Table of Contents

Analysis 2
Outline of the Problem 2
Stakeholders 4
Research 5
Features of the Proposed Computational Solution 10
Limitations of the Solution 11
Requirements for the Solution 11
Measurable Success Criteria 13
Design 18
Decomposition and the Structure of the Solution 18
Algorithms 21
The Usability Features 29
Key Variables, Data Structures and Classes 34
Test Data 39
Further Data 42
Development 46
The Database 46
Login System 51
Home Page 86
Progress Page 105
Calendar Page 110
Food Bank Page 113
Modifications 117
Additional Screenshots 122
Evaluation 129
Post Development Testing & Success Criteria Review 129
Maintenance Issues and Limitations of The Solution 136

Analysis

Outline of the Problem

The coronavirus has significantly impacted the well-being of society. Prior to the lockdown, many people were going to the gym, playing numerous sport activities and generally were becoming a lot more health conscious however once the lockdown was enforced upon us, all the progress people made soon became stagnant. This of course had a detrimental impact on the mental and physical health on society. The mental health aspect was thoroughly spoken about, which of course is a huge positive however a heavy toll was taken on the physical aspect of people. According to a survey carried out by the BBC, more than 40% of adults gained at least 3kg of undesired weight during the lockdown period. Now with restrictions being eased and lockdown being lifted, exercise and an objective to reach a healthy weight has been a top priority for many - this research was carried out by Savanta ComRes. The research company filed a report of their research with the main headlines being “there is a recognition of the importance of physical activity in response to the pandemic”. The NHS also published an article which stated, “People seeking NHS help to lose weight during the pandemic”. Of course, this establishes the fact that people are becoming more health-conscious, however when maintaining a healthy lifestyle, there are many misconceptions and myths. Such a problem can be solved if people are provided with an easy guide for their desired goal, backed with credible research. This is why I am proposing a web-based application where these services are provided to the user. My solution will consist of a front-end website that the user will see and it will also have a back-end containing a database.

This solution has features that are solvable via computational methodology such as the program will allow inputs for the user’s height, weight, age and their desired weight. This is an example of abstraction as I am abstracting these necessary details from reality and ignoring the details that are not required for the purpose of this solution. For example, the user’s ethnicity or appearance is not required to be inputted because it is not relevant to the main purpose of weight gain, weight loss or weight maintenance hence why I abstracted this information. The fact that the user will not need to list all their activities on a daily basis and would rather be asked to list their level of exercise from the options; light, moderate, intense is yet another example of using abstraction to remove the necessary details to the solution from the reality we live in. The user will be asked to select their desired goal out of weight gain, weight loss and weight maintenance, this idea of limited goal options is what the solution requires as opposed to the user’s entire aspirations throughout their experience of using the program. The idea of using abstraction in my solution has proved to be essential and my justification as to why this is the case is because abstraction simplifies the process to create the solution as I am only focusing on the details important to my solution.

Thinking ahead is another computational method I can apply to my solution. This involves thought about the different components of the problem and what the best way to handle them would be. At the bare bones of any program, there will always be inputs and outputs. This is no exception with regards to my proposed solution because my solution is supposed to interact with the user, this is why I am able to take this approach in my development of the project. Listing these aspects of the solution at this stage gives a brief overview of what will be required

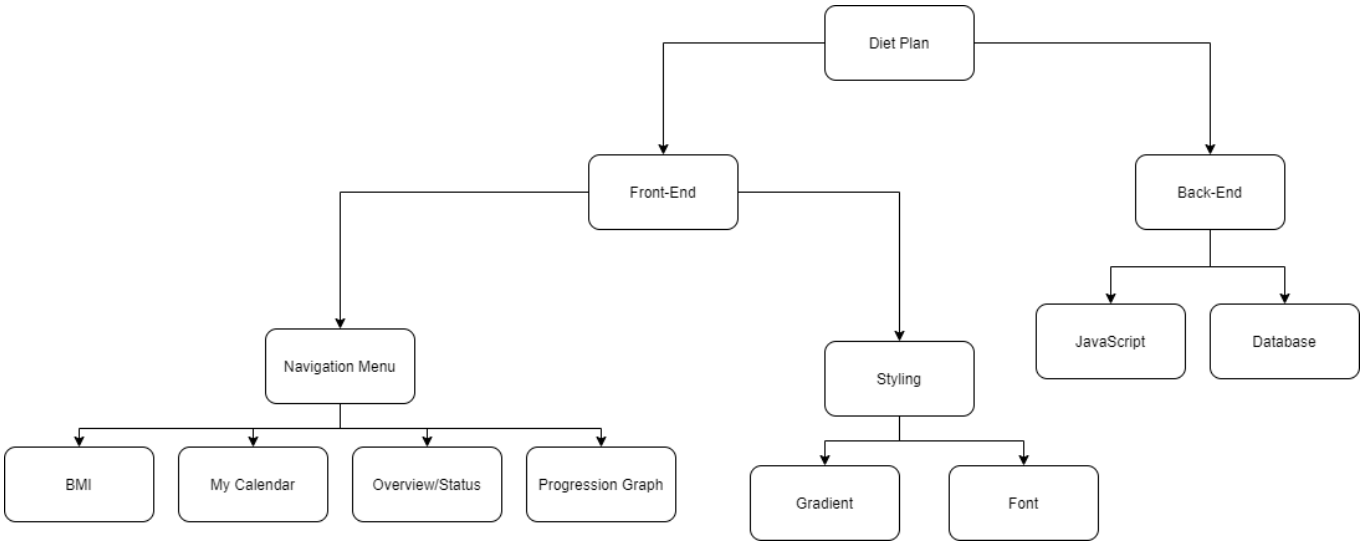
Inputs	Processes	Outputs

• Particular goal – Losing weight, building muscle mass/weight gain or generally just having a balanced diet • Age, weight, height • Desired weight • Eating preferences – if any • How many meals would the user prefer to eat daily? • Ask the user if they would like to search for a specific meal/cuisine • Ask the user if they would be fine with eating the same meal in consecutive days • Allow the user to select specific meals/cuisines if they would like to • The user can input their own meals – They can create a custom meal • The user can input their current weight to provide a log of their progress regarding weight loss/gain - for the graph	• If the goal is…(user option)…create a plan that reduces fat or maximizes protein • If the actual weight is…(user option)…create a plan with…calories a day • If eating preference is…(user option)…create a plan that includes/excludes certain meals • Search the database for the user's desired meal/cuisine - Search and filter system • Create a line graph for the user's progression over time	• The navigation menu • Show the user's recommended healthy weight – BMI • Show the user's goal • Show the calendar • Show the meal's for today • Show the results of the search system • Display the weekly plan • Display the graph • Display facts and figures from credible sources

Another computational method I can use is thinking logically. This allows room for thought when it comes to decision making and how they can affect the flow of the program. As my solution is heavily dependent on what the user inputs and then responsible for displaying the right output, it is very likely that there will be many decisions and conditions in my program. For example, if the user inputs their goal as "weight loss" the program should respond to their input by coming to the decision of displaying the calorie plan for "weight loss", not any other plan for another goal. This is why it is significant that decisions are used in the solution.

Thinking concurrently is the idea of completing more than one task at the same time, this takes into account what parts of the program can be executed efficiently. However my solution will not need to apply concurrent processing because there will not be a scenario where the program has a large task to do and divide it into multiple parts so they can be processed at the same time. Such a method is more suited to a game where graphics and sound have to be rendered at the same time.

The fact that my program has many different parts to it, thinking procedurally is another computational method that can be applied to my solution. Using this method provides a brief, clear



structure of the program by breaking it down into various sub-problems which also makes the entire solution much easier to manage. Given that my solution is a web application that involves front-end and back-end development, it is very straight forward to break down such a design because even before decomposing the solution, I know that there are two sections I need to work on separately. Furthermore, the various sub-problems that can be decomposed from my program is another reason as to why thinking procedurally is amenable to my solution.

The navigation menu will be a bar across the top of the screen and will allow the user to easily access different parts of the web application. When the user clicks on any of the options on the navigation bar; BMI, Calendar, Overview, Progression Graph, they will be directed to the web page corresponding to what they selected. Styling is another sub-problem as part of the collective solution. This is very important because the user will have to be able to read the web page with ease and the idea of using gradients throughout parts of the web pages will be appealing to the user as I discovered through existing examples. The back-end of the project will store the database. This is essential because the user will need an area to store their data through using the application. JavaScript is going to be used in order to provide communication between the front-end and back-end of the program. Thinking procedurally has allowed for an easy overview of the entire project. Using this computational method is also suited to the original problem because of the vast sections of the problem such as exercise, diet, preferences. All these sections can be worked on separately which is why it suits this methodology.

For these reasons, I can conclusively state why the problem can be solved by computational methods and also how my solution will use a computational approach. Using this methodology creates a clear path on how to complete my proposed solution, providing a clear structure makes the entire process a lot easier.

Stakeholders

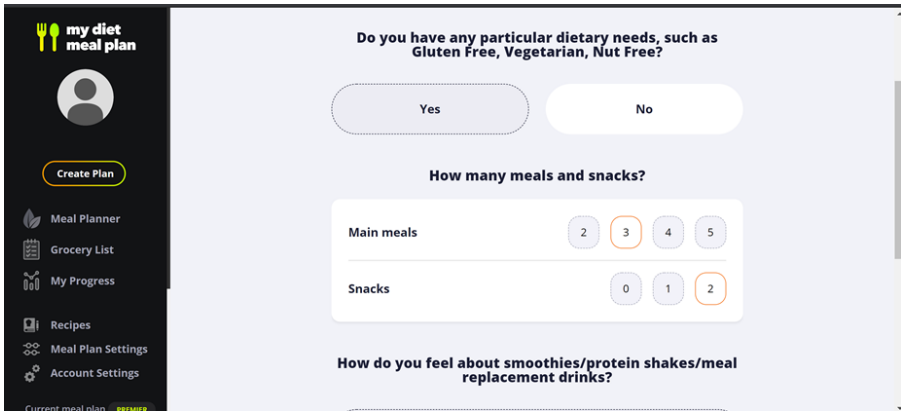
I have chosen two main stakeholders both of which are family relatives. One is an individual that is currently studying in the professional healthcare industry. The reason as to why I have chosen him as

a stakeholder is because he acquires knowledge from genuine healthcare organisations in the UK and will know exactly which sources of information are reliable and also various misconceptions regarding the main problem. My justification for the other individual is because he is someone that has investigated thorough research regarding the main problem I am trying to tackle. Unlike the other main stakeholder, He is not studying in the healthcare industry however this is a benefit because he represents another group of people. He transformed his body through a remarkable weight loss regime. This represents the audience my solution is trying to appeal to – People who want to change their lives and just quite simply, need help getting started. As he is still currently making progress on his goal to weight loss, he proves to be a perfect candidate to interview because he can explain what exactly would be helpful to him that should be included in my solution. Both main stakeholders have said that they are unhappy with their current diet and weight which makes them ideal clients to create the solution for because they both represent the wider audience my solution is being designed for. Their health-consciousness is a huge factor as to why the solution is appropriate to what they need. Furthermore, the two main stakeholders have many contacts creating a focus group for the three categories part of my solution, people who want to: Lose weight, gain weight and maintain weight/keep a balanced diet. Due to the stakeholders having a sound amount of knowledge, various focus groups and an own personal view of how using a program will be useful to aid their personal health goals, they are essential for the completion of my solution.

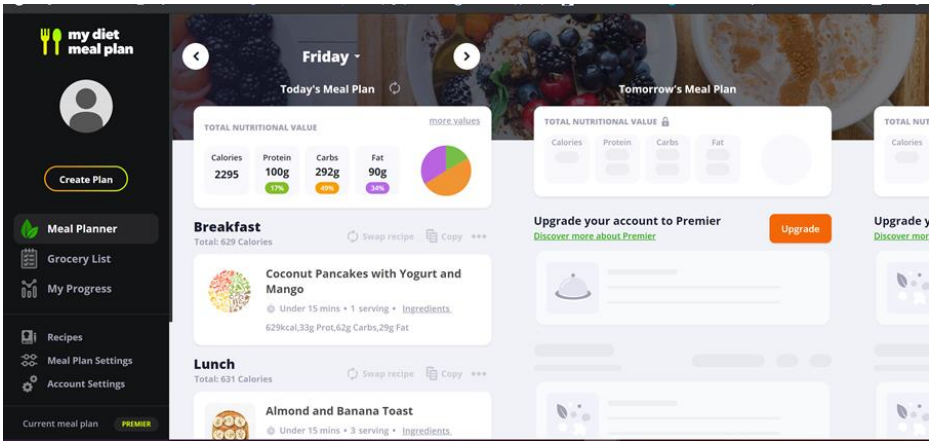
Research

As the problem of healthy dieting has been around for a very long time it is no surprise that there are many existing examples to analyze in preparation creating my solution. The three examples I have decided to analyze are: “Eat this much”, “My diet meal plan” and a mobile app known as “Life-sum”.

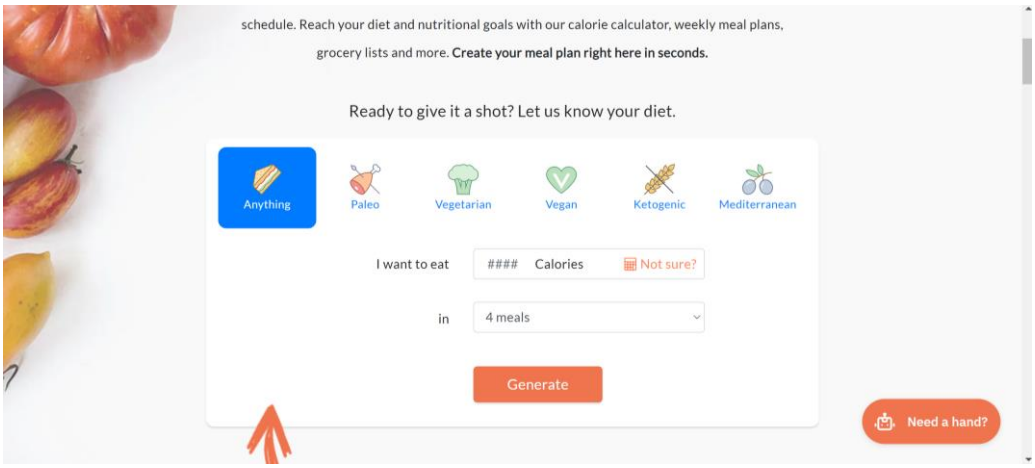
“My diet meal plan” has a website layout different to most examples, featuring a large navigation menu on the left-hand side in black and the actual web page displaying content on the right side. This distinct layout gives a more application-feel experience to the user as if it were an ordinary app on a portable device like a tablet. The website starts off by building a meal plan for the user and provides multiple choice questions which will determine the specifics of the user’s meals. It asks if the user has any particular dietary needs, how many meals and snacks the user would like to eat and how the user feels by incorporating protein shakes – this particular question was very unique compared to other examples and it stands to be an important question as protein shakes have become a trend in society of recent time and not everyone would like them.



Once this short questionnaire has been completed, the website displays the user's customized plan based on their answers and displays the nutritional structure of the plan as a pie-chart. Not only is this ascetically pleasing but it is very easier to read thanks to the percentage make up of each category in the pie-chart being listed to the left of it. The fact that the different sections are colour-coded adds to the ease of reading the chart and keeping track of what the user has to eat for the day. Another unique feature about this website is that it provides a preview of the future meal plans it has created for the user to the right of today's plan. This can allow the user to prepare both mentally and physically for the future meals upcoming. On this page it is made very clear on what the meals will be for the day with the use of bold headings and small boxes for each meal separately. Once the user clicks on a particular meal, an ingredients list comes up which also provides brief instructions on how to actually make the dish. This can save the user plenty of time from having to search up ingredients and instructions for their desired food on a separate tab. Featuring a set of instructions differs from other existing examples as they generally tend to list ingredients only.

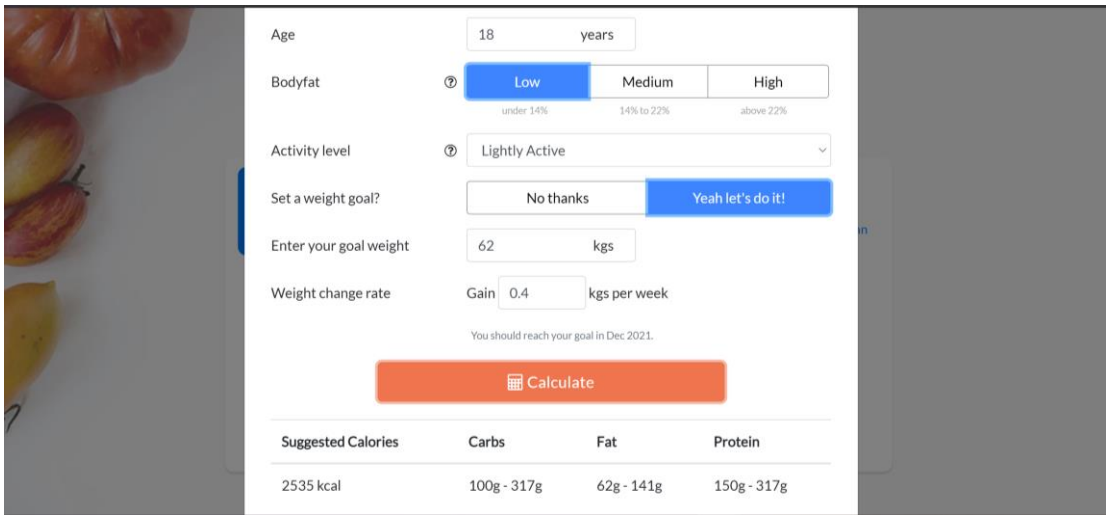


"Eat this much" is another existing example I have researched. When the user initially enters the website, they are presented with a dietary preference list and a calorie calculator option. This affects the meal plan the user will see afterwards as these preferences are taken into account when the website generates food plans.



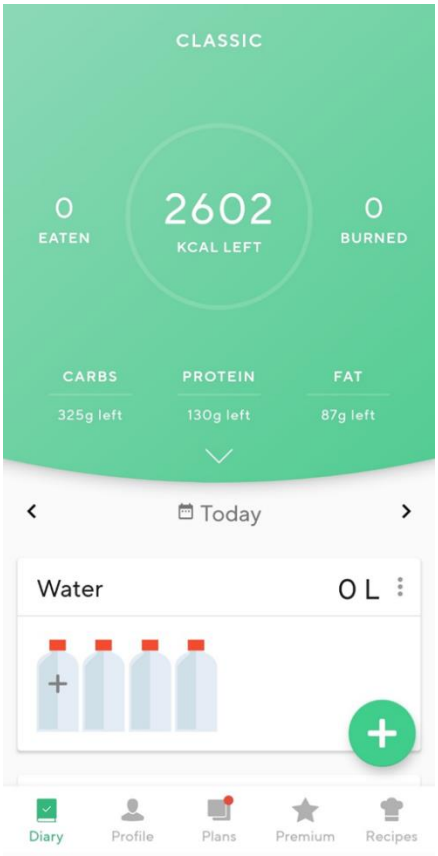
When the calculator icon is clicked, it brings a pop-up window asking the user what their current body status is. Similar to that of the previous example, the user can input their age, weight height etc. however this particular website allows the user to enter their bodyweight percentage which is a huge factor when it comes to creating the right meal. If the user decides to create an account with

the website, they can select even more options such as their daily price limit when spending on their food.

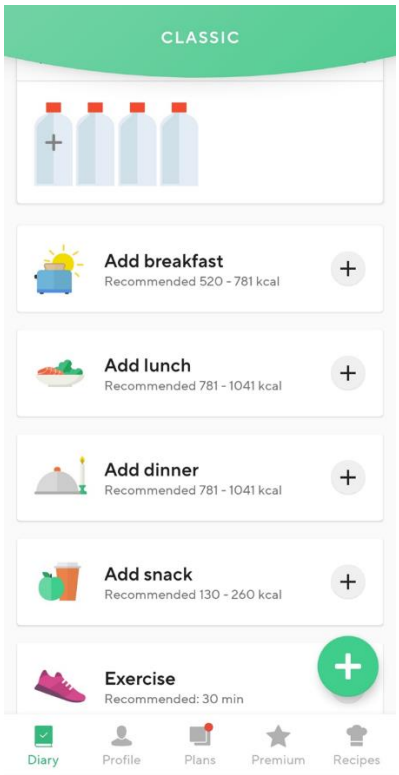


Following this, the website displays the user's meal plan for today based on the options they selected. It uses the same idea of implementing a pie-chart to show the percentage make-up of where the calories are coming from however where this website becomes unique compared to "my diet meal plan" is when the user clicks on the pie-chart to summon an enlarged window of the chart, displaying the specific nutrition make-up. This includes different types of sugars, fats etc. allowing the user to see in more detail what they would need to intake.

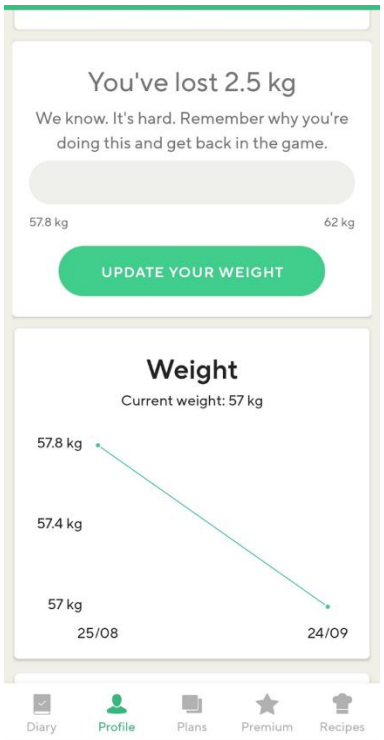
The final example I have researched is the mobile app "life-sum". This program comes very close to my proposed solution from its features to its appearance. Just like the previous examples, the user has to initially fill in a questionnaire from which the app will create a plan. Following this, the app presents the user with a clean minimalistic page featuring a bright green section displaying the calories eaten and burned along with the amount of carbohydrates, protein and fat needed. A unique feature that differs from others is the fact that there is a water-counter on this page. The user can customize how many liters of water each icon takes up and when they click on the icon, the bottle fills up acting as a checklist to the liters of water the user is required to drink daily.

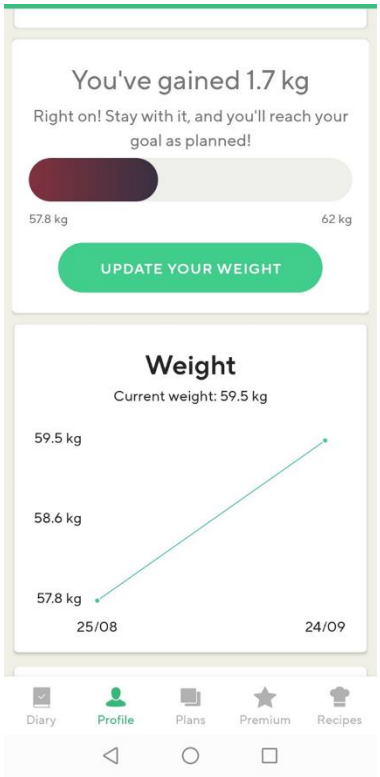


As the user scrolls down, they will find options that allow them to input what they have had during the day. If they do choose to input a meal, they have the option to search for the specific brands of the foods that they ate



Another feature of “life-sum” is that it allows the user to input their weight each day as a log of their progress towards their goal. The weight record is then presented in the form of a line graph which allows the user to easily check their progress in conjunction with the progress bar located above the graph.





Once I examined these examples, I decided to show them to my lead stakeholders to receive feedback and get a more concrete understanding of what exactly they would like to see from a visual perspective. In summary, they both preferred the use of light colours as they proved to be more

readable as well as providing a more comforting “feel” to the website. Although colour may not seem like a significant factor, it definitely has an impact on attracting users. They agreed that the use of vibrant colours should be the approach to take when styling the website. Another approach I took from the research was to include a navigation bar. I think the idea of placing the entire navigation bar vertically to the left of the screen was rather unique as most website generally place it horizontally however the stakeholders both concluded that positioning the bar horizontally at the top of the website looked more professional and symmetrical. Once again, the ascetics of the website is important to attracting consumers which is why I took their opinion on this into account. As for the functionality sector, both stakeholders wanted there to be a window that displays the meals for the day showing the specific nutritional details of the meals based on a questionnaire filled out by the user. This was featured in the existing examples I studied so it would be useful to use the examples as a template because I will be incorporating this feature into my solution. Another approach I was fond of was the idea of making nutritional details very easy to read for the user. The use of pie-charts to show nutrition and line-graphs to show progression transmits information clearly. This is why I will add the use of graphs to my solution.

Features of the Proposed Computational Solution

The proposed solution is a front-end website featuring webpages and a back-end containing the required database. This solution requires a back-end database because of the large amount of data handling related to the problem. All the food/meals the user can select from must be stored somewhere, furthermore with a filter to take into account the user’s eating preferences before generating a meal plan for them. Hence why using a database for this would be suitable. The reason as to why a front-end section is needed for the solution is to provide interaction and output for the user. The familiarity of websites in our society today is the reason why displaying large amounts of information is suited to displaying it on a website.

Feature	Justification
Use vibrant colours & gradients throughout the website	This feature was likened by the two lead stakeholders as they decided the use of colour is very important in a website to attract users
Navigation bar	Essential because it provides an easy method of accessing different web pages for the user
Status page	Displays the user’s meal plan for the day. This is needed in order to provide a simple way for the user check this information
Progression page	This page displays a line graph showing the user’s current progression in relation to their goal and is included in my solution because displaying this type of information is much easier to read when visualizing it
Calendar page	Shows a calendar to allow the user to plan further ahead if they wished
Questionnaire	Every existing example I have researched included this feature. It is essential in order to allow user customization to take place which is what I would like to have present throughout the solution. It will take into account the user’s preferences.

Log of the user's weight	This would allow the user to input their weight to provide a record. This is needed for the line graph to function.
BMI Calculator	This is required in order to indicate to the user what their ideal weight should be which is essential to their goal
Responsive	Making the website responsive is a feature that is needed because not all users will have the exact same device with the same screen size, therefore a responsive website is needed to provide a user with the same experience as another even though they may have a different screen size
Login System	Each user will have a unique set of preferences and goals, hence why there needs to be a login system to separate different user's information
Display credible research	This would be required in order to build trust between the user because they would be more inclined to use the website if there was credible research behind it

Limitations of the Solution

Limitation	Justification – How this is a limitation
Using research from credible outlets	If the solution was to be released publicly, I would need to acquire permission from companies such as the NHS to use their information on my solution. There is no guarantee that companies would be willing to allow this and it could be a complex process.
Range of meal options	Due to limited time, the actual size of the database may not be large enough to satisfy users. They may not find the meals/food that they are looking for due to the database being too small – Me potentially not adding enough data
Creating a secure website - HTTPS	In order to achieve this, I would require a certificate which is something I due not have access to. HTTPS can build trust with new users as they are more likely to trust a website that guarantees their data is secure
Host the website from a server	The finished solution will need a server to be hosted from however I do not currently own a server and do not have access to one so this poses as a limitation

Requirements for the Solution

Hardware	Justification

Mouse	Provides navigation for the user across the different sections/pages. This is needed so the user can access different areas of the website.
Keyboard	To allow the user to input information because the solution requires a large amount of information from the user, hence why this is needed.
Screen (Laptop, phone, monitor etc.)	This piece of hardware is needed in order to display the actual solution for the user, this is required because the user will need to physically see the program so they can interact with it.
Touchscreen	This is for users that are accessing the web-app via a phone. A touchscreen will be the way the user will navigate the web-app and the way the user will input information through a virtual keyboard. The solution needs details about the user and the solution needs to be displayed so the user can see it – both can be done with a touchscreen.
4GB of RAM	Most computers nowadays come with at least 4GB of RAM installed and because the solution is a web-app and not installed software, this will be more than enough main memory. However, to ensure the app performs well, the user should not use anything less than 4GB.
64GB Secondary Storage Unit	Many devices come with at least 64GB which is more than enough storage for the solution to function. Furthermore, this is a web-based application which will not need anything to be installed onto secondary storage, which is why this may not be important.
Router, supplied by an internet service provider	Hardware that allows the user's device to connect to the internet via a modem cable connecting to the router which connects to household devices. This is needed to allow the user to access the web-app. As it is a web-based solution, internet connection is required.

Software	Justification
Web Browser (Chrome, Firefox, Safari etc.)	This is needed in order to access/visit the web-app by using a URL. This would need to be HTML5 capable – which most browsers are
Operating System (Windows, IOS in Macs, Linux etc.)	Required in order to provide a platform to run the web browser which is needed to access the web-app
Phone Operating System (Android and IOS in iPhone)	Required in order to provide a platform to run the web browser which is needed to access the web-app

Measurable Success Criteria

Section	Criteria	Justification
Database	1. Create a database (Food)	This will be used to store all the information related to meals and foods which is needed so the solution can use this data to generate foods for the user.
	2. Create a database (Login details)	This will be required to store the login details of users and validate their identity when needed so that the sensitive data of users will be protected.
	3. Create a database (Exercise record)	This will keep a log of the amount of calories the user has lost through exercise. Exercise is important to many people which is why activities should be recorded
	4. Create a database (Previously Eaten Foods)	This is a record of all the foods eaten by the user. It will be used to provide an average nutritional intake over a given period of time by taking data points from this database, hence why it is needed.
	5. Create a database (Personal Food)	This is needed in case the user is unhappy with the choice of options in the main food database. They can store their own custom meals here.
	6. Create a database (Custom Plan for each user with their preferences)	A relational database that corresponds to the personalisation from each user. Because each user will not be the same, there cannot be fixed plans instead the user can customise their plans.
	7. Create a database (Facts from credible sources)	The data points from this database is needed to display facts on various places on each page of the web-app. Having a database stores all the possible facts that could be generated which is why it is required.
Styling	8. Create a green-blue gradient as a theme for backgrounds on each page	A personal preference from the stakeholders. They suggested the use of gradients because it has become a trend in websites of recent time. The use of the colour green symbolizes progression which is what the solution is trying to achieve for users. The use of the colour blue is because it is recognized as being versatile.
	9. The use of the font “Kumbh Sans” throughout each web page	A font chosen by the stakeholders. They concluded that this particular font looked the most modern and professional which would appeal to most users.
	10. The web-app must be responsive	This ensures that users that are using different screen sizes to access the web-app will not have their experience compromised and so the web-app will automatically adjust

Functionality		to suit their screen size to resolve this matter.
	11. A navigation bar presented horizontally across	This is required to allow easy navigation between different pages for the user. Placing this horizontally looks more symmetrical.
	12. Have the following headings in the navigation bar: Home, Calendar, Progress, Food bank	These are different webpages the user can access. Each providing their respective information corresponding to the heading. Having clear headings or sections allows the user to access these pages with ease and become familiar with which functions are located to their corresponding headings.
	13. On each page, display a fact box containing information randomly selected from the fact database	This portrays a stand-out message from a credible source. The users will need to trust my solution for them to use it and this is possible by using information from credible sources and displaying them in this manner.
Login	14. Ask the user to create an account	For new users, they would need to create an account to assign all information part of the solution to their unique name. This is needed so the user can use the solution to adapt to their details.
	15. Verify if the username has already been taken	To avoid collision and incorrectly displaying sensitive data to the wrong user.
	16. The user must create a secure password that consists of at least one capital letter and one number	As the user is inputting sensitive data, their account will need to be secure to prevent unauthorized access
	17. Option to login as existing user	For users that have previously created an account before. They will not need to create an entirely new account to use the solution again if they login using the right credentials, they will be taken back to their account. This is needed to allow the user to access their account multiple times and over a long time period.
	18. Verify the user's credentials	As a form of security, this authorises the user granting access to their account in order to prevent unauthorized access.
Questionnaire	19. Questionnaire for the user	This will take into their food preferences, current necessary health information. This is needed in order to recommend the right meals for the user.
	20. What is your goal?	A multiple-choice question which is important because it will decide how the solution will be used for the user.
	21. What is your current age, weight, height, body fat ratio, exercise level?	This information is required because the web-app will need to calculate how the user can reach their goal given their current situation

	22. What is your desired weight?	The web-app will recommend a meal plan based on the user's goal and their desired weight so this is question is significant
	23. What are your food preferences?	As the web-app will recommend a meal plan, taking into account the user's food preferences will filter the food database to the user's tastes to generate foods that fit to the user's liking
	24. How many meals would you like to eat during the day?	Every user may have a different preference in relation to this. The number of meals is also important to calorie intake, as calories will be divided amongst the number of meals
Home	25. Display the number of calories the user needs to eat	Essential to the solution as the whole purpose is to amend people's intake so that they can achieve their goal.
	26. Display the meal plan for the day	This is needed in order to show the user what meals are recommended for the day
	27. Allow the user to generate new foods from the databases	If the user is unhappy with a meal, they can generate a new one
	28. Allow the user to tick off each meal to mark as "eaten"	This provides a checklist style process for the user which is easy to use. It is also required for the previously eaten foods database
	29. Eaten meals should update the calorie & nutritional target for the day	This shows the user the details of how much more they need to eat for the rest of the day after eating foods already. This is important otherwise it becomes difficult for the user to know how much more they need to eat.
	30. Eaten meals should be added to the database (previously eaten foods)	This will be used as data points for calculating the user's average nutritional intake.
	31. Allow the user to add extra foods/meals	The user may decide to customise their meal plan for the day but keep the same plan layout for the rest of the time of using the solution.
	32. Allow the user to remove foods/meals	
	33. Update the calorie & nutritional target based on the user's custom meals	If the user enters a custom meal, the target has to be updated to show how much more intake the user will have to have. This is needed so that the user can easily see how much more they need to eat after consuming a custom meal.
	34. Display the nutritional values for each meal with the use of pie-charts	A pie-chart is a universal method of displaying proportions especially in the sector of dieting and health. Its colour-coded sections can easily be interpreted by the user which is why I have included this.
	35. Show the user's current calorie target for the day	The user will need to know how many calories they will have to consume throughout the day based on their goal

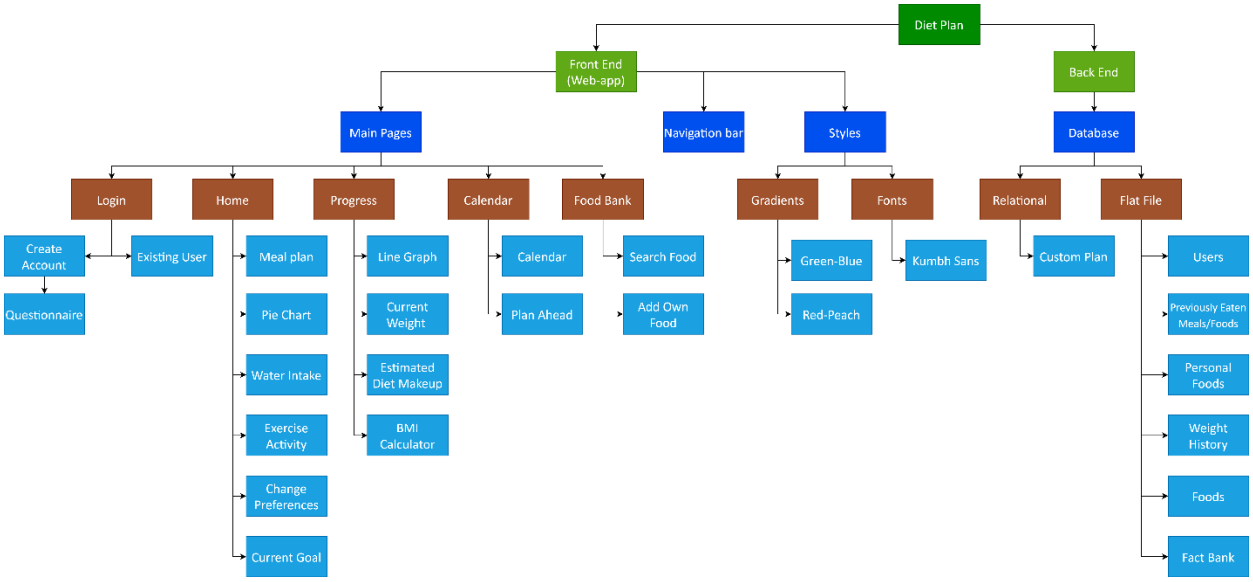
	36. Display the calorie target using a progress bar	This can provide a graphical representation of the target which is simple to follow. A pie chart is not needed as only calories need to be displayed for this, not other components. So a progress bar is sufficient enough
	37. Show the user's water intake	A feature taken from the existing example "life-sum". Water intake is a huge factor in the solution as it is important in dieting and so the solution needs to take account of it.
	38. Allow the user to input an exercise activity	Exercise is also a huge factor in dieting and may be required depending on the user's goal. Exercise burns calories and so this may affect the user's calorie target which is why exercise activities need to be inputted
	39. Provide a recommended exercise target	Exercise is a fundamental component of keeping healthy which is why there should be some guidance of what the user should do
	40. Provide an option for the user to change their food preferences	User's may want to change their preferences at times and so this gives them the option to do so
	41. Provide an overview of the user's current goal, allowing them to change it if desired	The user may reach their goal and then decide if they would like to set a new goal so this feature will allow this.
Progress	42. Allow the user to input their current weight	This is required in order to gain information about the user's weight
	43. Add current weight inputs to the weight database	This is required to provide data points for the weight record database
	44. Produce a line graph from data points in the user's weight log	This is needed so that the user can easily track their progress in terms of weight
	45. Review progress from the line graph on the user's weight, if no progress is made, recommend a change	This is to ensure that the user reaches their goal more efficiently if they are struggling to reach their goal over a long period of time
	46. Produce a line graph showing projected progress	A template or a guide the user can follow to assess whether they are meeting the standards of what the solution is suggesting. This is needed to make sure the user can easily track their progress and identify if amendments need to be made to their current routine
	47. Produce a line graph from data points in the user's exercise record	This is so that the user can keep track of their exercise level which is essential for remaining healthy
	48. Produce a pie chart that shows the user's diet makeup on the average	An easy-to-read pie chart is used often for nutritional values. The colour coding makes

	nutritional value intake overtime	it easy to interpret which is why it is a good way to represent this particular data
	49. Display the average nutritional intake for each month/week	This will provide an insight for the user into their progress from a different view as opposed to weight. It allows them to analyze their progress in a different way.
	50. Provide a BMI calculator that indicates ideal weight	BMI is a recognised measure of identifying a person's healthy weight. This is what will need to be worked out about the user to calculate how to reach that weight from their current weight.
	51. Display the user's estimated time to reach their desired weight	This is so that the user knows how long the process is estimated to last which also allows them to plan ahead.
Calendar	52. Provide a calendar	Users may want to keep track of time
	53. Allow the user to create meal plans for future dates	Once again, this allows the user to plan ahead and gives them time to buy groceries to make the meals
Food Bank	54. Allow the user to search foods from the main food database	In case the user would like to search for a specific meal
	55. Filter the database with the user's preferences	This shows the user exactly what they would like to see when searching for the database which saves them a lot of time
	56. Allow the user to "favourite" foods from the main food database	In the scenario where the user may like foods from the main food database and they may have it frequently, it can be added to their own database to provide a shortcut instead of having to search the main food database
	57. Allow the user to view their own database	This is so that the user can see all of their personal foods that they have added to their custom database
	58. Provide an input for the user to enter foods to their personal food database	To allow the user to add their custom foods to their own database. This is required in case the user wants the solution to generate personalised meals
	59. Display a window asking the user to enter all the required nutritional values for their custom food	This is important otherwise each time the user marks their food as "eaten" the web-app will not know how many calories to deduct from the daily target. Storing all nutritional values for every custom food means they can be deducted from the daily target accurately
	60. Produce a calculator that can work out nutritional values per 100g	Many food items have packaging that displays the nutritional values per 100g. This calculator saves time for the user from having to perform calculations of nutritional values manually

Design

Decomposition and the Structure of the Solution

Thinking procedurally has helped me get a better understanding of the entire solution. Taking this approach to the problem has allowed me to identify key points in the solution and break them down into individual sub-problems. This makes the entire project much more manageable and easier to follow. Hence why I have decided to use this methodology in creating a diagram for my solution to provide a clear structure on how my solution has been decomposed. I can also apply another computational approach known as thinking ahead. This is because I am planning out the subroutines for my solution which will be used later on.



Process	Explanation	Justification
Create Account	This will allow the user to enter a unique username and password	An account system is needed to protect the data of users
Questionnaire	A series of questions that will gather information about the user and select the ideal plan for them	This is required in order to identify the user's details and tailor a diet plan to suit their needs
Existing User	Allows the user to login to their account	To verify existing users
Meal Plan	Display the calories and nutritional values for the meals the user will be eating during the day based on their goal. The user can generate a new meal if they wish	This indicates to the user what they should eat during the day and gives a composition of what each food contains in terms of nutrition
Pie Chart	A coloured pie-chart will be used to display information	A colour-coded pie-chart is easily readable. Existing examples I looked at before used pie-charts which worked well in displaying the required

		information. Hence why I have decided to apply this feature to my solution
Water Intake	The user will be allowed to log their water intake for the day	Water is significant in the problem my solution is trying solve. Dieting and water intake is essential in order for the user to reach their goal. Logging water intake ensures the user is drinking enough
Exercise Activity	This will allow the user to log their exercise activities for the day	Similarly to water, exercise is also essential to my solution. Logging exercise activities measures if the user is getting the recommended exercise based on their goal
Change Preferences	The user can change their food preferences which will affect the meals randomly generated from the database.	This is a filter for the food database, allowing the user to quickly search for foods they desire
Current Goal	The user will be able to see their current goal and can update it if they wish. Depending on their selected goal, their meal plans will differ. There will also be an estimated time for the user to reach their goal.	As the whole solution is based on achieving one of the three goals; maintaining, gaining or losing weight, the user should be able to see and amend their goal
Line Graph	Display a line graph to show progress for the user. Based on the weight logs inputted by the user. If the user has made progress towards their goal, the graph will be green. If the user has not made their expected progress, the graph will be red.	A line graph is used to provide clear and easily-readable data for the user to see their progress. The idea of colour-coding progression and regression is also done so that the user can easily see their current path to their goal
Current Weight	This will allow the user to input their weight on a daily basis to create a weight log	Creating a weight log will be used to display the user's progress in relation to their goal. Past information will need to be compared with current information to make a judgement on current progress
Estimated Diet Makeup	The user can record all the meals they've eaten each day	This will provide a database of all their eaten meals which can be used to show their average nutritional values across a period of time
BMI Calculator	A calculator that can work out the user's ideal weight	Users may want to check what their recommended weight is

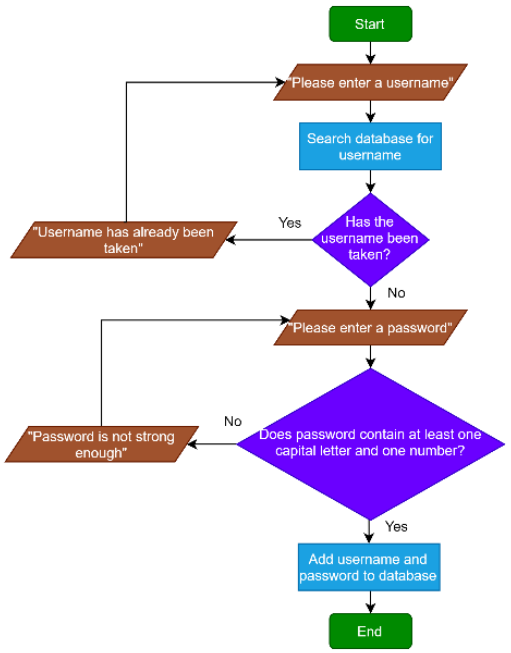
	according to professional research	based on their personal details. This is more efficient compared to users having to search for a BMI calculator in another tab in their browser
Calendar	Displays a calendar for the user	This allows the "plan ahead" process to take place. Providing a calendar can also allow the user to keep track of their time
Plan Ahead	The user will be able to click a date on the calendar and generate a meal in advance	Users may want to plan in advance and so including this process gives them the ability to do so
Search Food	Allows the user to search for food from the food database	Instead of manually scrolling through the entire database, the user can search for foods. Saving a lot of time.
Add Own Food	The user can input their own food/meals to create their own personal database. They would have to enter all nutritional details	Users may not find their desired food in the food database, so they can enter their own foods
Green-Blue	A green-blue gradient that will be used in parts of the web-app	Suggested by the two lead stakeholders, green represents progress which is what my solution is trying to achieve for its users. Blue is a versatile colour as it is used in many websites today.
Red-Peach	A red-peach gradient that will be used in parts of the web-app such as notices, warnings, regression.	A suggestion made by the lead stakeholders. This contrasts the green-blue colours that will be used in the solution. Hence why it will stand out, making it good for notices and an indicator for regression
Kumbh Sans	The general font that will be used throughout the web-app	A request made by the two lead stakeholders
Foods	A database containing all foods/meals listing all their nutritional values. When the user wants to generate a meal plan, foods will be selected from this database	By using a database, users will not have to carry out their own research into possible foods/meals. Using a database provides information in an easily accessible space saving a lot of time for users
Users	A database containing the username and passwords of all users	This is needed in order to verify users

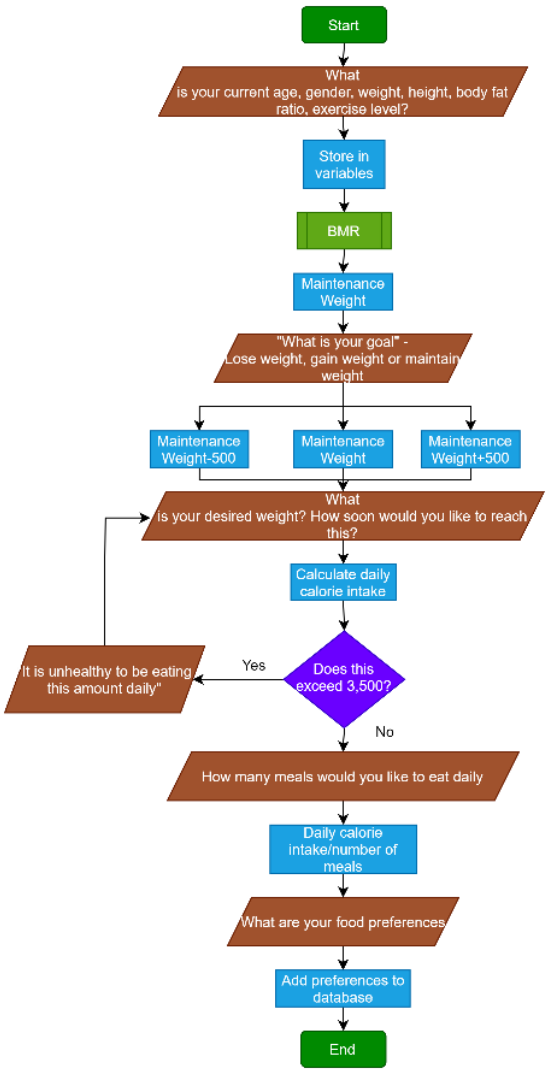
Exercise History	A database containing the number of calories burned corresponding to the date of the activity	Necessary to provide data points for the exercise line graph
Previously Eaten Foods	A database that stores all the foods eaten by the user	In order to provide an detailed insight into their average food intake across a period of time
Personal Foods	A personalized database for each user. This allows them to record their own foods/meals and add them to their own database.	This is needed in case the user is not satisfied with any of the options from the main food database
Weight History	A database that stores the user's weight value each day	This will be used as data points for the line graph. In order to show progress, previous data will have to be recorded hence why a database is needed in this case
Custom Plan	This will allow the solution to record various inputs that could be different to each user. Such as the number of meals they are willing to eat in a day or their average level of exercise	This is required because each user has their own personal preference. Setting fixed inputs removes both flexibility and personalization factors for the user which is less preferable
Fact Box	Random facts will be generated throughout the web-app. Each fact displayed will be taken from this database and will then be displayed in boxes so they are easily noticeable for the user.	This is needed because the user will need additional information besides what the web-app is designed for as the health sector is such a vast area of information. Creating a separate box for these facts ensures they are noticeable for the user.

