Deliverable 4: System Design + Implementation Report
SENG3011: THUMBNAILS

Zixiang Lin z5314168
Eric Phung z5234001
Bavan Manamohan z5208542
Saalai Kaaviya Salai Manimudian z5260340
Nitharshni Chennai Kumaravel z5255563

Table of Contents:

Github Repo:

Our github repo is located at:

https://github.com/DennisLin27/SENG3011_-Thumbnails-

API

Software Architecture (API)

Firstly, we have implemented a Pythonic scraper that will access the following website: ‘https://www.cdc.gov/outbreaks/’ and navigate to specific outbreak articles. Within these articles, the scraper parses the HTML code and collates relevant disease and publication information to store in a NoSQl database. Everyday at a set time, the scraper runs again to ensure that the data extracted from the website is timely, relevant and concurrent with the data already existing in the database. This ensures that if there are updates detected on the CDC website then the relevant fields in the database are updated before the API accesses them.

We followed the REST architectural style to build our API, which allows us to make use of the ‘HTTP GET’ command to retrieve data from the CDC website with inputs entered by the user. When requests are made by the user through the API with either the Swagger UI or URL interactions , a representation of the state of the resources collected in the database is transferred to endpoints via JSON. For this assignment we decided that having the API only communicate with the MongoDB database through the Pymongo library would allow for the overall structure of the system to be coherent as well as simplify the development process.

Technology Stack Justification (API)

Before selecting our API host platform, a thorough research process was conducted to ensure the best was selected:

Option	Pro’s	Cons
Architecture		
REST or RESTful (Representational State Transfer)	· Uses Bandwidth Efficiently · Simple to understand and learn · Scalable · Lightweight	· Can’t maintain states within sessions. · There’s no contract between the client and server o not recommended

	· Has distinct and clear end points · Multi-language client compatibility · Works well with existing operating environments (like firewalls) · Has higher level libraries for frameworks and widgets. · REST just has the HTTP processing overhead. · Can return XML, JSON, YAML etc. · flexible due to loose guidelines o made to suit developers needs	when transferring confidential data · Due to its flexibility, it requires detailed documentation to understand each API call
SOAP (Simple Object Access Protocol)	· Can work with any application layer protocol, such as HTTP, SMTP, TCP, or UDP. · Security, authorization, and error-handling are built into the protocol · Doesn't assume direct point-to-point communication o Good for complex transactions that require high security.	· More Bandwidth needed, which can slow page download · XML used in this is expensive to parse · Most SOAP stacks use SAX parsing which creates a big overhead o normal HTTP processing overhead plus XML parsing overhead · Not flexible due to standardised protocols that need to be followed
RPC (Remote Procedural Call)	· Server independent. · Process-oriented and thread-oriented models work well · Simple due to straightforward semantics · Development of procedures for remote	· Users are required to know procedure names or specific parameters in a specific order. · Context switching increases scheduling costs · RPC has no standard · Interaction based, so there is no flexibility relating to

	calls is general. · Code rewriting / re-developing is reduced · Allows for applications to be used in a distributed environment	hardware · Can only support TCP/IP protocol. · Doesn't yet work for wide-area networks.
Framework for RESTful API		
Flask	· Scalable · Flexible and Simple · Easy to Navigate · Lightweight o Doesn't need many extensions to work · Supports modular programming o Functionalities can be split into modules. o Allows structure to be more flexible and testable · Easy to follow documentation	· Doesn't have a lot of tools o Will need to add extensions which can slow the app · Hard to get used to big projects in modular form, esp. if you start half way · Flask is compatible with many technologies. Thus this becomes additional maintenance costs to handle those dependencies
Django	· Features needed to build web applications are built in · Has a standardised structure. Everyone using it will be on the same page · Secure	· Django ORM is not that great · Django evolves slowly · It has a large ecosystem with lots of configurations · Not suitable for smaller projects · Steep learning curve · Can't handle multiple requests at once
Express	· Can use JavaScript both on the back end and front end. This makes development easier · It can handle lots of requests and notifications	· Software needs to be built in JavaScript and Node.js · May find it difficult to understand the callback nature. · Need to deal the Callback hell issue · Node.js is not suited for

