Total Life Manager (TLM)

Logan Grisham, Sebastian Hardesty, Stephen Johnson

1. **Introduction**

**1.1 Abstract**

Total Life Manager is a streamlined web server application that allows users to track nutritional and health goals. The website provides tools to a user to identify habits and plan for future health milestones. Along with tools for tracking caloric intake and exercise, the website also assists users in keeping track of more general health statistics. The website tracks all the user's entered information and builds charts and graphs to help the user digest the information instead of getting confused or discouraged. Users can benefit from the site as a whole-life tool because important information to daily life, such as the weather, and a calendar are available on the dashboard.

**1.2 Purpose**

The purpose of Total Life Manager is to provide a service where a user can track their health habits and information. This service allows users to see their data and display it in a digestible system of charts and widgets. The daunting experience of working on health and habits is eased when the ability to visually understand trends and data is introduced. The entire goal of the service is to give users a method of approaching their personal health goals and identifying positive habits with the tools provided by the service. This will aid users in tracking their data and reaching the goals they set. The purpose the platform serves caters to a wide audience, whether experienced in health goals and achieving milestones, or not. The average user is expected to not have much experience, so the user interaction portions of the web application are simple in nature. They accept input in basic numbers and run all the computation for graphs and trends behind the scenes. This supports the purpose and intent of the service by keeping it accessible to all potential users and target audiences.

The system’s overall design is a web application that hosts a number of services to a user to track their health and habits. This service is centralized around a dashboard that the user can see the majority of their information from at a quick glance. The individual dashboard modules will come in a variety of different widgets placed around the screen. The main widget will display the user’s exercise, hydration, sleep, and calorie intake. The exercise will be in minutes, the hydration in ounces, the sleep in hours, and the calorie intake in calories. These will display rings that show a percentage of completion for the user’s individual goals. If a user has completed 10 minutes of their set 30 minutes workout goal, then the ring will be 33% of the way filled in a clockwise manner around the ring. The dashboard will also display a widget for the user’s graphs. This shows a preview of a user’s graphs and leads directly over to the graph page. The next widget is a weather API that displays basic weather information for a user so they can plan their day. Lastly, there is a widget for a calendar. The user will have a basic view of a calendar and be able to go to a full view that shows the user slightly more detailed information about events or other data that has been placed on the calendar. The entire website is reachable once the user is logged in through a convenient hamburger menu on the top of the page. This provides navigation to all the pages a user can access on the website, including a logout option to log the user out of the website. Additionally, the user can record information on the website through the graphs screen. The design keeps the desired interaction between user and graph in strict cohesion, so the user never misses the immediate opportunity to see their data represented visually.

**1.3 The Problem it Solves**

The problem Total Life Manager is seeking to address is the utterly overwhelming nature of health and fitness. Tracking data and information comes in a variety of ways, but most methods result in confusion, a lack of understanding of change, and lessening the threshold to achieve goals. This means more broadly that without a good way to track health and fitness information, a person can get discouraged easily and is more prone to give up. The solution that TLM posits is a simple way to track and show data to a user while taking the heartache out of interpreting and storing the data. This has been done through beautiful graphs, data visualization, and information clustering. In general, the user’s data is presented to them on a system of rings and graphs, and additional aids are present to help a user get all the necessary info for health and fitness goals in one spot.

1. **Technology**

**2.1 Technology overview**

The technology the system is predicated on has its foundation rooted in the LAMP stack. The LAMP stack consists of Linux, Apache, My Structured query language (MySQL), and PHP Hypertext Processor (PHP). The components each have their own characteristics and avenues of change that make the system function in its totality. To start, the linux component is key to the server and the hosting service the project lives on. Currently, the server resides on an Ubuntu Linux distribution that we have hosting the apache server. Naturally, next comes apache in the stack, which is the web server, and is the most common web server for linux powered web applications. A notable backend process is the MySQL component in the stack, which is hosted on the apache server. The server hosts a MySQL database which is critical for the operation of the web application. In terms of the stack, the final component is PHP. This is the server language that communicates between the frontend and backend, allowing the stack components to all work together.

The technologies used to power this product are as follows:

1. Platform: Firefox

2. Operating System: Windows 10

3. Languages: HTML, CSS, Javascript, PHP, MySQL

4. Libraries: Chart.js

The LAMP stack hosts the service and main functionality of the backend, but the list above is composed of the technologies that were used to construct the service itself. Starting with the platforms, the majority of the development has happened on FireFox and Google Chrome, with the exception of the graphs section, which was more accessible on the FireFox browser. The difference in the browsers in terms of development was negligible, since the material was primarily being displayed in the browser and edited in the Integrated Development Environment (IDE). The IDE the group defaulted to primarily for the development of the system was Visual Studio Code (VSCode). The browsers were geared toward facilitating the IDE’s output so were interchangeable. The physical operating system the program works on was both Windows 10 and 11. The service is a website, so it is accessible on both the platforms regardless of the native differences. Notably, the entirety of the backend and setup work was done from a Mac device but the display of the site will break on Mac.

The languages used for the project are standard in the industry for the development of websites and web applications. HyperText Markup Language (HTML), Cascading Style Sheets (CSS), and JavaScript (JS) were the backbone of the project. These three are the standard languages for web development, and were used to design and implement the early stages of the project. When development skewed down the path of database integration, the team quickly picked up the next web and server language, PHP HyperText Processor (PHP). PHP is used to communicate between the frontend and the backend for this service, and most notably is used for communication to the database. The database itself is a MySQL database, hosted on the linode server. The database was originally built on a secondary application called XAMPP that is used for PHP, Apache, and MySQL in the case of our specific requirements. Once created with this tool, the database was exported over to the live server and used for testing and then ultimately the production scenario.