Algorithms

The following algorithms portray an accurate description of the individual modules that collectively form my solution. The use of flowcharts provides a visual representation of the sub-problems identified when decomposing my solution and it gives a better idea of how the process is going to work. The use of pseudocode further describes these processes as part of my solution by offering a more accurate example of what the actual code will be. Using flowcharts in conjunction with pseudocode provides an easy-to-follow guide on how the individual processes should be completed which is why I have decided to implement the two. However, I have decided to use pseudocode for the larger, more complex algorithms that may not be so easily described by solely using flowcharts. These algorithms are smaller sub-problems part of my larger solution which correspond to the different areas of the program. Once each process has been completed, each can be joined in one

space forming the final solution. This explains why the algorithms that I am representing through flowcharts and pseudocode will form the complete solution.

<table>
<tr><th>Flowchart</th><th>Pseudocode</th></tr>
<tr><td colspan="2">Create Account</td></tr>
<tr><td> 
</td><td>Function CreateAccount ()
User = INPUT “Please enter a username”
Find = Search (user_database) for (user)
If find == TRUE then
  OUTPUT “Username already taken”
Else
  Write (user) to (user_database)
End IF
Password = INPUT “Please enter a password”
If capital in password then
  If integer in password then
  Write (user, password) to (user_database)
  Else
  OUTPUT “password needs to have at least one number and capital letter”
Else
  OUTPUT “password needs to have at least one number and capital letter”
End if</td></tr>
<tr><td colspan="2">Questionnaire</td></tr>
</table>



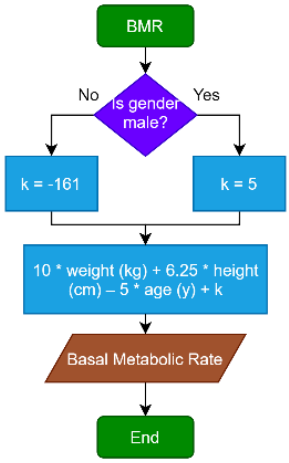
Function Questionnaire (Age, Gender, Weight, Height, Body fat, Exercise level)

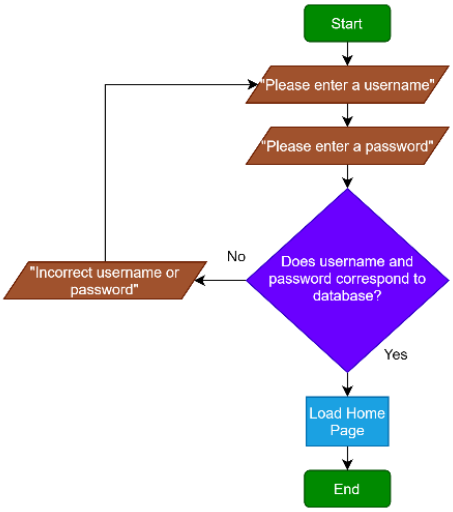
Function BMR (Age, Gender, Weight, Height, Exercise level)

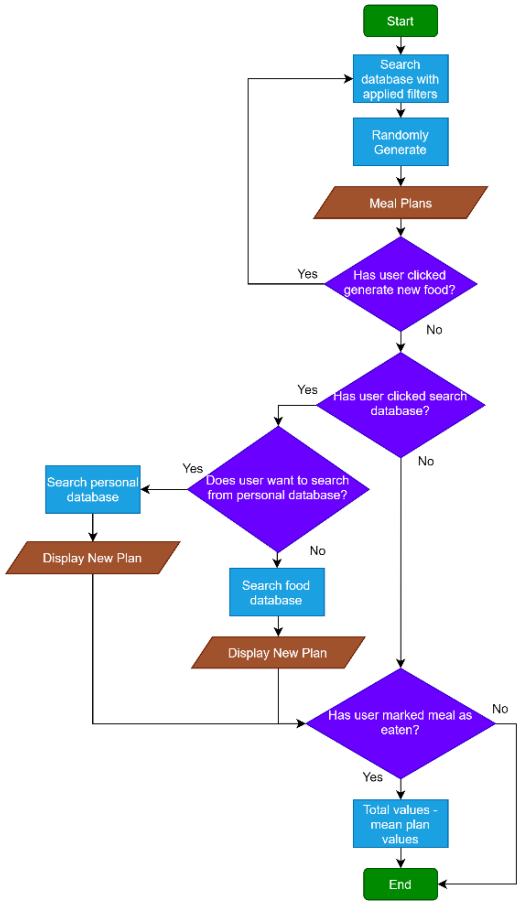
Goal = INPUT “What is your goal”
If goal == Maintain then
  Target = Maintain
  Protein = weight
  Write (target, protein) to (user_database)

Elif goal == Gain then
  Target = Maintain+500
  Desired_weight = Input “what is your desired weight”
  Protein = Desired_weight*2
  Diff = Desired_weight – weight
  OUTPUT “To reach this, it will take, (Diff/0.45), weeks”
  Write (target, protein) to (user_database)

Else then
  Target = Maintain-500
  Desired_weight = Input “what is your desired weight”
  Protein = Desired_weight
  Diff = weight – Desired_weight
  OUTPUT “To reach this, it will take, (Diff/0.45), weeks”
  Write (target, protein) to (user_database)
End if
Number_meals = input (“how many meals would you like to eat daily”)
Preferences = input(“What are your preferences”)
Write (Target/Number_meals) to (user_database)
Write (Preferences) to (user_database)

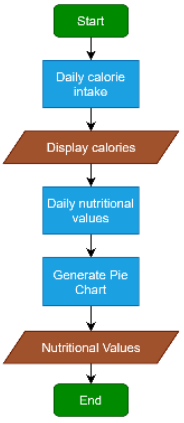
<table>
<tr>
<td> 
</td>
<td>Function BMR (Age, Gender, Weight, Height, Exercise level)

If gender = male then
K = 5
BMR = 10*weight + 6.25*height-5*age + k
Write (BMR) to (user_database)
OUTPUT BMR
Else
K = -161
BMR = 10*weight + 6.25*height-5*age + k
Write (BMR) to (user_database)
OUTPUT BMR
End if
End function</td>
</tr>
<tr>
<td colspan="2">Existing User</td>
</tr>
<tr>
<td> 
</td>
<td></td>
</tr>
<tr>
<td colspan="2">Meal Plan</td>
</tr>
</table>

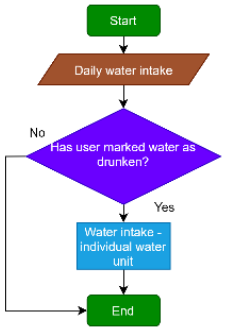



```
Search (food_database) for (meal, preferences)
Output meal
If user generates then
  Search (food_database) for (meal, preferences)
  Output meal
Else
   If user selects search database then
     If user selects personal_database then
       Input Search (personal_database) for meal
       Output meal
     Else
       Input Search (food_database) for meal
       Output meal
  Else
    If meal == eaten then
      Total values – meal plan values
End if
```

Pie Chart

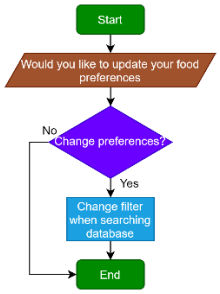


Water Intake



Exercise Activity

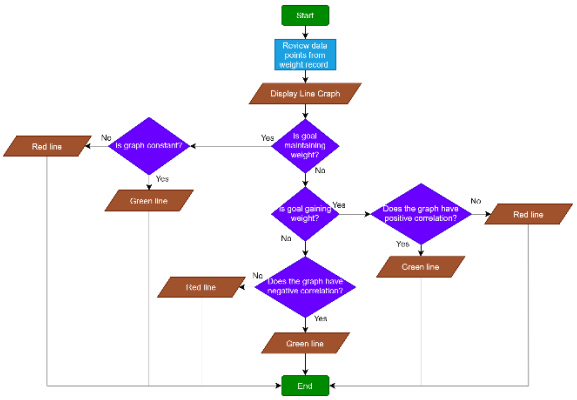
Change Preferences



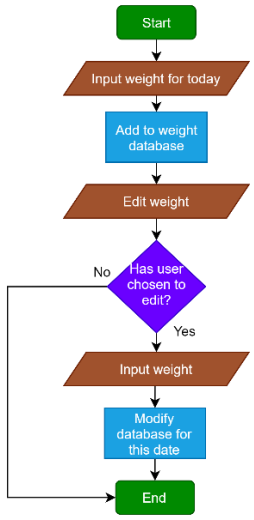
Current Goal

Output (goal)
If Input “Would you like to change your goal” == yes
  BMR (age, gender, weight, height, exercise level)
  If select == lose weight then
    Target = maintain-500
    Output target
  Elif select == gain weight then
    Target = maintain+500
    Output target
  Else
    Target = maintain
    Output target
End if

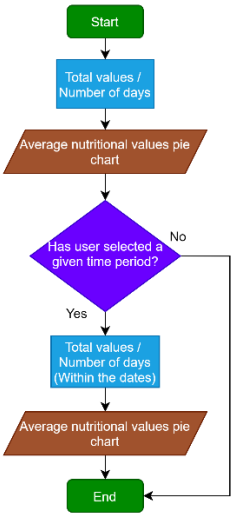
Line Graph



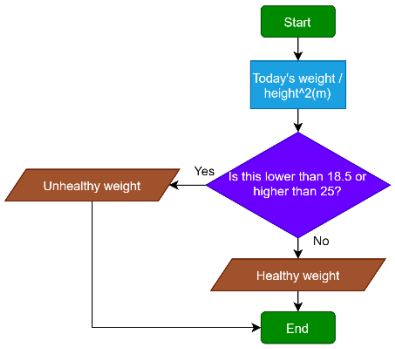
Current weight



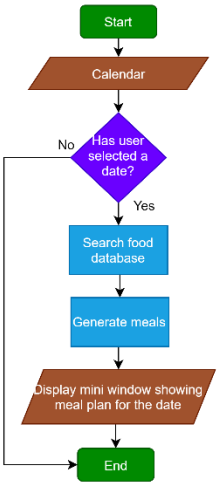
Estimated Diet Makeup



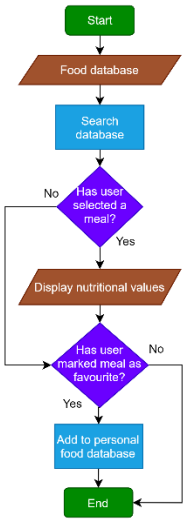
BMI Calculator



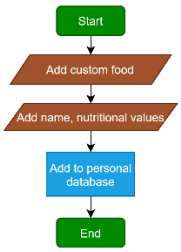
Calendar & Plan Ahead



Search Food

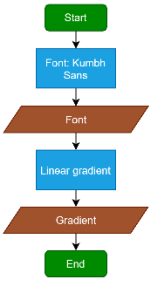


Add Own Food

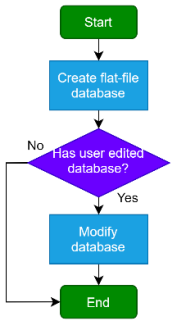
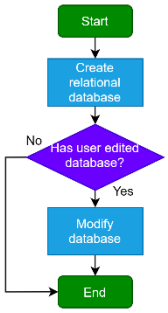


Value per 100g Calculator

Green-Blue, Red Peach, Kumbh Sans

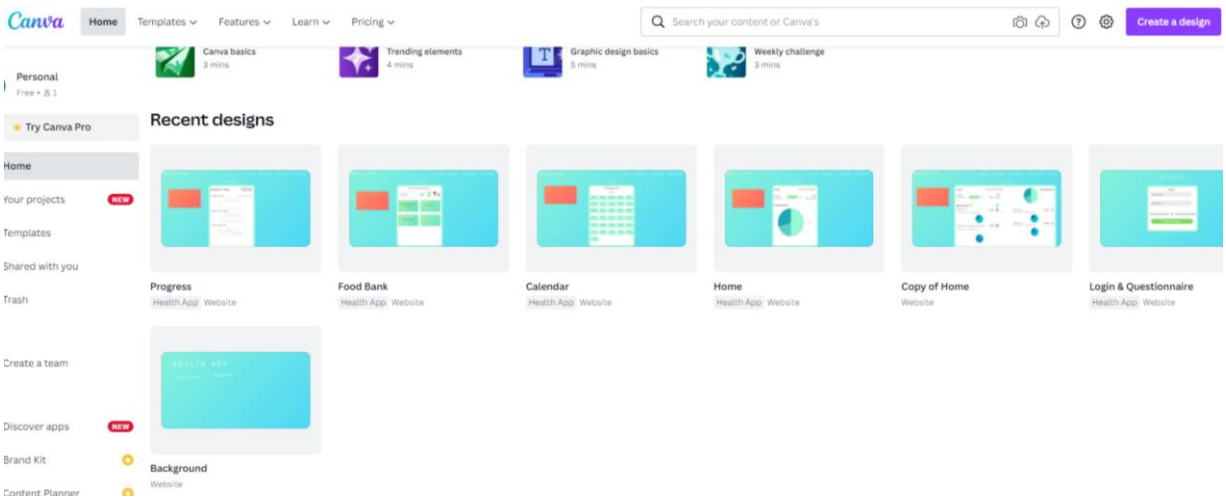


Foods, Previously eaten foods, Personal foods, Weight history, Users, Fact Bank

	
Custom Plan	
	

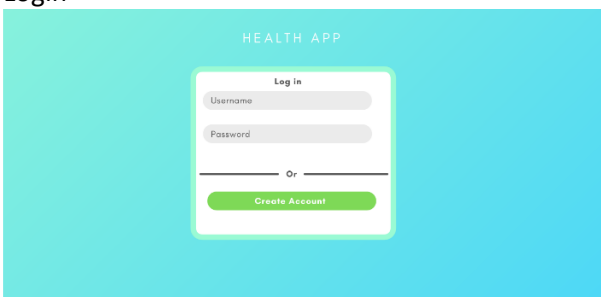
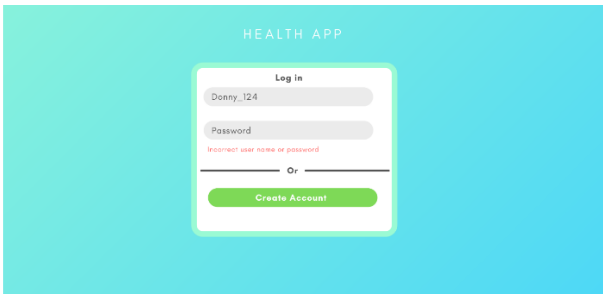
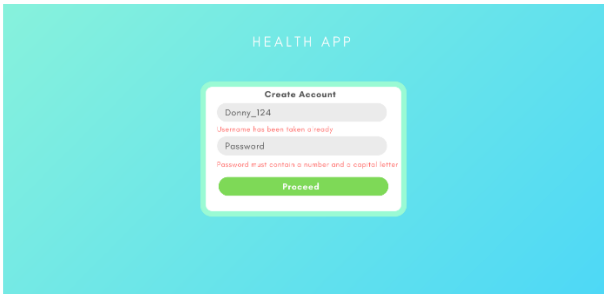
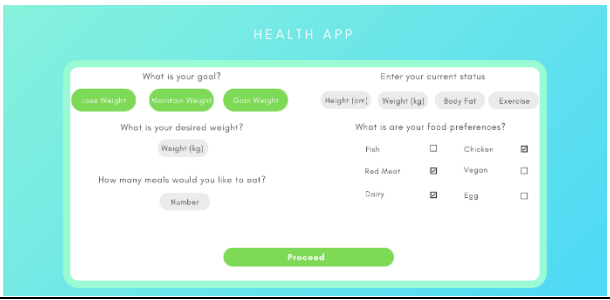
The Usability Features

This section portrays some of the features users will have access to. As the proposed solution is a web-app, it is logical to display a potential plan of what each page would look like visually to the user. This is based on the success criteria and the features of the proposed computational solution both of which are part of the analysis section of this report. Furthermore, the solution can be easily decomposed into various miniature processes that fall under multiple sections which represent pages. This is illustrated in the design section under the heading: Decomposition and the structure of the solution.



I used a website called “Canva” to design these templates because the website offered a wide range of tools that can be used to make attractive templates, something that was important to my stakeholders was in fact the idea of attracting users with an elegant design.

Page/Feature	Justification

Login     	The login page acts as an intermediary between the user and the actual program. This feature is necessary to valid authentic users and then forward them to being able to access the entire program as a whole. Keeping this design minimal focuses solely on the login page and makes it easy for the user to read the page. The program checks if the username that has been entered has already been taken. This is done to avoid data collision with other user's that share the same name hence why the program asks for a unique username for new users. As people may feel the data the program is using is sensitive and that they wouldn't want other people to access their plan and change their data. This is why creating a strong password is important therefore the program requests that the user includes at least one capital letter and number, making it harder to guess for others. The user has a choice of three options in terms of defining their goal this will then adjust the solution to the user's goal, providing the best plans for them. For calculation purposes, the user's height and weight have to be in cm and kg respectively. Entering body fat level leads to a drop-down menu that consists of three options; low, average and high. The same applies to exercise level; Light, moderate, heavy. Displaying a checklist for the user's food preferences makes it easy for the user to specify the foods they are comfortable eating which will lead to the solution recommending the right foods for the user.
Home	I have included this because most apps and devices nowadays use a home page that provides a brief overview of data that may be useful to the user. As this is a common layout, most people will find this easy to use.

This page follows a logical, list-type layout to ensure the user can view each piece of data one at a time and not be overwhelmed by the amount of information that is being displayed.

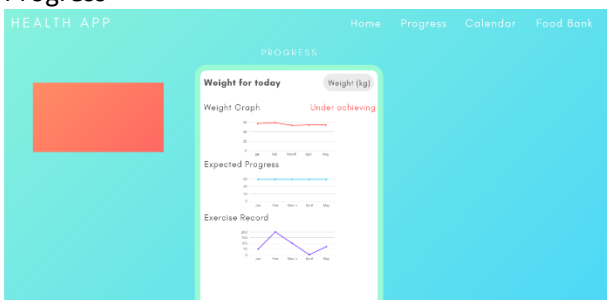
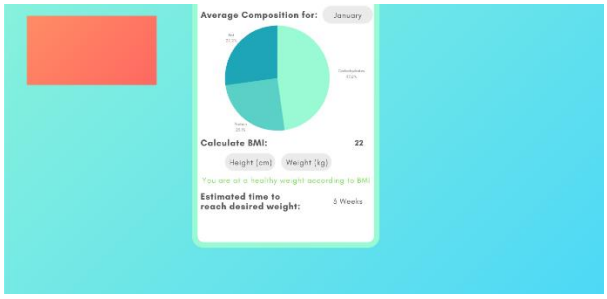
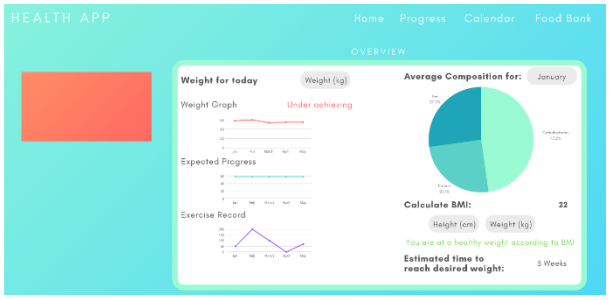
The use of a progress bar and several pie charts to display information is a different style of presenting information as opposed to ordinary text and so it becomes data that stands out to the user. Because this data is the most important, it should stand out and be easy to interpret.

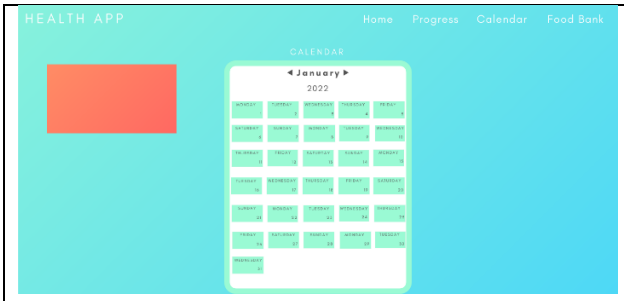
A checklist format to mark off the meals the user has eaten. As well as providing the option to add new meals by clicking the plus sign and remove meals by clicking the minus signs. The recycle icon next to “meal plans” generates a completely different set of meals but it will first offer a drop-down menu asking if the user would like to select food from the food database or their personal one. The same applies to the small recycling icons which will replace a single meal instead of the entire plan.

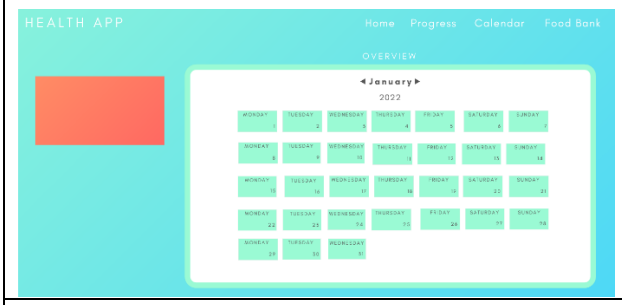
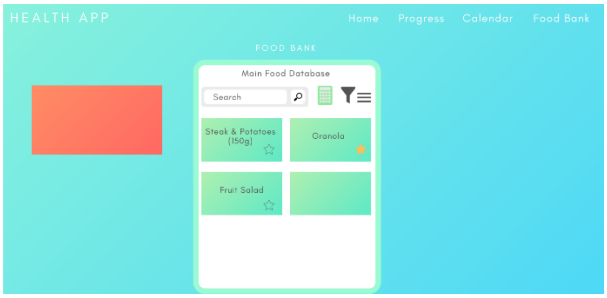
The user has the option to change their preferences by clicking the three lined button. Doing so will open a drop down menu providing a checklist of a number of preferences the user can choose from. The use of drop down menus is so that the user does not need to go to another page to access the feature, they can access the feature on the page that they are currently on. In this case, the use can modify their preferences while remaining on the home page

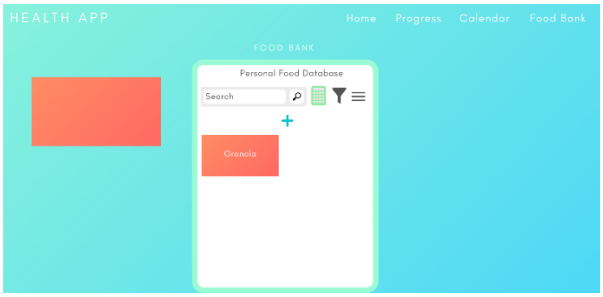
The user can add water by clicking the plus sign as well as logging their exercise activity and both water intake and exercise activities can be removed by clicking the minus sign

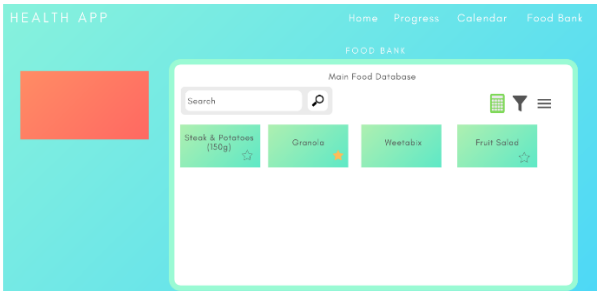
I have also created two formats of what the pages can look like. This is to illustrate how the pages would look for a user with a small screen size and how it would look for a large screen size. I have included the two layouts for each page in this section.

Progress    	This page is needed in order to keep an organized sector of all the other features that fall into this category such as: Weight record, exercise etc. The use of colour-coding the different line graphs help the user read them as they stand out differently from each other. This can also show the user's progress with the use of green to show "achieving goal" and red to show "under achieving". The use of line graphs is generally to show a change in a variable over a given time period. As this is common, users will be familiar with this style of representing data. Hence why I have included these features on this page to make interpreting data simple for the user. Using a pie chart to display average composition for a selected time period is justified because I have used pie charts to show nutritional composition in the home page when displaying foods. The different colours on the chart represent fat, protein and carbs – the main nutritional components that need to be amended for users. A drop-down menu will appear allowing the user to select a specific month or year or just show the data from when they first started using the app to the present day. BMI is a very popular method of calculating an ideal weight for someone. Although it does have its flaws, I have still included the feature of calculating a user's BMI when they enter their weight and height values. The idea of colour coding text to show if the BMI calculated is considered to be healthy or unhealthy saves the user time from having to research what BMI values mean. The last feature on this page shows the estimated time to reach the user's desired weight. This feature does not allow user input because there is no need to do so, this is just to tell the user how long it will take to reach their goal based on their current progress.
Calendar	Including this is essential because it is very useful for the user to keep track of time and generally plan ahead for the future even if

<table>
<tr>
<td> 

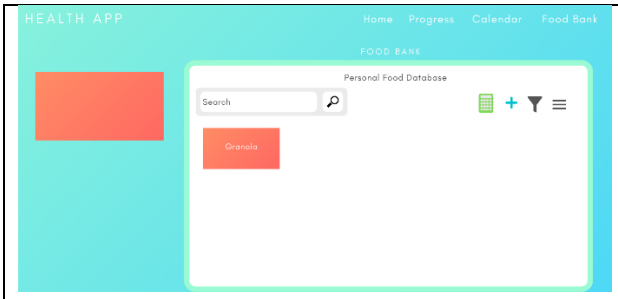

</td>
<td>they decide to use this for a purpose that does not coincide with the solution.</td>
</tr>
<tr>
<td>Food Bank
 




</td>
<td>This page is needed for the solution as this is where the user can interact with the main database that will be part of generating meals for the user.

Colouring these meal-boxes differently to each other makes it easier for the user to identify them both. The filter logo allows the user to apply food preferences as filters when searching the database to save time for the user if they would like to search for a specific type of food. The three-lined button switches the page to view the user's personal food database and vice versa. The plus sign on this page allows the user to add their custom foods to this database. When they want to add a meal, a small window will pop-up asking the user to input all the required nutritional values of their food. The small calculator represents the nutritional values per 100g calculator – This feature saves the user a lot of time if they need to calculate nutritional values for a certain weight of food.

Here I have illustrated what the page will look like on both formats on either a small screen or a large screen. This is because not all users will have the same screen size.</td>
</tr>
</table>

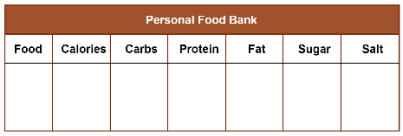
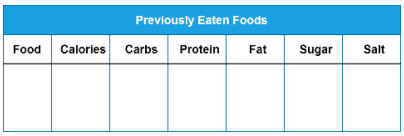
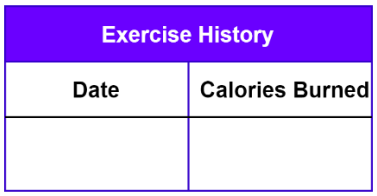
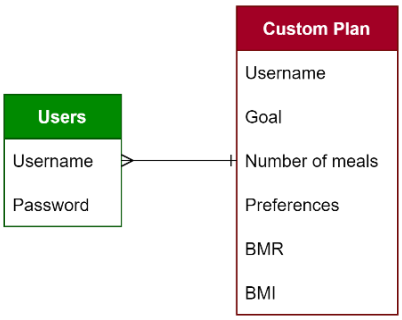
Key Variables, Data Structures and Classes

Process	Variables, Data Structures, Classes	Justification
Create Account	Create_Username Create_Password	The user is required to enter a username and password which will be stored in the two variables. This is so that a subroutine can be used that compares the two variables to the set of standards. The user will have to enter a unique username and a strong password, this will be checked by the subroutine.
Questionnaire	goal_answer desired_answer preferences_answer meals_answer	Creating separate variables for each question may seem like an inefficient method however it is much easier to trace through a subroutine with variables as opposed to normal code. The value of these variables is determined by the user's input which will be their answer to the questions. These variables will also be added to the database.
Existing User	Username password	This is required in order to compare what the user has inputted, to what is held in the database. Storing the user's inputs in variables allows this comparison to take place.
Meal Plan	Meal1 Meal2 Meal3 Total_Values Target_Values Meal_Values Target_Calories Mark (Boolean)	This is needed to generate the actual meals for the user from the main food database. Storing these in variables means they can be used in subroutines which is needed in case the user wishes to generate new meals. Furthermore, the app needs to

		adjust the main target counters when a meal has been eaten which can be done via storing data in variables.
Pie Chart	Total_Values (Array)	The total nutritional values will need to be displayed. As there are three components of nutritional values, an array would be suitable for storing such values. Arrays store values of the same data type which is suitable as all three values are likely to be floats
Water Intake	Water_Intake Water_Unit	The water intake target will need to be updated depending on how many water units the user marks. With the use of variables, the user can input many water units with each adjusting the water intake target.
Exercise Activity	Exercise_Target Activity Burned	These two variables will be used in a subroutine to allow the user to record multiple exercise activities. Hence why storing values in variables is suitable.
Change Preferences	Preferences (Array)	All preference options are values stored in an array. When the user changes their preferences, the values on their custom array are removed or added. Using an array here is more efficient than using separate variables for each preference.
Current Goal	Change_goal BMR (Function) Gain Maintain Lose	The Change_Goal variable checks if the user has decided to change their goal. If they have, the BMR function will be executed and will pass the user's latest status through its parameters to calculate a new BMR. As each goal has different requirements, it makes sense to store each goal in different variables. The user will then be able to select their new goal.
Line Graph	Weight_X (Array) Weight_Y (Array) Expected_X (Array) Expected_Y (Array) Exercise_X (Array) Exercise_Y (Array)	The x axis will show the time period and the y axis will show the user's weight. Both of which are taken from the personal weight database. Using an array here makes sense as we are only

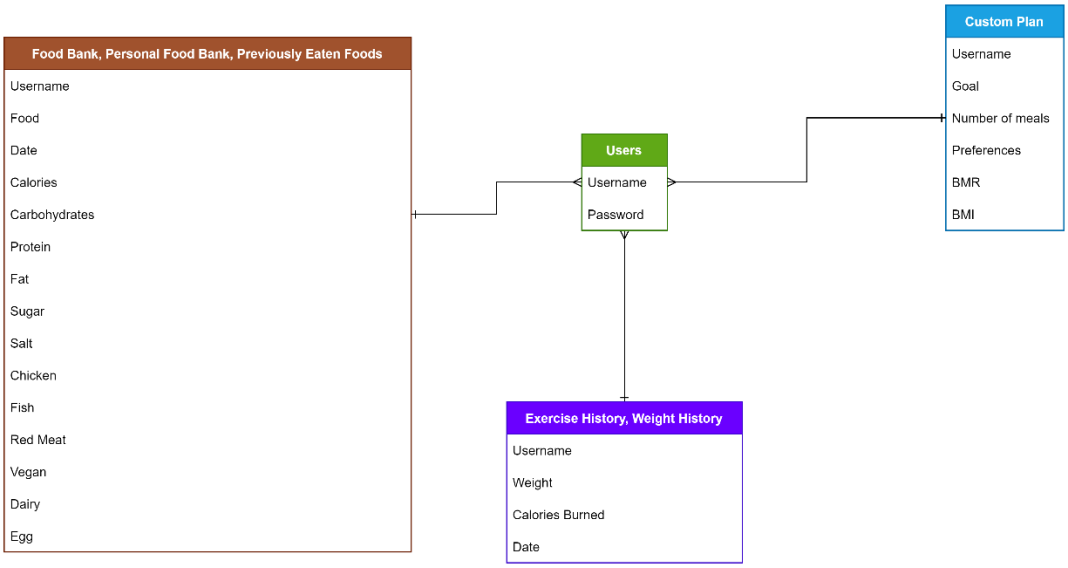
		working with one data type for each axis. Furthermore, arrays can store many data values in an orderly fashion which makes it a suitable choice for plotting data on a graph.
Current Weight	Today_Weight	A simple variable that will require the user to input their current weight and stores the value. This will then be added to the personal weight database. Using this is easy to implement hence why I have included this.
Estimated Diet Makeup	Average_Values	The user can select a time period from which the web-app will review the nutritional values consumed for the user over that period. The average will be calculated for each of the three indexes and will then be added to the average_values array. From here, the values of this array can be used to form a pie chart. The reason why using an array for this particular feature is to make the code more efficient. If variables were to be used, it will become very difficult to execute as for each data point from the database a variable will have to be used to store that value. Hence why using an array for a large number of values is needed.
BMI Calculator	BMI_Weight BMI_Height BMI (Function)	The user will be required to input values to these variables so that it can be passed through the parameters of the BMI function. As the user may want to calculate this multiple times at different time periods, a function is needed for this feature. Using the listed variables will be used as part of the function.
Calendar		
Plan Ahead	Plan_Ahead (Function) Date	When the user clicks on a date, they can generate a meal plan for that date. This will require a function as the user may want to repeat this for different dates

<table>
<tr><td></td><td></td><td>and generate different meal plans.</td></tr>
<tr><td>Search Food</td><td>Favourite (Function)
Meal</td><td>When the user clicks on a meal, they have the option to favourite the meal which will add it to their personal food database. This will execute the favourite function which will have the meal variable in its parameter. The use of a function here is so that this process can be executed many times with different values passed through its parameters hence why it is suited to this feature.</td></tr>
<tr><td>Add Own Food</td><td>Add_Food (Function)
Calories
Protein
Carbohydrates
Fat</td><td>This will allow the user to add their own meals to their own database with the four parameters; calories, protein, carbohydrates and fat. This is because the solution takes into account of these values and is based around these nutritional values entirely.</td></tr>
<tr><td>Green-Blue</td><td rowspan="3">No Variable Required</td><td rowspan="3">These are style related properties which is why variables will not be needed. However, these can be used from an external CSS file which can be applied throughout the web-app. Using an external file is more efficient than individually applying styles to each section of text and is suited because the web-app follows a general uniform. Hence why an external file should be used.</td></tr>
<tr><td>Red-Peach</td></tr>
<tr><td>Kumbh Sans</td></tr>
<tr><td>Users</td><td><table><tr><th colspan="2">Users</th></tr><tr><th>Username</th><th>Password</th></tr><tr><td></td><td></td></tr></table></td><td>This has two fields: username, password. This is so that the program can validate users to see if the typed password matches with what is written on the database as well as checking if the user is creating an account with a username that has already been taken – meaning it is already in the database.</td></tr>
<tr><td>Foods</td><td><table><tr><th colspan="7">Food Bank</th></tr><tr><th>Food</th><th>Calories</th><th>Carbs</th><th>Protein</th><th>Fat</th><th>Sugar</th><th>Salt</th></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table></td><td>This has seven fields: Food, calories, protein, carbohydrates, fat, sugar, salt. Each food must display these values as they are</td></tr>
</table>

<table>
<tr><td></td><td></td><td rowspan="3">fundamental to the entire solution's purpose. As the user will desire a wide range of options to choose from, there will be many records in the database each of them containing many nutritional values. Working with such a large amount of data is why a database is needed.</td></tr>
<tr><td>Personal Foods</td><td> 
</td></tr>
<tr><td>Previously Eaten Foods</td><td> 
</td></tr>
<tr><td>Exercise History</td><td> 
</td><td>This has two fields: Date, Calories Burned. When the user inputs an exercise activity, it is added to the database with the corresponding date. Storing such data in the database makes it easy to handle hence why it is suitable.</td></tr>
<tr><td>Weight History</td><td> 
</td><td>This has two fields: Date, Weight. When the user inputs their current weight each day, it will be added to the database.</td></tr>
<tr><td>Custom Plan</td><td> 
</td><td>Each user will have different ideas on how they would like to use the web-app. As every user is different, it does not make sense to have the solution create such a rigid plan with no personalization being taken into account. This is why creating a new database called custom plan is necessary. Classifying this database as relational allows each user to have a store of data that takes into account of more of their personal preferences.</td></tr>
<tr><td>Fact Bank</td><td> 
</td><td>This database will be used to store facts which will be displayed in boxes throughout the web-app. Because there are lots of facts that can be useful to the user, this means there is a lot of data that is being handled</td></tr>
</table>

After reviewing the design of my solution, I have concluded that the idea of searching through multiple different flat-file databases each of which corresponds to a separate user, is far too time-consuming and can be difficult to complete. Therefore, I have decided to restructure my design for

the database and merge the different flat-file databases into a large relational database. This means that there will have to be an extra layer of filtration when a user queries the database so that they can only view the data that belongs to them. I have illustrated the idea of using a relational database below;



Test Data

Testing should be executed in order to check if the program is working correctly. The following table shows the individual sub-processes I have decomposed the solution into, with added test data. Each test can be performed at the end of the development cycle and this ensures that each process is tested so that it can reach my requirements of the solution which is listed in the success criteria. Testing in the iterative development of the solution mainly involves looking out for syntax errors and logical errors. Although syntax errors will not need to be tested thoroughly because the IDE can correct them, logical errors will have to be tested over and over again to ensure the right outcomes are being presented by subroutines. The use of white box testing involves checking parts of my solution that the user will not see, which in my case is to make sure functions, if statements and input variables are accepting the right data so they work properly. This is why I have created a brief plan of testing using valid, invalid and erroneous test data. When developing my solution, it is likely that I could think of more test data to be used that I have not thought of in advance and so even though I have not planned some tests, testing on the spot can also add to the robustness of my solution.

Process	Test Data	Expected Outcome	Type	Justification
Create Account	“golden”	Password is not strong enough. Must contain at least one capital letter and one number	Invalid	As the username can take any value, it will not need to be tested however as the password has strength requirements, using invalid testing can make sure a message for the user is displayed in the likely event of the user

				choosing a password that is not strong.
Questionnaire	"-0.340"	Invalid height/weight	Invalid	As height and weight cannot be negative, this test must be executed to prevent further miscalculations part of the solution.
Existing User	"Donny_124" "Golden1"	Welcome!	Valid	For this test, there must be a username of Donny_124 and a password of Golden1 already present in the database. The actual test will consist of entering these exact details on the login page. This is needed in order to authenticate the user.
Meal Plan	Generate meal plan	Output new meal plan	Valid	This is to check if the right result is outputted when the user clicks on "generate meals" or if they decide to generate food from their personal database. This does not allow the user to input any extreme values hence why an invalid test is not suitable.
	Generate food from personal database	Output food	Valid	
Pie Chart	Calculate protein, carbs and fat	Output pie chart with the right proportions	Valid	The program will have to work out the proportion of the three components the user will need to eat based on their goal and target. A valid test method can ensure that this works properly hence why this method should be used.
Water Intake	Add water unit	This exceeds your recommended water intake for the day	Process	The user can input a water unit however if they input a water unit that exceeds the target, the following error should come up. The process is being tested to check if the program can identify if the water units have exceeded the water intake – the procedure needs to be checked.

Exercise Activity	Remove activity	No output	Invalid	The program should recognise that the user the exercise log for the day is empty and so no activity can be removed in the first place. As this is data the program should not expect, an invalid test is suited
Change Preferences	“Fish”	No output	Valid	The user will check boxes which represent preferences. These preferences should then be added or removed to the custom plan database. An invalid test is not needed here as the user will not have the option to enter extreme values.
Current Goal	“Gain”	Output new goal	Valid	The user will have three options to choose from to change their goal. As no extreme values can be entered, a valid test is suitable.
Line Graph	Generate points	Output line graph	Output	As this is just a line graph, there is nothing much to test apart from displaying the graph correctly. Hence why a valid test is necessary.
Current Weight	“green”	“Invalid weight”	Erroneous	Weight cannot take be a worded value hence an erroneous test is best suited for this because I am using a data type that should be rejected completely.
Estimated Diet Makeup	“January”	Output pie chart for January	Output	As this is just a pie chart, there is nothing much to test apart from displaying the graph correctly. Furthermore, there are not extreme values that the program needs to reject hence why a valid test is suitable.
BMI Calculator	“0.56”	“Invalid weight/height”	Invalid	The program should not take a value this small even though it is positive.

				Hence why an invalid test is needed.
Calendar	Calendar	Output calendar	Output	A valid test is needed to check if the program can display the calendar.
Plan Ahead	"21st"	Output meal plan for 21st	Valid	This valid test checks if the user can successfully generate a meal plan for a set date. An invalid test is not necessary as this does not require any specific input from the user which means they cannot input an extreme value. Hence why a valid test is needed.
Search Food	"Verb"	No Output	Valid	The program should accept this value and make the attempt to search for it. No extreme value is needed and so a valid test is necessary.
Add Own Food	"700", "60", "60", "30"	Add custom food to personal food database	Valid	The use of valid testing here is to check if the user is able to input these nutritional values and that they should be added to the corresponding database.
Database	Insert values into relational database	The correct values should be displayed when queried	Process	Manipulating data to and from the database is a huge part of my solution and so initially testing to see if the database can be searched and is recording data appropriately is essential.

In these test plans, I have not included the styling of the webpage as this is something that is not so complex and does not require much valid or invalid testing.

Further Data

Section	Test & Data	Expected Outcome	Type	Justification
Login page	**Creating an account:** "Theta" "Theta123"	Redirection to the questionnaire	Valid	The user needs to enter a username and password that satisfies the requirements. This needs to be tested so that users can be validated in order to use the web app.

	Creating an account: “$%^” “Password123”	No response	Erroneous	Most websites do not allow these characters to be entered as usernames when creating accounts. This must be tested to see how the web app will respond to such inputs.
	Logging into an account: “Theta” “Theta123”	Redirection to the home page	Valid	The user will not have the web app open at all times, hence why the final solution must allow the user to log back into their account. The user may close their browser and then go back to the web app.
	Questionnaire weight: “-999999999”	Error generated for the user	Invalid	This test is designed to check if the web app can process the user’s input and confirm if it is within an accepted boundary
	Questionnaire desired weight: “hello”	Error generated for the user	Erroneous	There is always a chance a user may enter the wrong data type for certain inputs, a fairly common theme in human error. Desired weight can only accept decimal numbers.
Home page	**Meal plan generation**	Foods displayed with nutritional values, corresponds to preference for number of meals	Process	This is the most important feature of my solution. Apart from clicking buttons, the user’s input is limited which makes it easy to use for the user as well as adding robustness to the solution because there is less to test.
	Delete a meal from the plan: When the plan for the day is empty	No response	Invalid	To add to the robustness of the solution, the web app should not do anything in this scenario where the user attempts to delete a meal from a plan that is already empty. The web app should recognise that there is nothing to delete from.

	Mark foods as eaten	An updated calorie target for the day	Valid	When the user eats a food, the web app should adjust the rest of the user's calorie target to take into account of the eaten food/s. This is another vital component of the solution.
	Water intake	Water intake counter should increment	Valid	A test to see if the user can log their water intake for the day with the click of a button.
	Exercise input: "250"	Displayed input	Valid	Testing this feature is a specification point from my success criteria.
	Exercise input: "10000000"	No response/error	Invalid	The web app should recognise when the user inputs the accepted data type, in this case numerical figures.
Progress page	**Input weight:** "ff$%"	No response	Erroneous	The webapp should recognise that the input is not a decimal number and so it should not be processed. This prevents any user from easily breaking the solution by having it process erroneous data
	Select month	The line graphs showing data points from the selected month	Valid	The user should be able to select a month and have their data points for the month displayed on graphs. This is so that the user can review their progress.
	BMI calculation: "165" "57"	A number displayed that indicated the BMI based on inputs	Valid	Based on input, the user's BMI should be calculated. This is a specification point on my success criteria.
	BMI calculation: "Testing"	No response	Erroneous	This feature is a calculator and so it should not operate on a string datatype.
Calendar	**Select month:** "July" "2022"	All days in the selected month displayed	Valid	Allows the user to plan ahead. User input is limited here, as they should be able to select a month.
	Generate food plan for a	An icon generated on the corresponding date	Valid	

	specific date: "7th July 2020"			No typing-input is used here as the web app is looking to generate a meal plan for a future date.
Food bank	**Add food to database:** "Fried Chicken Wings"	The new food should be shown along with the other foods part of the database	Valid	As this is a specification point from my success criteria, this feature should be tested.
	Calculate nutritional values: "watermelon"	No response	Erroneous	As it is a calculator, the range of inputs that should be accepted should be limited to numbers. Any other datatypes such as a string should not be processed by the web app.

Once I have developed my solution, it is possible that I could think of more tests to execute, this could help detect any potential flaws in the final solution.

Development

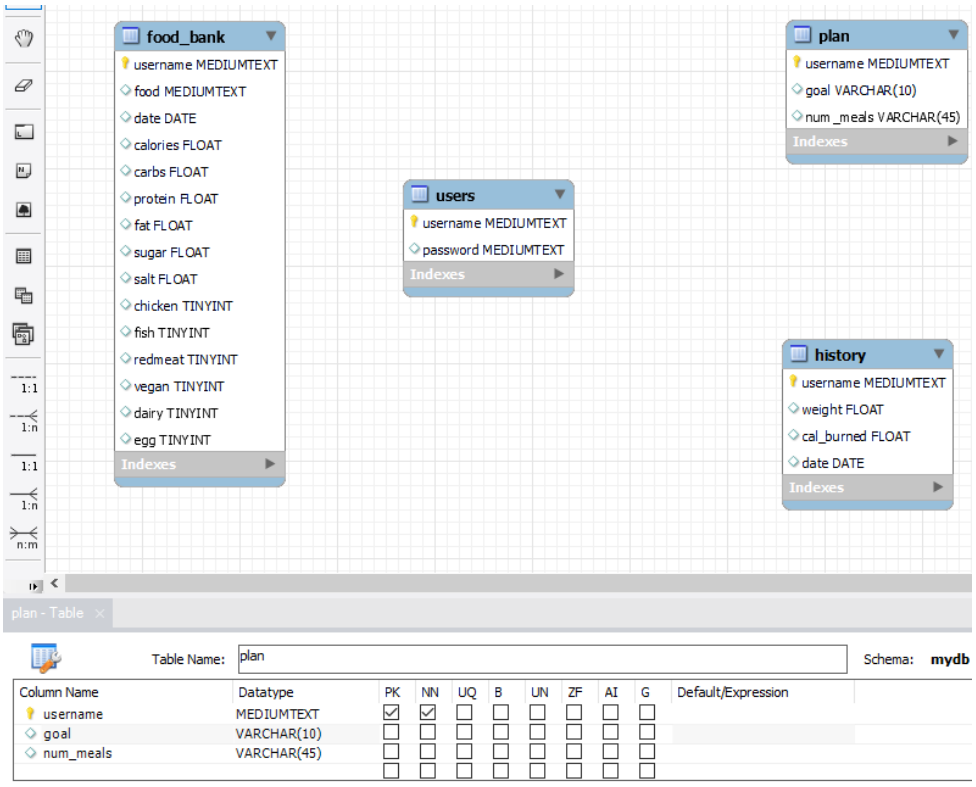
The Database

I decided to start with the database module of my solution. This is because the entire solution is centered around the relational database so it is logical to start developing this first in order to act as a platform to build on further.

