FITNESS APP DEVELOPMENT

FitFuel

by Silmoon Hossain- NewVIc

AQA A-level Computer Science Non-Exam Assessment (2024)

A-Level Computer Science NEA Project
Title: Fitness app
Student Name:Silmoon Hossain
Student ID: HOS21004547
Centre Number: 13227

Section 1: Analysis

1.1 Project Background

The fitness world is changing fast – and a big part of that is how technology helps us manage our health. Fitness apps are everywhere, and for good reason. They track our workouts, suggest what to eat, and keep us on the right path to reach our fitness goals.

But here's the thing: it's easy to feel a bit lost in the crowd of fitness apps. Lots of them feel impersonal– like a workout plan designed for a robot, not a real person. Or they're a pain to use, making it harder to actually form those healthy habits.

My research shows that people want a fitness app that feels more like a coach than a spreadsheet. Something that learns about you, gets what motivates you, and builds a plan that actually works with your life. And they want it to cover more than just workouts – they need help with food, with finding their workout groove, and staying balanced in a hectic world.

This project is about building the kind of fitness app I'd want to use myself. It'll be personalised, easy, and it'll go beyond the basics because fitness isn't just a workout, it's a whole lifestyle change. I'm betting there are a lot of people out there who want this too!

1.2 Current System

I therefore thoroughly examined MyFitnessPal, Fitbit, and AlphaProgression—three of the most widely used fitness applications available. I was interested in finding out what they excel at and, more significantly, where they left customers wanting more.

Let's break it down:

- MyFitnessPal: Great for tracking what you eat. Huge food database, folks love the barcode scanner, but... you're on your own with personalised meal plans or any workout advice. It's like having a food journal, not a coach.

- Fitbit: This one shines when it's hooked up to their watches. Tracks all your steps, sleep, the works. It edges out MyFitnessPal with some more tailored fitness plans. Still, those plans feel pretty generic, not made-for-me. Plus, users aren't always happy with the hardware syncing up right.

- AlphaProgression: Now, this one is cool if your goal is building muscle. Workouts designed for you based on what you want and what equipment you have? That's different! Plus, they nail the workout instructions with videos and everything. But the nutrition features are not there.

So, where's the opportunity here? Here's what I see:

It seems like everyone wants something more. More personalised plans – workouts and meals made for the real you. They also want better support – not just a list of exercises, but someone (or something) helping them to track their progress and their nutrition all in one place.

That's where my app comes in. Imagine AlphaProgression's personalization but for fitness at every level, plus that missing nutrition piece everyone wants. Throw in some AI magic to improve the feedback and that's the kind of user experience that blows the current competition away.

1.3 Research and Personas

Overview:

I conducted an online survey with an average response rate of 620 people per question, advertised in a fairly followed fitness account of an acquaintance on Instagram Stories.

I conducted direct observations in a Local Gym: I noted how gym-goers interacted with their fitness apps during workouts. Observed common frustrations, such as switching between multiple apps for different needs.

Gathered informal feedback from gym-goers on their ideal features in a fitness app. Demographics: Diverse range of respondents, primarily aged 16-40.

1.3.1 Online Survey Result:

{Full Online Survey can be found in Appendix A}

Important takeaways:

- Use of Fitness App: A significant 27% of participants use a fitness app for both nutrition and workout tracking. This indicates a considerable interest in comprehensive fitness solutions.
- Primary Goal: The most common primary goal for using a fitness app is weight loss (29%) and weight loss while building muscle at the same time

(14%). This shows that there's a significant need for all-inclusive fitness solutions.

- Important Features: The vast majority of participants (56%) favour personalised exercise plans. This highlights the need for fitness apps to be more personalised.
- Biggest Challenges: The most prominent challenge faced by users (48%) is the lack of personalization in current fitness apps. This demonstrate even more how customised fitness solutions are required.
- Preferred App Offering: Only 12% of participants are interested in apps offering solely workout plans, suggesting that users are looking for more holistic fitness solutions.
- Exercise Content Preference: A preference for interactive content with a coach (32%) indicates a desire for more engaging and interactive exercise guidance.
- Method for Fitness Advice: The majority (35%) prefer receiving fitness advice through video tutorials, highlighting the importance of visual learning in fitness education.
- Fitness Expertise Level: 32% of the participants consider themselves beginners, which suggests that the app should cater to users who are just starting their fitness journey, with simplified and easy-to-follow guidance.

1.3.2 User Personas Based on Research:

{The full interview can be found in Appendix A}

Abdullah, 35-year old male: Busy professional with family responsibilities. Seeks time-efficient workouts. Past injuries make him cautious about exercise form.
Goals: Improve fitness, lose weight, prevent injury.
Frustrations: Time constraints, lack of personalised guidance, risk of injury.

Abby, 20-year old college student: Aspiring to a healthier lifestyle but lacks routine. Overwhelmed by conflicting online advice.
Goals: Build muscle, increase energy, gain confidence.
Frustrations: Unsure where to begin, difficulty in staying motivated.

Zak, 17 year old college student: busy with college studies, wants to build strength and muscle and have fun at the gym, struggles to follow a set workout schedule and needs something more personalised based on his specific needs.
Goals: Build muscle and strength, track progress.
Frustrations: Struggle to follow a set plan, struggles to track exercises.

1.3.3 Linking Findings to User Needs:

- Customizable Workout Plans:
    - User Need: A strong desire for personalised fitness plans was evident among users of all age groups.
    - App Feature: The app incorporates an algorithm to create tailored workout plans, catering to individual user inputs like age, weight, and goals, with variations for different experience levels.
- Comprehensive Food Tracking Integration:
    - User Need: Users showed significant interest in tracking their nutritional intake easily and accurately.
    - App Feature: A food database is integrated into the app, allowing users to log foods and view graphical representations of their calorie and macronutrient intake.
- Access to an Extensive Exercise Library:
    - User Need: Participants wanted accessible, detailed guidance on exercise techniques.
    - App Feature: An extensive library of exercises, with detailed technique guides, addresses this need, ensuring safe and effective workouts.
- Interactive and User-Friendly Interface:
    - User Need: A strong preference for an intuitive and engaging app interface was frequently expressed.
    - App Feature: Using modern UI design principles build a user friendly and good-looking interface.
- Comprehensive Tracking Features:
    - User Need: Users expressed the desire to track various aspects of their fitness journey in an integrated manner.
    - App Feature: Users may log their body measurements, exercise progress, and food intake with this app, which gives them a comprehensive picture of their fitness journey.
- AI-powered Conversational Coach:
    - User Need: There was a noticeable interest in receiving personalised coaching and motivation without having to pay for an actual coach.
    - App Feature: An AI-powered chatbot within the app offers workout tips, nutrition advice, and motivational support, catering to this need for interactive guidance.

1.4 Objectives

1.4.1 Customizable Workout Plans:

- Develop an algorithm to create workout plans based on user input (age, weight, goals, etc.).
- Ensure plans vary exercises to target all major muscle groups.
- Include options for different experience levels: beginner, intermediate, advanced.
- Test Objective: Validate with users of varied profiles to ensure plans are appropriately tailored.

1.4.2 Comprehensive food tracking integration:

- Integrate a comprehensive food database that includes a wide range of foods, including various cuisines, brands, and whole foods.
- Allow users to log foods easily into a database and have graphs displayed for calories and macronutrients consumed.
- Test Objective: Verify the database covers a diverse range of foods by checking for all types of food. Check database interactions and user feedback.

1.4.3 Nutritional Breakdown of Meals:

- Provide a detailed nutritional breakdown for each logged meal, including calories and macronutrients.
- Allow the user to change portion size and unit of measurement.
- Test Objective: Log various meals and verify the accuracy and detail of the nutritional information provided. Log portion size and different units to check if the app calculates and displays all the amounts correctly.

1.4.4 User-Created Workout Plan Design:

- Provide an interface for users to design their own workout plans from the exercise library.
- Include elements such as selecting days of the week, specific exercises, sets, reps, and rest periods.
- Test Objective: Have users create workout plans of varying complexity. Check if they are: saved correctly in the database and retrievable for later use.

1.4.5 Extensive Exercise Library and Technique Guidance:

- Provide an extensive library of exercises with an image describing the exercise.
- Implement a search feature to enable the exercise library.

- Add a detailed technique guide on how to perform each exercise safely, with the correct tempo and common mistakes.
- Test Objective: Ask test users on the quality of the exercise techniques, ease of manoeuvrability though the list of exercises and test the search function.

1.4.6 Workout Logging and History:

- Allow users to select a type of workout (generated plan, custom plan, single workout).
- Enable logging of sets, reps, and weight lifted for strength-based workouts.
- Test Objective:Perform multiple workout logging sessions using different workout types.Ensure all data is stored correctly in the database.

1.4.7 Secure User Authentication and Data Handling:

- Implement a secure login system with hashed password storage.
- Ensure all personal user data is encrypted and stored securely.
- Test Objective: Conduct security testing to check for vulnerabilities in authentication and data storage.

1.4.8 Interactive and User-Friendly Interface:

- Utilise the Kivy framework for developing an engaging user interface.
- Design responsive elements and clear navigation pathways.
- Test Objective: Perform usability testing with a group of 5-10 test users to identify any navigation issues or design flaws.

1.4.9 Comprehensive Tracking Features:

- Enable tracking of body measurements, workout progress, and dietary intake.
- Implement visual progress graphs and statistical overviews.
- Test Objective: Enter sample data and assess the accuracy and clarity of graphs from test users.

1.4.10 AI-powered Conversational Coach:

- Develop a chatbot that can provide workout tips, nutrition advice, and general motivation.
- Ensure the AI can handle a variety of user queries accurately.
- Test Objective: Conduct a series of predefined queries to assess the AI chatbot's response accuracy and helpfulness.

1.5 Data Requirements

The key data the app will need to collect from users includes:

- Personal stats: weight, height, body fat percentage, gender, age
- Goals: weight loss/gain targets, muscle gain targets, target body fat percentage
- Experience level: workout frequency, lifting experience, cardio endurance
- Equipment access: gym membership status, types of home equipment owned
- Schedule/time availability: ideal workout days/times
- Workout tracking: exercises done, sets, reps, weight lifted, duration
- Nutrition tracking: meals, macros, calories, ingredients, portion sizes
- Body measurements: weight, body fat percentage

This data will allow the app to generate fully personalised fitness and nutrition programming tailored to each user while providing them with quantified progress tracking.

1.6 Acceptable Limitations

While this app offers exciting potential for users, it's important to understand the constraints of its current development phase:

- Development Stage: As a solo project with limited resources, the app may have some incomplete features or areas in need of refinement.
- Injury Considerations: The generated plans assume a generally healthy user. Those with existing injuries may need to consult with a healthcare professional to ensure the app's plans are safe and suitable.
- Focus on Guidance, Not Live Coaching: While the app provides instructional content and AI-powered insights, it does not replace the personalised feedback of a live fitness coach.
- Nutrition Tracking: Accurate nutrition tracking relies on manual food entry by the user. Future versions may have integrations for more efficient tracking.
- Workout Style: The app excels in providing balanced fitness programs but may not offer highly specialised plans for niche disciplines like powerlifting or advanced calisthenics.

These limitations reflect the app's ongoing development. I remain committed to expanding its capabilities and addressing these areas in future updates.

1.7 The Potential Solution

My research revealed many distinct gaps in the fitness app market, including more personalised training and nutrition plans, improved user interface design, and thorough health tracking. The proposed solution is a fitness app designed to address these issues effectively:

Tailored Fitness Programs:

- Solution: The app uses sophisticated algorithms to generate individualised workout plans based on user data like age, weight, goals, and fitness level.
- Impact: This customization ensures that users receive relevant and effective workout recommendations, moving away from the generic approach prevalent in current apps.

Interactive Fitness Coaching:

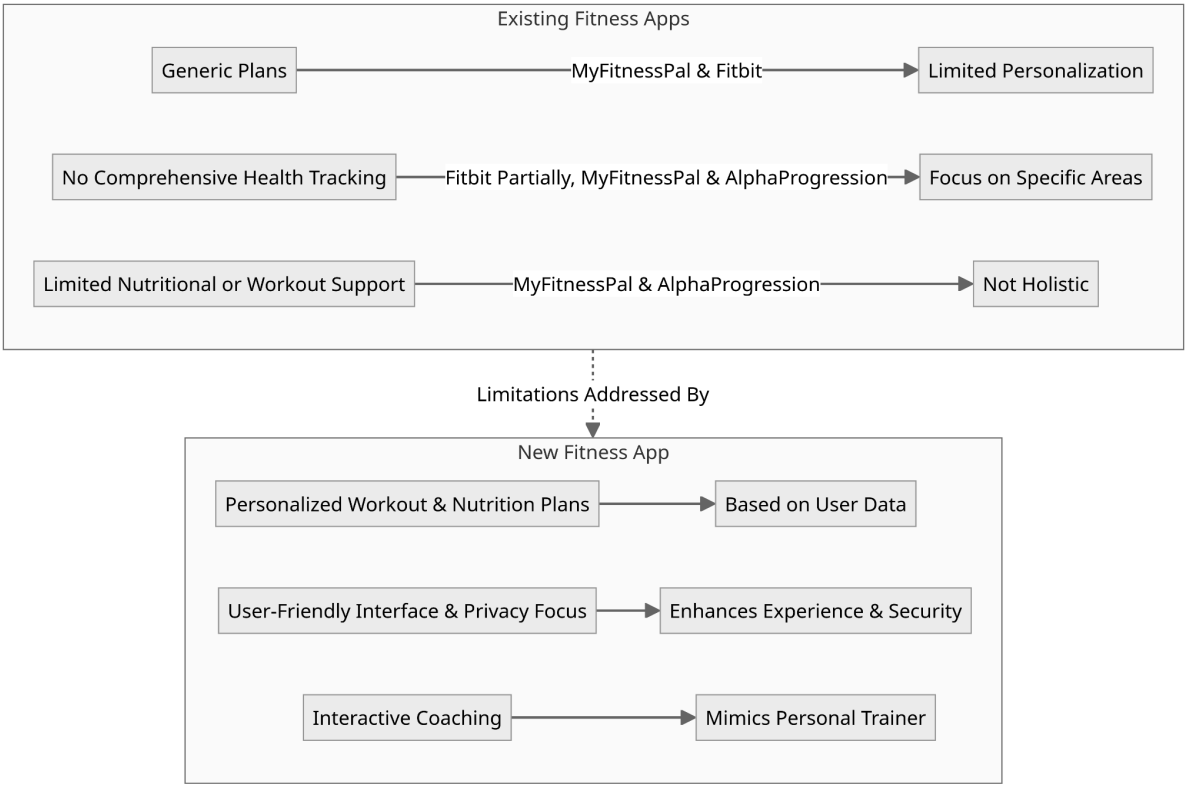
- Solution: The app includes a feature to provide personalised fitness advice and motivational support, based on user interactions and preferences.
- Impact: This feature replicates the guidance of a personal trainer, improving user engagement and offering more tailored fitness advice.