	· Large Open-source community · Easy integration of third-party services and middleware · Easy to learn	heavy computation
FastAPI	· Python web frameworks · Free · Fast · Data validation much simpler and faster due the Pydantic library · The autocomplete feature allows production in less time and effort during debugging	· IT community is small o lower external educational information like books, courses, or tutorials. · Not tested as much in comparison to others like Flask, so a little less reliable · Need to tie everything together, which can cause the main file to become very long or crowded
Web Services		
Google Firebase	· Database capabilities: o The NoSQL database (called Firestore) is better for storing large amounts of data o Helps to achieve a high performance level due to its flexibility and scalability · Faster Development: o Firebase offers a set of pre-requisites of backend o reduces the overhead costs and reduces dependency-based challenges. · It has a large following in the IT community	· Limited querying capabilities: · not made for processing complex queries as it relies on a flat data hierarchy · Limited Data migration · Platform-dependence · Android focused · Less support for iOS

	· Concise documentation · Quick and easy integration and setup	
Digital Ocean	· Easy · Scalable · Includes virtual machine app engine, storage and database · Quick deployment. · Reliable · Droplet backups are easy to enable and affordable. · Has good Documentation and tutorials	· It could be a little hard for beginners. · Setup of the server might take time. · No live support available · Doesn't offer some important pre-installed systems like Bitnami apps, ODOO, multisite WordPress, etc. · Expensive
Aws LightSail	· Fixed Pricing Model, relatively cheap too · Easy to use User Interface · Easy to set up Networking · Provides managed databases that you can attach to your instances. · Virtual servers. · Simplified load balancing. · Large IT community and wide range of resources	· Not suitable for enterprise workloads · limited when compared to AWS EC2 · No auto-scaling available · No metrics or dashboard to know bandwidth usage or storage use · Poor tech support from AWS
Python Anywhere	· Offers MySQL and SQLite · PythonAnywhere is more like a VPS; you SSH in (SFTP for paid users), you have static file storage, file upload, etc. · Has many deployment options including git/svn/hg. · Simple interface · Relatively cheap	· Doesn't support NoSQL databases like Mongo and Couch · Must push to GitHub or Bitbucket, then from a bash shell you pull it to PythonAnywhere, which is a bit tedious · only supports Python officially on the server side

Table 1: Displaying API hosting platforms

Web App

Software Architecture (Web App)

The application is split into 3 main features, The Symptom checker, The Doctor Finder and the Chemist Finder. In the symptom checker, user input fields like age, symptoms and location are sent from the frontend and stored in the SQL database; the location form implements a feature that autocompletes and specifies cities based on the country selected by feeding live input into the the Countries and Cities API. Once these all the required fields are filled, the application sends this information from the SQL database to the APIMedic API to retrieve a list of possible symptoms and the corresponding accuracy of these diagnoses . These diseases as well as other required fields in the database are then input into the team Dwen Dwen API where related articles and other necessary information are retrieved.

The Find a Doctor and Find a Chemist feature work very similarly to each other. Once the user begins typing in a location, the Google Maps Places API's autocomplete function will narrow down the user's search and provide several options for the user to click on. Once an option is clicked, the Google Maps Places API's nearby search function will be called to generate up to 20 relevant locations and display them on a Google Map. Information such as the name and street address of each location can also be accessed. Unlike the symptom checker, data regarding this feature is not stored.

Technology Stack Justification (Web Application)

Before choosing particular technologies for our stack, we conducted thorough research and analysed the advantages and disadvantages as shown below:

Language/ Platform	**Advantages**	**Disadvantages**
Python for Backend	• Rich native library support for web scraping: Scrapy, BeautifulSoup. • Python libraries automate the extraction process of web data once a day which is a significant advantage compared to those	• Has comparatively limited performance and requires more effort to scale • Dynamic typing nature might increase the number of errors • Limited performance in shortcoming compared to other languages like

	which do not. • Python web-scraping script can easily handle both scraping data and parsing into a structured format, while others can only do the first. • Concise and simple syntax • More gentler learning curve comparing to Node.js and Golang • Experiences from team members in Python	C++
Libraries: Scrapy	• Efficiently extract data from websites, process them and store them in a format • “Spider” which can automate the web scraping process everyday whereas BeautifulSoup does not • End-to-End	• BeautifulSoup is better at extracting specific pieces of information on a webpage.
Flask for Framework	• Very flexible when it comes to adding libraries or plugins, whereas other languages cannot afford too many changes. • Flask application requires much fewer lines of code for a simple task, whereas Django needs more than 2 times more lines of code than Flask. • Can build simple applications easily with less code while Django’s functionalities might be outdated for simple	• Admin features are not as prominent as in Django. • With Flask, developers have to work with different databases by using ORM systems for Python and SQLAlchemy as the SQL toolkit. SQL queries have to be written for common tasks. • Doesn’t have a ready-to-use feature to handle administration tasks, whereas Django does.

	applications.	
Sqlite for Databasing	• Easy to use, less requirements for setup • Can be easily embedded in an application since it is lightweight. • Reliable. Unlike PostgreSQL, it is simple to make a backup with SQLite. Allows writing smaller queries and hence reduces the likelihood of bugs. • Flexible. Supports all key programming languages including Python, Java, C++ and so on.	• SQLite has syntax and format-related limitations when you compare it to PostgreSQL or MySQL. • Volume related restrictions. SQLite is only prepared for low to medium traffic use cases.
MongoDB	• Document schema allows for flexibility to store and model data structures • Easily store json objects returned by scrapers • Efficient storage and search processes • Supported by AWS which is the provider we have chosen	• High memory usage • Documents have size limits • MongoDB does not have any transaction support • Joins are not supported
Javascript for Frontend	• Since JavaScript is able to run on a browser, it is extremely fast. • JavaScript can be inserted into any web page • The demand on servers is reduced as JavaScript is run on the client's side of the application. • Javascript is extremely powerful in terms of enriching user	• Different browsers can interpret javaScript differently. This makes it difficult to guarantee a uniform output. • The frontend code is visible to others. • If an error occurs in JavaScript, the webpage may stop rendering. • Debugging JavaScript is not as easy as other languages such as

	interactivity. Javascript has the capability to include “events” such as clicking and can respond to them by communicating with the application’s backend. • Webpages can also dynamically update themselves as a response to user activity without the page having to reload. • Additional functionality that cannot be achieved using HTML and CSS alone can be achieved with Javascript.	Python. • It only includes single heritance so cannot support object oriented characteristics. • If JavaScript is disabled on the browser, the code for the whole application would not run.
HTML for Frontend	• HTML is simple and easy to use • HTML can be easily integrated with JavaScript • HTML is beneficial in implementing the foundational structure and content of the page. • HTML is also simple and a platform-independent language that is supported by all browsers.	• Using HTML alone can result in a lot of code even for very simple pages. • HTMl is static in nature and implementing dynamic web pages are extremely difficult compared to PHP and JavaScript. • It is difficult to make aesthetic designs using HTML
CSS for Frontend	• CSS allows for greater control of the aesthetic of the webpage as it has more formatting options compared to HTML. • Using CSS means that there would be less code per web page allowing for faster download times. • CSS would also save	• There are different levels of CSS e.g CSS 1, 2, 3 which could cause confusion on which one to choose. • CSS is not compatible with all browsers, so extra testing would be needed to ensure compatibility.

	time since the code can be written once, and reused for different pages. • It also allows for the ease of maintenance as a simple change in the style would allow for all elements in the page to be updated automatically.	
Axure for frontend design	• Allows easy HTML and CSS code generation and exporting • The team is comfortable with the Axure interface and its offered features • No cost allowing easy access to members of the team that did not have it prior • Provides all UX features that the team has discussed in the brief initial UX design	• Axure has a learning curve when starting out as certain features may be hard to find • On Axure it is difficult to make complex interactive UX design such as popup windows • Requires prior knowledge in coding (HTML,CSS) to export UX • Creates HTML that is hard to read without prior experience.
VScode for an IDE	• The team has easy access to VScode as it has no cost • The team is comfortable with VScode being the main IDE, as all team members have prior experience. • Quick compilation times for large projects • VScode allows easy integration with other systems • Easy to use with Git • With the Python extension VScode is simple to use and offers all features the team	• High memory and CPU consumption • Steep learning curve when starting out