Aim: Model an entity relationship diagram in the MySQL Workbench interface.

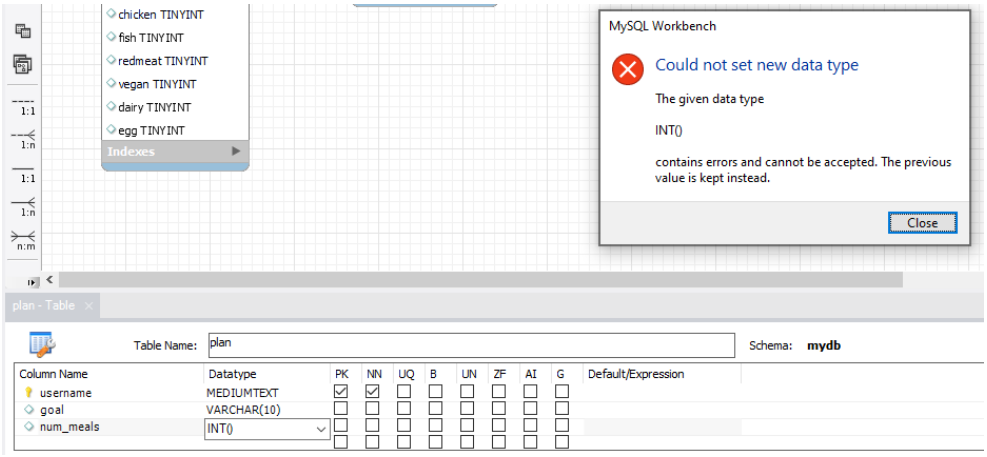
Justification: To familiarize myself with this database management system and to see if the model would be accepted.

Underneath each table name has datatypes listed Different datatypes needed to be identified – not all fields will require the same data



The different tables I had planned in my design

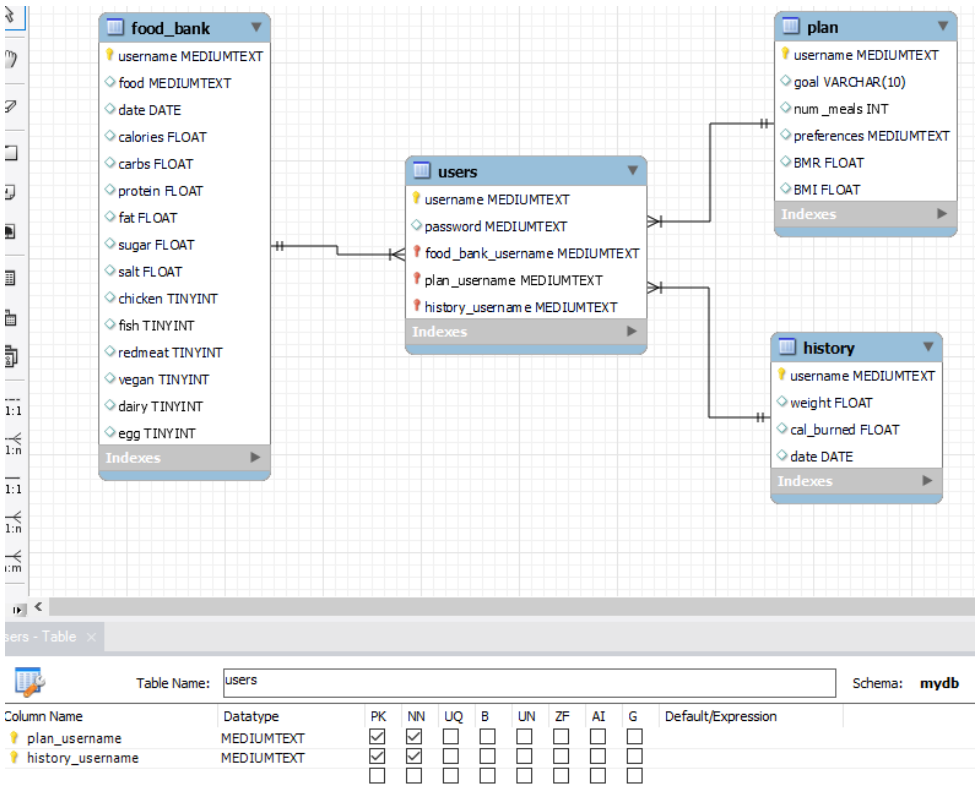
I managed to complete the “users” table and the “food_bank” table however whilst creating the “plan” table I encountered the following error;



Error was generated because I stated the datatype as INT() rather than INT without parameters.

After a quick search on google, I realised that MySQL is very peculiar when specifying datatypes. INT() requires a value for its parameter whereas stating the datatype as INT was accepted by the workbench.

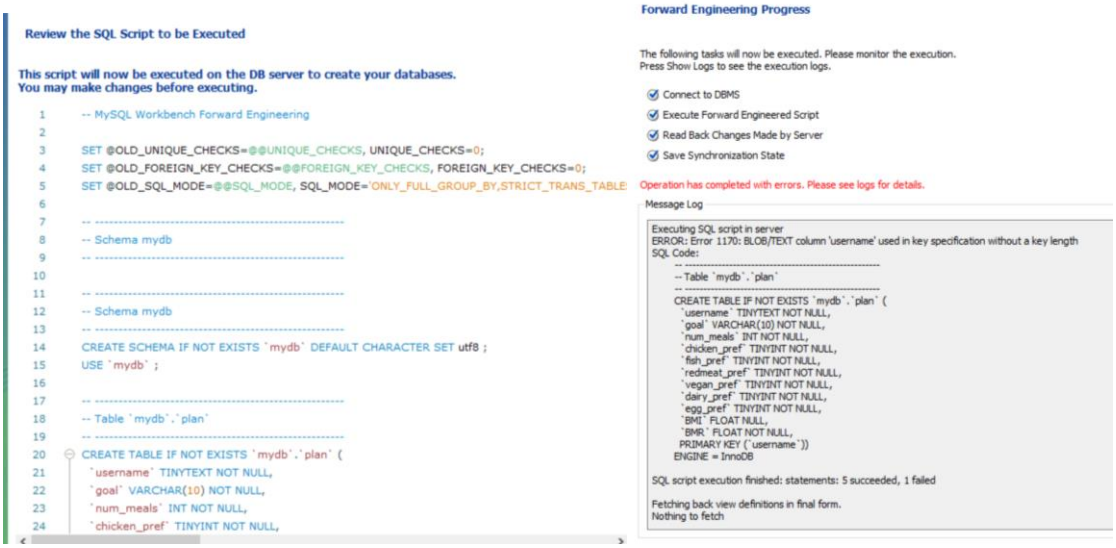
Following this, I decided to create the last table part of my plan – "history" and then established relationships between the tables to collectively form the database itself.



Aim: Physically generate the database from forward engineering the ER diagram.

Justification: The alternative to generating the database would be to manually enter SQL code, using the diagram for the generation is much more efficient.

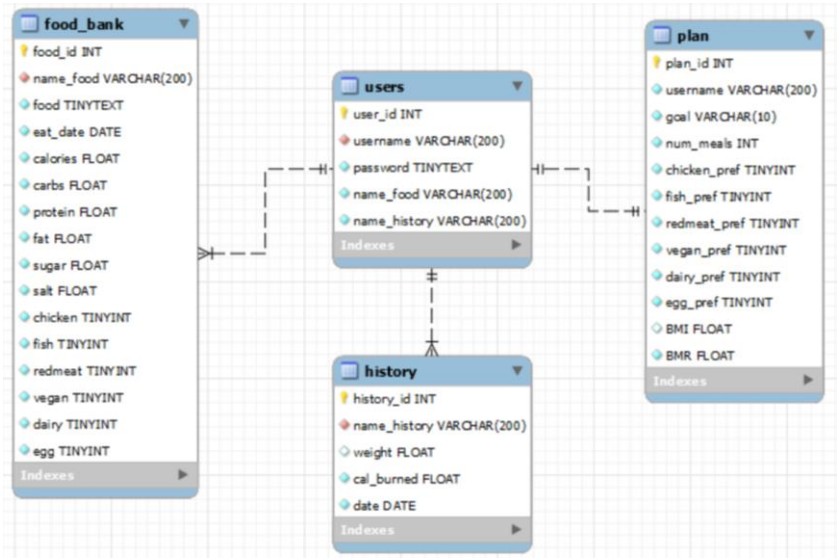
The code that was picked up by MySQL work bench from the ER diagram



The error that was generated because the varchar of the primary key was not specified

This error was produced during the process of generating the database. After researching online, I discovered that this is a common error when using a primary key with the data type TINYTEXT. I decided to change the username data type to varchar(200) although it has 55 characters less than TINYTEXT, It should still be enough given that the average username is much shorter than that.

After a few adjustments, this model was generated into a schema recognised by MySQL.

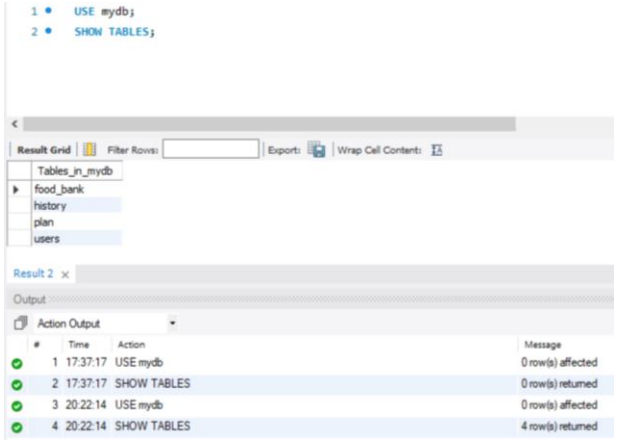


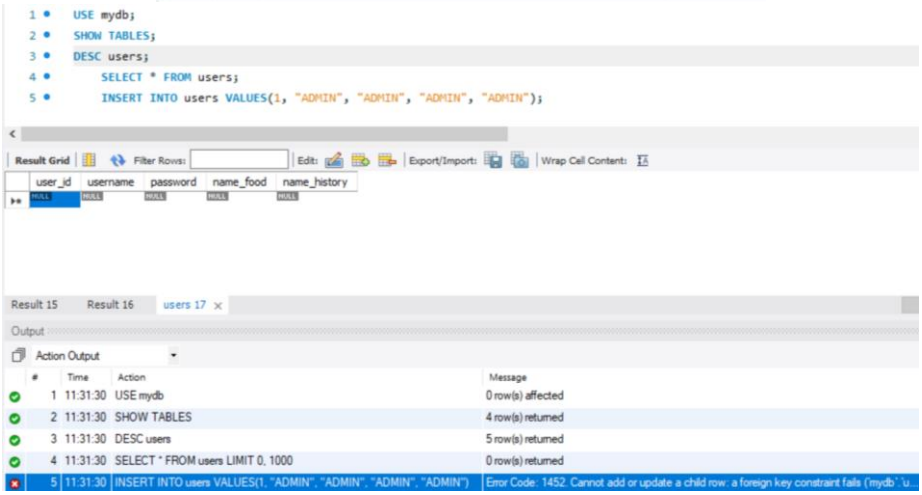
Test: Try inserting values into the newly created database to see if does so correctly.

Data: 1, “ADMIN” – Valid data

Justification: I needed to make sure there were no errors doing this as in the future, there will be many more insertions into the database.

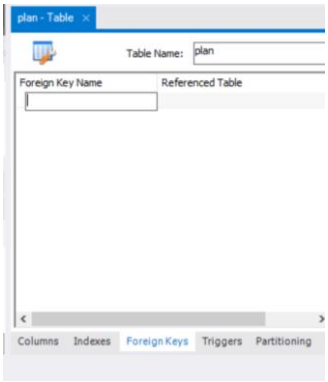
I first wanted to see how the Workbench interface would display the tables I created – this is what the SQL statement does





This error described a problem with the foreign key relationship I had setup in the database

The parent table of the foreign key relationship has to be users because it is the table that will be updated first as part of my finished solution – when a user first creates an account.



The foreign key name and referenced table is empty, this indicates the plan table has become the parent table when it should have been a child table to the users table.

Modification: Create a new database model so that the parent table is the "users" table

Justification: The "users" table had to be the parent table because it is the table that will have values inserted to it first, due to the nature of the solution – the users will have to create an account or log in before being able to use any other features.

Create Schema is referring to the creation of a new database on the MySQL server if it does not exist already

```
Review the SQL Script to be Executed

This script will now be executed on the DB server to create your databases.
You may make changes before executing.

-- MySQL Workbench Forward Engineering

SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;
SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0;
SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES

-- -----------------------------------------------------
-- Schema mydb
-- -----------------------------------------------------


-- -----------------------------------------------------
-- Schema mydb
-- -----------------------------------------------------
CREATE SCHEMA IF NOT EXISTS `mydb` DEFAULT CHARACTER SET utf8 ;
USE `mydb` ;

-- -----------------------------------------------------
-- Table `mydb`.`users`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `mydb`.`users` (
  `user_id` INT NOT NULL AUTO_INCREMENT,
  `username` VARCHAR(200) NULL,
  `password` TINYTEXT NULL,
  `name_food` VARCHAR(200) NULL,
  `name_history` VARCHAR(200) NULL,
  `userscol` VARCHAR(45) NULL,
  PRIMARY KEY (`user_id`))
ENGINE = InnoDB;


-- -----------------------------------------------------
-- Table `mydb`.`food_bank`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `mydb`.`food_bank` (
  `food_id` INT NULL AUTO_INCREMENT,
  `name_food` VARCHAR(200) NULL,
  `food` TINYTEXT NULL,
  `eat_date` DATE NULL,
  `calories` FLOAT NULL,
  `carbs` FLOAT NULL,
  `protein` FLOAT NULL,
  `fat` FLOAT NULL,
```

"Create table" is the SQL command that creates a table with the specified column names and data types that follow after.

The primary key constraint and index specifies which column in the table should act as the primary key and so it must be unique

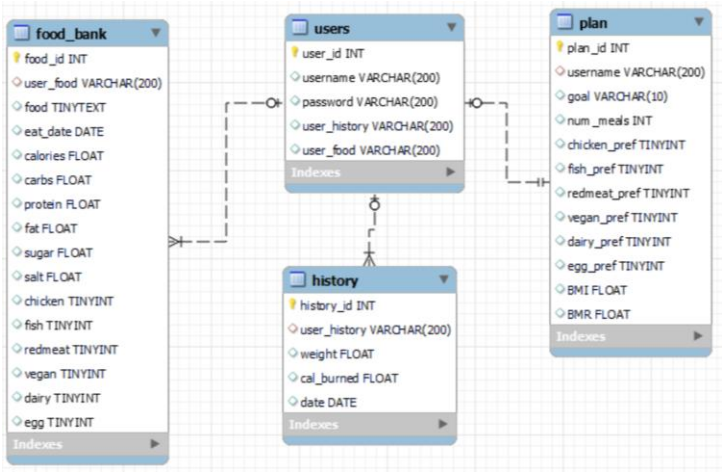
```
  `sugar` FLOAT NULL,
  `salt` FLOAT NULL,
  `chicken` TINYINT NULL,
  `fish` TINYINT NULL,
  `redmeat` TINYINT NULL,
  `vegan` TINYINT NULL,
  `dairy` TINYINT NULL,
  `egg` TINYINT NULL,
  PRIMARY KEY (`food_id`),
  INDEX `fk_food_bank_users1_idx` (`name_food` ASC) VISIBLE,
  CONSTRAINT `fk_food_bank_users1`
    FOREIGN KEY (`name_food`)
    REFERENCES `mydb`.`users` (`name_food`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION);


-- -----------------------------------------------------
-- Table `mydb`.`history`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `mydb`.`history` (
  `history_id` INT NULL AUTO_INCREMENT,
  `name_history` VARCHAR(200) NULL,
  `weight` FLOAT NULL,
  `cal_burned` FLOAT NULL,
  `date` DATE NULL,
  PRIMARY KEY (`history_id`),
  INDEX `fk_history_users1_idx` (`name_history` ASC) VISIBLE,
  CONSTRAINT `fk_history_users1`
    FOREIGN KEY (`name_history`)
    REFERENCES `mydb`.`users` (`name_history`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;


-- -----------------------------------------------------
-- Table `mydb`.`plan`
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS `mydb`.`plan` (
  `plan_id` INT NULL AUTO_INCREMENT,
```

The foreign key constraint has a "references" part to the statement which specifies which column in another table is acting as the parent in the foreign key relationship.

```
  `username` VARCHAR(200) NULL,
  `goal` VARCHAR(10) NULL,
  `num_meals` INT NULL,
  `chicken_pref` TINYINT NULL,
  `fish_pref` TINYINT NULL,
  `redmeat_pref` TINYINT NULL,
  `vegan_pref` TINYINT NULL,
  `dairy_pref` TINYINT NULL,
  `egg_pref` TINYINT NULL,
  `BMI` FLOAT NULL,
  `BMR` FLOAT NULL,
  PRIMARY KEY (`plan_id`),
  INDEX `fk_plan_users1_idx` (`username` ASC) VISIBLE,
  CONSTRAINT `fk_plan_users1`
    FOREIGN KEY (`username`)
    REFERENCES `mydb`.`users` (`username`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;
```

I edited the ER diagram and generated this new section of code to create a new database. I also changed the not null values to null as this would make it easier to insert data into the database. Null

values do not have to contain a value but if all columns were classed as not null, I would have to insert values into these columns every time.



This was the latest model to be accepted

There are many repeated columns in this database design, however changing the names slightly allowed this model to be generated successfully. If multiple columns from different tables had the same name, MySQL would treat them as tables that are linked together in a way. Now the database was created, I continued with my inserting-values test.

```
1 •  USE mydb;
2    INSERT INTO users(user_id, username, password)
3    VALUES(1, "ADMIN", "ADMIN");
4 •  DESC users;
```

Inserts into the specified table and columns with the values listed in brackets in the line below.

The 4th line describes the users table, meaning it shows the columns in the table.

```
1 •  USE mydb;
2 •  DESC users;
3 •  SELECT * FROM users;
4
```

Result Grid | Filter Rows: | Edit:

user_id	username	password	user_history	user_food
1	ADMIN	ADMIN	NULL	NULL
NULL	NULL	NULL	NULL	NULL

Result 7 | users 8

Output

Action Output

#	Time	Action
4	21:10:30	DESC users
5	21:10:30	SELECT * FROM users LIMIT 0, 1000

This displays all records in the users table and so showed that the values were inserted successfully

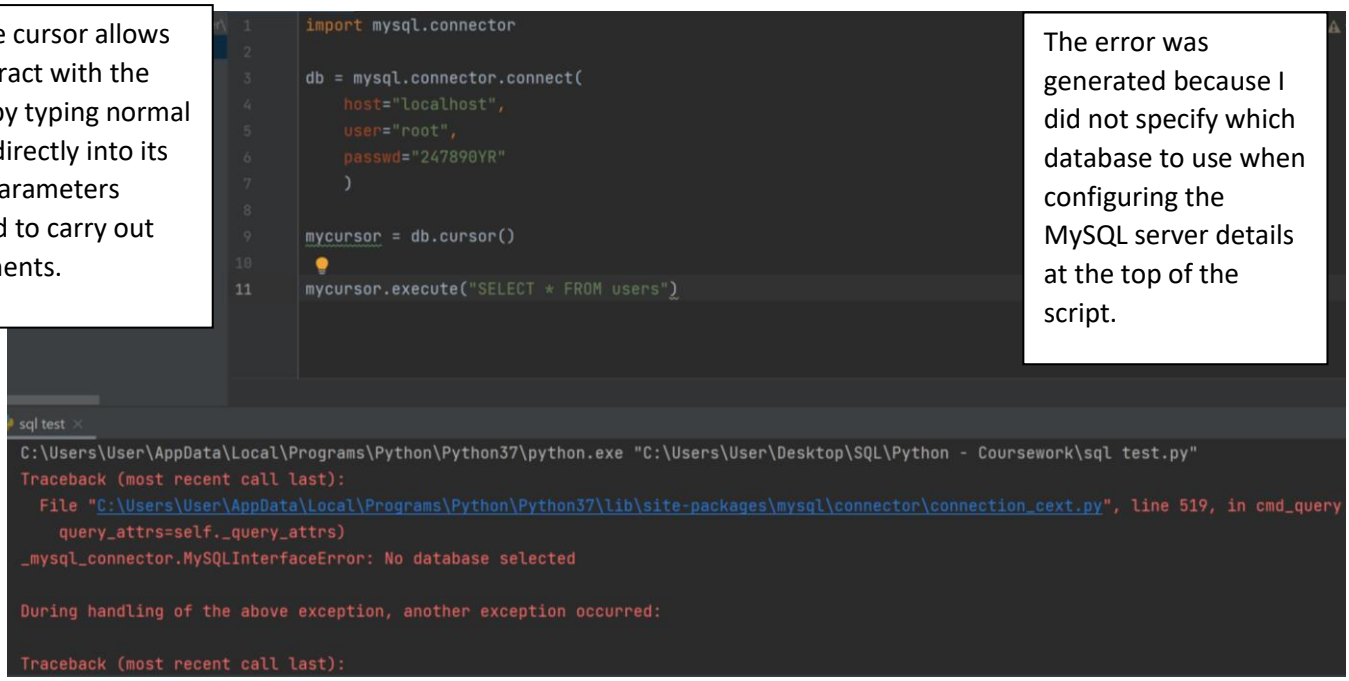
Login System

Aim: Run an SQL statement from a python file

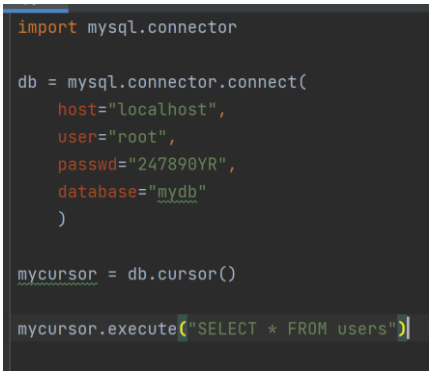
Justification: So, I can execute SQL statements/queries from any python file which is the backbone of my entire solution

This command ran successfully which meant python could import the connector with no issues. The connector comes part of installing MySQL and it allows python files to connect to the MySQL server.

A database cursor allows me to interact with the database by typing normal SQL code directly into its .execute parameters which used to carry out the statements.



The error was generated because I did not specify which database to use when configuring the MySQL server details at the top of the script.



I amended the code by entering the name of my database which returned no errors so it was fully connected to the database.

"Fetchall" is a function that retrieves all "select" statements stored in the database cursor

```
db = mysql.connector.connect(
    host="localhost",
    user="root",
    passwd="247890YR",
    database="mydb"
    )

mycursor = db.cursor()

mycursor.execute("SELECT * FROM users")

print(mycursor.fetchall())
```

```
    host="localhost",
    user="root",
    passwd="247890YR",
    database="mydb"
    )

mycursor = db.cursor()

mycursor.execute("SELECT * FROM users")

for i in mycursor:
    print(i)
```

```
(1, 'ADMIN', 'ADMIN', None, None)
```

My intention here was to fetch the record that currently exists in my database – ADMIN, and have python display the value. it printed the values correctly.

Aim: Use python code to validate the login details of a user

Justification: Familiarising myself with executing SQL in python and have a working user validation prototype would help me in the future part of the development.

To act as the login page, the code should ask the user to input something and check if that value is present in the "users" table. The following error was generated and most likely because the user variable used in the SQL statement was not recognised as a variable

This was another attempt at the issue but I tried assigning the variable Q1 to the actual SQL statement

```
    database="mydb"
    )

mycursor = db.cursor()

user = print(input(str("Please enter a username ")))

Q1 = "SELECT username FROM users WHERE username = "(user)

mycursor.execute(Q1)

for i in mycursor:
    print(i)
```

```
ADMIN
Traceback (most recent call last):
  File "C:/Users/User/Desktop/SQL/Python - Coursework/Login System.py", line 14, in <module>
    Q1 = "SELECT username FROM users WHERE username = "(user)
TypeError: 'str' object is not callable
```

I researched how variables can be used in python SQL statements and I found that with the use of string formatting it can be implemented - %s

Here I am using a variable called "user" which stores a user input.

This will be used as a filter when running queries.

```
    database="mydb"
    )
mycursor = db.cursor()

user = input(str("Please enter a username "))

mycursor.execute("SELECT username FROM users WHERE username =%s",(user,))

for i in mycursor:
    print(i)
```

%s – recognised as string formatting, it can use any python variable in the SQL statement.

```
C:\Users\User\AppData\Local\Prograr
Please enter a username ADMIN
('ADMIN',)

Process finished with exit code 0
```

This was the code that was finally accepted.

Adding a condition on the select statement verifies if the user's input exists in the database or not.

I am doing this because it is the main part of validating users – checking if they exist first or not.

```
import mysql.connector

db = mysql.connector.connect(
    host="localhost",
    user="root",
    passwd="247890YR",
    database="mydb"
    )
mycursor = db.cursor()

user = input(str("Please enter a username "))

check = mycursor.execute("SELECT username FROM users WHERE username =%s",(user,))

if check:
    print("The username ",user," exists")
else:
    print("INVALID USERNAME")

for i in mycursor:
    print(i)
```

```
mycursor = db.cursor()

user = input(str("Please enter a username "))

check = mycursor.execute("SELECT username FROM users WHERE username =%s",(user,))

if check != (user):
    print("INVALID USERNAME")

else:
    print("The username ", user, " exists")
```

```
C:\Users\User\AppData\Local\Programs\
Please enter a username ADMIN
INVALID USERNAME

Process finished with exit code 0
```

Even with "ADMIN" present in the database, the correct response was not returned. The result I was expecting was ADMIN to be displayed.

```
mycursor = db.cursor()

user = input(str("Please enter a username "))

mycursor.execute("SELECT username FROM users WHERE username =%s", (user,))

check = mycursor.fetchone()

if check:
    print("The username ", user, " exists")

else:
    print("INVALID USERNAME")
```

"Fetchone()" retrieves a single record only or none if no records are present

```
C:\Users\User\AppData\Local\Program
Please enter a username ADMIN
The username  ADMIN  exists

Process finished with exit code 0
```

After adding a "fetchone" statement, the code returned the right response. This makes it ideal for searching for a very specific value which is why it is suitable for searching for a person's exact username.

Test: Check if the system will not accept data because it does not exist in the database.

Data: "Donny"

Justification: This form of invalid testing was what I planned for in the design section, it adds to the robustness of my solution.

```
mycursor = db.cursor()

user = input(str("Please enter a username "))

mycursor.execute("SELECT username FROM users WHERE username =%s", (user,))

check = mycursor.fetchone()

if check:
    print("The username ", user, " exists")

else:
    print("INVALID USERNAME")
```

```
Please enter a username donny
INVALID USERNAME
```

The correct response was returned when I entered a value that should not be accepted.

Aim: Ensure the system allows the user to repeatedly enter more and more values until they enter a valid username.

Justification: This emulates the nature of a website – users will most likely keep entering lots of different values.

```
def login():
    user = input(str("Please enter a username "))


mycursor.execute("SELECT username FROM users WHERE username =%s", (user,))

check = mycursor.fetchone()

if check:
    print("The username ", user, " exists")

else:
    print("INVALID USERNAME")
    while (user) != check:
        login()
```

```
Please enter a username Donny
INVALID USERNAME
Please enter a username Donny
Please enter a username WOW
Please enter a username ADMIN
Please enter a username
```

Adding a while loop brought me closer to my idea however even when I entered ADMIN, the right response did not appear.

Using a function solved the issue by having the function run manually once but then have it call itself within itself – an example of recursion

```
def login():
    user = input(str("Please enter a username "))
    cursor.execute("SELECT username FROM users WHERE username =%s", (user,))
    check = cursor.fetchone()
    if check:
        print("Welcome",user,)
        print(check)
    else:
        print("Invalid")
        login()


login()
```

```
Please enter a username admin
Welcome admin
('ADMIN',)

Process finished with exit code 0
```

An issue arose regarding the case sensitivity of the data retrieved from the select statement. The program treated “admin” and “ADMIN” as the same value.

```
    user = input(str("Please enter a username "))
    cursor.execute("SELECT username FROM users WHERE username COLLATE utf8mb3_bin =%s", (user,))
    check = cursor.fetchone()
    if check:
        print("Welcome",user,)
        print(check)
    else:
        print("Invalid")
        login()

# TODO when user enters the correct user name and password, set the userid variable in pyhton to t
login()
```

```
Please enter a username admin
Invalid
Please enter a username ADMIN
Welcome ADMIN
('ADMIN',)
```

I added the COLLATE utf8mb3_bin which will instead look at the binary value of characters rather than their sort value when making comparisons.

A collation is a set of rules for comparing characters in a charset. The character set uft8mb3 may have multiple values with the same sort value. This is why “A” and “a” where being treated the same. The COLLATE statement reinforced case sensitivity.

I manipulated the code I wrote for validating usernames to then being able to validate passwords entered by the user as well

```
def password(user):
    userpass = input(str("Please enter your password "))
    cursor.execute("SELECT password FROM users WHERE username COLLATE utf8mb3_bin =%s", (user,))
    x = cursor.fetchone()
    if x:
        print("Here is your account",user,)
        print(x)
    else:
        print("incorrect passwd")
        password(user)

login()
```

```
password()
Login System
Please enter your password GOOGLE
Here is your account ADMIN
('ADMIN',)

Process finished with exit code 0
```

```
it(str("Please enter your password "))
'"SELECT password FROM users WHERE username COLLATE utf8mb3_bin =%s AND password COLLATE utf8mb3_bin =%s", (user, userpass))
:hone()
```

```
incorrect passwd
Please enter your password admin
incorrect passwd
Please enter your password ADMIN
Here is your account ADMIN
('ADMIN',)
```

The two variables; user and userpass, are being used in the SQL statement to filter queries. They are implemented using %s which is string formatting

Aim: Setup a flask application

Justification: This is the initialisation of my web app and so doing this now means I can easily link webpages to it and iteratively test each one.

This would involve me setting up a flask application and connect it to the M using templates for the actual website.

All flask applications need to be configured with a secret key to secure web session data related to the app. Hence why used the web config statement.

The flask module is being imported and so I can use its contents in my code.

```
from flask import Flask

def create_app():
    app = Flask(__name__)
    app.config['SECRET_KEY'] = 'carrot24'

    reutrn app
```

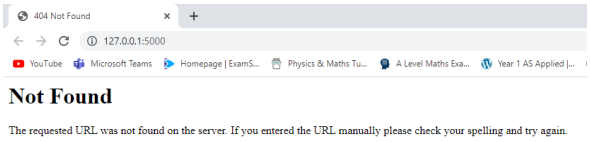
This code creates an instance of a flask application. The python script is named as “__init__” so that python treats the script as a module and so it can be imported to other scripts. This means that the entire “website” folder I have created, will be treated as a python package. I wanted to be able to separate my solution into various python files which all collectively form my solution. This was the idea of thinking procedurally hence why the “__init__” file is pivotal.

The first line shows that I am treating the folder “website” as a python package and importing the function: “create_app” from the other python script in this folder

The if statement here is to ensure that the web app is run only when the python file “main” is run as opposed to the web app running whenever main is imported. This is so that I have control over the web app’s instantiation and I do not unintentionally run the app

```
from website import create_app

app = create_app()

if __name__ == '__main__':
    app.run(debug=True)
```

The “app.run” function is what actually runs the app and, in its parameters, “debug=True” allows the app to update whenever I make a change to the python code.

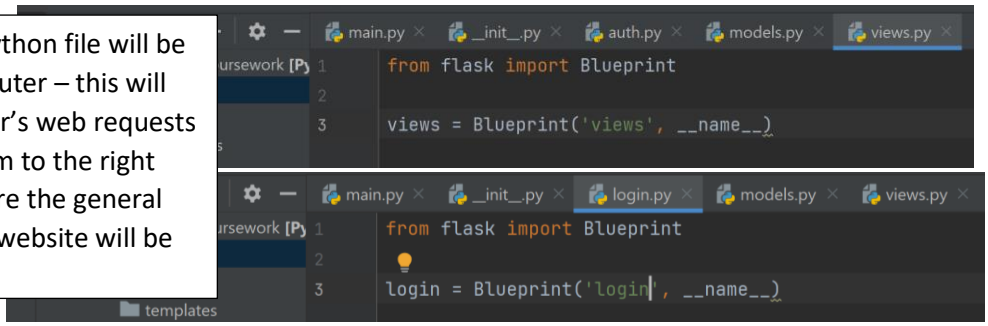


I was directed to this page. This indicated that the app is functional but it also means that no pages or routes belong to this webapp.

Aim: Setup different routes/URLs for the app

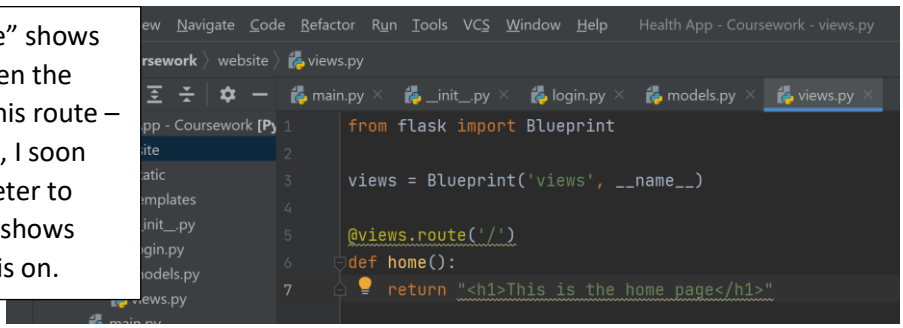
Justification: Creating prototype website routes will form a base/stepping stone that can be used when developing the final webpages, making the whole process easier

The “views” python file will be treated as a router – this will handle the user’s web requests and direct them to the right page; it is where the general routes for the website will be stored.



The blueprint package part of flask allows me to connect multiple different files to the web app. This coincides with my application of decomposition and thinking procedurally, mentioned in the design section. I want to work on each sub-module of my solution in separate files so that the entire project is much easier to handle. I defined the blueprint with the same name as its file name so that it will be easy to implement across multiple other files. I also defined a login blueprint; this is not in views because the login page functions differently to other pages. It is responsible for authenticating users.

The function: “home” shows what will be run when the user goes through this route – enters “/”. However, I soon changed the parameter to “/home” as it easily shows what page the user is on.



The @views.route command defines a route to a webpage and its parameter shows what needs to be typed in to the URL in order to go the corresponding page.

The variable “views” is a blueprint. This is how a blueprint is defined

Test: Add some basic HTML code to see it gets rendered by python.

Justification: Understanding this now sets up the entire process of making webpages with python and so I could prototype many different pages.

I added a basic HTML header. However, this blueprint must be registered before it can become functional.

In the “init” python file, I imported the variables views and login both of which have blueprints assigned to them.

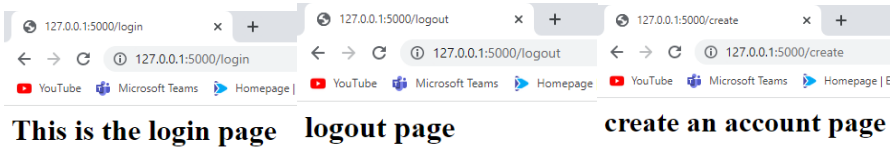
This means they can be used in the current python file

```python
from flask import Flask

def create_app():
    app = Flask(__name__)
    app.config['SECRET_KEY'] = 'carrot24'

    from .views import views
    from .login import login

    app.register_blueprint(views, url_prefix='/')
    app.register_blueprint(login, url_prefix='/')

    return app
```

The “app.register_blueprint” function is what actually registers the blueprints and makes the app recognise them.

For the two blueprints, I left the “url_prefix=/” to indicate that no extra characters need to be entered to access the corresponding route

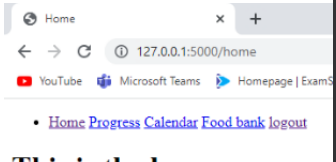
The app successfully adapted to my changes. By entering “/home” in the URL after the host address, I was routed to the home page which was defined as a function in the “views” python folder. I also amended the “login” python file and added a route to see if visiting it would work on the app.



Test: Link a route to a HTML webpage

Justification: To create prototype templates which can easily be edited to form the final products

Very basic HTML code that displays a title




```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8"/>
    <meta name="viewport" content="width=device-width, initial-scale=1"/>

<!-- -->
    <title>Home</title>
</head>
<header class="header">
    <nav>
        <ul class="main-nav">
```

Aim: Create the different routes and HTML pages that are part of my system

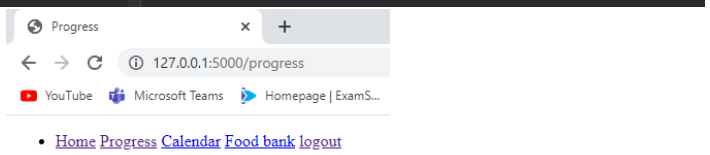
Justification: To start creating prototype templates which will eventually form the final pages.

Render template is a flask module that links HTML files to a route defined in python. Its parameters are the name of the HTML file.

```
from flask import Blueprint, render_template

views = Blueprint('views', __name__)

@views.route('/home')
def home():
    return render_template("home.html")

@views.route('/progress')
def progress():
    return render_template("progress.html")
```

Another aspect of flask that proved to be quite useful was Jinja. Using this allowed me to override the original home template with blocked tags which would be customisable for the progress page, whilst keeping everything else in the home.html file the same.

```
{% extends "home.html" %}

{% block title %}Progress{% endblock %}

{% block h1 %}This is the Progress page{% endblock %}
```



Using the built-in flask module, Jinja was much faster than writing a completely new html file from scratch for the other pages or even copying and pasting code between pages. I rendered more templates for the login page and the create account page, both of which worked just fine.

Aim: Adapt the routes part of the login system to allow and validate user inputs.

Justification: To create different pages that can take user inputs and then check if they meet set requirements – this is a standard feature of all login systems.

This was the structure I had so far in a python file I named as "loginsystem" which contains 3 defined routes.

```
from flask import Blueprint, render_template

loginsystem = Blueprint('loginsystem', __name__)

@loginsystem.route('/login', methods=['GET', 'POST'])
def login():
    return render_template("login.html")

@loginsystem.route('/logout')
def logout():
    return "<h1>logout page</h1>"

@loginsystem.route('/create', methods=['GET', 'POST'])
def create():
    return render_template("create.html")
```

I wanted to make sure that my login route and create route can accept POST requests. By default, the program can accept GET requests but to accept POST requests I had to change the route parameters.

Accepting POST requests was needed because the login and create account pages both need user inputs to work with for both validation and modifying the main database

Flash messages is another flask module that is useful for validation.

I have used them as error messages which will generate when the user does not meet a specific requirement depicted by the if and elif statements

Flask uses a request module which retrieves data entered by a user from a HTML input form box. I have stored these in variables so that it is easy to validate them.

```
@loginsystem.route('/logout')
def logout():
    return "<h1>logout page</h1>"

@loginsystem.route('/create', methods=['GET', 'POST'])
def create():
    if request.method == 'POST':
        username = request.form.get('username')
        password = request.form.get('password')
        if len(username) == 0:
            flash('You have not entered a username', category='error')
        elif not any(i.isupper() for i in password):
            flash('Your password needs at least 1 capital.', category='error')
        elif not any(x .digit() for x in password):
            flash('Your password needs at least 1 number')
```

Python's "len()" function retrieves the number of characters present in a string in its parameters

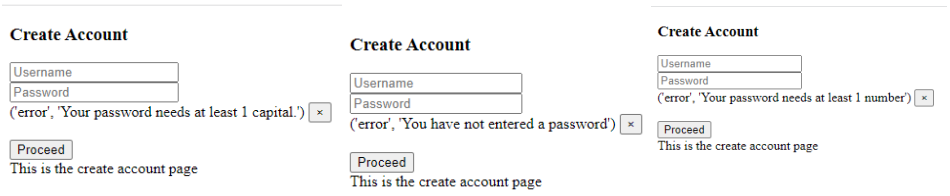
isupper() is a function that checks if there is an uppercase letter present. Combining this with a for loop means each index in a string is checked to see if there is a capital letter or not. I repeated this with the digit() function which checks if a number is present.

Given that my stakeholders wanted a set of requirements for the username and password, the flash message could easily inform the user of errors in what they entered. Such errors can be categorised by flask so they could be easily separated from other messages. In order to present the flash messages, I needed to enter statements in the HTML file. Jinja allows python code to be written in the HTML file so that is why I have written a with statement with an if statement and a for loop nested inside. By using two sets of curly brackets, I was able to refer to a variable according to Jinja syntax.

Test: Check if the create account form returns flashed messages when an error is detected upon the input of invalid data.

Data: admin, (BLANK), Admin

Justification: To confirm if the system can perform validation checks – An integral part to any login system.



The program was successfully able to recognise no capital letters were entered and it generated the right flash message. The app could also generate the other messages.

Aim: Connect the flask app to the MySQL database.

Justification: This is where all the data will be stored and so in order for the solution to be functional, there must be access to a database.

This is flask's own database connector for MySQL and inside the app function, I have configured the details of the database I am using

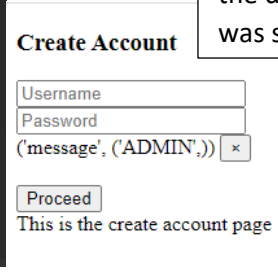
```
from flask import Flask
from flask_mysqldb import MySQL

def create_app():
    app = Flask(__name__)
    app.config['SECRET_KEY'] = 'carrot24'
    app.config['MYSQL_DATABASE_HOST'] = 'localhost'
    app.config['MYSQL_DATABASE_USER'] = 'root'
    app.config['MYSQL_DATABASE_PASSWORD'] = '247890YR'
    app.config['MYSQL_DATABASE_DB'] = 'mydb'
    mysql.init_app(app)
```

Test: Can the create account webpage validate a user that involves accessing the database?

Data: username = ADMIN, password = ADMIN

Justification: To confirm if the system can perform validation checks – An integral part to any login system.

```
@loginsystem.route('/create', methods=['GET', 'POST'])
def create():
    if request.method == 'POST':
        username = request.form.get('username')
        password = request.form.get('password')
        if len(username) == 0:
            flash('You have not entered a username', category='error')
        elif len(password) == 0:
            flash('You have not entered a password', category='error')
        elif not any(i.isupper() for i in str(password)):
            flash('Your password needs at least 1 capital.', category='error')
        elif not any(i.isdigit() for i in str(password)):
            flash('Your password needs at least 1 number', category='error')
        else:
            cursor.execute("SELECT username FROM users")
            flash(cursor.fetchone(), category='message')
```



"ADMIN" is present in the database so this test was successful

For the separate python files, I used mySQL.connector to connect to the database. This meant the separate python files would communicate with database and so they can pass the data to the app python file which recognises the data because it is also connected to the database.

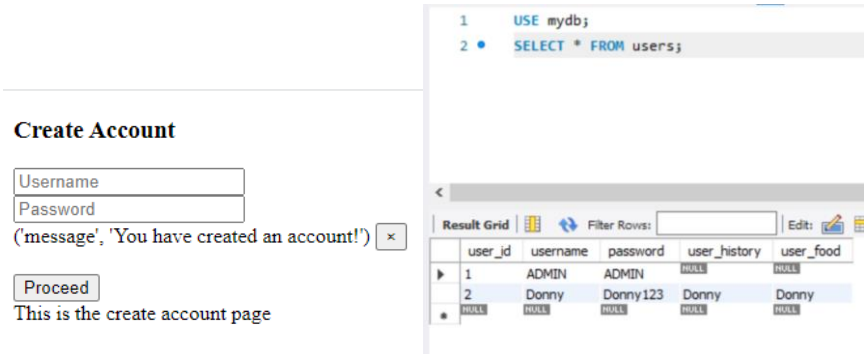
Test: Check if the system can insert values into the database

Data: username = Donny, password = Donny123

Justification: To confirm if the system can add to the database to create new accounts for users.

```
cursor.execute("INSERT INTO users (username, password, user_history, user_food) VALUES (%s, %s, %s, %s)", (username, password, username,
cursor.fetchone()
db.commit()
```

The SQL statement inserts the values of the username and password variables that have been passed into the SQL statement via string formatting - %s



I used MySQL workbench to execute all records in the users table and there I found that the values passed in through python were present – Successful valid test

I wanted the app to direct the user to the home page once they have created an account.