User-Friendly Interface:

- Solution: Developed with a focus on simplicity and ease of use, the app features an intuitive interface with clear navigation.
- Impact: This design addresses frequent user concerns related to complex app interfaces, with the goal of improving overall user experience and accessibility.

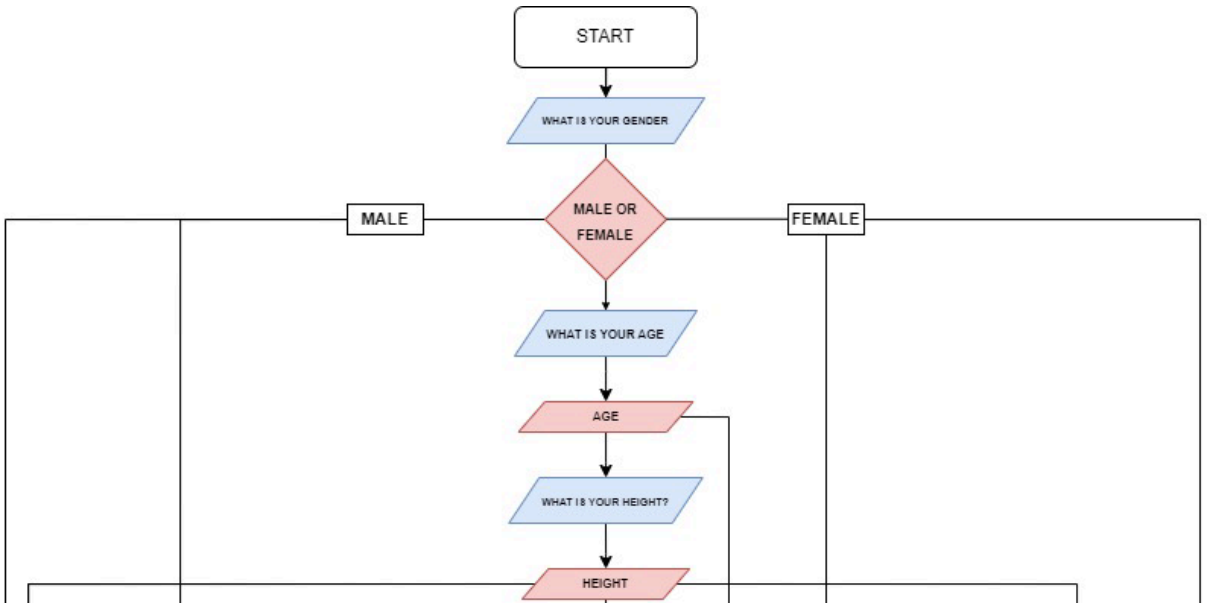
Overall, this app is aimed to be a user-friendly solution that addresses the weaknesses found in current fitness apps, with a focus on customisation, simplicity of use and good tracking features.

