

**Mobile Doctor**

**Prototype**

**Senior Project Technical Report**

Prepared for: Huanying Gu, Professor of CSCI-455 M02

New York Institute of Technology

Department of Computer Science

Prepared by:

Nusrat Bainya, nbainya@nyit.edu

Mohammad Ishtiaq, mishtiaq@nyit.edu

December 20, 2021

**Table of Contents**

1. Abstract 3

a. Purpose

b. Project Objective, Goals, and Scope

2. Introduction 4

a. Existing Systems

b. Proposed System

c. Technology & Tools

d. Software Engineering Process

3. System Analysis 10

a. Data Flow Diagram

b. Use case diagram

4. Database Design 12

a. Sqlite Database

b. Disease Database

5. Functionality & Implementation 18

a. Code Snippets with explanations

b. Covid API Explanation

6. Unit Testing 30

7. Conclusion 30

a. Future Plans

8. Meet the Team 31

9. Citation 33

**1. Abstract:**

*Purpose:*

With Covid-19 running wild for the past few years, the world and the people are scared of anyone with any sickness. Because of this, we have decided to create a small-scale application that can help users match their symptoms to diseases quickly, easily, and fast. Since doctors, physicians, or specialists cannot be contacted 24/7, we propose a small-scale mobile application to help diagnose a person’s symptoms or worries. The mobile application needs to be lightweight and not too complicated because we want the application to appeal to everyone. Allowing our application to run on both android and ios, will allow our application to reach more people and eventually help more people.

*Project objective, goals, and scope:*

Our objective is to create an application that will take user input (User Symptoms), and match them to a hand-made database of diseases provided by Prof. Huanying. We want to make this application available for both android and ios with the least amount of code differences. We will be using Xamarin to achieve this because it allows us to program the application in one language and during execution, xamarin is able to convert all the code, files, and libraries into either android or ios.

One of the goals of this project is to create a working database with SQLite. We will use this database system to store user inputs, which will be used to help match diseases to the symptoms. Another goal is to create a built-in local covid tracker. This will allow the user to see simple statistics about their local area (updated daily) and make the decision to venture outside. This application will also have a scroll view with local news articles about the covid-19 pandemic.

This application will be a prototype of the final product. This prototype will build a very simple interface, data management, and covid tracking system. Our symptom-disease matcher will have a very early version of our algorithm as getting the most accurate result in the medical context is hard for those who do not have a medical background.

**2. Introductions:**

*Existing Systems:*

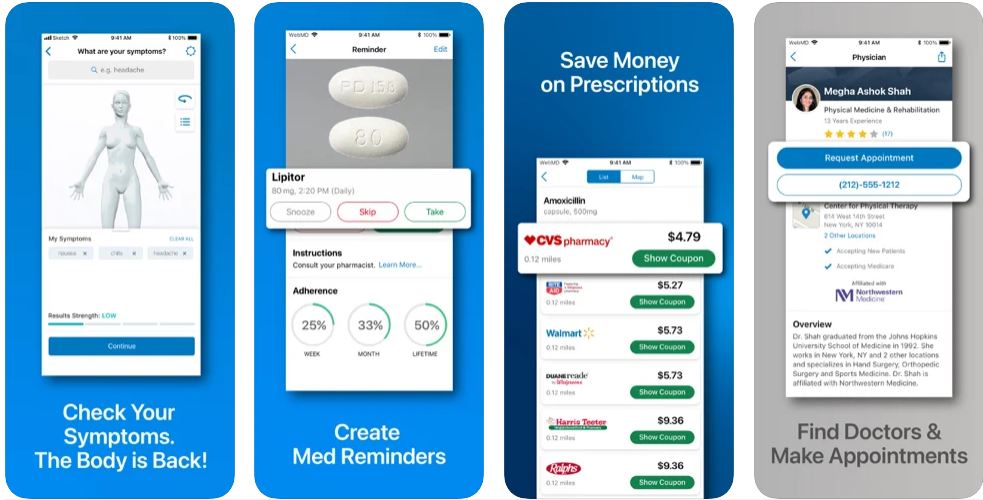


Figure 1.1

WebMD (Figure 1.1) - first of all, it is charge-free, the app is very simple to use. You need to enter your age, gender, and then you are to enter your symptoms. Also, you are required to answer some questions, to determine which symptoms bother you the most. There is also a section where you can save your current medicine and any past condition you had. Once you have completed all the relevant information, the symptom checker generates a list of potential conditions.



Figure 1.2

Mayo Clinic (Figure 1.2) - the app provides you with helpful tools to manage your health on the go. You are also able to search for reliable, research-backed answers to your questions about a specific disease, symptoms, and health procedures. Also, they have a request appointment option that connects you to the best specialties. The app provides the patients with an overview of all the functions, with links to the patient(s) information, appointments, messages, lab results, notes and documents, medication, and reports.

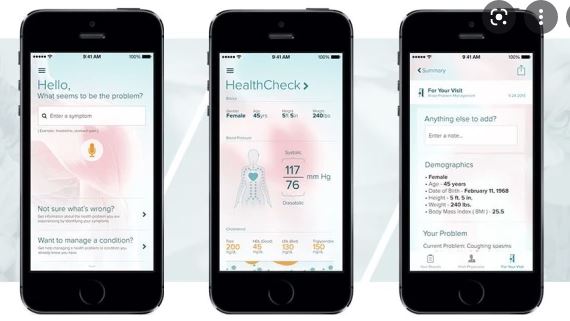


Figure 1.3

AskMD (Figure 1.3) - It is a personalized health consult app. When you first install the app, the first thing you see is they ask you about your symptoms. Not sure about your symptoms, they have this option for you to get information about the health problem you are experiencing. Then they ask you about your age, sex, height, and weight. Then, the summary of what you provided them with. The app also has a database of hospitals within the app that the users can browse and filter based on insurance. The app has voice recognition.

*Proposed System:*

After doing our research into existing applications, our team has narrowed down a system that addresses the downside to many applications these days. We propose a small-scale application that is extremely simple in design, which can cater to all user groups. Creating an application that does not add all the additional features like a sign-in system, profile system, or health tracker system will allow our application to achieve its ultimate goal. Our application will work 95% remotely and be disconnected from the network (wifi). With this, we can make an application that is fast, reliable, and never has downtime because the information is stored safely and locally on the user’s device. The Covid-19 tracker will allow the user to find local information quickly because all the information is automatically imputed based on the geolocation of the user. Our goal is to get the user to an answer asap because health waits for no one, and having to deal with an intricate UI is minimized.

*Technology & Tools:*

Xamarin:

Xamarin is an open-source platform for building applications that is cross-platform compatible. This allowed us to create an application that is going to work both on android, apple, and Windows devices. Being able to reuse code to a certain degree gave us a lot of freedom to experiment with different designs, databases, and libraries. Xamarin also allows us to program in a single programming language and have it automatically translated to other languages for other operating systems.

Visual Studio:

Xamarin is a specific plugin for visual studios. We were able to completely design, code, and implement everything through visual studios. It also comes with a real-time simulator that allows us to simulate a device and see real-time changes on screen. With this simulator, we built our UI, database, and got information from the various sensors on the simulated environment.

Programming Language - C#:

We choose to use C# due to its heavy documentation on many different Xamarin libraries. C# is also the main programming language for Xamarin, so everyone was able to learn this language and utilize their knowledge to build the application. Since we only had to program in C#. Xamarin is able to translate it across all operating systems quickly so that we didn’t have to manually do everything.

Github Desktop:

Github Desktop was used primarily to share code amongst the team members. As everyone worked on the code, GitHub desktop would compile and save the code so everyone’s work was synced together. We were also able to revert to old saves in case of huge errors that occur from code changes.

Lucidchart:

Lucidchart is an online chart-making tool. It allows us to create our DFD, ERD, and use case scenario diagrams. We are easily able to drag and drop different shapes that represent different data processes. We are able to share the charts amongst one another for easy access and editing of charts.