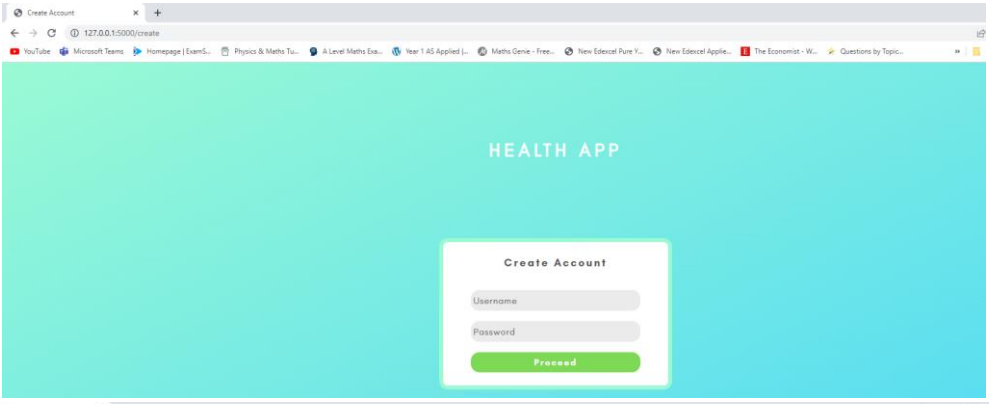
A module from flask allowed me to implement this easily – redirect(url_for()). In its parameters, I linked the home page route

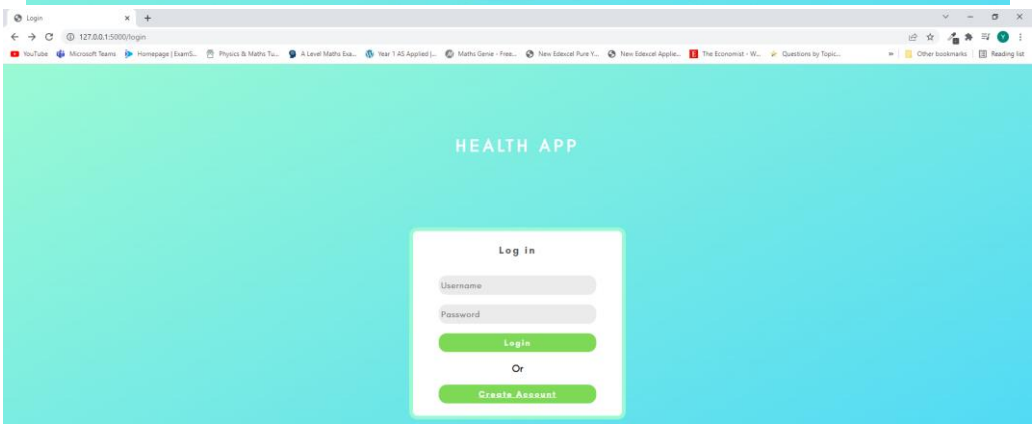
```
    if check:
        flash('That username already exists, please go to the login
    elif len(password) == 0:
        flash('You have not entered a password', category='error')
    elif not any(i.isupper() for i in str(password)):
        flash('Your password needs at least one capital letter.', ca
    elif not any(i.isdigit() for i in str(password)):
        flash('Your password needs at least one number', category='e
    else:
        cursor.execute("INSERT INTO users (username, password, user_
        cursor.fetchone()
        db.commit()
        flash('You have created an account', category='success')
        return redirect(url_for('homepage.home'))

return render_template("create.html")
```

Homepage.home is referring to a route in a python file called “homepage” that contains a function called “home”.

That function renders the HTML webpage for home

At this point I showed my stakeholders what I had so far. They made the request of completing the html and CSS design for the webpages. Justifying that it gives them a better idea of how the whole system looks and functions thus far. Although I had planned to leave the styling of webpages for last, sorting out the aesthetics now would mean I could just copy and paste code and apply classes to different webpages.





Thanks to Jinja templates provided by flask, creating the login page was very easy – everything was inherited from the create account page as I designed them to be similar as part of my usability features in the design section.

Aim: Review the database to see if any modifications need to be made; a change of datatype or adding new columns

Justification: With some fields, the varchar value may be unnecessarily large which takes up more storage space in the database. I am also doing this because from this point onwards, it is likely that I will be adding, querying and deleting records from the database a lot.

Simple SQL statement describes the users table

```
USE mydb;
DESC users;
```

Field	Type	Null	Key	Default	Extra
user_id	int	NO	PRI	NULL	auto_increment
username	varchar(200)	YES		NULL	
password	varchar(200)	YES		NULL	
user_history	varchar(200)	YES		NULL	
user_food	varchar(200)	YES		NULL	

```
USE mydb;
ALTER TABLE users
MODIFY COLUMN username VARCHAR(30);
DESC users;
```

Result Grid | Filter Rows: | Export: | Wrap Cell C

Field	Type	Null	Key	Default	Extra
user_id	int	NO	PRI	NULL	auto_increment
username	varchar(30)	YES		NULL	
password	varchar(200)	YES		NULL	
user_history	varchar(200)	YES		NULL	
user_food	varchar(200)	YES		NULL	

The varchar values of these columns do not need to be 200 – after some research, I found that the average username and password is 11 characters long

I repeated this for the columns; “password”, “user_history” and “user_food” as they all did not need a limit of 200 characters. This meant I also had to change the columns in the other tables that link to “users”.

Field	Type	Null	Key	Default	Extra
username	varchar(30)	NO	PRI	NULL	
goal	varchar(10)	NO		NULL	
num_meals	int	NO		NULL	
chicken_pref	tinyint	NO		NULL	
fish_pref	tinyint	NO		NULL	
redmeat_pref	tinyint	NO		NULL	
vegan_pref	tinyint	NO		NULL	
dairy_pref	tinyint	NO		NULL	
egg_pref	tinyint	NO		NULL	
BMI	float	YES		NULL	
BMR	float	NO		NULL	

I found that this table was not accepting null values. I decided to make sure it does accept null values because different values will be inserted in my tables at different times where the values of other columns may not be decided yet at the time.

I managed to change all columns apart from the primary key, to accept null values – This is not an issue as username will be the first column to obtain a value in this table

Field	Type	Null	Key	Default	Extra
username	varchar(30)	NO	PRI	NULL	
goal	varchar(10)	YES		NULL	
num_meals	int	YES		NULL	
chicken_pref	tinyint	YES		NULL	
fish_pref	tinyint	YES		NULL	
redmeat_pref	tinyint	YES		NULL	
vegan_pref	tinyint	YES		NULL	
dairy_pref	tinyint	YES		NULL	
egg_pref	tinyint	YES		NULL	
BMI	float	YES		NULL	
BMR	tinyint	YES		NULL	

I also had to change a column name in the history table – it was originally named username whereas I designed the table to have a column called user_history

```
1 • USE mydb;
2 • ALTER TABLE history change username user_history VARCHAR(30) NULL;
3 • DESC history;
4
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

Field	Type	Null	Key	Default	Extra
history_id	int	NO	PRI	NULL	auto_increment
user_history	varchar(30)	YES		NULL	
weight	float	YES		NULL	
cal_burned	float	NO		NULL	
date	date	NO		NULL	

Field	Type	Null	Key	Default	Extra
history_id	int	NO	PRI	NULL	auto_increment
user_history	varchar(30)	YES		NULL	
weight	float	YES		NULL	
cal_burned	float	YES		NULL	
date	date	YES		NULL	

I also had to repeat the changes for the food_bank table – in addition to this, I added a new column called "eaten" which is a Boolean/TINYINT datatype. If it is true, the user has eaten the food, false if the user has not

Field	Type	Null	Key	Default	Extra
food_id	int	NO	PRI	NULL	auto_increment
user_food	varchar(30)	YES		NULL	
food	varchar(100)	YES		NULL	
eat_date	date	YES		NULL	
calories	float	YES		NULL	
carbs	float	YES		NULL	
protein	float	YES		NULL	
fat	float	YES		NULL	
sugar	float	YES		NULL	
salt	float	YES		NULL	
chicken	tinyint	YES		NULL	
fish	tinyint	YES		NULL	
redmeat	tinyint	YES		NULL	
vegan	tinyint	YES		NULL	
dairy	tinyint	YES		NULL	
egg	tinyint	YES		NULL	
eaten	tinyint	YES		NULL	

These issues were present because of the number of times I regenerated the database after making modifications. For every time I modified the database, I did not drop the entire database and then re-create one again. Hence, why there are issues with the column's datatypes and names. This meant that I had to add another requirement for the user when they create an).

The statements I added were restraints on the number of characters that can be entered for the username and password via if statement and an ELIF statement – Both cannot be more then 30 as that is the limit on the column in the database

```
if len(username) == 0:
    flash('You have not entered a username', category='error')
if check:
    flash('That username already exists', category='error')
if len(username) > 30:
    flash('Username is longer than 30 characters', category='error')
elif len(password) == 0:
    flash('You have not entered a password', category='error')
elif not any(i.isupper() for i in str(password)):
    flash('Your password needs at least one capital letter.', category='error')
elif not any(i.isdigit() for i in str(password)):
    flash('Your password needs at least one number', category='error')
if len(password) > 30:
    flash('Password is longer than 30 characters', category='error')
else:
```

Aim: Create a functioning login page

Justification: It is one of the key points on my success criteria written in the analysis section, this is a method of separating the data of different people that use the app

```python
@loginsystem.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        username = request.form.get('username')
        password = request.form.get('password')
        cursor.execute("SELECT username FROM users WHERE username COLLATE utf8mb3_bin =%s", (username,))
        exist = cursor.fetchone()
        cursor.execute("SELECT password FROM users WHERE username COLLATE utf8mb3_bin =%s AND password COLLATE utf8mb3_bin =%s
        passcheck = cursor.fetchone()
        if exist:
            if passcheck:
                flash('You are logged in', category='success')
            else:
                flash('Incorrect password', category='error')
        else:
            flash('That username does not exist')

    return render_template("login.html")
```

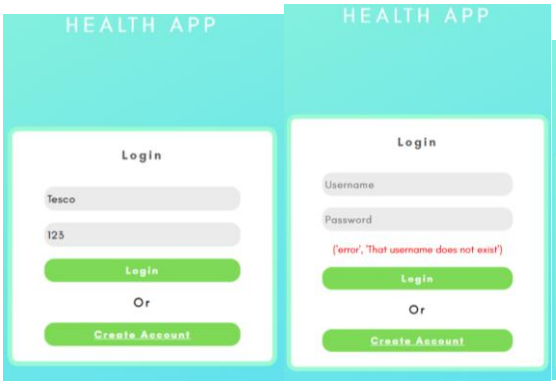
This was very simple to do because I had already created the “create” page and I worked on a draft login at the beginning of this module – so I used both as guidelines to incorporate this into the login function.

I performed a basic test to see if the app could recognise a valid username with a valid password which would flash a message if there is a success or an error.

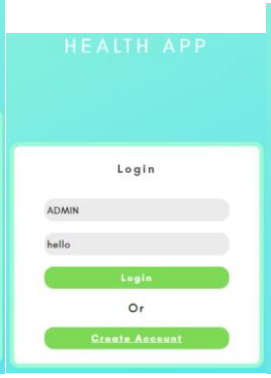
Test: Check if the login page can validate a user that exists in the database.

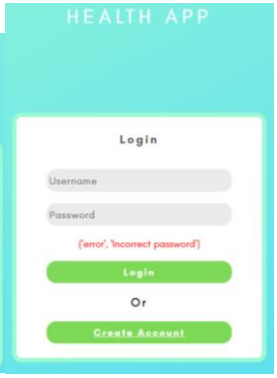
Data: username = Tesco, ADMIN, (BLANK), password = 123, hello, (BLANK)

Justification: To confirm if a user exists in the database – this is how users can be authenticated.



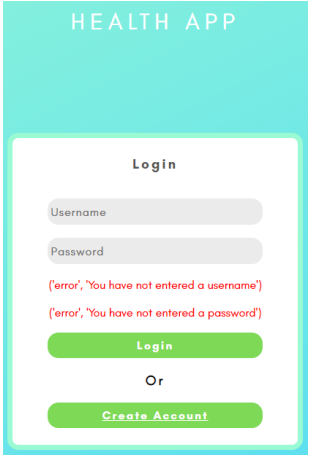
Incorrect username and password.





Correct username, wrong password.

No username or password entered

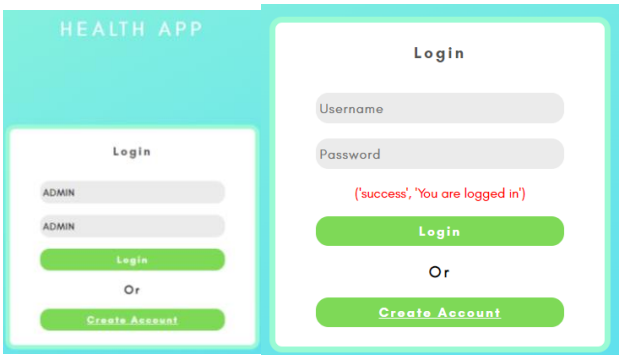


All invalid tests successfully returned the correct response.

ELIF would only run if the other two conditions above it were not met.

```
    exist = cursor.fetchone()
    cursor.execute("SELECT password FROM users WHERE username COLLATE
    passcheck = cursor.fetchone()
    if len(username) == 0:
        flash('You have not entered a username', category='error')
    if len(password) == 0:
        flash('You have not entered a password', category='error')
    elif exist:
        if passcheck:
            flash('You are logged in', category='success')
        else:
            flash('Incorrect password', category='error')
    else:
        flash('That username does not exist', category='error')

return render_template("login.html")
```

When correct details are entered.



This was indeed the expected result of the test and so this works just fine.

Originally, I set the password input field as a HTML password type which does not show the text that the user has entered. My stakeholder decided that using this in the input box could actually confuse users and cause a scenario where the user does not know what they typed as their password which means they cannot access their account. Hence why I adapted to their request and declared the password input field as text.

Aim: Create a login session for the user

Justification: One user must not be allowed to see the data of another user and so that is why they must be able to host their own session on the app and when they logout, their data is

Initially, I planned to use a library called flask_login that comes built-in with python flask however there was an issue with this. After researching, I found that a function known as “userMixin” is used when creating a database in python and it automatically configures an object and its methods needed for the flask login to work. This was not an option for me because I already have created a database, so I thought about creating a class that takes “UserMixin” as its parameter to apply the function to a connected database.

Test: How does python recognise a database connection – as an object?

Data: print(db)

Justification: To see if I could treat the existing database as an object which can have the module UserMixin applied to it – so that the flask login library can be used.

```
import mysql.connector
from flask_login import UserMixin
db = mysql.connector.connect(
    host="localhost",
    user="root",
    passwd="247890YR",
    database="mydb"
)
cursor = db.cursor()

print(db)
```

```
C:\Users\User\AppData\Local\Programs\Python\Python37\python.exe "C:/Users/User/D
<mysql.connector.connection_cext.CMySQLConnection object at 0x000002EDA5323B70>
```

Unfortunately, this test did not offer much insight as python just printed the port at which the connection is made to the database, rather than showing database tables in some sort of way

Having spent a lot of time thinking about how I could resolve this issue, I decided to research more on a function part of flask called flask.session – one source online described this as a much simpler alternative to flask login. This meant I had to remove all the functions that referred to flask login that I added and I had to import sessions from flask.

I added a new SQL statement that will return the user_id correspondent to the username entered, this may be easier to work with for the app because when I was using flask login, the app had trouble with recognising a string input.

```
    exist = cursor.fetchone()
    cursor.execute("SELECT user_id FROM users WHERE username COLLATE u
    id = cursor.fetchone()
    cursor.execute("SELECT password FROM users WHERE username COLLATE
    passcheck = cursor.fetchone()
    if len(username) == 0:
        flash('You have not entered a username', category='error')
    if len(password) == 0:
        flash('You have not entered a password', category='error')
    elif exist:
        if passcheck:
            flash('You are logged in', category='success')
            session["user"] = id
            return redirect(url_for('homepage.home'))
        else:
            flash('Incorrect password', category='error')
    else:
        flash('That username does not exist', category='error')
```

The session variable user, stores the user_id of the user that has just logged in to start their session.

The session pop function removes the user from the session directory and so they have logged out.

The if statement within the home function will check if there is a user in the session directory. If there is, they will be able to access their homepage however if no user has logged in, they will be redirected to the login page

```
@homepage.route('/home')
def home():
    if "user" in session:
        id = session["user"]
        return render_template("home.html")
    else:
        return redirect(url_for('loginsystem.login'))
```

```
@loginsystem.route('/logout')
def logout():
    if "user" in session:
        session.pop("user", None)
        return redirect(url_for('loginsystem.login'))
    else:
        return redirect(url_for('loginsystem.login'))
```

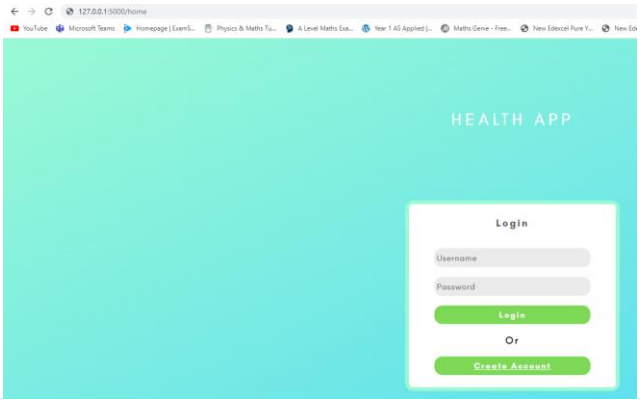
I copied the if statement from the home function and I pasted it in the logout function because they both require a checking method to see where the user should be redirected. The user should not be able to logout if they have not logged in first, so the else statement redirects them to the login page if the session is empty.

Test: Check if the flask module; sessions is functioning correctly

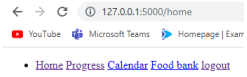
Data: Username = ADMIN, Password = ADMIN

Justification: To see if I could treat the existing database as an object which can have the module UserMixin applied to it – so that the flask login library can be used.

When I typed in "/home" the app generated the login page instead which means it successfully detected that no user was logged in



After I entered ADMIN and ADMIN (correct user details of a user in the database) I was redirected to the home page.



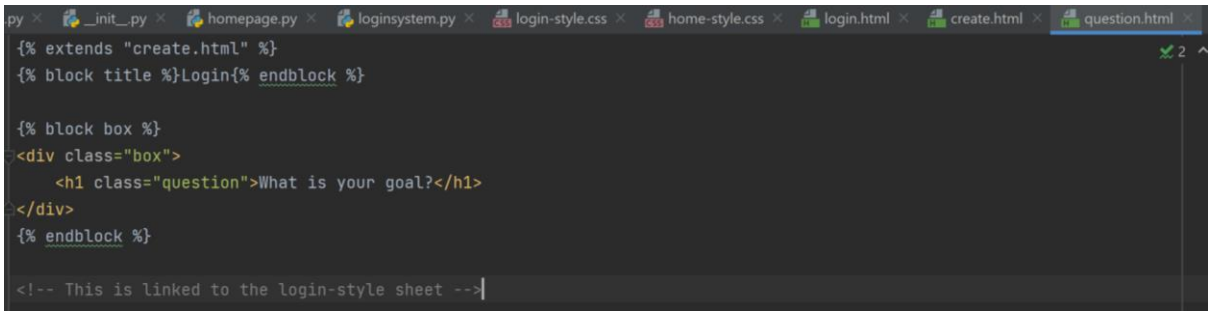
I then proceeded to logout via the logout link at the top of the home page and I was correctly redirected back to the login page however the flashed message was generated when it did not need to be

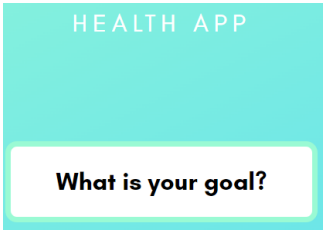
I removed this flashed message by deleting the code in the login function – I left this message as a temporary indicator to check if the app has logged in the user or not but now that I have a working login system, this message is redundant.

The outcome of this small experiment indicated a successful redirection of pages, adding session data and popping session data.

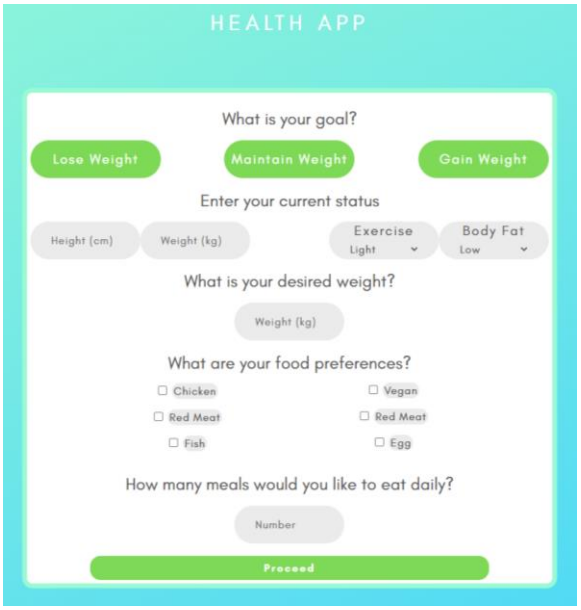
Aim: Complete the questionnaire page

Justification: Check if the session feature can insert data into the database as this is a key point on my success criteria





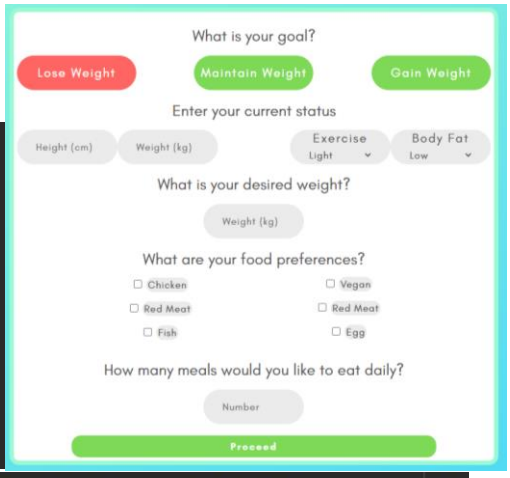
The question.html page was indeed using jinja templates provided by flask, to inherit the html page I created for the create account page. This made the designing of the question page much quicker and easier.



Having shown this prototype to my stakeholders, they said they would like the goal button to change colour to indicate that the user has chosen that option. I managed to implement this with JavaScript embedded in the html file.

I defined a JavaScript function that runs when the user clicks on one of the goal buttons

.getElementById refers to the id assigned in a HTML tag




```
<script language="javascript">
    function ChangeColor()
    {
        document.getElementById("1").style.backgroundColor = '#FF6562';
    }
    function ChangeColor2()
    {
        document.getElementById("2").style.backgroundColor = '#FF6562';
    }
    function ChangeColor3()
    {
        document.getElementById("3").style.backgroundColor = '#FF6562';
    }
</script>
```

```
<button class="goption" style= "float:left" type="submit" onclick="ChangeColor()" value="lose" id="1">Lose Weight</button
<button class="goption" style= "float:centre" type="submit"onclick="ChangeColor2()" value="main" id="2">Maintain Weight</
<button class="goption" style= "float:right" type="submit"onclick="ChangeColor3()" value="gain" id="3">Gain Weight</butto
```

For each button, I had to assign a different function that references their separate id so if the user clicked any of the three buttons, their colour would change

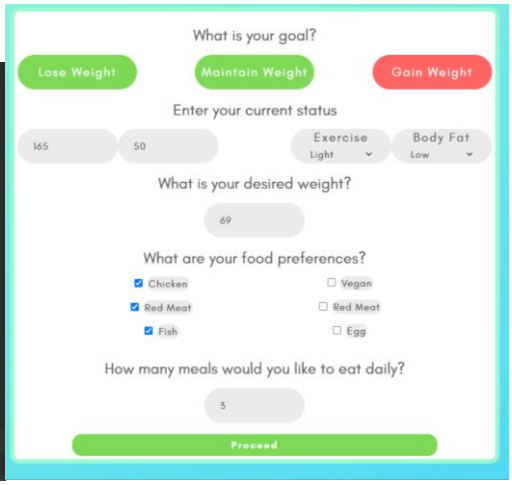
The small issue with the colour-changing buttons is that the user can select multiple buttons – this should not be allowed as the solution should only compensate for one goal. An option to work around this would be to check if the user has selected multiple goals or not and if they have, request that the user changes it.

Test: Check if the questionnaire page can take user inputs and validate them against a set of rules

Data: Lose weight, Maintain weight, Gain weight

Justification: This is part of validating user inputs and is a requirement on my success criteria

```
@homepage.route('/question', methods=['GET', 'POST'])
def question():
    if request.method == 'POST':
        height = float(request.form.get('height'))
        weight = float(request.form.get('weight'))
        exercise = request.form.get('exercise')
        bodyfat = request.form.get('bodyfat')
        dweight = request.form.get('dweight')
        chicken = request.form.get('chicken')
        fish = request.form.get('fish')
        redmeat = request.form.get('redmeat')
        vegan = request.form.get('vegan')
        dairy = request.form.get('dairy')
        egg = request.form.get('egg')
        numbermeals = request.form.get('numbermeals')
        if height < 54.6:
            flash('Invalid height', category='error')
        if height > 272:
```



This was a short experiment with the use of valid testing to see if the page can accept values within the specified range. The way I determined the height range was by searching for the world's tallest and shortest person, this is why the range I specified is ideal. The page was not allowing me to submit all the details I had entered, with the proceed button. I realized that I was missing a method attribute in the form tag within the HTML page. After changing this I found a new error.

```
{% extends "create.html" %}
{% block title %}Questionnaire{% endblock %}

{% block box %}
<div class="qbox">
    <form method="POST">
    <h1 class="question">What is your goal?</h1>
    <button class="qoption" style= "float:left" t
    <button class="qoption" style= "float:centre"
    <button class="qoption" style= "float:right"
```

ValueError

ValueError: could not convert string to float:

Traceback (most recent call last)

```
File "C:\Users\User\AppData\Local\Programs\Python\Python37\lib\site-packages\flask\app.py", line 2091, in __call__
  return self.wsgi_app(environ, start_response)
File "C:\Users\User\AppData\Local\Programs\Python\Python37\lib\site-packages\flask\app.py", line 2076, in wsgi_app
  response = self.handle_exception(e)
File "C:\Users\User\AppData\Local\Programs\Python\Python37\lib\site-packages\flask\app.py", line 2073, in wsgi_app
  response = self.full_dispatch_request()
File "C:\Users\User\AppData\Local\Programs\Python\Python37\lib\site-packages\flask\app.py", line 1518, in full_dispatch_request
  rv = self.handle_user_exception(e)
File "C:\Users\User\AppData\Local\Programs\Python\Python37\lib\site-packages\flask\app.py", line 1516, in full_dispatch_request
  rv = self.dispatch_request()
File "C:\Users\User\AppData\Local\Programs\Python\Python37\lib\site-packages\flask\app.py", line 1502, in dispatch_request
  return self.ensure_sync(self.view_functions[rule.endpoint])(**req.view_args)
File "C:\Users\User\Desktop\Health App\Health App - Coursework\website\homepage.py", line 21, in question
  height = float(request.form.get('height'))
```

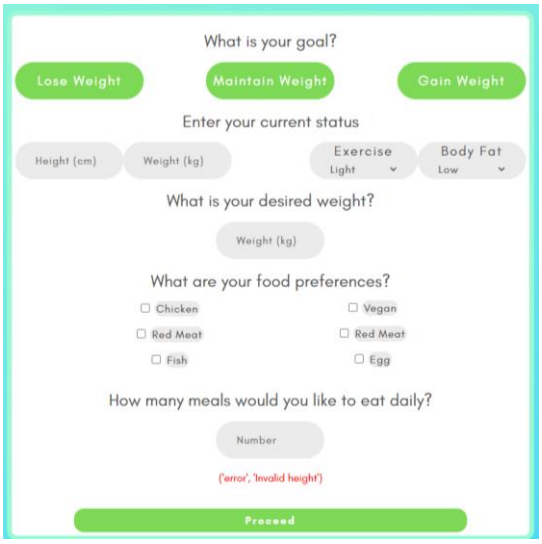
ValueError: could not convert string to float:

I tried declaring height as a float by creating a variable h that stores the user's input value but this resulted in the same error being generated

```
@homepage.route('/question', methods=['GET', 'POST'])
def question():
    if request.method == 'POST':
        h = request.form.get('height')
        height = float(h)
        weight = request.form.get('weight')
        exercise = request.form.get('exercise')
        bodyfat = request.form.get('bodyfat')
        dweight = request.form.get('dweight')
        chicken = request.form.get('chicken')
        fish = request.form.get('fish')
        redmeat = request.form.get('redmeat')
        vegan = request.form.get('vegan')
        dairy = request.form.get('dairy')
        egg = request.form.get('egg')
        numbermeals = request.form.get('numbermeals')
        if height < 54.6:
            flash('Invalid height', category='error')
```

```
@homepage.route('/question', methods=['GET', 'POST'])
def question():
    if request.method == 'POST':
        height = request.form.get('height', type=int)
        if height is None:
            height = 0
```

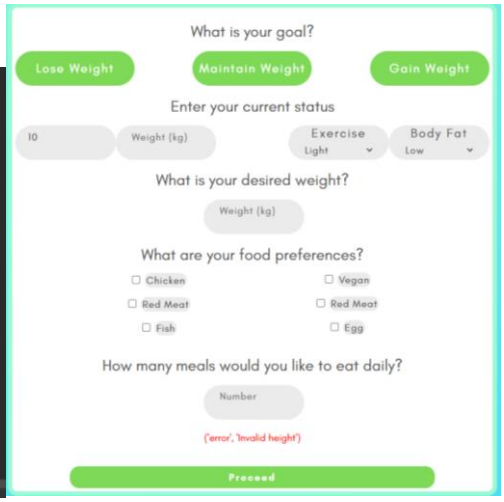
In the request form function, I specified the data type that should be inputted by the user and so the program could recognise that the height input should be an integer and not a string.



This prevented the previous errors however I was no longer able to enter in a value in any of the forms – indicating that the form is automatically being submitted to the python file. I examined the HTML file for this page and I found that in each option box, I wrote the tag button which meant as long as the user clicks on the box, its value will be submitted (even if it is empty). This is why I could not input anything into the boxes, the button was passing blank values to python. I replaced the button tags with div tags that use the same class embedded so it would look exactly the same.

For almost all visible grey buttons, I had used the button tag. I changed this to div which solved the issue I had before

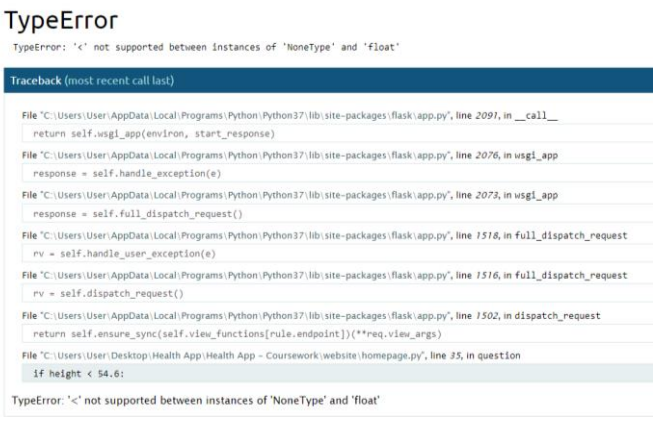
```
<h1 class="question">Enter your current status</h1>
<div class="qinput" style= "float:left">
    <label for="height"></label>
    <input class="type"
           type="number"
           id="height"
           name="height"
           placeholder=" Height (cm)"/></div>

<div class="qinput" style= "float:left">
    <label for="weight"></label>
    <input class="type"
           type="number"
           id="weight"
           name="weight"
           placeholder=" Weight (kg)"/></div>

<div class="qinput" style= "float: right">
```



Although I could now input values just fine, I still could not submit the form by clicking the proceed button.

Reviewing the HTML file led me to find 3 form tags being used. In order for the proceed button to function as a way of submitting the user's inputs, I needed to have only one form tag in the entire file. After removing the excess form tags, the proceed button worked successfully. The issue of automatically submitting inputs was resolved however if the user were to accidentally click proceed while leaving a form empty this error would appear:

When the user submits the form without entering a value in an input box, a “none” response is returned and so the app did not know how to handle this response.



The not part of the if statement refers to an empty form submitted by the user, hence the user has not posted any data, the is no data to get for python

```
fish = request.form.get('fish')
redmeat = request.form.get('redmeat')
vegan = request.form.get('vegan')
dairy = request.form.get('dairy')
egg = request.form.get('egg')
numbermeals = request.form.get('numbermeals')
if not request.form.get('height'):
    flash('Invalid height', category='error')
```

('error', 'Invalid height')

Proceed

This solved the issue and correctly produced an error message after validating that the form was empty. Now the task was to incorporate the other conditions I set for the height input before.

I found that writing elif statements worked successfully whereas if statements would not work because python would try an use the same blank value from the first if statements to then compare > or <

```
    fish = request.form.get('fish')
    redmeat = request.form.get('redmeat')
    vegan = request.form.get('vegan')
    dairy = request.form.get('dairy')
    egg = request.form.get('egg')
    numbermeals = request.form.get('numbermeals')
    if not request.form.get('height'):
        flash('Invalid height', category='error')
    elif height < 54.6:
        flash('Invalid height', category='error')
    elif height > 272:
        flash('Invalid height', category='error')
    #if weight < 35:
        #flash('Invalid weight', category='error')
    #if weight > 140:
        #flash('Invalid weight', category='error')
return render_template("question.html")
```

I declared the if statements on the weight variable as comments so that I could first test to see if the height variable can be handled properly first

The body fat level and exercise level were simple inputs that can be manipulated as both can only take three values, no empty value. As for the desired weight (dweight) input and the number of meals (numbermeals) input, they would both follow the same structure as the weight and height inputs I already set. To handle the checkboxes, I declared the type of the variables as Boolean values.

('error', 'chicken is preferred')

('error', 'Invalid desired weight')

('error', 'Invalid number of meals')

```
if chicken == True:
    flash('chicken is preferred', category='error')
```

Proceed

This was a short experiment to see if a checked box on the chicken variable would actually be processed as true and the result of the test proved that this was the case.

Leaving the checkbox empty generated the correct message – the final solution will not show a message for the checkbox, I aim doing this for testing purposes only

('error', 'NO CHICKEN')

('error', 'Invalid desired weight')

('error', 'Invalid number of meals')

Proceed

As shown here, the other errors were generated successfully so the validation of those inputs are correct.

This error was a test to see if python could accept the different option values, I listed for exercise level

('error', 'LIGHT')

```
if exercise == "light":
    flash('LIGHT', category='error')
```

At this point I went back to my design section to see if I missed out on any particular features that need to be included. I needed to add questions for the user's gender, age as these are factors that can affect the user's BMR/target calories. I added a gender question by copying and pasting the code I already wrote for the preferences question.

The first check is to see if both boxes are empty, if they are, an error will be shown

```
if not request.form.get('male'):
    if not request.form.get('female'):
        flash('You have not selected a gender', category='error')
elif male == True:
    if female == True:
        flash('You have selected two genders', category='error')
    else:
        flash('MALE', category='error')
elif female == True:
    flash('FEMALE', category='error')
```

Python checks if the male box is ticked, if it is it will check if the female box is ticked. If both are ticked then it will flash an error otherwise it recognises that only male has been ticked and so the user is a male

What is your gender?

Male

Female

('error', 'You have selected two genders')

What is your gender?

Male

Female

('error', 'You have not selected a gender')

What is your gender?

Male

Female

('error', 'MALE')

All outcomes apart from selecting female only were correctly generated but after adding an else statement to the python code, this was resolved.

What is your gender?

Male

Female

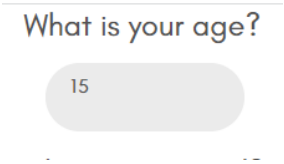
('error', 'FEMALE')

I repeated this process by adding a new question asking the user for their age.

This checks if the user has entered something and whether it falls within the specified range

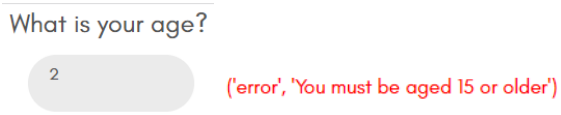
```
if not request.form.get('age'):
    flash('Invalid age', category='error')
elif age < 14:
    flash('You must be aged 15 or older', category='error')
elif age > 75:
    flash('You must be aged 75 or younger', category='error')
```

Valid data test however the input value is on the boundary of what should be accepted and what should not



This produced no errors – correct response

An example of invalid testing, this input value should not be accepted by the app



Produced an error that indicates the input value was too low – correct response

Using the current details, the user's BMR could be calculated. This is the main component of the user's goal and so this is why calculating BMR should be done before checking the user's goal input. As the value of the BMR depends on the gender of the user, I had to move the code I wrote for checking the user's gender, at the very bottom of the question page function in python. Using the formula that I wrote in the design section; I wrote two calculations of which include a constant named "k" – a recognised symbol for showing a constant.

```
    elif not request.form.get('male'):
        if not request.form.get('female'):
            flash('You have not selected a gender', category='error')
        else:
            k = -161
            BMR = (10 * weight) + (6.25 * height) - (5 * age) + k
    elif male == True:
        if female == True:
            flash('You have selected two genders', category='error')
        else:
            k = 5
            BMR = (10 * weight) + (6.25 * height) - (5 * age) + k

return render_template("question.html")
```

After the BMR has been calculated, I realized that the user's goal had to be handled by python. The best way to handle this would be to write a function outside of the question page code so that it can be called after the BMR calculation IN the question page. This is so that I do not need to repeat the user's goal validation for both genders, instead they can just call the function.

```
def goal(lose, main, gain):
    if lose == "lose":
        flash('Lose Weight selected')
```

```
elif not request.form.get('male'):
    if not request.form.get('female'):
        flash('You have not selected a gender', category='error')
    else:
        k = -161
        BMR = (10 * weight) + (6.25 * height) - (5 * age) + k
        goal(lose, main, gain)
elif male == True:
    if female == True:
        flash('You have selected two genders', category='error')
    else:
        k = 5
        BMR = (10 * weight) + (6.25 * height) - (5 * age) + k
        goal(lose, main, gain)
```

However, once I entered valid details in the question page, there was no message to say that I selected lose weight. Since handling Boolean values was very straightforward, I decided to try and

implement this when the user clicks on a goal button option – in the form of a checkbox. This is because they take up a true or false state by standard.

```
<h1 class="question">What is your goal?</h1>
<div style="align: center;">
    <input type="checkbox" value="false" name="lose" id="lose"/>
    <label class="option" for="lose">Lose Weight</label>

    <input type="checkbox" value="false" name="main" id="main"/>
    <label class="option" for="main">Maintain Weight</label>

    <input type="checkbox" value="false" name="gain" id="gain"/>
    <label class="option" for="gain">Gain Weight</label>
</div>
```

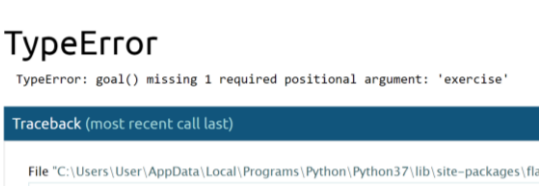
```
input[type=checkbox] {
    display: none;
}

input:checked + label {

    color: white;
    background-color: #FF6562
}
```

Converting the button options to checkboxes was simple because I already created checkboxes for the user's preferences. With simple CSS code, I was able to declare the state of the checkboxes before and after the user clicks on it.

The set of if statements check what options the user has selected

Unexpectedly, the CSS code applied to all checkboxes I had created on the HTML page. I was contemplating whether to remove the colour change on the other checkboxes but after speaking with my stakeholder, they suggested to leave this feature like this.

```
def goal(lose, main, gain, exercise):
    if lose:
        if main:
            return flash('Select one goal only',category='error')
        elif gain:
            return flash('Select one goal only', category='error')
        else:
            return flash('LOSEEE', category='error')
```



Error generated because I did not use the exercise variable in the goal function

This was a short test to see if the app could return the response "LOSEEE" after checking if I have selected only the lose weight option on the webpage. The exercise variable is also a parameter of this function. This is because the exercise level selected by the user will have an affect on their BMR – this is what the goal function is going to calculate.

This error message was produced correctly – the app was able to check if the user has selected more than one option for their goal.

This test result was only generated after I removed the exercise variable from the goal parameter. This means the goal checkboxes can be validated but the exercise variable was not.

```
def goal(lose, main, gain):
    if lose:
        if main:
            return flash('Select one goal only', category='error')
        elif gain:
            return flash('Select one goal only', category='error')
        else:
            return flash('LOSE', category='error')
    elif main:
        if lose:
            return flash('Select one goal only', category='error')
        elif gain:
            return flash('Select one goal only', category='error')
        else:
            return flash('MAINTAIN', category='error')
    elif gain:
        if lose:
            return flash('Select one goal only', category='error')
        elif main:
            return flash('Select one goal only', category='error')
        else:
            return flash('GAIN', category='error')
    else:
        return flash('You have not selected a goal', category='error')
```

This sequence of if statements should cover all the possible scenarios of which options have been selected by the user. If the user has not selected more than one goal, their selected option should be flashed as an error message

Test: Select various combinations when selecting the goal options

Data: Lose weight, Maintain weight, Gain weight

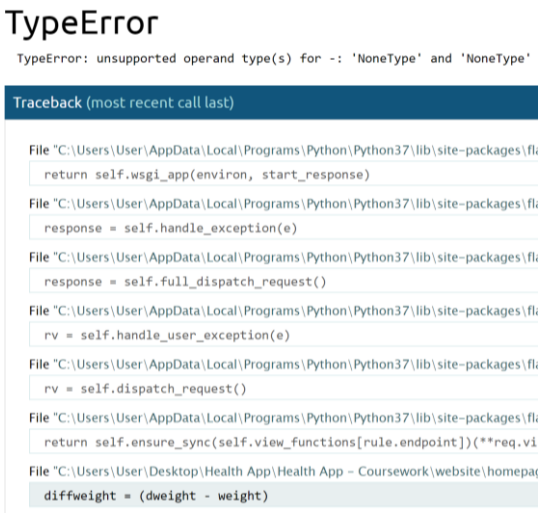
Justification: This is part of validating user inputs. The user can only have one goal at a time and so the app must not accept the user choosing multiple goals

All three messages were correctly shown, based on the user's selected options

At this point I realized that the desired weight entered by the user also has to be checked by the app to see if it matches their selected goal option. This is to add to the robustness of my solution and ensure it is not easily breakable. For example, if the user has selected "lose weight" but they set their desired weight as higher than their current weight, the app should respond to this by flashing an error message.

```
height = request.form.get('height', type=float)
weight = request.form.get('weight', type=float)
exercise = request.form.get('exercise')
bodyfat = request.form.get('bodyfat')
dweight = request.form.get('dweight', type=float)
chicken = request.form.get('chicken', type=bool)
fish = request.form.get('fish')
redmeat = request.form.get('redmeat')
vegan = request.form.get('vegan')
dairy = request.form.get('dairy')
egg = request.form.get('egg')
numbermeals = request.form.get('numbermeals', type=int)
diffweight = dweight - weight
```

I created a variable called diffweight which symbolizes the difference between the desired weight and the current weight entered by the user. I could perform the needed validation checks by writing if statements on the desired weight.



This error showed that there was an issue subtracting the two variables. Instead of continuing with the variable diffweight, I used the two variables: dweight and weight for my comparison instead.

If the user wants to lose weight, their desired weight has to be less than their current weight.

If they want to maintain their weight, their desired weight has to be equal to their current weight.

If they want to gain weight, their desired weight has to be greater than their current weight.

