1. Domain:

1.1 Application domain:

The domain consists of Determining nutritional values from meals,meals from nutritional values, and meals from ingredients.

1.2 Goals fulfilled by developing implementation:

The goals of the implementation are to:

- help users track all of their food's nutritional values, including a large list of nutrients and homemade meals,
- help users track their nutritional goals and their progress towards them,
- generate meal plans for homemade food to fit the user's nutritional goals and ingredients.

1.3 Stakeholders:

1.3.1 Producers of the software (Tawsi Studios)

The goal of Tawsi Studios is to have well defined specifications for production, as well as a product that follows the guidelines as well as possible for maximum marks and is unique and useful enough to export to the Android store.

1.3.2 Project markers (Dr. Reza Samavi, TAs)

The goal of the project markers is to receive code that is easy to navigate, understand, and test.

1.3.3 End users

The goal of the end users is to get a reliable and easy to use application. This means that a user unfamiliar with the specific implementation, but familiar with general app usage and nutrition tracking should be able to use the application with minimal instructions.

1.4 Entities:

- Meals
- Meals eaten
- Ingredients
- Ingredient nutritional values
- Application user
- User’s nutritional goals

1.4.1 Entity relationships:

Meals are composed of ingredients, and are either generated by the system or manually constructed by the user. All ingredients have pre - defined nutritional values per set quantity of ingredient. The application user takes part in determining the nutritional value goals, and makes the decision on what meals become meals eaten. The nutritional values of ingredients are matched to the user’s nutritional goals.

1.4.2 Ways in which entities are affected by system:

- The user’s nutritional goals are recorded and updated based on user need.
- The user’s meals eaten are recorded for user information and goal calculation purposes.
- The nutritional goals are modified by previous meals and how much they meet the goal.
- Meals can be generated by the system based off of a couple of user - given ingredients, and their nutritional goals.

2. Functional Requirements:

2.1 Food Information Retrieval by Name

The user must be able to manually input a collection of prefered foods into their meal plan. These prefered foods and their nutritional information should be able to be found by a name search within the existing entries in the dataset. This will be ensured through the combination of a prefix tree, and search autofill functionality. The prefix tree will be used to narrow down search results by the first few characters, then the autofill will present the users will options based on the dataset entries. This will all be performed in the back-end.

2.2 Meal plan storage

2.2.1 Option to save favourable meal plans

The user must be able to store a provided meal plan within the application for future consultation, as well as the last meal plan generated in order to prevent data loss in the event that the application is exited prematurely. This will utilize React Native's asyncstorage functionality by storing the values in a key-value pair that can be parsed into JSON then retrieved and presented to the user in a way they can read and interpret.

2.2.2 Option to compare saved data with goals

The user must be able to set nutritional goals based on the nutrients located in the dataset, which will be stored in asyncstorage in the same fashion that the meal plans will be stored. The user must be then able to compare all saved meal plans with this nutritional goal, and the application will return the progress the user has made towards their goal. This will be in the form of a percentage and will be calculated in the back-end by accumulating the nutrient amounts from previous meal plans in storage then dividing the sum by the goal.

2.2.3 Deletion of saved data

Just as the user must be able to save meal plans and goals, the user must be able to delete unwanted goals or plans. This must be accompanied with a feedback modal confirming that the user wants to delete the item, seeing as how once it's removed from asyncstorage, it cannot be recovered.

2.3 Meal planning

What can be viewed as the primary function of FoodFitter, the user must be able to acquire a set of foods that best match their nutritional goals or preferences. As described in Requirements 2.2.1 and 2.1, the user will search for and add manually foods that they prefer to eat. Then, the application will send a request to the back-end to fill in the remaining foods that are able to meet the user's nutritional requirements. Prior to submitting the request, users must be able to select the maximum margin of error, and their nutritional requirements (whether that be in a saved goal like in Requirement 2.2.2 or a new one added).

2.4 Performance of calculations in the backend

As mentioned in Requirement 2.3, the back-end must return a list of foods that satisfy the user's constraints. These calculations will utilize a greedy approach to the fractional knapsack problem (otherwise known as the "knapsack algorithm"). The idea is with the greedy approach is to sort the subset of items (those with the required nutrient information) by the weight/value ratio. Then, the items are added in descending order. That way, the nutrients will be maximized with a very small error rate. The whole problem can be solved in n log n time (Trivedi, 2018).

2.5 User Authentication and Storage Access

All information used in the application must be stored locally as stated in Requirements 2.2.1, 2.2.2, and 2.3. In order to use the application, the user must create and enter a 4-digit pin in order to confirm their identity. This can be simply implemented by storing the pin as a key-value pair in asyncstorage and comparing the entered value with it whenever the application opens. Due to space constraints, only one user is permitted per device. The pin is in place to ensure other individuals who use the device do not open the application and view the user's personal information, nor do they change it.