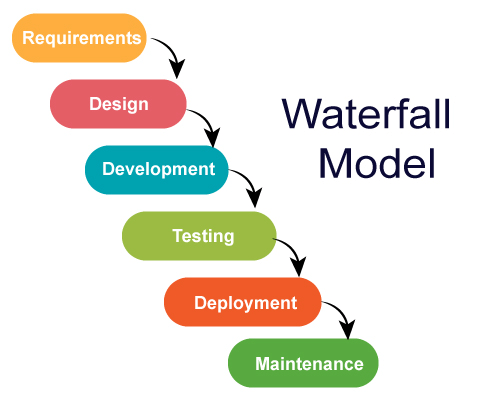
Google Drive:

Google Drive was used to save all documentation and research. Everyone was connected to the shared folder where we held our research on existing applications and the example dataset provided by the professor. It is simple, fast, and reliable for everything related to filing transfer and storing.

Discord:

Our main form of communication was through a chatting application called Discord. Discord allowed us to create a group chat and share material quickly and easily. We were able to host unlimited voice and video calls with quick sharing of screens. We also used the platform to discuss meeting times and helped organize material in respective categories.

*Software Engineering Model (Waterfall model):*



The software engineering model we have discussed and chosen for this project is the waterfall model. We start by finding what we need to complete the task at hand, this includes finding a Covid-19 tracking API, a dataset of diseases, a list of symptoms, learning Xamerian, ranking which database system we want to use, and researching existing applications to learn from and build off of. Then we moved into designing the application’s UI and trying to make it look visually pleasing and informative. Moving on to the development of the application through Xamerian forms and understanding how we are going to handle data storage, retrieval, and symptom checking. This is where we implement our learning from the “Requirements'' stage to create a ranking algorithm that can give the most accurate response to the user. We follow this up with some testing through test data or real-world data from our disease database. We also are able to upload the application to both an android or ios phone to see the results. We can deploy the application on the app store on their respective devices and keep maintenance going based on user feedback. Some of the feedback can include aspect ratio issues, refining our ranking system, updating the disease database and symptom data as we expand our medical research.

**3. System Analysis:**

*Data Flow Diagram (DFD):*

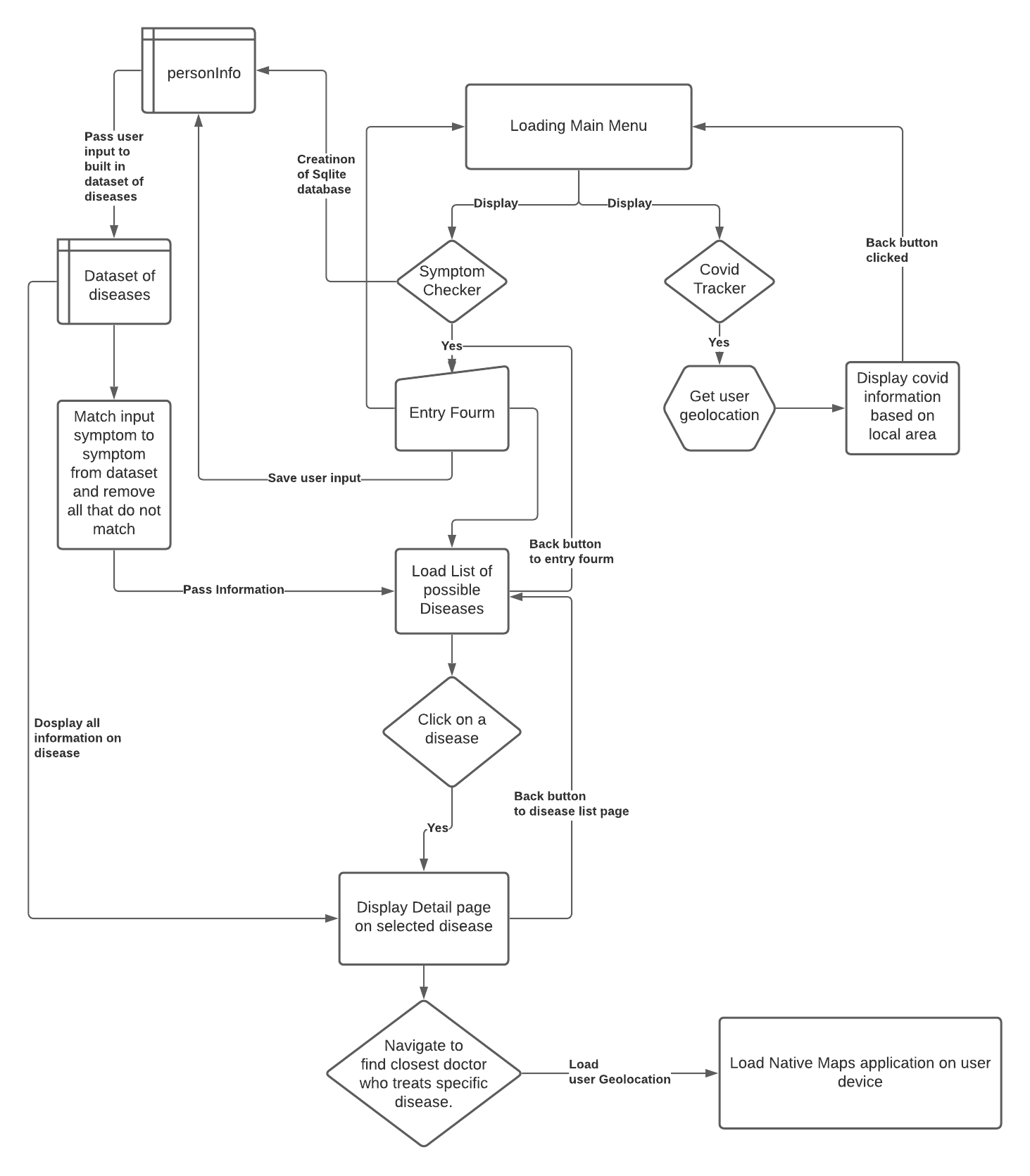
**

Figure 2.1

The above (Figure 2.1) DFD focuses on how data will flow through our 2 application functions. We start off by loading into the application and displaying the two application options. The Covid tracker function is small and does not require much data flow because it is simply retrieving geolocation and displaying news and statistics about covid in the local area. The symptom checker presents the “entry form” which once filled, will store the information into the SQLite database called “personinfo”. The symptoms entered into the “personinfo” database will then pass through the disease database “DiseaseDetail” and filter out diseases that match and do not. Finally, after the matching is complete, the user will be presented with the results in which they can select which disease they want to learn more about. The selected diseases information is then presented on a new page with the additional option to find a doctor near their geolocation. Each new menu has a back button on the top left corner of the screen which simply allows them to load the previous screen presented to them.

