Swipe Without Age and Gender - An React Native Application

Aden Kenny - 300334300

November 13, 2018

1 Architectural Design

Our applications uses a fairly standard three tier architecture. It has a presentation layer of TypeScript, a business layer of TypeScript, and finally a data layer of TypeScript. There is a strict communication protocol between the three different layers. The data layer can only interact with the business layer, the presentation layer can only interact with the business layer, and the business layer can interact with both the data layer and the business layer. This means that all interactions with external components must go through an intermediary (the business layer) before interacting with the with the user (the presentation layer).

The presentation layer is seen and interacted with by the users. This layer handles all user interaction, and the users should only ever see this layer. This layer, as we're using React Native, is done entirely in TypeScript with TSX. The structure and styling are implemented with TSX and React Native stylesheets respectively. It communicates only with the presentation layer. When communicating with the presentation layer it either sends information about options the user has selected or actions they have carried out. When receiving data from the business layer, the data is related to options or queries the user has made, and this data is then used to update the view the user sees and provide interactivity between the user and the app.

The business layer acts as a sort of middleman between the presentation and data layers. The business layer exists exclusively of TypeScript code with functions to parse data from the data layer, functions to update the presentation layer, and functions to modify the local state of the system. All interaction between the presentation layer and data layer must go through the business layer. The overall design means that when the user performs an action such as viewing another users profile, a message will be sent from the presentation layer to the data layer. The business layer will then update, and send a message to the presentation layer to update with some sort of loading behaviour, and a message is then sent to the data layer to fetch the required data to fully update the presentation layer. This data from the data layer is then sent to the business layer where it is parsed for use, and then finally a message is sent to the presentation layer to update to the final state required for the user action.

The data layer sends data to and from the external Firebase services (see the Firebase Integration section below). When fetching data, a message is sent from the business layer to the data layer telling it to fetch data from the external services, this data is then sent to the business layer. The reverse of this scenario is taken when sending data to the external services.

Our architecture in reality matches up pretty well with the three-tier model. The presentation and business layers are made up by the files that make up each page, and the data layer is made up by our two service classes that are injected into our first page. The presentation layer and business layer are fairly loosely coupled especially since there is no direct DOM manipulation in our code. The data and business layers are extremely loosely coupled, and the data layer could quite easily be swapped out with another with no adverse consequences.

2 Firebase Integration

Our application is a Tinder clone, therefore needed a significant amount of external interaction. We choose to use Firebase for our external storage as it is fairly easily to interact with, and there is good documentation available online.

As our service is account based, the first service we used from Firebase was the authentication service [1]. This was fairly easy to set up, and it handles all user authentication, from account creation, to logging in. The Firebase authentication service means that users can securely create an account, and their data is safely salted and hashed server side.

In our application each user has a profile picture, and in order to display this profile picture to other users the picture had to be stored server side. In order to solve this problem we used Firebase Storage. This allowed us to create a folder for each user where their profile picture is saved. When a user views the profile of another user, the profile picture of the second user is downloaded from our Firebase Storage.

Finally, we used Firebase's Realtime Database [2] to store user details, including personal information, and also to store conversations. Getting user details is fairly similar to profile pictures as mentioned above, but since the user data is all text, we can store it in a standard database rather than a specialised file storage solution. Conversations are also stored as text in the database, and both users have one conversation object. This means there are no concurrency issues, and there is a single source of truth without having server-side code.

In order to integrate our Firebase services into our app, we have a joint database and authentication service, and a reference to them can be gotten by any pages that need them through static getters. All database and authentication actions are in the data layer and are therefore separate and uncoupled from the business and presentation layers.

I found Firebase simple and pleasant to use, and the only gripe I had with it was the fact that it only supports a NoSQL database. NoSQL is useful in situations where you need to scale your database when you have large amounts reads and writes as the database can be distributed over ordinary, cheap servers. In the case of our app, there isnt much risk of the database needing to be scaled so a relational database probably would have been better for our needs. Additionally, the non-relational structure of a NoSQL database did not provide much of an advantage to us as our user data was fairly structured and uniform.