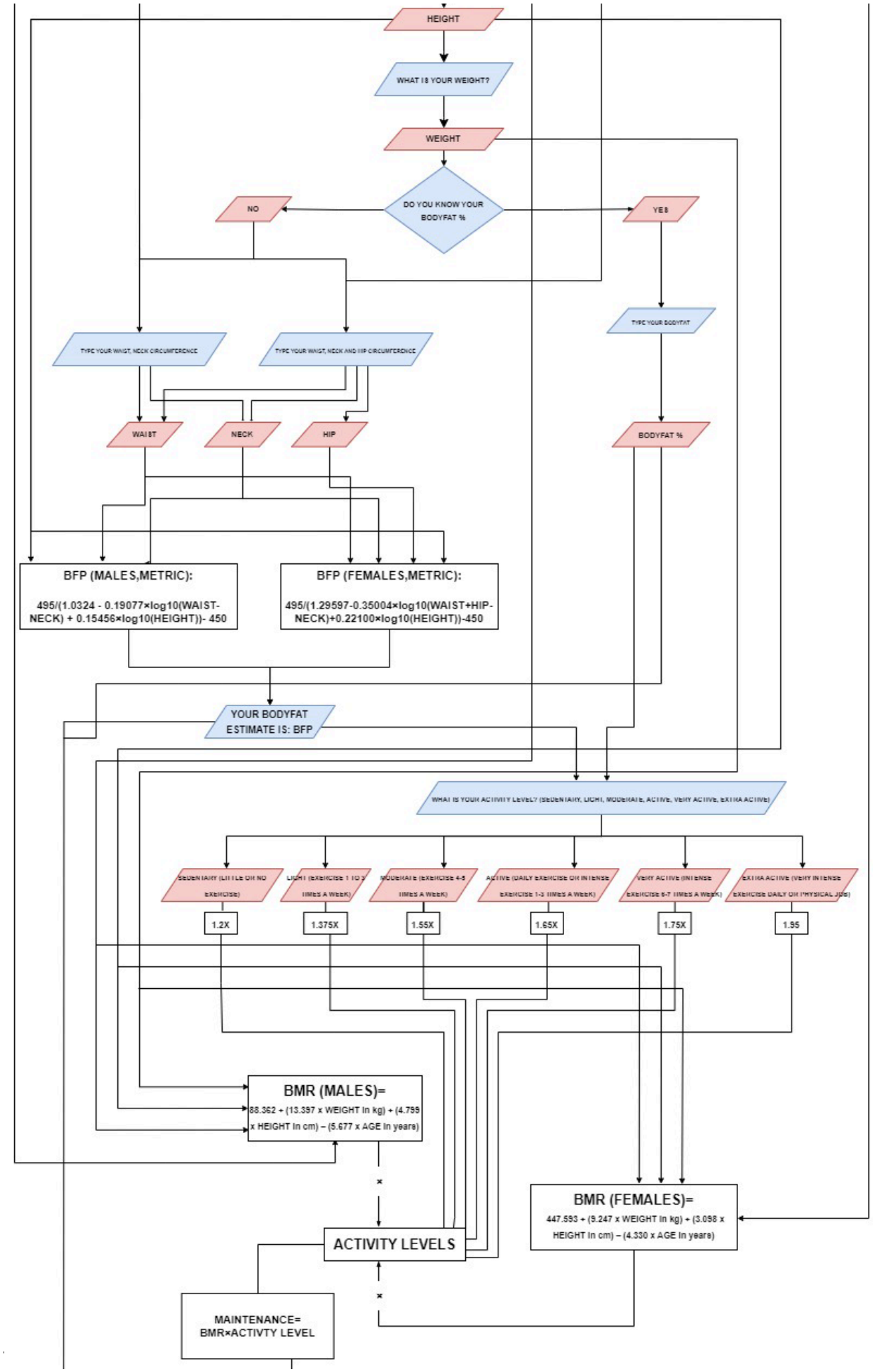
1.8 Model of Existing system

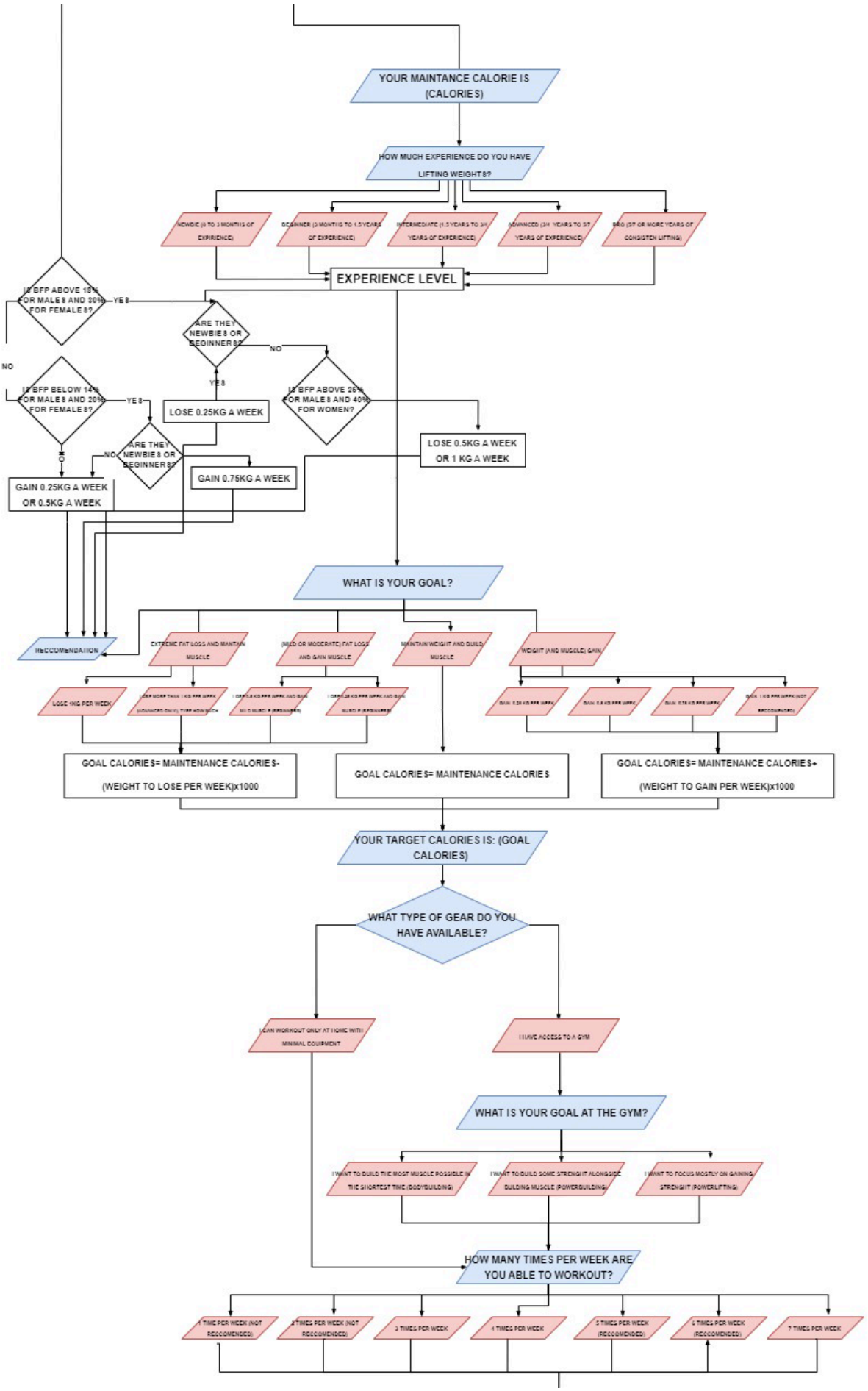


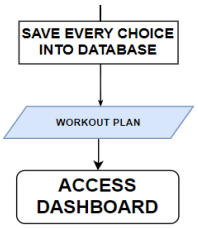
Section 2: Design

2.1 System Flowchart (User onboarding)

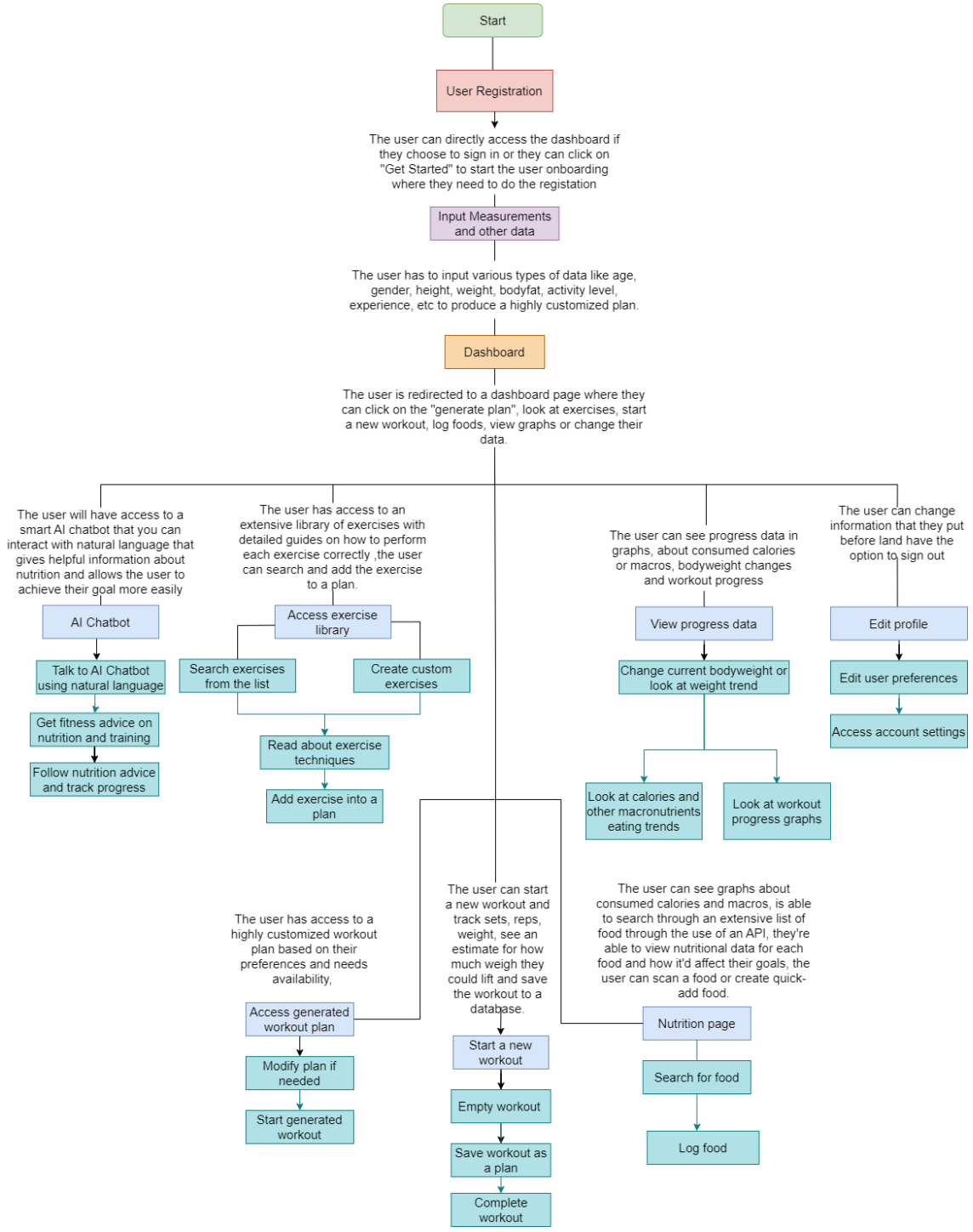



HEIGHT
WHAT IS YOUR WEIGHT?
WEIGHT
DO YOU KNOW YOUR BODYFAT %
NO
YES
TYPE YOUR BODYFAT
TYPE YOUR WAIST, NECK CIRCUMFERENCE
TYPE YOUR WAIST, NECK AND HIP CIRCUMFERENCE
WAIST
NECK
HIP
BODYFAT %
BFP (MALES,METRIC):
495/(1.0324 - 0.19077×log10(WAIST-NECK) + 0.15456×log10(HEIGHT))- 450
BFP (FEMALES,METRIC):
495/(1.29597-0.35004×log10(WAIST+HIP-NECK)+0.22100×log10(HEIGHT))-450
YOUR BODYFAT ESTIMATE IS: BFP
WHAT IS YOUR ACTIVITY LEVEL? (SEDENTARY, LIGHT, MODERATE, ACTIVE, VERY ACTIVE, EXTRA ACTIVE)
SEDENTARY (LITTLE OR NO EXERCISE)
LIGHT (EXERCISE 1 TO 3 TIMES A WEEK)
MODERATE (EXERCISE 4-5 TIMES A WEEK)
ACTIVE (DAILY EXERCISE OR INTENSE EXERCISE 1-3 TIMES A WEEK)
VERY ACTIVE (INTENSE EXERCISE 6-7 TIMES A WEEK)
EXTRA ACTIVE (VERY INTENSE EXERCISE DAILY OR PHYSICAL JOB)
1.2X
1.375X
1.55X
1.65X
1.75X
1.95
BMR (MALES)=
88.362 + (13.397 x WEIGHT in kg) + (4.799 x HEIGHT in cm) – (5.677 x AGE in years)
BMR (FEMALES)=
447.593 + (5.247 x WEIGHT in kg) + (3.098 x HEIGHT in cm) – (4.330 x AGE in years)
ACTIVITY LEVELS
MAINTENANCE= BMR×ACTIVTY LEVEL


YOUR MAINTANCE CALORIE IS (CALORIES)
HOW MUCH EXPERIENCE DO YOU HAVE LIFTING WEIGHT 8?
NEWBIE (0 TO 3 MONTHS OF EXPIRIENCE)
BEGINNER (3 MONTHS TO 1.5 YEARS OF EXPERIENCE)
INTERMEDIATE (1.5 YEARS TO 3/4 YEARS OF EXPERIENCE)
ADVANCED (3/4 YEARS TO 5? YEARS OF EXPERIENCE)
PRO (5? OR MORE YEARS OF CONSISTEN LIFTING)
EXPERIENCE LEVEL
IS BFP ABOVE 18% FOR MALE 8 AND 30% FOR FEMALE 8?
YES
ARE THEY NEWBIE 8 OR BEGINNER 8?
NO
YES
IS BFP BELOW 14% FOR MALE 8 AND 20% FOR FEMALE 8?
YES
LOSE 0.25KG A WEEK
IS BFP ABOVE 26% FOR MALE 8 AND 40% FOR WOMEN?
NO
ARE THEY NEWBIE 8 OR BEGINNER 8?
NO
GAIN 0.75KG A WEEK
LOSE 0.5KG A WEEK OR 1 KG A WEEK
GAIN 0.25KG A WEEK OR 0.5KG A WEEK
WHAT IS YOUR GOAL?
RECCOMENDATION
EXTREME FAT LOSS AND MAINTAIN MUSCLE
(MILD OR MODERATE) FAT LOSS AND GAIN MUSCLE
MAINTAIN WEIGHT AND BUILD MUSCLE
WEIGHT (AND MUSCLE) GAIN
LOSE 1KG PER WEEK
GAIN 0.25 KG PER WEEK
GAIN 0.5 KG PER WEEK
GAIN 0.75 KG PER WEEK
GAIN 1 KG PER WEEK (NOT RECCOMENDED)
GOAL CALORIES= MAINTENANCE CALORIES- (WEIGHT TO LOSE PER WEEK)x1000
GOAL CALORIES= MAINTENANCE CALORIES
GOAL CALORIES= MAINTENANCE CALORIES+ (WEIGHT TO GAIN PER WEEK)x1000
YOUR TARGET CALORIES IS: (GOAL CALORIES)
WHAT TYPE OF GEAR DO YOU HAVE AVAILABLE?
I CAN WORKOUT ONLY AT HOME WITH MINIMAL EQUIPMENT
I HAVE ACCESS TO A GYM
WHAT IS YOUR GOAL AT THE GYM?
I WANT TO BUILD THE MOST MUSCLE POSSIBLE IN THE SHORTEST TIME (BODYBUILDING)
I WANT TO BUILD SOME STRENGHT ALONGSIDE BULDING MUSCLE (POWERBUILDING)
I WANT TO FOCUS MOSTLY ON GAINING STRENGHT (POWERLIFTING)
HOW MANY TIMES PER WEEK ARE YOU ABLE TO WORKOUT?
1 TIME PER WEEK (NOT RECCOMENDED)
2 TIMES PER WEEK (NOT RECCOMENDED)
3 TIMES PER WEEK
4 TIMES PER WEEK
5 TIMES PER WEEK (RECCOMENDED)
6 TIMES PER WEEK (RECCOMENDED)
7 TIMES PER WEEK



2.2 Hierarchy Chart



2.3 Normalisation

Example of an unnormalised table for storing user workout plans

PlanID	UserID	PlanName	DayNumber	ExerciseName	Sets	Reps
1	101	New Plan	1	Hanging Leg Raise	2	6
1	101	New Plan	1	Bar Dip	3	10
1	101	New Plan	1	Back Extension	4	10
1	101	New Plan	2	Decline Push-up	1	15
1	101	New Plan	2	Assisted One Leg Squat	2	10
...	...	...	...	...	...	...
1	101	New Plan	7	Machine Shoulder Press	4	8

In this table:
PlanID is an identifier for the workout plan.
UserID references the user the plan belongs to (linked to my userdata table's id field, 101 is set as an example).
PlanName is the name of the workout plan that the user can set.
DayNumber indicates which day of the plan the row refers to.
ExerciseName is the name of the exercise.
Sets and Reps indicate how many sets and repetitions of the exercise should be done.

Issues with the unnormalised table:
Redundancy: The PlanName and UserID are repeated for every exercise on every day, leading to unnecessary duplication of data.
Update Anomalies: If I need to change the PlanName, it has to be updated in multiple rows, increasing the risk of inconsistent data if not all rows are updated correctly.
Insertion Anomalies: To add a new exercise, I need to insert a new row with repeated plan and day information, which is inefficient and can lead to inconsistencies.
Deletion Anomalies: Removing all exercises for a day (or the last exercise of a plan) would remove the record for that day (or the plan itself), potentially losing important information.

Normalised tables:

Workout Plans Table: I created a 'WorkoutPlans' table to store unique details about each plan, including a primary key (PlanID), the user ID (linking to the 'userdata' table), and the plan name. This eliminated the need to repeat the plan name and user ID for every exercise entry.

PlanID	UserID	PlanName
1	123	Push-Pull-Legs
2	456	Upper-Lower
3	123	Bro Split

Days Table: To maintain the structure of the workout plan across multiple days without repeating plan details, I established a 'Days' table. This table included a DayID as the primary key, a reference to the PlanID, and a DayNumber to indicate the sequence of days within a plan.

DayID	PlanID	DayNumber
1	1	1
2	1	2
3	1	3
4	2	1
5	2	2

Exercises Table: Recognizing that the same exercise could appear in multiple plans, I created an 'Exercises' table to store unique exercises. This table had an ExerciseID as the primary key and an ExerciseName column. By doing this, I could reference exercises in workout plans without duplicating exercise details.

ExerciseID	ExerciseName
1	Hanging Leg Raise
2	Bar Dip

3	Back Extension
4	Decline Push-up
5	Assisted One Leg Squat

DayExercises Link Table: Finally, to link exercises to specific days in a plan, I introduced a 'DayExercises' link table. This table included a DayExerciseID as the primary key, with foreign keys linking to the DayID and ExerciseID. It also included columns for sets and reps specific to each exercise occurrence.

DayExerciseID	DayID	ExerciseID	Sets	Reps
1	1	1	2	6
2	1	2	3	10
3	2	3	4	10
4	2	4	1	15

Example of unnormalized table to store logged food and consumed calories/macros in the database for each day:

user_id	label	kcal	protein	carbs	fats	fiber	portion_size	weight	unit	timestamp	total_calories	total_protein	total_fats	total_carbs	date	daily_target
123e4567-e89b-12d3-a456-556642340000	Grilled Chicken Breast	165	31.00	0.00	3.57	0.00	1.00	172.00	g	2023-03-09 12:30:00	1850	160	65	210	2023-03-09	2000 kcal
123e4567-e89b-12d3-a456-556642340000	Brown Rice	216	4.52	45.80	1.42	1.80	1.00	185.00	g	2023-03-09 13:15:00	1850	160	65	210	2023-03-09	2000 kcal

In this table:

We have data for a user (123e4567-e89b-12d3-a456-556642340000). Each row represents a food item entry, including its nutritional values and portion size, as well as the daily totals and target for the corresponding date.

Note how the total_calories, total_protein, total_fats, total_carbs, date, and daily_target columns are repeated for each food item entry, leading to data redundancy and potential issues with updates and deletions.

Normalised tables:

Food_items Table: To store unique details about each food item, including the user_id (linking to the 'userdata' table), label, kcal, protein, carbs, fats, fibre, portion_size, weight, unit, and timestamp. This eliminated the need to repeat these details for every daily total entry.

user_id	label	kcal	protein	carbs	fats	fiber	portion_size	weight	unit	timestamp
123e4567-e89b-12d3-a456-556642340000	Grilled Chicken Breast	165	31.00	0.00	3.57	0.00	1.00	172.00	g	2023-03-09 12:30:00
123e4567-e89b-12d3-a456-556642340000	Brown Rice	216	4.52	45.80	1.42	1.80	1.00	185.00	g	2023-03-09 13:15:00
123e4567-e89b-12d3-a456-556642340000	Mixed Vegetables	94	3.28	11.20	0.44	4.80	1.00	180.00	g	2023-03-09 19:45:00

Daily Totals Table: To maintain the daily nutritional totals and targets without repeating food item details, I established a 'daily_totals' table. This table included the user_id (referencing the 'userdata' table), total_calories, total_protein, total_fats, total_carbs, date, and daily_target.

user_id	total_calories	total_protein	total_fats	total_carbs	date	daily_target
123e4567-e89b-12d3-a456-556642340000	1850	160	65	210	2023-03-09	2000 kcal
4e8a2b1c-f5d3-4b67-9d8c-7e3f6b521234	2200	120	80	300	2023-03-09	2200 kcal

2.4 SQL Tables design

Userdata table:

Column Name	Data Type	Description
id	CHAR(36)	Primary key, unique identifier for each user
username	VARCHAR(255)	User's chosen username
email	VARCHAR(255)	User's email address
password	VARCHAR(255)	User's password
gender	VARCHAR(6)	User's gender
age	INT	User's age
height	DECIMAL(5,2)	User's height
weight	DECIMAL(5,2)	User's weight
experience_level	VARCHAR(50)	User's experience level in fitness/training
bodyfat	DECIMAL(4,2)	User's body fat percentage
activity_level	VARCHAR(255)	User's activity level
goal	VARCHAR(100)	User's fitness goal (e.g., weight loss, muscle gain)
calories	INT	User's calorie target
equipment	VARCHAR(255)	User's available exercise equipment
training_style	VARCHAR(100)	User's preferred training style
training_frequency	VARCHAR(100)	User's preferred training frequency
timestamp	DATETIME	Timestamp of user account creation
prioritized_muscle_groups	VARCHAR(255)	User's prioritised muscle groups for training

- Purpose: Stores user profiles, including personal statistics and preferences, which are essential for generating personalised workout plans. It captures user onboarding data and supports account management.
- Security: Passwords are stored as hashes using a strong hashing algorithm (bcrypt) along with a unique salt for each user to ensure security.

- Normalisation: The table is designed to avoid redundancy, following normalisation rules up to the third normal form (3NF). Each field contains only atomic data, ensuring that there are no repeating groups or composite fields.
- Data Integrity: Enforces data integrity through constraints. For example, email and username fields have unique constraints to prevent duplicates. Foreign keys are used to maintain referential integrity with other related tables.
- Indexing: The email and username fields are indexed for faster search operations during login and user lookups.

Food_items table

Column Name	Data Type	Description
user_id	CHAR(36)	Foreign key referencing the userdata table
label	VARCHAR(255)	Label or name of the food item
kcal	INT	Kilocalories in the food item
protein	DECIMAL(5,2)	Protein content in the food item
carbs	DECIMAL(5,2)	Carbohydrate content in the food item
fats	DECIMAL(5,2)	Fat content in the food item
fibre	DECIMAL(5,2)	Fibre content in the food item
portion_size	DECIMAL(5,2)	Portion size of the food item
weight	DECIMAL(6,2)	Weight of the food item
unit	VARCHAR(10)	Unit of measurement for weight and portion size
timestamp	DATETIME	Timestamp of when the food item was logged

- Purpose: Captures detailed nutritional data for each food item logged by the user. This data supports the nutrition tracking feature of the app.
- Data Types: The table uses appropriate data types, such as decimal for macronutrients, to maintain precision. The portion_size field allows users to record the exact amount consumed.
- Relationships: Includes a foreign key user_id linking to the UserData table, which enables tracking of each user's food intake.
- Performance: Indexes on user_id and timestamp to optimise queries that retrieve a user's food log history.