	needs ● VScode saves a lot of time with its intelligent code completion feature	

Table 2: Justification of chosen technology stack

The technical architecture of our application consists of

- A backend that is implemented in Python and a Flask web application framework. We chose Python as our main programming language and Flask as our framework as all our group members are extremely familiar and fluent in its use. Python is also supported by an abundance of useful libraries and frameworks, making it ideal for the development process of our application.
- A SQLite database as the main database to store user information and inputs collected through the 'Symptoms Checker' form. SQLite was chosen as our database as it is simple to use, compatible with Python and unlikely to cause corruption of the data stored.
- A frontend interface which was implemented using HTML, CSS and JavaScript. These languages were chosen for the frontend as they can be easily integrated together, to create aesthetic and interactive web pages.
- Google Chrome as the primary web browser, The application was developed to ensure compatibility with Google Chrome as it is free, easy to use and a popular browser.

API Utilisation

There are two APIs utilised for the 'Symptoms Checker' form component on the "Select a Country" and "Select a City" pages. This included a Countries and Cities API, which allows a user to select a country and returns a list of cities in that specific country. The user is then able to select their city.

The ApiMedic API is used on the 'Diagnosis Results' page, This API allows us to map a list of symptoms that a user inputs through the preceding pages in the 'Symptom Checker' form to a correlating list that includes the possible diseases and the accuracy percentage of each disease's likelihood.

The returned diseases and location are then fed into the Dwen Dwen API, to generate reports for our 'Disease Reports' dropdown bars for each disease. In these dropdown bars, we present information such as a possible disease, accuracy of results, a brief description of the outbreak and a link to an outbreak article.

The Google Maps Places API was used in the 'Find a doctor' and 'Find a chemist' pages. Here, a user is able to locate doctors and chemists within a 2km radius of the address they entered

API	**Link**
Priaid health service resources	https://sandbox-healthservice.priaid.ch/docs.html
Location API (autofill):	https://documenter.getpostman.com/view/1134062/T1LJjU52#fec5b515-b5e1-41c5-8ab4-2cc49bec7694
Team Dwen Dwen's API	https://seng3011-dwen.herokuapp.com/docs/#/Articles/get_articles
Google Maps Places API	https://developers.google.com/maps/documentation/places/web-service

Table 3: Displaying APIs and Links

Symptom Checker

Our app has many innovative and useful features such as the symptom-disease matching feature and locating nearby doctors and chemists. The symptom-disease matching is essentially a list of questions that collects information such as age group, symptoms, location, and additional information about the user. The app then collates this all together and returns the possible diseases or illnesses they might have and display it from most to least likelihood.

Doctor Page

Following online research of patient diagnosis, only 21% of people usually follow through to get a consultation with a doctor after experiencing symptoms, meaning that 4 in 5 people who search for a diagnosis fail to see a doctor, and consequently fail to get an early diagnosis for their possible condition. As a means to increase the rate of doctor consultations following online diagnosis, The application contains a simple and accessible find a doctor feature.
The Doctor page consists of

- An information button to help users understand how this feature will function.
- A search feature that allows the user to input a valid location in the Google Maps Places database
- An autocomplete search bar that suggests valid locations that match or contain the same characters as the current input
- Google maps
- Markers that show the entered location are displayed, with a black house symbol and all doctor offices in a 3km radius displayed with a red marker.
- Doctor office information, when a user clicks on a doctor marker the address and doctor name will be displayed in a popup window.
- Zoom in and out feature, allows the user to traverse the map making it easier to find a location respective to their current location
- Street view feature that allows the user to know what the building will look like from their perspective when they do choose to visit the doctor.

Chemist Page

The application follows the timeline of a diagnosed patient, once a patient detects symptoms there is a symptom checker feature to provide a list of possible diagnoses. Once they know possible diseases and the severity of their condition, comes the doctor consultation and diagnosis. And oftentimes, following diagnosis, comes the medication stage where patients need to seek out a chemist to buy medicine recommended by the doctor. To ease the process of finding a chemist, The application contains the find a chemist feature which works similarly to the previous find a doctor feature with the difference being, the app provides a list of pharmacy locations as opposed to doctor offices.
The Chemist page consists of

- An information button to help users understand how this feature will function.