3 UX decisions

3.1 Color Scheme

We have employed a warm colour scheme for our user interface (see Figure 1 for a screenshot of the home page). We chose this colour palette as warm colours can have a major impact on how a user interacts with a user interface. Warm colours are "associated with passion, energy, impulsiveness, happiness, coziness, and comfort. [3]. There are two focuses we tried to draw from a warm colour palette, the passionate and impulsive side, and the happy and comforting side. The passionate, energetic size was considered to be extremely important as these are key things in a Tinderlike dating app. If a user feels energetic and passionate, they are much likely to use the app and make connections with other people. Additionally, if a user feels more impulsive they are more likely to sign up for the app, or to swipe right on people, and this means that the energetic feel that a warm colour scheme brings helps to persuade a potential user to use our app.

The happy and comforting nature of the warm colour palette was also considered to be extremely important to the overall feel of our app. Meeting and talking new people can be very stressful process and we want our users to feel as comfortable as possible.

Our primary colour is a pale yellow colour which can be seen in Figure 1, on the navigation bar. This yellow is present on all screens (through the nav bar) and it helps to produce a warming effect that evokes pleasant feelings [3]. The pale yellow nav bar is then complemented by a peach-orange colour that is present on most non-call to action buttons in our app. This colour is halfway between yellow and orange and helps to provide a warming effect to buttons that are meant to be pleasant to the user and have pleasant results. Examples of this include the button that takes the user to the cruise screen, the "login" button (see Figure 1), and the "message hub" button.

Additionally, our call to action buttons (see the sign up button in Figure 1) are a vibrant candy red that draws the users eye and encourages them to the action, and they fit in our warm colour scheme by providing a sense of "energy and confidence" [3].

3.1.1 Alternative Color Scheme

Figure 2 shows an alternate colour scheme that was considered. It is a much cooler palette, with blues and purples being the dominant colours. Cool colours are said to be "often associated with calm, trust, and professionalism." [3]. An overall cool colour palette would have then delivered an overall image of competency and professionalism. We considered the warm colour palette's energy, coziness and the comfort it inspires in users to be more important the aforementioned benefits of the proposed cool colour scheme. This is because we favoured a warm, playful, and happy feel for the app and we considered that be important in a dating app. A cool colour palette would suit an app that targets a more professional background such as LinkedIn.

3.2 Law of Common Region

We kept the law of common region [4] in mind when designing our user interface. The law of common region states that "Elements tend to be perceived into groups if they are sharing an area with a clearly defined boundary." [4], and we have taken into account this phenomenon. Items which are related, such as groups of buttons, are grouped together in common region and share common features (size, colour). A good example of this can be seen in Figure 3 where two distinct groups can be noticed. Firstly we have the grouping of the three warmly coloured buttons in the upper centre of the screen. These buttons provide the user with options to go to the main functionality of the app. These buttons are seen as grouped together due to being in the same area, and due to the sharing common features such as colour and size.

There is another clear grouping at the bottom of the screen in Figure 3. The two buttons in the footer are used to access pages that provide more information to the user about the app. These buttons are grouped together both by the extremely clear sharing of area (the footer), and by having a common colour and size (grey and 45% of the width of the page respectively).

These two clear groupings show that elements being together are strongly perceived as groups as they share a clearly defined boundary and share common features, in this case size and colour. This is a clear example of the law of common region in practice, and we kept this in mind when designing our interface by making sure that items that we wanted to be perceived as a group were together and similar to each other.

3.3 Hick's Law

Hick's Law states that "the time it takes to make a decision increases with the number and complexity of choices." [5]. We took this into account when designing user interface. The main idea behind Hick's Law is that the time it takes a user to do something or make a decision will increase with the amount of options that a user has. In our app we tried to reduce the number of options a user has available to them at any time. Less choices means the user will spend less time contemplating each of them, and they will move onto a choice quicker, which is ideal for our app.