```
def goal(lose, main, gain, weight, dweight):
    if lose:
        if main:
            return flash('Select one goal only', category='error')
        elif gain:
            return flash('Select one goal only', category='error')
        elif dweight >= weight:
            return flash('Your desired weight does not match your goal', category='error')
        else:
            return flash('LOSE', category='error')
    elif main:
        if lose:
            return flash('Select one goal only', category='error')
        elif gain:
            return flash('Select one goal only', category='error')
        elif dweight != weight:
            return flash('Your desired weight does not match your goal', category='error')
        else:
            return flash('MAINTAIN', category='error')
```

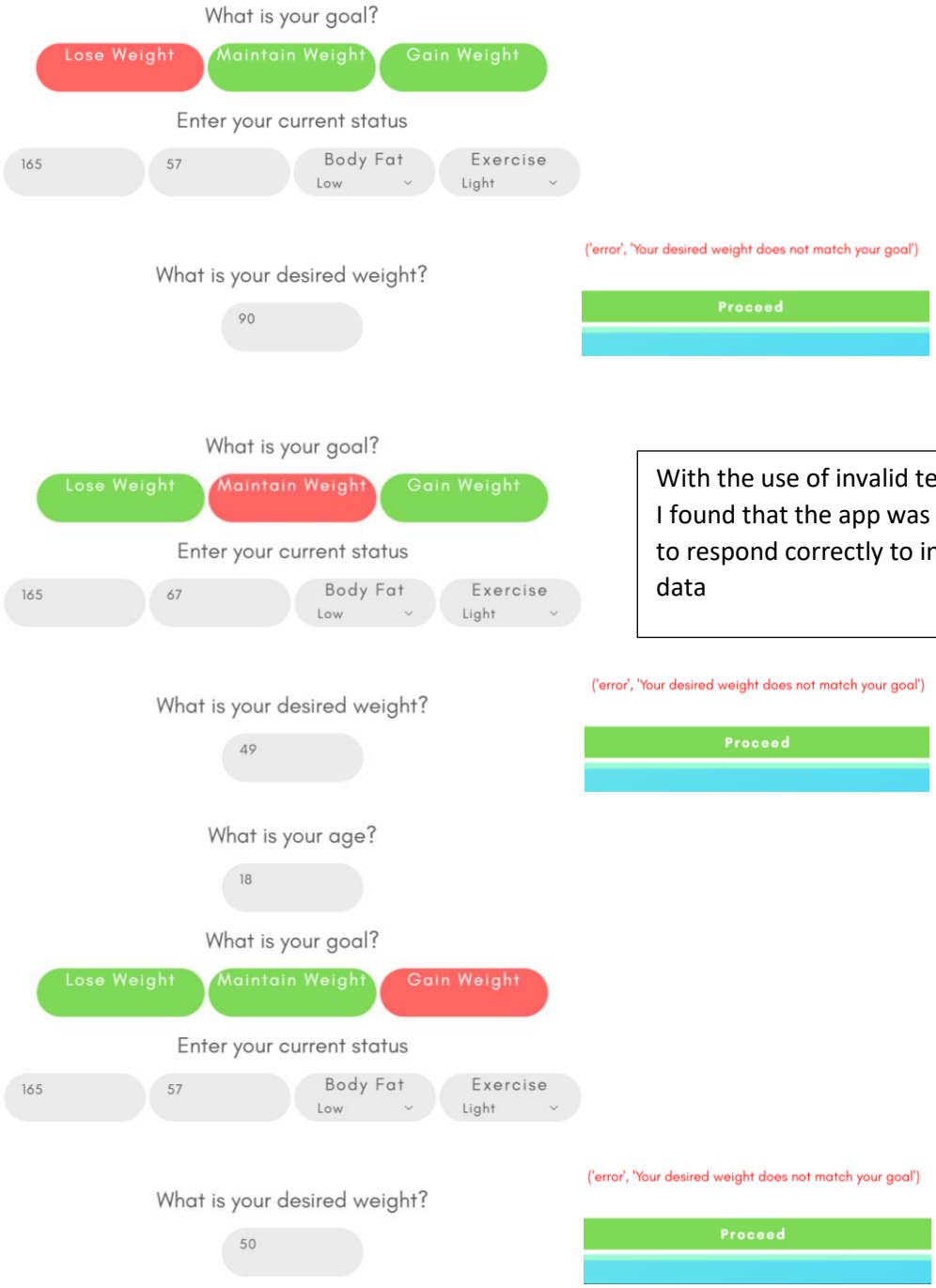
```
elif gain:
    if lose:
        return flash('Select one goal only', category='error')
    elif main:
        return flash('Select one goal only', category='error')
    elif dweight <= weight:
        return flash('Your desired weight does not match your goal', category='error')
    else:
        return flash('GAIN', category='error')
else:
    return flash('You have not selected a goal', category='error')
```

If the app can validate that the user has not entered an invalid weight, the app should flash an error to show which goal option they selected – this is for testing purposes only, final solution will not generate an error

Test: Check if the app inspects the desired weight entered by the user to see if it matches their goal

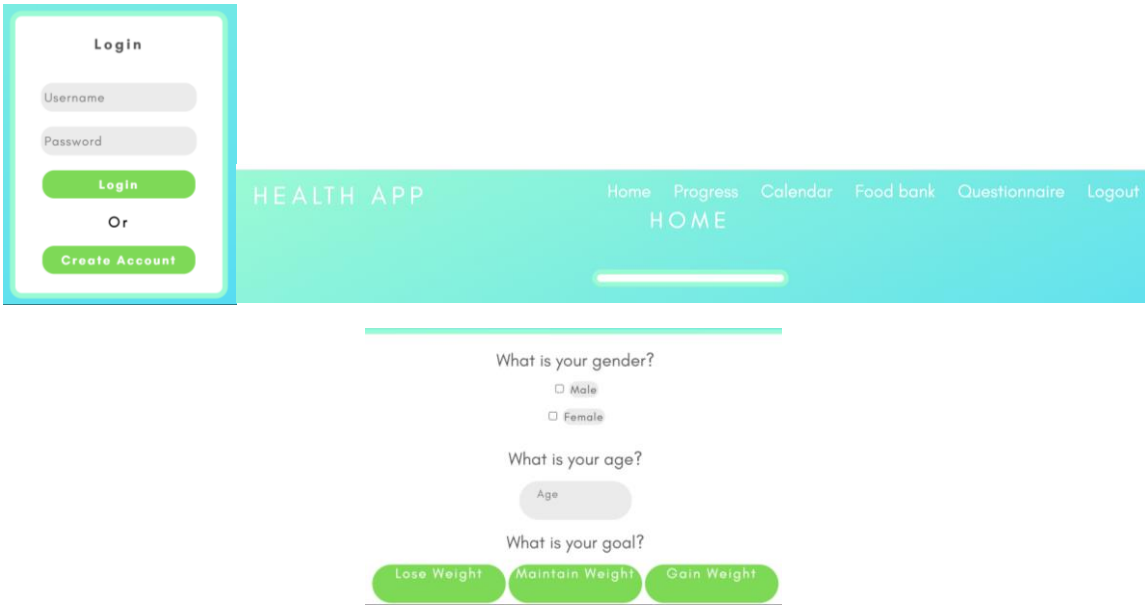
Data: Lose weight, Maintain weight, Gain weight, 90, 49, 50

Justification: This is part of validating user inputs. If the user selects “lose weight” they cannot enter their desired weight to be higher than their current weight etc.

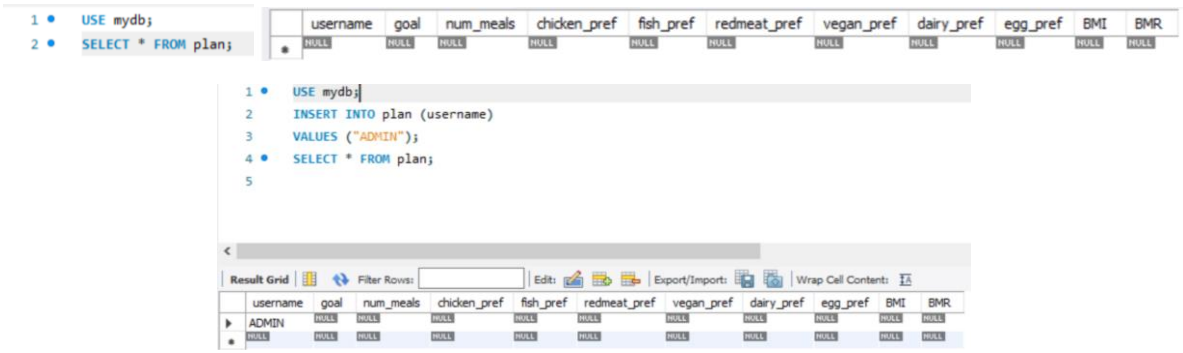


With the use of invalid testing, I found that the app was able to respond correctly to invalid data

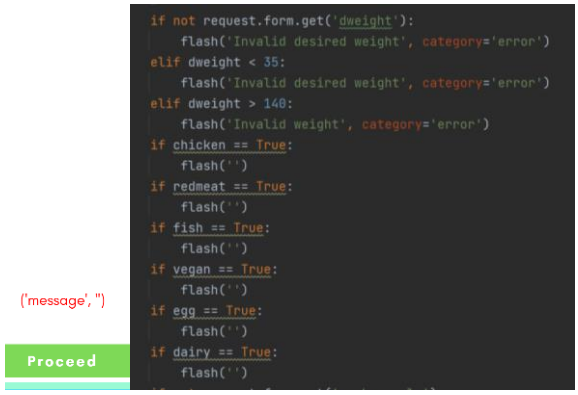
The next stage was to insert the data collected from the inputs on the page, into the database. I initially added the session statements so that there has to be a user logged in already for them to use the question page. I also changed the session dictionary by inserting the username of the user instead of the user_id. This is because when inserting values into other tables, they will not all have the user_id column in their tables and instead will have the username. So if the session dictionary stores the username it will be much easier to query the database. I also decided to add a link to the question page in the home page on the navigation bar. This will make it easier for the user to access the question page.



Before completing the questionnaire, I decided to check the plan table to see if ADMIN was in its username column.



I went back to the question page and entered values that should be accepted by the system however I received this unexpected message.



This was unusual as the only flash messages I listed were error messages. I looked through my python code and I found that I had not yet specified what should happen if the user selects different food preferences. Every time the user completes the questionnaire, the values in the plan table must be updated. To test if I could do this, I executed an UPDATE SQL command in the MySQL workbench.

```
USE mydb;
UPDATE plan SET goal = "lose" WHERE username = "ADMIN";
SELECT * FROM plan;
```

username	goal	num_meals	chicken_pref	fish_pref	redmeat_pref	vegan_pref	dairy_pref	egg_pref	BMI	BMR
ADMIN	lose	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

This test meant that I correctly followed the syntax of the SQL statement to be executed and so I could implement this in the python code.

This section would run the valid function and store the returned value in a variable named goal.

```
goal = valid(lose, main, gain, weight, dweight)
cursor.execute("UPDATE plan SET goal = %s WHERE username = %s",(goal, sessionid,))
cursor.fetchone()
db.commit()
return redirect(url_for('homepage.home'))
```

The SQL statement should update the plan table according to the goal value

Before I ran this code, I wanted to fix the issue with the food preferences. The app should update the database according to the user's selected options, much like the goal being updated in the database.

```
if chicken == True:
    cursor.execute("UPDATE plan SET chicken_pref = 1 WHERE username = %s", (sessionid,))
    cursor.fetchone()
    db.commit()
if redmeat == True:
    cursor.execute("UPDATE plan SET redmeat_pref = 1 WHERE username = %s", (sessionid,))
    cursor.fetchone()
    db.commit()
if fish == True:
    cursor.execute("UPDATE plan SET fish_pref = 1 WHERE username = %s", (sessionid,))
    cursor.fetchone()
    db.commit()
if vegan == True:
    cursor.execute("UPDATE plan SET vegan_pref = 1 WHERE username = %s", (sessionid,))
    cursor.fetchone()
    db.commit()
if egg == True:
    cursor.execute("UPDATE plan SET egg_pref = 1 WHERE username = %s", (sessionid,))
    cursor.fetchone()
    db.commit()
if dairy == True:
    cursor.execute("UPDATE plan SET dairy_pref = 1 WHERE username = %s", (sessionid,))
```

mysql.connector.errors.ProgrammingError

```
mysql.connector.errors.ProgrammingError: Failed processing format-parameters; Python 'tuple' cannot be converted to a MySQL type

Traceback (most recent call last)

File "C:\Users\User\AppData\Local\Programs\Python\Python37\lib\site-packages\mysql\connector\conversion.py", line 181, in to_mysql
  return getattr(self, "_{0}_to_mysql".format(type_name))(value)

During handling of the above exception, another exception occurred:

File "C:\Users\User\AppData\Local\Programs\Python\Python37\lib\site-packages\mysql\connector\cursor.py", line 431, in _process_params
  res = [to_mysql(i) for i in res]
File "C:\Users\User\AppData\Local\Programs\Python\Python37\lib\site-packages\mysql\connector\cursor.py", line 431, in <listcomp>
  res = [to_mysql(i) for i in res]
File "C:\Users\User\AppData\Local\Programs\Python\Python37\lib\site-packages\mysql\connector\conversion.py", line 186, in to_mysql
  "MySQL type".format(type_name))

During handling of the above exception, another exception occurred:

File "C:\Users\User\AppData\Local\Programs\Python\Python37\lib\site-packages\flask\app.py", line 2091, in __call__
  return self.wsgi_app(environ, start_response)
File "C:\Users\User\AppData\Local\Programs\Python\Python37\lib\site-packages\flask\app.py", line 2076, in wsgi_app
```

This showed that there was an issue with the way I was using sessionid as a variable that can be called into a statement, specifically a tuple that was being used.

Test: Write different SQL statement by using different formatting

Data: 1, "ADMIN" – Valid data

Justification: I needed to find a way to use values in lists or tuples that will be used when executing SQL statements so that I can easily use the value stored in the session variable.

```
from flask import Blueprint, render_template, request, flash, redirect, u
import mysql.connector

db = mysql.connector.connect(
    host="localhost",
    user="root",
    passwd="247890YR",
    database="mydb"
)
cursor = db.cursor()

cursor.execute("SELECT * FROM plan")
print(cursor.fetchone())

cursor.execute("SELECT username FROM plan WHERE username =%s",("ADMIN",))
print(str(cursor.fetchone()[0]))
```

```
('ADMIN', 'lose', None, None, None, None, None, None, None, None)
ADMIN
```

The result of this test showed that when an SQL statement is executed in python, it is returned in the form of a tuple. This proved to be an interesting result and so for further experiments I referenced an index when executing the SQL statement in python.

```
cursor.execute("SELECT * FROM plan")
print(cursor.fetchone()[1])

cursor.execute("SELECT username FROM plan WHERE username =%s",("ADMIN",))
print(str(cursor.fetchone()[0]))
```

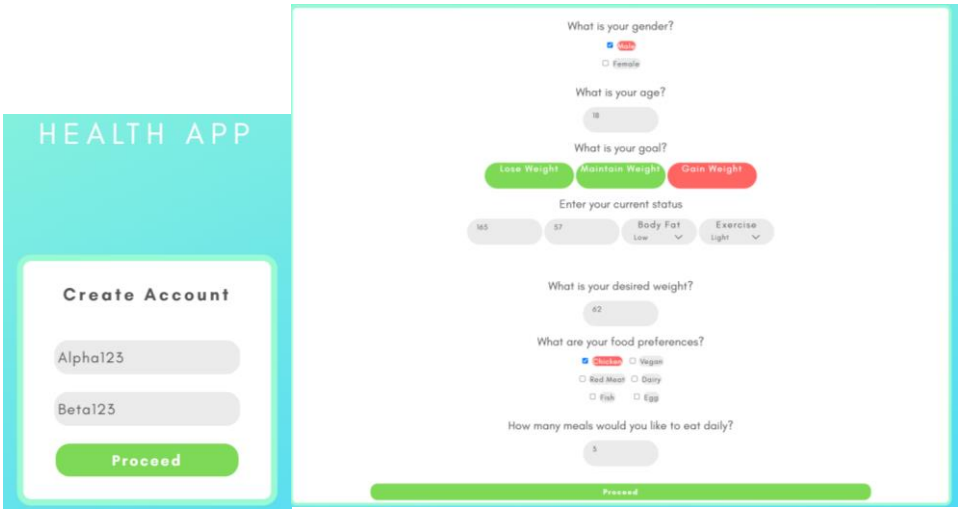
```
lose
ADMIN
```

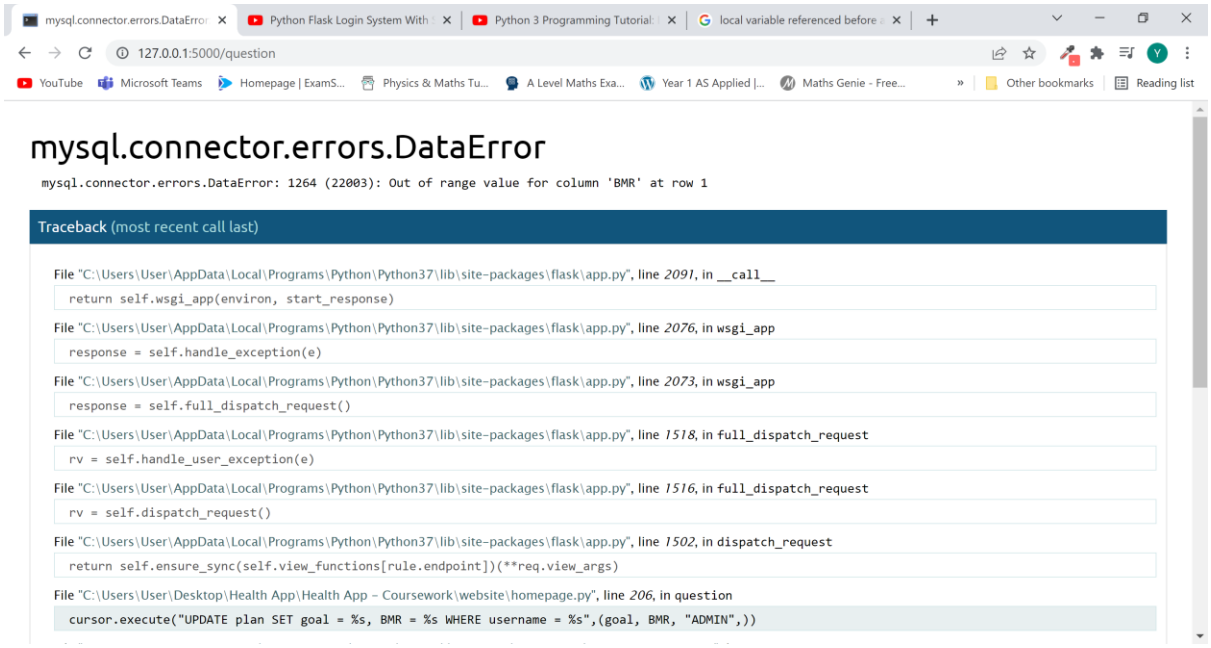
I found that when an index is specified, a tuple is not returned and instead the value held in the database with its original data type is returned. This potentially was the problem I had when trying to execute SQL statements by using the flask session module.

```
if chicken == True:
    cursor.execute("UPDATE plan SET chicken_pref = 1 WHERE username = %s", (sessionid[0],))
    cursor.fetchall()
    db.commit()
```

```
('ADMIN', 'lose', None, 1, None, None, None, None, None, None, None)
```

I added an index when trying to string format the session variable and I found that this worked successfully. From here, I attempted to create a new account and then complete the questionnaire to see if the app could adapt to the new user without me specifying a different index such as sessionid[1].





This error was generated when I tried to complete the questionnaire. This did not make sense to me because the code worked in a separate python file which meant that there was an issue within the questionnaire python file, especially the various conditions I wrote as if statements.

```
def constant(k, male, female):
    if not request.form.get('male'):
        if not request.form.get('female'):
            error2 = flash('You have not selected a gender', category='error')
            return error2
        else:
            k = -161
            return k
    elif male:
        if female:
            error2 = flash('You have selected two genders', category='error')
            return error2
        else:
            k = 5
            return k
```

I moved the various if statements that validate the user's gender to a function that will be called

```
        flash('Invalid number of meals', category='error')
        return render_template("question.html")
    elif numbermeals < 3:
        flash('The minimum number of meals should be 3', category='error')
    elif numbermeals > 7:
        flash('The number of meals exceeds 7', category='error')
    k = 0
    BMR = 0
    constant(k, male, female)
    BMR = target(weight, height, age, k, exercise, BMR)
    goal = valid(lose, main, gain, weight, dweight)
    cursor.execute("UPDATE plan SET goal = %s, BMR = %s WHERE username = %s",(goal, BMR, "ADMIN",))
    cursor.fetchone()
    db.commit()

    return redirect(url_for('homepage.home'))

return render_template("question.html")
```

This data type refers to using 4 digits and then 2 digits for the decimal places

```
USE mydb;
ALTER TABLE plan
MODIFY COLUMN BMR DECIMAL(6, 2);
DESC plan;
```

Result Grid | Filter Rows: | Export: | Wrap

Field	Type	Null	Key	Default	Extra
chicken_pref	tinyint	YES		NULL	
fish_pref	tinyint	YES		NULL	
redmeat_pref	tinyint	YES		NULL	
vegan_pref	tinyint	YES		NULL	
dairy_pref	tinyint	YES		NULL	
egg_pref	tinyint	YES		NULL	
BMI	float	YES		NULL	
BMR	decimal(6,2)	YES		NULL	

Before I ran the program, I realized that the BMR column in my table was a float data type that could not store enough digits for the BMR that will be calculated. Once I did this, I entered data into the question page.

Test: Check if data has been inserted into the plan database

Data: SELECT * FROM plan

Justification: The questionnaire page must be able to add values into the table as this is part of my success criteria.

```
USE mydb;
SELECT * FROM plan;
```

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

username	goal	num_meals	chicken_pref	fish_pref	redmeat_pref	vegan_pref	dairy_pref	egg_pref	BMI	BMR
ADMIN	gain	NULL	0	NULL	0	NULL	NULL	NULL	NULL	1825.50
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

I ran this SQL statement and it was successful. To solve the issue of running conditions on the user's inputs, I completely re-arranged python code in the questionnaire page and I moved it to a separate python file so that I could remain organized.

main.py | __init__.py | loginsystem.py | homepage.py | questionnaire.py | question.html | home.html

These are three functions that will be called later on. They take parameters from the value that the user inputs on the html

```
def checkgoal(lose, main, gain, weight, dweight):
    if lose:
        if dweight >= weight:
            return 1
        else:
            return "lose"
    elif main:
        if dweight != weight:
            return 1
        else:
            return "main"
    elif gain:
        if dweight <= weight:
            return 1
        else:
            return "gain"
```

```
def constant(male, female):
    if female:
        k = -161
        return k
    if male:
        k = 5
        return k

def target(weight, height, age, k, exercise):
    BMR = (10 * weight) + (6.25 * height) - (5 * age) + k
    if exercise == "light":
        BMR = BMR * 1.2
        return BMR
    elif exercise == "moderate":
        BMR = BMR * 1.5
        return BMR
    elif exercise == "heavy":
        BMR = BMR * 1.9
        return BMR
```

The main change I made to these functions included removing any flashed error messages – I moved this inside the main webpage function to avoid any confusion when validating

This function called question is the python code that represents the questionnaire webpage.

All these variables have the user's inputs stored in them so I can run conditional requirements

```
@questionnaire.route('/question', methods=['GET', 'POST'])
def question():
    if "username" in session:
        username = session["username"]
        if request.method == 'POST':
            male = request.form.get('male', type=bool)
            female = request.form.get('female', type=bool)
            age = request.form.get('age', type=int)
            lose = request.form.get('lose', type=bool)
            main = request.form.get('main', type=bool)
            gain = request.form.get('gain', type=bool)
            height = request.form.get('height', type=float)
            weight = request.form.get('weight', type=float)
            exercise = request.form.get('exercise')
            bodyfat = request.form.get('bodyfat')
            dweight = request.form.get('dweight', type=float)
            chicken = request.form.get('chicken', type=bool)
            fish = request.form.get('fish', type=bool)
            redmeat = request.form.get('redmeat', type=bool)
            vegan = request.form.get('vegan', type=bool)
            dairy = request.form.get('dairy', type=bool)
            egg = request.form.get('egg', type=bool)
```

For every checkbox, the datatype being used is Boolean as these values are very easy to handle – they can only be two states

The AND, OR operators were much more efficient than writing extra if statements

AND – two conditions must be true for the whole statement to return true

OR – One of the two conditions must be true to return the whole statement as true

```
numbermeals = request.form.get('numbermeals', type=int)
condition = 0
if not request.form.get('male') and not request.form.get('female'):
    flash('You have not selected a gender', category='error')
elif male and female:
    flash('You have selected two genders', category='error')
elif not request.form.get('age'):
    flash('Invalid age', category='error')
elif age < 14 or age > 75:
    flash('You must be aged between 14 and 75', category='error')
elif not request.form.get('lose') and not request.form.get('main') and not request.form.get('gain'):
    flash('Your have not selected a goal', category='error')
elif lose and main and gain:
    flash('You have selected more than one goal', category='error')
elif lose and main:
    flash('You have selected more than one goal', category='error')
elif lose and gain:
    flash('You have selected more than one goal', category='error')
elif main and gain:
    flash('You have selected more than one goal', category='error')
elif not request.form.get('height'):
    flash('Invalid height', category='error')
```

```
elif not request.form.get('weight'):
    flash('Invalid weight', category='error')
elif weight < 35 or weight > 140:
    flash('Invalid weight', category='error')
elif not request.form.get('dweight'):
    flash('Invalid desired weight', category='error')
elif dweight < 35 or dweight > 140:
    flash('Invalid desired weight', category='error')
elif not request.form.get('numbermeals'):
    flash('Invalid number of meals', category='error')
elif numbermeals < 3 or numbermeals > 7:
    flash('The number of meals should be between 3 and 7', category='error')
else:
    goal = checkgoal(lose, main, gain, weight, dweight)
    if goal == 1:
        flash('Your desired weight does not match your goal', category='error')
    else:
        k = constant(male, female)
        BMR = target(weight, height, age, k, exercise)
        cursor.execute("UPDATE plan SET goal = %s, num_meals = %s, BMR = %s WHERE username = %s", (goal, numberme
        db.commit()
        if chicken:
```

The function checkgoal is called only when all the other conditions above it have been met

Using an integer as an error indicator worked successfully

These statements represent the user's food preferences, the database will be updated according to their options

Cursor.execute is a built-in function when using python to connect to a MySQL database

It executes SQL statements written in its parameters

As no records are being fetched, the fetchone() command is not needed

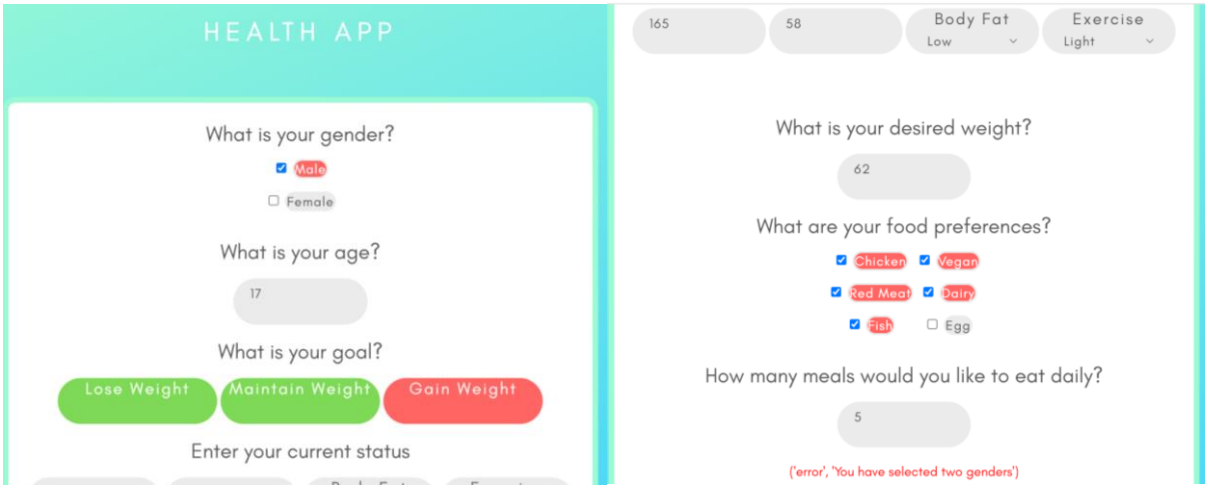
If all conditions have been met, the user will be redirected to their home page

If the user has not met all conditions, they will just see the question page

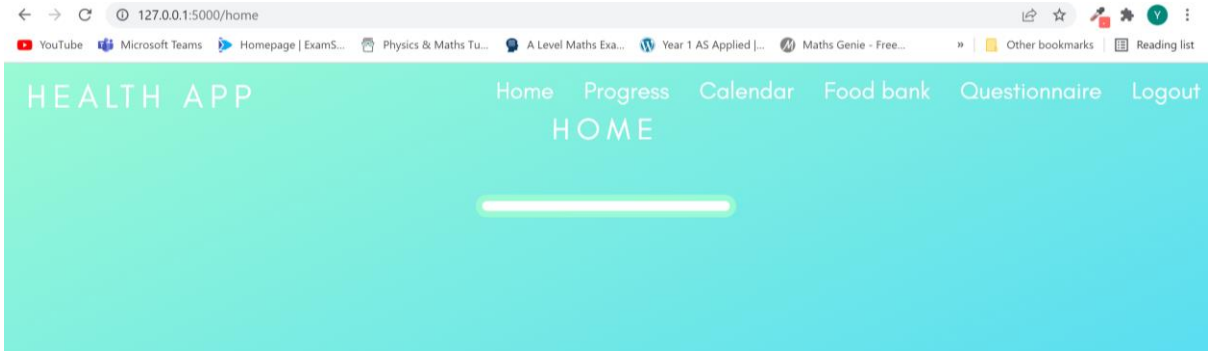
If the user has not logged in, they will be redirected to the login page

```
if chicken:
    cursor.execute("UPDATE plan SET chicken_pref = 1 WHERE username = %s", (username,))
    db.commit()
else:
    cursor.execute("UPDATE plan SET chicken_pref = 0 WHERE username = %s", (username,))
    db.commit()
if redmeat:
    cursor.execute("UPDATE plan SET redmeat_pref = 1 WHERE username = %s", (username,))
    db.commit()
else:
    cursor.execute("UPDATE plan SET redmeat_pref = 0 WHERE username = %s", (username,))
    db.commit()
if fish:
    cursor.execute("UPDATE plan SET fish_pref = 1 WHERE username = %s", (username,))
    db.commit()
else:
    cursor.execute("UPDATE plan SET fish_pref = 0 WHERE username = %s", (username,))
    db.commit()
if vegan:
    cursor.execute("UPDATE plan SET vegan_pref = 1 WHERE username = %s", (username,))
    db.commit()
else:
```

```
                db.commit()
            else:
                cursor.execute("UPDATE plan SET egg_pref = 0 WHERE username = %s", (username,))
                db.commit()
            if dairy:
                cursor.execute("UPDATE plan SET dairy_pref = 1 WHERE username = %s", (username,))
                db.commit()
            else:
                cursor.execute("UPDATE plan SET dairy_pref = 0 WHERE username = %s", (username,))
                db.commit()
            return redirect(url_for('homepage.home'))

    return render_template("question.html")
else:
    return redirect(url_for('loginsystem.login'))
```



While entering values, I done multiple tests to see if the right errors could be generated and they were. Once I entered the full form with valid values, I was successfully redirected to the home page as I planned. This was invalid and valid testing taking place.



Returns all records from the specified table, in this case: plan

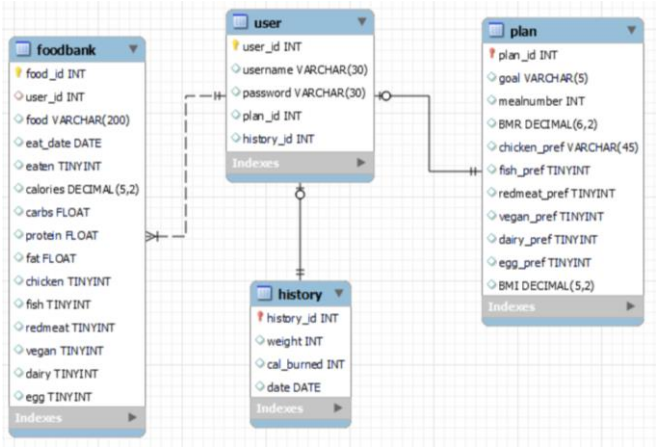
```
USE mydb;
SELECT * FROM plan;
```

username	goal	num_meals	chicken_pref	fish_pref	redmeat_pref	vegan_pref	dairy_pref	egg_pref	BMI	BMR
ADMIN	gain	5	1	1	1	1	1	0	NULL	1837.50

Alternatively, this could have been written in python but I wanted to take a less riskier option in case of an error

I ran a query in the MySQL workbench and found that the inputs from the question page have been inserted. From a very large iteration of fails, tests and successes, the database and the login system of my solution was concluded after this last test resulted in success.

Looking back on my database structure, it does not necessarily follow the rules of foreign keys – A field acting as the primary key in one table, appears in another.



This is a much better database design as all foreign key fields are present in another table as a primary key. However, the database design I currently have is functional and is accepted by MySQL so I do not have a reason to change it as of now. If I find any issues with my current database, I can always create this database and move to using this one.

Home Page

Aim: Have the homepage display the user's goal.

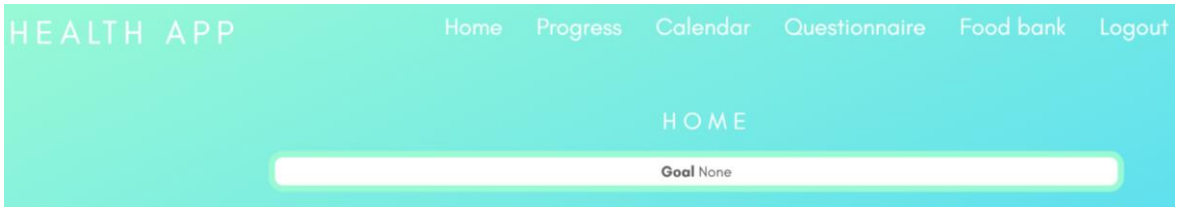
Justification: This is the first specification point from my success criteria from the analysis section

The app checks if the user has a session and so if they are then the home page will be rendered, if the user is not signed is then they will be directed to the login page.

```
@homepage.route('/home')
def home():
    if "username" in session:
        username = session["username"]
        cursor.execute("SELECT goal FROM plan WHERE username = %s", (username))
        goal = cursor.fetchone()
        if goal == "lose":
            displaygoal = "Losing Weight"
        elif goal == "main":
            displaygoal = "Maintaining Weight"
        elif goal == "gain":
            displaygoal = "Gaining Weight"
        else:
            return redirect(url_for('questionnaire.question'))
        return render_template("home.html")
    else:
        return redirect(url_for('loginsystem.login'))
```

This is a function that checks what the user's goal is.

```
homepage = Blueprint('homepage', __name__)

def displaygoal(goalcheck):
    if goalcheck == "lose":
        goal = "Losing Weight"
        return str(goal)
    elif goalcheck == "main":
        goal = "Maintaining Weight"
        return str(goal)
    elif goalcheck == "gain":
        goal = str("Gaining Weight")
        return goal
```



This was an unexpected result because I was logged in using the “ADMIN” username and I completed the questionnaire with this account, declaring the goal as “Gain weight” but this was not shown on the page.

Test: Run the same code in another python file

Justification: To diagnose what the issue is. It is easier to focus on the issue if I move the code to a file separated from the entire app

```
db = mysql.connector.connect(
    host="localhost",
    user="root",
    passwd="247890YR",
    database="mydb"
)
cursor = db.cursor()

username = str(input("Enter something "))
cursor.execute("SELECT goal FROM plan WHERE username = %s", (username,))
goal = cursor.fetchone()
print(goal)
```

```
Enter something ADMIN
('gain',)
```

The result of this test indicated that the value fetched from the SQL statement was not a standard string, it was a tuple because that is the data structure used when running database queries.

Modification: Use an index to target a specific value in the tuple returned by the query

Justification: So that the value can then be used in a variable

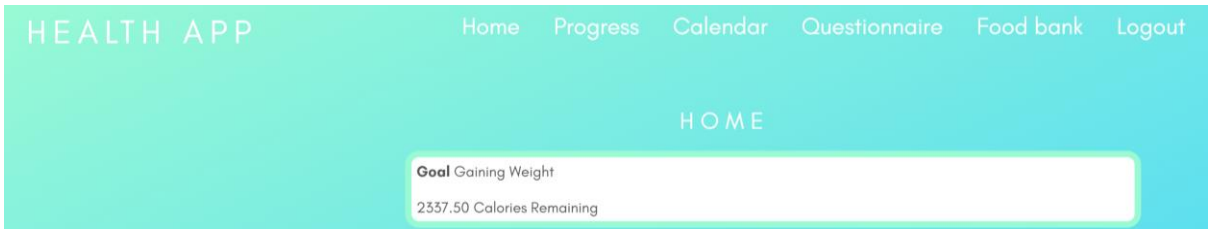
```
db = mysql.connector.connect(
    host="localhost",
    user="root",
    passwd="247890YR",
    database="mydb"
)
cursor = db.cursor()

username = str(input("Enter something "))
cursor.execute("SELECT goal FROM plan WHERE username = %s", (username,))
goal = cursor.fetchone()[0]
print(goal)
```

```
Enter something ADMIN
gain
```

Calories is referring to fetching the record from the plan table in the BMR column because that is the number of calories needed to maintain weight.

```
def calcheck(username, calories):
    cursor.execute("SELECT goal FROM plan WHERE username = %s", (username,))
    goal = cursor.fetchone()[0]
    if goal == "lose":
        calories = calories-500
        return calories
    elif goal == "main":
        return calories
    elif goal == "gain":
        calories = calories+500
        return calories
    else:
        calories = "You do not have a calorie target"
        return calories
```

Depending on the goal, the calories displayed varies.



This was displayed successfully.

Aim: Create a progress bar for the user

Justification: This is a specification point from my success criteria from the analysis section

```
USE mydb;
UPDATE food_bank SET food = "Steak and mashed potatoes" WHERE user_food = "ADMIN";
UPDATE food_bank SET eat_date = '2022-02-19' WHERE user_food = "ADMIN";
UPDATE food_bank SET calories = 304 WHERE user_food = "ADMIN";
SELECT * FROM food_bank;
```

```
USE mydb;
INSERT INTO food_bank (user_food, food, eat_date, calories, eaten)
VALUES ("ADMIN", "Grilled Chicken", '2022-02-19', 602.5, 1);
SELECT * FROM food_bank;
```

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

food_id	user_food	food	eat_date	calories	carbs	protein	fat	sugar	salt	chicken	fish	redmeat	vegan	dairy	egg	eaten
1	ADMIN	Steak and mashed potatoes	2022-02-19	304.00	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	1
4	ADMIN	Grilled Chicken	2022-02-19	602.50	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	1
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

In order to see if the progress bar works, there has to be food data to work with and so I inserted values into the database, in the “food_bank” table. In the eaten column, I inserted “1” meaning the app should deduce that the user has eaten the food and so the progress bar should adapt along with the calories remaining number. There was still the issue of working out how the app should run queries based on specific dates.

Test: Use the datetime python module to return the date for today

Justification: The app needs to workout all the user’s eaten foods for the present day, which is always changing.

I imported date time and used this function to return the date for today.

```
today = datetime.date.today()

username = str(input('Enter something '))
cursor.execute("SELECT calories FROM food_bank WHERE user_food = %s AND eat_date = %s AND eaten = 1",(username, today))
todayprogress = cursor.fetchall()
print(todayprogress)
```

```
Enter something ADMIN
[(Decimal('304.00'),), (Decimal('602.50'),)]
```

"Today" is being used in filtering the SQL statement via string formatting (%s).

This showed the datetime function worked when being used in the SQL statement – the statement returned calories of each food eaten corresponding to today's date.

Using a for loop once will refer to each food retrieved however using another for loop nested inside, points to the calorie value of each food without being in a tuple format.

```
def caleaten(todayprogress):
    if todayprogress:
        total = 0
        for i in todayprogress:
            for x in i:
                total = total + x
        return total
    else:
        return 0
```

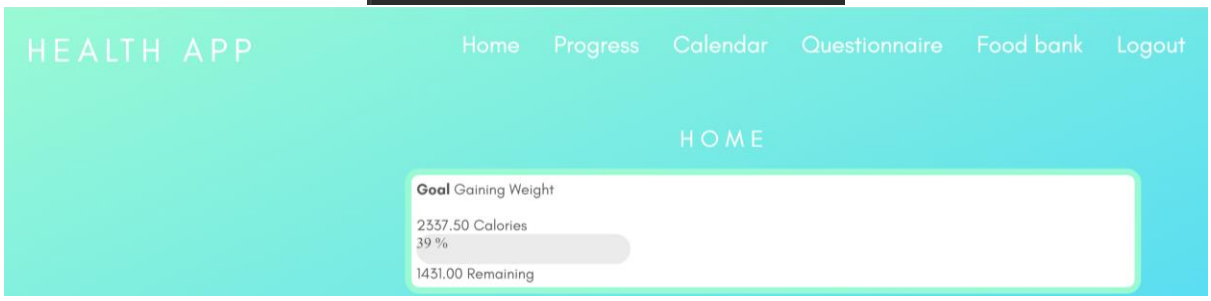
"If todayprogress" checks if there are data points in the SQL statement, if there are none, then 0 is returned.

```
def home():
    if "username" in session:
        username = session["username"]
        username = str(username)
        cursor.execute("SELECT goal FROM plan WHERE username = %s", (username,))
        goalcheck = cursor.fetchone()[0]
        cursor.execute("SELECT BMR FROM plan WHERE username = %s", (username,))
        calories = cursor.fetchone()[0]
        goal = displaygoal(goalcheck)
        target = calcheck(username, calories)
        cursor.execute("SELECT calories FROM food_bank WHERE user_food = %s AND eat_date = %s AND eaten = 1", (username, today
        todayprogress = cursor.fetchall()
        eaten = caleaten(todayprogress)
        remaining = target - eaten
        percentage = round((eaten/target) * 100)
        return render_template("home.html", displaygoal=goal, target=target, remain=remaining, percent=percentage)
    else:
        return redirect(url_for('loginsystem.login'))
```

Total variable stores the total calories from what the user ate.

Using two sets of curly brackets indicates that a python variable is being passed to the HTML page.

```
<div class="box">

  {% block content %}

    <p1 class="subheading">Goal</p1>
    <p1 class="text" >{{displaygoal}}</p1>

    <br/>
    <br/>


    <p1 class="text">{{target}} Calories</p1>
    <div class="bar" class="text">{{percent}} %</div>
    <p1 class="text">{{remain}} Remaining</p1>

    {% endblock %}
<!-- -->
</div>
```

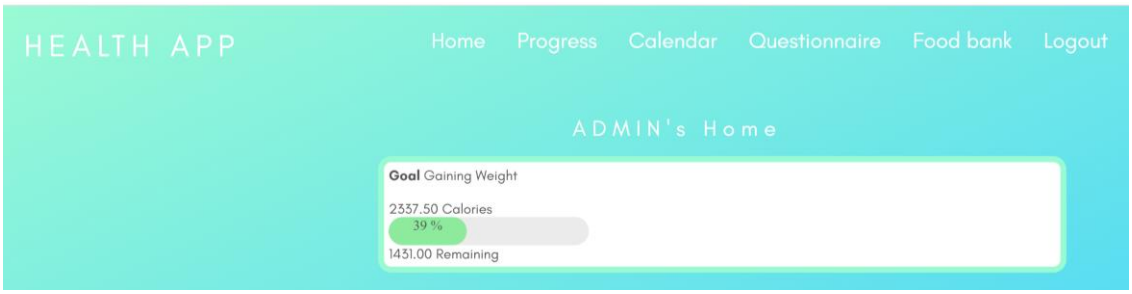


The percentage figure and the calories remaining successfully adapted to the user's eaten foods in the database.

I passed in the percent variable to a javascript function that retrieves the progress bar element and sets the width according to the percent variable.

```
    </div>
    <p1 class="text">{{remain}} Remaining</p1>

    {% endblock %}
<!-- {{percent}}-->
</div>

</body>
</html>

<script>
    document.getElementById("progressbar").style.width = "{{percent}}%";
</script>
```

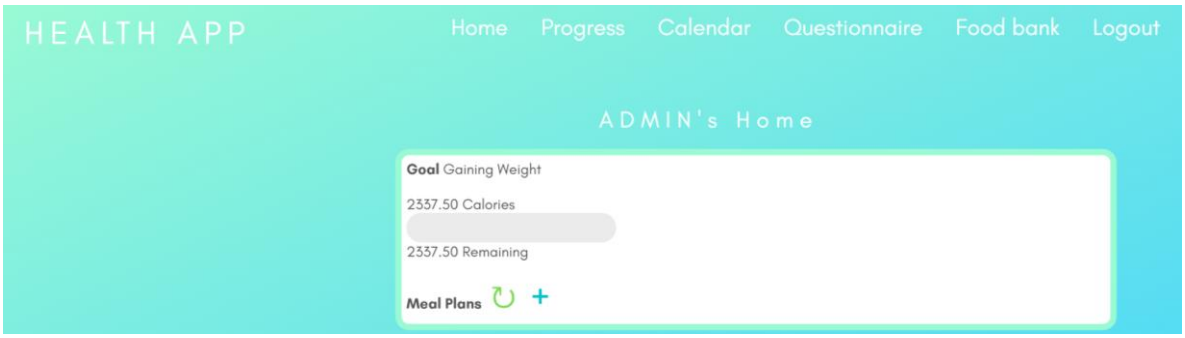
Having the user's username display in the heading of the page was an idea likened by my stakeholders.



The progress bar consists of two parts: one bar that was grey and another that was a different colour. This gives the illusion of a progress bar when actually there are two bars of the same height but different colour, being overlayed together.

Aim: Generate a meal plan for the user

Justification: This is a specification point on my success criteria and is the main feature of the solution.



I used simple Unicode icons to symbolize a button for generating meals and adding extra meals.

The idea of generating meals that take into account the user's preferences was quite tricky. I could not just retrieve the user's entire preferences and then use that to filter a query because for example, the user may like chicken and fish but the app will only generate meals that have chicken AND fish rather than no fish, but chicken or no chicken, but fish. This meant I had to highlight all the foods the user did not like and use that to filter a query rather than directly copying what the user likes and does not like.

All is the variable that contains all the user's preferences.

```
def recursion(all):
    if all[0] == 0:
        if all[1] == 0:
            if all[2] == 0:
                if all[3] == 0:
                    if all[4] == 0:
                        cursor.execute("SELECT * FROM food_bank WHERE food_user = 'ADMIN' OR food
    if all[1] == 0:
        list.append("fish")
if all[2] == 0:
    list.append("redmeat")
if all[3] == 0:
    list.append("vegan")
if all[4] == 0:
```

I tried using a recursive function but this did not bring me closer to solving the issue.

```
(1, 1, 1, 1, 1, 0)
['egg']
```

I found that running a series of if statements and then creating a list of all the food preferences the user did not eat gave me something to work with.

Test: Use python formatting to use variables as column names

Data: chicken, egg

Justification: If this is successful, then I may have a solution to generating customized meal plans. I already can retrieve the user's preferences so this would bring me closer to the solution.

```
chicken = 'chicken'
egg = 'egg'


cursor.execute('SELECT * FROM food_bank WHERE {} = 0 OR {} = 0'.format(chicken, egg),)
print(cursor.fetchall())
```

```
[(1, 'ADMIN', 'Steak and mashed potatoes', datetime.date(2022, 2, 19), Decimal('304.00'), None, None, None, None, None, 0, 0, 1, 0, 0, 0, 1), (4,
```

The result of this test showed that I could use variables as column names in a SQL statement, not just as values of the statement.

The generate function takes in the username variable and column as parameters. Column is a list of preferences the user does not eat.

```
def generate(column, username):
    if len(column) == 1:
        cursor.execute("SELECT * FROM food_bank WHERE user_food = 'ADMIN' OR user_food = %s AND %s = 0",(username, column[0],))
        return cursor.fetchall()
    elif len(column) == 2:
        cursor.execute("SELECT * FROM food_bank WHERE user_food = 'ADMIN' OR user_food = %s AND %s = 0 AND %s = 0",(username,
        return cursor.fetchall()
    elif len(column) == 3:
        cursor.execute("SELECT * FROM food_bank WHERE user_food = 'ADMIN' OR user_food = %s AND %s = 0 AND %s = 0 AND %s = 0",
        return cursor.fetchall()
    elif len(column) == 4:
        cursor.execute("SELECT * FROM food_bank WHERE user_food = 'ADMIN' OR user_food = %s AND %s = 0 AND %s = 0 AND %s = 0 A
        return cursor.fetchall()
    elif len(column) == 5:
        cursor.execute("SELECT * FROM food_bank WHERE user_food = 'ADMIN' OR user_food = %s AND %s = 0 AND %s = 0 AND %s = 0 A
        return cursor.fetchall()
    elif len(column) == 6:
        cursor.execute("SELECT * FROM food_bank WHERE user_food = 'ADMIN' OR user_food = %s AND %s = 0 AND %s = 0 AND %s = 0 A
        return cursor.fetchall()
    else:
        cursor.execute("SELECT * FROM food_bank WHERE user_food = 'ADMIN' OR user_food = %s", (username,))
        return cursor.fetchall()
```

I used string formatting to use the different values in the column list to query the database.

Using indexes pointed to the different values in column.

Column creates a list of food preference names the user does not like.

```
username = str(input("Enter something "))
cursor.execute("SELECT chicken_pref, fish_pref, redmeat_pref, vegan_pref, dairy_pref, egg_pref FROM plan WHERE username = %s"
pref = cursor.fetchone()
column = []

if pref:
    if pref[0] == 0:
        column.append("chicken")
    if pref[1] == 0:
        column.append("fish")
    if pref[2] == 0:
        column.append("redmeat")
    if pref[3] == 0:
        column.append("vegan")
    if pref[4] == 0:
        column.append("dairy")
    if pref[5] == 0:
        column.append('egg')
else:
    pass

print(generate(column, username))
```

The generate function is called and then displayed.

Now to put this to the test, I needed to work with values in the database.

Test: Run the generate function to see if the app can produce the correct meals

Justification: If this is successful, then I may have a solution to generating customized meal plans. I already can retrieve the user's preferences so this would bring me closer to the solution.

```
USE mydb;
UPDATE plan SET chicken_pref = 0;
SELECT * FROM plan;
```

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

username	goal	num_meals	chicken_pref	fish_pref	redmeat_pref	vegan_pref	dairy_pref	egg_pref	BMI	BMR
ADMIN	gain	5	0	1	1	1	1	0	NULL	1837.50
Alpha	NULL	NULL	0	NULL	NULL	NULL	NULL	NULL	NULL	NULL
Beta	main	6	0	1	1	0	1	0	NULL	1831.50
Theta	main	4	0	1	1	0	1	1	NULL	1336.80
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

I updated all records in the plan table so that it shows that all users do not like chicken – no chicken meals should be displayed.