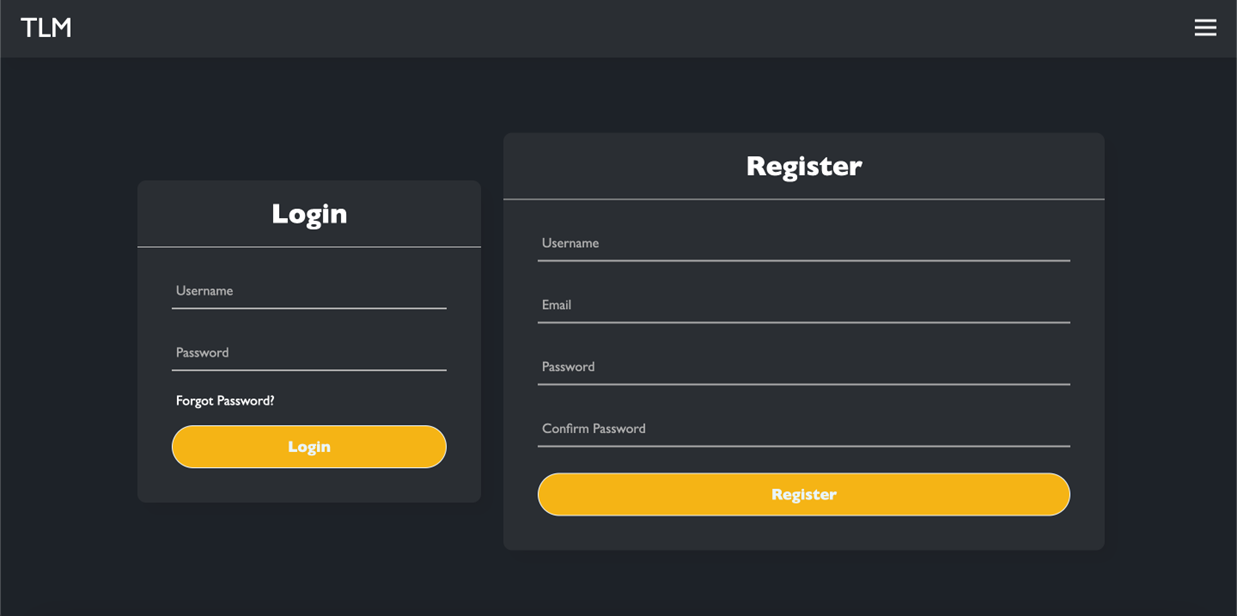
**2.2 Difficulties coming up to speed.**

The Technology was not the easiest adaptation for our group, as we are not all web-based developers. The first technology group that was difficult to adjust to was the server and server languages. As the first hurdle the group had to overcome, the server setup only created a mild amount of work to be solved. This was due to the team’s lack of experience in specifically Linode, the server host, and how to host a web server off of the service. The initial setup of the Linode server consisted of installing the linux distribution, which in this case was Ubuntu 20.04 LTS. The group member concerned with server setup happened to have Linux experience, so the hosting off of Ubuntu was not a problem in the slightest. The early resistance came when installing the rest of the stack on the Ubuntu distribution. This was the apache components and the MySQL database. Learning how Linux used these tools and how to run them through the command line was certainly foreign to the team, but not overtly inaccessible.

Next on the list of technologies was the basic web languages: HTML, CSS, JS. The web languages are familiar to everyone in the group, but as software engineers, the group has only had a minute amount of experience with them. The HTML and CSS were key to get configured early-on, and by configure what is meant is the generalized structure of the website and styling. The structure was important to guide the conceptualization of components, such as login and register systems, as well as widget components on the dashboard. The styling seems like a disjoint part of the process at first, but in reality it helped shape the components at a similar level to the HTML. This is due to understanding where and how components worked, as the introduction of CSS generally made the experience easier to understand as opposed to blank boxes with no order. Javascript was not used as much in the main construction of the website, and is more specifically for the graphs section.

This technology was hard to spin up on, since the JS was the host for the external library the team utilized, Chart.js. Chart.js was a serious hurdle to cross for the team, and was possibly the most significant barrier to success that was encountered. The library is a vastly simpler way to visualize data, and powers the graphs page, widget, and data rings. This was integral to the success of the project, but was completely alien to the team in terms of experience. The gap in knowledge is more apparent when considering that the team already had only minimal experience in web languages. This library has an abundant amount of support for graphing and user interface (UI) customization. A key idea in making the data easy to see is facilitated by Chart.js’s togglability for variables on a graph. This feature posed several pertinent issues, such as needing to determine graph limits and scalability. This was most notable on the graph showing all data sets combined, which includes a wide range of input values.

Another key technology that the team encountered a significant amount of friction on was, simply put, the integration of the frontend and backend communication. This happens on the project with the use of PHP as a technological glue between the two ends. PHP takes action when the web pages are accessed and is called when tasks involving the database are needed. This is clearly a critical system function and without it, the system will not work. The language was entirely new, and therefore hard to spin up, and its legitimate application was confusing to the team. Eventually the team switched the entirety of the HTML files over to PHP as conceptually the team had no idea how to implement it separately. The purpose that the new files serve is the same as HTML files and even uses the HTML code just as the old files, they just have a new language as the file extension allowing the php code to be run on every file. This was hard to spin up on, as the team needed to use things like session checking to ensure data is not accessible without the user being logged in.

*Figure 1. The Login.php Screen.* 

The majority of the early PHP issues came from this login screen above. Originally the team was unaware how to communicate from the login and register boxes to the database. The language has a variable amount of approaches to this when it comes to MySQL databases. Over the course of multiple tutorials and examples, there were over 4 different ways to do a task as simple as connecting to the database. The confusion around the language primarily comes from the use of different constants that have a shocking lack of documentation. Once the confusion was sorted out, the system was online, but much later than expected.

**2.3 Did the technology solve the problem?**

The technology that we chose seems mostly adequate at solving the problem at hand. Solving the problem is also notably just making a service that accomplishes a goal. The overall technology in the stack does indeed serve as a steady base for the project however. The linux server and the components involved are all adequate for the website to run and operate on seamlessly. The basic web languages are a requirement for a web project, and weren’t necessarily a choice we could avoid as long as we stayed “stock.” A technology choice the team could have made would be a different stack or collection of components. In addition to the basics, the group picking Chart.js solved an immense amount of drawing and graphing issues, albeit causing issues of its own. The utilization of API’s further expanded the toolset the team had, and allowed the import of things such as weather data for the weather widget on the dashboard. This technology certainly improved the quality of the website and accomplished the goal of incorporating weather information to the application.

**2.4 Changes in technology, and Why.**

The topic of changing technologies was discussed several times, and even later in the development cycle than is normally expected. Initially the major technology changes proposed were a difference in entire stacks. The original proposal was the LAMP stack that the team ultimately implemented for the project. While starting the early design, the team realized that we may be able to do more complicated things and could fully implement the widget system using the React.js framework. React.js is a javascript library that would make implementing containers easier and more effective, but the team had zero experience in the library or its structure and implementation. This was what fueled the opposition to LAMP architecture, by way of the MERN stack. MERN is comprised of MongoDB, Express.js, React.js, and Node.js. This stack is clearly more oriented to web development through javascript libraries and a new database style that implements NoSQL.