An excellent example of this in our app is the conversation page. This is the page a user is taken to after they have matched with another user and want to chat to them. The user has only three options on this page. The first is option is to click the back button at the top to be taken back to the conversation hub, the second option is to view the other users profile, and finally the third option is to type and send a message to the other user. This is a very small collection of possible user options, especially when compared to other messaging apps such as Messenger or WhatsApp where the user is presented with a myriad of options. Less options, as per Hick's Law, means that users will spend less time trying to make choices are more time interacting with our app. We kept this purposely in mind, cutting out options we considered to be unnecessary all through out the app, and this resulted in a focused app where the user doesn't have too many options which means they should be able to use the app faster and get on with other things that are far more important in their lives.

3.3.1 Hick's Law, Again

An alternative design decision that we considered was making the app more complex, as in adding more choices for the user. This is illustrated when considering the initial design document, it stated that the app would have more messaging functionality (sending of pictures and other multimedia), but we saw the complexity more options adds, as per Hick's Law. Other examples of options that were considered that could have made the app more complex include splitting up of functionality in multiple pages and elements, and a settings menu. Ultimately we decided against most of these options as we felt they added needless complexity that would have made it harder to use our app.

4 React Native Critique

In this section React Native will be contrasted and compared to Ionic, and React (the web version, which will be referred to as ReactJS for clarity).

React Native and Ionic are similar as they are both mobile development frameworks that allow the creation of cross platform applications using JavaScript. A major difference between the two is Ionic uses HTML and Sass for creation and styling of the user interface and React Native uses JavaScript, in the form of JSX. React Native's approach of using JavaScript for the application logic and the user interface presents both advantages and disadvantages when contrasted to Ionic which uses HTML and Sass for the user interface, or ReactJS which is allows the mixing of HTML, CSS, and JSX.

JSX is used by both React Native and ReactJS and Facebook, the creators of React Native and ReactJS present their idea of the main advantage of JSX [6]. They state that in ReactJS (or React Native) "Instead of artificially separating technologies by putting markup and logic in separate files, React separates concerns with loosely coupled units called components that contain both.". The usage of JSX in React Native means that there is no HTML or CSS used. Instead the user interface and styling of the user interface are defined in the same JavaScript file where the application logic is contained. Facebook therefore seems to think that a major disadvantage of normal web or non-native web development is the usual three file system of HTML, CSS, and JavaScript. They refer to this as "artificially separating technologies", but it is fairly clear that it is a good idea to separate the user interface as a whole from the application logic [7], and the React team's argument that a standard separation of concerns is "artificial" is frankly laughable as it can, quite easily, be argued that having items in the same file/class means that they are not truly separated.

In ReactJS (the web version) the usage of JSX is optional, and I feel that this is a good compromise when compared to React Native that forces the usage of only JSX. Additionally, in ReactJS, a programmer can use standard HTML elements (e.g. ⟨div⟩, ⟨li⟩, ⟨b ⟩, etc.) but in React Native, a programmer cannot use standard HTML elements as HTML is not supported (without a significant amount of effort and third party libraries). This means that a programmer will find it difficult to simple tasks in React Native that would be trivial in a situation when HTML could be used. For a good example of this is is trying to show a list of bullet points, in HTML it is as simple as using ⟨li ⟩and ⟨ul ⟩tags. In React Native you'd either have to deal with a FlatList and all the added complexity that comes with that, or use a ListView and use the unicode character for a bullet point at the start of each line [8], which is quite frankly absurd. I think the ideal scenario for React Native would be to allow the mixing of React components, and HTML elements, as it allows the programmer to have a choice rather than being railroaded into a situation which may prove to make their job more complex, which is the exact opposite of what you want out of a framework/language.