```
Enter something Beta
[('Steak and mashed potatoes',), ('Grilled Chicken',), ('Weetabix',)]
```

This was not the expected outcome of the test – Grilled chicken should not have been displayed.

```
USE mydb;
SELECT food FROM food_bank WHERE (user_food = "ADMIN" OR user_food = "Beta") AND chicken = 0 AND vegan = 0 AND egg = 0;
```

food
Steak and mashed potatoes
Weetabix

I executed a query in MySQL to see if there was an issue with the statement, I wrote but the correct response was returned in the workbench.

Modification: Write the same SQL statement in python with the use of python formatting rather than string formatting

Justification: This is to test if there is an issue with the way I was formatting the variables into the SQL statement.

```
cursor.execute("SELECT food FROM food_bank WHERE {} = 0 AND {} = 0".format(chicken, egg,))
print(cursor.fetchall())
```

```
[('Steak and mashed potatoes',), ('Weetabix',)]
```

The result of this test showed that there was an issue with string formatting.

```
        column.append("redmeat")
    if pref[3] == 0:
        column.append("vegan")
    if pref[4] == 0:
        column.append("dairy")
    if pref[5] == 0:
        column.append('egg')
else:
    pass
print(column)
foods = generate(column, username)
print(foods)
```

I used python formatting instead of string formatting for this statement and the correct response was returned.

```
Enter something ADMIN
(0, 1, 1, 1, 1, 0)
['chicken', 'egg']
[('Steak and mashed potatoes',), ('Weetabix',)]
```

Test: Change the preferences of a user

Justification: To see if the correct response is generated which would mean the app could adapt to any changes

```
USE mydb;
UPDATE plan SET chicken_pref = 1 WHERE username = "Theta";
SELECT * FROM plan;
```

username	goal	num_meals	chicken_pref	fish_pref	redmeat_pref	vegan_pref	dairy_pref	egg_pref	BMI	BMR
ADMIN	gain	5	0	1	1	1	1	0	NULL	1837.50
Alpha	NULL	NULL	0	NULL	NULL	NULL	NULL	NULL	NULL	NULL
Beta	main	6	0	1	1	0	1	0	NULL	1831.50
Theta	main	4	1	1	1	0	1	1	NULL	1336.80
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

```
    elif len(column) == 5:
        cursor.execute("SELECT food FROM food_bank WHERE (user_food = 'ADMIN' OR user_food = %s) AND {} = 0 AND {} = 0 AND {} = 0 AND {} = 0 AND
        return cursor.fetchall()
    elif len(column) == 6:
        cursor.execute("SELECT food FROM food_bank WHERE (user_food = 'ADMIN' OR user_food = %s) AND {} = 0 AND {} = 0 AND {} = 0 AND {} = 0 AND
        return cursor.fetchall()
    else:
        cursor.execute("SELECT * FROM food_bank WHERE user_food = 'ADMIN' OR user_food = %s", (username,))
        return cursor.fetchall()

username = input("Enter something ")
cursor.execute("SELECT chicken_pref, fish_pref, redmeat_pref, vegan_pref, dairy_pref, egg_pref FROM plan WHERE username = %s", (username,))
pref = cursor.fetchone()
print(pref)
column = preferences(pref)
print(column)
print(generate(column, username))
```

```
spare
C:\Users\User\AppData\Local\Programs\Python\Python37\python.exe "C:/Users/User/Desktop/Health App/Health App - Coursework/website/spare.py"
Enter something Theta
(1, 1, 1, 0, 1, 1)
['vegan']
[('Steak and mashed potatoes',), ('Grilled Chicken',), ('Weetabix',)]
```

This left me with a working method to generate foods based on the user's preferences.

Test: Write an SQL statement that will randomly return a set number of meals.

Justification: This is a step in the right direction for generating a meal plan for users as foods should be randomly selected and then displayed.

In order to randomly generate food based on the user's number of meals, the "food_id" can be used as an index when using the random function in python.

Since I wanted the nutritional values of foods to be displayed, I realized I had to create a new column called amount in the foodbank table which refers to the weight of the food. The protein, carbs and fat column will refer to percentages.

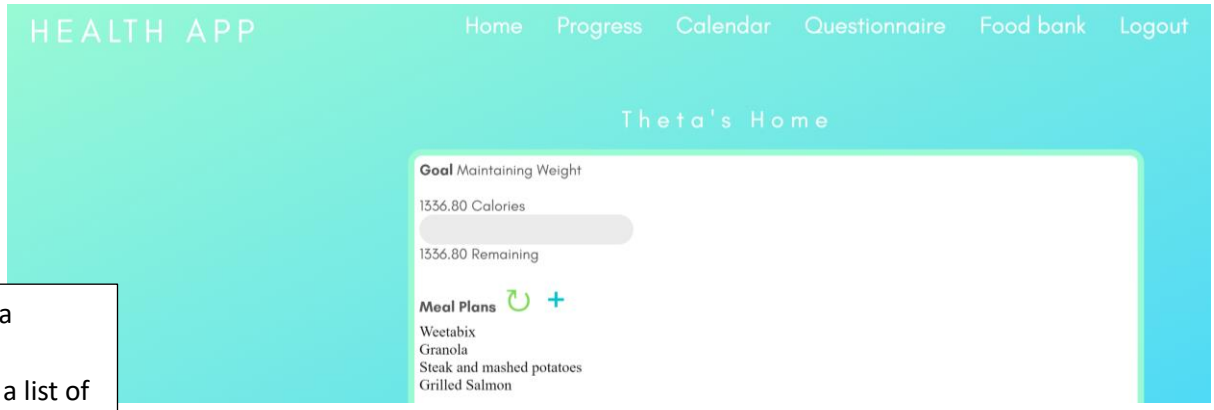
RAND limit(number) returns a random set of records from the table.

```
USE mydb;
SELECT * FROM food_bank ;
SELECT * FROM food_bank ORDER BY RAND()LIMIT 5;
```

food_id	user_food	food	eat_date	calories	carbs	protein	fat	sugar	salt	chicken	fish	redmeat	vegan	dairy	egg	eaten
6	ADMIN	Grilled Salmon	2022-02-20	250.00	NULL	NULL	NULL	NULL	NULL	0	1	0	0	0	0	1
4	ADMIN	Granola	2022-02-20	235.00	NULL	NULL	NULL	NULL	NULL	0	0	0	0	1	0	1
1	ADMIN	Steak and mashed potatoes	2022-02-19	304.00	15	70	15	NULL	NULL	0	0	1	0	0	0	1
7	ADMIN	Pasta Salad	2022-02-20	250.00	NULL	NULL	NULL	NULL	NULL	0	0	0	1	0	0	1
3	ADMIN	Weetabix	2022-02-20	315.00	NULL	NULL	NULL	NULL	NULL	0	0	0	0	1	0	1

Five random records were fetched successfully, in no particular order - this test result meant I could use the exact same code in python.

When I used this code in the python file, the meals successfully rendered to the webpage.

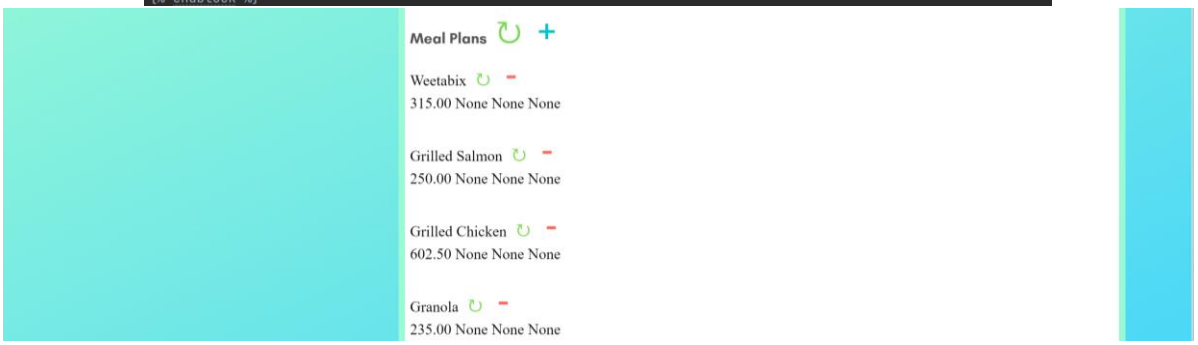


"Foods" is a variable containing a list of meals generated for the user.

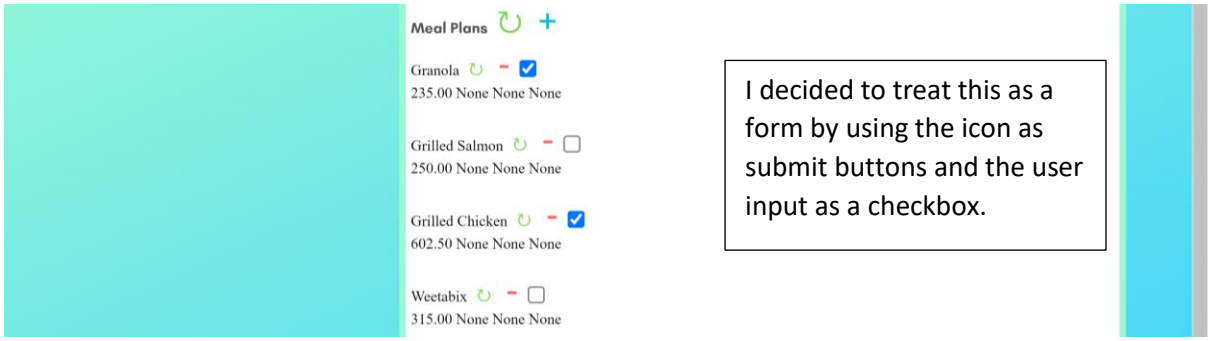
For each food, there will be a green button and a red button.

```
<button class="icon" onclick="addfood()" style="color: #00C2CB;" type="button" id="add" value="false"> +</button>
<br/>
{% for i in foods %}
<p1>{{ i[0] }}</p1>
<button class="minigen" >&#8635</button>
<button class="delete">-</button>
<br/>
<p1>{{ i[1] }}</p1>
<p1>{{ i[2] }}</p1>
<p1>{{ i[3] }}</p1>
<p1>{{ i[4] }}</p1>
<br/>
<br/>
{% endfor %}
<br/>
<br/>

{% endblock %}
```

The index of each item in the food list is the name, calories, carbs, protein and fat values. I wrote separate paragraph tags for each index in the food list.



The meal plan was displayed as intended along with the two other icons which symbolize generating a new individual meal to replace an existing one, and a delete icon which will delete the corresponding meal.



I decided to treat this as a form by using the icon as submit buttons and the user input as a checkbox.

I used checkboxes because they proved to be successful when working on the questionnaire page.

Aim: Complete the basic structure for the home page.

Justification: Organizing this now will make designing the functionalities of each feature on this page a lot easier.

Having an exercise input was part of my specification and so I moved onto this. I realized that there was no exercise column in the plan table that I could quickly refer to.

I added an exercise column of a VARCHAR datatype – the maximum string length is 15 characters.

```
1 • USE mydb;
2 • ALTER TABLE plan
3   ADD exercise VARCHAR(15);
4 • SELECT * FROM plan;
```

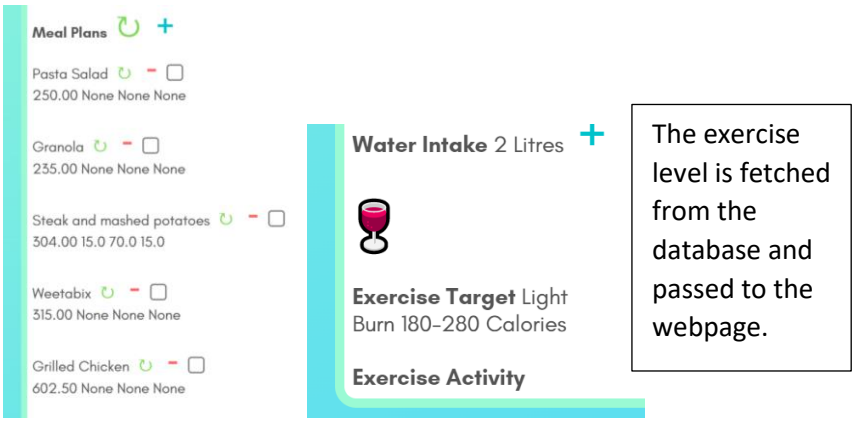
username	goal	num_meals	chicken_pref	fish_pref	redmeat_pref	vegan_pref	dairy_pref	egg_pref	BMI	BMR	exercise
ADMIN	gain	5	0	1	1	1	1	0	NULL	1837.50	NULL
Alpha	NULL	NULL	0	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
Beta	main	6	0	1	1	0	1	0	NULL	1831.50	NULL
Theta	main	4	1	1	1	0	1	1	NULL	1336.80	NULL
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

I wanted to add the user's exercise level to the plan table when the user completes the questionnaire page. I added a simple statement in the questionnaire python file; INSERT INTO plan (exercise) which will insert the user's input from the choice of three options; light, moderate, heavy. I completed the questionnaire page again so that the database will now take into account of exercise level.

I found that the newly created exercise column was updated.

username	goal	num_meals	chicken_pref	fish_pref	redmeat_pref	vegan_pref	dairy_pref	egg_pref	BMI	BMR	exercise
ADMIN	gain	5	0	1	1	1	1	0	NULL	1837.50	NULL
Alpha	NULL	NULL	0	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
Beta	main	6	0	1	1	0	1	0	NULL	1831.50	NULL
Theta	main	5	1	1	1	1	1	0	NULL	1987.80	light
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

After this, I went back to designing the home page.



The exercise level is fetched from the database and passed to the webpage.

The basic structure of the home page was now complete. In terms of functionality, the app had to generate a number of meals so that the total calories of the entire meal plan is not less than the user's calorie target for the day.

While the total calories in the plan is less than the user's target, run the function to generate a new set of foods.

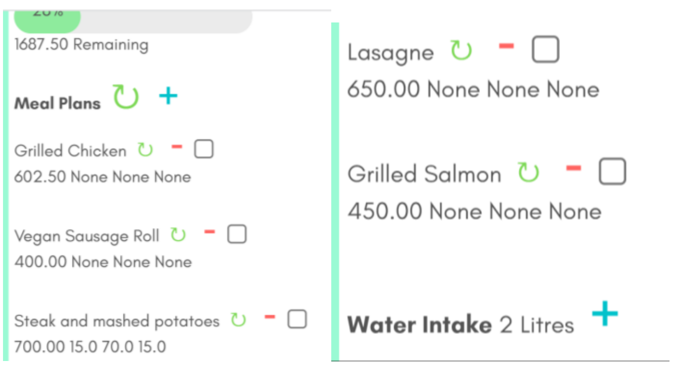
```
    elif len(column) == 6:
        cursor.execute("SELECT food, calories, carbs, protein, fat FROM food_bank WHERE (user_food = 'ADMIN' OR user_food = %s
        return cursor.fetchall()
    else:
        cursor.execute("SELECT food, calories, carbs, protein, fat FROM food_bank WHERE user_food = 'ADMIN' OR user_food = %s
        return cursor.fetchall()

def regen(foods, target, column, username, foodno):
    totalplan = 0
    for i in foods:
        totalplan = totalplan + i[1]
    while totalplan < target:
        generate(column, username, foodno)
        foods = regen(foods, target, column, username, foodno)
    return foods
```

```
File "C:\Users\User\Desktop\Health App\Health App – Coursework\website\homepage.py", line 107, in regen
  foods = regen(foods, target, column, username, foodno)
File "C:\Users\User\Desktop\Health App\Health App – Coursework\website\homepage.py", line 107, in regen
  foods = regen(foods, target, column, username, foodno)
File "C:\Users\User\Desktop\Health App\Health App – Coursework\website\homepage.py", line 107, in regen
  foods = regen(foods, target, column, username, foodno)
File "C:\Users\User\Desktop\Health App\Health App – Coursework\website\homepage.py", line 107, in regen
  foods = regen(foods, target, column, username, foodno)
File "C:\Users\User\Desktop\Health App\Health App – Coursework\website\homepage.py", line 107, in regen
  foods = regen(foods, target, column, username, foodno)
File "C:\Users\User\Desktop\Health App\Health App – Coursework\website\homepage.py", line 107, in regen
  foods = regen(foods, target, column, username, foodno)
File "C:\Users\User\Desktop\Health App\Health App – Coursework\website\homepage.py", line 107, in regen
  foods = regen(foods, target, column, username, foodno)
File "C:\Users\User\Desktop\Health App\Health App – Coursework\website\homepage.py", line 107, in regen
  foods = regen(foods, target, column, username, foodno)
File "C:\Users\User\Desktop\Health App\Health App – Coursework\website\homepage.py", line 107, in regen
  foods = regen(foods, target, column, username, foodno)
File "C:\Users\User\Desktop\Health App\Health App – Coursework\website\homepage.py", line 107, in regen
  foods = regen(foods, target, column, username, foodno)
File "C:\Users\User\Desktop\Health App\Health App – Coursework\website\homepage.py", line 107, in regen
```

I initially added a while loop that will call itself each time until the total calories of the food list is not less than the user's calorie target however this resulted in an error – stack overflow.

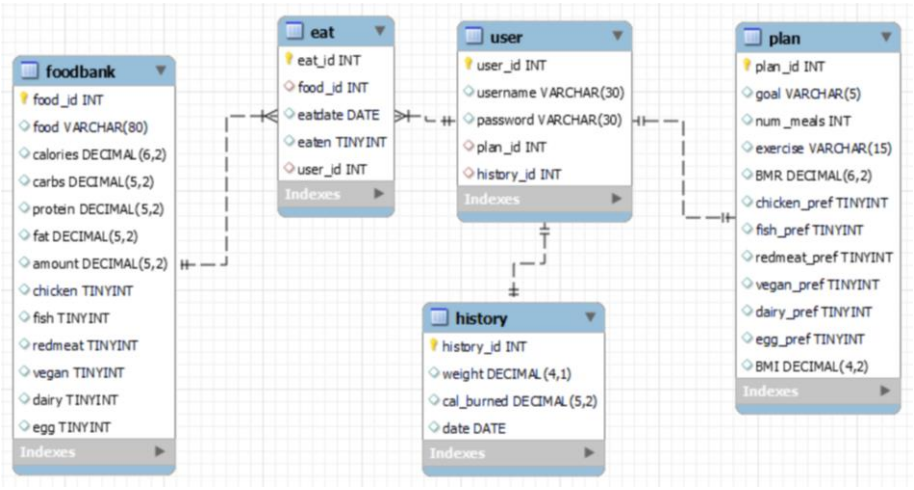
```
def regen(foods, target, column, username, foodno):
    totalplan = 0
    foodplan = ""
    for i in foods:
        totalplan = totalplan + i[1]
    if totalplan < target:
        foods = generate(column, username, foodno)
        foodplan = regen(foods, target, column, username, foodno)
        return foodplan
    else:
        return foods

username = input("Enter something ")
```

Instead of a while loop, I replaced it with an if statement that will constantly check if the total calories of the meal plan.



This resulted in no errors and so it meant the app could generate a meal plan that does not fall under the user's calorie target. However, I realized that if the user marks a food as eaten, it will be added again to the food database. Not only does this mean that the same food will appear more than once with almost identical values but it will also mean that when the user generates a food plan on a separate occasion, they may end up selecting the same food multiple times in one plan which is not very useful for the user.

This made me think about my database model and how adding another table called "eaten" which stores all the foods that the user would like to eat or has eaten could refer to foods in the food_bank table via the food_id.



I decided to restructure my database to incorporate the new table but also to change the foreign keys within the tables. This is because with the previous database model, the foreign keys were not the primary keys of other tables.

Test: Check if a join command in SQL can be used between two tables

Justification: This would be a key component to working out the values of what the user has marked as eaten.

```
1 • USE health;
2 • UPDATE eat SET eaten = 0 WHERE food_id = 2;
3 • SELECT * FROM eat;
```

eat_id	food_id	eatdate	eaten	user_id
1	1	2022-02-25	1	1
2	2	2022-02-25	0	1
NULL	NULL	NULL	NULL	NULL

My intention was to see if I could return the records where the user has marked a certain food as eaten. This would also involve checking the user's user_id that corresponds to the eaten food.

```
1 • USE health;
2 • SELECT calories FROM eat INNER JOIN foodbank ON eat.food_id = foodbank.food_id;
```

calories
602.50

The inner join fetches the record where the value in a column is equal to the value of a column in another table.

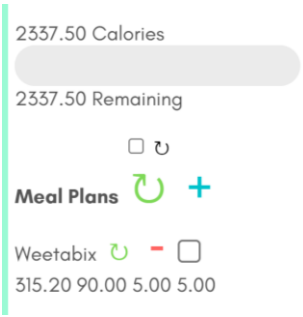
In this case, it returns records from the foodbank table if the food_id appears in the eat table.

This returned the correct result as only one record was returned.

```
if "username" in session:
    username = session["username"]
    username = str(username)
    cursor.execute("SELECT user_id FROM user WHERE username = %s", (username,))
    id = cursor.fetchone()[0]
```

Following the restructure of my database, I decided to create an id variable that will be used throughout each python file. When the user logs in, their username is stored in the session dictionary and so I wrote an SQL statement that takes in the username via string formatting, to retrieve the user_id correspondant to the username. This will allow me to run queries with my newly created database as the tables use ids, not usernames.

If the user clicks the checkbox and then clicks the green button, the form gets submitted, passing the value "new" to python.



Python checks if the user has checked the box – if new is true

```
if new:
    foods = generate(column, foodno)
    foodplan = regen(foods, target, column, foodno)
    todayplan(foodplan, today, id)
```

```
<form method="POST" >
<input type="checkbox" style="display: inline-block;" id="new" name="new" value="false"/>
<button class="icon" type="submit" >&#8635</button>
```

I created another checkbox which made sure I was able to generate new meals but only when the user clicks on the checkbox and then proceeds to click the large green arrow icon. This is just a prototype so far, the main functionality of generating meals works but the final solution would have changes in terms of the styling. The same functions I created before are being used because this is what generates the meal plan – in this case, a new plan.

Aim: Create a feature that allows the user to delete meal plans.

Justification: This is a specification point from my success criteria.

```
<input type="checkbox" style="display: inline-block;" id="delete" name="delete" value="{{ i[0] }}"/>
<button class="delete" type="submit">-</button>
```

The index refers to the name of a food in the user's plan.

This button was part of the for loop I wrote directly into the HTML file that displays the different meals. I managed to create a delete feature in a similar style to generating new meals, the user has to check the box and then click the red "- "icon to delete. When the user confirms the removal of a food, the value "i[0]" is passed to python. This value is different for each food, as each food has a different name.

```
def deletefood(delete, id, today):
    cursor.execute("SELECT food_id from foodbank WHERE food = %s", (delete,))
    foodid = cursor.fetchone()[0]
    cursor.execute("DELETE FROM eat WHERE food_id = %s AND user_id = %s AND eatdate = %s", (foodid, id, today,))
    db.commit()
```

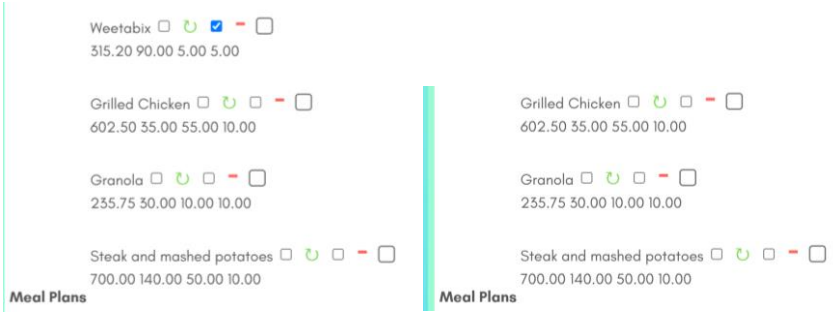
```
if delete:
    deletefood(delete, id, today)
```

DELETE FROM will remove the record from the SQL query, where the conditions are met.

I created a function that simply takes in the value passed from HTML, and runs a query to see which food needs to be deleted from the eat table.

Test: Select a checkbox to delete a specific food from the food plan.

Justification: This is to see if I have a working solution for the delete food feature part of my success criteria.



Only three meals remain in the plan.

This test resulted in success.

Aim: Create a feature that allows the user to mark foods as "eaten"

Justification: This is a specification point from my success criteria.

```
<input type="checkbox" style="display: inline-block; transform: scale(1.5);" id="eaten" name="eaten" value="{{ i[0] }}"/>
<button type="submit"></button>
```

```
if eat:
    eatmark(eat, id, today)
```

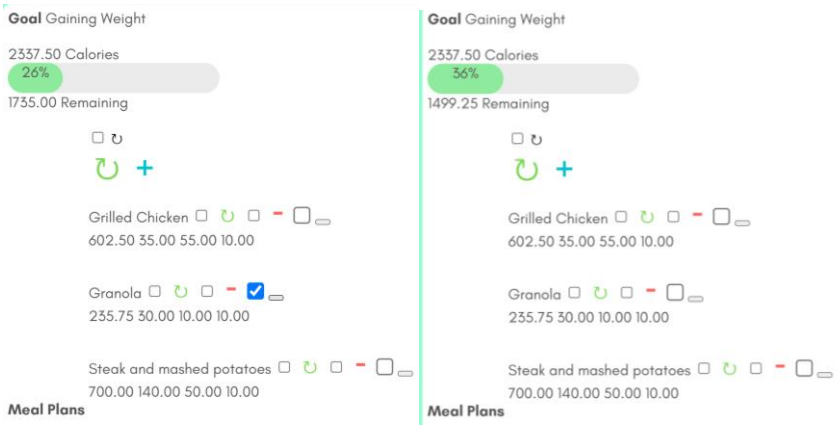
Function is run when user submits eaten value.

```
def eatmark(eat, id, today):
    cursor.execute("SELECT food_id from foodbank WHERE food = %s", (eat,))
    foodid = cursor.fetchone()[0]
    if eat:
        cursor.execute("UPDATE eat SET eaten = 1 WHERE food_id = %s AND user_id = %s AND eatdate = %s", (foodid, id, today,))
        db.commit()
```

I used the update SQL statement because a record already exists in the table, since the user has generated meals already.

Test: Mark a specific food as eaten to see if the progress bar will adapt.

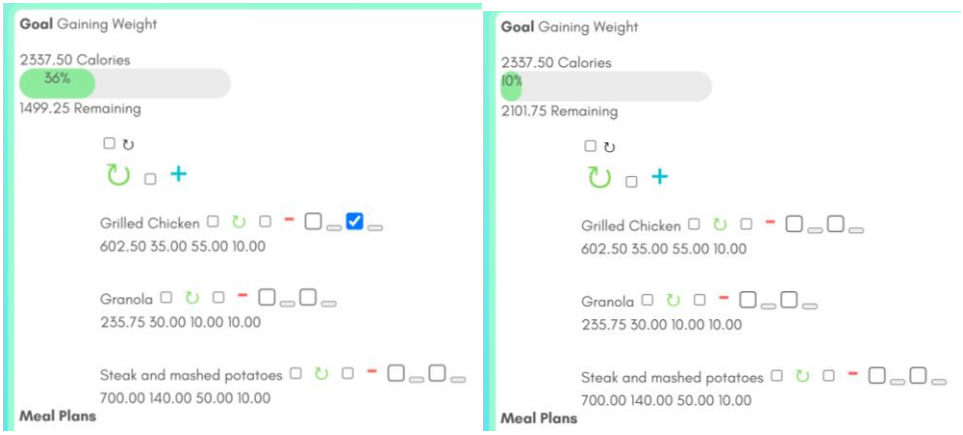
Justification: This is to check if the database and app will acknowledge the changes made by the user.



This showed the user can mark foods as eaten and the progress bar will adapt to the change. The calories remaining counter also adapted. By checking the largest checkbox, the database is updated to mark that the user has eaten the food.

Aim: Create a feature that allows the user to "unmark"

Justification: This is a specification point from my success criteria.



The last checkbox allows the user to "unmark" the food that they might of marked as eaten by accident. The function I created to achieve this was almost identical to the function used to mark foods as eaten, only difference was in the SQL statement; I updated the column to 0 instead of 1.

This is a prototype but in the final solution I plan to have the check box remain checked unless the user clicks on it again, if they uncheck the box, the database should update the corresponding food to "0" meaning the user has not eaten it.

Aim: Create a feature that allows the user to add extra meals to the existing plan.

Justification: This is a specification point from my success criteria.

```
if addfood:
    addfoodid = add(column,)
    cursor.execute("INSERT INTO eat (food_id, user_id, eatdate) VALUES (%s, %s, %s)", (addfoodid, id, today,))
    db.commit()
```

I knew that when adding an additional meal, the user's preferences must be taken into account and so I used the column variable containing a list of all their preferences and I used it in a function.

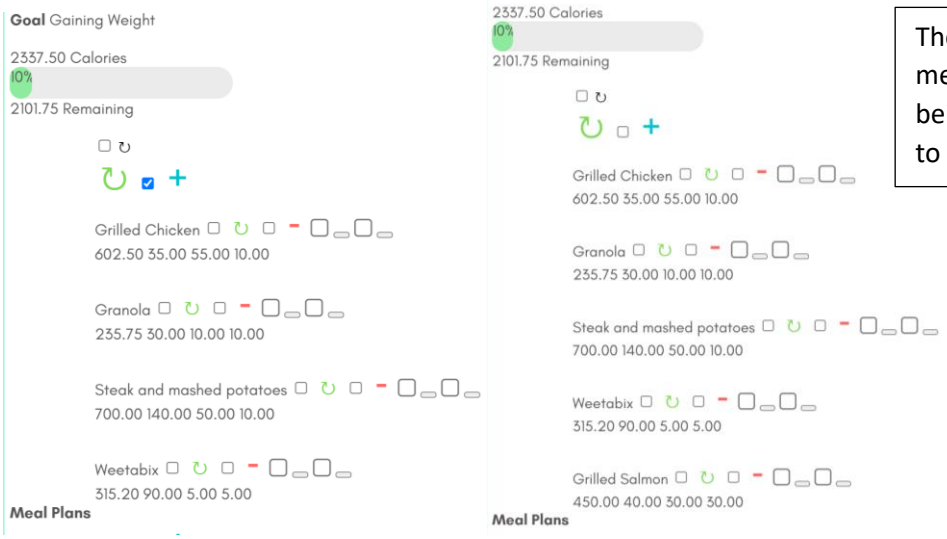
This code was almost identical to what I used for the function that creates the meal plan. Only difference is I limited each possible query to return one record.

```
def add(column):
    if len(column) == 1:
        cursor.execute("SELECT food_id FROM foodbank WHERE {} = 0 ORDER BY RAND()LIMIT 1".format(column[0],))
        return cursor.fetchone()[0]
    elif len(column) == 2:
        cursor.execute("SELECT food_id FROM foodbank WHERE {} = 0 AND {} = 0 ORDER BY RAND()LIMIT 1".format(column[0], column
        return cursor.fetchone()[0]
    elif len(column) == 3:
        cursor.execute("SELECT food_id FROM foodbank WHERE {} = 0 AND {} = 0 AND {} = 0 ORDER BY RAND()LIMIT 1".format(colum
        food = cursor.fetchone()[0]
        return food
    elif len(column) == 4:
        cursor.execute("SELECT food_id FROM foodbank WHERE {} = 0 AND {} = 0 AND {} = 0 AND {} = 0 ORDER BY RAND()LIMIT 1".fo
        return cursor.fetchone()[0]
    elif len(column) == 5:
        cursor.execute("SELECT food_id FROM foodbank WHERE {} = 0 AND {} = 0 AND {} = 0 AND {} = 0 AND {} = 0 ORDER BY RAND()
        return cursor.fetchone()[0]
    elif len(column) == 6:
        cursor.execute("SELECT food_id FROM foodbank WHERE {} = 0 AND {} = 0 AND {} = 0 AND {} = 0 AND {} = 0 AND {} = 0 ORD
        return cursor.fetchone()[0]
    else:
        cursor.execute("SELECT food_id FROM foodbank ORDER BY RAND()LIMIT 1")
        return cursor.fetchone()[0]
```

I string formatted the index variables containing the column names of the user's disliked food types which is stored in the column variable.

Test: Add a new meal to the existing meal plan – without changing any of the other meals.

Justification: This functionality test is important to see if I found a method to achieving the success criteria point.

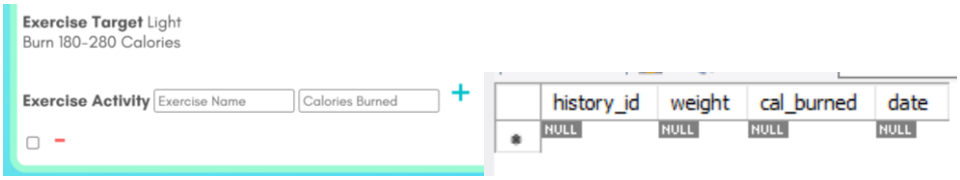


There are now 5 meals on the plan being displayed to the user.

The app successfully added a new meal to the existing plan

Aim: Create a feature that allows the user to input exercise activities.

Justification: This is a specification point from my success criteria.



Modification: Add a new column to the history table.

Justification: This will provide a method of identifying which values belong to which user. Without this, there will be no way for the app to display the user exercise activities.

I reviewed the history table and realized amendments had to be made to it once again. There was no user_id column in this table, so there would be no way to identify what data points belong to which user. Since only one table needed to be changed, it made no sense to completely generate a new database, given that I have already inserted many values for testing purposes already.

```
1 •  USE health;
2 •  ALTER TABLE user DROP CONSTRAINT fk_user_history1;
```

```
1 •  USE health;
2 •  ALTER TABLE history
3    ADD CONSTRAINT fk_history_user FOREIGN KEY (user_id) REFERENCES user(user_id)
4    ON DELETE CASCADE
5    ON UPDATE CASCADE;
6 •  DESC history;
```

Field	Type	Null	Key	Default	Extra
history_id	int	NO	PRI	NULL	auto_increment
weight	decimal(4,1)	YES		NULL	
cal_burned	int	YES		NULL	
date	date	YES		NULL	
user_id	int	YES	MUL	NULL	

CASCADE on update and delete refers to the scenario where the child table will be updated or deleted if the parent table is.

Since there was an existing foreign key constraint on the history_id, I had to drop the foreign key constraint. I then added a user_id column to the history table so that each record in the history table now belongs to a user. I followed this by adding a new constraint that references the user_id column in the user table which is the primary key of that table. The CASCADE feature helps maintain referential integrity easier as if the user_id of the user table is changed, the user_id of the history table will change accordingly.

```
<form method="POST">
<label for="burninput"></label>
<input class="type" type="number" id="burninput" name="burninput" placeholder="Calories Burned"/>
<button class="icon" style="color: #00C2CB;" type="submit"> +</button>

<br/>
{% for x in caloriesburned %}
<br/>

<p1 class="text">{{ x[0] }} calories burned</p1>

<input type="checkbox" style="..." value="{{ x[0] }}" id="deleteburn" name="deleteburn"/>
<button class="delete" type="submit">-</button>

{% endfor %}

<br/>
<br/>
<p1 class="subheading">Total calories burned today: {{ totalcalburn }}</p1>

</form>
```

This was a simple task, if the user inputs an exercise, insert it into the database.

```
if not burninput:
    pass
elif 0 < burninput < 9999:
    cursor.execute("INSERT INTO history (cal_burned, date, user_id) VALUES (%s, %s, %s)", (burninput, today, id,)
    db.commit()
```

Creating the feature of logging the calories a user has burned was very simple as I had already done a similar thing to this when designing a method to let the user delete meals.

```
if deleteburn:
    cursor.execute("SELECT cal_burned FROM history WHERE user_id = %s AND date = %s", (id, today))
    burncheck1 = cursor.fetchone()[0]
    if burncheck1:
        cursor.execute("DELETE FROM history WHERE cal_burned = %s AND date = %s AND user_id = %s", (deleteburn, today, id,))
        db.commit()
    else:
        pass
```

```
def totalburn(caloriesburned):
    total = 0
    for i in caloriesburned:
        total = total+i[0]
    return total
```

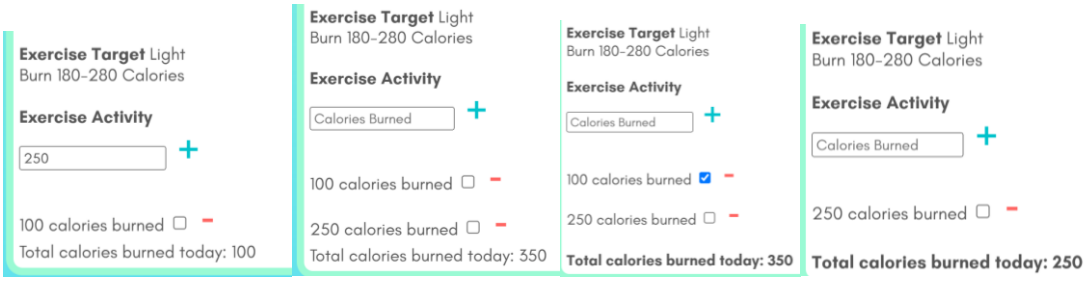
Caloriesburned is a tuple that stores all the fetched values from the SQL database. The for loop isolates the values in the tuple so I can perform operations on them, such as adding them together.

I also used similar code to design a function that deletes an exercise activity and I used a function similar to calculating the total calories in a meal plan – I used it to calculate the total calories burned from all data points in the history table.

Test: Input calories burned, delete calories burned.

Data: 100, 280.

Justification: This is to test if the database and app can adapt and display any changes to the history table.



I was able to log exercise activities and then delete them. Even the total calories burned counter was changed accordingly – the app would adapt to any changes.

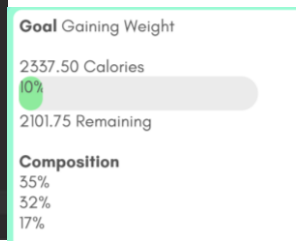
Aim: Display the average composition of the meal plan to the webpage.

Justification: This is a specification point from my success criteria.

```
def nutrition(plan):
    carbs = 0
    protein = 0
    fat = 0
    totalamount = 0
    for i in plan:
        carbs = carbs + i[2]
        protein = protein + i[3]
        fat = fat + i[4]
        totalamount = totalamount + i[5]
    if carbs == 0:
        carbspercent = 0
    else:
        carbspercent = round((carbs/totalamount) * 100)
    if protein == 0:
        proteinpercent = 0
    else:
        proteinpercent = round((protein/totalamount) * 100)
    if fat == 0:
        fatpercent = 0
    else:
        fatpercent = round((fat/totalamount) * 100)

    return (carbspercent, proteinpercent, fatpercent,)
```

The plan variable contains a list of the user's meal plan along with the nutritional values of each meal.

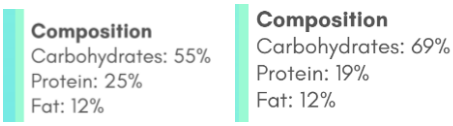
I used python's built-in round function which rounds the calculation result to the nearest integer, makes it easier to read for the user.



I used a function in the python file which has been rendered to the html page. The function calculates the average carbs, protein and fat amounts of all foods in the meal plan and displays them. This makes it easier to create a pie chart by using these values.

Test: Generate a new meal plan

Justification: This is to test if the latest feature (meal composition percentages) can render successfully.



This was a successful test result.

Aim: Create a pie chart for the average composition and the individual meals.

Justification: This is a specification point from my success criteria.

I created a very basic pie chart purely with CSS in similar fashion to designing the progress bar.

I used the variables that displayed the three percentages (carbs, protein and fat) in CSS code so that the pie chart will adapt to any changes to the variables – such as a newly generated meal plan.

I repeated the process so that the page would display the composition for the individual foods.

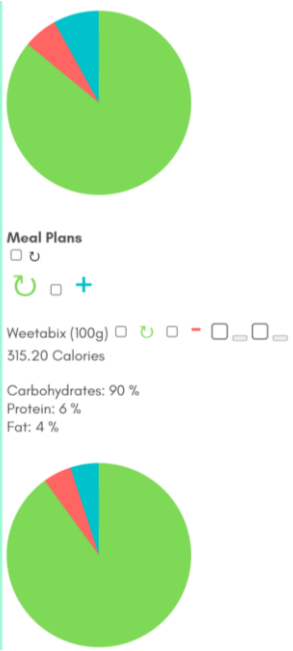
```
TypeError: unsupported operand type(s) for *: 'decimal.Decimal' and 'float'

Traceback (most recent call last)

File "C:\Users\User\AppData\Local\Programs\Python\Python37\lib\site-packages\flask\app.py", line 2091, in __call__
  return self.wsgi_app(environ, start_response)
File "C:\Users\User\AppData\Local\Programs\Python\Python37\lib\site-packages\flask\app.py", line 2076, in wsgi_app
  response = self.handle_exception(e)
File "C:\Users\User\AppData\Local\Programs\Python\Python37\lib\site-packages\flask\app.py", line 2073, in wsgi_app
  response = self.full_dispatch_request()
File "C:\Users\User\AppData\Local\Programs\Python\Python37\lib\site-packages\flask\app.py", line 1518, in full_dispatch_request
  rv = self.handle_user_exception(e)
File "C:\Users\User\AppData\Local\Programs\Python\Python37\lib\site-packages\flask\app.py", line 1516, in full_dispatch_request
  rv = self.dispatch_request()
File "C:\Users\User\AppData\Local\Programs\Python\Python37\lib\site-packages\flask\app.py", line 1502, in dispatch_request
  return self.ensure_sync(self.view_functions[rule.endpoint])(**req.view_args)
File "C:\Users\User\Desktop\Health App\Health App - Coursework\website\homepage.py", line 282, in home
  return render_template("home.html", displaygoal=goal, target=target, remain=remaining, percent=percentage, username=username, foods=plan,
  exercise=exercise, caloriesburned=calburn, totalcalburn=totalcalburn, composition=composition)
```

HTML was trying to declare values as decimal but python was using float.

I tried copying and pasting the code I used for the main pie chart however I received this error.

For each food in the plan, a piechart is created. For the carbs, protein and fat values, carbs and protein all multiplied by 360 and divided by 100 and then displayed using CSS. This is because there are 360 degrees in a circle. I found that leaving fat as 0 in the CSS code signalled that what ever section of the pie chart left over after rendering carbs and protein, will be fat.

```
<div class="piechart" style="background-image: conic-gradient(
#7ED957 {{ ((i[2]*360)/100) }}deg,
#FF6562 0 {{ (((i[2] + i[3])*360)/100) }}deg,
#00C2CB 0);">
</div>

<script>
document.getElementById("{{ i[2] }}").innerHTML = Math.round({{ (i[2]/i[5])*100 }});
document.getElementById("{{ i[3] }}").innerHTML = Math.round({{ (i[3]/i[5])*100 }});
document.getElementById("{{ i[4] }}").innerHTML = Math.round({{ ((i[4]/i[5])*100) }});
</script>
```

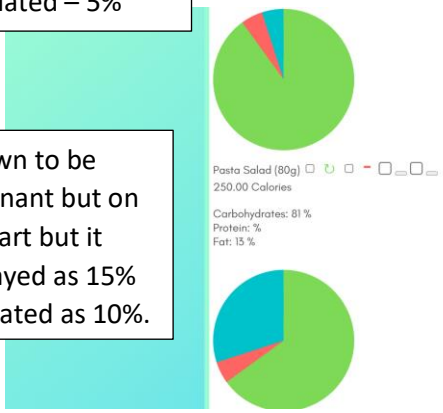


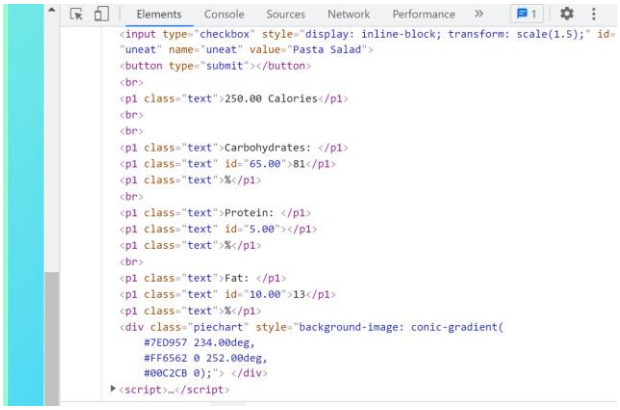
I used basic JavaScript to round the calculations – the issue was python declared values as float whereas HTML recognised it as decimal.

The pie charts were rendered on the webpage.

Protein percentage is blank even though I could see that the value was calculated – 5%

Fat is shown to be very dominant but on the pie chart but it was displayed as 15% and calculated as 10%.





I did encounter an issue with the individual food pie charts. In some cases, the percentages where either blank or of incorrect values and the pie chart would not match the percentage values too. These errors were produced because I was using CSS to display values and render pie charts. CSS is a static language specifically designed for styling. So, displaying varied values and pie charts was out of its comfort zone. Ideally, JavaScript would be more suitable for this.

Given that most of the page's functions were complete, I decided to leave the individual food pie charts for now.

Progress Page

I started off by completing the structure of the page just like I had done for the other pages.

Aim: Allow the user to input their weight for the day

Justification: This is a specification point from my success criteria.

The history table is checked to see if a record already exists with today's date.

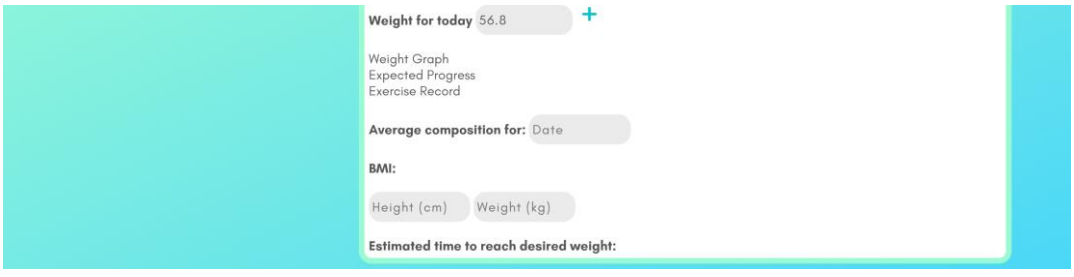
```
weight = request.form.get('weight', type=float)
if 20 < weight < 140:
    cursor.execute("SELECT * FROM history WHERE user_id = %s AND date = %s", (id, today))
    check = cursor.fetchone()
    if check:
        cursor.execute("UPDATE history SET weight = %s WHERE user_id = %s AND date = %s", (weight, id, today))
        db.commit()
    else:
        cursor.execute("INSERT INTO history (weight, user_id, date) VALUES (%s, %s, %s)", (weight, id, today))
        db.commit()
```

If a record exists, update the table otherwise insert a new record into it.

I used very similar code when designing the input exercise feature and so this was quite simple.

Test: Input a weight on the progress page

Justification: This is to test the current prototype and see if it works.



The user did not input an exercise activity for this date so even though this is blank, it is not an issue.

