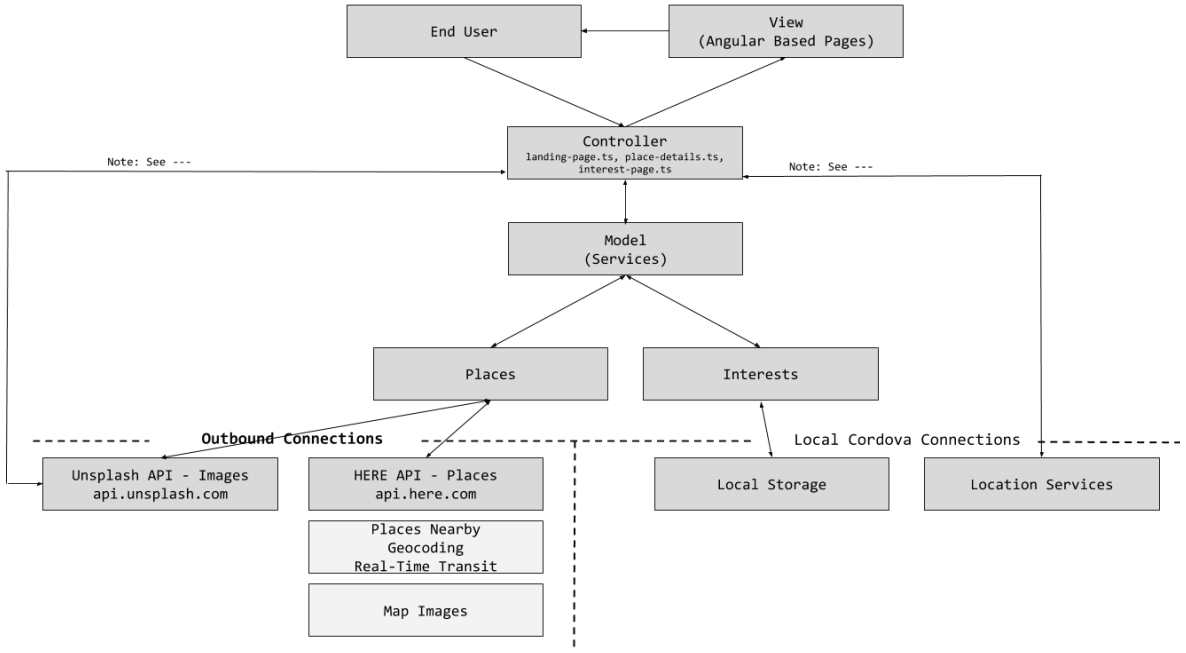
Brice Wilbanks
SWEN 325 Assignment 1 Report
August 12, 2019



## Architecture

The architecture for the mobile app is laid out similar to the traditional Model-Controller-View architecture popular in the software engineering field and presented in the book, Microsoft Mobile Application Architecture Guide. In this architecture, users are presented with a response from the view. Requests sent by the user are directed to the controller. Within the controller, data is requested from the model, parsed, and shipped to the view to complete the circuit.
Below is a diagram showing how the mobile app is setup with regards to the MVC architecture.



*View*

The view is comprised of the Angular based HTML pages. Since all the pages in the app rely on external sourced data, the view is substantially reduced without data being loaded from the controller. On the homepage of the app, the places results, filter chips, location vicinity information, and images are generated from these external sources and are sent from the controller. All data results shipped from the controller use Angular's built-in ngModel.

*Controller*

The controller is where most of the parsing, requests, and organization happen in the app. The user sends requests to the controller to process, generate, or update data. These interactions are page specific and are laid out below.

`User Requests to Controller`

| `landing.page.ts` | `place-details.page.ts` | `interest.page.ts` |
|---|---|---|
| `-Update Location`<br>`-Search Location by Keyword`<br>`-Filter Chips (timebased, location based)` | `-See Place Details`<br>`-Get Transit Details` | `-Toggle Interest` |

The controllers only job is to parse data from the model. This further reinforces the MVC model throughout the app. However, there are two exceptions where the controller directly accesses data without a model:

1. The `setLocation` method does not directly access a model before retrieving the location. Since the data processing is happening at this level, and location access is a one line Geolocation request from a Cordova plugin. It would be space inefficient to create a seperate location service for this application.
2. The `headerImage` method accesses an external API without passing through a model. The reason this API is not set up as a service under the model is similar to the above reason given. Since the API is RESTful and there is only a one line HTTP call, it would be inefficient to have this set up in another service. If the app were to scale, it would be more effective for maintainability and readability to create a separate service for images. Additionally, if there were more image resources, they could all be grouped in an external service.

*Model*

The model of the mobile app was constructed with two services, the `places` and `interest` services. Both of the services serve specific reasons as listed below:

| Places Service | Interest Service |
|---|---|
| This service handles all HERE API requests for | This service is used for all operations where the |

| | |
|---|---|
| places. Some methods include:<br>-Get a list of places, given a location<br>-Get a list of places, given a URL<br>-Get a list of places, given a category<br>-Get geocode match, given a search query<br>-Get transit information, given a stop ID | users personal preference/interest plays a role. For example, "Filter" chips on the homepage, generating the cards where users select their preferences, and toggling categories.<br><br>Additionally, this service is only service agent in the app that uses the Ionic Storage API for local storage. |

*Mobile Considerations in Architecture*

1. Fault/Inaccurate GPS Responses - in-order to combat spotty and unreliable GPS coordinates, the mobile app stores known GPS location information in the local storage. This comes in handy when there is an error retrieving the location of a user. Errors can be a result of users blocking location permissions, poor connections, or GPS dead zones around the world. When this happens, the app is able to pull the last known locations of the user to give accurate results.
2. Internet Connectivity - Due to the nature of cell phones, they often times experience poor connections with cell towers or, often times, no connection at all. While it is impossible to get refreshed place data without an internet connection, the app attempts to minimize the bandwidth required to get data. It does this through the web fundamentals of "lazy loading"[1]. After analysis, the places results data is, on average, 4kb.