Daily_totals Table

Column Name	Data Type	Description
user_id	CHAR(36)	Foreign key referencing the userdata table
total_calories	INT	Total calories consumed for the day
total_protein	INT	Total protein consumed for the day
total_fats	INT	Total fats consumed for the day
total_carbs	INT	Total carbohydrates consumed for the day
date	DATE	Date for the daily totals
daily_target	VARCHAR(20)	Daily target for calories, macros, etc.

- Purpose: Stores daily nutritional totals for each user, including calories, macronutrients (protein, fats, carbs), and daily targets.
- Normalisation: This table is in the third normal form (3NF), as it contains only atomic data related to daily nutritional totals, avoiding any redundancy or repeating groups.
- Data Integrity: The user_id field is a foreign key referencing the userdata table, ensuring referential integrity. The combination of user_id and date can be used as a unique constraint to prevent duplicate entries for the same user and day.
- Indexing: The user_id and date fields can be indexed for faster lookups and filtering operations.

Day_exercises Table:

Column Name	Data Type	Description
day_exercise_id	INT	Primary key, unique identifier for each day exercise
day_id	INT	Foreign key referencing the workout_days table
ExerciseID	INT	Foreign key referencing the exercises table
sets	INT	Number of sets for the exercise
reps	INT	Number of repetitions for the

		exercise

- Purpose: Stores the exercises assigned to each workout day, including the number of sets and repetitions.
- Normalisation: This table is in the third normal form (3NF), as it contains only atomic data related to day exercises, avoiding any redundancy or repeating groups.
- Data Integrity: The day_id field is a foreign key referencing the workout_days table, and the ExerciseID field is a foreign key referencing the exercises table, ensuring referential integrity.
- Indexing: The day_id and ExerciseID fields can be indexed for faster lookups and filtering operations.

Exercises Table:

Column Name	Data Type	Description
ExerciseID	INT	Primary key, unique identifier for each exercise
Name	VARCHAR(255)	Name of the exercise

- Purpose: Stores a catalogue of exercises that can be included in workout plans
- Normalisation: This table is in the third normal form (3NF), as it contains only atomic data related to exercises, avoiding any redundancy or repeating groups.
- Data Integrity: The ExerciseID field is the primary key, ensuring uniqueness of each exercise entry.

Workout_days Table:

Column Name	Data Type	Description
day_id	INT	Primary key, unique identifier for each workout day
plan_id	INT	Foreign key referencing the workout_plans table
day_number	INT	Number indicating the day in the workout plan

- Purpose: Stores the structure of workout plans, including the sequence of days and their mapping to specific exercises.

- Normalisation: This table is in the third normal form (3NF), as it contains only atomic data related to workout days, avoiding any redundancy or repeating groups.
- Data Integrity: The plan_id field is a foreign key referencing the workout_plans table, ensuring referential integrity.

Workout_plans table

Column Name	Data Type	Description
plan_id	INT	Primary key, unique identifier for each workout plan
user_id	CHAR(36)	Foreign key referencing the userdata table
plan_name	VARCHAR(255)	Name of the workout plan

- Purpose: Stores the workout plans created by users, including the plan name and the associated user.
- Normalisation: This table is in the third normal form (3NF), as it contains only atomic data related to workout plans, avoiding any redundancy or repeating groups.
- Data Integrity: The user_id field is a foreign key referencing the userdata table, ensuring referential integrity.

Chats table

Column Name	Data Type	Description
chat_id	INT	Primary key, auto-increment identifier for each chat
user_id	CHAR(36)	Foreign key referencing the userdata table
AI	VARCHAR(255)	Name or identifier of the AI assistant
timestamp	DATETIME	Timestamp of the chat
sender	TEXT	Sender of the message (user or AI)

- Purpose: Logs interactions between the AI coach and the user, preserving the context and continuity of conversations.
- Auto-Incrementing ID: Utilises an auto-increment chat_id for unique identification and easy retrieval of conversation threads.

- Timestamps: Uses a datetime field to store the exact time of each message, crucial for reconstructing the flow of conversation.

2.5 Samples of possible SQL queries

Updating User Profile:

```
UPDATE userdata
SET gender = %s, age = %s, height = %s, weight = %s, experience_level =
%s, body fat = %s, activity_level = %s, goal = %s, calories = %s,
equipment = %s, training_style = %s, training_frequency = %s,
prioritized_muscle_groups = %s, timestamp = %s
WHERE id = %s;
```

Inserting New User:

```
INSERT INTO userdata (username, email, password, id)
VALUES (%s, %s, %s, %s);
```

Adding User ID:

```
INSERT INTO userdata (id)
VALUES (%s);
```

Retrieving User by Username:

```
SELECT id, username, password
FROM userdata
WHERE username = %s;
```

Updating Exercise Plan:

```
UPDATE exercise_plans
SET day1 = %s, day2 = %s, ..., day7 = %s
WHERE user_id = %s;
```

Inserting a Chat Message:

```
INSERT INTO chats (user_id, message, timestamp, sender)
VALUES (%s, %s, %s, %s);
```

Getting Chat History for a User:

```
DELETE FROM chats
WHERE chat_id = %s;
```

Deleting a Chat Message:

```
DELETE FROM chats
WHERE chat_id = %s;
```

Logging Food Items:

```
INSERT INTO food_items (user_id, label, kcal, protein, carbs, fats,
fiber, portion_size, weight, unit, timestamp)
VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s);
```

Updating Daily Totals After Food Logging:

```
REPLACE INTO daily_totals (user_id, total_calories, total_protein,
total_fats, total_carbs, date)
SELECT user_id, SUM(kcal), SUM(protein), SUM(fats), SUM(carbs),
DATE(timestamp)
FROM food_items
WHERE user_id = %s AND DATE(timestamp) = %s
GROUP BY user_id, DATE(timestamp);
```

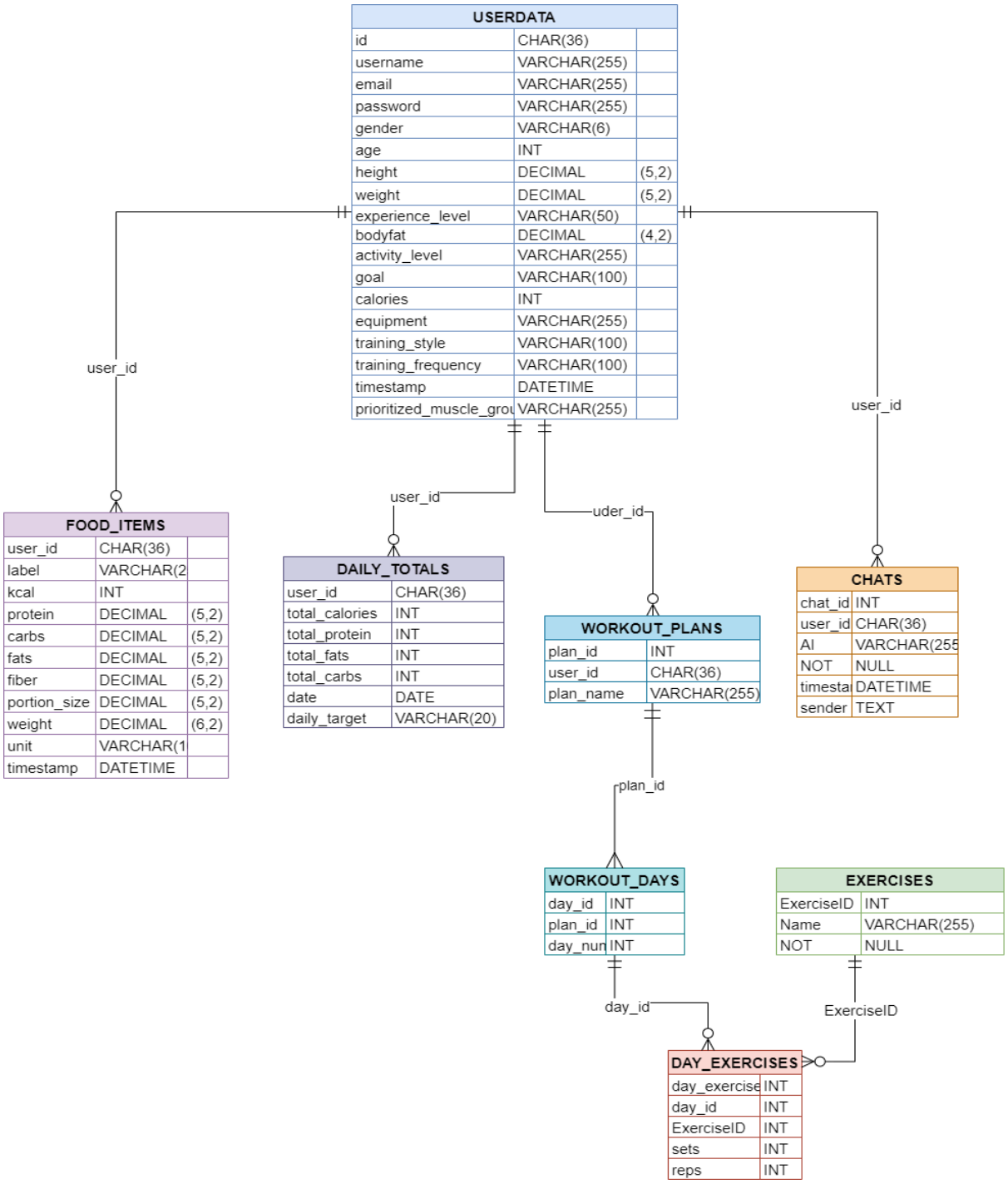
Retrieving Food Items for a Specific Date:

```
SELECT daily_target, total_calories, total_protein, total_fats,
total_carbs
FROM daily_totals
WHERE user_id = %s AND DATE(date) = %s;
```

Retrieving Daily Nutritional Values:

```
SELECT daily_target, total_calories, total_protein, total_fats,
total_carbs
FROM daily_totals
WHERE user_id = %s AND DATE(date) = %s;
```

2.6 Entity Relationship Diagram (ERD)



Relationships:

- **USERDATA** ||--o{ **FOOD_ITEMS** : "user_id"
  - Type: One-to-Many
  - Explanation: A single user (USERDATA) can log multiple food items (FOOD_ITEMS). The user_id in FOOD_ITEMS acts as a foreign key referencing the primary key (id) in USERDATA.

- **USERDATA** ||--o{ **DAILY_TOTALS** : "user_id"
  - Type: One-to-Many
  - Explanation: A single user (USERDATA) can have multiple daily nutritional summaries (DAILY_TOTALS). The user_id in DAILY_TOTALS acts as a foreign key referencing the primary key (id) in USERDATA.

- **USERDATA** ||--o{ **WORKOUT_PLANS** : "user_id"
  - Type: One-to-Many
  - Explanation: One user (USERDATA) can have multiple workout plans (WORKOUT_PLANS). The user_id in WORKOUT_PLANS acts as a foreign key referencing the primary key (id) in USERDATA.

- **USERDATA** ||--o{ **CHATS** : "user_id
  - Type: One-to-Many
  - Explanation: One user (USERDATA) can have many chat entries (CHATS). The user_id in CHATS acts as a foreign key referencing the primary key (id) in USERDATA.

- **WORKOUT_DAYS** ||--o{ **DAY_EXERCISES** : "day_id"
  - Type: One-to-Many
  - Explanation: A single workout day (WORKOUT_DAYS) can have multiple exercises tied to it (DAY_EXERCISES), with the day_id serving as a foreign key to link them.

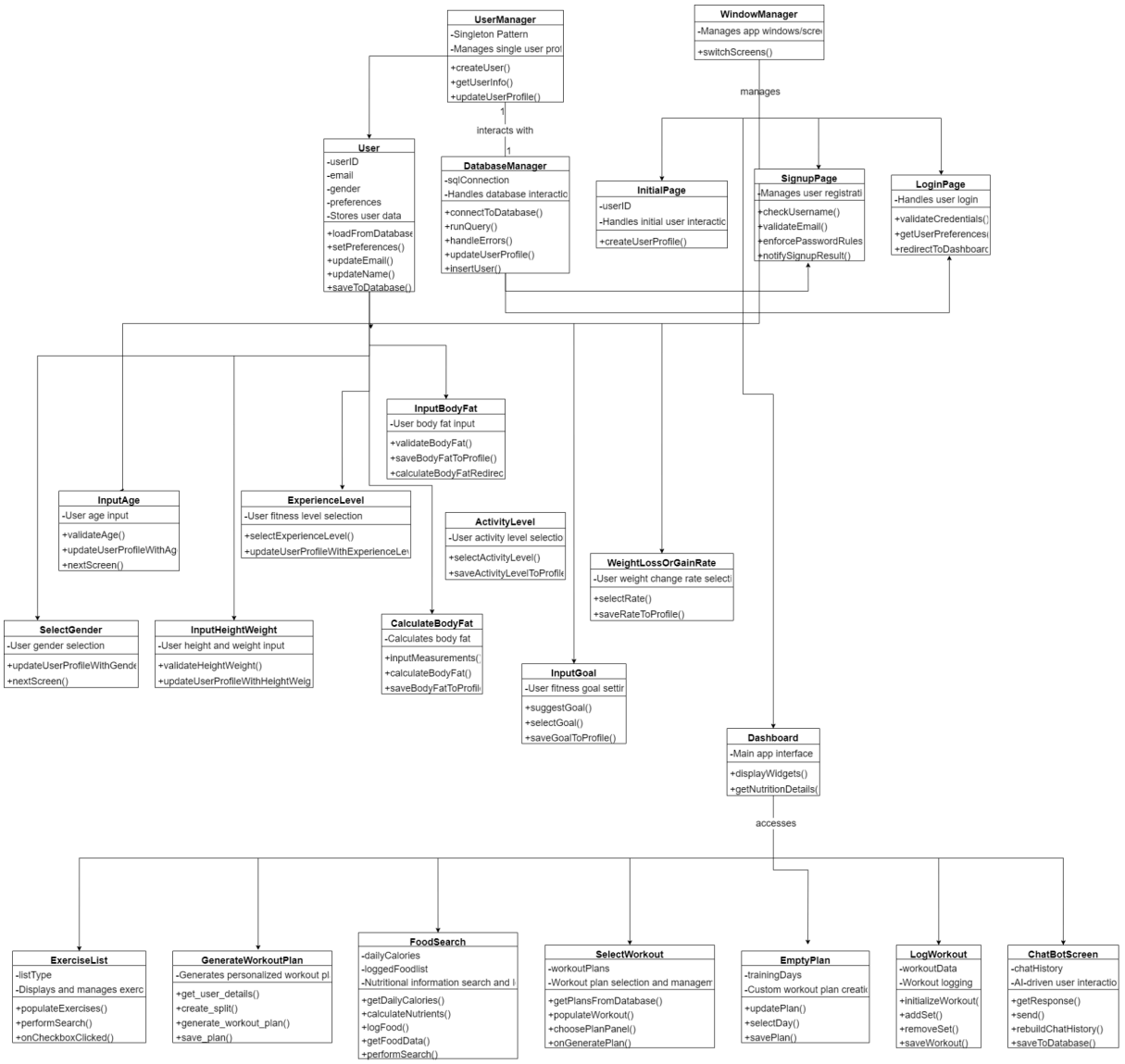
- **EXERCISES** ||--o{ **DAY_EXERCISES** : "ExerciseID"
  - Type: One-to-Many
  - Explanation: A single defined exercise (EXERCISES) can be included in multiple workout days (DAY_EXERCISES). The ExerciseID serves a foreign key.

