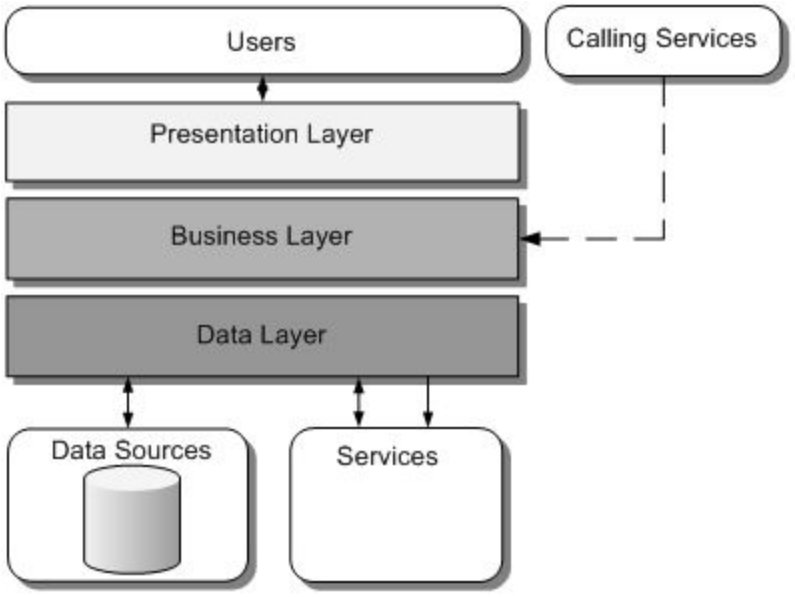
Anime2U Mobile Application Development **Niran De Silva 300387070**

Overall architecture:



The architecture of a mobile app adheres to a strict communication protocol between three main layers. These layers are the presentation, business and data layers. Where the presentation layer can only communicate with the business layer and the data layer can only communicate with the business layer, whilst the business layer can communicate with both data and presentation as a middle man. All postings and queries related to the external component or public API had to be processed through the business layer and fed into the presentation layer(HTML) as a readable and presentable object.

The architecture of my app starts off with the presentation layer which is where the user will be interacting and querying functions from my business layer which in turn call services from the data layer. The presentation layer serves as a platform for the user to essentially get and post data without having any interaction with the inner layers. The presentation layer is written mostly in HTML with a bit of CSS when object and component styling was required. The HTML determines the layout and viewing dynamics of the user interface. Components such as button and select options are directly linked to the business layer on most occasions through (click) where the typescript is responsible for functionality. Dynamic updates on the presentation layer is directly linked to updates that are of local field data as well as database status. These updates are then displayed dynamically on the user interface through function calls from the business end that fetches data from the data layer. The presentation layer is responsible for the arrangement of this processed information in order to form a simple yet easy to understand graphical user interface that users are able to intuitively interact with.

The business layer is the layer which manages most of the functionality and domain logic of the system. Whilst the presentation layer manages the arrangement of the data

is stored locally, the business layer manages how my app stores this data, creates it and changes it. Typescript is the main language used at this layer. The layer itself written in this typescript uses functions to manage the outcome of the user interacting with the interface. Functions call upon locally stored data about the user which is initialized upon entry of the page in the local constructor. When the constructor establishes the connection between the user and their own unique profile in the database, the user is then able to click on the interface, calling upon functions which fetches data through connection that was established earlier. The business layer uses services which are contained in the data layer to serve as a connection platform in order to get and post data to and from Firebase respectively. User profile information is read and updated in this layer in order to personalise the user experience and provide them with the necessary data promised by the app description. In terms of my app, the user selects and anime to add to his or her list, which will call upon a function in the business layer first which then calls upon services in the data layer in order to store this anime in the list for later viewing. The same process is viewed in order to fetch anime data after it has been added but this time a getter function is called which accesses a database query in the data layer. This will ultimately update the relevant state of the presentation layer after query returns the current contents of the users profile and what he or she has added to list prior.

Lastly the data layer is responsible for the simplified access of data relevant towards the functionality of the app. The business layer acting as the middleman uses functions to call upon the data stored in the database, which in this case was the Firebase real time database storage system. URL access was used throughout the app as a service for accessing and updating objects in the real time database. In terms of getters, updateChanges() and valueChanges() gave me access to the contents of firebase list contents as well as the random keys that the contents come under. Firebase has a system where keys are automatically generated which each new addition to the list. So access is only possible by knowing key values, which is where snapshot changes worked in conjunction with value changes in order to provide relevant access to leaf nodes in my database. The data layer in this case is a separate typescript page which handles the accessing of data so that the business layer does not get too cluttered dealing with pushing and querying functions at each page. Abstracting this idea of data access into a layer of its own allows the clean access of data without the need to reinitialise the database and its contents at each instance of retrieval. However in the case of the anime API I used, the data layer and business layer are almost unified as there is no posting to the API nor is it accessed at many pages. Anime JSON fetching is dependant on the URL used so at each page the data that is fetched is different but only

by the margin of minor URL manipulation so separate services did not need to be used for this case.