**External Components**

Within the mobile app, there are two major external data services being used. The first is the HERE Places API[2] which provides most of the functionality found within the data layer of the apps architecture. There are five major components of the HERE API that were employed in the mobile app:

1. Places by Location - Given a GPS coordinate and optional category information, the API provided a list of places nearby, with the ability to access more places with the same query. This allowed for the construction of a seemingly endless list.
2. Place Details - Each location in the list provided above has a unique identifier. Using this identifier, the HERE API can return more information about the place. This provides the information for the `places-details` page found in the app.
3. Geocoding - Given a string query, the API is able to return a possible match for the vicinity, city, and country anywhere around the world. This is used for the Location Search feature on the `landing` page.
4. Map Images - Given a coordinate, the API can return a hotlinked image showing the point on the map. This was used to display the minimap on the `places-details` page.
5. Real-time Transit Data - Given a transit stop ID, the API is able to return real-time transit information; For example departure times and line destinations.

With its wide selection of entry points, the HERE API suited the needs for the user to access place information. To organize all the entry points and available calls, all access of the HERE API was placed into a separate Ionic Service called the `places-service` and each entrypoint was therein a service agent. This architecture is mentioned and suggested by the Microsoft Mobile Application Architecture Guide (62)[3]. This structure removes the clutter of retrieving information from the business and controller layers, thus increasing readability and maintainability. Additionally, this architecture allows different parts of the business layer to request data from the service layer, thereby eliminating repeated code and allowing reuse.

---

[1] More Information: https://developers.google.com/web/fundamentals/performance/lazy-loading-guidance/images-and-video/
[2] More Information: https://developer.here.com/documentation/places/topics/what-is.html
[3] Book provided in Lecture; Additionally found here: http://robtiffany.com/wp-content/uploads/2012/08/Mobile_Architecture_Guide_v1.1.pdf

The HERE API places by location entrypoint can be filtered by category when making requests to the API. These queries based on the user's interest are handled by the interest service before being used as parameters, and in some cases promises, in the places service agents. Once the category based query terms are formatted in the `interest-service` and bundled at the business layer, they are directly inserted into API requests in the `places-service` service agents. This structure and separation further organizes the code to where the service agent is solely for requests to the HERE API. The HERE API returns data in traditional JSON formatting, however the Ionic HTTPModule then returns this data as an observable. The business layer subscribes to this returned observable and parses the data accordingly to ship to the presentation layer.

Another major component integrated into the app was the Unsplash Photos API[4]. This API provides royalty free, stock images for web and mobile apps. The entrypoint on the API allowed for a text query to search images. Therefore, the business layer creates queries and calls the API recursively until a photo match is found for the users location. Once a match is found, the Unsplash API requires photographs be attributed to the provided author and for images to link directly to the Unsplash content delivery network. This is known as hotlinking and allowed the app to have high quality photos without image processing or having its own database. Since there is only one entry point being used for this component, there is no separate service and requests are handled directly in the business layer. It would be space inefficient and somewhat counterintuitive to have a photo request be separated into its own service. Furthermore, the Unsplash Photo API would not be effective if written in the `places-service` due to the differing tasks of the HERE Places API and Unsplash Photo API.

[4] More Information: https://unsplash.com/developers

**Ionic as a Framework**

There are three main ways to develop mobile apps in 2019[5]:

1. Using the native platforms. Specifically, directly through XCode with Swift or through Android Studio with Kotlin/Java. The disadvantages of this method is the time to deployment needed for cross-platform mobile apps. Essentially, the Java or Kotlin code written for Android will not run on an iPhone, and vise versa for Swift code. Therefore, the app must be developed twice and this doubles development time and cost.
2. Building a hybrid mobile app using well-known web technologies. Ionic is an example of this method of developing a mobile app. The ability, and the advantage of this method, is to only code once and deploy everywhere. While this theoretically sounds like a great idea, it fundamentally has disadvantages discussed below.
3. The third, and final, mainstream method of developing mobile apps is using UI Frameworks. Some examples of these frameworks include Flutter and React Native. The difference between these frameworks and hybrid app development is that these frameworks do not use a webview. This comparison is discussed below.

As mentioned, Ionic falls into the second category of options to develop mobile apps: building a hybrid app. This method of app development comes with its advantages and disadvantages. Namely, the advantages include:

1. Code reuse for multiple platforms. The ability to code once and deploy everywhere drastically decreases web development time. An Ionic app can be bridged to native app SDKs or be used as a progressive web app. This is where Ionic shines.
2. Ionic uses traditional web frameworks known in the community. Since Ionic is built on the Angular CLI, projects are built directly using Angular and resemble Angular projects in structure and files. In Ionic 4, the view components have adopted the StencilOne library[6]. This allows any framework to be used in conjunction with Ionic and has allowed Ionic with Vue, React, and Javascript.
3. Ionic components are well developed and versatile, thus making them popular to develop with. The ability to easily implement pre-built web components makes the Ionic framework powerful for web and native compiled development.
4. Ionic shines with testing in browser. Other frameworks, such as React Native, require an emulator to run and test the mobile app. This limits the functionality without proper tools built into the emulator. Since Ionic apps are just web applications, a developer can run and test their application in any web browser. Additionally, any tools available in the browser for developers can be used to debug and audit the application. Having tools to test bandwidth availability, location overrides, and mobile functionality are essential to planning for the unique considerations in mobile development.