- **WORKOUT_PLANS** ||--o{ **WORKOUT_DAYS** : "plan_id"
  - Type: One-to-Many
  - Explanation: A single workout plan (WORKOUT_PLANS) can include multiple workout days (WORKOUT_DAYS). plan_id is the foreign key.

2.7 IPSO table

Input	Processing	Storage	Output
User stats: weight, height, age, gender	Calculate calorie target from weight and height	User Profile table stores user stats, goals, preferences	Display user Calorie target on profile
User goals: lose weight, gain muscle etc	Based on user goals and BMI, calculate recommended daily calories and macronutrients	Nutrition Planner module stores recommended nutrition plans	Show user their recommended daily nutrition info
User workout history and experience	Generate custom workout plan based on goals, experience, injury history	Workout Planner module stores generated workout plans	Display user's customised workout schedule
User exercise equipment access	Substitute unavailable exercises in plan with equipment alternatives	Exercise Library module stores exercise details, videos	Show exercises in plan with links to instructional videos
Meal nutrition data: calories, carbs, protein, fat	Compare meals logged to daily recommended nutrition	Daily Nutrition table stores nutrition consumed	Display remaining calories and macros for day
Workout data: exercises, sets, reps, weight	Calculate volume lifted for each exercise over time	Progress Tracking module stores exercise metrics over time	Display weight lifting progression chart
User questions and requests	Respond with personalised answers and tips using AI	AI Coach module stores conversation history	Display conversational responses from AI assistant

2.8 OOP Diagram



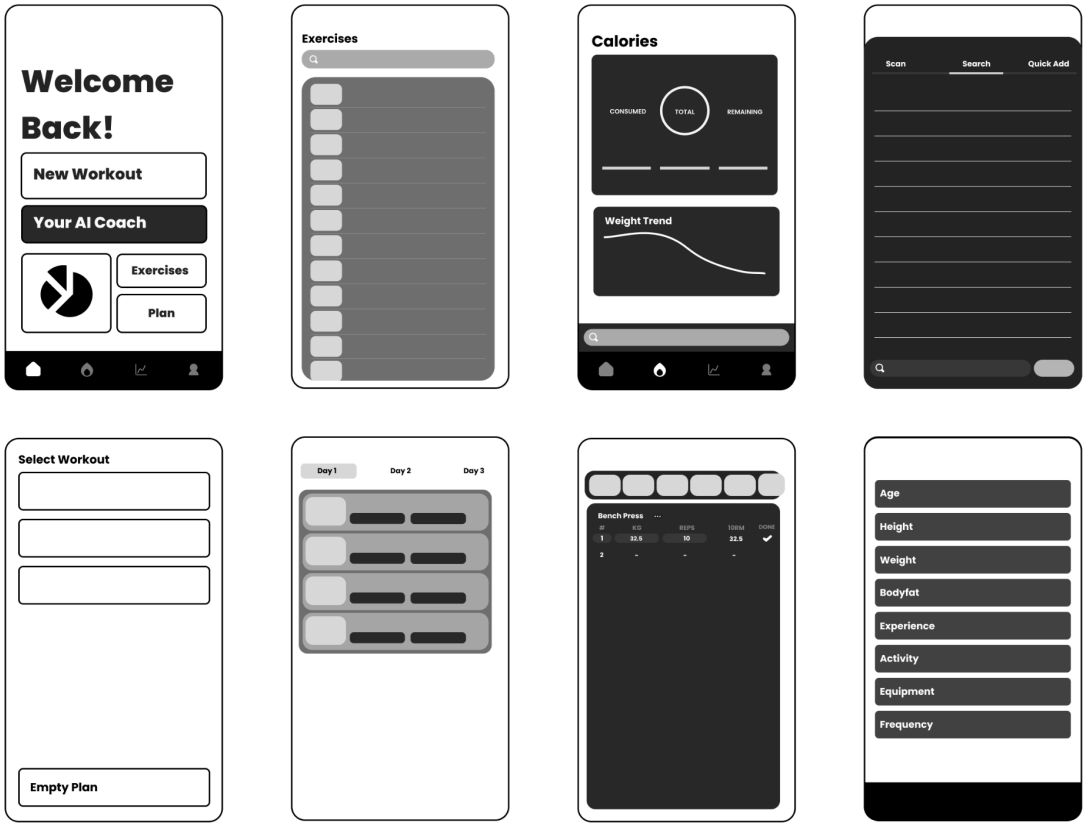
2.9 Low Fidelity User Interface

2.9.1 User Onboarding

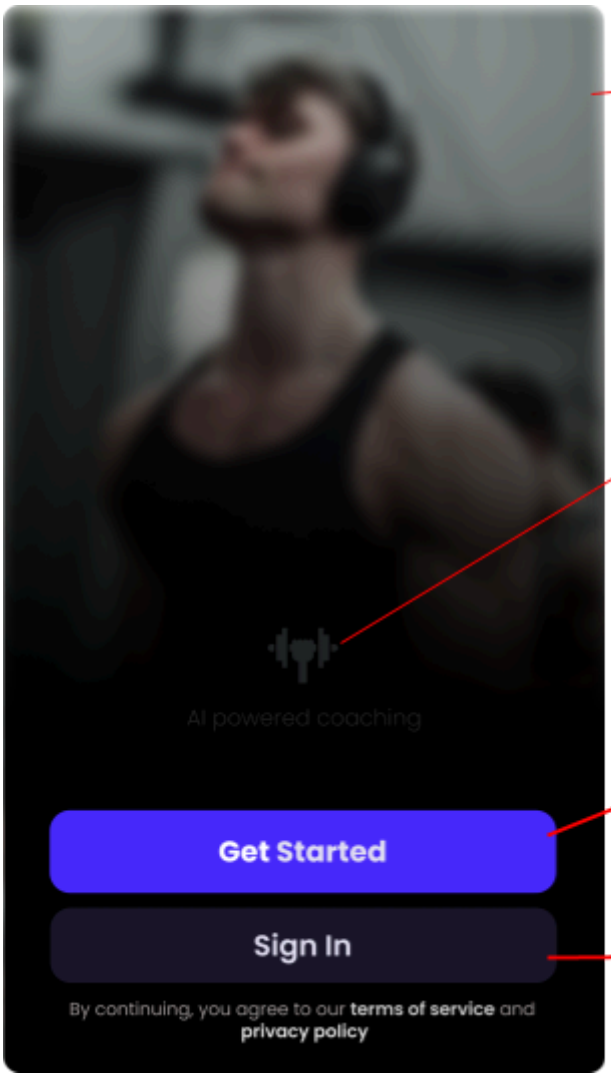
This is a rough layout for how the user onboarding process will be.


Get Started
Sign In
Sign Up
Select your gender
Input your age
Input your height and weight
What is your experience level
Do you know your bodyfat percentage
Calculate bodyfat
What's your activity level?
What is your goal?
Your Daily calories is
What type of equipment do you have access to?
What do you want to train for?
How many days per week do you want to train for?
What muscles do you want to prioritize?
Sign Up

2.9.2 Dashboard and other functionalities



2.10 High Fidelity User Interface



First page of the app

Logo of the app

Button to start the app to collect data and other informations about the use

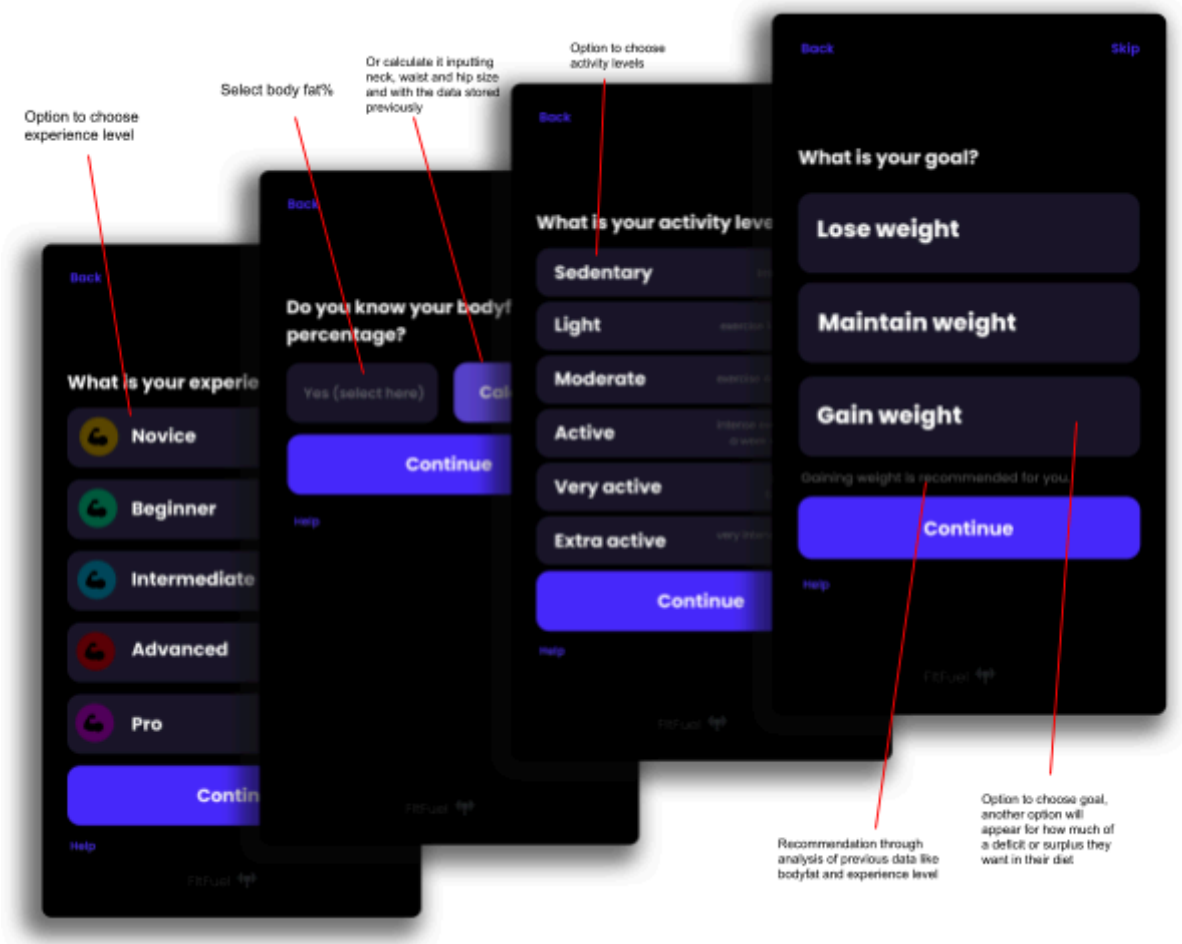
Button to sign in if you already have an account on the app