I believe the styling and architecture of my code adheres to the three tier model decently as there is clean and obvious separation between presentation and business layers especially so due to the two different languages being used to prove its functionality. I believe the overall functionality at the business layer in greatly cohesive as all the pages in the layer are all built to serve its respective purpose. All functions in each page are relevant to the purpose of the page and correlates to what the user receives from it visually, through the presentation layer. The layers themselves are also fairly loosely coupled especially that of the business and data layers which can both operate to a certain degree independently. The presentation layer however has a stronger connection toward the business layer and is therefore transitively coupled with the data layer due to most of its functionality being dependant on the data that is posted and viewed by the user.

Firebase & Jikan API - External components

My application serves as a platform for those exploring anime and want to keep track of the shows they are watching. A profile is created for each user so that they can explore, add anime to their lists and keep track of relevant news. On the profile end an external storage had to be used in order for users to have their private profile with their own unique lists, so Firebase real time database was used to accomplish this. Firebase's real time database is quite well documented and easy to understand when dealing with data storage and has an easy to interpret UI when dealing testing data storage. Another external component used was the JIkan API which is considered to be the unofficial MyAnimeList API which contains over thousands of titles of anime, movies and OVA's. This was a major requirement as I needed source all the titles possible in order to keep the service relevant to the modern anime fan.

As mentioned earlier most of the app functionality is based under the premise that the user has an account, so for account creation and authorisation I used Firebase authentication. Specifically I used the register with email and password option as this is the most commonly used type of authentication across the industry. The Firebase authentication SDK provides this functionality and also allows for potential password resets through email verification which could easily be another layer in terms of user profile management [1].

In my app there is a user profile page as well as a search page where firebase storage is being used. The real time database is a cloud service which stores all firebase related objects as JSON and allows for for all the users to receive live updates when data is changed within the database [2].The search page allows for users to search for anime they like and then have a choice to add them to their lists. These lists are initialized along with userID, profile picture and the name of their favourite anime upon registration. While the registration sets up the basic flow or tree like structure in the real time database, the search page offers the user the ability to update the initially empty lists. On the profile page there is a basic count of how many anime are being watched by the user, completed and yet to watch. This count is fetched from the number of anime contained in each of these firebase lists unique to the user that has been authenticated. When entering any of the lists contained in the profile page, there is more information that is presented such a pictures, titles, score etc… This is due to the whole JSON object of the anime being stored in the database, which is then retrieved to present to the user when they want to view the current state of the shows they are watching.

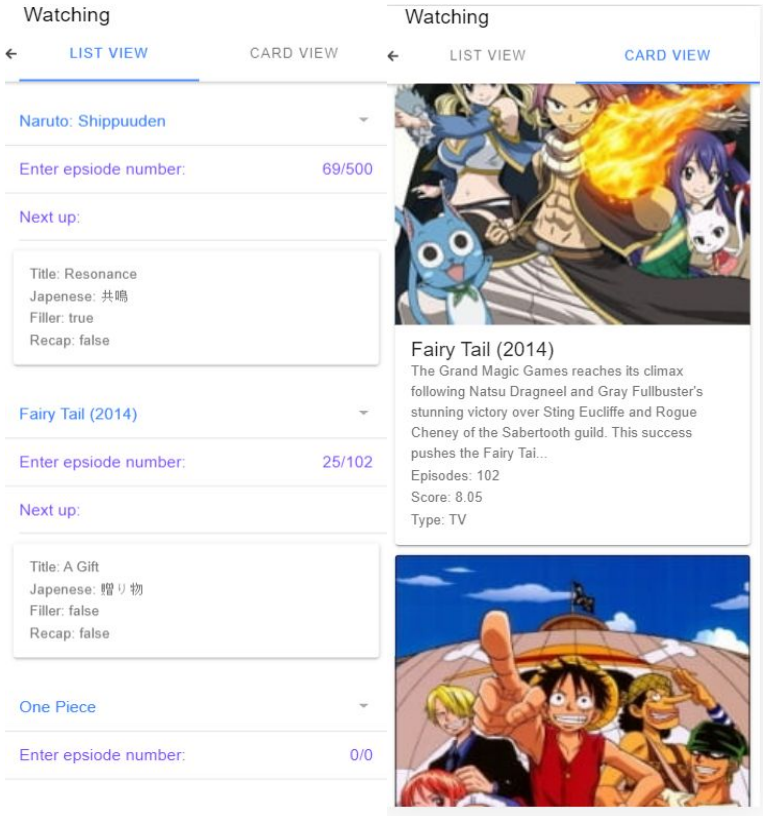
In order to weave in the real time database into my app architecture I injected these services in order to get and post anime within the profile and search pages. By injecting the service into the business layer, retrieving and updating becomes simplified as the firebase specific notation such as valueChanges() and updateChanges() (for Firebase lists) are separately implemented in the data layer alongside promise statements help the data layer safely distribute data before the presentation layer tries to access undefined values [3]. These services and states of the database are statically accessed throughout the two pages that need access to them and can be implemented anywhere in my app as the styling of the data layer allows for it to be loosely coupled against the other layers. By using Firebase real time database I was essentially subscribing to a NoSql type database. Now in terms of scalability I am at a huge advantage as there is a lot of vertical scalability to support my app if it grasped the attention of thousands, however this is not the case. SO in that respect having a vertical style of scalability would have been more appropriate using a relational database. However in terms of hierarchical structure the NoSql database trumps relational which the leading reason as to why it was appropriate over relational for the functionality of my app. The tree like (non uniform) structure I mentioned before was as simple as it was due to the optimizations of NoSql.