3. Non-Functional Requirements:

3.1 Security

3.1.1 Method of personal user information storage

Privacy and security of personal user information (i.e. email, full name) must be ensured by two qualities. The first being that all storage is local to the device. The second being that only one user is permitted per device, and that user can only use the application after entering a pin. That way, the only authentication required is a pin and the risk of two different users viewing each other's information is non-existent.

3.1.2 Method of user preference storage

Similar to personal user information, privacy and security of user preferences (meal plans, nutrient requests) must be ensured through the local nature of the device storage, the single user restriction, and the pin required to use the application.

3.2 Capacity/Computations

3.2.1 Maximum, average, and minimum concurrent users

The back-end must return the meal plan within a tolerable amount of time. Research shows that users are willing to wait an average of 37.6 seconds with feedback for a "quick" task to be completed, such as clicking on a hyperlink (Nah, 2004). If a request to fetch the information from the data set takes 1 second, then the average expected number of concurrent users for the early stage of development should be 37. The maximum number of concurrent users doing a request should be capped at 50. Finally, the minimum number of concurrent users should be 16 (Nah, 2004).

3.2.2 Storage saving approach to meal plans

The only generated meal plans that should be saved are either those the user requests to save, or, the last meal plan generated to supply a "recovery" option to users if they exit the application prematurely. If the user decides to not save a specific meal plan before exiting, the meal plan should not be committed to asyncstorage (as outlined in Requirement 2.2.2)

3.2.3 Device memory limitations

Taking into consideration that most modern applications on Android are restricted to a maximum usage of 32 MB of RAM (Dubroy, 2011), the application is not permitted to rise above this limit on either platform. This is required even though many modern phones and tablets have

upwards of 1 GB of RAM because the application must be backwards compatible. Many older phones have a maximum of 192 MB of RAM (Dubroy, 2011), and as a result begin to slow down considerably when a certain percentage is being utilized (>17%).

3.3 Performance

3.3.1 Maximum back-end request time

As outlined in Non-Functional Requirement 3.2.1, the users shouldn't be waiting more than 50 seconds for a retrieval from the dataset. In addition to this, the back-end software must perform the calculations for the algorithm, which runs in n log n time. Due to the size of the dataset (153 nutrients and over 5000 individual foods), it is reasonable to cap the request time to be 5 minutes at the absolute maximum. Realistically, this is the worst case scenario in which every food in the data set has similar nutrient information, which is highly unlikely.

3.3.2 Restriction on usage of device resources

Utilizing system resources is required, but very taxing on the device. FoodFitter does not utilize many resources, like music or entertainment applications do. It does still have to utilize the wake_lock, write_external_storage, read_internal_storage, toast_window, system_alert_window, write_clipboard, read_clipboard and vibrate resources in the case of an Android device. Since the cap on fitness applications in terms of daily unique resource storage is 11 resources (Fritsch et al, 2017), FoodFitter does not need to be concerned with its resource usage.

3.4 Reliability

3.4.1 Guarantee of genuine data

The guarantee of genuine data must be ensured. This will be done through transparency of the dataset. Since it is an open dataset, there is a document to evaluate the results against. This will reassure users that the information they receive is genuine, up to date information they can verify themselves.

3.4.2 Existence of an error log

In order to isolate problems and determine which element of the system is behaving abnormally (or in other words, unreliably), an error log should be stored in asyncstorage. Whenever the application is unable to perform a function, the user should be able to send the system log to support, which will detail why the error occurred and when it occurred.

3.4.3 Minimal margin of error

The reliability of the application must be tested through the quantitative comparison of the achieved margin of error, and the prefered maximum margin of error provided by the user. If the achieved margin of error continuously exceeds 40% for over 10 different nutrient specifications, then that shows that the accuracy of the results may be poor and the reasons for such must be discovered and their impact must be minimized.

3.5 Maintainability

3.5.1 Ability to handle an update/expansion/replacement of the dataset

As time passes, more food options and nutrients will be added both to the existing dataset and other viable datasets. A time will likely come where the current dataset will be replaced with another one, most likely of a different format. The application must be able to handle this change without having to have the developers make drastic changes to the back-end implementation. The module for reading the csv file should have as little coupling as it can manage so that it can be modified and the new update can be rolled out posthaste.

3.5.2 Ability to be backwards compatible with older mobile operating systems

Taking into consideration that the user demographic will be quite large in terms of age, users will have anything from the newest phone or tablet to a first edition device. FoodFit must avoid relying heavily on device features unique to newer operating systems (i.e. apple's health application). As it currently stands, FoodFit must remain as isolated as possible and function as a truly stand-alone application.

3.5.3 In-depth documentation for both the end user and developers

The documentation provided for the application, both on the end user side and the developer side, needs to be easy to read, descriptive, and frequently maintained. This way, any changes made to the application can be introduced in a way such that new developers and users can understand and work with the change.

