextField) An uneditable that can display the converted output and the placeholder that says “aud$”. It can delegate events to ***ConverterUIViewController***

Convert: ***triggerBtn*** (UIButton) whos .touchUpInside event performs the currency conversion and updates the ***auOutputTF***.

A ***inProgIA*** (UIActivityIndicatorView) over the ***auOutputTF*** view as some visual feedback to the user while waiting for the currency to be converted.

Layout

The elements in the ***ConverterUIViewController*** should be arranged via the AutoLayout constraints. The ***currencySelectorPV*** should be left aligned and managed the rest of the elements placement via ***UIStackView.***

Nice to have requirements

Haptic feedback, trigger medium (UIImpactFeedbackGenerator.FeedbackStyle.medium) on pressing the ***triggerBtn*** and (UIImpactFeedbackGenerator.FeedbackStyle.heavy) on conversion completion.

A ***previousConversionBtn*** (UIButton), to backtrack (rewind) through their previous conversions.

Monitor the ***textFieldDidEndEditing*** method of the UITextField delegate for currencyInput and convert the value automatically without the ***triggerBtn*** button interaction.

A ***exRateLastUpdatedLbl*** (UILabel) that's displayed below the text field, which shows when the exchange rates were last updated.

An extra view with a UITableViewController that shows the conversions they did over the course of using the app.

The text in the app

All written text in the app that comes from backend code would come from ***Localizable.strings*** file and be used with ***NSLocalizedString(key, comment)***.

Backend

The backend involves all the code that would be separated from the UI code and when necessary, the *DispatchQueue* will be used when updating UI elements. The aim is to keep the code as loosely coupled as possible, in order to ensure it's reusable across apps, as outlined in the following posts,
https://mydaytodo.com/writing-maintainable-ios-swift-code/
https://mydaytodo.com/decouple-ios-code-with-protocol/

The purpose of those posts is, our aim is to present the currency conversion information to the user. The user doesn't need to show where the app gets its data from. Hence, the backend will be built such that, all the user facing UIViewController needs to know is to call class XYZ to get data in native Swift data types. It doesn't care how or where class XYZ gets the data from. This means we can completely re-write class XYZ without affecting the UIViewContoller as long as the UIViewController gets the data it needs.

Constants.swift: This is a class that would store all static values e.g.

1. URL text(String) used to make a network call to fetch some API data
2. CGFloat values used to round the UI elements

API.swift: class to fetch the currency data from an external API. Pre-Process the data and convert it to a format used to populate the ***currencySelectorPV*** (UIPickerView) in the UIViewController. It can return the data via either a closure or a protocol. A ***results*** Protocol would mean, the ***ConverterUIViewController*** can implement it, such that the API class can notify it every time it fetches and updates the values.
Potential libraries to use: SwiftyJSON and AlmoFire

Additional requirement: Save and fetch the external API url from Firebase/Firestore. Doing this means, we can update the external URL without having to release an update for the app.

Backup.swift: In the likely scenario that the user is in a place where there's no network connection, the app can benefit from some offline functionality. The first time, the app fetches the currency data from an external API, it can create a local backup of the data using JSONEncoder and read it via JSONDecoder. Therefore, in case the app finds itself in a place with no internet connection, it can resort to the backup data. However, adding this functionality mandates adding the ***exRateLastUpdatedLbl*** to the UI to notify the user, they are potentially looking at out of date exchange rate value.

ErrorNotifications.swift: This class contains a set of static methods that can prompt the user in case of an error. For starters, it will use the UIAlertController with the strings and messages coming from

FeedbackHelper.swift: This would be a very simple class that would contain a static method that can be called to trigger haptic feedback when necessary.

Questions

1. Are any of the listed "nice to have" features required?
2. Should we give users the ability to check (backtrack) their last conversion?
3. Can we notify users of errors via dialogs?
4. I haven't worked with currency data, so not sure how to use the following data?


```
 "buyTT":"0.7042",
"sellTT":"0.6367",
"buyTC":"0.7065",
"buyNotes":"0.7065",
"sellNotes":"0.6367",
"SpotRate_Date_Fmt":"20200217",
"effectiveDate_Fmt":"20200217T081006,26+11:00",
"updateDate_Fmt":"20200217T081006,26+11:00",
```