The other external component I used was the Jikan API in order to fetch through thousands of anime titles. My new page in particular takes advantage of the API as it fetches the JSON, of max 50 per page, of the top anime in the world currently. By

appending further specific subcategories you are able to fetch the top that is airing, most popular and yet to come. These three sections provide most of the content of my news feed which is a place for users to go and see what is trending in the anime world right now. Same logic applies for the search page as well which uses a search typed URL along with a search query parameter which in this case is the key press of the user giving dynamic updates to the presentation layer domain by constant calls to the API [4]. Whilst this was a great feature to have, it may not be feasible on a large scale as the Jikan API has its limitations for calls allowed to be used within a certain timeframe.

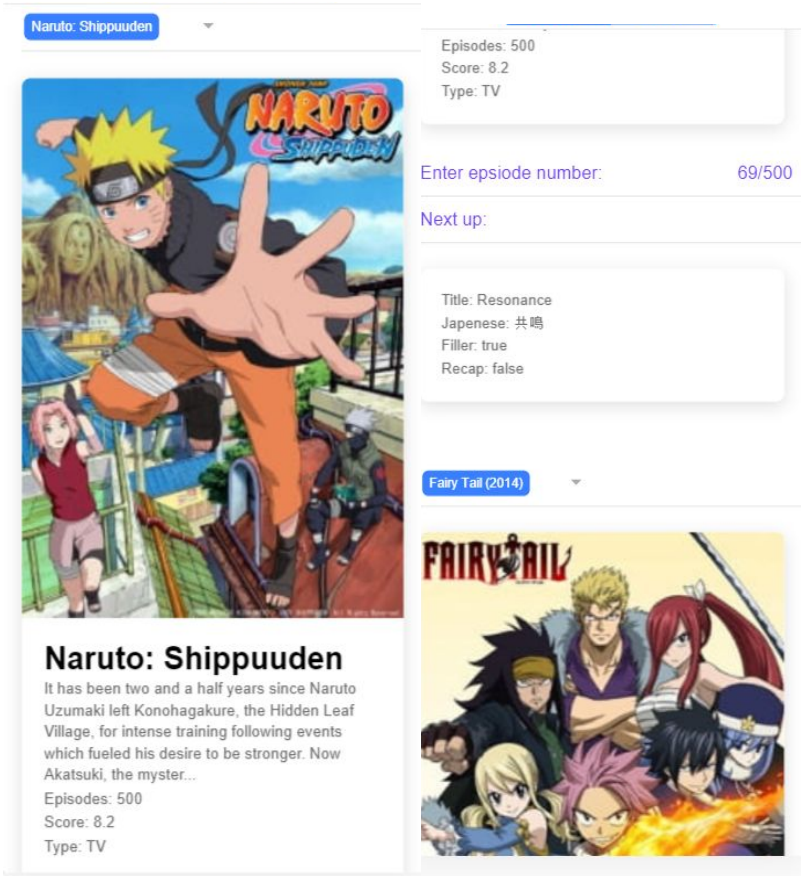
UX Decisions: Segmentation

In ionic there is a segmentation component that allows for the simulation of viewing two different html scripts or pages. Near the header there the page is segmented into two sections, which are list view and card view. Now these two are interconnected in the sense that they both display anime that user has added to their list, however as the word “view” suggests they have two very different perspectives on this output. If we take the watching section, which contains most of the deliverables of this app, we can see that the list view serves as the sole editorial region for each anime. The list view allows the anime to be displayed in list form and gives the user options to move the anime between other lists as well as update the episode number they are on. Removal from the list is also done in this view. So this view essentially serves as the section with the highest interaction rate. The card view on the other hand is the view which utilises the ion card component like the news feed section in order to provide a more detailed scope on the anime. Showing the image of the anime to the user alongside a brief synopsis helps them visualise and remember the anime they are watching especially for the potential power user who may consume over and beyond the average user. By giving separation to these tightly coupled segments the user is freed from the potential clutter on the main business end which is the list view, but through this



simplification, is not deprived of the detailed more graphical view achieved through segmentation.

