Design Details

SIXTYHWW SENG3011 Report

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

Contents

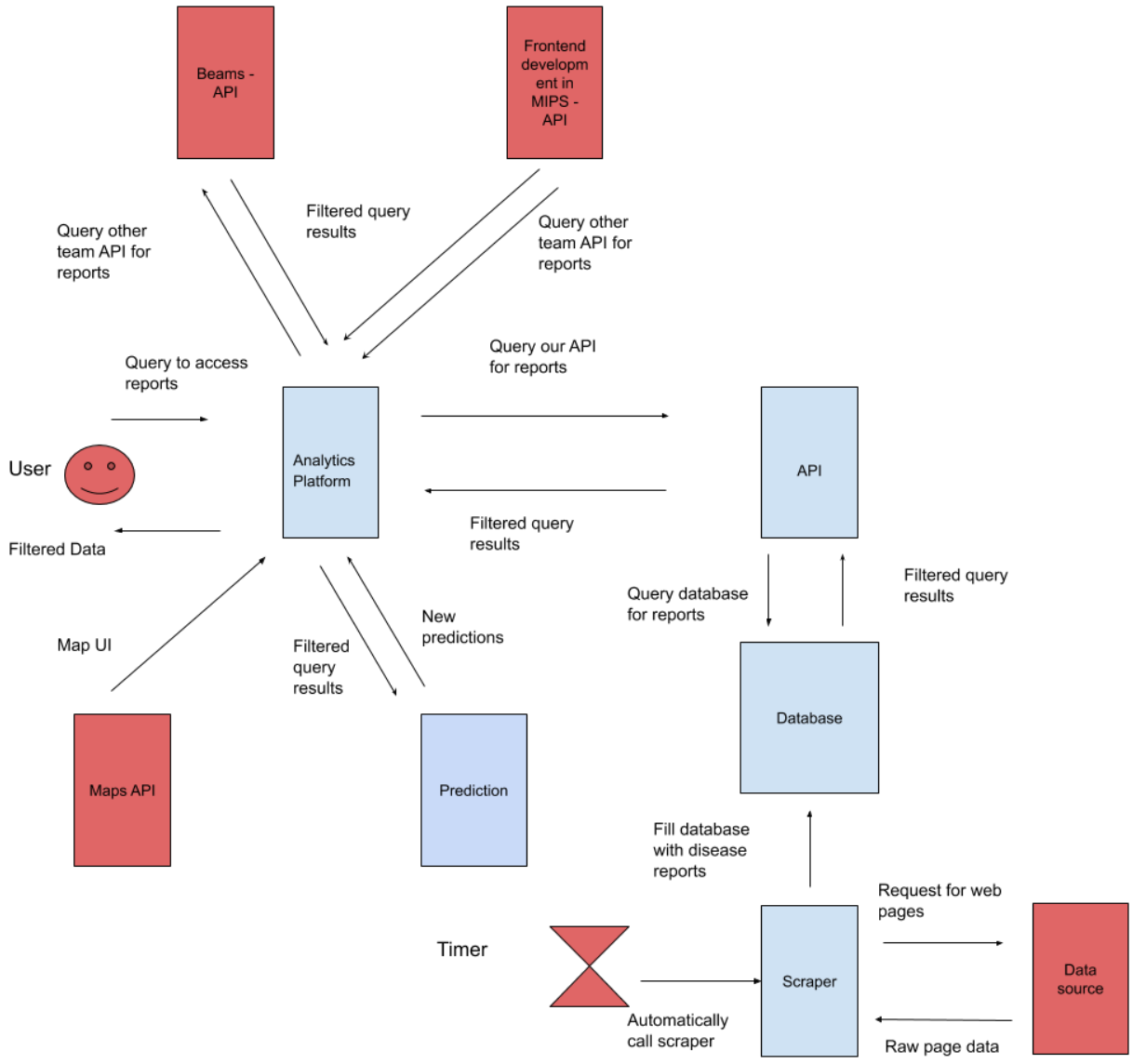
API and Frontend Development

The following describes our intention and method of developing the API module and how we will provide it the ability to run in Web service mode.

