Design Details

SIXTYHWW SENG3011 Report

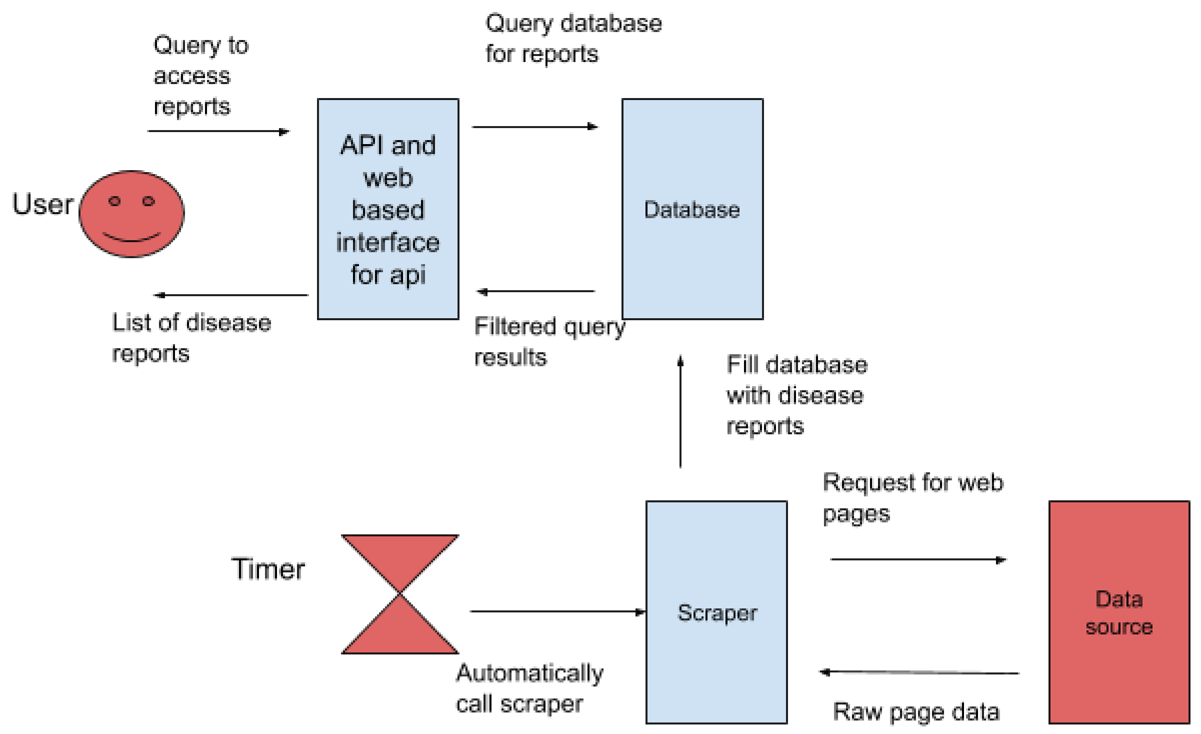
Nikil Singh (z5209322)
Joshua Murray (z5207668)
Tim Thacker (z5115699)
Daniel Ferraro (z5204902)

1. Describe how you intend to develop the API module and provide the ability to run it in Web service mode

Technology Stack:

Frontend
React JS
Backend/API
Express
Node
Database
SQLite3
Scraper
Cheerio
Axios
Node
Hosting
AWS