```
USE health;

SELECT * FROM history;
```

history_id	weight	cal_burned	date	user_id
1	56.8	NULL	2022-02-28	1
NULL	NULL	NULL	NULL	NULL

This indicated a successful test result.

Aim: Display average nutritional composition for a month selected by the user.

Justification: This is a specification point on my success criteria.

Test: Check if all records from a specific date can be returned – filter by month.

Justification: This could help retrieve the correct data points for the line graph.

```
USE health;
SELECT * FROM history WHERE monthname(date)='March';
```

history_id	weight	cal_burned	date	user_id
NULL	NULL	NULL	NULL	NULL

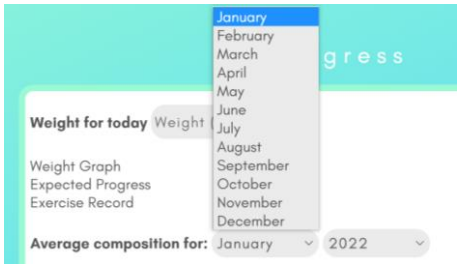
```
USE health;
SELECT * FROM history WHERE monthname(date)='February';
```

history_id	weight	cal_burned	date	user_id
1	56.8	NULL	2022-02-28	1
NULL	NULL	NULL	NULL	NULL

```
USE health;
SELECT * FROM eat WHERE monthname(eatdate)='February';
```

eat_id	food_id	eatdate	eaten	user_id
24	6	2022-02-26	NULL	1
25	8	2022-02-26	NULL	1
26	1	2022-02-26	NULL	1
28	2	2022-02-26	NULL	1
29	4	2022-02-26	NULL	1
37	8	2022-02-27	0	1
38	7	2022-02-27	0	1
39	6	2022-02-27	0	1
40	3	2022-02-27	0	1

This fully established a way to use month names to filter records returned.



I used the same type of input from the questionnaire page – a selection of different options (months).

In terms of the code behind this function, I re-used what I had written for displaying the user's average nutritional intake from the home page – only difference was that I was now filtering the records with a month.

Aim: Produce line graphs that shows the user's progress

Justification: This is a specification point on my success criteria written in the analysis section

Initially, I planned to use a python library called matplotlib but as I was using flask, each graph I would make would have to be stored as a png file in order to be rendered to the html page. This is completely inefficient, as the multiple users can have multiple different graphs and to store each

one as an image to then render to HTML takes up too much storage space. Instead, I decided to use a JavaScript library called Chart.js to create dynamic graphs.

```
USE health;
ALTER TABLE plan ADD column desired DECIMAL(4,1);
SELECT * FROM plan;
```

plan_id	goal	num_meals	exercise	BMR	chicken_pref	fish_pref	redmeat_pref	vegan_pref	dairy_pref	egg_pref	BMI	desired
1	gain	5	light	1837.50	1	1	1	1	1	0	NULL	NULL
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

I realized that in order to check progress, the app needs to know what the user's goal is specifically, and so I added a new column to the plan table which stores the user's desired weight.

```
USE health;
ALTER TABLE plan ADD column current DECIMAL(4,1);
SELECT * FROM plan;
```

plan_id	goal	num_meals	exercise	BMR	chicken_pref	fish_pref	redmeat_pref	vegan_pref	dairy_pref	egg_pref	BMI	desired	current
1	gain	5	light	1831.50	1	1	1	1	1	0	NULL	62.5	NULL
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

```
        cursor.execute("UPDATE plan SET current = %s WHERE plan_id = %s", (weight, id))
        db.commit()
        cursor.execute("UPDATE plan SET desired = %s WHERE plan_id = %s", (dweight, id))
        db.commit()
        return redirect(url_for('homepage.home'))
```

These two statements add the user's desired weight and the user's current weight to the plan.

I made changes to the questionnaire page so that when the user enters their desired weight, it gets added to the database.

```
USE health;
ALTER TABLE plan ADD column plandate DATE;
SELECT * FROM plan;
```

plan_id	goal	num_meals	exercise	BMR	chicken_pref	fish_pref	redmeat_pref	vegan_pref	dairy_pref	egg_pref	BMI	desired	current	plandate
1	gain	5	light	1831.50	1	1	1	1	1	0	NULL	62.5	57.5	NULL
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Adding a date column in the plan table will make things a lot easier when generating the line graph. The graph will be able to display the exact date where the user completed the questionnaire and will also help displaying the "weeks remaining to reach desired weight" feature.

I went about creating a function that will calculate the values that a graph needs in order to show the user's expected progress. By working out how many weeks it will take for the user to reach their desired weight, I used another module called datetime in python.

Test: Use a datetime function to add a number of weeks to a date

Justification: This could help retrieve the correct data points for the line graph.

```
username = str(input('Enter '))
cursor.execute("SELECT user_id FROM user WHERE username = %s", (username,))
id = cursor.fetchone()[0]
cursor.execute("SELECT plandate FROM plan WHERE plan_id = %s", (id,))
first = cursor.fetchone()[0]
print(first)

time = first + datetime.timedelta(weeks=5)
print(time)
```

```
Enter ADMIN
2022-03-01
2022-04-05
```

The datetime.timedelta function allowed me to add weeks to a date.

This was a successful test result.

At a 500-calorie deficit, the average person should lose 0.5kg per week and at a 500-calorie surplus, a person should gain 0.5kg per week. In order to calculate the date for which the user will reach their goal, I understood that I would have to find a way to add weeks or months to a date.

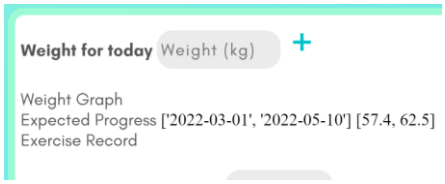
The code checks if the user has values in the desired column and the current column from the plan table.

```
cursor.execute("SELECT desired FROM plan WHERE plan_id = %s", (id,))
desired = cursor.fetchone()
cursor.execute("SELECT current FROM plan WHERE plan_id = %s", (id,))
current = cursor.fetchone()
expectedx = []
expectedy = []
if desired and current:
    amount = 0
    amount = float(desired[0] - current[0])
    if amount < 0:
        amount = amount*-1
        amount = amount/0.5
    else:
        amount = amount/0.5
    weeks = round(amount)
    cursor.execute("SELECT plandate FROM plan WHERE plan_id = %s", (id,))
    first = cursor.fetchone()[0]
    last = first + datetime.timedelta(weeks=weeks)
    expectedx = [str(first), str(last)]
    expectedy = [float(current[0]), float(desired[0])]
```

The difference between the two is calculated and then divided by 0.5 which gives the number of weeks needed to reach the desired weight of the user.

The two lists; expectedx and expectedy are the x and y values for the line graph.

The number of weeks is then added to the date of the user's questionnaire plan – this gives an exact date.



I was able to send the values on the x and y axis of the expected progress graph, to the HTML page. Only two data points are needed here, the date and weight of when the user completed the questionnaire and the date of when the app expects the user to reach their desired goal.

Test: Run a SQL statement to see if records between two dates can be returned.

Justification: This could help retrieve the correct data points for the line graph.

```
USE health;
SELECT * FROM eat WHERE eatdate BETWEEN '2022-02-01'AND '2022-03-01';
```

eat_id	food_id	eatdate	eaten	user_id
24	6	2022-02-26	NULL	1
25	8	2022-02-26	NULL	1
26	1	2022-02-26	NULL	1
28	2	2022-02-26	NULL	1
29	4	2022-02-26	NULL	1
37	8	2022-02-27	0	1
38	7	2022-02-27	0	1
39	6	2022-02-27	0	1

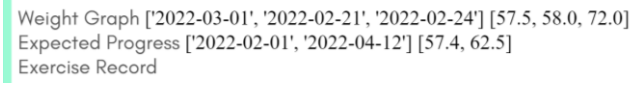
This small test showed me how to select records between dates.

A list called x, containing all the dates that correspond to a weight is returned by the function.

```
def weightgraph(id, today):
    cursor.execute("SELECT current FROM plan WHERE plan_id = %s", (id,))
    firstweight = cursor.fetchone()[0]
    cursor.execute("SELECT plandate FROM plan WHERE plan_id = %s", (id,))
    dayone = cursor.fetchone()[0]
    cursor.execute("SELECT DISTINCT weight, date FROM history WHERE date BETWEEN %s AND %s AND user_id = %s", (dayone, today,
    all = cursor.fetchall()
    x = []
    y = []
    for i in all:
        x.append(float(i[0]))
    for i in all:
        y.append(str(i[1]))
    return (x, y)
```

I added the same SQL statement to select records between dates; the current date and the date of the user completing the questionnaire page.

Y is a list that contains the weight values corresponding to the dates.

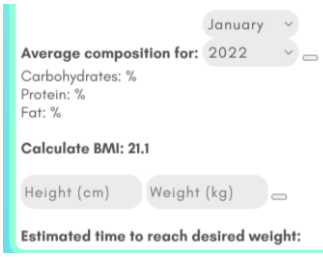


I passed the x and y lists to the HTML page and they were successfully accepted.

```
Weight Graph
['2022-02-21', '2022-02-24'] [58.0, 72.0]
Expected Progress ['2022-02-01', '2022-04-12'] [57.4, 62.5]
Exercise Record [] []
```

I repeated the same process for the exercise record graph values. This is empty because I did not input any exercise activities.

Similar code to the home page, only difference is that records for a specific month are being returned.




```
def average(month, id):
    totalcarb = 0
    totalprotein = 0
    totalfat = 0
    totalamount = 0
    list = []
    cursor.execute("SELECT carbs, protein, fat, amount FROM eat INNER JOIN foodbank ON eat.food_id = foodbank.food_id WHERE
    nutrition = cursor.fetchall()
    if nutrition:
        for i in nutrition:
            totalcarb = totalcarb + i[0]
            totalprotein = totalprotein + i[1]
            totalfat = totalfat + i[2]
            totalamount = totalamount + i[3]
        carbs = round((totalcarb/totalamount)*100)
        protein = round((totalprotein/totalamount)*100)
        fat = round((totalfat/totalamount)*100)
        return (carbs, protein, fat)
    else:
        return list
```

I was also able to retrieve the percentages that need to be displayed on a pie chart by using similar code I wrote before.

Aim: Create a feature that calculates the user's BMI

Justification: This is a specification point on my success criteria written in the analysis section

Writing a comma and then a 1 inside the round function refers to rounding to one decimal place.

```
def BMIcalc(height, weight, id):
    height = (height/100.0)**2.0
    BMI = round(weight/height, 1)
    cursor.execute("UPDATE plan SET BMI = %s WHERE plan_id = %s", (BMI, id))
    db.commit()
    return BMI

def estimated
```

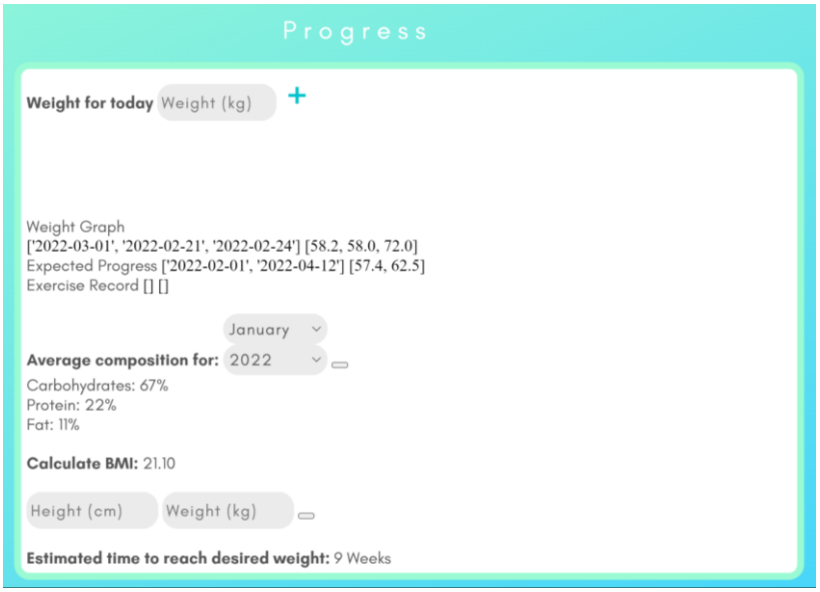
The weight and height variables are the user's inputs from the progress page.

I created a BMI calculator that produces a value to one decimal place.

```
def estimated(id, today):
    cursor.execute("SELECT desired FROM plan WHERE plan_id = %s", (id,))
    desired = cursor.fetchone()[0]
    cursor.execute("SELECT weight FROM history WHERE date = %s AND user_id = %s AND weight IS NOT NULL", (today, id))
    current = cursor.fetchone()
    if not current:
        cursor.execute("SELECT weight FROM history WHERE user_id = %s AND weight IS NOT NULL ORDER BY date DESC LIMIT 1", (id
        current = cursor.fetchone()
        if not current:
            cursor.execute("SELECT current FROM plan WHERE plan_id = %s AND current IS NOT NULL",(id,))
            current = cursor.fetchone()[0]
        else:
            current = current[0]
        if current:
            diff = desired - current
            if diff < 0:
                diff = diff*-1
            weeks = round(float(diff)/0.5)
            if weeks == 0:
                weeks = ("Your have reached your goal")
                return weeks
            else:
                return weeks
        elif not current:
            weeks = ("Enter your current weight")
            return weeks
    else:
        diff = desired - current[0]
        if diff < 0:
            diff = diff*-1
        weeks = round(float(diff)/0.5)
        if weeks == 0:
            weeks = ("Your have reached your goal")
            return weeks
        else:
            return weeks
```

This code was a modified version of the expected progress line graph. The user's desired weight would be subtracted from the user's current weight. The current weight will be searched for in the

history table but if there is no record in that table, the current weight value will be taken from the plan table.



The estimated time was calculated during the generation of the expected progress graph values.

I managed to complete the feature so that the app can calculate the estimated time to reach the user's desired goal. Although the line graphs and pie charts have not been rendered, the values that they need to use have been generated.

Calendar Page

Aim: Display all days in a month chosen by the user

Justification: This is a specification point on my success criteria written in the analysis section

I used jinja template provided by flask, to inherit the home page that I designed before.

Test: Use python datetime to display days

Justification: This is to see if I can get a working solution to be used in the main feature on the calendar page.

Days is a list that stores the date of all days in a given month. For each day in the range. All months start with one and the end will be the last day of the month, stored in num_days.

```
import mysql.connector
import datetime, calendar
year = 2022
month = 2
num_days = calendar.monthrange(year, month)[1]
days = [datetime.date(year, month, day) for day in range(1, num_days+1)]
print(days)
```

Num_days stores the last day in a month – if February was the month, num_days will store "28".

```
6), datetime.date(2020, 2, 7), datetime.date(2020, 2, 8), datetime.date(2020, 2, 9), datetime.date(2020, 2, 10),
11), datetime.date(2020, 2, 12), datetime.date(2020, 2, 13), datetime.date(2020, 2, 14), datetime.date(2020, 2, 15),
datetime.date(2020, 2, 16), datetime.date(2020, 2, 17), datetime.date(2020, 2, 18), datetime.date(2020, 2, 19), datetime.date(2020, 2, 20),
datetime.date(2020, 2, 21), datetime.date(2020, 2, 22), datetime.date(2020, 2, 23), datetime.date(2020, 2, 24), datetime.date(2020, 2, 25),
datetime.date(2020, 2, 26), datetime.date(2020, 2, 27), datetime.date(2020, 2, 28), datetime.date(2020, 2, 29)]
```

utilizing the datetime module part of python to generate all the days in a month, depending on which month selected by the user.

The user's input from the webpage will be stored in the year variable and month variable – both use flask request.form to get the data.

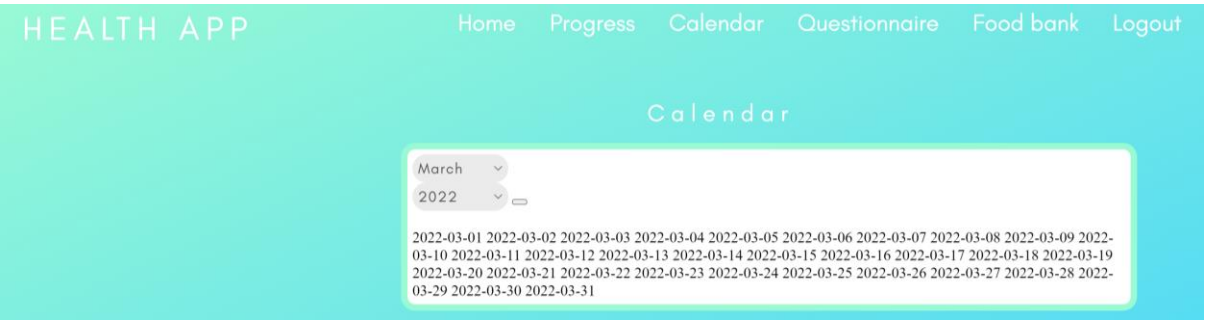
```
@foodbankpage.route('/calendar', methods=['GET', 'POST'])
def calendarpage():
    if "username" in session:
        username = session["username"]
        username = str(username)
        cursor.execute("SELECT user_id FROM user WHERE username = %s", (username,))
        id = cursor.fetchone()[0]
        days = []
        if request.method == 'POST':
            year = request.form.get('year', type=int)
            month = request.form.get('month', type=int)
            num_days = calendar.monthrange(year, month)[1]
            days = [datetime.date(year, month, day) for day in range(1, num_days + 1)]
        return render_template("calendar.html", days=days,)
    else:
        return redirect(url_for('loginsystem.login'))
```

I integrated the code into the python webpage file but I made sure the user's input would be taken into account.

Test: Display all the days in a month

Data: March, 2022

Justification: This is to see if I can get a working solution to be used in the main feature on the calendar page.

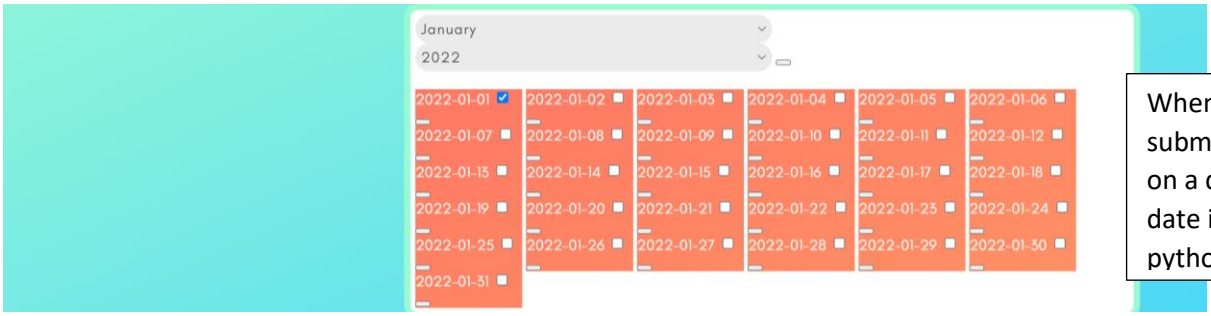


I added some basic CSS styling to make this more presentable for the user.

Test: Add a meal plan for a specific date.

Data: January, 2022, 2022-01-01

Justification: This is to see if I can get a working solution for this feature.



When the user submits a button on a date, the date is sent to python.

```
<input type="checkbox" style="display: inline-block;" id="new" name="new" value="{{ i }}">
<button type="submit"></button>
</input>
```

If the user has already generated a meal plan for the given date, it will be deleted and a new one will be inserted.

```
new = request.form.get('new')
if new:
    cursor.execute("SELECT food_id, eatdate, user_id FROM eat WHERE eatdate = %s AND user_id = %s AND eatdate IS T
    check = cursor.fetchall()
    if check:
        cursor.execute("DELETE FROM eat WHERE eatdate = %s AND user_id = %s", (new, id))
        db.commit()
        foods = generate(column, foodno)
        foodplan = regen(foods, target, column, foodno)
        for i in foodplan:
            cursor.execute("INSERT INTO eat (food_id, eatdate, eaten, user_id) VALUES (%s, %s, %s, %s)", (i[0], ne
            db.commit()

    else:
        foods = generate(column, foodno)
        foodplan = regen(foods, target, column, foodno)
        for i in foodplan:
            cursor.execute("INSERT INTO eat (food_id, eatdate, eaten, user_id) VALUES (%s, %s, %s, %s)", (i[0], ne
            db.commit()
```

I added a check box along with a button displayed for each date box. When the user clicks on this, they will generate a food plan for that date. I used the same code from the homepage as this was almost the same feature, only difference is that there was now a date that had to be factored.

```
1  USE health;
2
3  SELECT * FROM eat;
```

eat_id	food_id	eatdate	eaten	user_id
39	6	2022-02-27	0	1
40	3	2022-02-27	0	1
41	2	2022-02-27	0	1
58	3	2022-02-28	NULL	1
63	7	2022-02-28	NULL	1
69	8	2022-01-01	0	1
70	2	2022-01-01	0	1
71	3	2022-01-01	0	1
72	6	2022-01-01	0	1
73	4	2022-01-01	0	1

I ran this query which showed that the feature works.

Aim: If there is already a meal plan that exists on a specific date, signal an icon to the user.

Justification: The user may not know if they have already generated a meal on a date already, there has to be a way to give send the message to the user.

For each day in a given month, check if there is a meal plan generated for that day. If there is, that date will be added to a list called signal.

```
days = [datetime.date(year, month, day) for day in range(1, num_days + 1)]
for i in days:
    day = str(i)
    cursor.execute("SELECT eatdate FROM eat WHERE eatdate = %s AND user_id = %s LIMIT 1", (day, id,))
    present = cursor.fetchone()
    if present:
        signal.append(i)
```

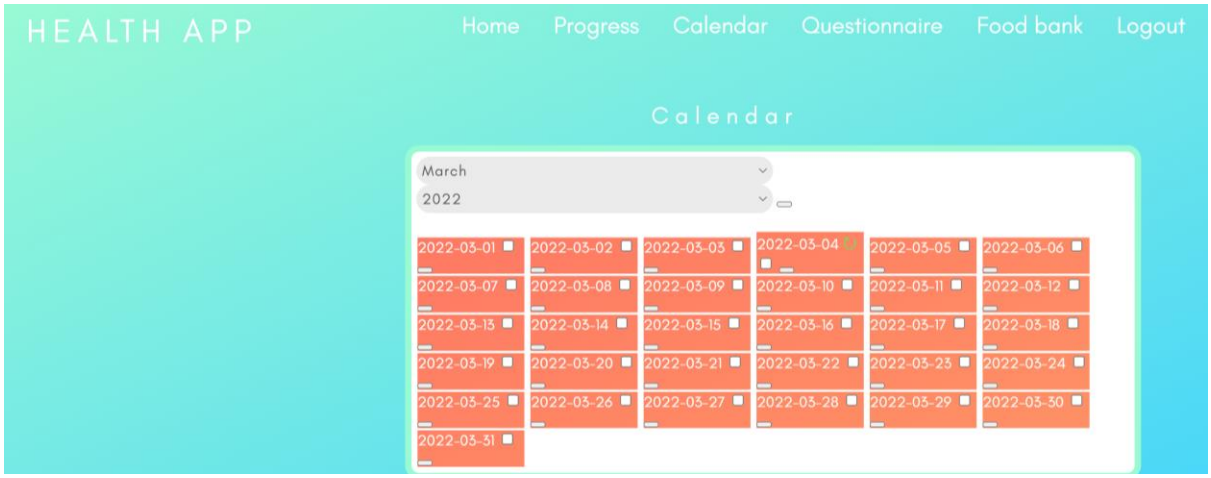
```
{% for i in days %}
<div class="datebox" style="display: inline-block">
<p1 class="text" style="color: white">{{ i }}</p1>

    {% for a in signal %}
        {% if a == i %}
        <p1 style="font-size: 20px; color: #7ED957;">&#8635</p1>
        {% endif %}
    {% endfor %}


<input type="checkbox" style="display: inline-block;" id="new" name="new" value="{{ i }}">
<button type="submit"></button>
</input>
</div>
```

I added a for loop with an if statement embedded inside it so that each day is checked to see if it appears in the signal list. If it does, an icon will be generated to show a meal plan already exists.

Test: Add a meal plan for a specific date.

Data: March, 2022, 2022-03-04

Justification: This is to see if I can get a working solution for this feature.



This icon was produced and so the user can now see easily where a food plan has been generated. Finally, the calendar page was concluded.

Food Bank Page

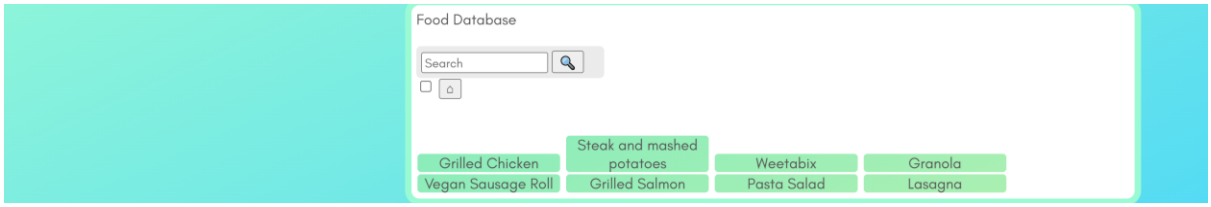
Aim: Create a search system on the food bank page, allowing the user to search for foods.

Justification: This is a specification point on my success criteria.

Similar to the questionnaire page, I added an input box with a submit button.

```
cursor.execute("SELECT food FROM foodbank")
foods = cursor.fetchall()
```

I also added a simple SQL query that returns a list of all foods in the foodbank table.



Upon load, the page displayed all foods stored in the food database.

Test: See if the search feature works.

Data: granola.

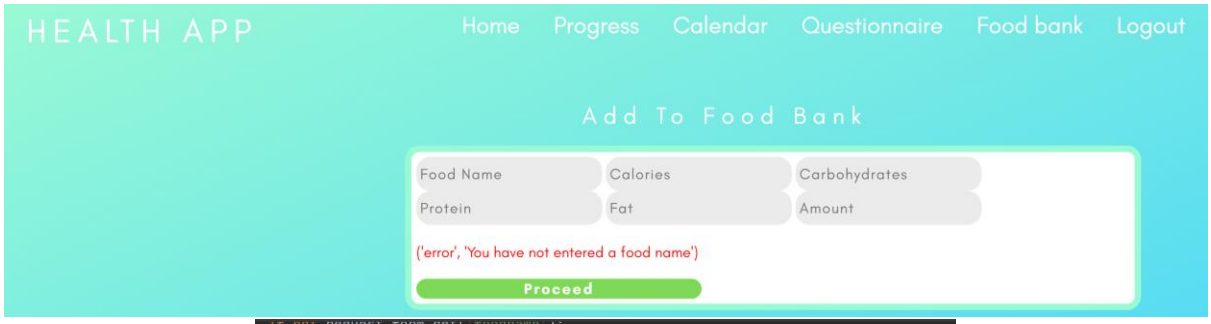
Justification: This is to see if I can get a working solution for this feature.

The correct result was returned successfully. I added a plus icon which symbolizes adding food to the database.

Aim: Create a feature that allows the user to add foods to the food bank.

Justification: This is a specification point on my success criteria.

I realized that this feature could not be placed on the same page because two forms cannot exist on one page. So, I created another page and used a hyperlink on the plus icon which directs the user from the food bank page to the new one.




```
    flash('You have not entered a food name', category='error')
elif len(foodname) > 80 or len(foodname) < 4:
    flash('Food name is an invalid length', category='error')
elif any(i.isdigit() for i in str(foodname)):
    flash('Food name cannot contain any numbers', category='error')
elif not request.form.get('calories'):
    flash('You have not entered calories', category='error')
elif calories > 2000 or calories < 40:
    flash('Invalid number of calories', category='error')
elif not request.form.get('amount'):
    flash('You have not entered an amount for the food', category='error')
elif amount < 30 or amount > 500:
    flash('Invalid food amount', category='error')
elif not request.form.get('carbs'):
    flash('You have not entered carbs', category='error')
elif carbs > amount or protein > amount or fat > amount:
    flash('Nutritional value exceeds amount')
elif not request.form.get('protein'):
    flash('You have not entered protein', category='error')
elif not request.form.get('fat'):
    flash('You have not entered fat', category='error')
elif (carbs + protein + fat) != amount:
    flash('Nutritional values do not add up to the food amount', category='error')
```

If all these conditions were met, the user's inputs would be inserted into the database.

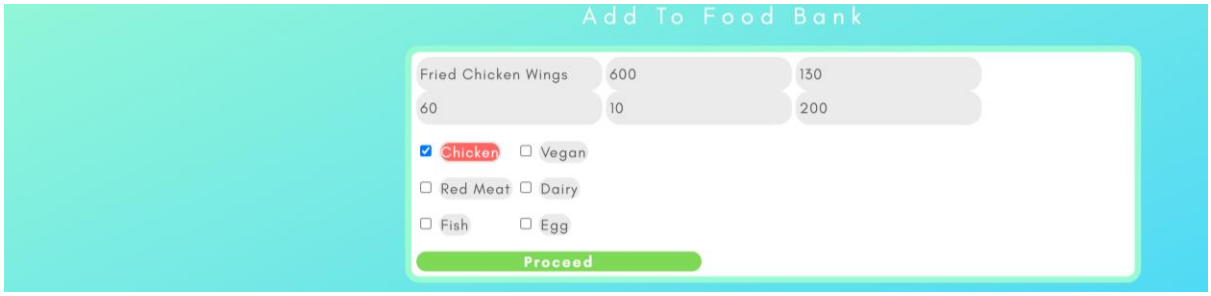
```
cursor.execute("INSERT INTO foodbank (food, calories, carbs, protein, fat, amount, chicken, fish, redmeat, vegan, dairy, e
db.commit()
```

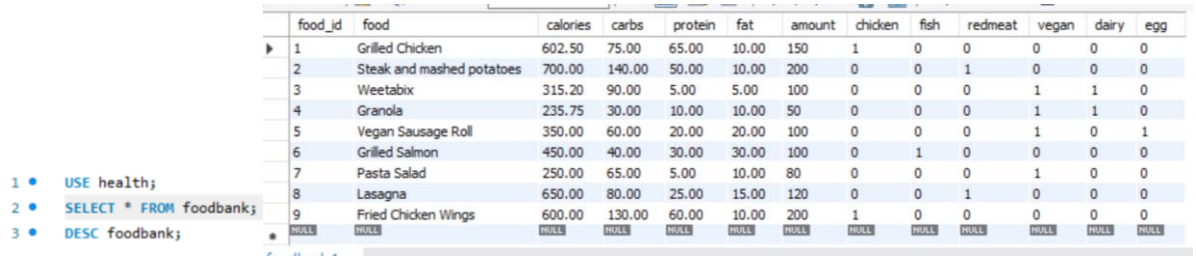
Given that I had already created the questionnaire page, I used similar methods for the validation on this page.

Test: Add food to the database by filling out the form on the new webpage.

Data: Fried Chicken Wings, 600, 130, 60, 10, 200, chicken

Justification: This is to see if I can get a working solution for this feature.

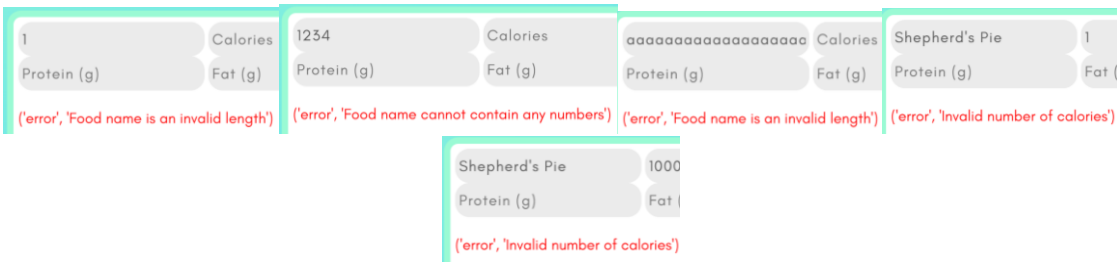




```
USE health;
SELECT * FROM foodbank;
DESC foodbank;
```

food_id	food	calories	carbs	protein	fat	amount	chicken	fish	redmeat	vegan	dairy	egg
1	Grilled Chicken	602.50	75.00	65.00	10.00	150	1	0	0	0	0	0
2	Steak and mashed potatoes	700.00	140.00	50.00	10.00	200	0	0	1	0	0	0
3	Weetabix	315.20	90.00	5.00	5.00	100	0	0	0	1	1	0
4	Granola	235.75	30.00	10.00	10.00	50	0	0	0	1	1	0
5	Vegan Sausage Roll	350.00	60.00	20.00	20.00	100	0	0	0	1	0	1
6	Grilled Salmon	450.00	40.00	30.00	30.00	100	0	1	0	0	0	0
7	Pasta Salad	250.00	65.00	5.00	10.00	80	0	0	0	1	0	0
8	Lasagna	650.00	80.00	25.00	15.00	120	0	0	1	0	0	0
9	Fried Chicken Wings	600.00	130.00	60.00	10.00	200	1	0	0	0	0	0
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

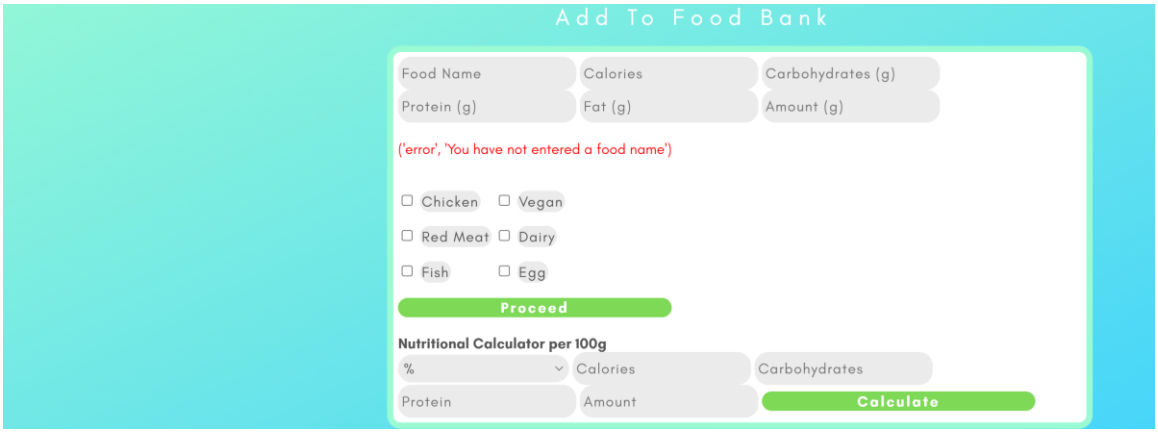
This was successful and so the user could add food to the database.



I performed a range of invalid tests to see if the correct response could be returned in each unique scenario.

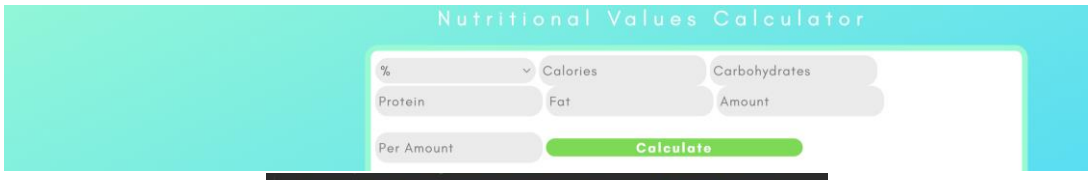
Aim: Create a feature that allows the user to calculate nutritional values based on their inputs.

Justification: This is a specification point on my success criteria.



I initially added the inputs for the calculator on the same page however there was an issue with this. When I clicked calculate, the app would process the input values part of the other section. I worked around this by creating another route that will link to yet another web page.

I wrote a calculator icon that is actually a link to the new page.



Each value inputted by the user is checked against a set of rules to validate if they are suitable – by using if and elif statements.

```
    flash("You have not completed all inputs", category='error')
elif not request.form.get('calc1'):
    flash("You have not completed all inputs", category='error')
elif calories > 2000 or calories < 20:
    flash("Invalid calorie value", category='error')
elif amount > 500 or amount < 20:
    flash("Invalid amount value", category='error')
elif not request.form.get('calc2'):
    flash("You have not completed all inputs", category='error')
elif carbs > amount or carbs < 0:
    flash("Invalid carbs amount", category='error')
elif not request.form.get('calc3'):
    flash("You have not completed all inputs", category='error')
elif protein > amount or protein < 0:
    flash("Invalid protein amount", category='error')
elif not request.form.get('calc4'):
    flash("You have not completed all inputs", category='error')
elif fat > amount or fat < 0:
    flash('Invalid fat amount', category='error')
```

Per amount is the amount in grams that the user wants to calculate for. Amount is the amount they are using correspondent to the values.

This calculation will be carried out if the user has chosen % as their units.

```
else:
    carbs = carbs/100
    carbs = round(carbs*per)
    list.append(carbs)
    protein = protein/100
    protein = round(protein*per)
    list.append(protein)
    fat = fat/100
    fat = round(fat*per)
    list.append(fat)
    list.append(per)
```

```
else:
    carbs = carbs/amount
    carbs = round(carbs*per)
    list.append(carbs)
    protein = protein/amount
    protein = round(protein*per)
    list.append(protein)
    fat = fat/amount
    fat = round(fat*per)
    list.append(fat)
    list.append(per)
```

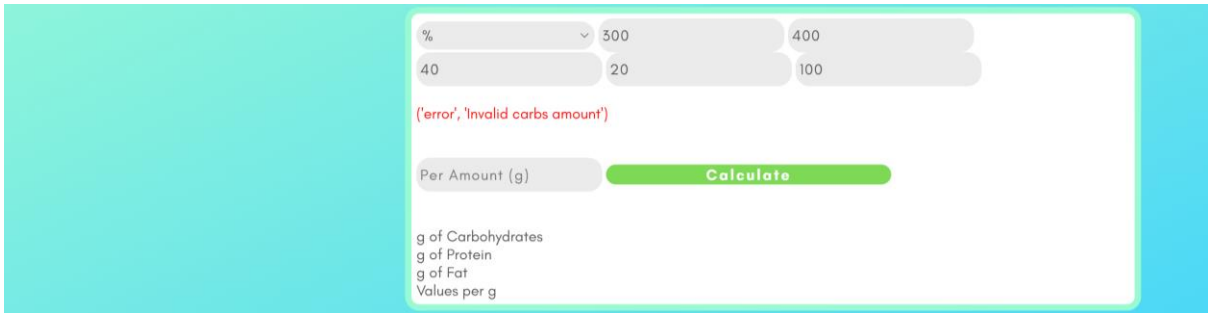
This will be carried out if the user has chosen (g) as their units.

At the end of each calculation, a list of the results will be generated and passed to the HTML page.

Test: Calculate nutritional values.

Data: %, 300, 400, 40, 20, 100.

Justification: This is to see the correct response will be returned when receiving invalid data.

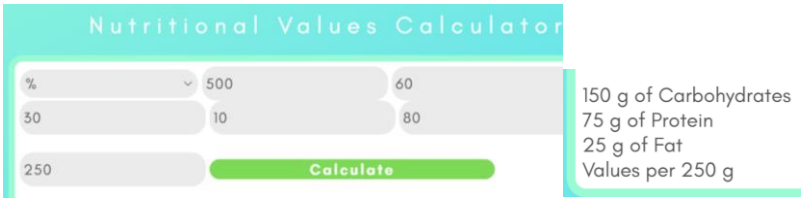


The error was successfully generated.

Test: Calculate nutritional values.

Data: %, 500, 60, 30, 10, 80, 250.

Justification: This is to see if the results can be calculated from the valid data received.

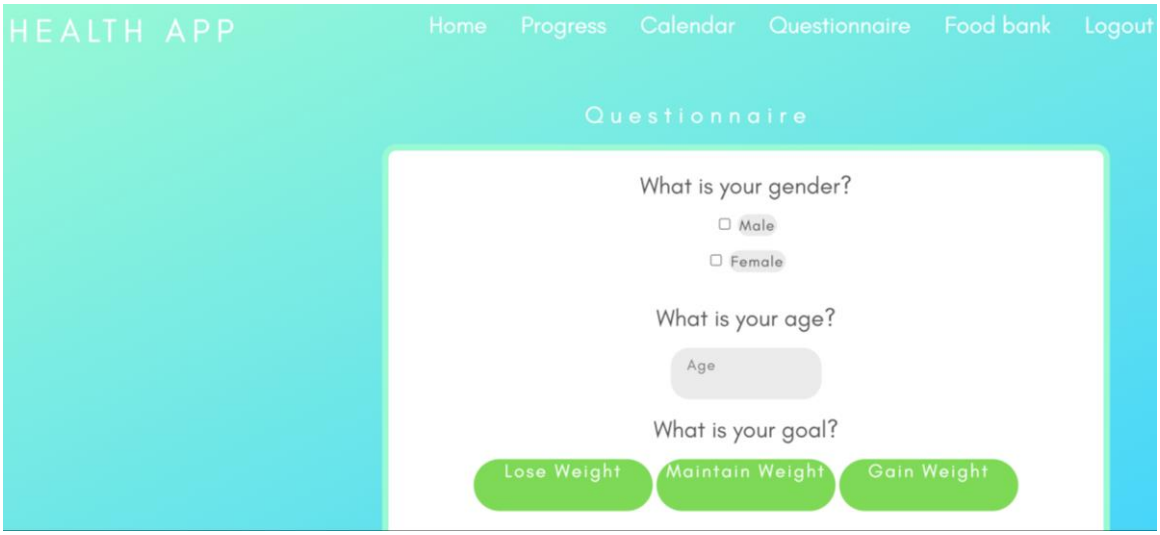


This was successful as the app also took into account of the amount the user was calculating values for – in this case, per 250g.

I concluded this page by adding a home icon which redirects the user to the foodbank page when clicked. This was an ordinary hyperlink to the foodbank page and just provides a quick route to it.

Modifications

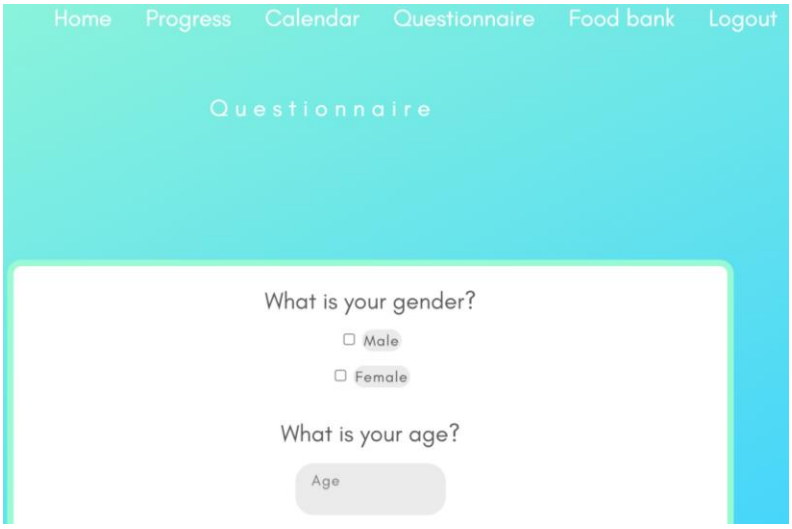
Questionnaire



Modification: Change the styling of the questionnaire page so that it follows the uniform of home page and progress page etc.

Justification: When the user is on the questionnaire page, it will make it easier for the user to navigate from here, to other pages – this is why I decided to make this change.

I made adjustments to the questionnaire by applying the same layout I have been using for the other pages. I did this by changing the jinja template code in the questionnaire HTML file and made it inherit the home page instead.

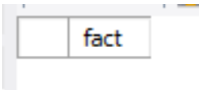


The new home page was rendered successfully.

Home

Modification: Add a fact box that randomly generates a fact from the database and displays it on all pages apart from the login/create account page.

Justification: This was a specification point from my success criteria.



I then decided to create a new table in my database that does not reference any other table. This table is used to store facts that I will enter into it. I found a paragraph from a trusted health website – the NHS and inserted it into the database.

SQL statement returns a random record from the fact table.

```
cursor.execute("SELECT fact FROM factbox ORDER BY RAND()LIMIT 1")
fact = cursor.fetchone()[0]
```

This fact box was rendered on all pages except for the login page and create account page which was intentional.

Modification: Change the water intake feature so that each time the user clicks the add button, a new glass icon is generated.

Justification: I had not finished the water intake feature from before.

When the user clicks the + button, this JavaScript function is run.

```
<script>
function addwater() {
  var drink = document.getElementById("water").innerHTML
  document.getElementById("water").innerHTML = "🍷" + drink;
}
</script>
```

The variable "drink" contains what is written in the tag with the id "water".

The glass icon will be added to the current contents of the tag.

When I clicked the + button, a new glass was added each time.

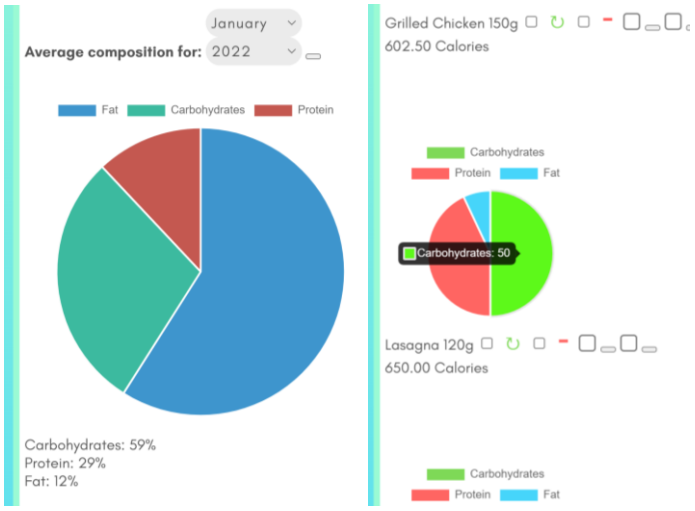
Modification: Use Chart.js to create pie charts rather than pure CSS.

Justification: When using pure CSS, the values of the charts and the charts themselves could not render properly, so to fix this, JavaScript can be used.

I used the same code as I did for the graphs for the progress page, only difference is there was one set of data and the type is declared is pie rather than line.