Final Technology Stack

Frontend
React JS
Google Maps API
Geonames API
Beams API (Group API)
Frontend Development in MIPS API (Group API)
Backend/API
Node
Express
Sqlite
Yamljs
Swagger-ui-express
Prediction
Express
Node
Regression
Database
SQLite3
Scraper
Cheerio
Axios
Node
Hosting
AWS

Software Architecture



Database

The database uses the schema found at Appendix A, and its respective ER-diagram for that schema is found at Appendix B.

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. Users and the diseases and locations they follow are separate from reports and articles as we use the API to retrieve information related to the data they follow to populate the feed. This allows for better compatibility when using other groups API as their semantics may slightly differ. It also decouples the frontend from the database. Furthermore, the API will communicate with the database to insert data and retrieve required data.

Prototype Designs for Frontend

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 can be seen at appendix C. This displays a live report feed, a live outbreak map and a specific disease feed. The specific disease can be selected.

Web app landing page when signed in can be seen at appendix D. This displays a live report feed, a live outbreak map and a followed disease feed.

Search design can be seen at appendix E. This displays a auto-fill in drop menu with potential results in terms of diseases and locations.

Submit a user report page design can be seen at appendix F. This displays a section to fill in information about the disease and location, along with a section to fill out information for that report.

User profile design can be seen at appendix G. This displays the users information, diseases followed and any report submissions.

Disease “profile” design can be seen at appendix H. This displays a live report feed, along with a live outbreak map and a feed containing user submitted reports related to that disease.

Location “profile” design can be seen at appendix I. This displays a live report feed, along with a live outbreak map and a feed containing user submitted reports related to that location.

Web app user feed “card” design can be seen at appendix J. This displays information about the report, such as source, location and date. It also displays a brief snippet of the details of the report.

API Inner Workings

The following discusses the current thinking on how parameters can be passed to our module and how those results are collected.

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

Stack Choice and Justification

The following presents and justifies implementation language, development and deployment environment (e.g. Linux, Windows) and specific libraries that were planned to use.

General Design

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.

Prediction: Will be implemented using Node JS and will use the regression library to perform the actual prediction.

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 $8.50. $7 was spent on domain registration and $1.50 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.
Prediction	Regression	It performs all the functions needed. The documentation is also adequate and simple 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 adds too much overhead.
JS Database Library	Sqlite (node library) (Chosen)	This is a 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. Regression was chosen for prediction as it provided exactly what was needed and was quick and easy to learn. It also provided for some flexibility in the type of equation produced.

Other API and Algorithms Used

Beams API

We chose this API primarily as it provided articles from a source different from ours (ProMed vs Global Incident Tracker). It also had endpoints that were well suited to conform to our webapp’s needs making it fairly easy to integrate.

Frontend Development in MIPS API

This API was chosen for similar reasons to Beam’s API. It was from a different data source (CDC instead of ProMed and Global Incident Tracker), and its endpoints were easy to use and add to our project).

Regression Library for Prediction

The library performs linear least-squares fitting to fit an equation to the set of data provided. This was chosen as it provided the exact functionality needed in terms of finding an equation and then performing the prediction. The documentation was adequate, and it was easy to use.

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.

Prediction

Extracting the actual data will be checked by hand using various countries and their reports. White box testing will be performed on the inner workings to ensure the correct data reaches the actual prediction making point by hand.

Challenges and Shortcomings

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.

Furthermore, with extracting data for prediction it was difficult to obtain exactly what was needed just from our data source. Also there would be gaps in information for certain countries due to the lack of information from the data source. To overcome these, any information where data could not be properly extracted would be ignored and areas with slight gaps in information would be filled in with averages based on before and after events for a particular date.

Shortcomings

The data source allocated did not provide much information about the actual reports, rather just a headline and a link to another webpage. This resulted in reports being an almost non-existent component of our dataset. This heavily impacted and limited our options on features our front-end could implement. Furthermore, the lack of actual report information impacted the prediction algorithm since there was less data to extract meaningful statistics from,therefore producing less accurate predictions that does not provide the full scope of a disease.