[5] More Information: https://itnext.io/should-you-use-ionic-4-fa04daebaffd
[6] More Information: https://ionicframework.com/blog/introducing-stencil-one-1-0-0/

However, these advantages come at a cost. The Ionic disadvantages hinder the Hybrid frameworks ability to stay mainstream in the app development industry. Most prominently, these disadvantages include:

1. Since Ionic allows for Angular code to develop mobile applications, Ionic applications are in reality just a web app. With that, they are wrapped in what is called a WebView to be formatted for mobile devices. Since it is just wrapped in a web view, an Ionic app cannot natively access phone components, such as Geolocation. To accomplish a connection to a phone natively, a bridge must be used. Ionic uses the Apache Cordova framework to facilitate this connection.
2. Using a bridge, such as Cordova, limits performance ability and speed of the application. Being separated from the native SDK makes development challenging, especially for debugging. Mobile apps are unable to access the specific native components without the bridge. Having more elements in the development allows more things to break. With Angular and then the Cordova bridge, there are more points for there to be faults in compilation, therefore making the Ionic framework more unreliable than native frameworks.
3. Lastly, as referenced above, the drawback that Ionic has is the lack of ability to interact directly with the native SDKs. Not being able to directly interact with native SDK code may hinder the performance of the application.

These two disadvantages pose major issues to the Ionic framework. Because of the rapid growth of native framework solutions, such as React Native, NativeScript, and Flutter, the decline of hybrid built mobile applications may be accelerating. The need in the development community for Hybrid development may cease to exist in the future. Ionic is seeing this industry change and is working on its own open-source Cordova bridge replacement named Capacitor. With Capacitor, there are no Cordova imports and it can be used to develop native apps without the Ionic framework[7]. With Ionic's move to Capacitor, it stops depending on old technology and creates a more reliable platform for cross-platform developers.

Ionic shines with this cross-platform development. Jeff Delaney, a Google developer expert for Firebase and consultant specializing in progressive web apps, mentions there is one specific case in app development where Ionic is best suited[8]:

- In an environment where the primary focus of the app is as a progressive web app and the secondary focus is as a mobile application. Since most modern PWAs are developed using React or Angular, using these frameworks with Ionic allows access to Ionic's functional and reliable web components. Additionally, web developers can cross-platform develop and launch their applications on mobile platforms without additional development costs.

While Ionic is an option for developing mobile applications in 2019, it is far from the industry standard, or even the recommended framework, for that matter. If the mobile apps use case is only for mobile deployment, Ionic's hybrid model may pose performance issues that can be easily rectified with other frameworks such as React Native. The move for Ionic to develop its own open source bridge, Capacitor, is a step in the right direction to keep Ionic in the considerations of developers.

[7] More Information: https://link.medium.com/ilXaDurI5Y
[8] More Information: https://itnext.io/should-you-use-ionic-4-fa04daebaffd

**UX/UI Decisions**

For the mobile app, there were four UX/UI considerations. Two of the UI decisions have two alternatives and two of the decisions have one alternative. The main crux of the mobile app is to limit the amount of barriers of entry to users. There is no log-in screen that users must use, results should be a main and upfront focus of the app, and information should be relevant and accessible at the users fingertips.

*Place Details Page - Swipe Out / Item Button / Icon Button*

This UI decision was made for the `page-details` page and was regarding the proper way to display access to external information. Please see the three alternatives below:

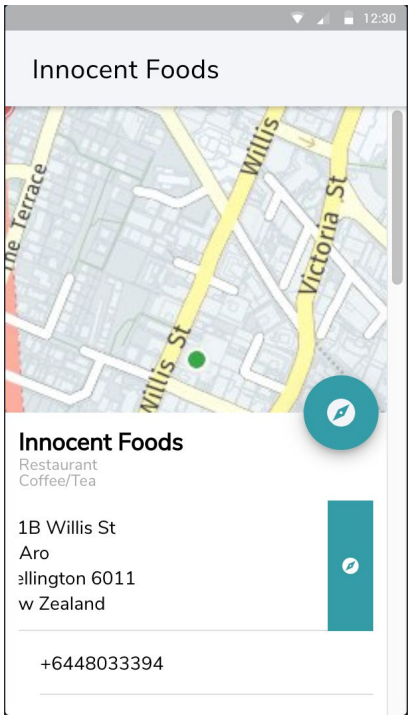



Image 1a
Current UI

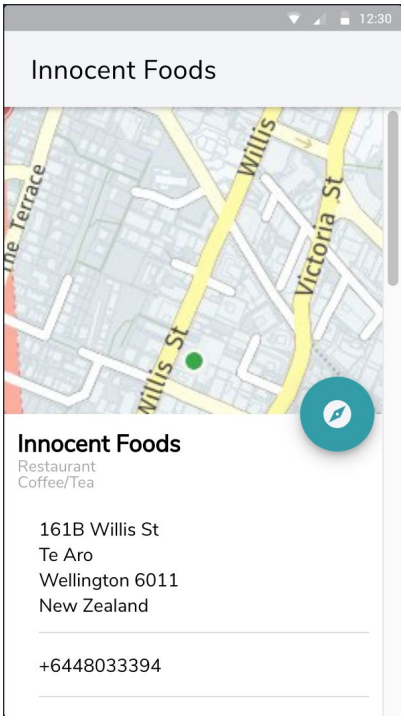



Image 1b
List Item Button Alternative

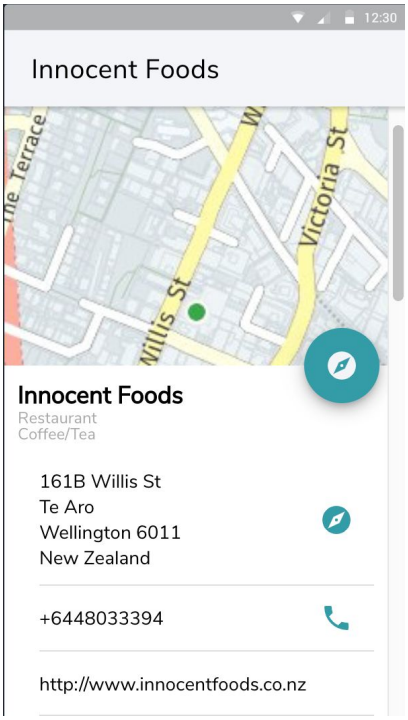



Image 1c
Icon Alternative

*Discussion of Current UI*

In the above images, the problem the app is attempting to solve is how to best display access to external information, This includes linking to external mapping application, such as Google Maps, opening up the default phone application for phone calls, or accessing a webpage via the default browser. All of these options are external and are not built into the app. Therefore, when opening these external applications, there is an excessive amount of time and CPU processing power required. This process can sometimes slow down a users phone, potentially locking it up. Additionally, this app's target audience are people visiting a place that they are unfamiliar with. Mobile must make special considerations since devices often only hold a part of a person's attention (Luke Wroblewski, 12)[9]. Because of this, the mobile app is designed to minimize clumsy and accidental clicks that slow the user down or open external apps. The sliding cards that reveal more options proved to be the best alternative.

[9] See book: https://static.lukew.com/MobileMultiDevice_LukeWsm.pdf

*Analysis of List Item Button Alternative (Image 1b)*
Image 1b shows the alternative of just having list based buttons. The user could click anywhere on the list item to open up the external component. It is challenging to display using this medium, however the list items are in fact buttons. This is counterintuitive to the problem described above about people only dedicating their partial attention. Additionally, users utilize the same screen real-estate to scroll. Users would be more prone to accidentally pressing a button under this model. To further the case against this UI decision, there are certain list items where more than one action might exist. See the table below for some examples of this:

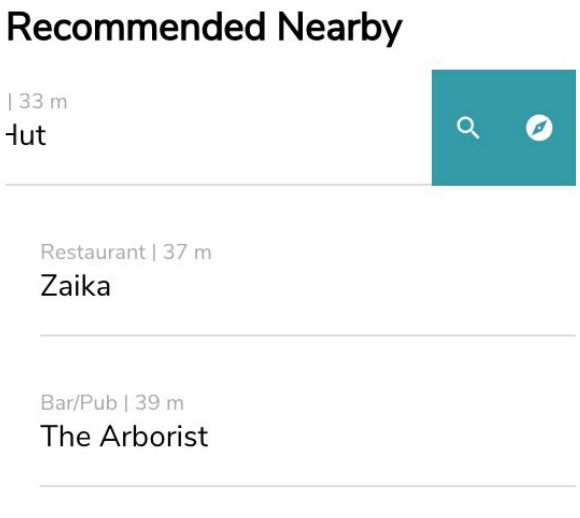



Image 1d
Recommended Nearby list items have two options

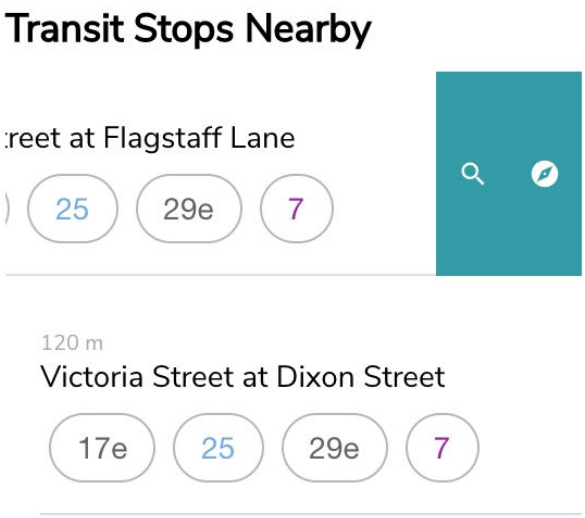



Image 1e
Transit Stop Nearby also have two options

The images above show two areas on the same `places-details` page where there are two options for each list item. In both cases, users can "Search" about the location, i.e. "Learn More", or users can "Navigate" there using an external application. In the case where no icons are present and there is no sliding options feature, there is only one decision the user can make, therefore limiting their accessibility and the applications usefulness.

*Analysis of the Icon Alternative (Image 1c)*
In this UI alternative, the users are instead presented with icons on the right-hand side of each list item. These icons are the exact same in style as the slideout icons, but with inverted color. This alternative presents a few issues with screen real-estate. First, the icon and the label must compete for space. For longer information items, such as a web address, the competition would limit the view of information and accessibility of the user. Image 1c shows just this where the web address overflows in the icon space, therefore forcing the icon to wrap to an unseen line. UI externalities, such as this one, pose an issue to the uniformity and presentation of the app. A second screen real-estate issue would be posed with multi-option list items. These are described in the paragraph above and would further have issues with space competition against the actual text of the page.

*Place Details Page - User Training*

In order to ensure users understand to slide out options presented above, the app implements Luke Wroblewski's suggestion of "Just In Time Education" (Luke Wroblewski, 26). With this, users are presented with an overlay and demo of the card sliding on their first time opening a place detail page. See below for screenshots:

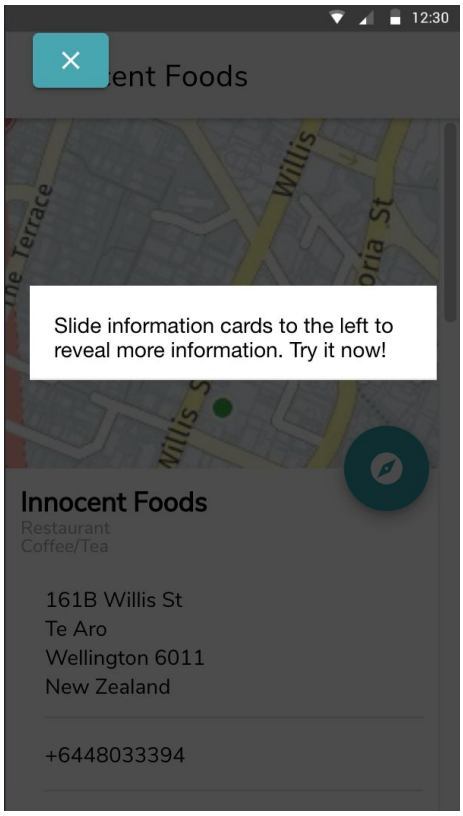



Image 1f
Overlay Card

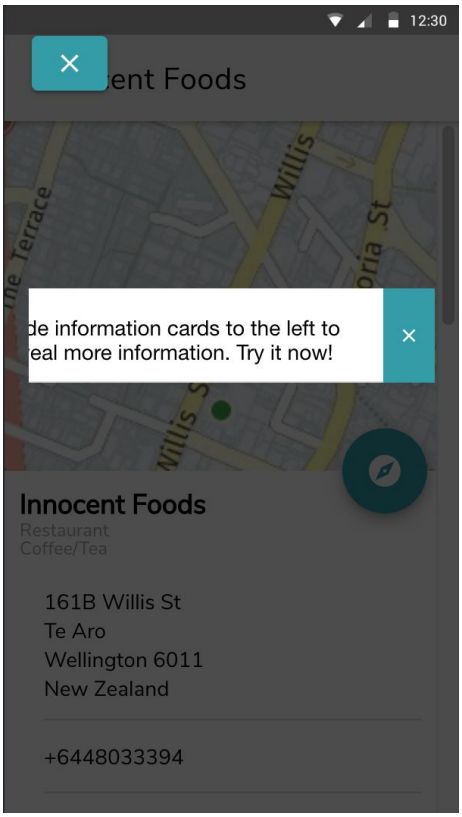



Image 1g
Overlay Card with Sliding Demo

This only displays on the first open of the page details page, Just in Time for the user to use the feature. The app uses this method over a tutorial at the beginning of the app. Luke Wroblewski explains in his mobile usability book, Mobile & Multi-Device Design, "Most people (sometimes over 90%) skip over intro tours as quickly as possible and those that don't rarely remember what they were supposed to learn." Therefore, the sliding Just in Time Education tutorial is useful to end users.

*Landing Page - Loading Methods*

This UI decision was made regarding how to show the user page content is loading.

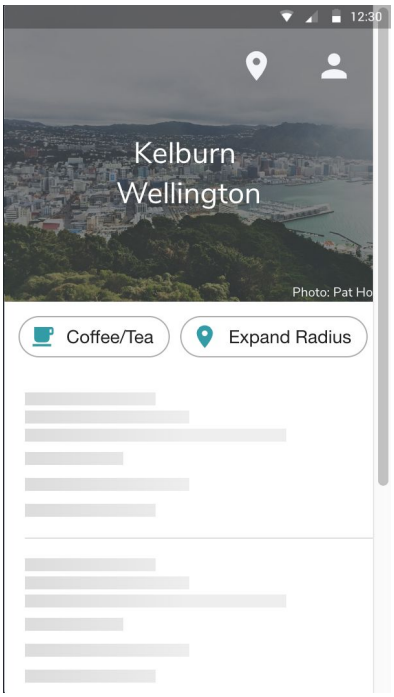



Image 2a
Current UI w/Skeleton Loading

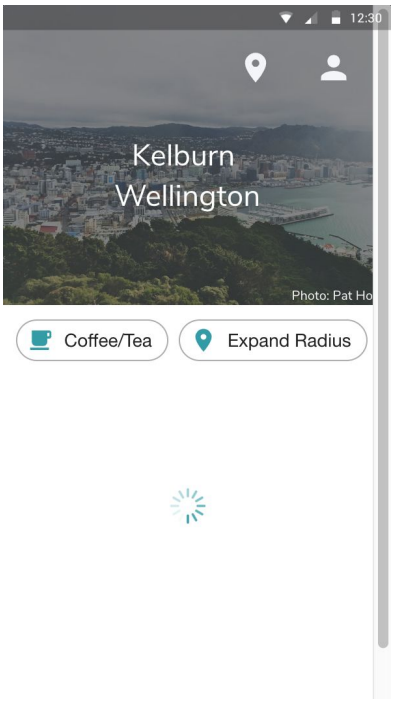



Image 2b
Spinning Lines Alternative

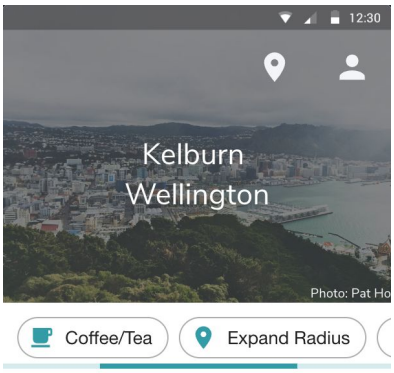



Image 2c
Progress Bar Alternative

*Current UI*

The current UI displays a skeleton loading UI where the place based cards will be generated. Luke Wroblewski explains in his book that the intention behind loading screens and progress bars are good, however, the result is rarely favorable. The loading bar disenfranchises the user base when content is generating. To solve this problem, skeleton loading has taken off as an alternative to the progress bars and loading circles. The skeleton screen gives the impression that the screen is building content and something is happening. Users value having the impression that the page is working. Thus, in this mobile app, the skeleton screen is used for loading information.

*Analysis of the Spinning Lines Alternative (Image 2b)*

In this example, a spinning loading circle is used in the center of the screen. Not only does this method go against the loading guidelines proposed by Luke, but it leaves empty space around the icon.