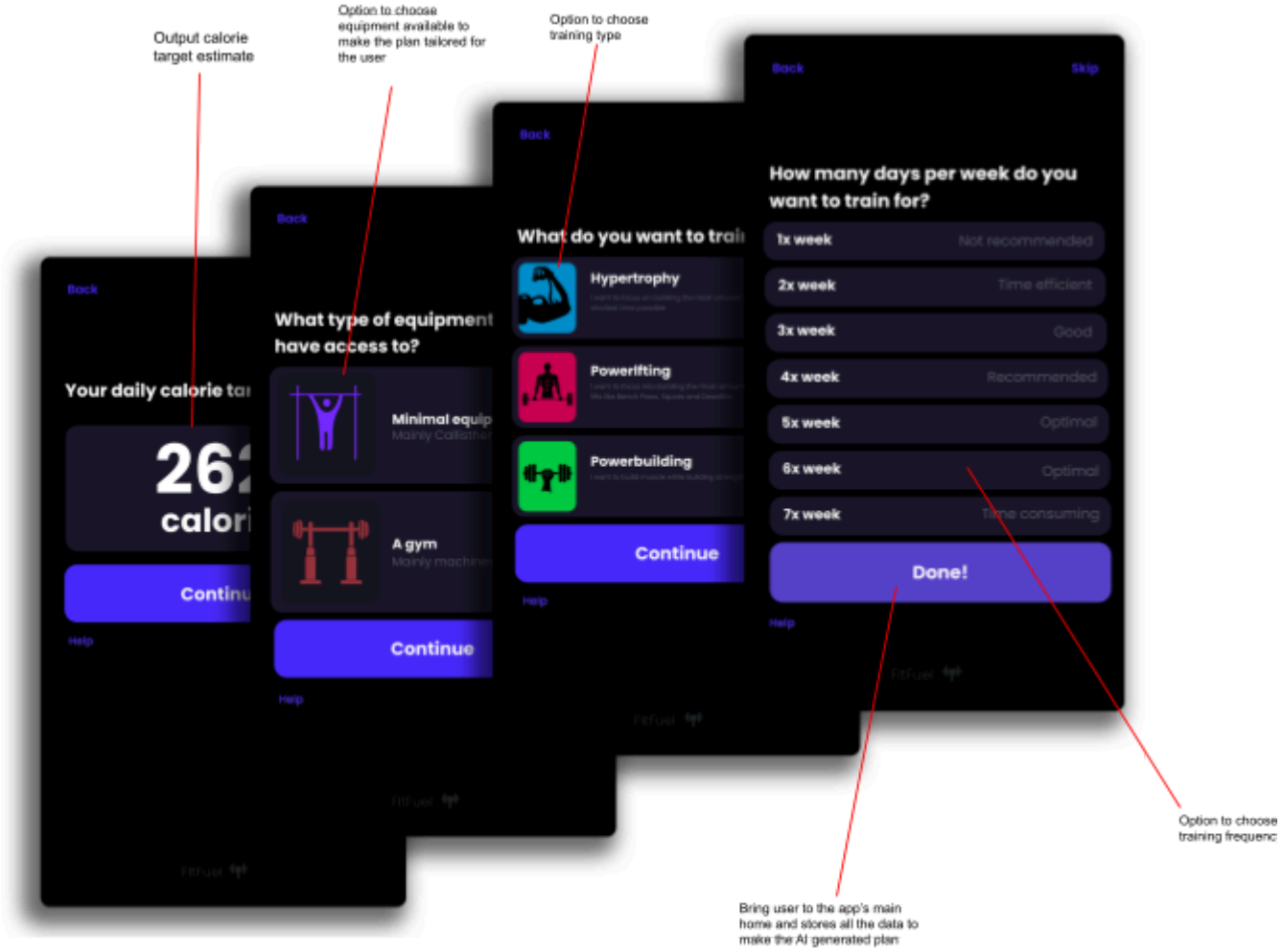
Dark colour scheme, blue as primary colour

I made the UI using a website called Figma, edited pictures and icons using Photoshop

Figma


Option to not say anything if the user doesn't want to
Option to select gender
Option to go back
Input age
Option to skip the whole survey
Input height and weight
Toggle between imperial and metric
Skip
Back
What is your gender?
Male
Female
Prefer not to say
Help
How old are you?
21
Continue
What is your height and weight?
175.70
cm
67.20
kg
Option that will tell the user why the app needs the data
App name and logo
Button to continue


Option to choose experience level
Select body fat%
Or calculate it inputting neck, waist and hip size and with the data stored previously
Option to choose activity levels
Back
Skip
What is your experie
Novice
Beginner
Intermediate
Advanced
Pro
Contin
Help
Do you know your bodyf percentage?
Yes (select here)
Continue
What is your activity leve
Sedentary
Light
Moderate
Active
Very active
Extra active
Continue
What is your goal?
Lose weight
Maintain weight
Gain weight
Continue
Recommendation through analysis of previous data like bodyfat and experience level
Option to choose goal, another option will appear for how much of a deficit or surplus they want in their diet



(this design may be changed in future iterations)

2.11 Test Plan

Objective	Purpose of test	Test Data	Expected Result	Comments
1.4.1	Validate the workout plan algorithm tailors to different user profiles and preferences.	User input data sets for age, weight, goals, and experience level.	Algorithm generates a customised workout plan targeting all major muscle groups and matching the user's experience level.	Test with diverse user profiles ranging from beginner to advanced.
1.4.2	Verify the coverage and functionality of the food tracking database.	Diverse entries of food items including various cuisines,	The database recognizes and logs all entered food	Ensure that the food database is comprehensive and

		brands, and whole foods.	items. Display accurate caloric and macronutrient information in the user interface.	interactions are logged correctly.
1.4.3	Ensure the app provides accurate nutritional breakdowns and adjusts for portion sizes.	Various meals with different portion sizes and units of measurement.	The app displays correct nutritional information adjusted for portion size and unit changes.	Test the flexibility and accuracy of the meal logging feature.
1.4.4	Test manual food entry functionality for custom meals.	Custom meal entries with user-defined calorie and macronutrient values.	The app allows for manual food entry and correctly retrieves and displays saved food information.	Check for ease of use in the UI and proper connection to the database.
1.4.6	Evaluate the quality and accessibility of the exercise library and technique guidance.	Search queries for different exercises.	The exercise library provides a detailed guide on exercise techniques with images and is easily searchable.	Review the exercise content and search functionality with test users.
1.4.7	Security testing for authentication/data handling.	Simulated login attempts, access to encrypted data.	Secure login, encrypted and secure personal data storage.	Use hash and encryption verification methods.
1.4.8	Conduct security testing on user authentication and data handling.	Attempted logins and access to personal data.	User authentication is secure with hashed passwords. Personal data is encrypted and stored securely.	Perform penetration testing and check encryption methods.
1.4.8	Perform usability testing on the app's interface for navigation and design effectiveness.	Navigation paths through the app's interface.	Users find the interface engaging and easy to navigate without any design	Test with a small group to identify UX/UI issues.

			flaws.	
1.4.10	Test the tracking features for accuracy and clarity in displaying progress.	Sample data for body measurements, workout progress, and dietary intake.	The app accurately tracks and displays progress in a visually clear and understandable manner.	Check for the precision of visual progress representations.
1.4.11	Evaluate the AI-powered conversational coach's ability to respond accurately to user queries.	A series of predefined user queries.	The AI chatbot provides helpful and accurate responses to the queries.	Monitor the chatbot's performance and user satisfaction.

Section 3: Technical Solution:

3.1:Main Functionalities:

User Profile Management:
I implemented a feature where users could register and personalise their profiles, including essential details like age, gender, weight, height, and fitness experience level. This information was crucial for customising workout and nutrition plans, making sure that each user received a tailored experience that aligned with their fitness goals.

Personalised Workout Plans:
I developed an algorithm capable of generating customised workout plans based on the user's input data. This algorithm considered various factors, such as the user's fitness level (beginner, intermediate, advanced), available equipment, workout preferences, and specific fitness goals. I ensured the plans were diverse, incorporating exercises that targeted all major muscle groups for a balanced workout routine.

Nutrition Tracking:
Integration of a food database was a significant part of the project, allowing users to log their daily food intake easily. I implemented functionality to track calories and macronutrients, displaying this information in graph format. This feature was designed to help users monitor their nutritional intake and make informed dietary choices aligned with their fitness objectives.

Nutritional Breakdown of Meals:
I added a detailed nutritional breakdown for each logged meal, showing the calories, proteins, carbs, and fats. Users had the flexibility to adjust portion sizes and units of

measurement, ensuring they could accurately log their meals regardless of how the food was consumed or packaged.

Extensive Exercise Library and Technique Guidance:
I created an extensive library of exercises, complete with images and detailed guides on proper execution, common mistakes, and tips for each exercise. This was easily accessible through a search function, allowing users to find exercises suited to their workout plans and learn the correct techniques for safe and effective workouts.

Interactive and User-Friendly Interface:
Using the Kivy framework, I developed an engaging and intuitive user interface. The design focused on responsive elements and clear navigation pathways, allowing users to interact with the app seamlessly and access its features effortlessly.

Comprehensive Tracking Features:
The app provided comprehensive tracking features, allowing users to monitor their workout progress, dietary intake, and body measurements over time. I included visual progress graphs and statistical overviews to give users a clear and motivating overview of their fitness journey.

AI-powered Conversational Coach:
I introduced an AI-powered chatbot to offer users workout tips, nutrition advice, and motivational support. This conversational coach was designed to respond accurately to a variety of user queries, providing personalised guidance and improving the overall user experience.

3.2: Technologies Used

Python: The core programming language used for developing the app's backend logic. Python's versatility and readability make it ideal for handling the app's complex algorithms and data processing tasks.

Kivy: A Python library for developing multitouch applications. It is used for creating the app's user interface, providing a smooth and dynamic user experience across various devices and platforms.

KivyMD: An extension of Kivy that adheres to the Material Design principles. It enhances the visual elements and user interactions within the app, offering a modern and clean interface.

MySQL: Employed for database management. It stores and manages user data, workout plans, and nutritional information, ensuring data persistence, security, and efficient retrieval.

Requests: A simple HTTP library for Python, used to send HTTP requests easily. Used for interacting with the food database API by fetching data from the web within the app.

Langchain: A library which offers chatbot functionality and natural language processing tasks through the OpenAI API.

Bcrypt: A library for hashing passwords. It's a secure way to store user passwords in the database, as it provides cryptographic hashing with salting.

Class Name	Inheritance	Description
UserManager	Singleton Pattern	Handles all things user-related. Ensures only one user is active at a time and interacts with the database to store and retrieve user data.
User	Object	Represents a single user within the app. Stores things like email, age, etc., and has methods to change those details.
DatabaseManager	Object	Handles database interactions for user profiles, exercise info, workout plans, and preferences. Prioritises secure database practices
WindowManager	ScreenManager	Controls screen changes in the app, making navigation smooth.
InitialPage	Screen	The app's landing page – the first thing users see.
SignupPage	Screen	Where new users create their accounts. Checks that their input is valid before creating an account.
LoginPage	Screen	Where users log in. Checks usernames/passwords and lets users into the main app area if correct.
ActivityLevel, InputGoal, WeightLossOrGainRate,InputEquipment,InputBodyfat, InputTrainingStyle,InputTrainingFrequency,InputAge, SelectGender, ExperienceLevel	Screen	These screens all capture different user info during setup to tailor the app experience.
DisplayCalories	Screen	Displays the calculated daily calorie intake for the user based on their fitness goals and other metrics.
CalculateBodyFat	Screen	Calculates the user's body fat percentage based on provided measurements.
PrioritizeMuscleGroups	Screen	Enables users to select muscle groups they want to focus on for their workouts.
GeneratePlan	Screen	It considers the user's goals, experience, and available equipment to create personalised routines.
Dashboard	Screen	The main "home" screen after login. Shows progress updates, daily calories, etc.
ExerciseList	Screen	A searchable library of exercises. Users can view details and add exercises to their workouts.
FoodSearch	Screen	Lets users search for food, view nutritional info, and

		log what they eat.
SelectWorkout	Screen	Where users choose a workout plan, whether it's pre-made, self-created, or just a single workout for the day.
EmptyPlan	Screen	A blank canvas for users to build their own custom workout plans.
LogWorkout	Screen	This screen lets users track their workouts as they happen, logging sets, reps, and weights.
ChatBotScreen	Screen	Where users interact with the AI chatbot for workout/nutrition tips and general motivation.
FitnessApp	MDApp	The main application class responsible for initialising and running the application.

CustomWidget classes:

NavBar	CommonElevationBehavior, MDFloatLayout	A custom navigation bar for layout management with material design effects.
SavedFoodListItem	TwoLineListItem	An item component for a saved food list with two lines of text and a custom right container.
RoundedRectProgressBar	Widget	A widget displaying a progress bar with rounded corners and a label indicating progress.
SwipeToDeleteItem	MDCardSwipe, EventDispatcher	A swipeable card item supporting swipe-to-delete functionality and content switching or removal.
DraggableArea	FloatLayout	A floating layout area supporting drag interactions within draggable panels or containers.
DraggablePanel	FloatLayout	A customizable panel that can be dragged up or down with position and behavior control properties.
RoundedRectangle2	Button	A button with rounded corners and customizable colors for

		normal and pressed states.
CircularProgressBar	AnchorLayout	A circular progress bar component with customizable bar color, thickness, and text.
HorizontalProgressBar	BoxLayout	A horizontal progress bar with a label and customizable bar color, height, and maximum value.
ProfileCard	MDFloatLayout, CommonElevationBehavior	A float layout designed for user profile information display with material design elevation.
CalendarWidget	BoxLayout	A widget displaying a calendar view with a customizable top bar and scrollable content.
BodyweightGraph	BoxLayout	A layout for displaying a graph of bodyweight changes over time with data management properties.
BottomNavExampleScreen	MDScreen	A screen example showcasing bottom navigation bar usage.
HorizontalBar	MDBoxLayout	A layout component representing a horizontal bar with a customizable position property.
RoundedButton	Button	A custom button with rounded corners and specific color properties for default and pressed states.
SelectableRectangle	RoundedButton	A rounded button toggleable between selected and unselected states.
LimitTextInput	TextInput	A text input with a character limit and optional dot auto-insertion functionality.
FoodListItem	TwoLineListItem	A list item for food with a custom right container, similar to SavedFoodListItem.
ListCard	MDCard	A card for listing items with editable details.
ExerciseItemPlan	RecycleDataViewBehavior,	A list item for exercise plans

	TwoLineAvatarListItem	with a checkbox, text, and image.
RoundedImage	ButtonBehavior, Image	An image component with button behavior for displaying rounded images.
ExerciseItem	TwoLineAvatarListItem	A two-line list item for exercises with text, secondary text, and an image.
SmallerTouchIconButton	MDIconButton	An icon button with a reduced touch area for finer touch interactions.
SetExercisePlan	MDCard, EventDispatcher	A card for setting exercise plans with adjustable sets and reps and repositionable exercises.
SavePlanItem	MDFloatLayout	A layout displaying saved plan items with a day number and exercise images list.
ImageButton	ButtonBehavior, Image	A basic button with image content.
WorkoutImage	ImageButton	An image button for workout content with adjustable opacity.
WorkoutRow	MDGridLayout	A grid layout row for workout set details with customizable properties and animations.
SetNumber	RoundedRectangle2	A rounded button for selecting a number, typically within a workout context.
ChooseItem	MDFloatLayout	A float layout for item selection in various contexts within the app.