Our data source did not provide an easy means of accessing reports from the home page, so we needed to scrape the home page for article pages, and then scrape each article page. Scraping websites is inherently slow, and while this isn't a big issue once you have a large data set built up, it caused problems early in the development cycle. Whenever there was a bug that resulted in the database being reset, it would take a long time for the database to be refilled and usable again.

Another major shortcoming was that our system didn't allow for proper compatibility with other group API in terms of how their data is parsed and incorporated with all the features of our system. As a result, the data once incorporated did not fit exactly into the design layout for the feed when compared to our data source.

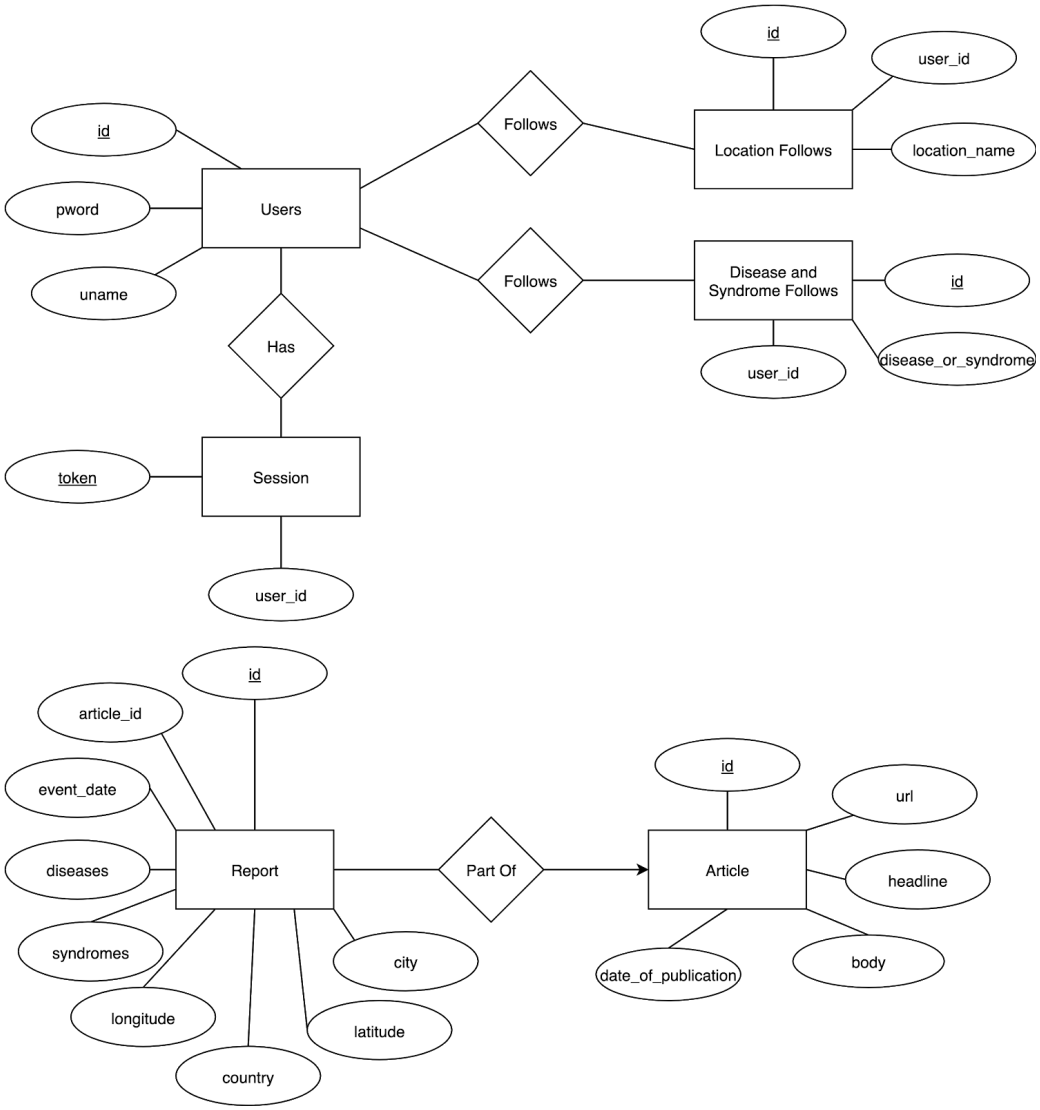
Appendix

A: Database Schema

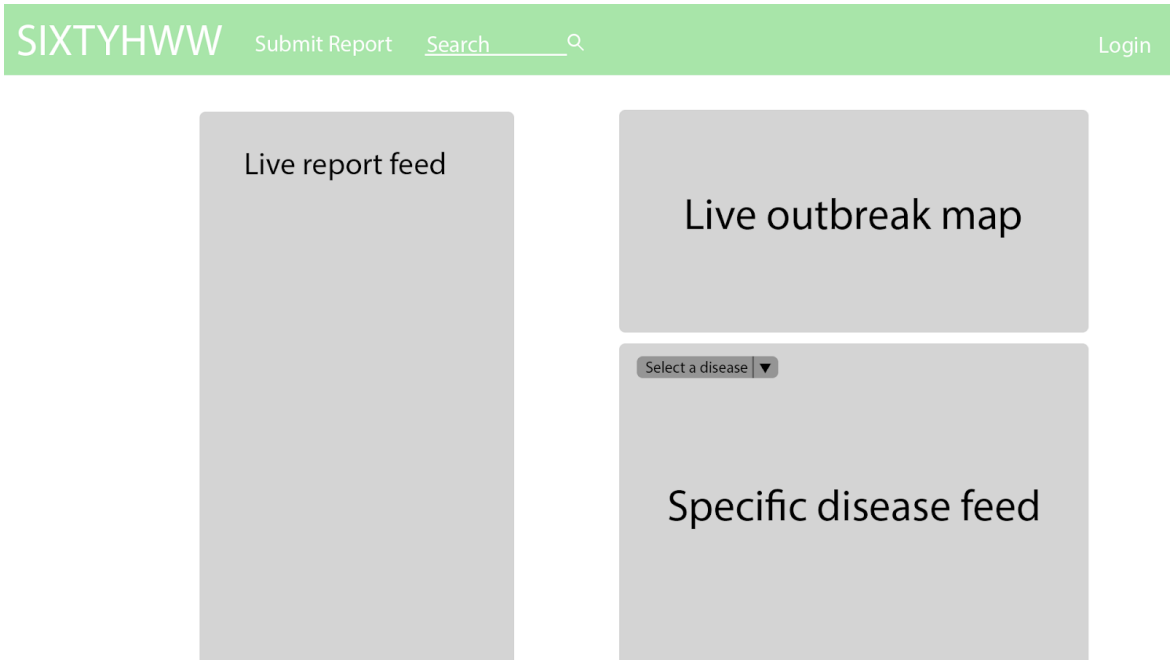
```
CREATE TABLE IF NOT EXISTS users (
 id    integer primary key autoincrement,
 uname text not null,
 pword text not null
);

CREATE TABLE IF NOT EXISTS sessions (
 token   text primary key,
 user_id text not null
);

CREATE TABLE IF NOT EXISTS location_follows (
 id            integer primary key autoincrement,