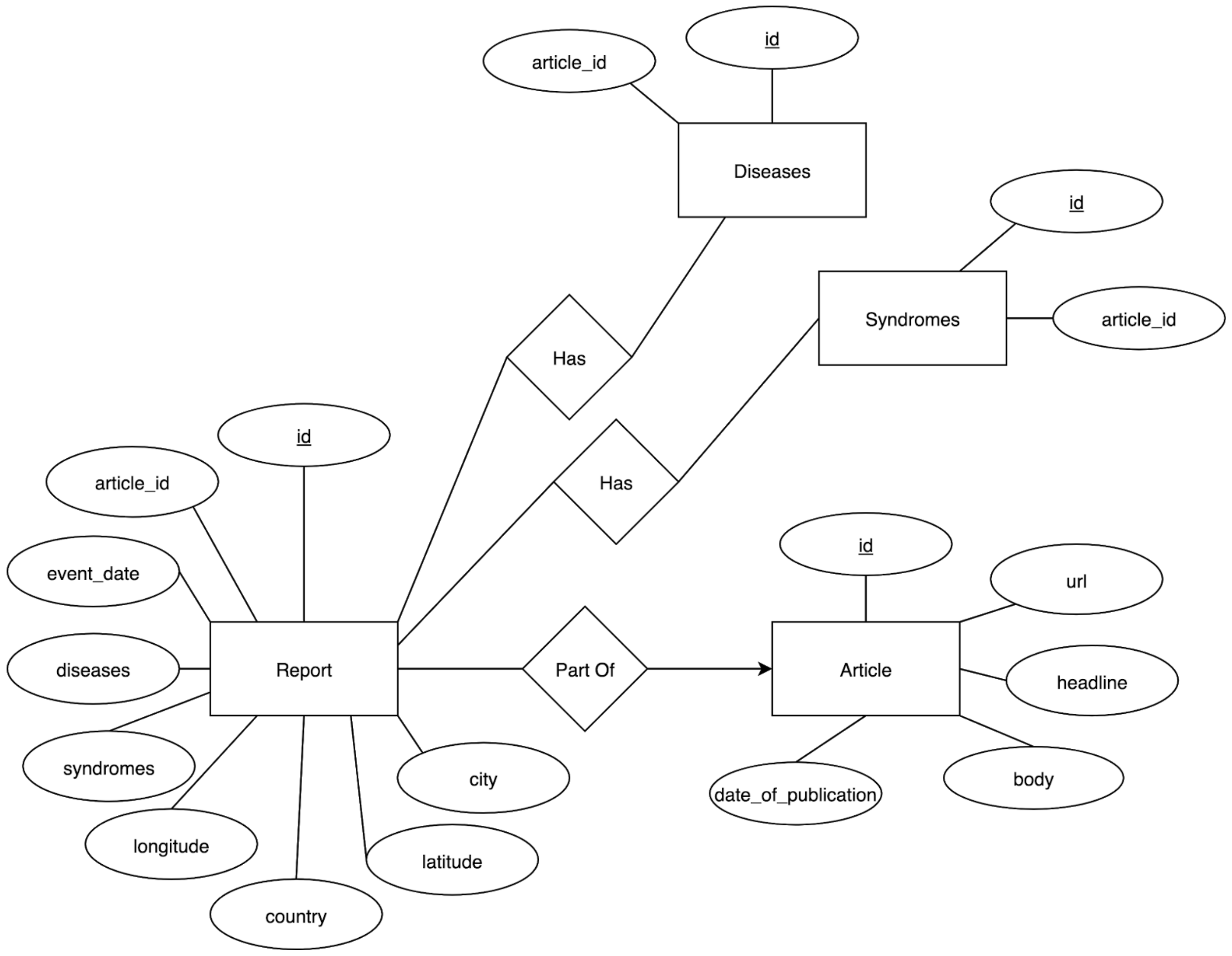
Software Architecture:



Database:
The database uses the following schema,

```
CREATE TABLE IF NOT EXISTS articles (
 id                  integer primary key autoincrement,
 url                 text not null,
 headline            text not null,
 body                text not null,
 date_of_publication date not null
);

CREATE TABLE IF NOT EXISTS reports (
 id         integer primary key autoincrement,
 article_id integer not null,
 diseases   text not null,
 syndromes  text not null,
 event_date date not null,
 country    text not null,
 city       text not null,
 latitude   text not null,
 longitude  text not null,
 foreign key (article_id) references Article(id)
);

CREATE TABLE IF NOT EXISTS disease (
 id         integer primary key autoincrement,
 article_id integer not null,
 foreign key (article_id) references Article(id)
);

CREATE TABLE IF NOT EXISTS syndrome (
 id         integer primary key autoincrement,
 article_id integer not null,
 foreign key (article_id) references Article(id)
);

CREATE UNIQUE INDEX articles_idx ON articles (url);
CREATE UNIQUE INDEX reports_idx ON reports (diseases, syndromes, event_date, country,
city, longitude, latitude);
```

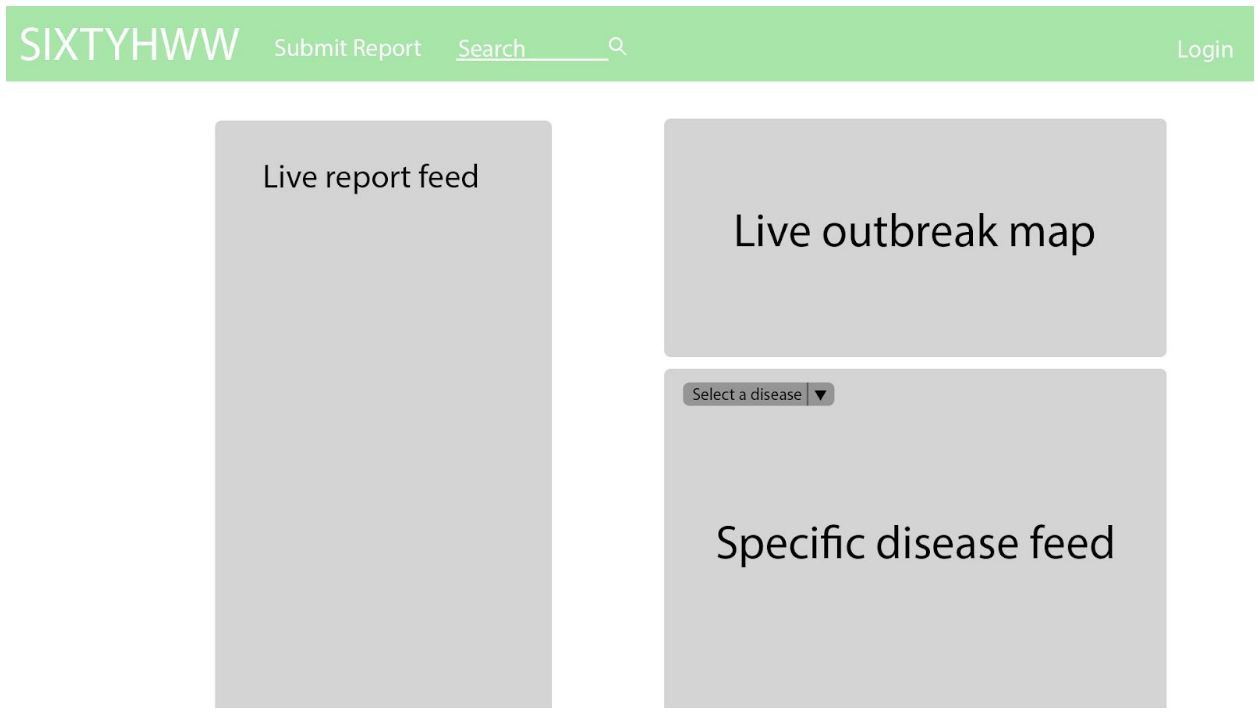
The respective ER-diagram for that schema is as follows:



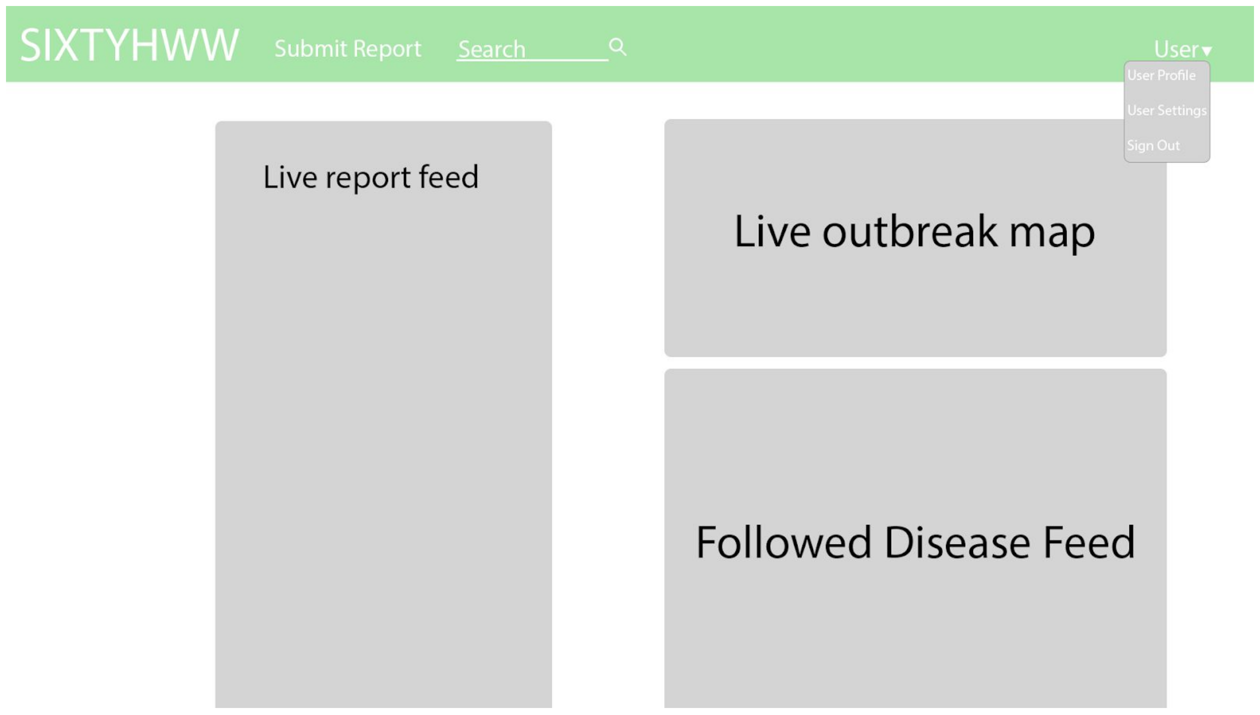
The reasoning for this design of the database is it allows a report to be associated with an article, which in turn allows it to be linked to an article if need be. The database also allows the scraper to collect data passively and have a place to store it. Diseases are made into their own entity so for future iterations we can allow users to subscribe to a specific disease. This in turn would populate the users specific feed with reports on that disease. Syndromes are given their own entity as they will be continuously cross-referenced and it allows the list of syndromes to grow in future iterations. Furthermore, the API will communicate with the database to insert data and retrieve required data.

The following are prototype designs for the web pages itself. Any card marked “feed” will contain cards following the user feed card design prototype, carrying relevant information.