*Use-Case-Diagram:*

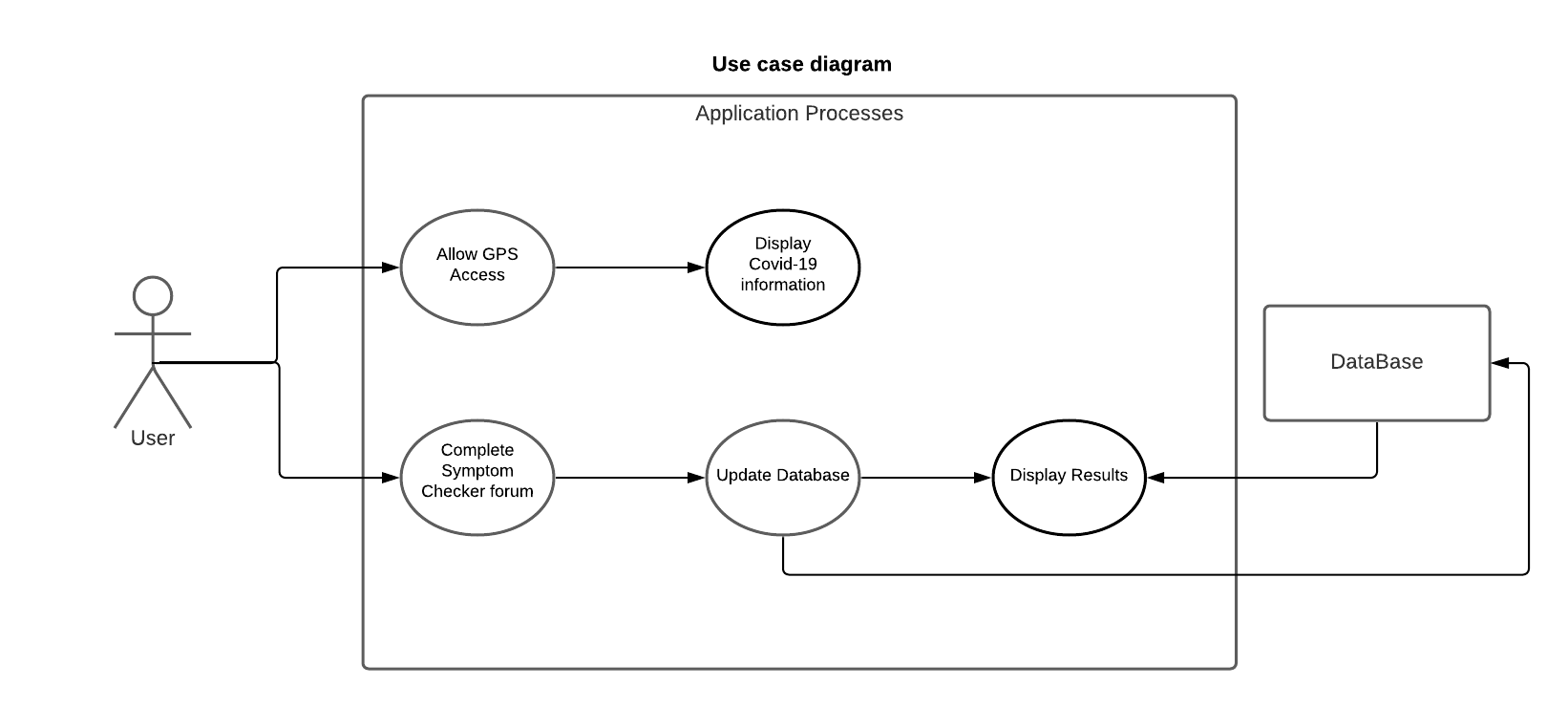


Figure 2.2

Figure 2.2, displays our use case diagram. We can see that the only entity is the “User” because our application has no login or admin modes. This helps our goal of making the application simple and straightforward. The user only has two options and both require little to no actions from the user, hitting our second goal of making the application easy to use for all users.

**4. Database Design:**

*Database Creation / Connections: User Symptoms*

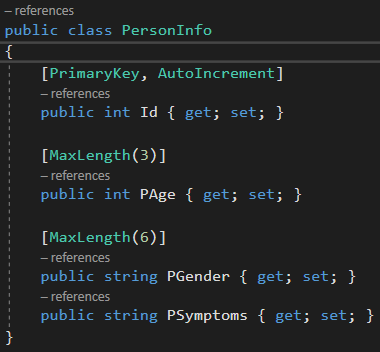


Figure 3.1

This (Figure 3.1) code snippet displays the information stored in our SQLite after the user completes the entry forum. We use “Id” as our primary key and with the Nuget SQLite package, it has the additional property to automatically increment as more information is added to the database. We use the abbreviation od “P” to represent “Person”, which is connected to all the attributes like “PAge”, “PGender”, “PSymptoms”. We store the “PAge” as an integer with a max limit of 3 digits and no decimal values. Following that, both “Pender” and “Psymptoms” are saved as strings for convenience later on in the application.

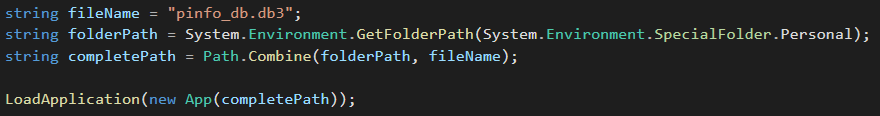


Figure 3.2.1

In Figure 3.2.1, SQLite needs a specific location to be stored for use on Android. First, we start by giving our database a name, in this case, it is named “pinfo\_db.db3”. Next, we store the folder location (as a string) in which the database file needs to be stored, and we have to save it to a “Special” personal folder which is not normally accessible via android functionality. Finally, we make a string that will concatenate the “folderPath” and “fileName” with the file building function “Path. Combine”. Finally, we pass the “comlppetePath” to the overridden constructor for “LoadApplication” which can be seen in Figure 3.2.3.

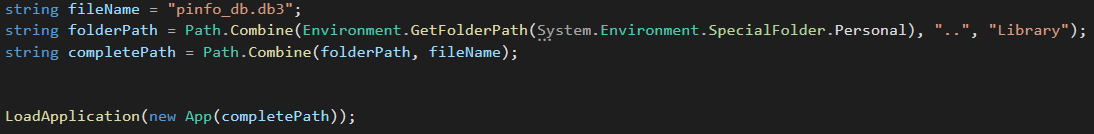


Figure 3.2.2

In Figure 3.2.2, we are doing everything the same as in Figure 3.2.1, with the exception of where we are storing the database as you can see in “folderPath”. For iOS, we have to add the extension to “Library” because this is where user information from applications is stored. The following process is the same as Figure 3.2.1. This instance shows how Xamerian lowered the number of changes needed to connect functionality to a specific operating system.

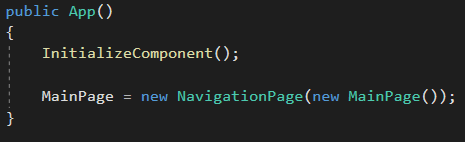


Figure 3.2.3

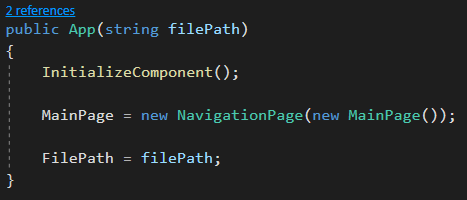


Figure 3.2.4

Figures 3.2.3 display the function that is run as soon the user opens the application. Here it will display the first UI page to be displayed, which in this case is our “MainPage”. But as you can see in Figure 3.2.4, we have a function override. As we see in the previous Figures 3.2.1 and 3.2.2, we have a line of code that says “LoadApplication(new App(completePath));”. This code tells our application to launch using the override of the “App” function because we want to pass the “completePath” as a parameter called “filePath”. We do this because we want to allocate the specified space in the specified folder to the database file. At the end of the function override, we store the FilePath so that we can reference it at any point in the future.

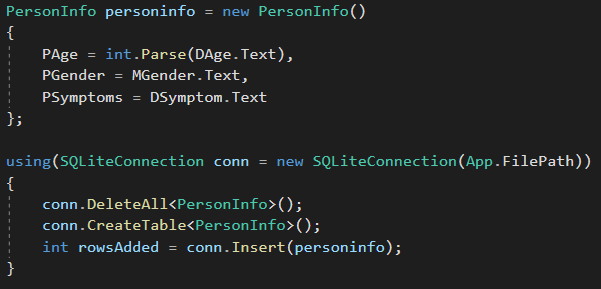


Figure 3.3

After the user has completed the entry forum, the information from the entry fields is used to populate a new personinfo object. By default, all entry fields in Xamarin are stored as strings, but we need to do a simple conversion for the “DAge” because we want to store it as an integer value. Now that we have stored the information in an object, we want to open a connection to the database we created in Figures 3.2.1 and 3.2.2. With the “using” function, SQLite does us a favor by opening the database connection and closing it once we have navigated away from the page. This function ensures that the database is not left open at all times and can save us from errors later on when we try to access it. We pass the “App.FilePath” from Figure 3.2.4 to distinguish where the database file is located, stored, and updated on the given operating system. We first start by deleting all entries in the database. We do this to ensure that every time we call this database to retrieve or store data, we will always have one instance of it. This helps keep code clean, and manageable. If the command cannot be run, SQLite will simply skip the command and continue with the next without throwing an exception. Following the deletion, we then create a blank table called “PersonalInfo” and populate it using the information from the object we created in the code above the database connection function.