 user_id       integer not null,
 location_name text not null,
 foreign key (user_id) references users(id)
);

CREATE TABLE IF NOT EXISTS disease_and_syndrome_follows (
 id                    integer primary key autoincrement,
 user_id               integer not null,
 disease_or_syndrome   text not null,
 foreign key (user_id) references users(id)
);

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 articles(id)
);

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

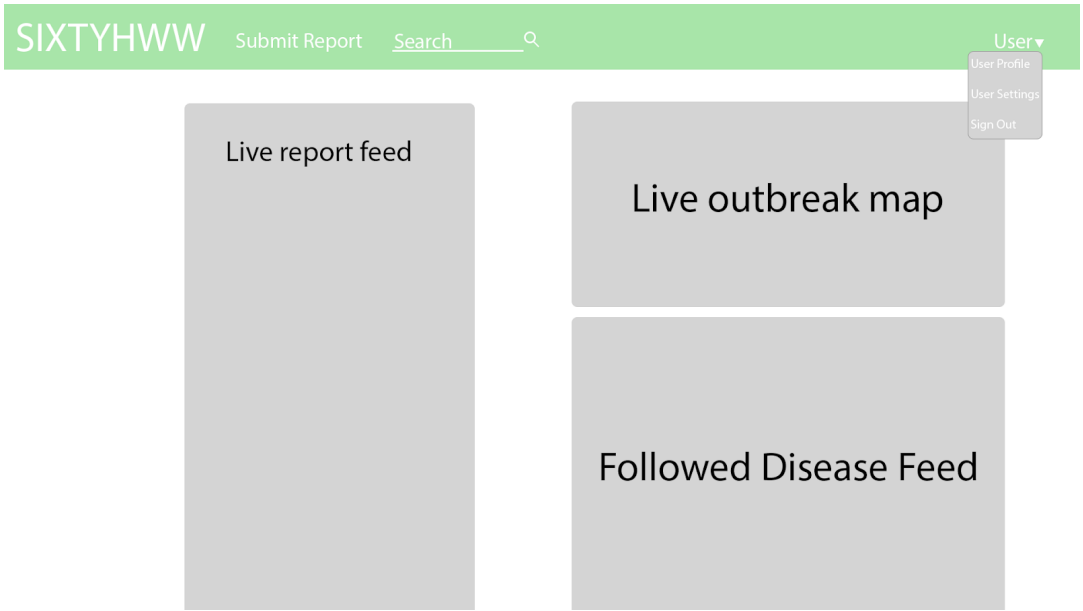
B: ER Diagram


id
user_id
location_name
Location Follows
Follows
id
pword
uname
Users
Follows
Disease and Syndrome Follows
id
disease_or_syndrome
user_id
Has
token
Session
user_id
id
article_id
event_date
diseases
Report
Part Of
id
url
headline
Article
body
date_of_publication
syndromes