That makes this stack not only a new style, but all the four technologies were completely unknown to the team. This meant that although an understanding of the potential capabilities was present, the skill to implement the technologies after an extended period of learning them would be more of a hindrance than anything. The team decided that the LAMP stack would be more fit to convey the project in a timely manner. However, as the project progressed, the team occasionally entertained new and more relevant technologies, including readdressing the MERN stack, but this was ultimately again trumped by the LAMP stack. The team believed this was the best course of action due to the cost of change, lack of experience, and inability for immediate support.

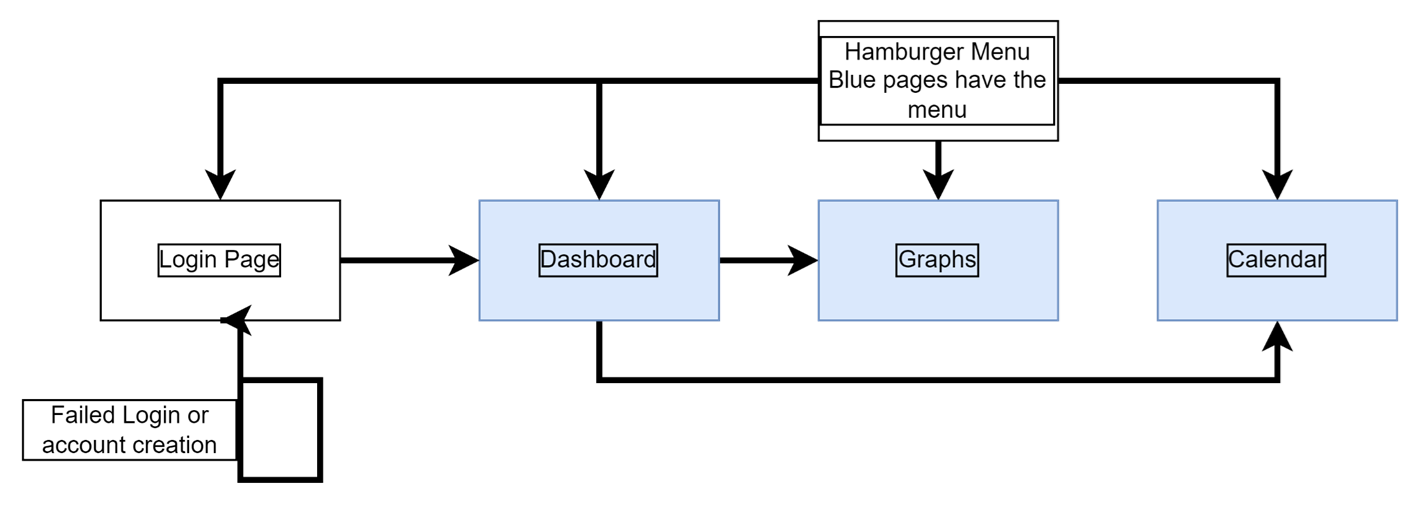
The design feature that shifted the tone of the graphs and charts was the inclusion of Chart.js. Originally the group was going to draw the graphs manually, or use some framework from the MERN stack but this would’ve caused a whole host of unfamiliarity and confusion. The inclusion of the Chart.js library meant that the project could build and graph data much more seamlessly. So while not necessarily a replacement of technology, the inclusion of Chart.js was a technological adoption that superseded the use of base Javascript through means of extending the functionality of it.

1. **Design**

**3.1 Brief design overview**

The system design is a collection of pages to span the content of the website. In order to describe the design, there will be a brief overview of the system and then an in-depth description of all the components.The system design changed over several iterations of additions and reductions in pages and functionality. The main notable pages that were present from the start and stayed through the entire design are the login and register page, the dashboard, and graphs page. The addition to the system to supplement the navigation of pages with a more accessible interface was the hamburger menu. As long as the user is logged in to the website they can access the menu from any page with the exception of the login page, as this could cause bugs.The calendar page has also been in and out of functionality. Below is the block diagram of the applications page interaction and reachability. The display shows the page layout, and the access that the hamburger menu has across the application.

The diagram page all users first access is the login page, where the user can login and register, which subsumes the old landing page that was taken out of the page deck. Once logged in, the page directly links to the dashboard, which is a part of the section of blue pages in the diagram. The blue pages represent the pages accessible by the hamburger menu. All of these can access each of the other blue pages, as well as the login page, via the logout option on the bottom of the hamburger menu.

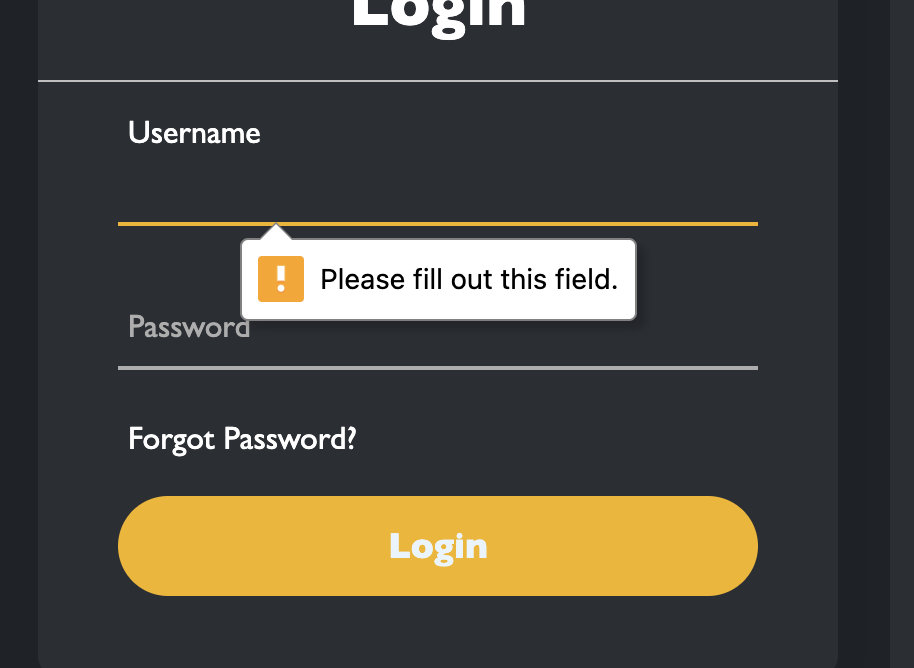


*Figure 2. The website's current block diagram.*

**3.2 All classes and Pages**

The first page to examine is the login page. This page is a PHP file that conducts several tasks before the page is even displayed. Firstly, the PHP code on the page checks if the user is already logged in to the service. If the session is set to show the user is logged in, the page will redirect the user to the dashboard. In most cases, the user will not be, which moves the page onto the rest of the PHP code. Secondly, the page will attempt to connect to the database. If this is not possible, the page will become entirely unresponsive and display an error message. This ensures the database service is running and is connected to the website. Upon the page posting the data from user input, the code checks if the user has entered all the required data in the login boxes, and if all data has been entered a connection to the database is made, and the page makes a request to the database. If the username exists, and the password matches the one in the database, the site flags the user as now logged in. When this occurs, the user is immediately redirected to the dashboard.