*Analysis of the Progress Bar Alternative (Image 2c)*

In this example, a progress bar is used that moves to the right from the left. While this option leaves empty space below the bar, it could be filled with other static content in the future. However, for this iteration of the app, and the wish for a lack of loading bars, this option is not ideal. Additionally, the loading bar may confuse users into thinking it is instead a scroll bar for the filter chips above. Scroll bars on mobile are generally against the UI standards.

*Landing Page - Location Search*

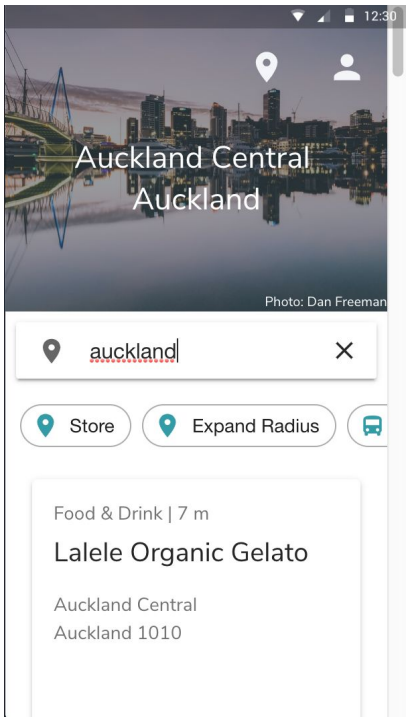



Image 3a
Current UI

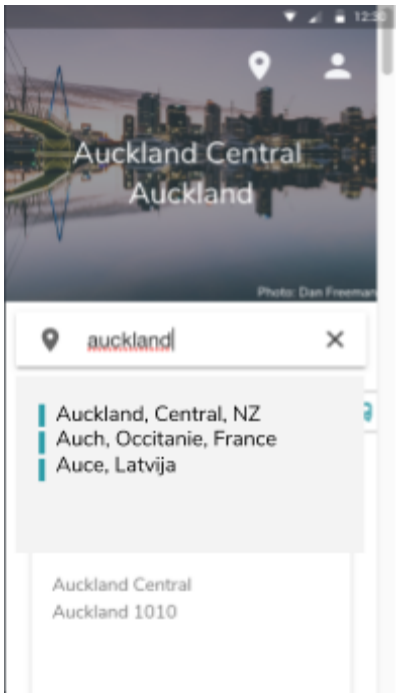



Image 3b
Mockup - Selection List

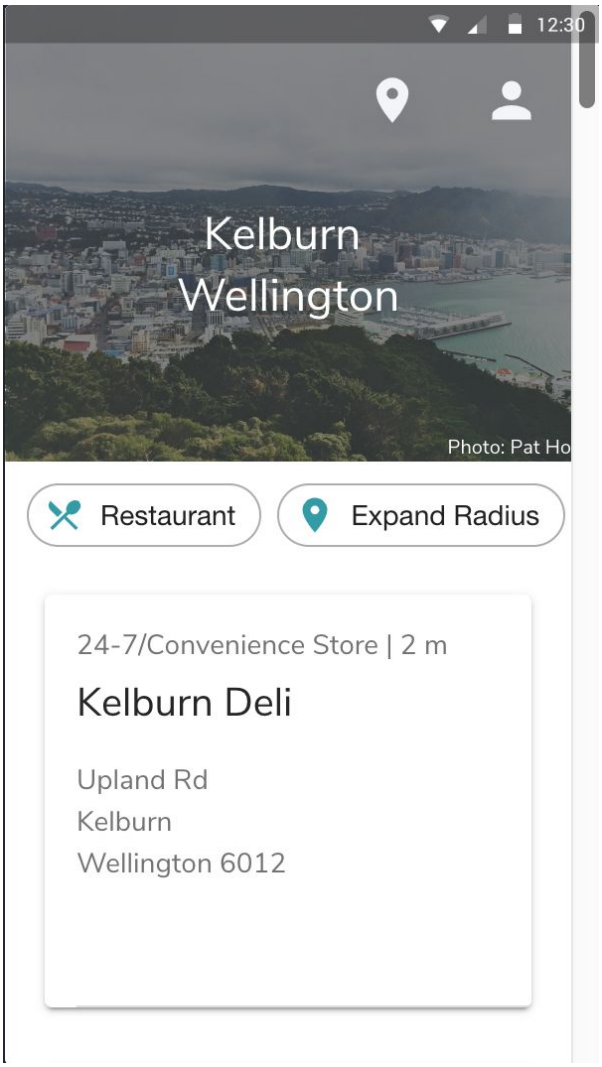
*Current UI*

The current UI is listed on the left-hand side and shows the toggled location search bar. This is where users can input a specific location and refresh the results to show places near the geocoded search query. In the example above, the user had entered “auckland” into the search bar. The current UI automatically refreshes the results view based on the geocoded location and result. Therefore, if the user inputs any query, the view automatically refreshes. The reasoning behind this is to give the user the appearance of speed and access. There is no middle step to completing a location search and no secondary taps. The information is available when the user needs it.