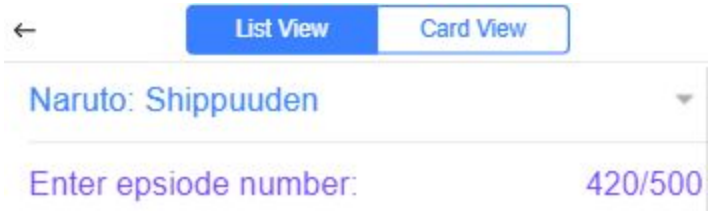
An alternative to the segmentation of the anime lists is use a drop down component on each anime title, giving the same view as the card view but all on the list page. So essentially having one page that does the work of both but with a show and hide type drop down giving the card view element. In a narrow perspective this would greatly simplify the overall UI and add further dimensions to one mian page. The show and hide would allow the user to quickly peek at the details of the anime for a refresher before updating the status of their anime. However the issue here is the clustering of too many functionalities. Although the idea of a drop down is great in terms of raw efficiency and screen estate, it amplifies the user's thought process when using the page. Although a drop down section may not be a huge intellectual obstacle, it reduces the simplicity of the main business end when multiple expansions occur. The main business end should be as simple and inviting as possible in order for the broader audience to have a positive outlook on the app, which coexists with the Polar Books's idea of not diverting course of action when the momentum is in one direction [5]. An argument here may be that a power user may prefer compact and efficient nature of having drop down that they can take advantage of, however with a more business outlook you can say that the power user only amounts to a smaller fraction of the general user group and so a simple and more inviting interface that is cleanly divided into cohesive segments is more appropriate for the wider user group.



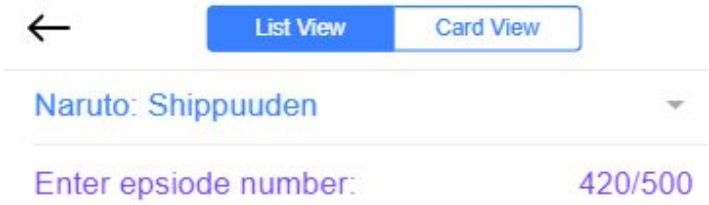
The top two images show the original state of the app with the list view and the card view.
The bottom two images show the alternative state of the app after the segmentation is replaced by the drop down effect.

UX Decisions: Navigation

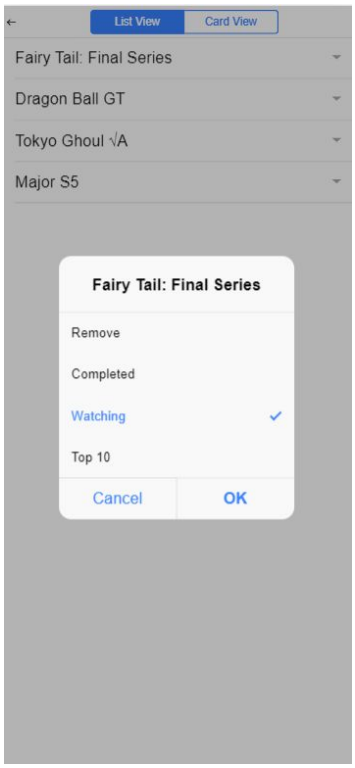
My main method of navigation across the app is through tabs which separate my app into a news feed section, a profile page section and a search section. The main business end is in the profile section as this where user information and user anime lists are stored. Once in the profile section you have the option to visit any of the leaf nodes which are different anime lists you can add to (watching, completed, yet to watch and top 10). Now at these sections it is vital to have back button as these pages have no tabs to escape and rely on being able to enter the profile page in order to successfully navigate through tabs. The back button I have chosen automatically updates based on the OS using the app. However the back button is small and sit on the top left as shown by the image.

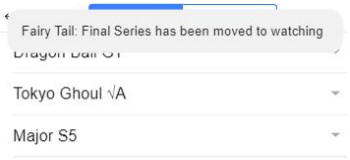


An alternative here would be have a larger back button in order for the user to access it with ease and not waste effort finding the button. In app development smooth navigation is key where “reasonable navigation system and a smooth path to the task allow users to quickly achieve their goals and form a smooth user experience”[5]. The main focus here is the path back to the profile page where the user can view their updates, whether it be from moving an anime from page to another or to have the holistic count view of how many anime are in each of their lists. Having a larger back button could lower the sense required to achieve this. Now the counter argument that justified my initial my choice, was based on the premise that this is for Mobile applications. What I mean by that is most mobile phones have their own native back button and accompanying the native back button with another bigger back button is essentially a waste of screen real estate. Although a large button is visually more guiding, it is can also prove to be too much of a nuisance when there is now a higher chance of accidentally pressing the back button when coupled with native back buttons. Keeping the button small and simple allows the user to cleanly navigate without the hassle of having to go back to the page they were on previously due to lackluster UI design.