This requires the database to be both reachable, and have the same information. The process includes a password hash and salt function so that if the password is the same as another user, it is allowed since the salt will be different. The page has a series of error checking functions that occur both in the HTML and in the PHP code. As mentioned above, the HTML required function will ensure that the boxes in the form have data in them, and if there was an issue with this data and it was allowed through, there is a separate check in the PHP code. The page will only progress to the login process if it has confirmed on both routes that there are no errors involved with the data a user has entered into the form. This is pictured below on the login page.



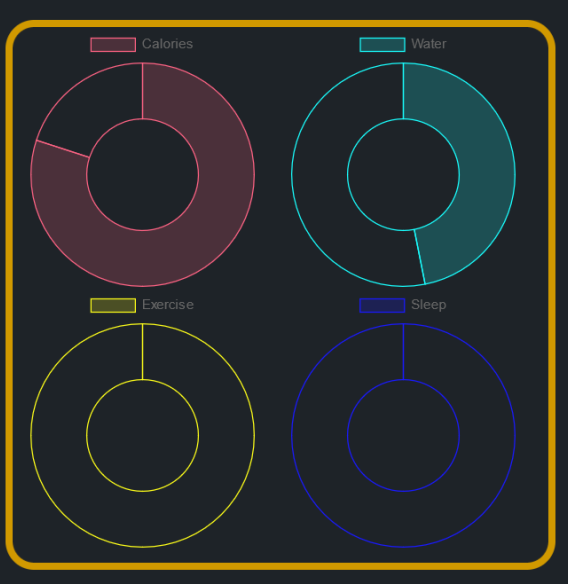
*Figure 3. The Login.php Login form showing data entry verification.*

After the user successfully logs into the service and they are on the dashboard, there are several options and sets of information available to the user. The first option is the user can see a widget that represents the graphs page. This will display the graphs at a quick glance and provide a redirect for the user to the main graphs page. This is stylistically a link, since the main functionality will be on the graphs page. This widget is a reconstruction of the graph page, but on a model that only shows one category as opposed to all of the categories on one graph. The operation of this widget is a large script that is described in the script section of the design analysis. Generally speaking, the script creates the graph widget, and fills the data to the graph if the user data exists, for all categories, and defaults the view to the week view and calorie graph. This can be toggled through a drop down menu to change the data type, but not the view size. Any changes to view size can only be done on the main graphs page. While hovering over the data points on the graph with the mouse cursor, a brief description of the data is presented to the user. This includes the amount (in the unique units for the category), and the date the data point is linked to. This works for all data points on both the graph widget on the dashboard, and the main graph on the graphs page.

The next widget is the main widget on the page, the data rings. The rings are a circular data visualizer that shows the user's completion towards a specific goal in a clockwise rotation around the ring that fills up as the user contributes to the goal. The same script that runs for the previous widget, the graph widget, also constructs and populates the data rings. The data is filled through a call to the database when the user reaches the dashboard that pulls all the current day’s input data. The initial values for all will be zero if no data is available, and the rings will not display any fill percentage.

Once a user enters data on the graphs page, and the dashboard is reloaded, the data will appear on the data rings as the correlated percentage to the total of the goal. If the user has a goal of eating only 600 calories in a day, and consumes 200, the ring will fill up to 33% of the way as that is 200/600.

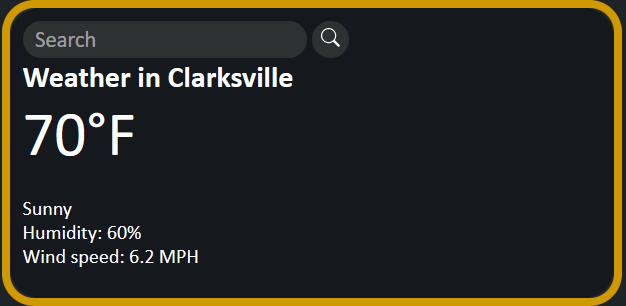
The picture below intentionally excludes the numbering on the main website to focus on the ring completion and variable ring options. One a ring is completely filled to the specified number, pictured below in the known bugs section, the ring turns from transparent, as in the picture below, to an opaque version. Any increase over the specified limit will work properly in the data inside the ring, and will display to the graph widget, but will not affect the ring in any capacity once it is full. That being said, once data is entered to the data rings, the information is immediately sent over to the graph widget and plotted. This also creates a call to the database to save the information to the user’s account. The default date assigned when a user creates data from the ring widget is the current date.



*Figure 4. Data rings on the dashboard widget with numbers removed.*

Finally, the user will see the weather widget on the dashboard. This widget is a direct API to a weather service and pulls the data directly from their source. The source is the OpenWeatherApp API. This widget displays all the data for the current city including the:

* Temperature
* Cloud coverage
* Humidity
* Wind Speed

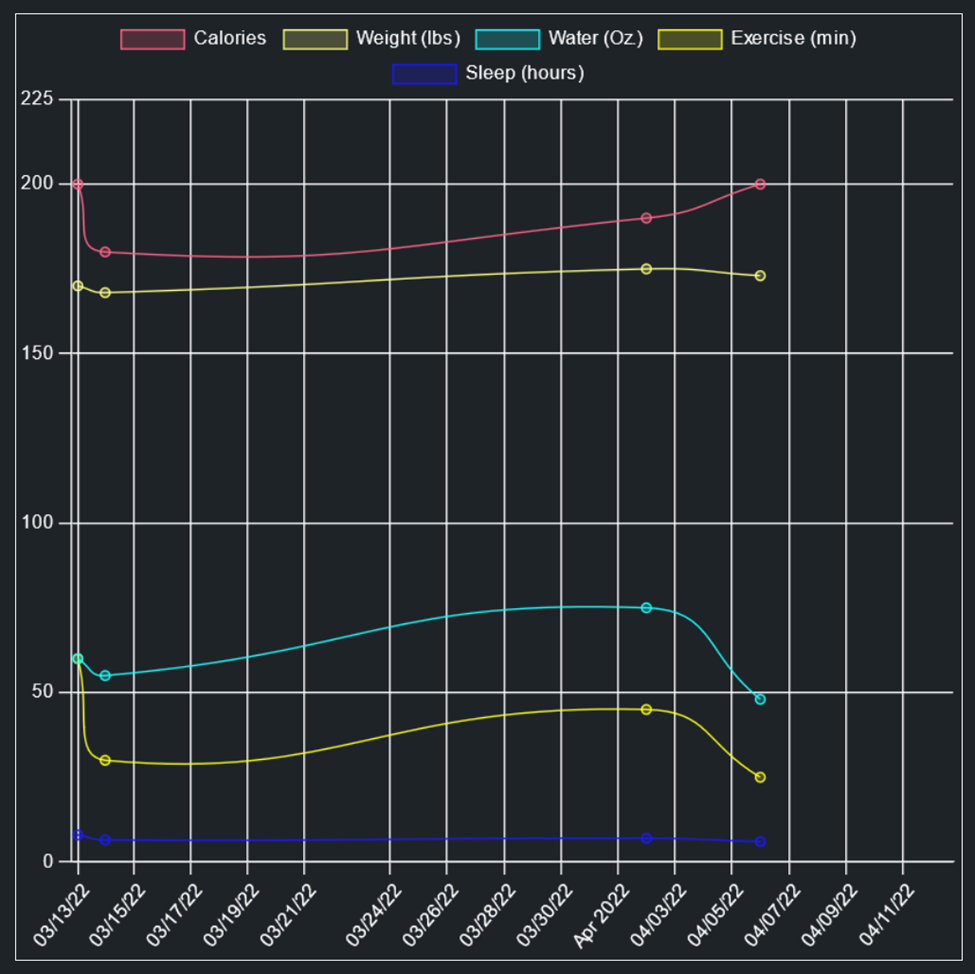


*Figure 5. The weather widget with data displaying Clarksville,Tennessee.*

The weather widget is powered by a script of its own, called weather.js in the public\_html folder. This script will create a blank copy of the weather widget when a user enters the dashboard. Blank in this case is void of data, but the main categories of weather data will be present on the widget. When a user utilizes the search function for a city and state, the data will populate accordingly. This is further described in the scripts section preceding the overall design section.

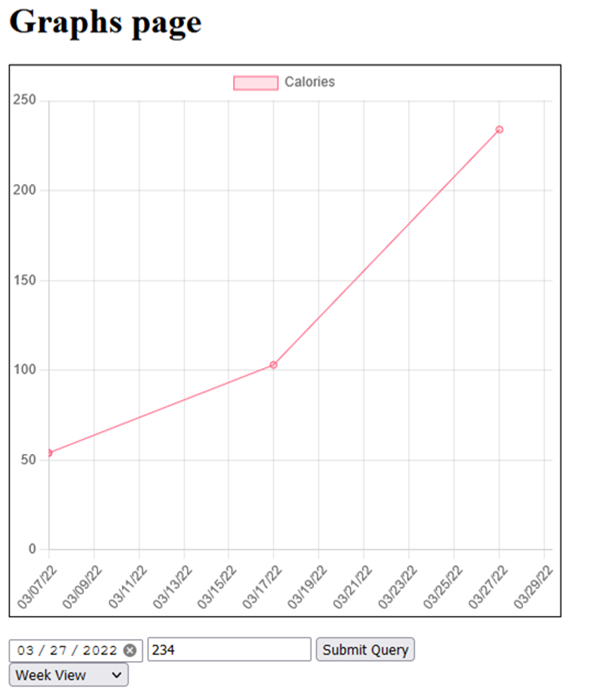
Outside of the dashboard, the first page the user can access is the Graphs page. The user can access this through either the hamburger menu, or the graphs widget. Once the user enters the page, they will immediately see a large graph with all the users data plotted over time. This graph will include all the categories the user can enter data for. The page does this by fetching all data for the user from the database and expanding the graph to the data range on the X axis, and numerical range on the Y axis to accommodate all the categories.

The user can toggle different characteristics of the graph from the graph directly. This happens when the user clicks a button along the top where there is typically a graph title, but on this graph there is instead a collection of color-coded buttons that are assigned to categories of data. This will allow the user to view only the lines they are interested in, with the default setting having all the lines turned on. This is pictured in the graph below, where the separate lines are coordinated with the category at the top of the graph.



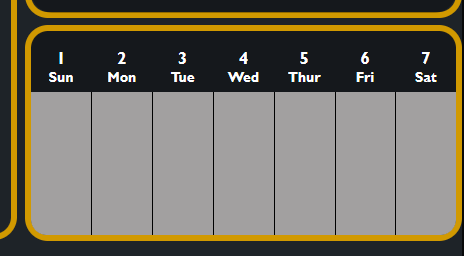
*Figure 6. The chart from the main chart page showing multiple data types displayed together in the month view.*

Data is represented in the graph with a series of dots representing data points where a user entered information. The data point is over the date the user entered the information on, and aligned with the numerical amount on the Y axis of said event. The graphs additionally draw lines in between the data points to show the user’s trends. Originally the graph drew a directed line between the data, but in order to better show a trend, the graph now draws soft lines to bend near a data point and show the user a more gradual representation and not sharp robotic connections. The original model was only capable of showing one directed line, and will be used as the widget on the main page, pictured below. This can cause slight confusion when trends are being analyzed and lines are being drawn, because the tension of the line being too low can cause the line to appear to be moving the wrong way. For example, in a large view with a sizable pool of data, the graph will be stretched out across the screen. Then, if two points are close enough, say one day after the next, the graph will curve the line to the next dot, and the line will appear to move backwards at the middle of the curve, even if only slightly. In the graph above it is not a direct representation, but on the calories line, the points on May 13th and May 15th are close enough that the curve is starting to take a reverse S shape, which is only exaggerated when data points are closer, and have larger variance in the specified value.



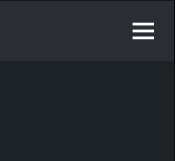
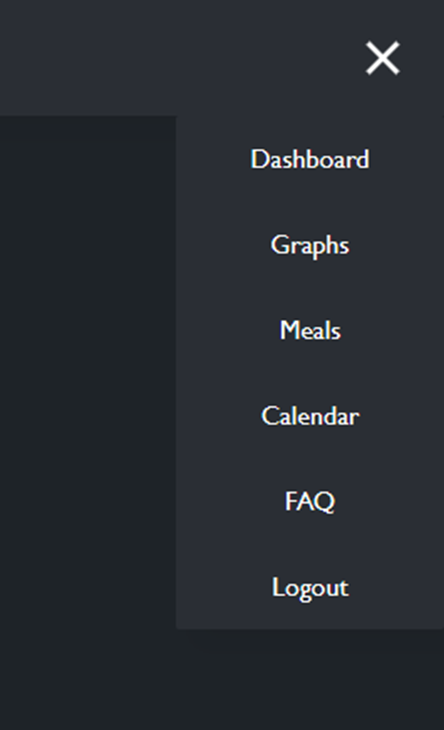
*Figure 7. Graph mockup from halfway through production with rigid data point connections and only one data type.*

The calendar page is currently unavailable, but the calendar widget on the dashboard is a brief insight into the idea and structure of the page. Currently, the widget does not display events or information for the user yet, but the calendar is more a fitness and health related system, than a general use calendar. This being said, the page was intended to pull data from the user account, such as goals the user has set. The implementation would come in the form of a user’s current goal if it is within the current week, and would display a percentage of progress on the tile. The page would make a call to the database for the currently logged in user, and return the user’s goals stored on the user data table. This would return the date value associated with the goal, and calculate current contribution to the goal as a percentage of the total desired goal. This would operate similarly to the data rings on the dashboard, where a percentage is a simple calculation printed on the calendar tiles.



