Simon Tsai

Summary

Approach

At a high level, I wanted to implement an app that was within the constraints of the random-user API. Given that the API supported fetching multiple users containing contact information, I decided to go with a contacts-based app, which streamlines the process for reaching out to your contacts. My overall approach was to test drive the mechanism for user fetching and parsing before working on the flow and UI of the app.

Features

On the first screen, the user can see a list of a hundred contacts. Each row displays a contact's avatar and their name. Tapping on a contact transitions to the second screen, which again displays the contact's avatar in addition to their address, cell phone number, and email address. Tapping on the phone number displays a prompt with an option to call the contact. Similarly, tapping on the email address takes the user to a mail composer in the Mail app. The phone number and email address can be disabled if they are invalid or if the system is unable to open them. I also added a Today extension (widget) that displays up to three contacts with birthdays for today.

Additional Features (with more time)

If I had more time, I would add a few more features to the app such as search (local) and filtering. I would also consider pagination, but that would involve a tradeoff between searching and memory because the random-user API doesn't support keyphrase querying. If users are more inclined to search, it would be better to search across a larger number of random contacts, implying that the app should display many results to the user in one go. If users don't search very frequently, it may be better to fetch results in small batches and paginate. Another feature I would add is a transition from the widget to the main app after tapping on a birthday contact.

I'd estimate around two or three hours to include these additional features.

Robust (with more time)

There are several ways I could make the app more robust. Currently, I handle errors by displaying an alert view with the option to try again. A more graceful process would be to present an error screen with an explanation for failures and a way to recover. Another area I could improve is the parsing of the response payload for users. I wrote the parsing logic with the assumption nothing would change since I am always using versions 1.1 of the random-user API, which the documentation asked to specify to prevent possible breaking changes in any of their future updates.

However, for safety, I would handle any possible changes in the API by throwing an parsing error and updating the UI for a given field.

To further ensure app quality, I would distribute it on TestFlight, so I can get a real sense of the app's stability through actual users. Some UI testing is also necessary to verify all the screens' layouts are in working order. Snapshot testing and Xcode's UI testing frameworks would be two approaches to consider. Finally, I'd focus on continuous integration, which is something I've wanted to integrate into my workflow. Personally, I haven't had much experience in this area, but I would definitely spend the time to learn more about the CI process and maintain stable builds with each iteration of the app.