Another major design decision made by the developers of React Native is the usage of their own stylesheets rather than the usage of CSS. They state that one of the advantages of this is the fact that this means all styling is in JavaScript rather than having to worry about knowing and using another language (CSS). Unfortunately, React Native's stylesheets just feel like cheap ripoffs of CSS, rather impressively, they manage to incorporate the (many) bad parts of CSS with none of the (few) good parts. The fact that React Native doesn't use CSS means that any CSS orientated tooling such as CSS superscripts (Sass, PostCSS) or even simple IDE features like autocomplete are useless to the developer. Documentation of React Native's stylesheets is also poor, especially relative to CSS, which has a massive amount of information out there. Overall React Native's stylesheets present no appreciable advantages over CSS, but present many disadvantages as they are quite new compared to CSS and haven't had the time to mature. Additionally, ReactJS allows the use of CSS for styling which makes using ReactJS a much more pleasant development experience when compared to React Native.

One of the biggest issues with React Native is any previous HTML and CSS knowledge is rendered obsolete due to React Native's focus on faux-HTML and faux-CSS. Any developer coming in React Native is likely to have HTML and CSS knowledge, and with the way React Native forces you to use user interfaces, this previous knowledge isn't particularly useful. Facebook presents this as an advantage as they say that you only have to know one language rather than three, but HTML and CSS aren't the most complex languages, and are fairly easy to learn, so I feel that this point falls a little flat.

My biggest personal gripe is the fact that when creating an Android version of a React Native application, you feel like you're not getting the same experience as you would if you were targeting iOS. There are numerous bugs that we found over the course of development that were due to React Native, and they were Android specific. Quite frequently you would see that the bug had been raised as an issue a significant length of time ago, and the issue was either ignored by the React Native developers or they'd stated that it wasn't a priority to fix as. This is in contrast to any iOS bugs which were quickly fixed.

Ionic feels like a suitable alternative to native (Android, iOS) development as it provides a completely different development experience when compared to native development, whereas React Native just feels off. Little things that were simple in Ionic, like unordered lists, were far more difficult than they should have been in React Native. Overall, if I wanted to develop a cross-platform app in the future I'd use Ionic, and if I wanted to do something more more I'd just an Android app and an iOS app. React component architecture works well in ReactJS when it can be mixed with HTML and CSS, but it really falls flat in React Native.

Overall, I find React Native to be extremely impressive, not because it's actually good, but because it makes me miss Ionic, HTML, and CSS. This is a major triumph of mankind, and puts it in the running for the eighth wonder of the world.

5 Appendices

5.1 Division of Work

For this project I worked in a pair with Simon Pope (3003343009). The app idea and architecture are a joint work, and then we divided the pages between us and then worked on them separately.

5.1.1 Joint Work

- Architectural Design
- Data services (entire data layer)

5.1.2 Aden

- About Page
- Conversation Page
- Help Page

- Home Page
- Loading Page
- Message Hub Page
- Register Page

5.1.3 Simon

- Cruise Page
- Edit Details Page
- Hub Page
- Match Page
- Post Review Page
- Review Page
- View Match Page
- View Profile Page

References

[1] "Firebase Authentication — Firebase." `https://firebase.google.com/docs/auth/`, 2018. [Online; accessed 31-August-2018].

[2] "Firebase Realtime Database — Firebase Realtime Database — Firebase." `https://firebase.google.com/docs/database/`, 2018. [Online; accessed 31-August-2018].

[3] "How To Use Color To Enhance Your Designs — Vanseo Design." `http://vanseodesign.com/web-design/color-meaning/`, 2018. [Online; accessed 31-August-2018].

[4] "Law of Common Region — Laws of UX." `https://lawsofux.com/law-of-common-region.html`, 2018. [Online; accessed 31-August-2018].

[5] "Hicks Law — Laws of UX." `https://lawsofux.com/hicks-law.html`, 2018. [Online; accessed 31-August-2018].

[6] "Introducing JSX - React." `https://reactjs.org/docs/introducing-jsx.html`, 2018. [Online; accessed 6-October-2018].

[7] P. A. Laplante, *What Every Engineer Should Know About Software Engineering (What Every Engineer Should Know)*. Boca Raton, FL, USA: CRC Press, Inc., 2007.

[8] "React Native 'Unordered'-style List - Stack Overflow." `https://stackoverflow.com/questions/39110460/react-native-unordered-style-list`, 2016. [Online; accessed 7-October-2018].