*Figure 8. Calendar widget in its current state.*

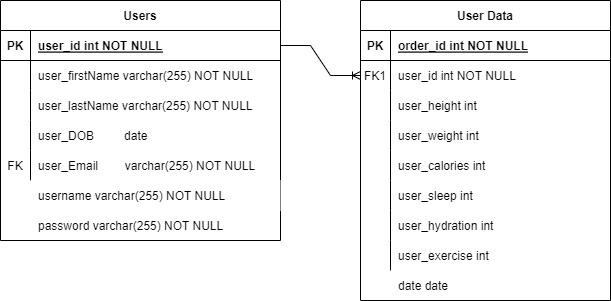
The main method of navigation on the website is the hamburger menu. The menu is a drop down menu with a list of available pages the user can access. This menu is accessible from all the webpages a user has access to while logged in. This menu utilizes a transform when pressed to change the hamburger symbol to an X symbolizing the closing action of the menu when pressed. When the menu is displayed, it shows the available pages to navigate to in a vertical list containing the: Dashboard, Graphs, Calendar, and Logout options. The picture below shows the original menu, hence there are more options that are planned for the future work section. Since we cannot demonstrate the action of the menu transforming, below there is an additional picture of the menu when it is closed to show the hamburger menu icon as it appears on all pages.



*Figure 9. Both the hamburger views combined, one open(left) and one closed(right).*

**3.3 Database Tables**

The site’s database runs off a simple two table system. The two tables stand for a user account, and a user data entry table. The user account ties basic information to a user, including their login information, which is used to tie all data to a specific account. The user will only have one user table that tracks all their basic information, and then multiple of the second table tied to all users. The second table has a collection of different categories for data for user input and tracking. These categories are height and weight, in case they change, calorie intake, sleep, hydration, exercise and a date for the entry. All of the categories are tracked as integers with the exception of the date, which is tracked as a date in the table. This information is further specified by unit of measurement on the main website, but the database can interpret these strictly as integers.



*Figure 10. The systems Entity Relationship Diagram (ERD).*

Per the diagram, the tables are connected by the user ID, which is the primary key in the original table. This makes the system easier to track data all tied to the same user ID. Currently, the user is tied to an email address, but the system doesn’t implement this address yet, which will be discussed in the future work section. The user will be required to enter an email address for registration however, but it will be untouched from there and onwards in version 1 of the system. The database tables are also a one to many relationship, due to the nature of the user and their entries to the website. The users table will only have one of each user, but every time a user makes an entry, the entry will be whatever the relevant sections of the user data table are. This means a user is bound to have an immense number of these tables in version one, since data is all entered at different times and locations depending on the page the user submits the data on.

**3.4 Scripts on the Server**

There are three active scripts on the website, the first being hamburger menu operation. This script is simply a listener that waits for the menu to be clicked. This script is tied directly to the menu on all the pages and is active when the user is interacting with the menu itself. This script file works on all the pages and helps display the menu as seen in the picture above in the previous section. The second script is for the weather API module on the dashboard. This script firstly establishes the data that is pulled from the service, which is comprised of:

- Name for the city entered in the search bar

- Icon and description for current weather condition

- Temperature and humidity for the area

- Wind speed

The API also includes the search bar where a user enters the city and state of the desired location, in “city, state” format. Once a user hits submit on the widget, all the above list information is pulled and displayed to the user. The script only activates once the user submits the search, which leaves all the boxes for weather data and information empty until the user submits an initial search.

The third script being used on the website is for the data rings widget and the graph widget, which are the main components of the dashboard. To start, the script creates all the donut rings. The script initializes all the rings to a default zero percentage of the total goal. The script currently declares these upper goal boundaries as constants and when initialized to zero, the base value will start the ring empty. Using the fill function array, the ring will increase the value of the current level in the donut, increasing the fill around the circle of the ring in a clockwise rotation.

The script then builds the graph widget on the page. It starts by drawing the graph used for the widget, sets the default view to the week view, and sets the default category to calories. The script then builds all of the graphs so the user can toggle between views when they interact with the drop down menu location in the widget. The final part of the script fills the donuts with the information it pulls from the database and then updates the graph with similar database pulled data for the user.

**3.5 Necessary Extra Data Files**

There is only one file that is a pure data file in the system that is necessary to the system's operation. This is the systemData.php file that is placed in the project's root folder. This file is excluded from any version control, which in our case is the github repository. The purpose of this file is to house the login and connection information for the database, which all files will require in order to do any sort of database operation. Without this file, the pages will not link to the database, and will not be able to display any information since all the pages on the website are dependent on this file.