*Disease Database Creation / Population*

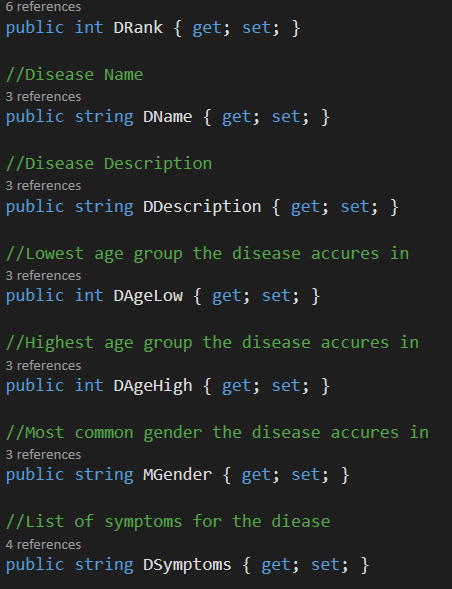


Figure 4.1.1

In Figure 4.1.1, we create the attributes that will be used to populate our disease database. The information is going to be populated by hand with the dataset the professor has provided us. The addition of attributes like DAgeLow, DAgeHigh, MGender will help refine the search in the future to find the best Disease match to the user’s inputs. We also set DRank to a double value which is detailed more in the algorithm in Figure 5.1.3.



Figure 4.1.2

In Figure 4.1.2 we start by creating an ObservableCollection with the specification from the DListModel class called “DiseaseDetails”. The ObservableCollection will then be populated by the list of objects from the database being created down below called “DiseaseDetails”. We can see that we have 2 test diseases in the database with every attribute populated. DRank starts at 0.0 for every disease because this value will be changed to rank the diseases and is held as a double value to get the decimal value. The “DSymptoms” attribute is where the symptoms of the disease are being held, and later we will turn this string of symptoms into a list of symptoms.

**5. Functionality & Implementation:**

*Symptom to Disease Algorithm:*

On execution, the algorithm we connect to our SQlite database to retrieve the previously inputted user symptoms. Doing a type checking to make sure if the retrieved information is not null, and if it is we input a blank string. Following that we can transform this information into a list of symptoms to more easily match them to the database of diseases. Next we run through this new list of user symptoms and remove redundant spaces, this step ensures we have a consistent list of symptoms that match with the database. Following that, we loop through the disease database and as we go through it, we store the specific diseases symptoms to a list just like we did for the user symptoms. We do our redundancy checking and move into the symptom matching phase. First we check each symptom from the disease list we created before to every symptom the user has entered which is stored in the list we created at first. Whenever we find a match between the two lists, we have the fractional value to the rank of the specific disease we are indexed at. If there are no matches at all with a specific disease, it is removed from the database temporarily for the display in the new page, but if anything is matched, they will be displayed based on their rank value in a descending order (1 to 0). We use the decimal value because we want to ensure those diseases that have a 100% match are placed at the top, and if multiple diseases have the same ranking, they are listed based on alphabetical order.

*Symptom to Disease Algorithm Code Explanation:*

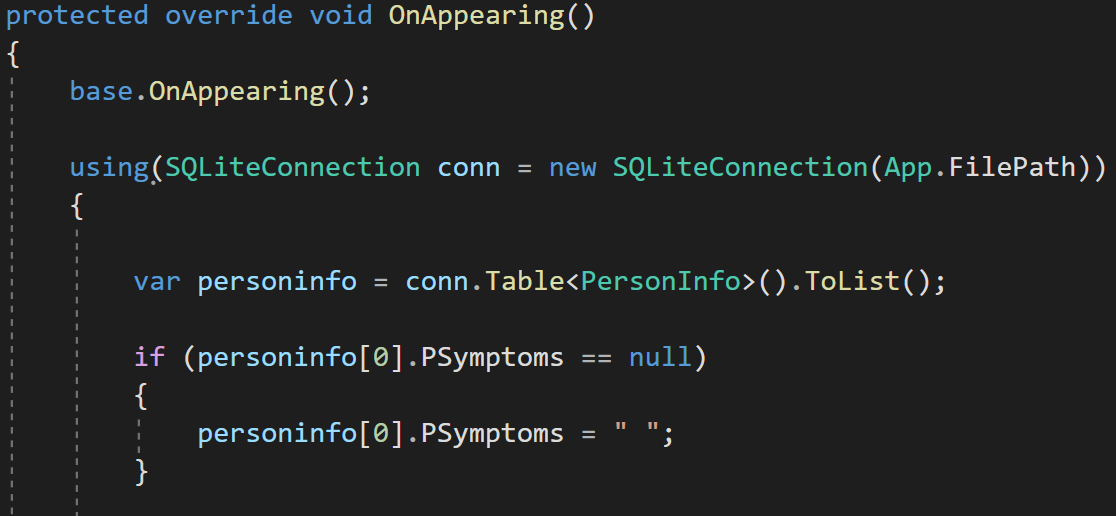


Figure 5.1.1

Starting in Figure 5.1.1, we see the function OnAppearing(), which is a set of code that will run before the specific page will load on the screen. Here we are going to rank our diseases based on the user inputs and then pass the list so that it can be displayed on the screen. Next, we connect to the SQLite database using the using() command from SQLite just like in Figure 3.3. We store the data from the only table in the database to a variable named personalinfo, and here we do some checking by making sure that the PSymptom attribute is not null (or was never changed). If it is, we simply add a blank string to the database as a placeholder.

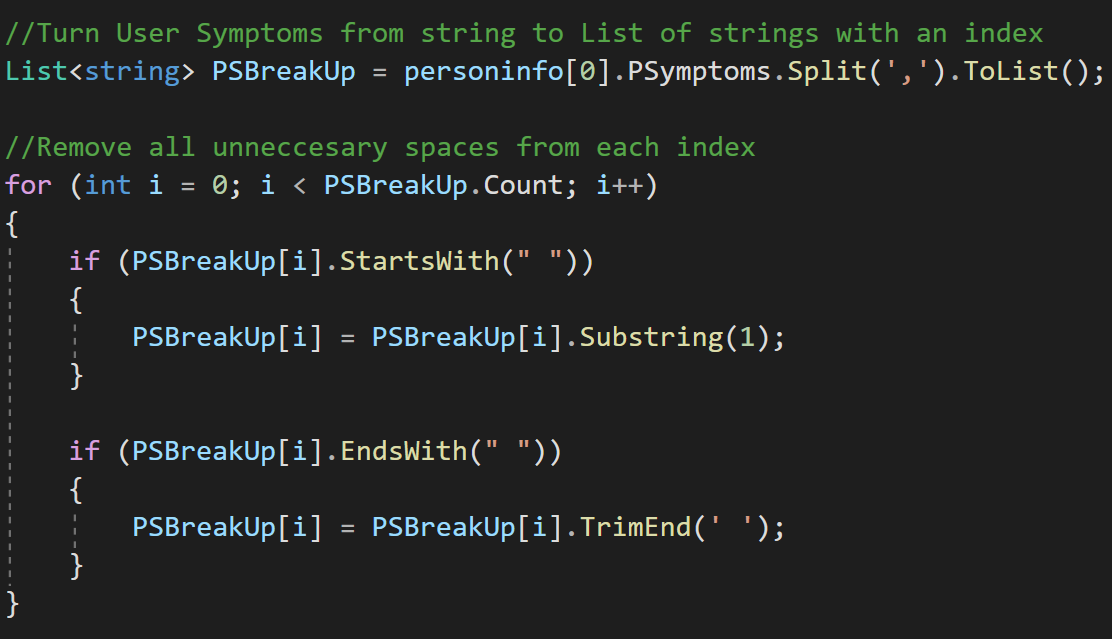


Figure 5.1.2

Here we create a new list of strings by breaking the PSymptoms by the specified character ‘,’. To ensure more clarity, we run through the new list and remove any redundant spaces from the beginning and end of the list entries.