- A search feature that allows the user to input a valid location in the Google Maps Places database
- An autocomplete search bar that suggests valid locations that match or contain the same characters as the current input
- Google maps
- Markers that show the entered location are displayed with a black house symbol and all chemists in a 3km radius displayed with a red marker.
- Chemist information, when a user clicks on a chemist marker the address and chemist name will be displayed in a popup window.
- Zoom in and out feature, allows the user to traverse the map making it easier to find a location respective to their current location
- Street view feature that allows the user to know what the building will look like from their perspective when they do choose to visit the chemist.

UI Design Prototyping

Before the final design was chosen, we designed two other prototypes for the application as shown below. This allowed for the reflection and discussion of each other's ideas. As a result, we were able to derive our final design as shown in our 'Use Cases and Requirements Report'.

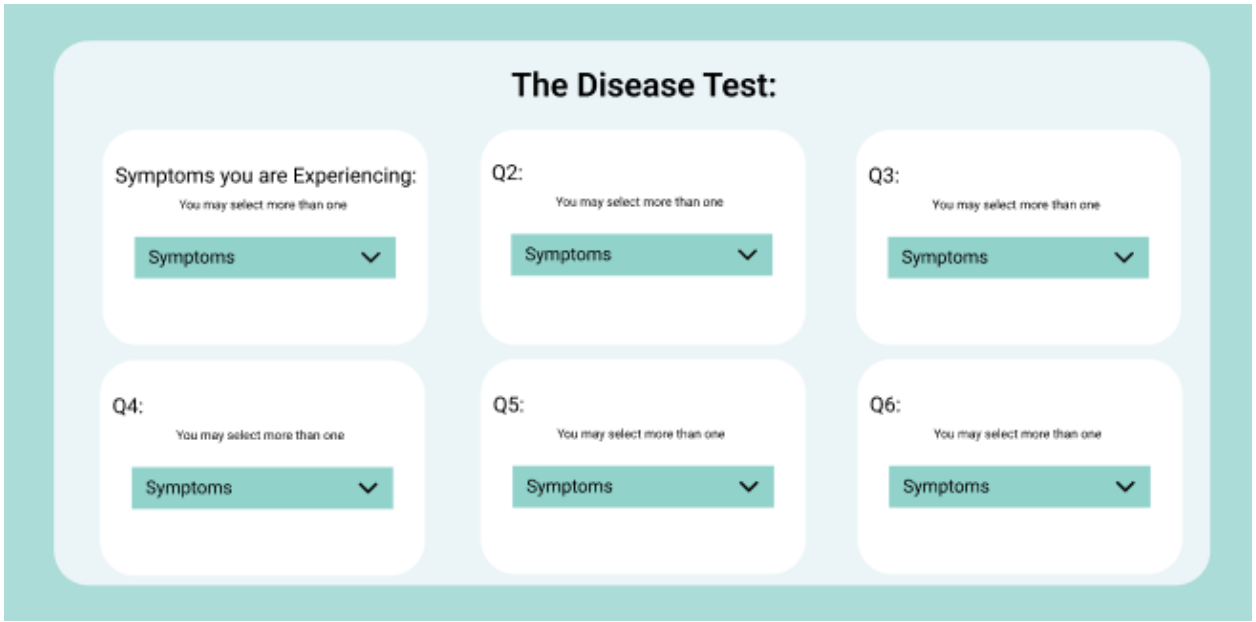

Figure 1: Displaying an initial UI prototype

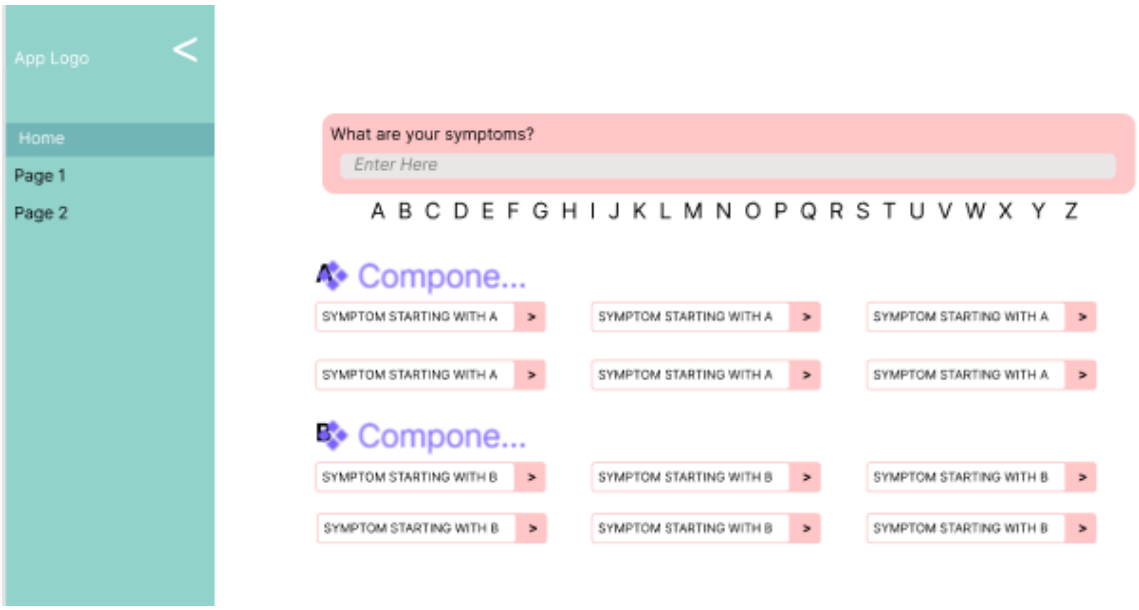

Figure 2: Displaying an initial UI prototype

Key Benefits/ Achievements

There are a significant number of people who do not have good access to doctors in their location all around the world. There are also quite a number of people who cannot afford the expenses of a doctors' consultation, especially for what they consider to be minor injuries and illness, thus often leading them to dismiss them even though they need immediate care. This has been one of the major reasons for the rise in popularity of e-health applications.

A survey done by YouGov in Britain shows that 47% of the public have searched for possible diagnosis online for their current symptoms, and only one in five (21%) have then followed through to get a consultation with the doctor. This shows the need for users to acquire accurate, evidence-based medical information in order to make better decisions about their health care. While there are many similar applications available in the market, many unfortunately are inadequate at providing precise diagnosis, do not have or have poor differential diagnoses which can lead to increased stress and anxiety for the user and tend to be very complex to understand and use.