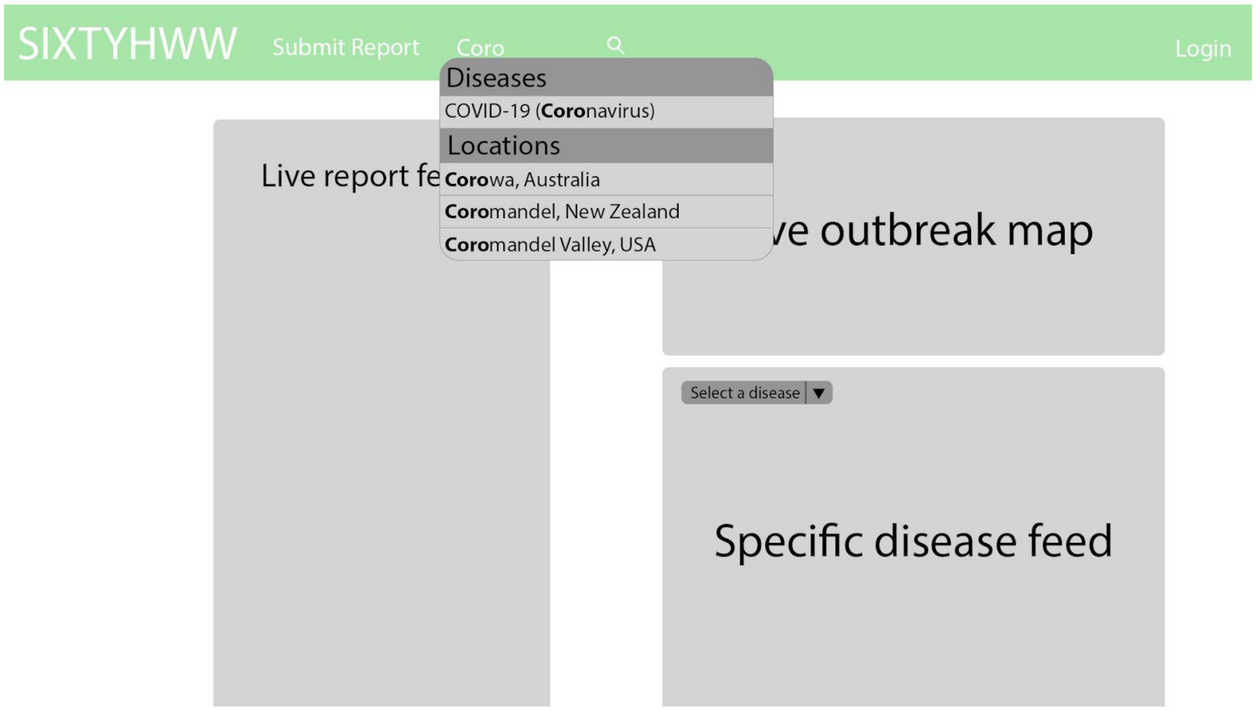
Web App landing page design when signed out:



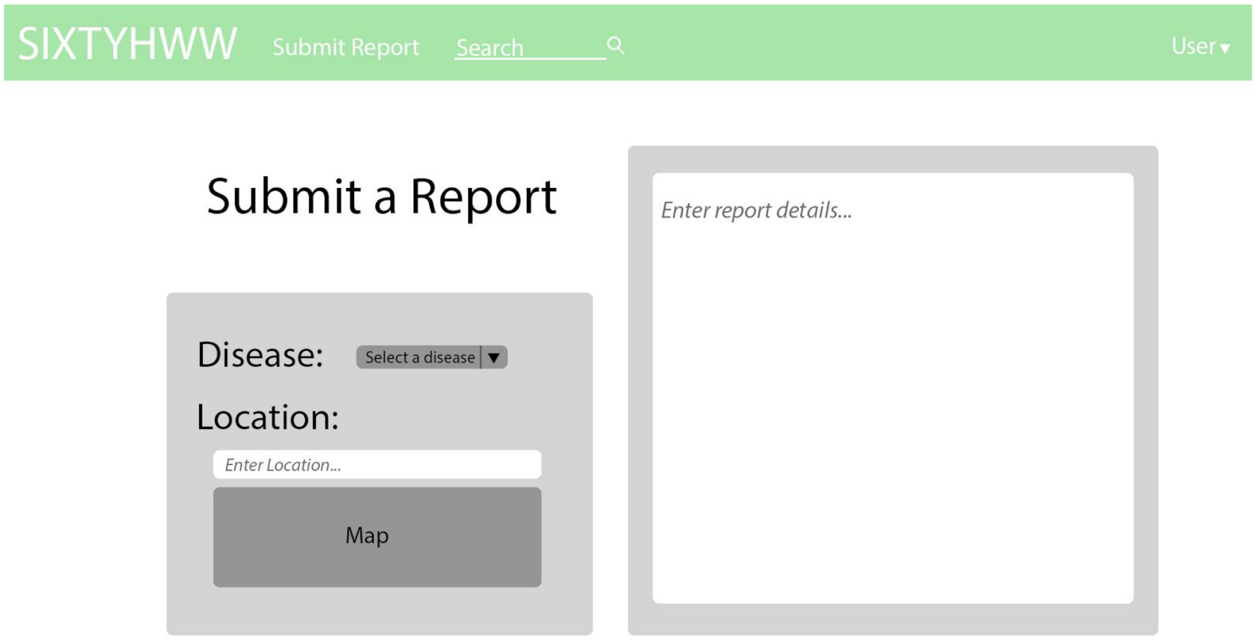
Web App landing page when signed in:



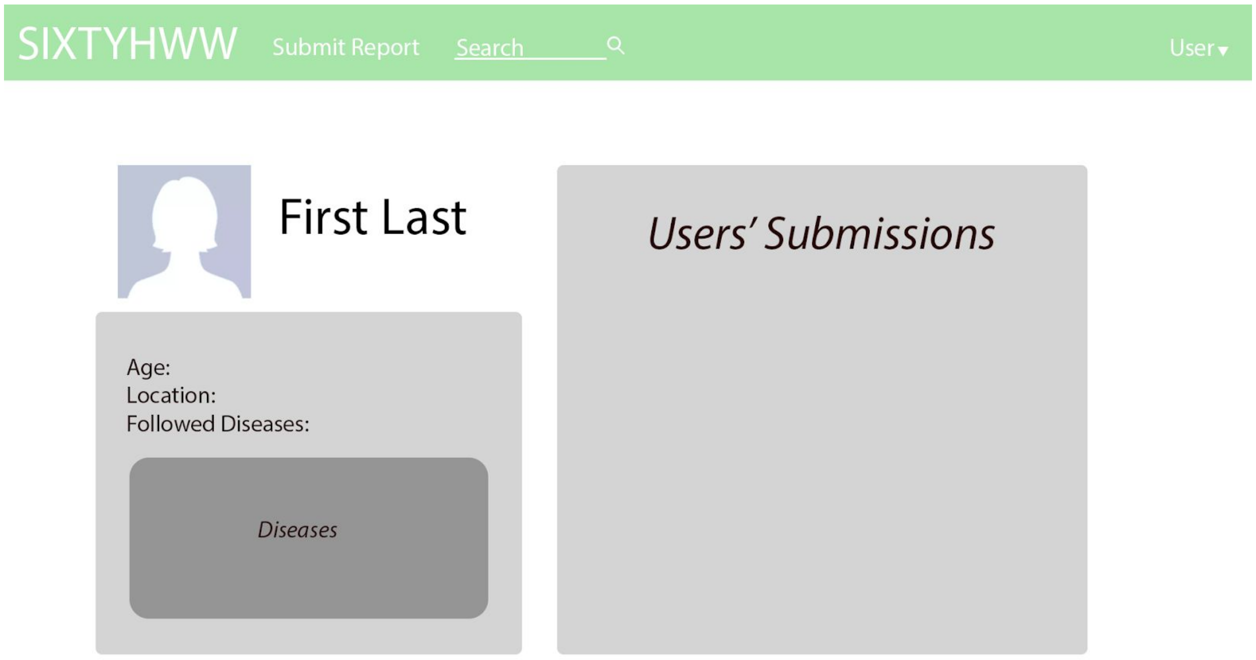
Search design:



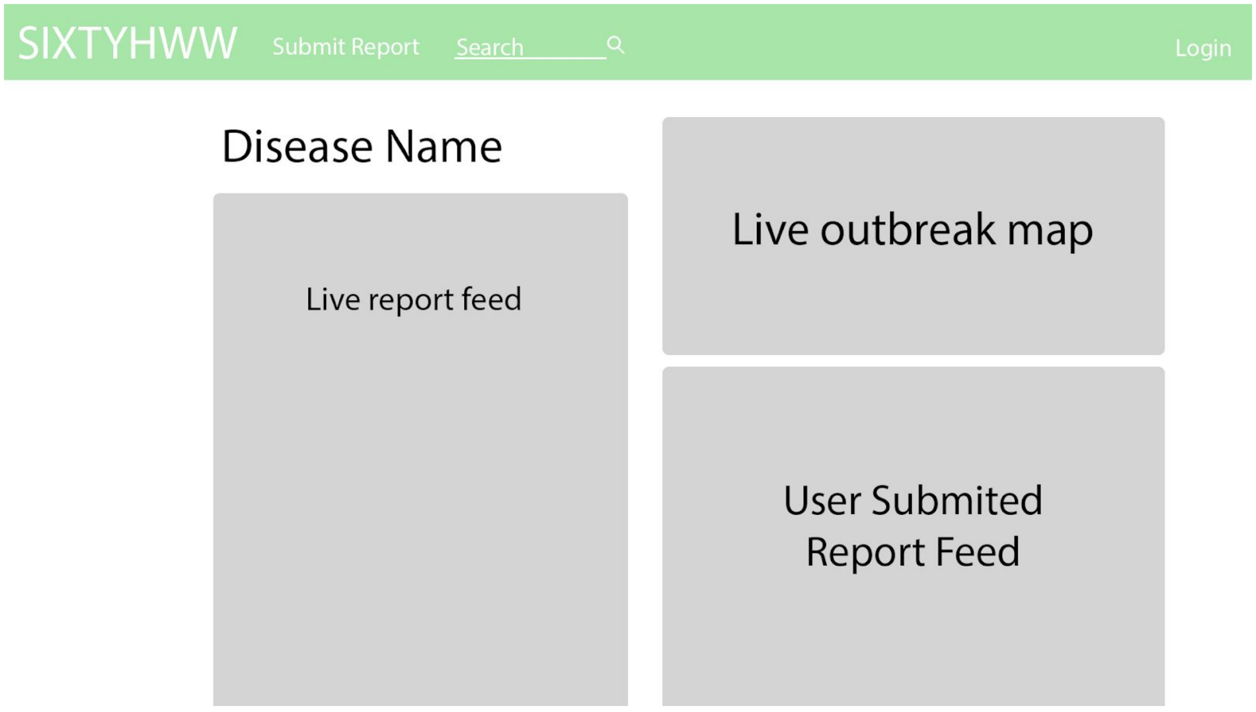
Submit a User report page design:



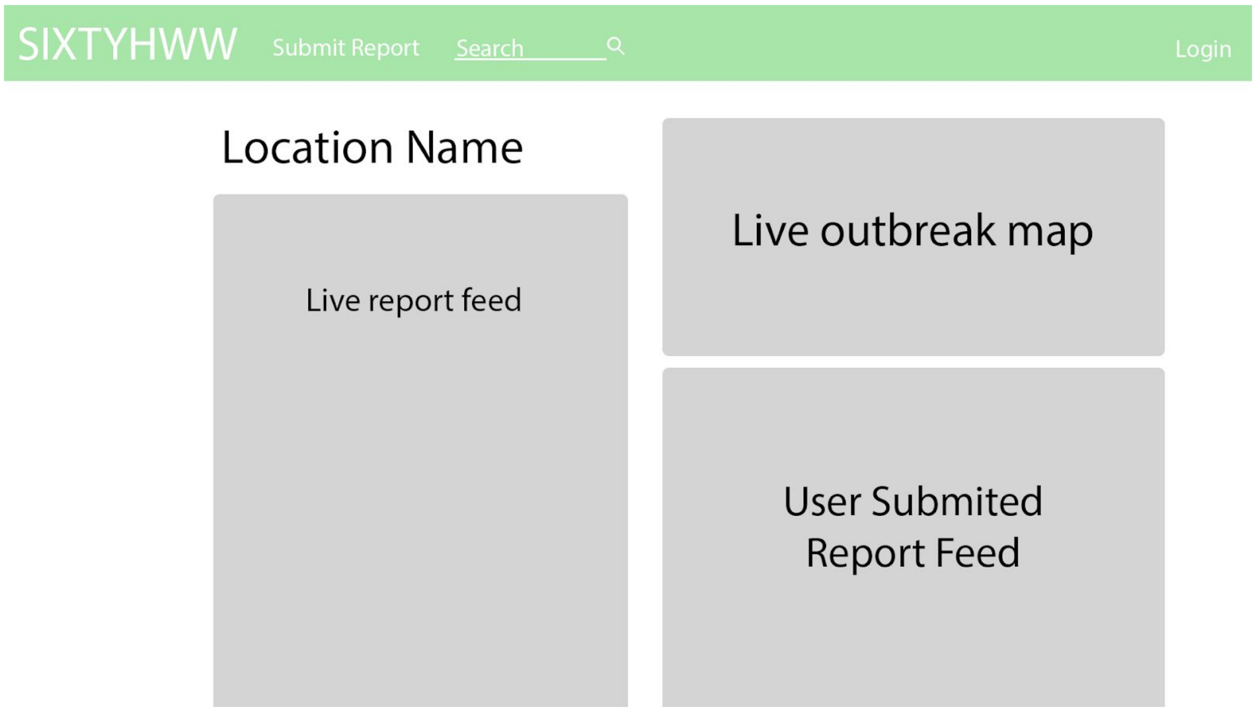
User Profile design:



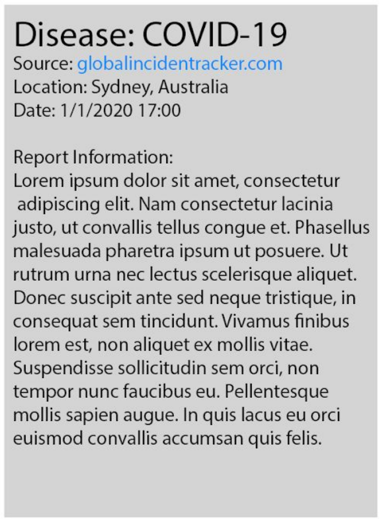
Disease “profile” design:



Location “profile” design:



Web app user feed “card” design:



Disease: COVID-19
Source: globalincidentracker.com
Location: Sydney, Australia
Date: 1/1/2020 17:00

Report Information:
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nam consectetur lacinia justo, ut convallis tellus congue et. Phasellus malesuada pharetra ipsum ut posuere. Ut rutrum urna nec lectus scelerisque aliquet. Donec suscipit ante sed neque tristique, in consequat sem tincidunt. Vivamus finibus lorem est, non aliquet ex mollis vitae. Suspendisse sollicitudin sem orci, non tempor nunc faucibus eu. Pellentesque mollis sapien augue. In quis lacus eu orci euismod convallis accumsan quis felis.

2. Discuss your current thinking about how parameters can be passed to your module and how results are collected. Show an example of a possible interaction. (e.g.- sample HTTP calls with URL and parameters)

Endpoint: /search
Method: GET
Arguments/Body:

Param	Restraints
startDate	must respect the following format: “yyyy-MM-ddTHH:mm:ss”. Input can NOT be empty.
endDate	must respect the following format: “yyyy-MM-ddTHH:mm:ss”. If input is left empty, defaults to current date
keyTerms	This input contains a comma separated list of all the key terms you want to get news about. This input can be empty or omitted in the case where the user doesn’t want to restrict his search.
Location	The user should be able to search disease reports by a location name (city/country/state etc.), which is a string to be matched with the content in the disease report. This input can be empty or omitted in the case where the user doesn’t want to restrict his search.