longitude
country
city
latitude

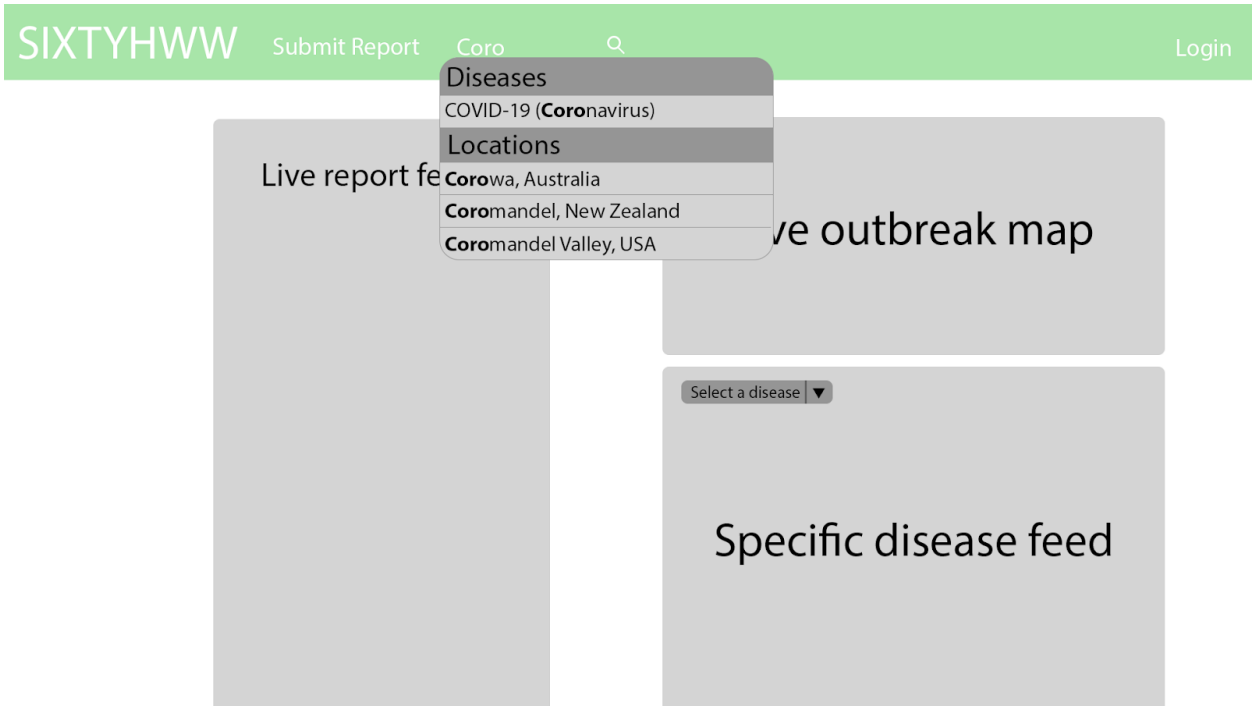
C: Landing Page Signed Out



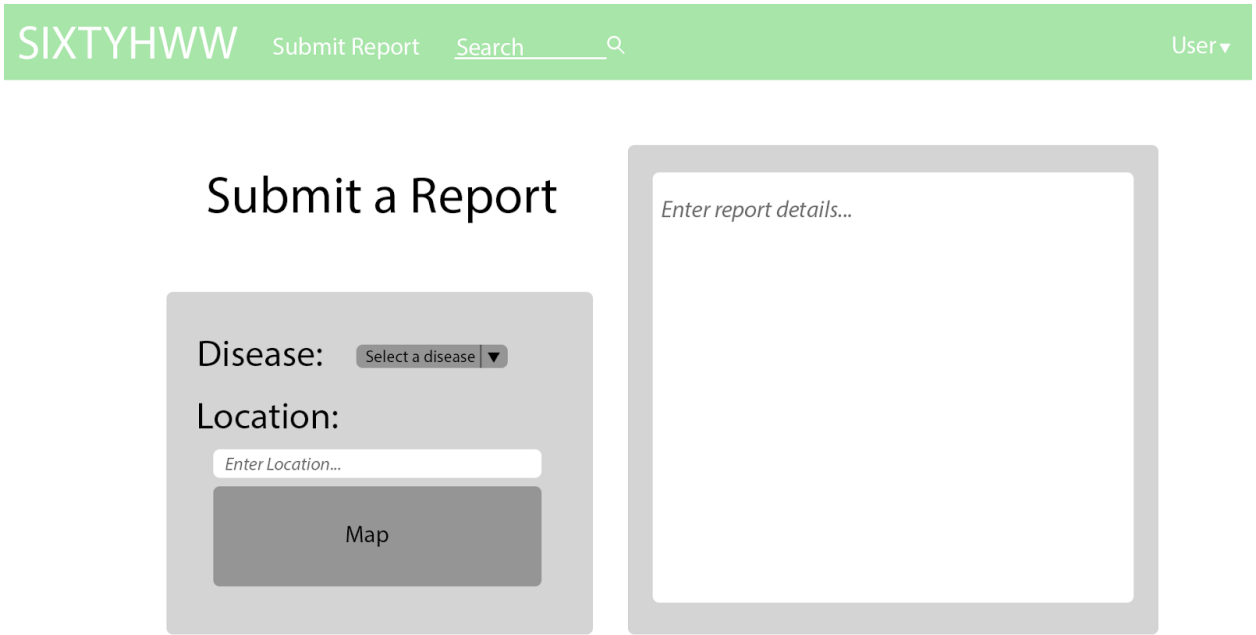
D: Landing Page Signed In



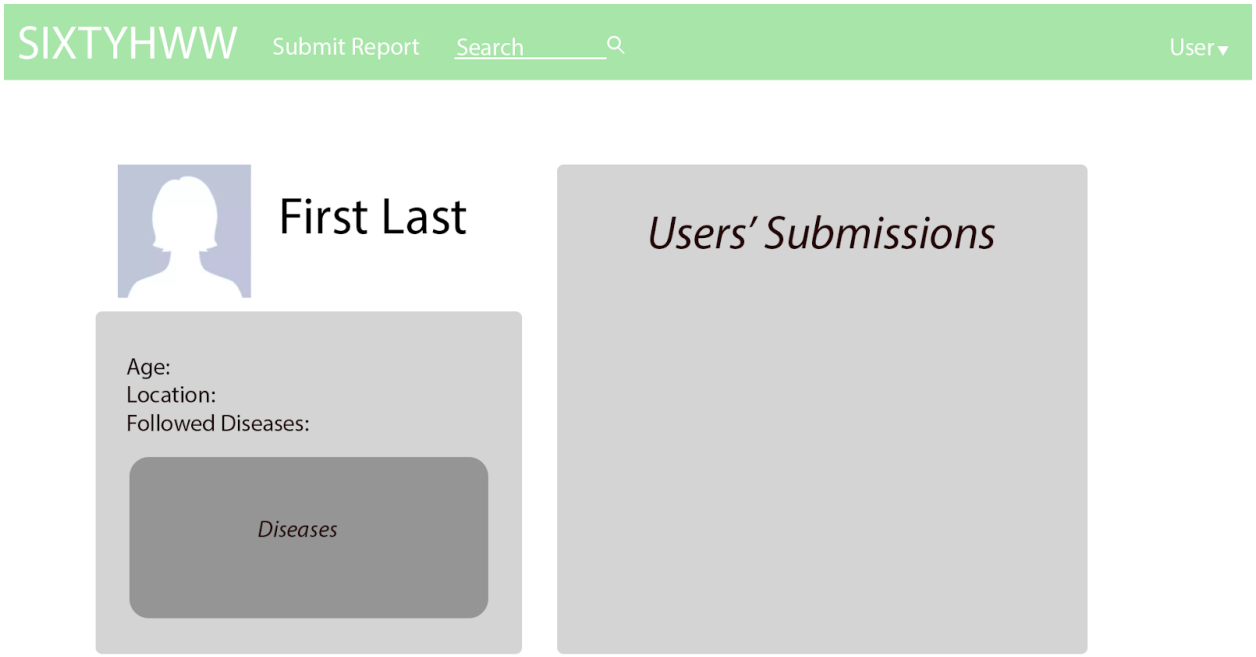
E: Search



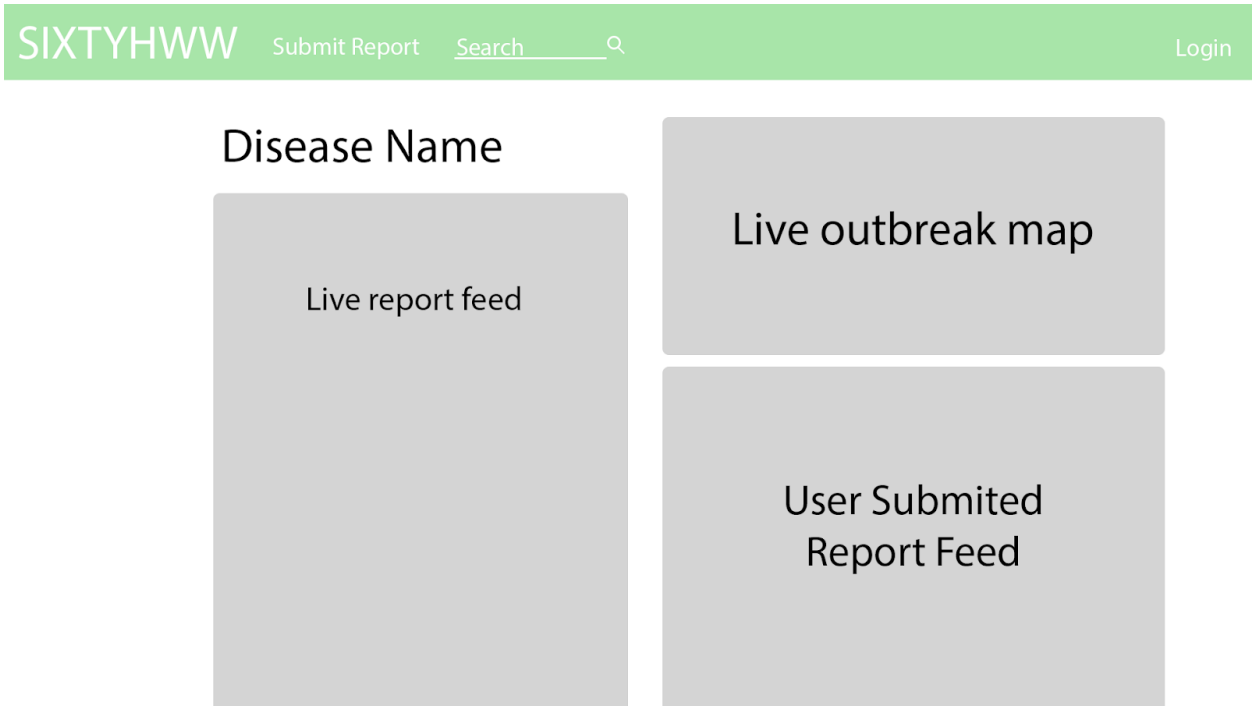
F: Submit Report Page



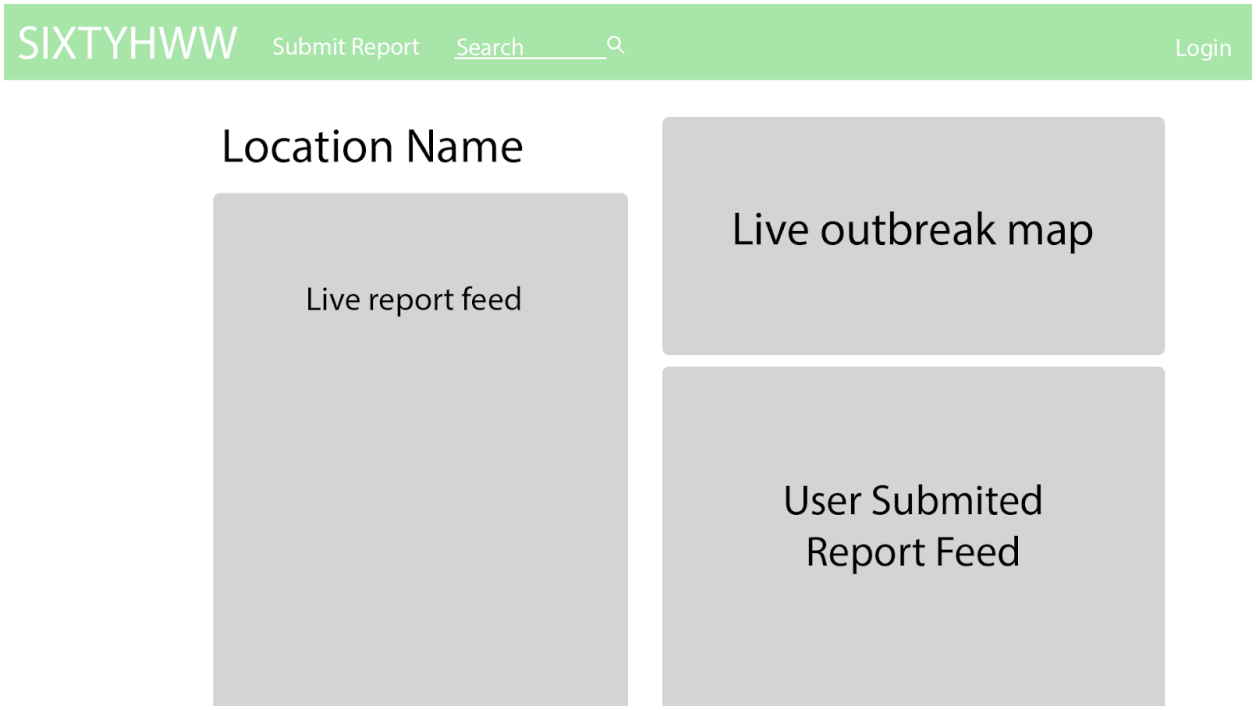
G: User Profile



H: Disease Profile



I: Location Profile



J: User Feed Card