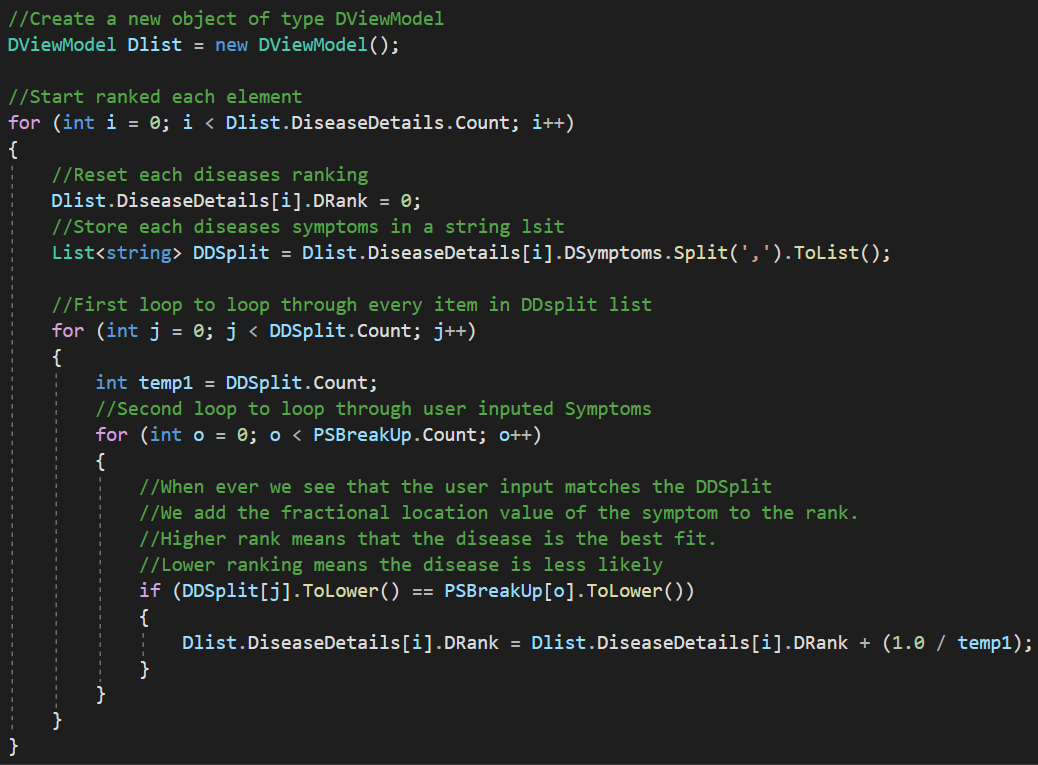


Figure 5.1.3

To start the ranking process, we create a reference/object for the database holding our disease called “Dlist”. We open the first for loop to run until we have cycled through all of the diseases in the database. During this process, we are also turning every instance’s symptom string to a list of symptoms and splitting them by the ‘,’ like before. We then open another loop that runs until we have run through every instance of the DDSplit list that was just created before. Then we open another for loop to run through all of the PSBreakUp lists that were created and hold all the symptoms the user entered in a list form. We then check if the symptoms from the user match the symptoms from the disease database, and if they do, we add the decimal value of that to the rank. Since we store DRank as a double we can divide the number of symptoms the disease has by 1 every time we get a match and add that to the specific disease rank we are currently indexed at through the first for loop. After running through everything, we now have a Disease database fully ranked and ready to be filtered through.

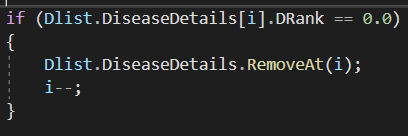


Figure 5.1.4

With this final edit to the algorithm, we want to remove any redundant items from the list. Anything that returns with a ranking equal to 0 needs to be removed because none of the inputted symptoms matched with that specific disease. We also move back 1 index in the for loop to ensure the loop does not exceed its limit and everything runs smoothly.

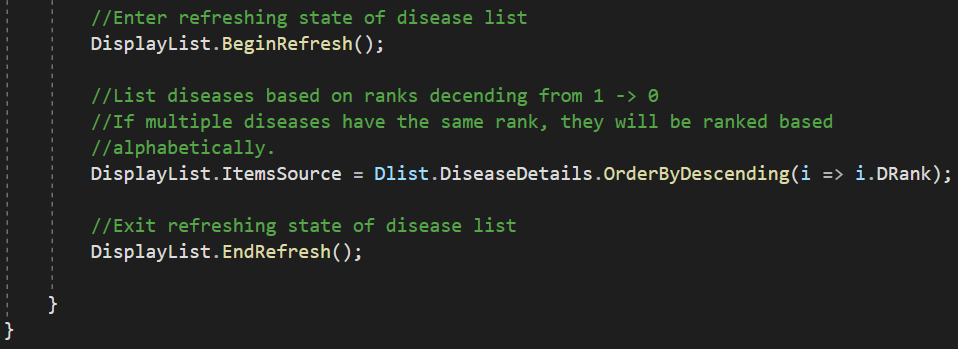
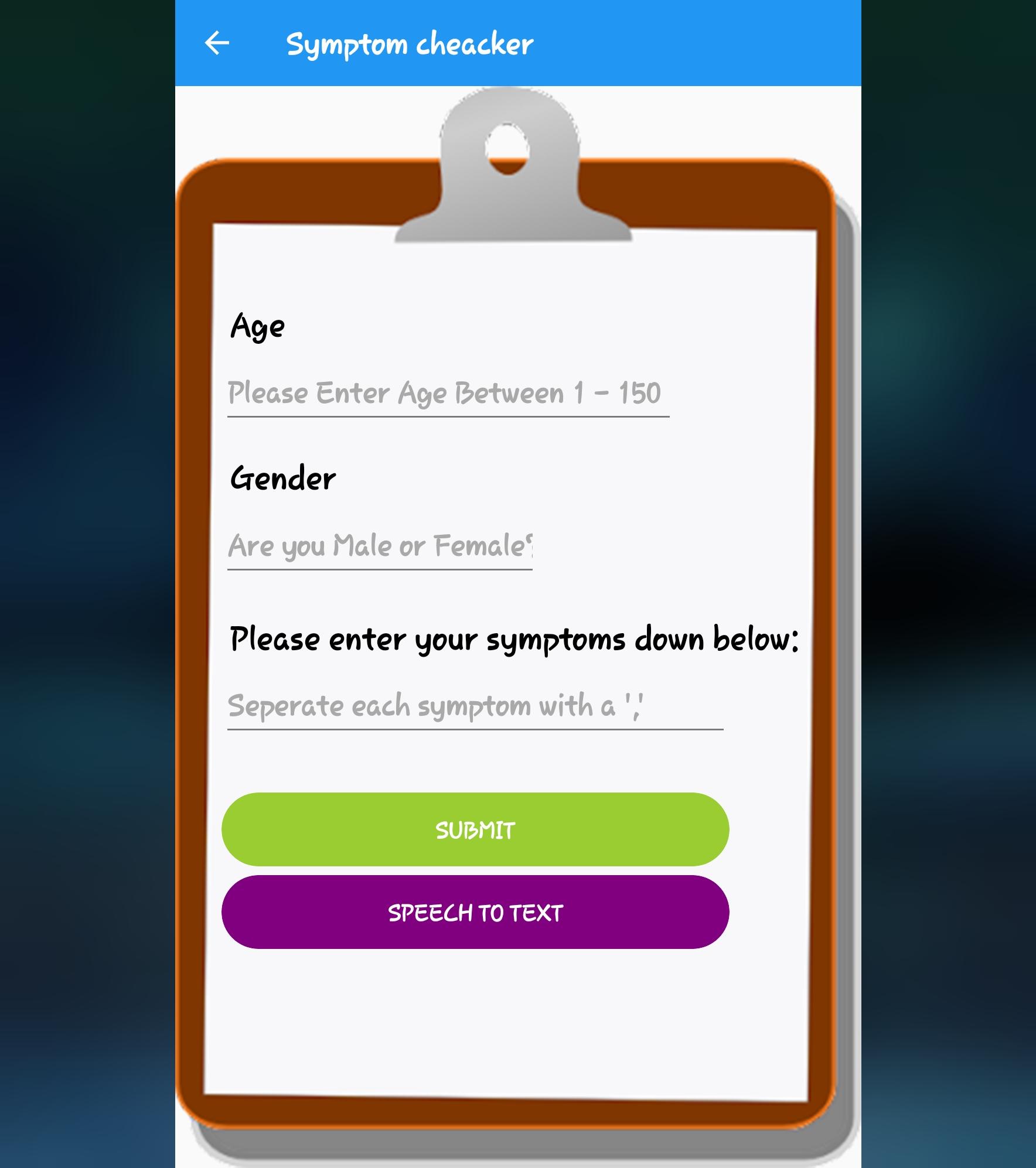