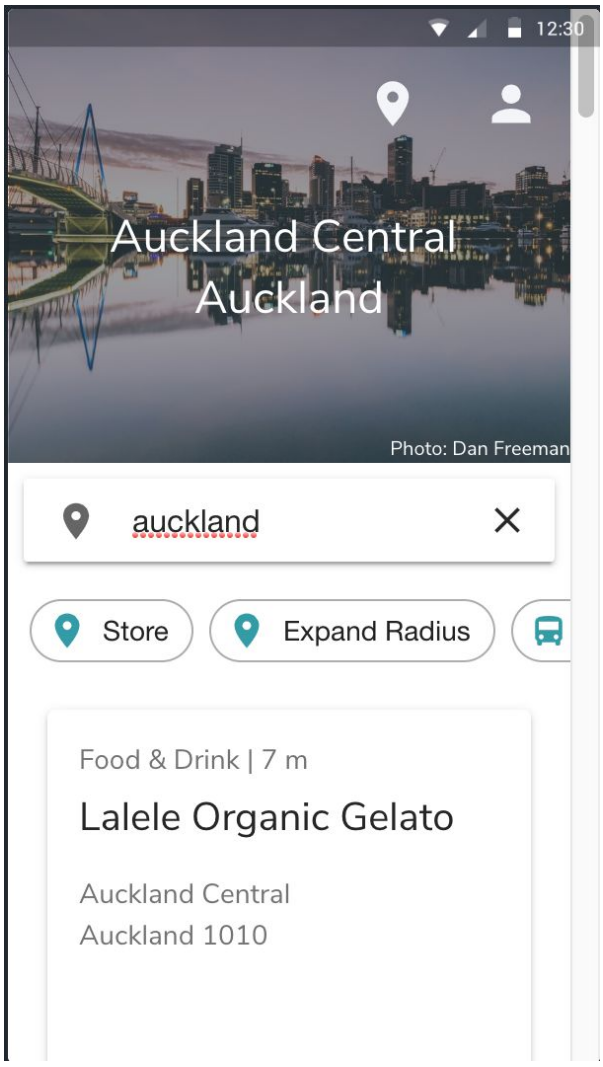
*Analysis of the Selection List Alternative (Image 3b)*

In this mockup, the user is presented with options based on their search query. The list is populated with as many results that are returned from a geocode request with the query. This would allow the user to select their choice. However, the autocomplete suggestion list adds the extra barrier that the users must press to access information. This is counter-intuitive to the main function of the app: to provide relevant place information at that moment. The design decision to not have a login for the mobile app supports this decision. The mobile app attempts to limit the barriers of entry required by the user to start using it. Having another tap to select an item hinders the instant access to information available with the instant loading of results.

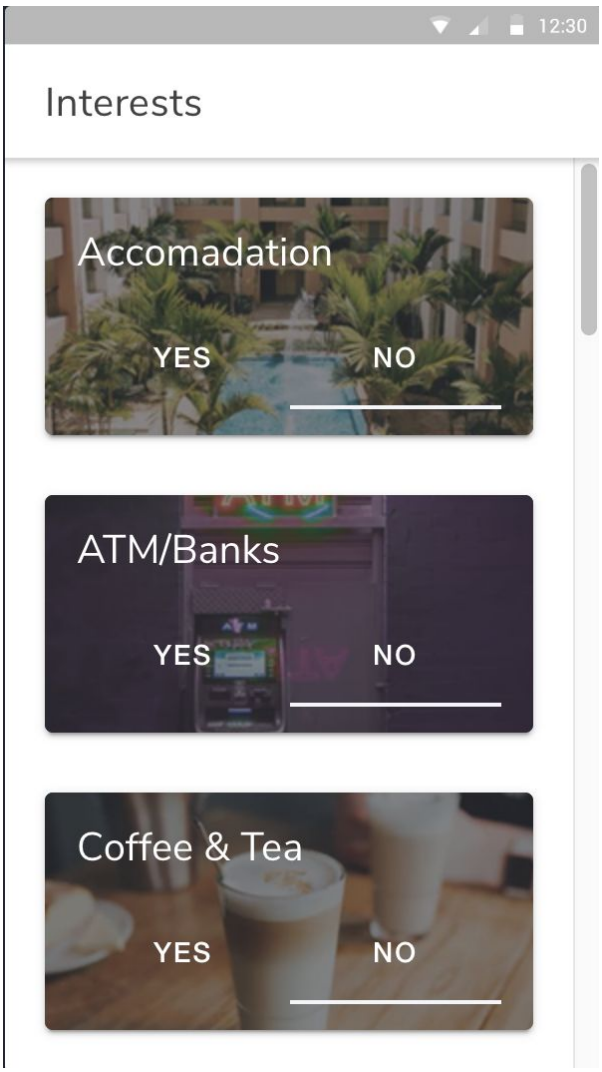
**Appendix**



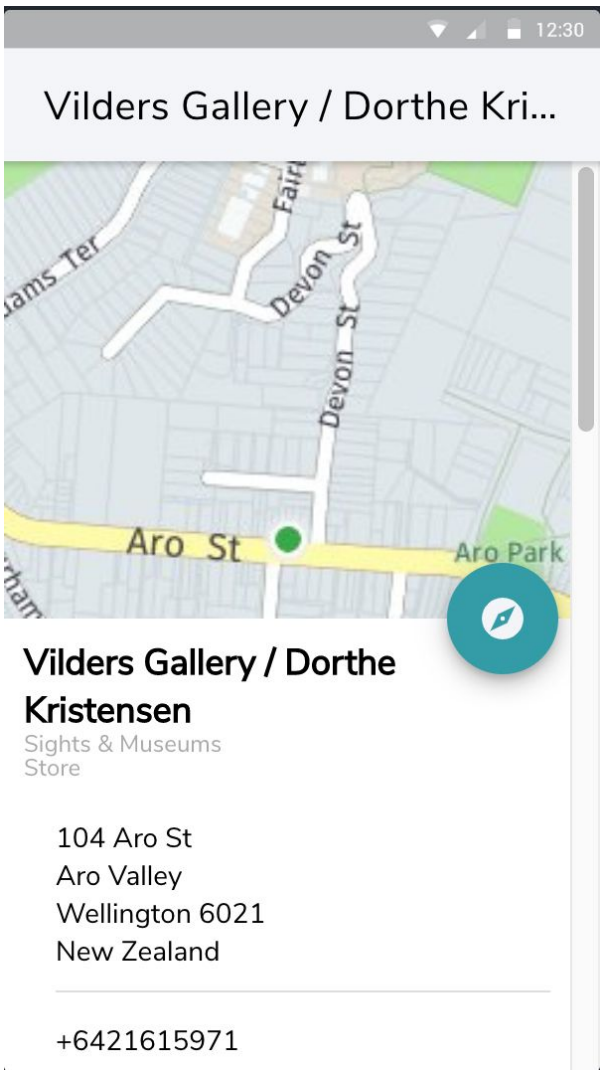

Landing Page
This is the main page for the app. The infinite list is populated based on the user's location and interests. Chips are generated directly above the results to allow for list filtering. The chips are on a horizontal slider. The header image is queried using the users location. Settings and location search are available using the upper right buttons.