1. **Deployment**

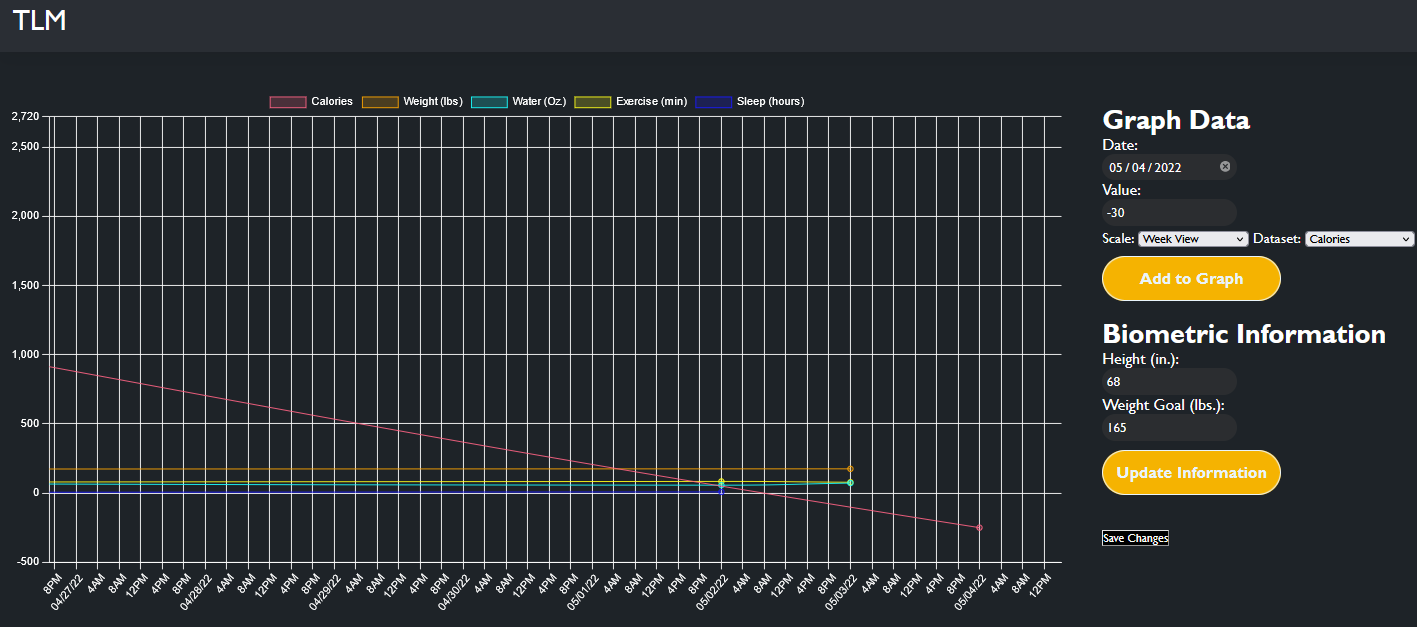
The website requires a few key components to be deployed. Firstly, the service is a web application, so it requires a server that can host apache web servers, and a MySQL database. The database will be described last, so the first core step is the configuration of the server. The server requires an installation of the LAMP stack, which can be done as a package on some distributions, or individually on others. Individually, the requirements are Apache, and MySQL. These must be present and running on a server before the project can be loaded. Once those services are live, the website files need to be uploaded in their current configuration to the website. This means that they need a project folder, with separate database and public\_html folders. The last insertion aside from this is the system data file that will host server connection information that is encapsulated from users and is not included on the version control software.

The database was originally designed in phpMyAdmin. As a MySQL database, the structure is a two-table design as listed above in the design documentation. To create the database either the ERD must be followed from scratch, or a template SQL file will need to be created based off of the TLM SQL structure, just without the website’s data in the case of a new and unique copy. If the user or service requires an exact copy, then the current SQL file on the server can be copied and transposed to any new system for reference or use.

1. **Known Bugs**

**5.1 Bug overview**

The known bugs in the system are both obvious and discreet, which could cause confusion for the users and developers alike. The graphs and by effect, Chart.js, have some quirks that make the website in turn have several issues. The first issue is based on the team's work however, and the bug is that users can technically enter negative values into the graphs. Naturally, no user would have any reason to do this, but this has not been addressed on our end, and causes the graphs to draw incorrectly. This can break both the main graphs page, as well as the data rings on the dashboard, and the graph widget on the dashboard.



*Figure 11. The Latest iteration of the graphs page. This shows a negative value when entered on the graph.*

Pictured above is the graph when a negative value is entered. As you can see from the large red line representing calories, in the final production if a negative value is entered, then the graph is scaled to now show negative numbers. This can cause issues in the other categories because there would be large overlapping lines and it may become increasingly difficult to see the data when the graph is forced to show a wide range of values for the negative entry. This would be fixed ideally with a function that checks the value and converts all entered numbers to positive numbers regardless of the user’s input.

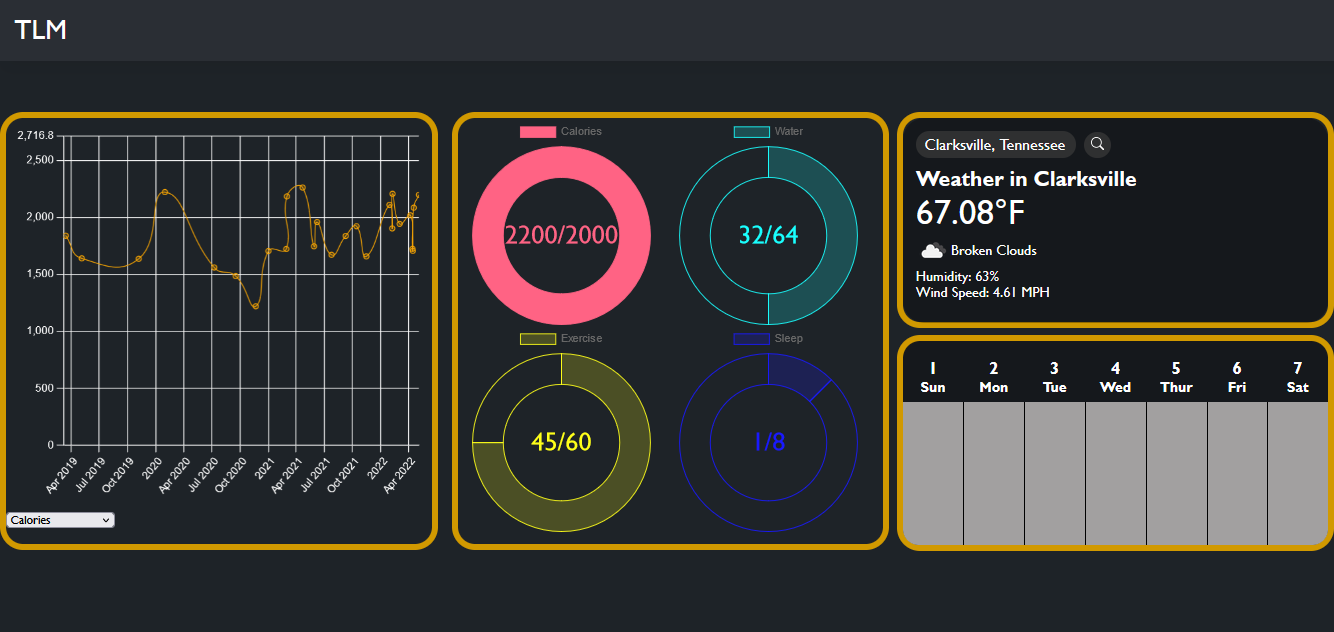
The next bug related to the graphs is the lack of limits on date entry. Technically a user can go back and set the date to dates many years ago, which will draw the graph in a manner that disrupts the current data. If a user adds an entry 10 years in the past it will draw a line from that date all the way to the modern date and skew the visualization aspect of the graph. That is a major bug, as the entire point of the service is the idea of easy to see data.