3.3: SQL Tables:

3.3.1 Table creation

```
CREATE TABLE userdata (
    id CHAR(36) PRIMARY KEY,
    username VARCHAR(255),
    email VARCHAR(255),
```

```
    password VARCHAR(255),
    gender VARCHAR(6),
    age INT,
    height DECIMAL(5,2),
    weight DECIMAL(5,2),
    experience_level VARCHAR(50),
    bodyfat DECIMAL(4,2),
    activity_level VARCHAR(255),
    goal VARCHAR(100),
    calories INT,
    equipment VARCHAR(255),
    training_style VARCHAR(100),
    training_frequency VARCHAR(100),
    timestamp DATETIME,
    prioritized_muscle_groups VARCHAR(255)
);

CREATE TABLE food_items (
    user_id CHAR(36),
    label VARCHAR(255),
    kcal INT,
    protein DECIMAL(5,2),
    carbs DECIMAL(5,2),
    fats DECIMAL(5,2),
    fiber DECIMAL(5,2),
    portion_size DECIMAL(5,2),
    weight DECIMAL(6,2),
    unit VARCHAR(10),
    timestamp DATETIME,
    FOREIGN KEY (user_id) REFERENCES userdata(id)
);

CREATE TABLE daily_totals (
    user_id CHAR(36),
    total_calories INT,
    total_protein INT,
    total_fats INT,
    total_carbs INT,
    date DATE,
    daily_target VARCHAR(20),
    FOREIGN KEY (user_id) REFERENCES userdata(id)
);
CREATE TABLE day_exercises (
    day_exercise_id INT AUTO_INCREMENT PRIMARY KEY,
    day_id INT,
    ExerciseID INT,
    sets INT,
    reps INT,
    FOREIGN KEY (day_id) REFERENCES workout_days(day_id),
    FOREIGN KEY (ExerciseID) REFERENCES exercises(ExerciseID)
);

CREATE TABLE exercises (
    ExerciseID INT AUTO_INCREMENT PRIMARY KEY,
    Name VARCHAR(255) NOT NULL
);

CREATE TABLE workout_days (
    day_id INT AUTO_INCREMENT PRIMARY KEY,
    plan_id INT,
    day_number INT,
    FOREIGN KEY (plan_id) REFERENCES workout_plans(plan_id)
);

CREATE TABLE workout_plans (
    plan_id INT AUTO_INCREMENT PRIMARY KEY,
    user_id CHAR(36),
    plan_name VARCHAR(255),
    FOREIGN KEY (user_id) REFERENCES userdata(id)
);

CREATE TABLE chats (
    chat_id INT AUTO_INCREMENT PRIMARY KEY,
    user_id CHAR(36),
    AI VARCHAR(255) NOT NULL,
    timestamp DATETIME,
    sender TEXT,
```

```
    FOREIGN KEY (user_id) REFERENCES userdata(id)
);

mysql> DESCRIBE chats;
+-----------+--------------+------+-----+---------+----------------+
| Field     | Type         | Null | Key | Default | Extra          |
+-----------+--------------+------+-----+---------+----------------+
| chat_id   | int          | NO   | PRI | NULL    | auto_increment |
| user_id   | char(36)     | YES  | MUL | NULL    |                |
| AI        | varchar(255) | NO   |     | NULL    |                |
| timestamp | datetime     | YES  |     | NULL    |                |
| sender    | text         | YES  |     | NULL    |                |
+-----------+--------------+------+-----+---------+----------------+
5 rows in set (0.01 sec)

mysql> DESCRIBE daily_totals;
+----------------+-------------+------+-----+---------+-------+
| Field          | Type        | Null | Key | Default | Extra |
+----------------+-------------+------+-----+---------+-------+
| user_id        | char(36)    | YES  | MUL | NULL    |       |
| total_calories | int         | YES  |     | NULL    |       |
| total_protein  | int         | YES  |     | NULL    |       |
| total_fats     | int         | YES  |     | NULL    |       |
| total_carbs    | int         | YES  |     | NULL    |       |
| date           | date        | YES  |     | NULL    |       |
| daily_target   | varchar(20) | YES  |     | NULL    |       |
+----------------+-------------+------+-----+---------+-------+
7 rows in set (0.00 sec)

mysql> DESCRIBE day_exercises;
+-----------------+------+------+-----+---------+----------------+
| Field           | Type | Null | Key | Default | Extra          |
+-----------------+------+------+-----+---------+----------------+
| day_exercise_id | int  | NO   | PRI | NULL    | auto_increment |
| day_id          | int  | YES  | MUL | NULL    |                |
| ExerciseID      | int  | YES  | MUL | NULL    |                |
| sets            | int  | YES  |     | NULL    |                |
| reps            | int  | YES  |     | NULL    |                |
+-----------------+------+------+-----+---------+----------------+
5 rows in set (0.00 sec)

mysql> DESCRIBE exercises;
+------------+--------------+------+-----+---------+-------+
| Field      | Type         | Null | Key | Default | Extra |
+------------+--------------+------+-----+---------+-------+
| ExerciseID | int          | NO   | PRI | NULL    |       |
| Name       | varchar(255) | NO   |     | NULL    |       |
+------------+--------------+------+-----+---------+-------+
2 rows in set (0.00 sec)

mysql> DESCRIBE food_items;
+--------------+--------------+------+-----+---------+-------+
| Field        | Type         | Null | Key | Default | Extra |
+--------------+--------------+------+-----+---------+-------+
| user_id      | char(36)     | YES  | MUL | NULL    |       |
| label        | varchar(255) | YES  |     | NULL    |       |
| kcal         | int          | YES  |     | NULL    |       |
| protein      | decimal(5,2) | YES  |     | NULL    |       |
| carbs        | decimal(5,2) | YES  |     | NULL    |       |
| fats         | decimal(5,2) | YES  |     | NULL    |       |
| fiber        | decimal(5,2) | YES  |     | NULL    |       |
| portion_size | decimal(5,2) | YES  |     | NULL    |       |
| weight       | decimal(6,2) | YES  |     | NULL    |       |
| unit         | varchar(10)  | YES  |     | NULL    |       |
| timestamp    | datetime     | YES  |     | NULL    |       |
+--------------+--------------+------+-----+---------+-------+
11 rows in set (0.00 sec)

mysql> DESCRIBE userdata;
+--------------------------+--------------+------+-----+---------+-------+
| Field                    | Type         | Null | Key | Default | Extra |
+--------------------------+--------------+------+-----+---------+-------+
| id                       | char(36)     | NO   | PRI | NULL    |       |
| username                 | varchar(255) | YES  |     | NULL    |       |
| email                    | varchar(255) | YES  |     | NULL    |       |
| password                 | varchar(255) | YES  |     | NULL    |       |
| gender                   | varchar(6)   | YES  |     | NULL    |       |
```

```
| age                        | int          | YES  |     | NULL    |       |
| height                     | decimal(5,2) | YES  |     | NULL    |       |
| weight                     | decimal(5,2) | YES  |     | NULL    |       |
| experience_level           | varchar(50)  | YES  |     | NULL    |       |
| bodyfat                    | decimal(4,2) | YES  |     | NULL    |       |
| activity_level             | varchar(255) | YES  |     | NULL    |       |
| goal                       | varchar(100) | YES  |     | NULL    |       |
| calories                   | int          | YES  |     | NULL    |       |
| equipment                  | varchar(255) | YES  |     | NULL    |       |
| training_style             | varchar(100) | YES  |     | NULL    |       |
| training_frequency         | varchar(100) | YES  |     | NULL    |       |
| timestamp                  | datetime     | YES  |     | NULL    |       |
| prioritized_muscle_groups | varchar(255) | YES  |     | NULL    |       |
+----------------------------+--------------+------+-----+---------+-------+
18 rows in set (0.00 sec)

mysql> DESCRIBE workout_days;
+------------+------+------+-----+---------+----------------+
| Field      | Type | Null | Key | Default | Extra          |
+------------+------+------+-----+---------+----------------+
| day_id     | int  | NO   | PRI | NULL    | auto_increment |
| plan_id    | int  | YES  | MUL | NULL    |                |
| day_number | int  | YES  |     | NULL    |                |
+------------+------+------+-----+---------+----------------+
3 rows in set (0.00 sec)

mysql> DESCRIBE workout_plans;
+-----------+--------------+------+-----+---------+----------------+
| Field     | Type         | Null | Key | Default | Extra          |
+-----------+--------------+------+-----+---------+----------------+
| plan_id   | int          | NO   | PRI | NULL    | auto_increment |
| user_id   | char(36)     | YES  | MUL | NULL    |                |
| plan_name | varchar(255) | YES  |     | NULL    |                |
+-----------+--------------+------+-----+---------+----------------+
3 rows in set (0.00 sec)
```

3.3.2 Example of data in the table:

```
+--------------------------------------+----------+------------------+--------------------------------------------------------------+--------+
| id                                   | username | email            | password                                                     | gender |
+--------------------------------------+----------+------------------+--------------------------------------------------------------+--------+
| 9f4f7305-8907-43d5-9897-35f43eff95eb | john     | john123@gmail.com | $2b$12$ZLLuizVhLaY3zipwZK4QCe4B/0MR7Tk4w6FLE.rnVYeJ1PVivKTJi | male   |
+--------------------------------------+----------+------------------+--------------------------------------------------------------+--------+

+------+--------+--------+-----------------+---------+-----------------------------------------+------------+--------------------------------+
| age  | height | weight | experience_level | bodyfat | activity_level                          | goal       | equipment                      |
+------+--------+--------+-----------------+---------+-----------------------------------------+------------+--------------------------------+
|   18 |   1.70 |  66.00 | intermediate     |   18.00 | Moderately Active, exercise 4-5 times a week | Lose Weight | Gym, mainly weights and machines |
+------+--------+--------+-----------------+---------+-----------------------------------------+------------+--------------------------------+

+----------------+--------------------+---------------------+--------------------------+
| training_style | training_frequency | timestamp           | prioritized_muscle_groups |
+----------------+--------------------+---------------------+--------------------------+
| Hypertrophy    | 5x week            | 2023-11-18 11:09:53 | chest,biceps,calves      |
+----------------+--------------------+---------------------+--------------------------+

mysql> SELECT * FROM food_items;
+--------------------------------------+--------------------------+------+---------+--------+--------+-------+-------------+--------+---------+----------------
-----+
| user_id                              | label                    | kcal | protein | carbs  | fats   | fiber | portion_size | weight | unit    | timestamp
|
+--------------------------------------+--------------------------+------+---------+--------+--------+-------+-------------+--------+---------+----------------
-----+
| 262efaa4-1a2d-484e-8de8-32966c8a6a82 | Chicken                  |  161 |   12.10 |   5.87 |   9.71 |  0.79 |        1.00 | 228.00 | serving | 2024-01-17
01:04:42 |
| 262efaa4-1a2d-484e-8de8-32966c8a6a82 | Chicken Fried Chicken    |  240 |   14.11 |  11.09 |  15.22 |  0.46 |        1.00 | 286.00 | serving | 2024-01-17
01:04:42 |
| 262efaa4-1a2d-484e-8de8-32966c8a6a82 | Chicken                  |  215 |   18.60 |   0.00 |  15.10 |  0.00 |        1.00 | 250.00 | serving | 2024-01-17
01:04:42 |
| 262efaa4-1a2d-484e-8de8-32966c8a6a82 | Chicken Broth, Chicken   |   11 |    0.47 |   0.47 |   0.59 |  0.00 |        1.00 | 425.00 | serving | 2024-01-17
01:04:42 |
| 262efaa4-1a2d-484e-8de8-32966c8a6a82 | Chicken Broth, Chicken   |   11 |    0.47 |   0.47 |   0.59 |  0.00 |        1.00 | 425.00 | serving | 2024-01-17
01:04:42 |
| 262efaa4-1a2d-484e-8de8-32966c8a6a82 | Chicken Stock, Chicken   |    4 |    0.82 |   0.00 |   0.00 |  0.00 |        1.00 | 245.00 | serving | 2024-01-17
01:04:42 |
+--------------------------------------+--------------------------+------+---------+--------+--------+-------+-------------+--------+---------+----------------
-----+

mysql> SELECT * FROM workout_days;
+--------+---------+------------+
| day_id | plan_id | day_number |
+--------+---------+------------+
|     45 |      30 |          1 |
|     46 |      30 |          2 |
```

```
|     47 |      30 |          3 |
|     48 |      30 |          4 |
|     49 |      30 |          5 |
|     50 |      30 |          6 |
|     51 |      30 |          7 |
|     52 |      31 |          1 |
|     53 |      31 |          2 |
|     54 |      31 |          3 |
|     55 |      32 |          1 |
|     56 |      32 |          2 |
|     57 |      32 |          3 |
+--------+---------+------------+
13 rows in set (0.00 sec)

mysql> SELECT * FROM workout_plans;
+---------+--------------------------------------+------------+
| plan_id | user_id                              | plan_name  |
+---------+--------------------------------------+------------+
|      30 | 262efaa4-1a2d-484e-8de8-32966c8a6a82 | New Plan 3 |
|      31 | 262efaa4-1a2d-484e-8de8-32966c8a6a82 | New Plan   |
|      32 | 4b5cf120-9852-446c-97db-cd15250f6ef1 | 3 day plan |
+---------+--------------------------------------+------------+
3 rows in set (0.00 sec)

mysql> SELECT * FROM exercises;
+------------+-----------------------------------+
| ExerciseID | Name                              |
+------------+-----------------------------------+
|          1 | Hanging Leg Raise                 |
|          2 | Bar Dip                           |
|          3 | One-Handed Bar Hang               |
|          4 | Back Extension                    |
|          5 | Lying Leg Raise                   |
|          6 | Pull-Up                           |
|          7 | Crunch                            |
|          8 | Plank                             |
|          9 | Side Plank                        |
|         10 | Chin-Up                           |
|         11 | Kneeling Plank                    |
|         12 | Kneeling Side Plank               |
|         13 | Incline Push-Up                   |
|         14 | Kneeling Push-Up                  |
|         15 | Push-Up                           |
|         16 | Glute Bridge                      |
|         17 | Bodyweight Squat                  |
|         18 | BodyWeight Lunge                  |
|         19 | Bench Dip                         |
|         20 | Close-Grip Push-Up                |
|         21 | Tricep Bodyweight Extension       |
|         22 | Inverted Row                      |
|         23 | Decline Push-up                   |
|         24 | Assisted one leg squat            |
|         25 | Pistol Squat                      |
|         26 | Sissy Squat                       |
|         27 | Reverse Cable Flyes               |
|         28 | Cable Lateral Raise               |
|         29 | Tricep Pushdown With Rope         |
|         30 | Tricep Pushdown With Bar          |
|         31 | Cable Crunch                      |
|         32 | Cable Curl With Bar               |
|         33 | Overhead Cable Triceps Extension  |
|         34 | Cable Wide Grip Seated Row        |
|         35 | High to Low Cable Chest Fly       |
|         36 | Mid Cable Chest Fly               |
|         37 | Low to High Cable Chest Fly       |
|         38 | Cable Close Grip Seated Row       |
|         39 | Dumbbell Romanian Deadlift        |
|         40 | Stiff-Legged Deadlift             |
|         41 | Dumbbell Curl                     |
|         42 | Dumbbell Preacher Curl            |
|         43 | Hammer Curl                       |
|         44 | Standing Calf Raise               |
|         45 | Dumbbell Wrist Curl               |
|         46 | Dumbbell Shoulder Press           |
|         47 | Dumbbell Lying Triceps Extension  |
|         48 | Spider Curl                       |
|         49 | Hanging Knee Raise                |
|         50 | Dumbbell Chest Press              |
|         51 | Incline Dumbbell Press            |
|         52 | Barbell Wrist Extension           |
|         53 | Seated Calf Raise                 |
|         54 | Barbell Wrist Curl                |
|         55 | Romanian Deadlift                 |
|         56 | Dumbbell Lateral Raise            |
|         57 | Dumbbell Shrug                    |
|         58 | Close-Grip Bench Press            |
|         59 | Overhead Press                    |
|         60 | Barbell Shrug                     |
|         61 | Incline Bench Press               |
|         62 | Barbell Curl                      |
|         63 | Barbell Preacher Curl             |
|         64 | Incline Dumbbell Curl             |
|         65 | Bench Press                       |
|         66 | Decline Bench Press               |
|         67 | Good Morning                      |
|         68 | Barbell Lunge                     |
|         69 | Box Squat                         |
|         70 | Dumbbell Rear Delt Row            |
|         71 | Reverse Dumbbell Flyes            |
|         72 | Barbell Upright Row               |
```

```
|         73 | Barbell Lying Triceps Extension     |
|         74 | Bulgarian Split Squat               |
|         75 | Dumbbell Lunge                      |
|         76 | Barbell Back Squat                  |
|         77 | Barbell Triceps Extension           |
|         78 | Front Squat                         |
|         79 | Dumbbell Row                        |
|         80 | Concentration Curl                  |
|         81 | Dumbbell Chest Fly                  |
|         82 | Walking Lunge                       |
|         83 | Barbell Row                         |
|         84 | Dumbbell Pullover                   |
|         85 | Deadlift                            |
|         86 | Barbell Front Raise                 |
|         87 | Dumbbell Front Raise                |
|         88 | Goblet Squat                        |
|         89 | Clean and Jerk                      |
|         90 | Push Press                          |
|         91 | Squat Jerk                          |
|         92 | Hip Thrust                          |
|         93 | Pec Deck                            |
|         94 | Machine Chest Press                 |
|         95 | Machine Shoulder Press              |
|         96 | Machine Lateral Raise               |
|         97 | Seated Smith Machine Shoulder Press |
|         98 | Seated Leg Curl                     |
|         99 | Leg Extension                       |
|        100 | Hack Squat Machine                  |
|        101 | Machine Bicep Curl                  |
|        102 | Lat Pulldown With Pronated Grip     |
|        103 | Belt Squat                          |
|        104 | Leg Press                           |
|        105 | Hip Adduction Machine               |
|        106 | Smith Machine Bench Press           |
|        107 | One-Handed Lat Pulldown             |
|        108 | Seated Machine Row                  |
|        109 | Smith Machine Squat                 |
|        110 | Machine Crunch                      |
|        111 | Reverse Machine Fly                 |
|        112 | Lying Leg Curl                      |
|        113 | Lat Pulldown With Supinated Grip    |
|        114 | Hip Abduction Machine               |
|        115 | One-Handed Cable Row                |
|        116 | Cable Curl With Rope                |
|        117 | Kneeling Ab Wheel Roll-Out          |
+------------+-------------------------------------+
117 rows in set (0.00 sec)


mysql> SELECT * FROM day_exercises;
+-----------------+--------+------------+------+------+
| day_exercise_id | day_id | ExerciseID | sets | reps |
+-----------------+--------+------------+------+------+
|             107 |     45 |          6 |    3 |   12 |
|             108 |     46 |          3 |    3 |   12 |
|             109 |     46 |          5 |    3 |   12 |
|             110 |     46 |          6 |    3 |   12 |
|             111 |     47 |          5 |    3 |   12 |
|             112 |     47 |          6 |    3 |   12 |
|             113 |     47 |          3 |    3 |   12 |
|             114 |     47 |          1 |    3 |   12 |
|             115 |     48 |          1 |    3 |   12 |
|             116 |     48 |          4 |    3 |   12 |
|             117 |     48 |          5 |    3 |   12 |
|             118 |     48 |          6 |    3 |   12 |
+-----------------+--------+------------+------+------+

mysql> SELECT * FROM daily_totals;
+--------------------------------------+----------------+--------------+-----------+-------------+-----------+--------------------+
| user_id                              | total_calories | total_protein | total_fats | total_carbs | date       | daily_target       |
+--------------------------------------+----------------+--------------+-----------+-------------+-----------+--------------------+
| 262efaa4-1a2d-484e-8de8-32966c8a6a82 |           3339 |          136 |       127 |        1057 | 2024-01-14 | 2800, 190, 250, 115 |
| 262efaa4-1a2d-484e-8de8-32966c8a6a82 |            986 |          263 |        49 |         191 | 2024-01-15 | 2800, 190, 250, 115 |
+--------------------------------------+----------------+--------------+-----------+-------------+-----------+--------------------+
```