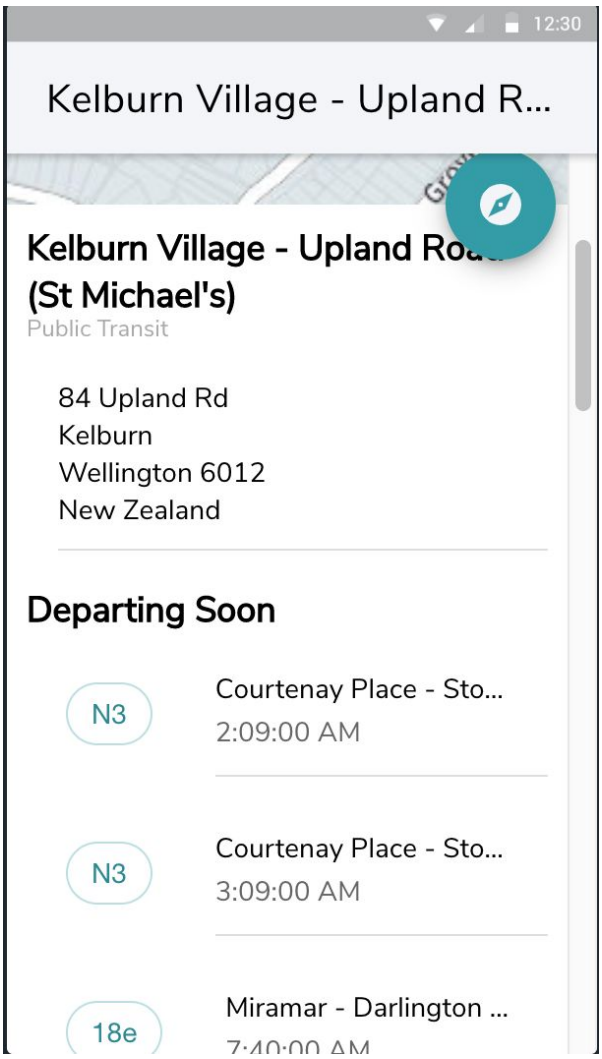

Landing Page - Search Toggled
This is the same landing page with the search toggled on and a query entered. The page is live-reloaded to refresh to the users request. The filter chips are changed based on their surroundings.

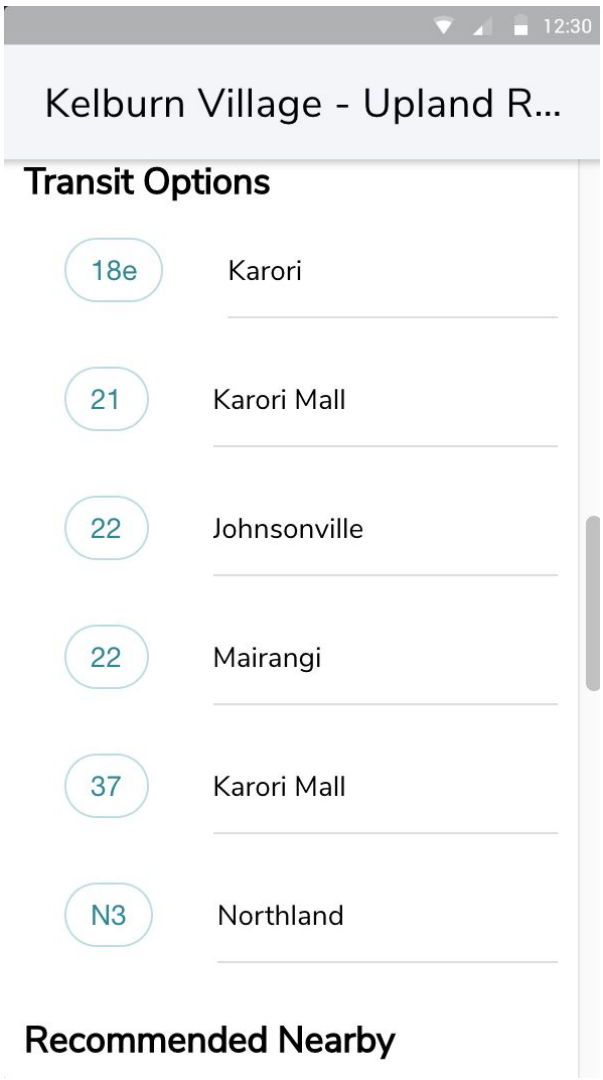



Interest Page
This page shows a list of interests with photos for the user to choose from. The slider toggle can either be in the yes or no position. Upon a users first use, the selection is defaulted to yes. This list is populated down for ~15 categories.




Place Details Page - Place
This page shows information using the “Place” layout. The view is different using the transit layout, as shown below. Users can navigate to the destination using the floating action button in the upper right. The list items are slidables, as described in the UI report above.




Place Details Page - Transit
Both of these screenshots show the transit layout of the place details page. Instead of showing business details, the page is altered to display transit information. Transit timetables are listed as well as information about line service to the station.