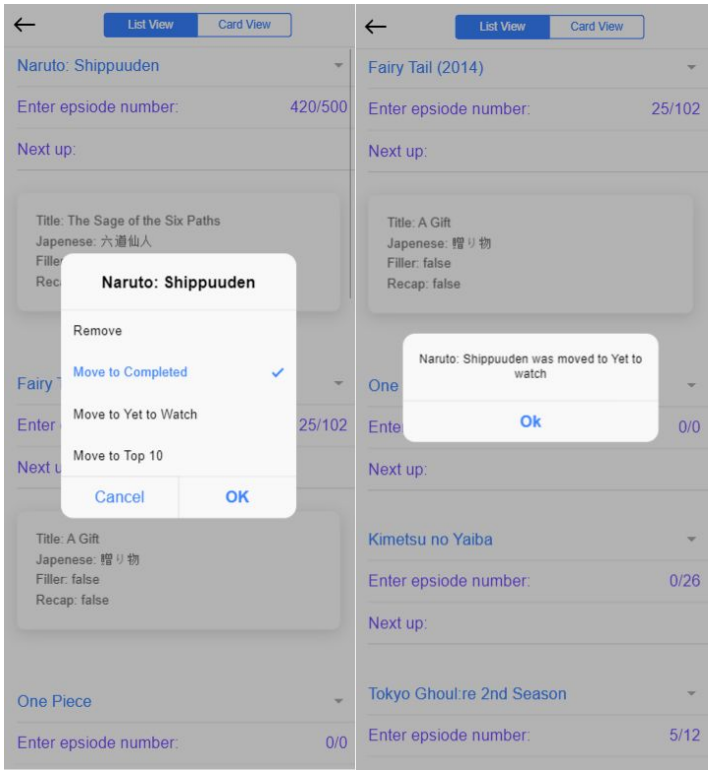
UX Decisions: Toast





When attempting to move any anime from one page to another or attempting to remove one, a toast is presented to the user. A toast is a notification style drop down that presents itself for 3 seconds before disappearing. The point behind it is to notify the user that they have actually removed an anime from the list or moved it to another list after they have completed to the action through the ionic selection pane. Its serves merely as a reminder of the action they have completed. The purpose is to combat instances where the user misclicks and is notified of the action that was completed in order for them to adjust accordingly, or when the user selects an option and presses outside the selection box instead of the ok which ends up making no changes. The last example is especially important as users can be misled by the ion select box dynamics into thinking that they have moved an anime to another list. Finally the main reason the toast is presented is to simply guide the user through the application, as the simple interface can prove too vague to some users and giving brief details to an action may prove helpful. The downside to this approach is the fact that functionalities in the top bar such as switching between views and pressing the back button is halted for the three second duration due to the toast covering the sub header. There are options for the toast appearing at the bottom or the side however this style of toast in not visually pleasing to the eye. The dynamic of looking down for a notification is not a commonly used strategy across other IOS or Android apps and so the traditional user will therefore not feel accustomed to looking anywhere but the top layer for any notifications.

The alternative for this is to use a more detailed ion select box in order to combat the issue of the new user feeling misled by the simplicity of the UI. As I explained earlier the initial user may appreciate the simplicity of the UI however may not understand certain dynamics such as moving between lists unless some kind of tutorial is given. As a lot of users do not actually pay attention to tutorials, giving more details to the ion select option labels can prove to be a successful option. The other change here is that an alert is used instead of the toast functionality in order to free the user of the top layer coverage that occurs when the toast is presented. With the alert the user is alerted of the action they have made and given an ok option to continue with other interactions once they are ready. This also ensures that the user has read the change they have made before pressing okay which may not occur within the three second timeframe of the toast (user dependent). However this alternative falls into the trap of not performing tasks optimistically [5]. The Polar mobile book states that people use mobile apps to use in order to “feel” like the functionality is faster in order to eclipse the fact that the server response time may be slow (due to the difficulty of optimizing CSS or HTML variables) [5]. Giving an alert of a change like the alternative shows feels more like a web application decision as opposed to a mobile one. Although technically it allows the user to operate at a faster speed by the mechanics of the alert, it does not necessarily feel that way when two pop ups occur consecutively. The key word here is “feel”, and this is more of a matter of perspective as opposed to physical mechanics. Giving the toast feels more fluid and “mobile like”. Allowing the user to see and carry on without having to click on a pop up window in front of them to exit. This fluidity is paramount in mobile app navigation whether its on a page to page level, or internal component level. These are the reasons why I chose to go with the initial toast controller as opposed to the alert alternative.



Ionic Framework

The structure of Ionic, as briefly explained in my architecture section, is split into three parts which is Typescript, HTML and SCSS to be exact. Typescript is a superset of Javascript and for the most part inherits all the functionality of it. The same goes for SCSS as it is a superset of CSS and is supposed to be a language which inherits more conciseness over CSS. This split allows for the main functionality of the application to be written in typescript, where all click commands and slide options mechanics are dealt with. The arrangement of the ionic components are done in the HTML page with a limited styling, while the SCSS page is used to further style the HTML elements and components with more preciseness. Once the three languages are understood a clear flow begins to show between them and an appreciation is found for this separation that ionic as a framework provides. However on a beginner note the three languages themselves may prove as a massive learning curve especially when deprived of any front end coding experience. Luckily Java being my preferred language helped me quickly grasp the potentially steep typescript semantics after a short period of time.