Return:
List of filtered articles based on the supplied arguments, or 4XX Error if arguments not formatted correctly

Examples:
```
/search?startDate=2020-01-01T17:00:00&keyTerms=coronavirus&location=sydney
/search?startDate=2020-01-01T17:00:00&endDate=2020-03-01T17:00:00&keyTerms=coronavirus
/search?startDate=2020-01-01T17:00:00&endDate=2020-03-01T17:00:00&location=sydney
/search?startDate=2020-01-01T17:00:00&endDate=2020-03-01T17:00:00&keyTerms=coronavirus&location=sydney
```

Endpoint: /articles
Method: GET
Arguments/Body: N/A
Return: Returns 20 most recent articles

Example:
/articles

Endpoint: /articles
Method: GET
Arguments/Body:

Param	Restraints
N	Number of articles to grab, sorted by most recent. Maxed at 50.

Return:
N most recent articles sorted by data

Example:
/articles?n=20

Endpoint: /articles/{id}
Method: GET
Arguments/Body:

Param	Restraints
id	int id of article to grab

Return:
single article where id is {id}, or 4XX Error if not found.

Example:
/articles/1000

Endpoint: /articles/{id}
Method: DELETE
Arguments/Body:

Param	Restraints
id	int id of article to delete

Return: n/a if id exists, or 4XX Error if id not found
Example:
/articles/1000

Endpoint: /articles/{id}
Method: PUT
Arguments/Body:

Param	Restraints
ID	ID of the article to update. Cannot be empty
startDate	must respect the following format: “yyyy-MM-ddTHH:mm:ss”. Input can NOT be empty.
endDate	must respect the following format: “yyyy-MM-ddTHH:mm:ss”. Input can NOT be empty.
keyTerms	This input contains a comma separated list of all the key terms you wish to modify in the article. Can be omitted
Location	String containing the updated location of the report. Can be omitted

Return:
n/a for a successful update, 4XX error on missing id or malformed arguments.

Examples:
/search?id=1000&startDate=2020-01-01T17:00:00&keyTerms=coronavirus&location=sydney
/search?id=1000&startDate=2020-01-01T17:00:00&endDate=2020-03-01T17:00:00&keyTerms=coronavirus
/search?id=1000&startDate=2020-01-01T17:00:00&endDate=2020-03-01T17:00:00&location=sydney
/search?id=1000&startDate=2020-01-01T17:00:00&endDate=2020-03-01T17:00:00&keyTerms=coronavirus&location=sydney

3. Present and justify implementation language, development and deployment environment (e.g. Linux, Windows) and specific libraries that you plan to use.
API: Our API will be written in node.js using the express framework and the json body-parser library. The API requests will be made through JSON data in the body of the request.

Scraper: Our web scraper will be built in node JS using the cheerio library. We chose this as the library is easy to use, relatively lightweight, and it makes it extremely simple to create JSON objects from the scraped reports from a JavaScript program. This means that it will work seamlessly with the API and database.

Database: Our database will be constructed using SQLite3.

Frontend: The frontend for our web app will be constructed using typical web programming languages such as HTML5, CSS and JavaScript.

Deployment: Our API will be deployed on AWS. This allows us to easily host our API on the internet, and run our scraper on a schedule for a price of $8. $7 was spent on domain registration and $1 for route and domain management. For local development, the API and the scraper will both be running on Linux operating systems.

Chosen Stack:

	Language/ Libraries	**Justification**
Development Environment	Linux (Chosen) Gentoo	This development environment is used by all members of the group. It is also the most suitable for software development as it’s run on the CSE servers.
	Windows	Only one member has Windows, while everyone could develop on a linux environment.
	MacOS	Only one member has MacOS, while everyone could develop on a linux environment.
Primary Language (Backend and Frontend)	JavaScript (Chosen)	All members of the group are familiar with this language. It has libraries for simplifying JSON object reading. Furthermore, it is more suitable for designing the web interface of the API compared to other choices.

	Python3	Although everyone in the group has plenty of experience using this, the libraries available in JS compared to those available in Python3 were simpler, easier to use and fit the specifications of the project better.
	Go	Majority of the members within the group do not have enough experience in developing with Go.
	C++	Only half the group has sufficient experience in developing with C++. Furthermore, web development in C++ is much more difficult when compared to the other languages shown above and has a larger learning curve.
Database	SQLite3 (Chosen)	We chose sqlite3 because it is lightweight and allows for fast development. Our current design with the scraper/api requires only trivial interactions with the database, and therefore we don't need a DMS that includes extra functionality. Additionally, the majority of the members within the group have experience using this.
	PSQL	PSQL provides more functionality than required for the scope of this project. Additionally, the libraries for PSQL in JS are more complicated and have a slightly larger learning curve in using it.
	MySQL	All the members of the group are not too familiar with MySQL. It requires a server to be installed and configured with systemd and requires authentication to be setup which slows down our workflow.
Frontend Framework	React (Chosen)	Has an extensive library for frontend development and is an industry standard. It has a large learning curve, but if learnt can be greatly beneficial. However, in terms of cost-benefits it is not feasible within the time frame. However, some members do have experience using React and there are far more learning resources.
	Vue	Has the smallest learning curve however no members have any experience in it. Furthermore, all members have experience in writing pure JavaScript so Vue could be learnt more quickly. Having a framework for frontend has more benefits than costs