The main focus of Medihealth is to cater to these limitations and give the best experience to our users. This is achieved by providing personalised and accurate information about the current disease/ illness that the user may be experiencing and lists them from most to least likelihood. A very good

symptom API has been used in this web application to ensure that our users only have access to the most up-to-date and reliable information. Also, the simple UI allows for easy comprehensibility, which is extremely vital as a large portion of our target market are elderly people trying to seek quick tips to treat their current ailments or advice to seek professional help if need be.

The symptom-disease matching feature allows the users to be able to treat injuries and conditions that are not life-threatening but need immediate attention (such as ear aches and chills) from home. This can also encourage patients with possible critical issues such as a stroke or heart attack to get an idea of how severe their condition could be and seek professionals and emergency care asap. Also, the information collected from this survey can be used by scientists, health professionals and epidemiologists to determine trends and patterns available between symptoms, location, age groups and current outbreaks to discover better treatment and prevention methods that can be administered to the population in the future.
The ‘locate nearby doctors and chemists’ lets the users know of the possible places they can go to get necessary consultation and medicine when need be. This will be especially useful when trying to locate specialists within the area that the user lives in.

By combining the ‘Symptom Checker’, ‘Find a Doctor’ and ‘Find a Chemist’ features, we aim to differentiate ourselves from the rest of the market by not only providing the ability to retrieve an online diagnosis, but to go one step further and provide accessible tools which encourage users to follow-up with medical professionals and medicate any illnesses they may have.