A big advantage of using the ionic framework for this assignment was the user friendly interfaces I was able to create in a remarkably short period of time. Ionic utilises Javascript, HTML and CSS layers in order to create beautiful UI designs. On the presentation layer HTML especially takes advantage of the ionic components such as card, segment, select etc… in order to create professional looking interfaces without having to manually create the shadow of the card of the border of the segment, to name a few. HTML as a language however does have its downsides as it is not as readable as the typescript layer which verbally makes more sense than the HTML. However a lot of UI manipulation can be done purely using HTML which came as a huge advantage. I found myself doing most of the styling using the HTML language rather than SCSS as the sizing, arrangement and compositions of a lot of the ionic components are already user friendly and align very well with the rest of the app by default. Components such as card view had a lot of documentation surrounding it, with many alternative templates for customization which gave me more than enough options to build a basic app off of. Ionic documentation is simple and straightforward, most of my problems were solved by checking the docs. After using the docs to create tabbed navigation, it was mostly a matter of using the right components for UI design, before I got section like services and the data layer which required further research.

I also found the Ionic CLI very powerful in terms of graphical representations and easy to navigate. As this is built in to Ionic it is easily accessible and adaptable to other platforms.

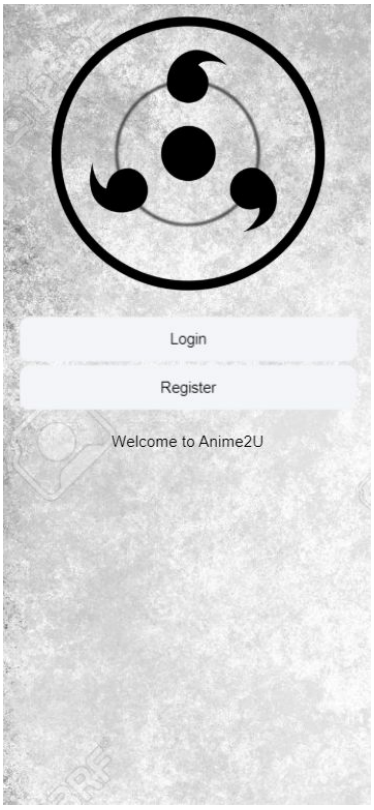
There are disadvantages when it comes to performance of an Ionic as it is not native to a specific device however for the scope of this assignment this was not an issue. An example of this is when dynamically updating the UI on key press, which has a slight lab in performance with faster typing, which is a combination of API connection access issues and ionic as a framework. For flawless performance an obvious solution would be a native framework which is slower in terms of development but more optimised.

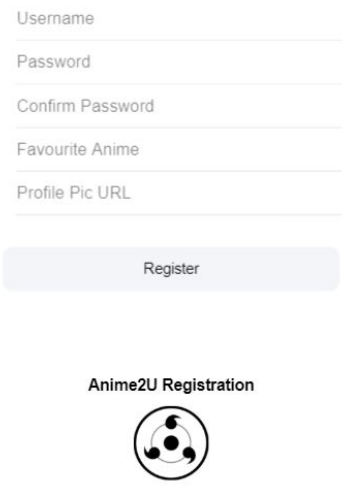
Typescript as a language comes in handy for a first time mobile app developer as it uses static typing which allows for type checking at compile time which is a feature not apparent in traditional Javascript. However Typescript does not support abstract classes like Java or Javascript, however something similar can be replicated with the prototype feature on a Typescript class.

A specific skill set of AngularJs is also recommended when using the ionic framework which is is a learning curve of its own however is widely used in the industry so is very well documented and bugs are easily able to be resolved. AngularJs has a very good separation of dependency injection concerns as well as the capability to create an maintain pages in an app in a cohesive and loosely coupled manner. Routing proved to be the trickiest obstacle to overcome especially with tabbed routing. A routing module had to be created in order to achieve this and this was by far the most complex aspect of the application development I had. Documentation for specific bugs and errors in angular had a small disadvantage as there are many versions of angular and most of the documented ones are that of previous generations, so a lot of research had to be done at times to resolve Angular 7 specific issues.

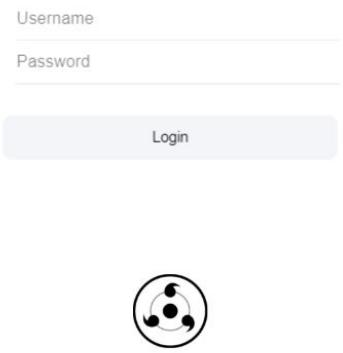
Overall my experience with Ionic was a pleasant one. Most of my concerns initially surrounded the idea of creating an app that was actually functional. Due to the typescript being so well documented and bearing decent resemblance to the back end Java programming language in most notations aspects, I was able to deploy basic app functionality fast. This gave me time to play around with the many well documented Ionic components which was able to add a layer of attractiveness to my application. Due to Ionic understanding the essence of industry style mobile apps, a lot of default templates given in documentation has been tailor made and styled to look professional. This saved me the time of learning SCSS to a further degree and ensured successful completion of a well functioning Anime2U service.