		within the timeframe given.
	Angular	No members in the group have any experience with this. There is a large learning curve because of features such as typescript and different types of classes including modules, components, services, pipes etc.
Scraping/ DOM Parser	Cheerio (Chosen)	Dom parsing was the most applicable approach for scraping with global incident map due to the structure of the website. Therefore we decided to develop with scraper using cheerio as it is very well documented and easy to learn in a small time frame.
	jsdom	It is more complex as it allows for more control but is not worth the time to learn for the benefits it would provide. Additionally, the documentation was alright but would take time learning.
	htmlparser2	The documentation was not very good, and not worth the extra time to learn.
API Creation	Express JS (Chosen)	This is the standard lightweight framework for developing Node.js web applications. It is all we require for making a simple Restful API.
HTTP Requests	Axios (Chosen)	Has better documentation when compared to the other options. It is also simple to learn in the given timeframe.
	request	It is simple and has good documentation, but that documentation has depreciated which may cause problems. However, it has too many features that will not get used creating unnecessary overhead.
	got	Contains a lot of functionality that we don't require, which add too much overhead.
JS Database Library	Sqlite (node library) (Chosen)	This is an wrapper around SQLite3 and the calls to run queries return Promises which allows for the more modern async await syntax to be used instead of callbacks. Callbacks can get out of control with nesting so we opted for this option.
	Sqlite3	This is the official library for interfacing with sqlite3 databases. It has been contributed to by people such

		as Ryan Dahl who created Node.js and is the most used library for sqlite databases. It does however require the use of callbacks which are hard to maintain in large applications.
Server Hosting Site	AWS (Chosen)	AWS requires payment for continued use after a certain level of usage has been reached. It is also compatible with the rest of our stack and is more reliable.
	Heroku	Used to host the API online. Additionally, it is also free to host on for the scope of the project. Not compatible with the entirety of our stack.

Overall Justification of Chosen Stack
Linux was chosen as the development environment as each member has access to a Linux environment and is the most suitable for software development. Node JavaScript was chosen as the primary language for backend and frontend due member experience and was more suitable for API development. SQLite3 was chosen as the choice of database because of member experience and it provided the functionality required for the scope of the project. React was chosen for the frontend framework as some members had some experience using it and the availability of resources for learning it. Cheerio was chosen for web scraping as it was well-documented and easy to learn. Express JS was chosen for API creation due to it being simple for the creation of a Restful API. Axios was chosen for handling http requests because it has more documentation and was easier to learn in the given timeframe. For the JS Database Library Sqlite was chosen as it serves to be a wrapper around sqlite3 and allowed for the use of async functions. For server hosting site AWS was chosen because it was more reliable and provided compatibility with the entire stack.

General Guide in Testing of System
Note: More detailed version available in the Testing Documentation.
Database:
Each query will be tested individually. The database will be in the state of tables having no entries, all tables with entries, and a mix of tables with and without entries. The expected results will be calculated by hand, and then compared to the actual results for each query tested.

Scraper:
A subset of the datasource pages will be scraped by hand following the prescribed 'Manual Data Access Process'. The results of the handscraped data will be compared against the results of the scraper to ensure the data is being mapped correctly.

API:
Each endpoint for the API will be tested with various input parameters that covers the general set of inputs. The inputs used will be both valid and invalid. Furthermore, the API will interact with the database while in its various states. These states include tables having entries, no entries or a combination of both.

Frontend:
Each site page will be navigated to and ensured the data appearing will be correct. Will also test all forms of user input with valid and invalid data. Additionally, the feed will be tested for various users with different preferences.

Login System:
Different sets of username passwords will be tested to check if they are correct. The usernames and passwords will include valid and invalid inputs.

Challenges Addressed
One major challenge was using Heroku at first for hosting the API. This had issues with SQLite3 as the database would be deleted every 24 hours. The two options were to change the database to PSQL or change the hosting service. We chose to change over to AWS because it provides better compatibility with SQLite3. Furthermore, it was easier to change the hosting service than to refactor all the code to use PSQL instead. Furthermore, some of the functionality provided by PSQL was beyond the scope of this project.

Another challenge faced was learning to use Vue. The documentation was available but was not as easy to pick up as the documentations for React. Upon further discussion between members and their past experience with both frameworks, it was decided to use React instead. Although it was more difficult to learn, the members of the group had slightly more experience with it and it would potentially provide a better end result for the frontend.

Additionally, another challenge encountered was generating swagger documentation automatically against doing it manually. Some of the documentation was automatically generated but it did not provide enough details. As a result, swagger editor was used to add in all the required details which was extremely time consuming. The details added included information about parameters and response codes.

Shortcomings

- The data source does not fit the structure of the spec well.
- Scraper is slow.