Figure 5.1.5

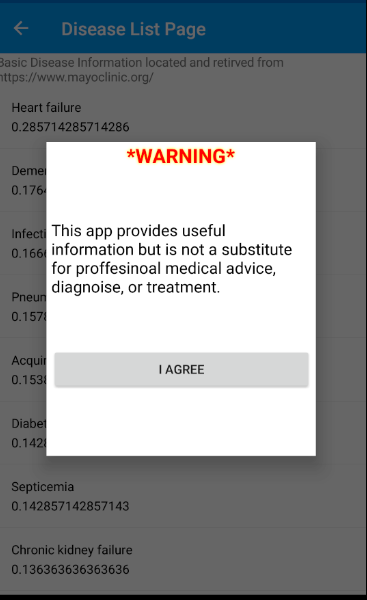
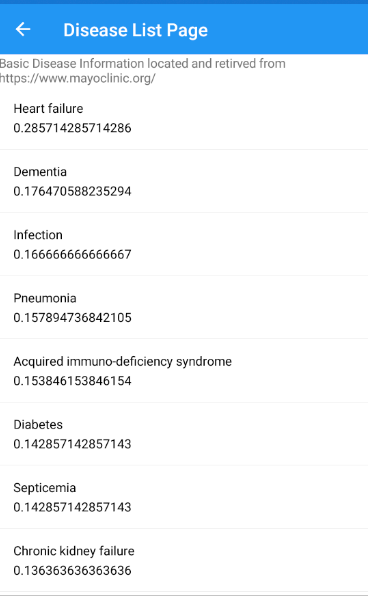
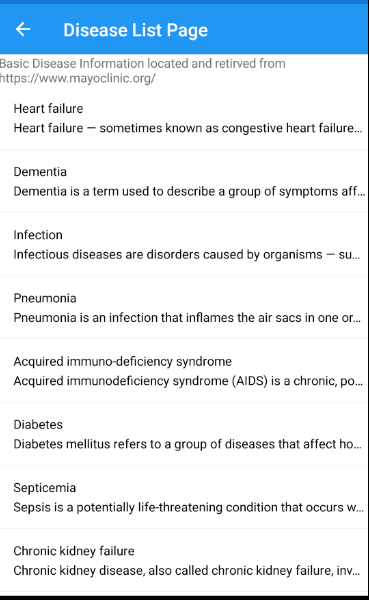
We start by setting our List-display in Xamarin to refresh. This will allow all changes that will happen to the list to be displayed once we are completed with the filtering process. For our simple filtering process, we sort all the diseases based on their specific ranks from highest to lowest. If 2 or more diseases have the same ranking, the list will display them in alphabetical order.

*Entry Form:*

To our left (Figure 5.2.1) we can see the entry forum as it is portrayed in the application. As mentioned in Figure 3.3, we are asking the user to input their age, gender, and list of symptoms. Then the user presses “Submit”, and the user's information is saved to the SQLite database we explained about in Figure 3.3. We also have case checking throughout this page in the forum of warning pop-ups. If the code realizes that any field in the form is empty and the submit button is pressed, then the application will display a popup explaining which field needs to be filled.

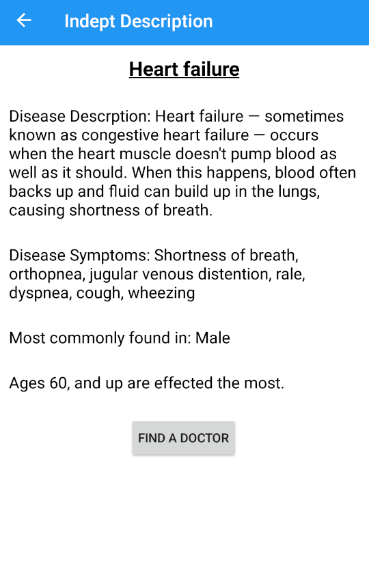
Figure 5.2.1

*List Page:*



(Figures 5.2.2, 5.2.3, 5.2.4) (Left to Right)

When we successfully submit our forum, the algorithm from Figure 5.1.1 will take the information from the SQLite database and match it with the inputs. Then a quick disclaimer will appear on the screen warning the user that they shouldn’t solely rely on this application suggestion as a substitute for professional aid. The user is locked from the information until they have pressed “OK” in Figure 5.2.2, ensuring they have seen the message. Following this we have Figures 5.2.3 and 5.2.4 which show the same data but in different forms. 5.2.3 displayed the name of the disease and its corresponding rank after being filtered and sorted in descending order from largest to smallest. This Figure is our test case and the final product will look like what is shown in Figure 5.2.4. This shows the disease's name and then a short description of the disease under it. In grey text at the top of the list, we have a second informative disclaimer stating that the information on disease has been pulled from Mayo Clinic’s website for proof checking.

*Detailed Description Page:*

In Figure 5.3, we have clicked on one of the diseases listed in the previous page. Here our database populates a template page which is re-used to save time and resources. Every section describes the disease in more depth and finally there is a button at the bottom of the page that can help navigate the user to a professional that handles the specific disease.

Figure 5.3

*Finding the right doctor:*

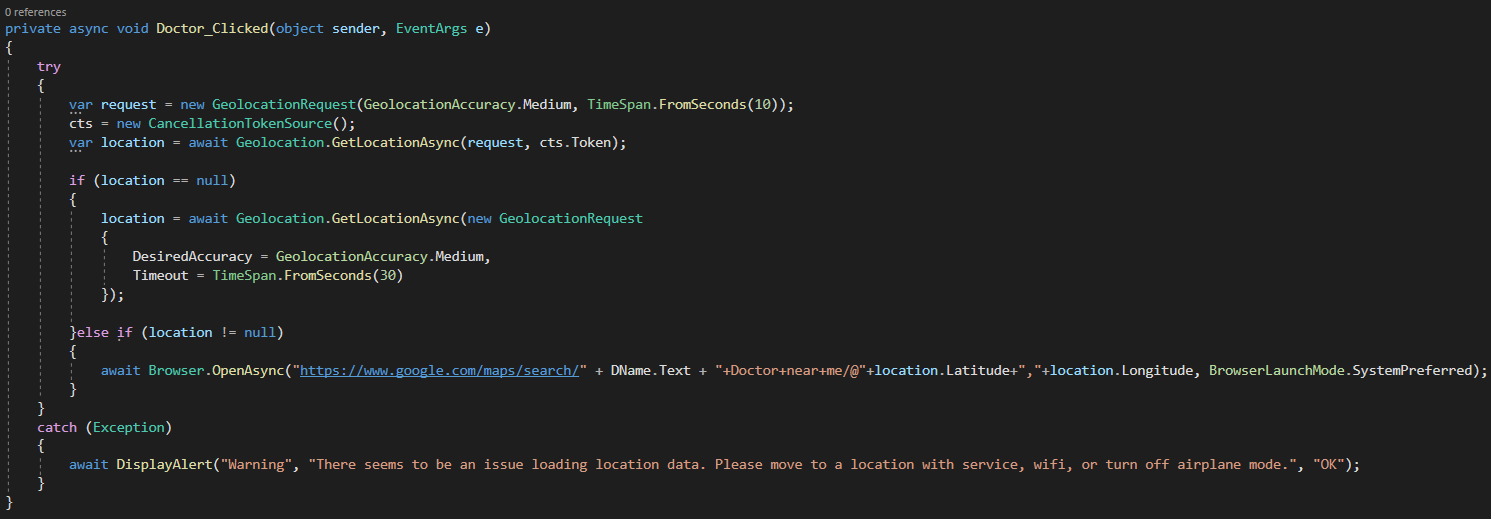


Figure 5.4

In Figure 5.4, we are handling the geolocation request to find a doctor close to the user based on the disease they have selected. We handle the geolocation by asking the user to give permission to location if not already done earlier. This allows us to store the different outcomes like longitude, latitude to a variable called location. The location variable can refresh and update whenever the user wants to find a doctor near them. We have set it to run in a try:catch situation because the user could be on airplane mode or in a location where GPS signals cannot be reached, in which, we display a warning message to inform the user. If we successfully locate the GPS coordinates, the system will open the native maps application on the user's device and use google maps as the default maps application. We pass the string request in a url form with the addition of longitude and latitude, and once this information is passed, the app moves to the google maps browser or application to provide a list of outcomes to the user.