Appendix - Pages with description

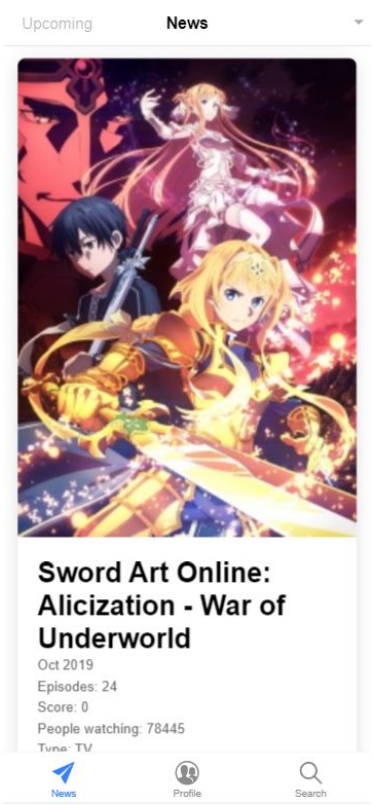


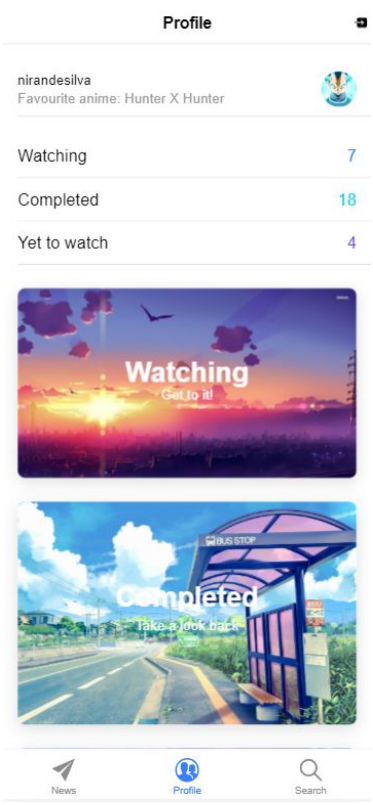


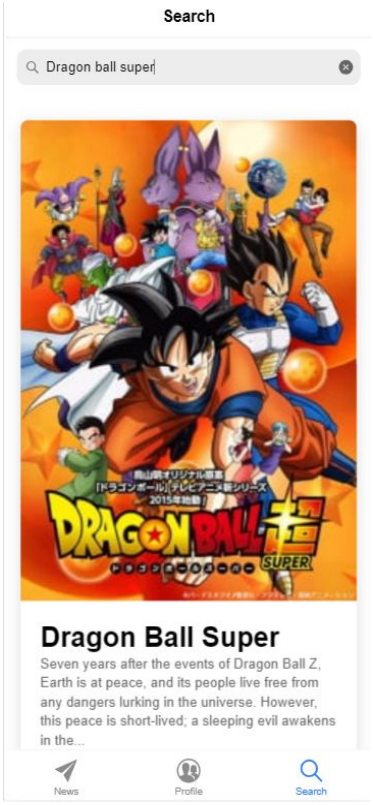
Anime2U Registration

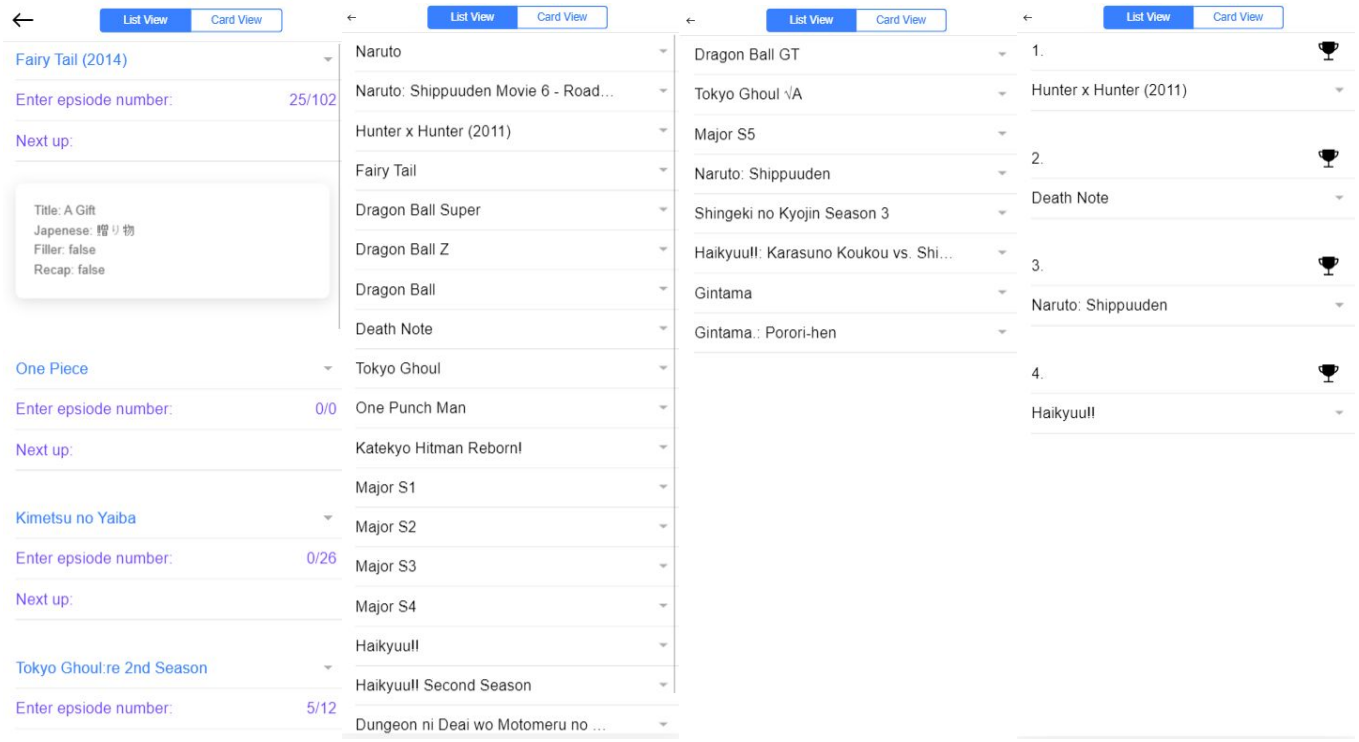


Anime2U Login









Home Page - First row left side

The home page is a basic introduction to the app and shows the logo of the Anime2U service. It provides the user two options, login and registration. The homepage has styled to wow the user and give them perspective of the app they are using which is essentially an anime fan app. The logo itself is a clan symbol called the Sharingan (Uchiha clan), from a popular anime called Naruto.

RegisterPage - First row middle

The register page provides the user a place to enter their personal details in order to create an online account on the Anime2U service via the Firebase authentication process. In this case it is using email and password authentication with a few personalised variables such as favourite anime to give the user a more unique feeling and gives them a platform for self expression which will be displayed on their profile page upon entry of the app. The logo at the bottom allows for navigation back to the main home screen, discrete but stylish.

Login Page - First row right side

The login page provides the user a place to enter their login credentials of a pre existing account in the Firebase authentication base. The logo at the bottom allows for navigation back to the main home screen, discrete but stylish.

News Page (tab) - Second row left side

The news page allows for the user to experience information about the top anime in the world right based on overall score and viewers. This information is retrieved from the Jikan API which is the unofficial MyAnimeList API, so it kept up to data with the latest trending anime related information. The ion select option at the top allows the user to filter the displayed anime between top-most popular, top-upcoming and top-airing. This tab is meant purely for viewing and exploring a limited set of titles across various genres.

Profile Page (tab) - Second row middle

The profile page allows for the user to view personal information that was created along with profile creation such as profile image and favourite anime. This also serves as the main hub for viewing the anime the user has added to their lists. The lists contained in the profile page is watching, completed, yet to watch and top 10 lists. The user can check the contents of any of these lists by clicking on any of the respective cards. There is also a count for each of the lists except the top 10 (as there can only be 10), in order for the user to have numeric idea of their consumption. This is also where the logout button is contained which upon confirmation will navigate the user back to the home page.