```
<script>
new Chart(document.getElementById("main"), {
    type: 'pie',
    data: {
      labels: ["Carbohydrates", "Protein", "Fat"],
      datasets: [{
        backgroundColor: ["#7ed957","#FF6562","#46D5FA"],
        data: [{{ composition[0] }},{{ composition[1] }},{{ composition[2] }},]
      }]
    },
    options: {
      title: {
        display: true,
      }
    }
});
</script>
```

See page 118 – I completed the progress graphs first, then the pie chart of the home page.

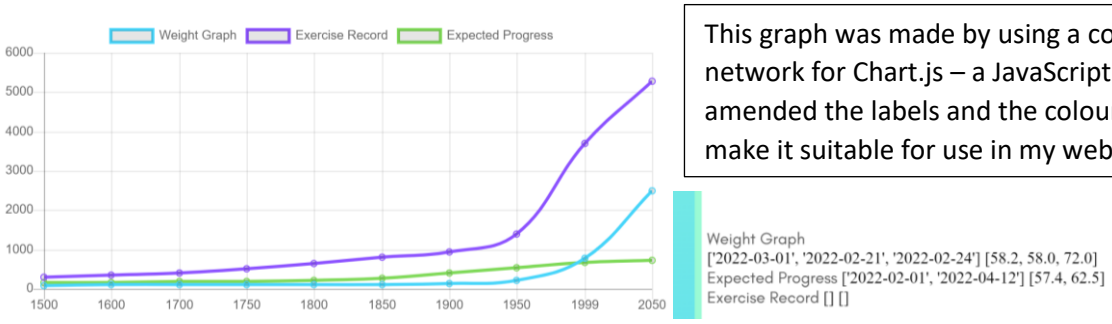


I copied the exact same code and used it for the individual foods on the meal plan.

Progress page

Modification: Add line graphs to the progress page by using the x and y values already generated from before.

Justification: I had not finished this feature from before, it is part of my success criteria.

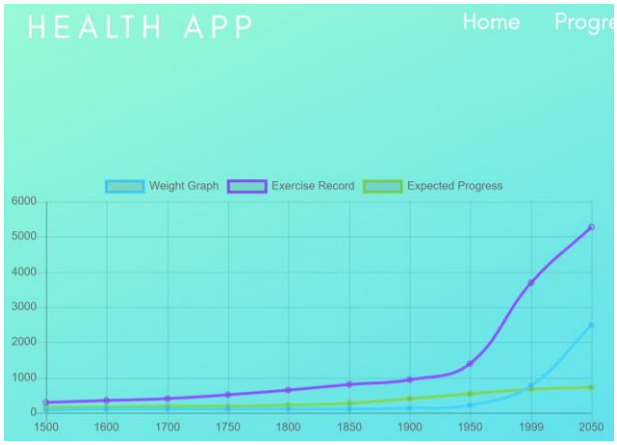


This graph was made by using a content delivery network for Chart.js – a JavaScript library. I amended the labels and the colours of the graph to make it suitable for use in my web app.

I found a line graph online which I could use as a template however when copying and pasting this into a page on my flask app, it could not render.

```
@progresspage.route('/spare', methods=['GET', 'POST'])
def spare():
    if "username" in session:
        username = session["username"]
        username = str(username)
        cursor.execute("SELECT user_id FROM user WHERE username = %s", (username,))
        id = cursor.fetchone()[0]
        return render_template("spare.html")
    else:
        return redirect(url_for('loginsystem.login'))
```

I created a spare html file which I used for the sole purpose of testing to see if the line graph could be rendered on a page that is part of my flask app.



I then copied everything from the progress page and pasted it into the spare html file and then changed the name of the file to progress.

This solved the issue, however I still had to ensure the graph takes in the data points passed from python.

This is the basic configuration for creating graphs in JavaScript, using Chart.js. The values I had calculated before, were now being used in the code – weightx and weighty.

```
<script>
new Chart(document.getElementById("line-chart1"), {
  type: 'line',
  data: {
    labels: {{ weightx | safe }},
    datasets: [{
        data: {{ weighty | safe }},
        label: "Weight Graph",
        borderColor: "#46D5FA",
        fill: false
      },
    ]
  },
  options: {
    title: {
      display: true,
    }
  }
});
</script>
```

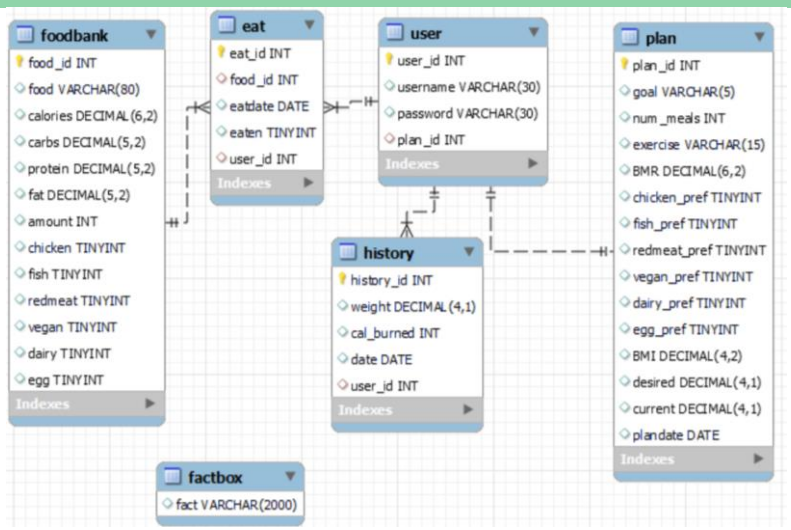
The keyword “safe” written after the variables refers to signalling to jinja template that the variables can be rendered safely – this is a general rule when rendering graphs in a flask app.

Adding | safe to the variables being sent from python solved the issue and so I created three different line graphs – weight, expected progress, exercise record.

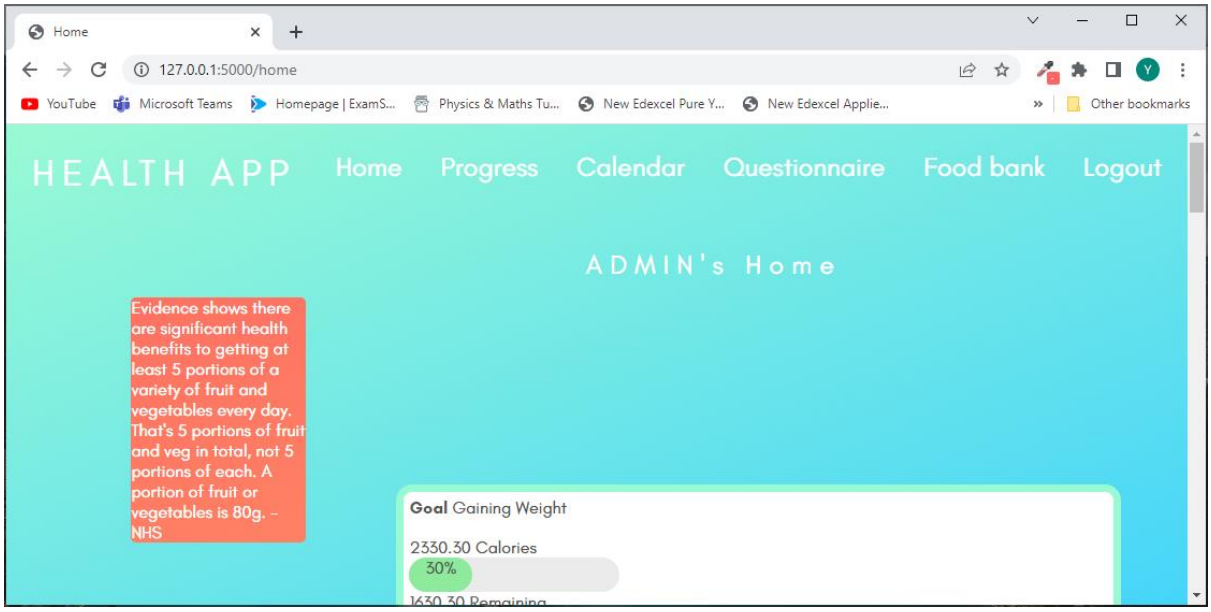
The canvas tags are where the line graphs get rendered on the web page.

```
<div style="width: 50%; height: 50%;">
<canvas id="line-chart1"></canvas>
</div>

<div style="width: 50%; height: 50%;">
<canvas id="line-chart2"></canvas>
</div>

<div style="width: 50%; height: 50%;">
<canvas id="line-chart3"></canvas>
```

Additional Screenshots

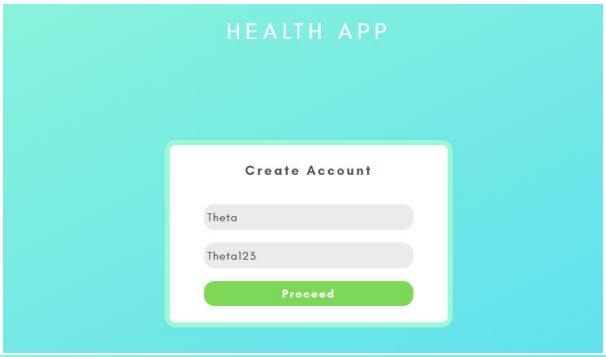


At the end of development, this was the final database model I had created.

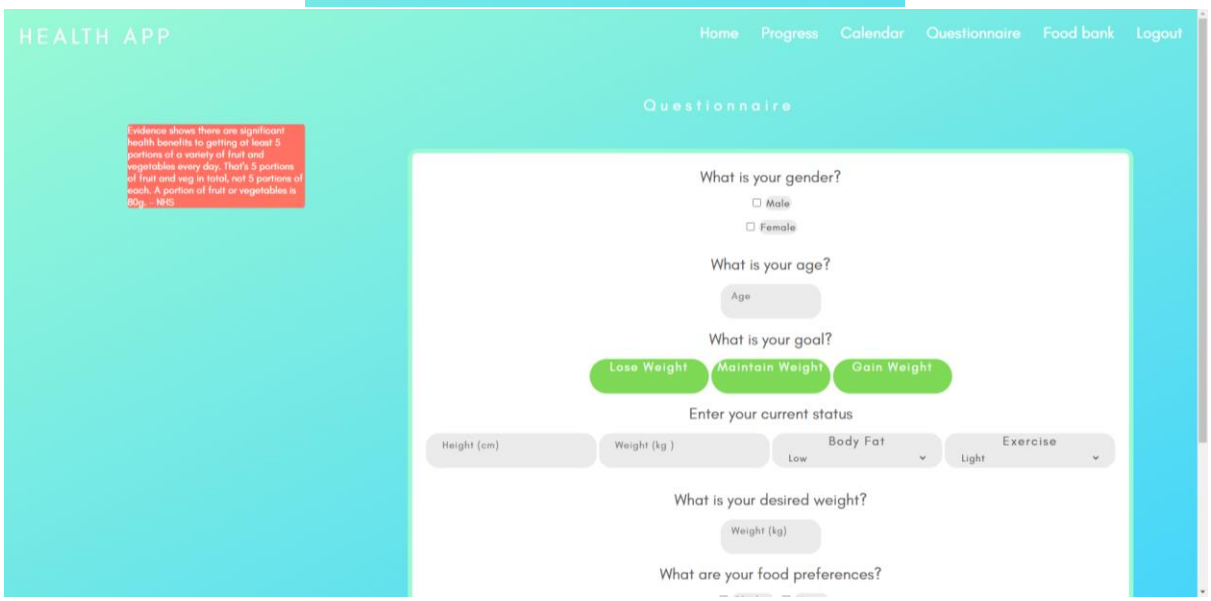


This screenshot shows how the webapp looks in a smaller window rather than the default window size when opening a browser.

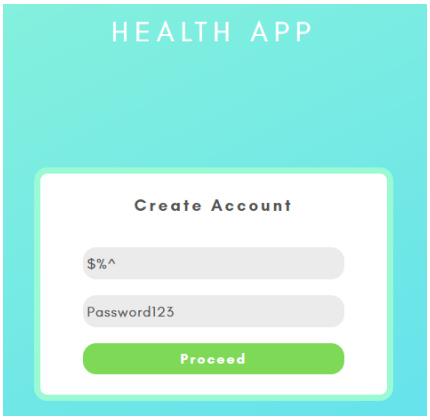
The following screenshots represents evidence of post development tests that were based on the test plan I created in the design section, under the header; "Further data".

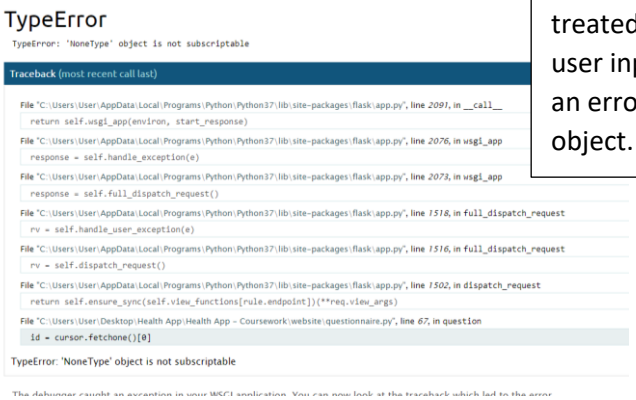


An account was created successfully as I was redirected to the questionnaire page.

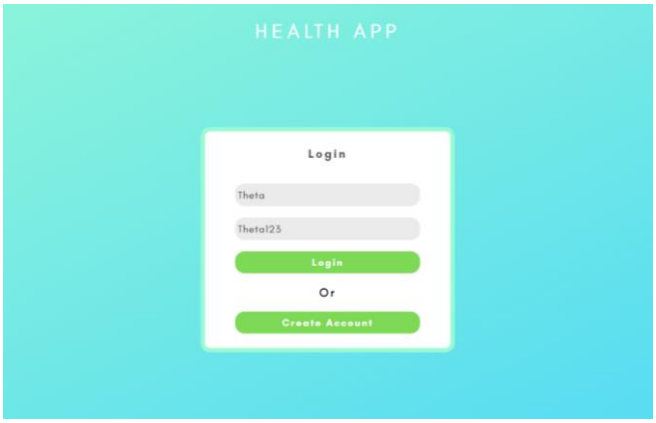


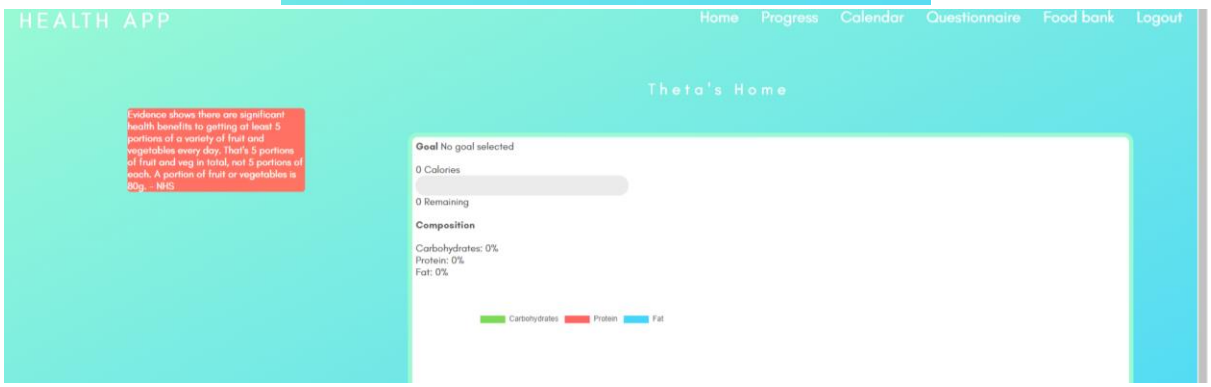
When entering uncommon characters, I found that the web app treated it as an empty user input, resulting in an error of a none-type object.



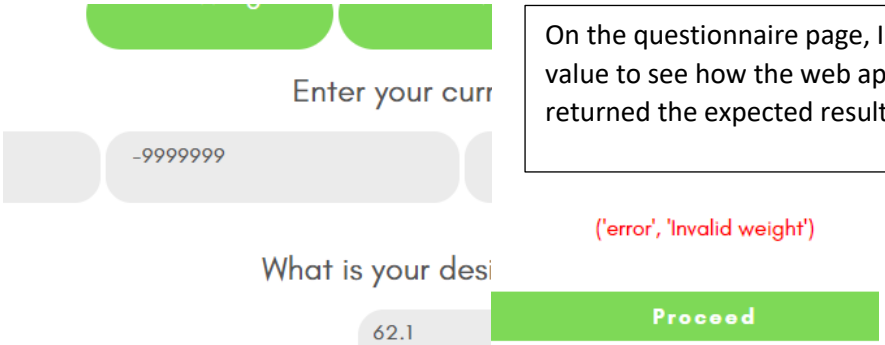


After creating the “Theta ” account, I was redirected to the home page upon logging in with the same details.

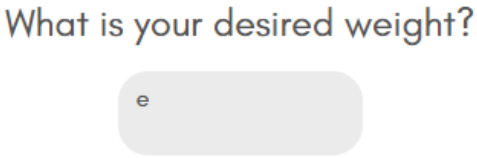


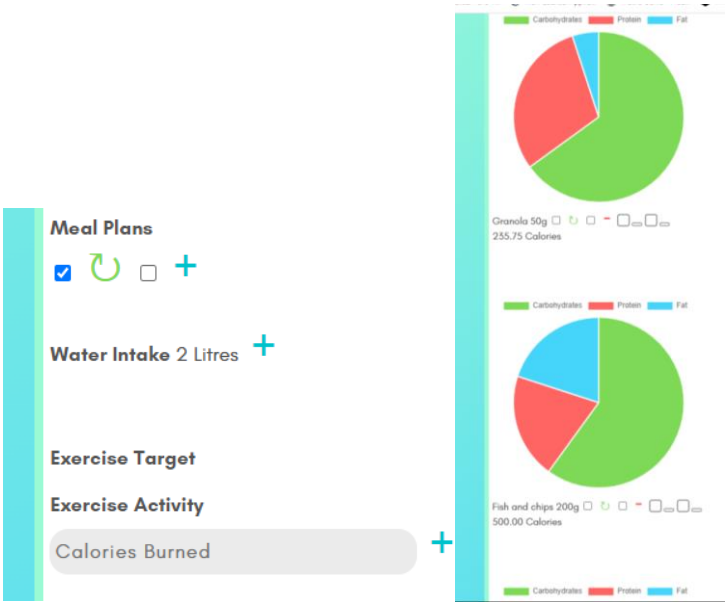


On the questionnaire page, I entered an invalid data value to see how the web app would respond and it returned the expected result of which was an error.



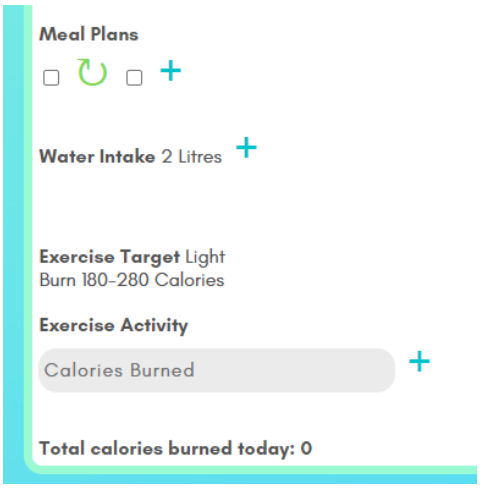
In my test plan, I had the intention of entering “hello” as a form of erroneous data however HTML did not render the input of the characters apart from “e”. Representing a mathematical expression.

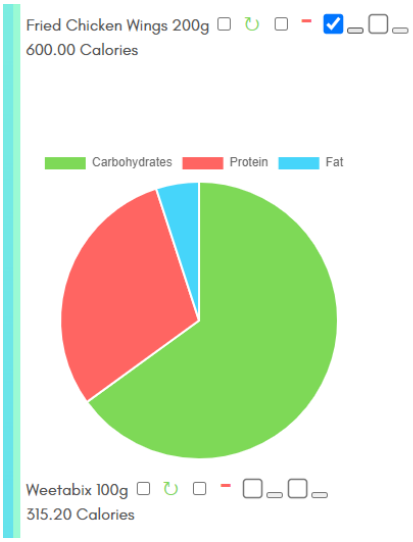




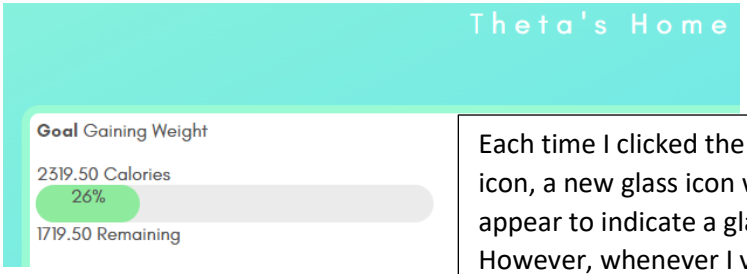
After checking the box and then clicking the green icon, an entire meal plan was generated correctly.

I planned to try deleting a meal when there was an empty plan. However, during the iterative development, I designed this feature in a way that ensures the user can only delete meals if there is a plan generated.



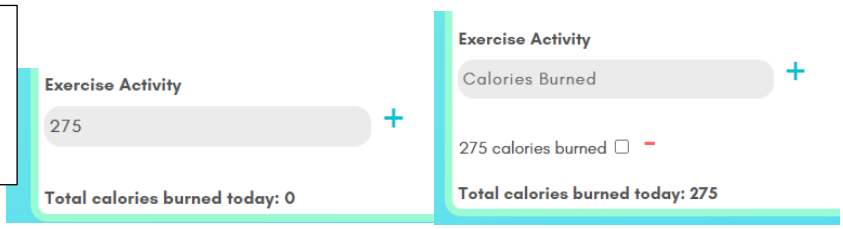


My next test was to check if the web app could mark a selected meal as eaten. After inputting data in the form of a checkbox, followed by a small submit button, the progress bar adjusted accordingly once I refreshed the page manually.



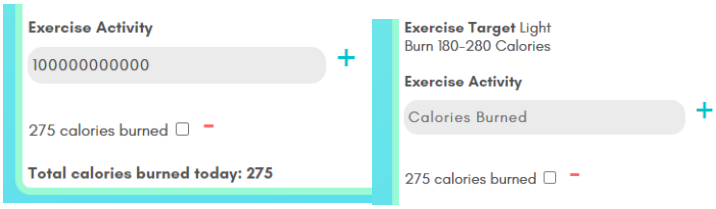
Each time I clicked the blue "plus" icon, a new glass icon would appear to indicate a glass of water. However, whenever I visited another page, the entire icon sequence generated, was cleared.

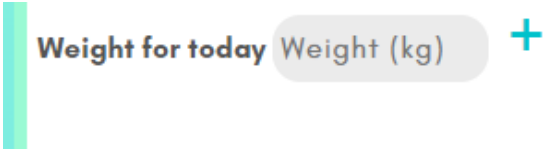
Whilst using valid data, I was able to log exercise activities correctly.



The number of calories entered would be displayed and I could delete it too.

With the use of invalid data, I was able to confirm that the web app did not log data values that were deemed in appropriate/out of boundary.

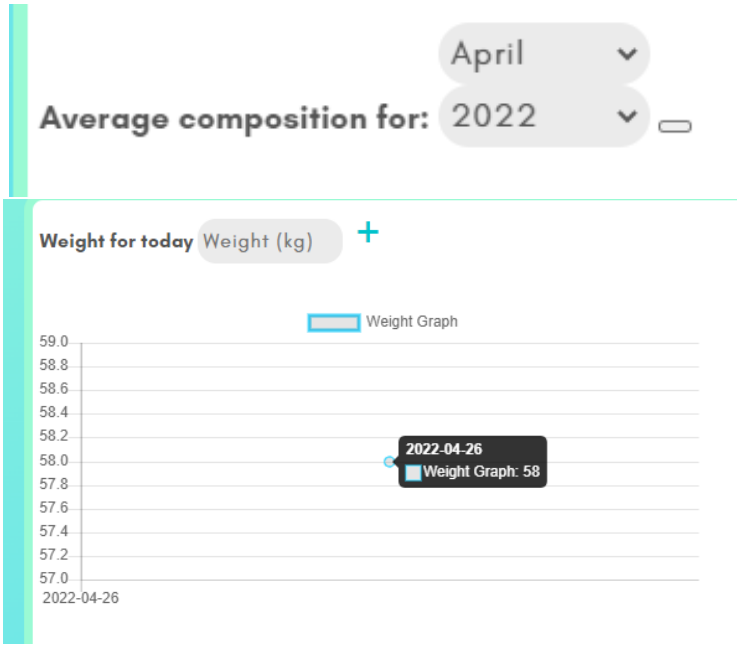




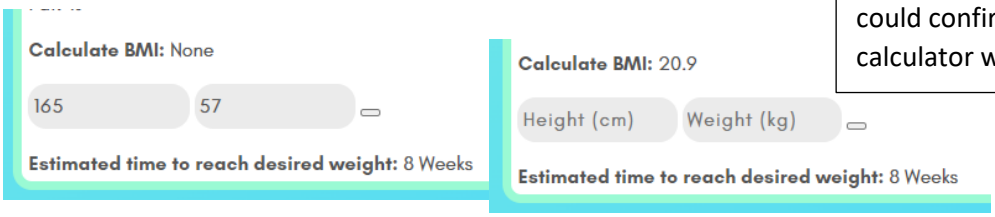
On the progress page, my plan was to enter "££$%" as the current weight to see how the web app would respond to erroneous data being inputted in this feature.

Entering erroneous data was not displayed, HTML did not display exactly what I inputted using my keyboard.

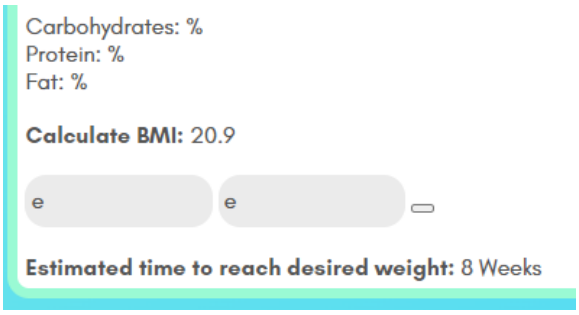
For this test, my intention was to select a month to see if the web app could display the correct points on the graph.



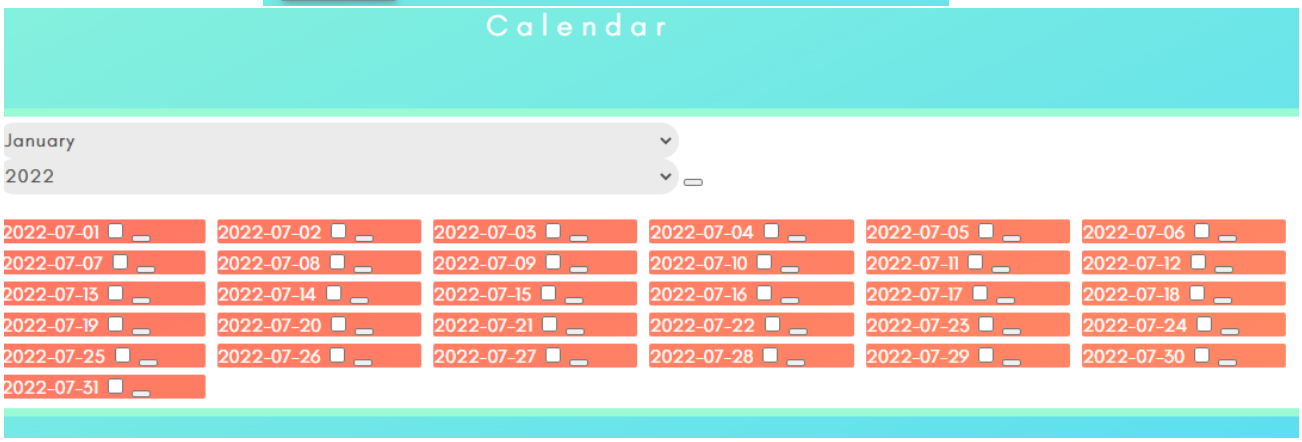
With the use of valid data, I could confirm that the BMI calculator worked successfully.

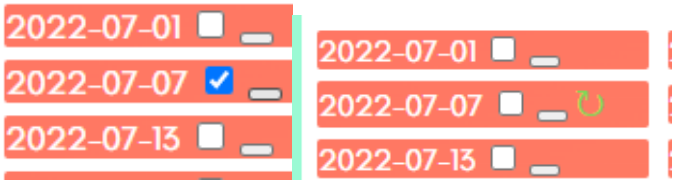


I then wanted to use erroneous data to see what would be the outcome and similarly to previous tests, the inputs would not show on the page inside the text box. Apart from “e”.



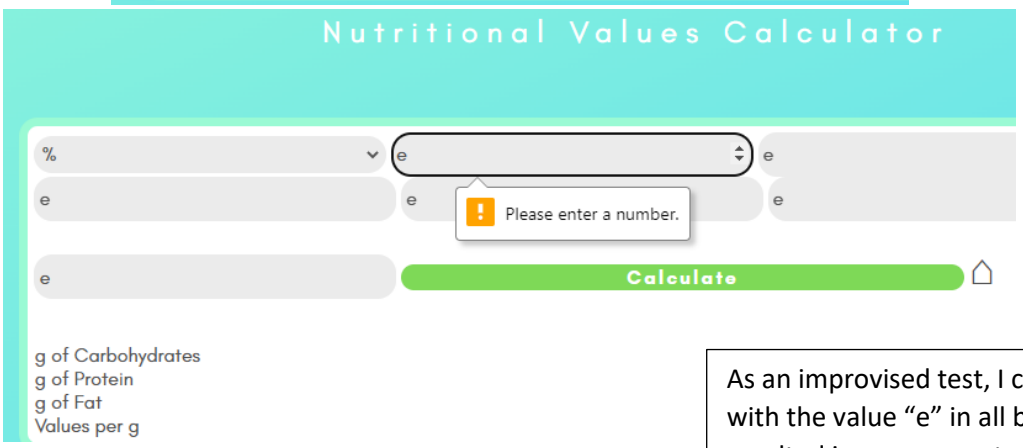
I selected a month and then clicked the small grey button which resulted in all the dates for the month being displayed.





After clicking the checkbox and the grey submit button, the page was refreshed. I then proceeded to selecting “July” again to find that a green icon appeared next to the date I selected. This showed that a meal plan was successfully generated for that date.

I attempted to type in "watermelon" into all boxes here to see how this feature would respond. This is a calculator that is expecting numerical values and so this also resulted in only "e" being displayed.



As an improvised test, I clicked calculate with the value "e" in all boxes. This resulted in an error automatically generated from the front end of the web app. Through HTML.

Evaluation

Post Development Testing & Success Criteria Review

I have cross-referenced evidence with the success criteria and have explained the extent to which I met the criterion. I have also justified why partial or unmet criteria was left and how it could be addressed in further development. Finally, I provided evidence of usability features and justified their success, partial success or failure as effective usability features.

The blue-coloured font is referring to tests that I planned in the design section, under "Further data" on page 43.

Criteria	Reference	Explanation
1. Create a database (Food)	Page 49 and page 152	**Partially met** – This is because my database model changed many times throughout the iterative development process. The two references show evidence of testing the database to see if it can correctly store values inserted into it. Point number 5 was not met because during the design stage I decided a relational database was a much better idea than each user having their own flat-file database. That would have taken up a lot of storage which is not efficient at all.
2. Create a database (Login details)		
3. Create a database (Exercise record)		
4. Create a database (Previously Eaten Foods)		
5. Create a database (Personal Food)		
6. Create a database (Custom Plan for		

each user with their preferences)		
7. Create a database (Facts from credible sources)		
8. Create a green-blue gradient as a theme for backgrounds on each page	Page 61	**Fully met** – I used pure CSS on the HTML body tag to create a linear gradient which can be applied to all the HTML files I was using for the different pages. The screenshots illustrate this.
9. The use of the font "Kumbh Sans" throughout each web page	Page 61	**Partially met** – I did use a custom font throughout the webpages however I decided to use Glacial Indifference as I thought it suited the theme of the pages better. This was not an issue as my stakeholders supported the decision.
10. The web-app must be responsive	Page 122	**Partially met** – If the window in which the webapp was to be minimized, the content did adjust to an extent. The main idea was to incorporate JavaScript so that the navigation bar could be dynamic and the entire webapp could adjust to various different screen sizes, however I did not have time to fully achieve this as I would have liked to.
11. A navigation bar presented horizontally across	Page 84	**Fully met** – I created a list which were hyperlinked to their corresponding webpages and styled the list horizontally to look like a navigation bar.
12. Have the following headings in the navigation bar: Home, Calendar, Progress, Food bank	Page 78 **Usability Feature**	**Partially met** – During the development stage, I decided to add "questionnaire" to the navigation bar. I thought it would be useful if the user could repeat the questionnaire multiple times, not just once when they initially create an account. This is why this specific point was partially met, I made a change to it.
13. On each page, display a fact box containing information randomly selected from the fact database	Page 118	**Fully met** – A red box to the left of each page contains a random fact retrieved from the fact table in the database. This was displayed on all the intended pages hence why it was fully met.
14. Ask the user to create an account	Page 61 **Usability Feature** Page 124 – Valid data Page 125 – Erroneous data	**Fully met** – the final page was created and I integrated all of the validation code I wrote into the python file that will render this page. The end result was a form that the user could fill out to then be directed to another page, this is why this feature was successful.

15. Verify if the username has already been taken 16. The user must create a secure password that consists of at least one capital letter and one number	Page 58	**Fully met** – The evidence shows that I carried out multiple validation tests to ensure the app could return the correct responses for each scenario.
17. Option to login as existing user 18. Verify the user's credentials	Page 63 **Usability Feature** Page 125 – Valid	**Fully met** – I carried out multiple tests to see if all the relevant validation checks could be completed successfully and they were. The user could enter details and if they were valid, they would be logged in and redirected to their home page, so this feature was a success.
19. Questionnaire for the user 20. What is your goal? 21. What is your current age, weight, height, body fat ratio, exercise level? 22. What is your desired weight? 23. What are your food preferences? 24. How many meals would you like to eat during the day?	Page 83 **Usability Feature** Page 125 – Invalid data Page 126 – Erroneous data	**Fully met** – Using invalid and valid testing allowed me to test for all the possible scenarios and each one resulted in the correct response being generated. The user could easily fill out the easy-to-read form and it would check if valid data has been entered. If it has, the user will be redirected to their home page and so this feature was a success.
25. Display the number of calories the user needs to eat	Page 86	**Fully met** – It adapted for each user as one user logged out and another logged in, it was also displayed at the top of the home page which was what I had planned initially.
26. Display the meal plan for the day	Page 92 Page 126 – Process	**Fully met** – The home page rendered the meal plans for the user as the data was fetched from the database.
27. Allow the user to generate new foods from the databases	Page 92 **Usability Feature**	**Fully met** – By the click of a button, a new list of meals was generated as a brand-new meal plan tailored to the user. The user could keep generating more by clicking the button. Hence why this feature was a success.
28. Allow the user to tick off each meal to mark as "eaten"	Page 97 **Usability Feature**	**Fully met** – By checking a box and then clicking a button, the user could mark foods as eaten and this could be done to multiple foods even after generating new plans – successful feature.
29. Eaten meals should update the calorie &	Page 97	**Fully met** – Upon marking foods as eaten, the calorie target was updated to show

nutritional target for the day	Page 126 – Valid data	how many calories were left that needed to be eaten. Furthermore, if the user un-marked a food, the calorie target would also adapt to this.
30. Eaten meals should be added to the database (previously eaten foods)	Page 97	**Partially met** – This is because I changed the database structure compared to what I had when creating the success criteria. Instead of using multiple flat file databases for each user, I used a large relational database that all users can access. This was not instructed by success criteria but it was a better, more efficient option.
31. Allow the user to add extra foods/meals	Page 99 **Usability Feature**	**Fully met** – The user could click the checkbox then click the button to add a new food to the existing plan – successful feature.
32. Allow the user to remove foods/meals	Page 97 **Usability Feature** Page 126 – Invalid data	**Fully met** - The user could click the checkbox then click the button to delete an exiting food from the existing plan – successful feature.
33. Update the calorie & nutritional target based on the user's custom meals	Page 97 **Usability Feature**	**Fully met**- Even when the user customized their meal plan by adding or removing foods, the calorie target would still adapt to their changes once they marked foods as eaten.
34. Display the nutritional values for each meal with the use of pie-charts	Page 118	**Fully met** – The nutritional values were displayed by using graphs part of the JavaScript library, Chart.js. The pie charts were colour coded as well, making it very easy to interpret for the user.
35. Show the user's current calorie target for the day	Page 86	**Fully met** – This was displayed on at the top of the page.
36. Display the calorie target using a progress bar	Page 88	**Fully met** – A grey bar that would update upon refreshing the page once the user marked foods as eaten. The bar could also display a percentage of completion along with a light blue bar representing this.
37. Show the user's water intake	Page 118 **Usability Feature**	**Fully met** – The user could click a button and a new glass icon would appear, indicating how many glasses of water the user drank.
38. Allow the user to input an exercise activity	Page 101 **Usability Feature** Page 127 – Valid data	**Fully met** – When inputs the calories they burned in the given text box and then click the plus button, the exercise activity will be displayed. The user can even remove an activity if they wanted to by clicking a checkbox and then clicking

	Page 127 – Invalid data	another button. This was a successful feature.
39. Provide a recommended exercise target	Page 99	**Fully met** – The user's activity level when they complete the questionnaire, is retrieved from the database and displayed on the home page.
40. Provide an option for the user to change their food preferences	Page 77 **Usability Feature**	**Partially met** – I found that it would be better to allow the user to re-do the questionnaire page in order to change their food preferences. This is because the user may want to change other details as well such as their goal. Instead of having separate features that allows the user to change their preferences and their goal, the user can just complete the questionnaire page again. This was not what I had planned but it proved to be successful and a better solution.
41. Provide an overview of the user's current goal, allowing them to change it if desired	Page 86 **Usability Feature**	**Partially met** – The user's current goal is displayed on the home page which is very easy to see as it is placed at the top. However, instead of creating a new feature that allows the user to change their goal, they can just complete the questionnaire page again. This was not part of my initial plan and so this specification point was partially met.
42. Allow the user to input their current weight 43. Add current weight inputs to the weight database	Page 104 **Usability Feature** Page 127 – Erroneous data	**Fully met** – The user could input their current weight at the top of the progress page. If data deemed unsuitable or invalid was entered, the weight would not be added to the database. If valid data was entered, the weight would be added to the database but also rendered on the line graph on the progress page. This feature was successful.
44. Produce a line graph from data points in the user's weight log	Page 120 Page 128 - Valid	**Fully met** – A line graph with dates on the x axis and a weight value interval on the y axis was displayed. When the user's mouse cursor hovers over the line graph, the individual data points will be highlighted.
45. Review progress from the line graph on the user's weight, if no progress is made, recommend a change	Page 120	**Not met** – Unfortunately I did not have time to work on this specification point. Although in further development, the app can check if the current date is equal to the last date on the expected progress graph. If the current date is greater than that date and the user's current weight is not equal to their desired weight, it clearly shows that the user did not reach

		their desired weight in the time projected by the app. A message to the user can then be sent of a recommendation of what to change. Thanks to the date time module in python, it is easy to check if a date is greater than another date so this would be an easy feature to add to my solution.
46. Produce a line graph showing projected progress	Page 120	**Fully met** - The two data points on this graph were; the most recent date of the user completing the questionnaire, with the user's weight at the time and the date calculated by the app which shows when the user is expected to reach their desired weight.
47. Produce a line graph from data points in the user's exercise record	Page 120	**Fully met** – The calories burned were placed on the y axis and the date corresponding to each exercise activity were placed on the x axis. The graph and its datapoints were rendered successfully.
48. Produce a pie chart that shows the user's diet makeup on the average nutritional value intake overtime 49. Display the average nutritional intake for each month/week	Page 118 **Usability Feature**	**Partially met** – A colour coded pie chart that displays the average nutritional intake for a given month selected by the user was rendered successfully. The user could choose what month and year to use for their diet makeup however they could not choose the week. Given that the main change/progress made in diet happens over a long period of time, I thought it would not be so useful for the user if there was an option to see their intake for individual weeks. Although implementing this feature would be simple as I had already created options for the user to select the month and year, selecting the week could be done in similar fashion with similar code.
50. Provide a BMI calculator that indicates ideal weight	Page 108 **Usability Feature** Page 128 – Valid Page 128 – Erroneous	**Partially met** – The user could enter their height and weight values in order to calculate their BMI. The result was successfully displayed on the progress page and stored in the database however I did not have time to display a message to the user on whether their BMI value is healthy or not – I assumed they had this knowledge already. In further development, this can be implemented with a simple if statement, if the BMI value is less than or greater than certain

		values, display a message to the user that they are not at a healthy weight.
51. Display the user's estimated time to reach their desired weight	Page 108	**Fully met** – At the bottom of the progress page, the estimated time was displayed using weeks as its units.
52. Provide a calendar	Page 110 Page 129 – Valid data	**Fully met** – The user could select a month and a year to choose from and then a list of all the dates corresponding to the user's inputs would be displayed in small individual boxes.
53. Allow the user to create meal plans for future dates	Page 111 **Usability Feature** Page 129 – Valid data	**Fully met** – By clicking a checkbox and then a submit button, a meal plan corresponding to the date selected would be generated. Upon refreshing the page, a green icon would be shown next to the date that was selected for generating a meal plan.
54. Allow the user to search foods from the main food database	Page 112 **Usability Feature**	**Fully met** – The user could type in the input box for a specific food they are looking for and then confirm this search by clicking the button with the magnifying glass icon. If a food exists in the database that matches the user's search, it would be displayed in the form of a small box.
55. Filter the database with the user's preferences		**Not met** – Unfortunately, I could not finish this in the given time frame however in further development of the solution, I could use a similar technique to generating meal plans tailored to the user. By creating a list of all the user's disliked food preferences, I could use this directly in an SQL statement to return a filtered set of results.
56. Allow the user to "favourite" foods from the main food database		**Not met** – This was not met because this specification point became redundant as a result of changing my database structure. I moved away from the idea of each user having their own flat file database.
57. Allow the user to view their own database		
58. Provide an input for the user to enter foods to their personal food database	Page 114 **Usability Feature**	**Partially met** – The user would be directed to a different page where they could input food to the main foodbank database. Since no user had their own personal database, I modified this feature so that the data will be added to the foodbank instead.
59. Display a window asking the user to	Page 114 **Usability Feature**	**Partially met** – On the food input page, the user could enter all nutritional values

enter all the required nutritional values for their custom food	Page 115 – Valid data	and the app would validate each input to ensure they are valid. On the other hand, I could not keep this feature on the foodbank page and have a pop-up window that allows the user to write the inputs on there. This required a large amount of JavaScript which I could not implement due to the time restriction.
60. Produce a calculator that can work out nutritional values per 100g	Page 116 **Usability Feature** Page 130 – Erroneous data	**Fully met** – I was able to create an icon that directs the user to a calculator page. Here, the user can choose from two unit options and then complete the form which will calculate values based on the user's input. A slight modification made here was that the app would calculate values for any given amount asked by the user, not just per 100g. This gave more options for the user and proved to be a good change.

Maintenance Issues and Limitations of The Solution

Issue or Limitation	Improvements/Resolution
Archiving the data of users – If there is an increasingly large number of users, there may not be enough space to store all their data on the current web server that is being run on my laptop. Furthermore, each user will have a large amount of data to store as the solution holds data from the past months to allow the user to review their progression.	Using an actual server that is solely dedicated to storing data for this solution would tackle this particular maintenance issue as servers have a very large amount of storage. However, they are expensive and could require specialist workers to maintain such servers.
Phone-screen friendly – Although the webapp was responsive, mobile apps tend to use a slightly different interface. For example, the horizontal navigation bar, would not fit on a phone screen and refreshing the page for every time an input is taken, is not suitable for a phone.	This could be dealt with by creating an app for android or IOS – the two main operating systems of phones. Creating an app based on the features of the webapp will ensure phone users have the solution best suited to their situation. Although, this creating an app is very expensive. Particularly, maintaining the app and developing new features to satisfy users.
Security – The webapp has a login system but it does have flaws. If the user was to forget their password, they have no option of resetting it. Furthermore, no form of encryption is used when the user creates a password.	Flask has a built-in function called password hash which applies a hashing algorithm to a given string. This will add some level of security when the user creates a password. When resetting passwords, an email is sent to the user – this is the case with most websites. Applying this would be difficult because I would have to work out how I could program a bot to send an email to a certain user and then have the user's input added to the database.
Validation upon adding food to database – At the moment, the web app does not check if a	This should be a fairly simple issue to resolve because I have already proved that the web

new food being added by the user already exists in the food database or not. This could lead to duplicated data in the backend database which takes up storage.	app can successfully check if a username already exists in the database. The same method can be used to check if a food name exists in the database by running a query based on the food name that the user plans to add to the database. However, for this to be effective, the solution should also be able to take into account of the number of spaces the user enters between words. This is to ensure that if the user tries to add for example; "Orange" to the food database but it already exists, the user might try adding " Orange ". With the additional spaces, the web app might not treat this as "Orange". This could result in duplicated data with slightly different names being stored in the database.
Searching the food database manually – When the user searches the food bank using the search bar, they have to type the exact name of the food for it to be displayed. They cannot type something similar and still get a return of similar results. For example; when searching for granola, if the user types in "gran" and clicks search, nothing will show up.	An SQL command could be used to resolve this issue; LIKE. This could see food names that are similar to what the user inputted, be displayed even if the user has not entered the exact name of a food.
Static method of logging water intake – The water intake feature is very limited as there are no units to indicate how much water is classed as a glass. Furthermore, whenever the user visits a new page after logging water intake, the water icons clear and so when the user goes back to the home page, they have to input the entire water log again.	A method to work around this would be to create another table in the backend database that is responsible for storing a water icon, the user's id and the date. The water icon used in my web app were part of the Unicode character set. Hence why the exact same icon could be stored in the database as MySQL could recognise water icons as strings.
Static progress bar – When the user marks a meal as "eaten", they have to refresh the page in order to see what their remaining calories is on the calorie target. This is because I used CSS to style the progress bar.	With the use of JavaScript, we could see a similar dynamic to how the pie charts were rendered on the page. This could allow the user to mark foods as eaten and the progress bar would instantly adjust without the need of refreshing the page.
Erroneous data not being detected – When the user enters data that is unexpected, the web app does not display it at all. This was an observation I made after my post development testing after numerous times where I tried using erroneous data to test multiple features.	During the iterative development, I wrote HTML input tags to be a certain type of input. This is a standard HTML feature, that allows certain input tags to accept passwords, numbers only or even strings only. Instead of relying on HTML solely, JavaScript can be used as well to validate the user inputs further. With the use of JavaScript, special characters such as "£$%^&*" can be shown in the HTML input area and the web app could then generate a customized error that could say "Invalid input, special characters".

Generating meals from the calendar – The web app does not prevent the user from generating meal plans on days that have already passed. The user could generate plans for days that will never come and so the meal plan takes up unnecessary space in the database.	The python code for the calendar web page can compare the present day's date with the date that the user chose to generate a meal plan. If the date chosen is less than today's date, then an error can be generated to tell the user that they have selected a date that has already passed. This can be achieved with the use of the python time module, furthermore, dates can be filtered within SQL commands as this is what I used to design the calendar feature.