*Covid Api:*

The API we are getting covid information from is “Johns Hopkins University COVID-19 Data in MongoDB Atlas”. Since Johns Hopkins University (JHU) saves all their data as flat CSV files, our user would need to constantly download this large file whenever they wanted to query from it. But MangoDB takes the CSV file and uploads it to their own cloud database, which allows us to quickly query for fast and free data. This database is updated frequently (almost daily), which means the data is uptodate for most or all areas. Our team has decided to use this database to display covid-19 information specifically in the country of the United States, with a focus on New York state, and New York City (NYC). This limits us to only provide information for users in NYC but for this prototype we feel it is a good start.

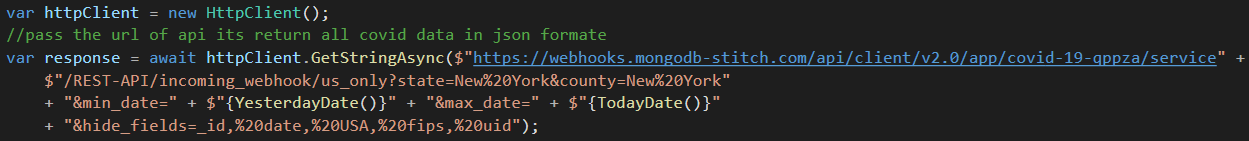


Figure 6.1

In Figure 6.1, we are setting up our connection by running a httpClient() request. We then pass the request as an http request to the database by doing “httpClient.GetStringAsync($"https://webhooks.mongodb-stitch.com/api/client/v2.0/app/covid-19-qppza/service" + $"/REST-API/incoming\_webhook/**us\_only**?**state=New%20York**&**county=New%20York**" + "&min\_date=" + $"{**YesterdayDate()**}" + "&max\_date=" + $"{**TodayDate()**}" + "&hide\_fields=\_id,%20date,%20USA,%20fips,%20uid");”

This URL provides some important information and filtration specification for the information we are querying for. First we start by telling the database we want information about US only, followed by the state of New York, and finally with the specific subDomain of New York. Next we query to get information from the most recent entries, which would give the user the most updated information on covid in that area. This is achieved by passing the users current system time (Year/Month/Day/Military Time). The functions “TodayDate()” and “YesterdayDate()” are explained in Figure 6.2.1, and 6.2.2 below.

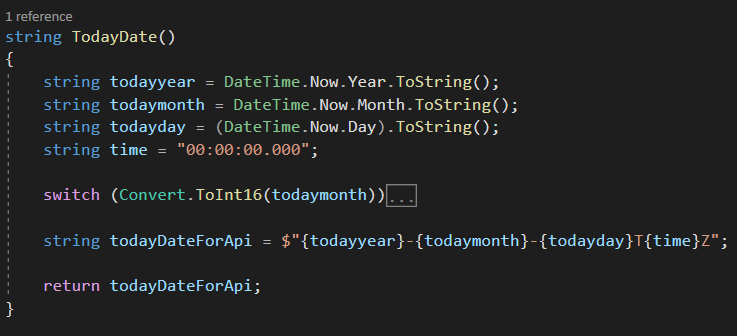


Figure 6.2.1

Figure 6.2.1 shows the function we call to get the system's current time in the format of (Year/Month/Day/Military Time). This format is specified by the API so we string concatenate and return the outcome to the http request.

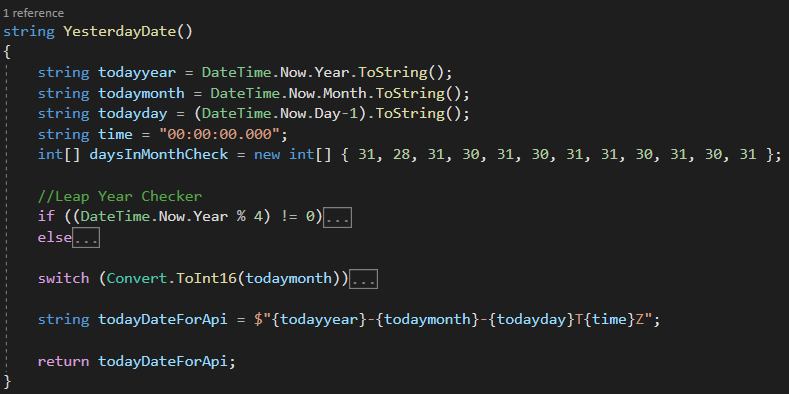


Figure 6.2.2

Figure 6.2.2 displays the functions being used to get yesterday's date. Everything's the same as in the todayDate() function but there are a few more checks included in this one. We added an int array which holds the amount of days in each month because the first check we do is if the year is a leap year. If so, we change the month of February to have 29 days. If not, we proceed to check if the user's current day is the 1st of any given month, in which the day before is supposed to be the last day of the previous month. We do this by setting the month value one less than the current one, but this leads to issues of, “What if the current date is the 1st of january 2022?”. Then we do a large check on all variables by moving 2022 to 2021, and setting the month to 12, and check the int array to see what the last day in December is and we set all that to the variables that are then concatenated to the url.

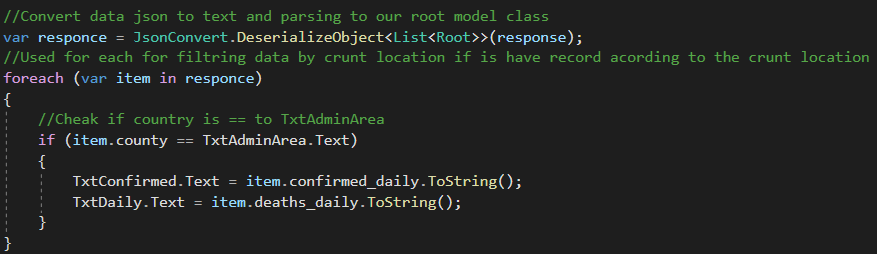


Figure 6.3

Figure 6.3 continues the process of displaying the information to the user after the http request is sent with the filters. Since the http request will return the data in json format, we pass it through our root model class, which makes it easier to read in Xamarin. We do a last check to see if the information from the location specified is the same, and once we pass that we set the text fields that correspond to the confirmed daily cases, and confirm daily death to the information we retrieved.



Figure 6.4 shows an example of information being displayed after being retrieved from the database. As you can see the geolocation populates the location information while the api fills in the confirmed daily cases and daily death field.