Search Page (tab) - Second row right side

The search page allows for the user search across thousands of anime titles available through the Jikan API. The UI is updated dynamically through key press and will show the user a card view representation of the anime title with an image and various other details such as rating, viewers etc… Each card also has an ion select pane that allows for the user select which list they want to add the anime to from the lists I described in the Profile page. Upon selection the user lists will be updated.

Watching Page - Third row left side

The watching page allows for the user to view their previously added anime titles and shows the amount of episodes the anime has. There is an ion label with an user input area which prompts the user to enter the current episode they are on in order to keep track of their state in the anime. When the user updates their episode number the section thats says next up will update and give details about the next episode in the anime series. This is mainly to hype the user for the next episode before they watch and let them know if it is a filler or not. All of this is contained in the list view which is one two segments. The second segment is simply a different perspective of the users added titles, which gives the previously seen card view in case the user requires a visual to remind themselves of the anime they are watching. The user also has the option to remove the anime from the list or to move it another list. E.g can move the anime to completed upon watching the last episode of the anime.

Completed Page - Third row left side
Yet to watch Page - Third row middle
Top 10 Page - Third row right side

These pages are very similar to what I described in the watching page however does not contain any of the episode number inputs or next up episode material, just the title and the ability to move the anime around the lists and removal. Card view is also the same. Top 10 has a small difference with a trophy icon to show the user that this is one of their special all time animes.

References:

[1]"Firebase Authentication | Firebase", Firebase, 2019. [Online]. Available: https://firebase.google.com/docs/auth. [Accessed: 19- Aug- 2019].

[2]"Firebase Realtime Database | Firebase", Firebase, 2019. [Online]. Available: https://firebase.google.com/docs/database. [Accessed: 19- Aug- 2019].

[3]"Firebase Android Playground — Realtime Database", Medium, 2019. [Online]. Available: https://proandroiddev.com/firebase-android-playground-realtime-database-560d4e18404a. [Accessed: 19- Aug- 2019].

[4]"Jikan · Apiary", Jikan.docs.apiary.io, 2019. [Online]. Available: https://jikan.docs.apiary.io/#reference. [Accessed: 19- Aug- 2019].

[5]Static.lukew.com, 2019. [Online]. Available: https://static.lukew.com/MobileMultiDevice_LukeWsm.pdf. [Accessed: 19- Aug- 2019].