3.6 Usability

3.6.1 Accessibility of fonts

Every font utilized within the FoodFit user interface must be easy to read for all users, taking into account those with dyslexia or other difficulties with reading ornate fonts. For this, fonts such as helvetica are recommended or even comic sans due to the fact that it "weighs down" letters to make them easier to read.

3.6.2 High degree of background-text contrast

Any modal or paragraph of text displayed must have a high degree of contrast between the background and text displayed. This is not only helpful for those with reading difficulties, but those who are colour blind or who have mild visual impairment.

3.6.3 Presence of intuitive keystrokes

Every feature in FoodFit should be accessible through a button. Many applications nowadays make use of keystrokes such as a double tap or pinch. While these may be a nice addition, it is best to simplify the interface as much as possible to accommodate a wider range of users.

3.6.4 Screen-reader compatibility of buttons and text

Screen positioning of buttons and text should be in a manner such that a screen reader can highlight the entire screen and the user can logically determine what the paragraphs of text are trying to say and also locate the buttons to move to the next screen. Unnecessary symbols (such as the trademark or @ symbol for instance) should be avoided, because they will be recited in an unfamiliar or confusing manner.

3.6.5 Visibility of help documentation

The help symbol should be clearly displayed on all pages in a visible area. It should also be symbol that evidently conveys help (such as a question mark) so any user can recognize it and press it for help documentation.

3.7 Portability

3.7.1 Cross-platform compatibility

FoodFitter should be able to run on both Android and iOS, the two most popular and widely used mobile platforms. This way, a wide array of users are able to use the application to its fullest and have their experience tailored for the features of their mobile operating system.

3.7.2 Multi-device compatibility

FoodFitter should be able to run both on a tablet (which could be of varying dimensions) or a phone (which also could vary in dimension). This should be achievable without distorting any text or imagery used in the applications user interface.

3.7.3 International localization

All text displayed in the application should be stored in a file in such a manner that another language can be added and the implementation shouldn’t have to be changed. That is,

a reference to the text should be passed and not the actual words itself. This way, users will be able to use the application from a number of different countries worldwide.

4. Requirements on the development and maintenance process:

4.1 System testing

4.1.1 Unit testing

Unit testing will be performed on every function in the system during the development process using JUnit testing suites. Each function shall pass on every test case (a total of five per function, one created by each member of the team). This will ensure that the components function individually.

4.1.2 Interface testing

The user interface will have been tested to the satisfaction of 10 randomly selected users of different demographics. These users will experience the application and grade the interface on the requirements listed in section 3. The interface will meet at least 6/10 on average, for each of the elements listed under section 3: non-functional requirements.

4.1.3 Integrative function testing

The system will be tested on its satisfiability of the functional requirements to the satisfaction of 10 randomly selected users of different demographics. These users will experience the application and grade the interface on the requirements listed in section 2. The interface will meet at least 8/10 on average, for each of the elements listed under section 2: functional requirements.

4.2 Priority of functions

The function of utmost priority, and shall be first in development, is the food name retrieval function (2.1). This is due to many other functions relying on it in order to meet their respective requirements. The second priority is 2.4 Performance of calculations in the backend, because of the same argument. 2.2 Meal plan storage is the next priority because it is dependent on the previous functions, but further functions will depend on it. For example, it must employ food name retrieval to access data about the foods, and calculations on that data to display relevant information.

Following the development of meal plan storage, 2.3 Meal planning will be implemented because it depends on the aforementioned functions. A barebones functionality can be

achieved with just these functions listed so far. The last priority is thus 2.5 User authentication and storage access, which will enrich the application experience but not take away from its usability.

4.3 Anticipated changes

4.3.1 Database updates

Due to continued research, the nutritional values of foods may be updated in the future. Or else, innovative techniques will create new types of foods that may become popular. The database must be able to accommodate such changes, and the application must still work despite the additional of new foods or the updating of their nutritional values.

4.3.2 Hardware changes

Mobile phones and operating systems are constantly updated. The app must be continuously maintained and tested on newer devices and operating systems, as per the requirements outlined in 4.1, in order to ensure usability in the future.

References

Dubroy, P. (2011). *Memory Management For Android Apps*. [ebook] p.3. Available at: https://dubroy.com/memory_management_for_android_apps.pdf [Accessed 25 Feb. 2018].

Fritsch, L et al. (2017). *How much Privilege does an App Need? Investigating Resource Usage of Android Apps*. [PDF] Karlstad University, p.5. Available at: https://www.ucalgary.ca/pst2017/files/pst2017/paper-63.pdf [Accessed 25 Feb. 2018].

Nah, F. (2004). *A study on tolerable waiting time: how long are Web users willing to wait?*. [PDF] p.26. Available at: http://sighci.org/uploads/published_papers/bit04/BIT_Nah.pdf [Accessed 25 Feb. 2018].

Trivedi, U. (2018, January 10). *Fractional Knapsack Problem*. Retrieved February 25, 2018, from https://www.geeksforgeeks.org/fractional-knapsack-problem/