Figure 6.4

Speech Removal Text

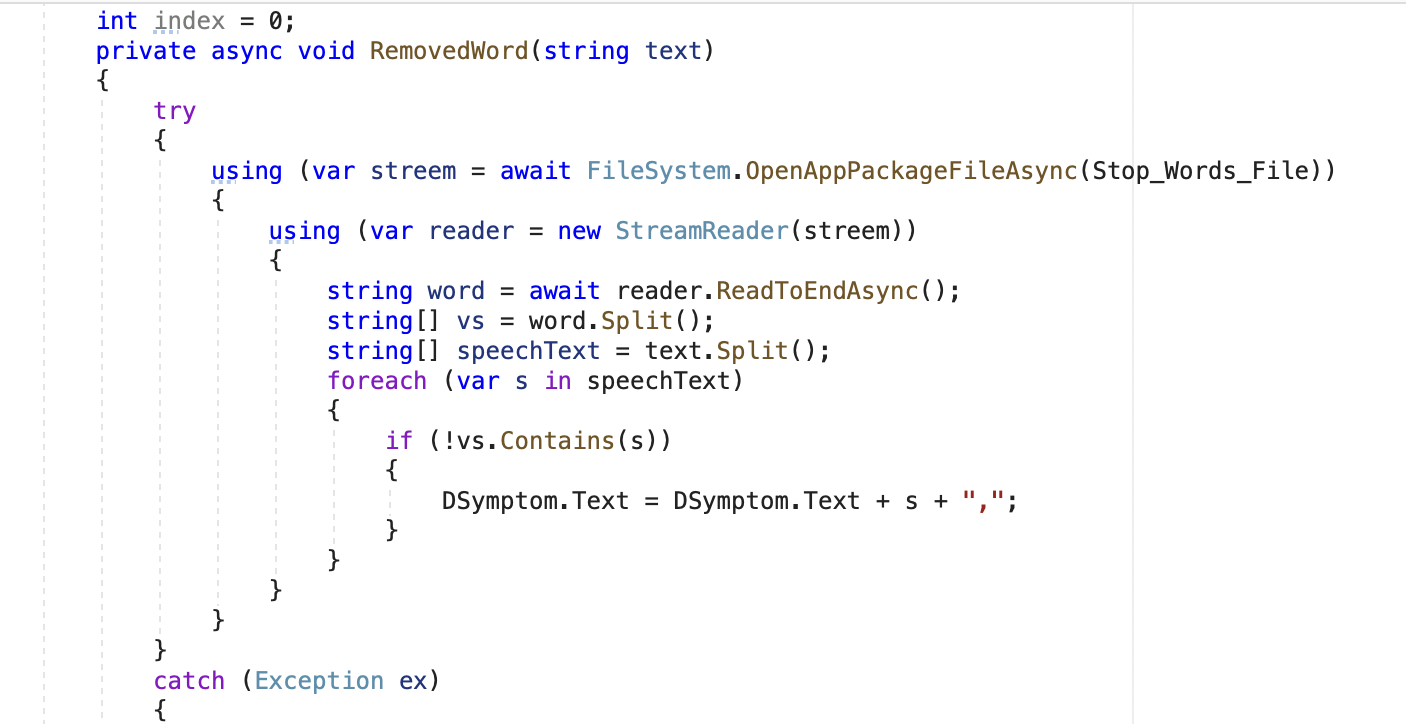


Figure 6.5.1

Speech to text , in Figure 6.5.1 it shows how we created the string for the speech to text. We have created a logic that takes the string of speech and checks the extra words from the text file if it contains these kinds of words(Figure 6.5.2), if it does it is removed from the string and the remaining shows in the test box.

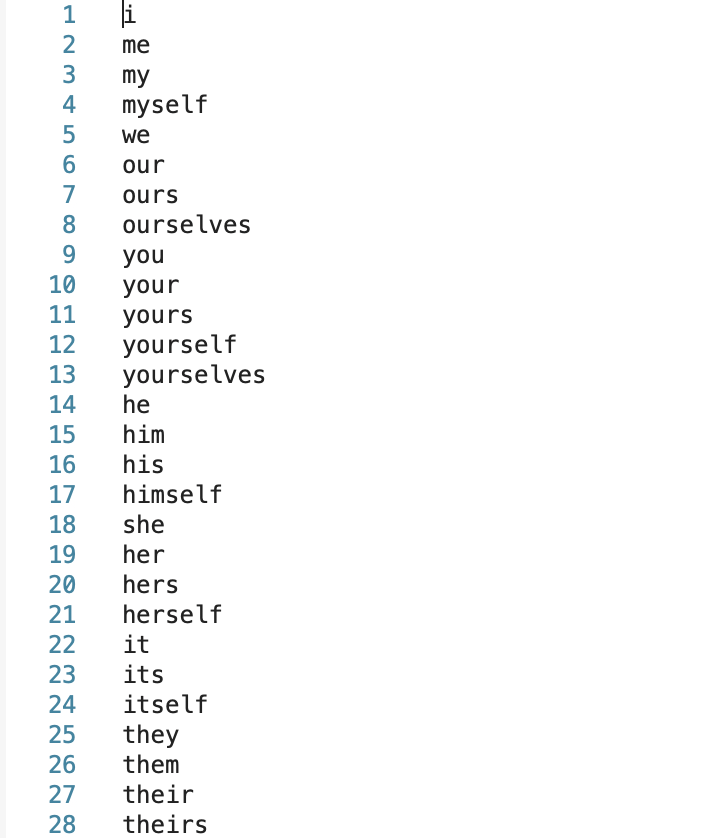


Figure 6.5.2

**6. Unit Testing:**

Since this application is built to be run on both android and ios devices, we have used the built-in emulator for the respective operating system to perform testing. Now we are able to download the applications to our devices and see all the functionality in action. The first issue that arose was that some phones have different securities and required us to change some code to accommodate for this. This happened when Mohammad was trying to connect his phone using GPS because he has an app that allows him to fake his GPS signal. This issue is mostly caused by the user side but can be mitigated by giving a warning for this issue to help the user resolve the issue. Our second issue is that the aspect ratio on new phones is quite different. Mohammad has a galaxy z fold 3, which has 2 unconventional aspect ratios. With some tweaking, the app is able to scale to most aspect ratios besides tablets.

**7. Conclusion:**

With the current standing of the application, we have achieved what we were tasked to do at the beginning of the semester. With the additional features like the disease filtration with an algorithm, google maps integration to find potential doctors near the user, local covid tracking information, and news systems. We were able to learn many technologies like Xamarine and database techniques using SQlite and Firebase and refined our skills on connecting to API(s) that can help with future projects. This application is successful at being lightweight by only needing 50 MB of storage data compared to WebMD’s mobile application that takes more than 100 MB (a fresh application that was never opened).

*Future Plans:*

Some of the plans that were cut from this build due to time and other constraints have been looked into slightly to see if they would be feasible. We could connect 2 different applications together using Firebase because one application can be for administrative purposes and the other for users. The administrative application would be allowed to add, edit and push new builds of the databases and changes needed to be added to the application. Since the administrator can do all these items, the application for the users can be updated constantly with new, refined and updated information. Firebase would allow us to manage everything in real time and provide a better user experience this way. Another great idea that was scrapped was a Over the Counter (OTC) application that would help the user find an OTC medicine that could help them immediately. Being able to run to a drug store if a doctor, or hospital is out of reach is a very important feature that we are sad to remove for this prototype.

A lot of API’s are currently subscription based and come with a fee whenever you want to query from them. For that reason we scrapped the idea of displaying covid news under the covid cases information in the covid tracking section of the application. We can display a simple window for a google search but it does not work best for the information we are trying to portray in a simple manner.

**Meet the Team:**

*Mohammad Ishtiaq (Senior)*

The team leader was Mohammad Ishtiaq, who is currently a senior here at NYIT. This senior project experience has been a rough one. With team members not completing tasks on time, changing of schedule to make time to meet outside of class, and the mis-communications between team members. This was the first time doing mobile development on any platform or for any course, so the material needed to be learned was vast. Overall I think I have learned every technology listed in the *Technologies* section of this report. Xamerian was a big disappointment because even though it can work on most operating systems, it has a vast amount of limitations because of this. But I was able to learn, and experiment with new technologies and find a deeper understanding of algorithms, databases, and APIs. My groupmates contributed but it wasn’t enough to finish the project with the projected goals at the beginning of the semester. I hope this project is able to help my groupmates understand the importance of teamwork and making time for a course because the time management was all over the place.

Jobs:

* UI Design
* Database (Disease & User)
* Algorithm design
* Google Maps Integration
* Unit Testing
* Technical Report:
  + Abstract (Full)
  + Introduction (Except Existing Applications)
  + System Analysis (Full)
  + Database Design (Full)
  + Functionality & Implementation (Except Covid Api)
  + Unit Testing (Full)
  + Conclusion

*Nusrat Bainya:*

Was incharge of creating the Covid-19 tracker application. This evolved to an additional Covid-19 new space as well because we had enough time.

Jobs:

* Covid-19 API
* Technical report:
  + Existing Application
  + Covid API explanation

**Citation**

“ASKMD - Get Answers - Manage Conditions and Symptoms.” Sharecare,

www.sharecare.com/askmd/get-started.

“Coronavirus Smartable API Documentation (Smartableai).” RapidAPI,

rapidapi.com/SmartableAI/api/coronavirus-smartable/.

“GitHub Desktop.” GitHub Desktop, desktop.github.com/.

“Mayo Clinic - Apps on Google Play.” Google, Google,

play.google.com/store/apps/details?id=com.mayoclinic.patient.

REST API Mongodb Covid-19 Open Data,

documenter.getpostman.com/view/1678623/SzfDx54T?version=latest.

“Symptom Checker with Body from Webmd - Check Your Medical Symptoms.” WebMD,

WebMD, symptoms.webmd.com/.

“Symptom Checker.” Mayo Clinic, Mayo Foundation for Medical Education and Research,

www.mayoclinic.org/symptom-checker/select-symptom/itt-20009075.

“WebMD: Symptom Checker - Apps on Google Play.” Google, Google,

play.google.com/store/apps/details?id=com.webmd.android.

“Xamarin.forms: .NET.” Microsoft, dotnet.microsoft.com/en-us/apps/xamarin/xamarin-forms.