A known bug related to the login and registration pages are leftover code that is no longer used. The Register.php page still exists, but is in fact no longer used and is depreciated. The Login.php page is used to handle both the login and register systems now. This means if a user navigates to the Register.php page they will be accessing a part of the site that should not be available. The code is not necessarily functional, but could be a potential exposed point on the site. The page has not been removed due to having relevant code that will be used for version two of the software, therefore if it were accessed, there is still viable code. The code should not technically complete any task, but will show a collection of boxes and forms as if it had some functionality.

An additional bug on the current build is that the data rings on the home page are not correctly pulling information from the database. They will take data into the rings through a pop-up prompt on the screen, and will correctly add the data to the graph. This means that information is correctly sent to the database, but is not being redisplayed back on the user’s dashboard. This means the rings will also not persist during a user's stay on the site. When the user clicks to a new screen, the data will be present on the graph, but will not repopulate the rings due to this bug. This serves as a moderate inconvenience to a user, as they make think data is missing, when in reality it currently serves as a temporary way to see goal completion.

**5.2 Bugs related to technology choices**

The website is limited to FireFox currently, as there are many known bugs with the website being accessed from other browsers. To start, the website will incorrectly display all CSS work in the Google Chrome browser. This occurred when testing both Google Chrome and FireFox, wherein the rings on the dashboard would be displayed at wildly inappropriate sizes and in locations they should not have been. This would mean it was being loaded completely incorrectly for just that browser. This bug is what made the decision more clear on what the intended platform would end up becoming. In the final version on firefox, the CSS all matches up nicely on the dashboard.



*Figure 12. The Latest iteration of the dashboard. This shows the correct styling and placement of the widgets in the FireFox browser.*

A bug that is present as another holdover from the original design is the landing page. This can actually be accessed at any point if the user leaves the end of the URL for the specific page out of the link. The user will land on a blank index.php page that says a quick message to indicate which page it is. The fix to this bug would be to add a session check to the page since it is a PHP file and to automatically redirect the user to the Login.php page, such as the other incomplete pages should do.

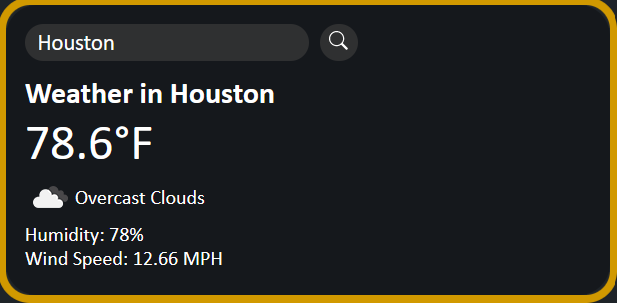
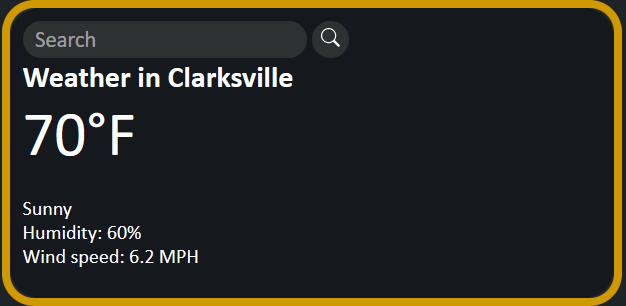
1. **Future Work**

The future work for the program is extensive for several key reasons. The future work originally included a large amount of features and pages, but now has the addition of content that had to be cut from version one of the system. Several items that were cut were time restricted, whereas several items were cut due to lack of experience and would have hindered other systems on the site. This is not to say the pages are a negative portion, and in fact they all directly promote the idea of the page.

The first section will be content that was originally meant to be a part of version one, but was dropped due to the aforementioned restrictions and circumstances. On that front, the Meals page, FAQ and Support page, and Landing page were cut from the webpage set. The Meals page was originally intended to provide the user with a multitude of healthy meal options. This page implemented an API from the United States Department of Agriculture (USDA). The information would provide nutritional data that could be used to give users specific details on food options available to them. This page was dropped due to time constraints ultimately as it did not support the original idea of the website meant for the version one. In the future implementation, meals would be provided from various sources and possibly other API’s, while incorporating the USDA’s API information to show specific characteristics on the individual ingredients. The purpose of this system, that is on the surface a roundabout and overcomplicated combination of information, is to help cater to individuals with specific health concerns, or dietary restrictions.

The next page that was removed from the initial release version was the FAQ and Support page. This page was meant to provide basic questions and answers to users on the website and even contact information for any support requests for the team. This was removed as the site is not going into full production currently, therefore it was an easy task that would have been more of a filler page then a true representation of any tangible learning application. The final page cut from the original design was the Landing page. This page was designed to catch all users coming to the website and display the logo for the website and several pictures in a slideshow for users to see. This page would then redirect users to the login and register page, which is now done by the index page. This means the user never sees the page that would be the landing page at all and is directly taken to the login and register page, which negates the purpose of having the Landing page entirely. If the user were to manually navigate to the directly with no specific target in the URL, they would end up on this page, as mentioned above in the known bugs section.

Another Feature in the next version of this product is location data for user accounts. This would start as adding a location to a user’s account, only as low as the city the user is in, not an address or any personal information. This would be directly for use with the weather API. The idea being that the user can search their location with the weather API’s search bar, but the account location could automatically pull the information for the user so they have a default location set with the API. This would stop the box from appearing blank for the user upon their arrival to the dashboard as it currently does.



*Figure 13. Weather widgets with different locations, showing the functionality and possibility of allowing automatic location detection.*

Originally, the system attempted to implement a hashing and salting aspect to user passwords. This is all handled by PHP when you utilize the inlaid functions, and was implemented early in the applications development. The system was having trouble as changes were made and code and tables were rearranged so they were removed for version one. Ideally, those functions can be reinstated in true form in the next version in order to ensure account safety for users on the service. The functions are not inherently complex to implement, they just add extra steps to the login and register system, which hindered the teams ability to test and accurately understand the interconnectedness of components.

Future work for the website in general includes a daunting list of features and compatibility. The main goal of the site eventually would be cross platform and cross-browser support. The plan would begin as getting support for other browsers, such as Google Chrome and Microsoft Edge. After the support for browsers was finished, the push for mobile support would be next. Mobile devices account for a significant portion of web-based traffic and could help promote the service, as well as be the predecessor for a mobile app.