3.3.3 Database Management in the code

```python
class DatabaseManager:

    def __init__(self, db_config):

        # Initialize the database manager with the given configuration

        self.db_config = db_config

        # Create a connection pool with the given configuration

        self.connection_pool = pooling.MySQLConnectionPool(pool_name="mypool",

                                                           pool_size=2,

                                                           **self.db_config)
```

```
    def start_transaction(self):
        # Start a new transaction
        self.transaction_connection = self.connection_pool.get_connection()
        self.transaction_connection.start_transaction()

    def commit_transaction(self):
        # Commit the current transaction if it exists
        if self.transaction_connection:
            self.transaction_connection.commit()
            self.transaction_connection.close()
            self.transaction_connection = None

    def rollback_transaction(self):
        # Rollback the current transaction if it exists
        if self.transaction_connection:
            self.transaction_connection.rollback()
            self.transaction_connection.close()
            self.transaction_connection = None

    def execute_query(self, query, params=(), commit=False, fetch=None):
        # Execute a query with the given parameters
        connection = self.connection_pool.get_connection()
        result = None
        cursor_created = False
        try:
            cursor = connection.cursor()
            cursor_created = True
            cursor.execute(query, params)
            if commit:
                connection.commit()
            if fetch == 'one':
                result = cursor.fetchone()
            elif fetch == 'all':
                result = cursor.fetchall()
            if query.lower().startswith('insert'):
                result = cursor.lastrowid
        except mysql.connector.Error as e:
            logging.error(f'Database error: {e}')
            if connection.in_transaction:
                connection.rollback()
            raise
        finally:
            if cursor_created:
                self._drain_cursor(cursor)
                cursor.close()
            connection.close()
        return result
```

```python
@staticmethod
def _drain_cursor(cursor):
    # Drain the cursor to ensure all results have been fetched
    try:
        while cursor.nextset():
            pass
    except mysql.connector.Error as e:
        logging.error(f'Error draining cursor: {e}')


# nutrition database methods ----------------------------------------------------------------------------------
def update_user_profile(self, user):
    # Update the user profile in the database
    timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    query = """UPDATE userdata SET gender = %s, age = %s, height = %s, weight = %s,
    experience_level = %s, bodyfat = %s, activity_level = %s, goal = %s, calories = %s, equipment = %s,
    training_style = %s, training_frequency = %s, prioritized_muscle_groups = %s, timestamp= %s WHERE id = %s;"""
    params = (user.gender, user.age, user.height, user.weight, user.experience_level,
              user.bodyfat, user.activity_level, user.goal, user.calories, user.equipment, user.training_style,
              user.training_frequency, user.prioritized_muscle_groups, timestamp, user.username)
    self.execute_query(query, params=params, commit=True)


def insert_user(self, username, email, password):
    # Insert a new user into the database
    if isinstance(password, str):
        password = password.encode('utf-8')
    hashed_password = bcrypt.hashpw(password, bcrypt.gensalt())
    query = "UPDATE userdata SET username = %s, email = %s, password = %s WHERE id = %s;"
    self.execute_query(query, params=(username, email, hashed_password, username), commit=True)


def insert_username(self, username):
    # Get a user from the database by username
    query = "INSERT INTO userdata (id) VALUES (%s)"
    self.execute_query(query, params=(username,), commit=True)


def get_user(self, username):
    # Get a user from the database by username
    query = "SELECT * FROM userdata WHERE username = %s;"
    return self.execute_query(query, params=(username,), fetch='one')


def update_exercise_plan(self, username, plan_data):
    # Update the exercise plan for a user in the database
    query = "SELECT plan_id FROM exercise_plans WHERE username = %s;"
    existing_plan = self.execute_query(query, params=(username,), fetch='one')

    if existing_plan:
        update_parts = [f"{day} = %s" for day in plan_data.keys()]
```

```
            update_query = f"UPDATE exercise_plans SET {', '.join(update_parts)} WHERE username = %s;"
            params = tuple(plan_data.values()) + (username,)
        else:
            columns = ', '.join(plan_data.keys())
            placeholders = ', '.join(['%s'] * len(plan_data))
            insert_query = f"INSERT INTO exercise_plans (username, {columns}) VALUES (%s, {placeholders});"

              params = (username,) + tuple(plan_data.values())

        return self.execute_query(update_query if existing_plan else insert_query, params=params, commit=True)

    def insert_chat(self, username, message, sender):
        # Insert a new chat message into the database
        timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        query = "INSERT INTO chats (username, message, timestamp, sender) VALUES (%s, %s, %s, %s);"
        self.execute_query(query, params=(username, message, timestamp, sender), commit=True)


    def get_chats_by_username(self, username):
        # Get all chat messages for a user from the database
        query = "SELECT * FROM chats WHERE username = %s ORDER BY timestamp DESC;"
        return self.execute_query(query, params=(username,), fetch='all')

    def delete_chat(self, chat_id):
        # Delete a chat message from the database
        query = "DELETE FROM chats WHERE chat_id = %s;"
        self.execute_query(query, params=(chat_id,), commit=True)

    def insert_logged_foods(self, username, label, kcal, protein, carbs, fats, fiber, portion_size, selected_weight, unit_sequence,
date):
        # Insert a new logged food item into the database
        timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")

        query1 = """
        INSERT INTO food_items (username, label, kcal, protein, carbs, fats, fiber, portion_size, weight, unit, timestamp)
        VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s);
        """
        self.execute_query(query1, params=(username, label, kcal, protein, carbs, fats, fiber, portion_size, selected_weight,
 unit_sequence, timestamp), commit=True)
        query2 = """
        REPLACE INTO daily_totals (username, total_calories, total_protein, total_fats, total_carbs, date)
        SELECT username, SUM(kcal), SUM(protein), SUM(fats), SUM(carbs), DATE(timestamp)
        FROM food_items
        WHERE username = %s AND DATE(timestamp) = %s
        GROUP BY username, DATE(timestamp);
        """
        self.execute_query(query2, params=(username, date), commit=True)
```

```
def get_food_items(self, username, date):
    # Get all food items for a user from the database
    query = """
    SELECT username, label, kcal, protein, carbs, fats, fiber, portion_size, weight, unit, timestamp
    FROM food_items
    WHERE username = %s AND DATE(timestamp) = %s;
    """
    return self.execute_query(query, params=(username, date), fetch='all'


def get_daily_values(self, username, date):
    # Get the daily values for a user from the database
    query = """
    SELECT daily_target, total_calories, total_protein, total_fats, total_carbs FROM daily_totals
    WHERE username = %s AND DATE(date) = %s;
    """
    print(query)
    print(username)
    print(date)
    return self.execute_query(query, params=(username, date), fetch='all')


# workout database methods ------------------------------------------------------------------------------------


def create_workout_plan(self, user_id, plan_name):
     # Create a new workout plan in the database
    query = "INSERT INTO workout_plans (user_id, plan_name) VALUES (%s, %s);"
    return self.execute_query(query, params=(user_id, plan_name), commit=True)


def add_workout_day(self, plan_id, day_number):
    # Add a new workout day to a workout plan in the database
    query = "INSERT INTO workout_days (plan_id, day_number) VALUES (%s, %s);"
    return self.execute_query(query, params=(plan_id, day_number), commit=True)


def add_exercise_to_day(self, day_id, exerciseid, sets, reps):
     # Add an exercise to a workout day in the database
    query = "INSERT INTO day_exercises (day_id, exerciseid, sets, reps) VALUES (%s, %s, %s, %s);"
    return self.execute_query(query, params=(day_id, exerciseid, sets, reps), commit=True)


def get_or_create_exercise(self, exercise_name):
    # Get or create an exercise in the database
    query = "SELECT exerciseid FROM exercises WHERE name = %s;"
    result = self.execute_query(query, params=(exercise_name,), fetch='one')


    if result:
        return result[0]
    else:
        insert_query = "INSERT INTO exercises (name) VALUES (%s);"
        return self.execute_query(insert_query, params=(exercise_name,), commit=True)
```

```
def check_and_override_plan(self, user_id, plan_name):
    # Check if a workout plan already exists and override it if necessary

    # Check if the plan already exists
    plan_id_query = "SELECT plan_id FROM workout_plans WHERE user_id = %s AND plan_name = %s;"
    plan_id = self.execute_query(plan_id_query, params=(user_id, plan_name), fetch='one')

    if plan_id:
        # If the plan exists, delete it and its associated days and exercises
        delete_exercises_query = """
        DELETE de FROM day_exercises de
        JOIN workout_days wd ON de.day_id = wd.day_id
        WHERE wd.plan_id = %s;
        """
        self.execute_query(delete_exercises_query, params=(plan_id[0],), commit=True)

        delete_days_query = "DELETE FROM workout_days WHERE plan_id = %s;"
        self.execute_query(delete_days_query, params=(plan_id[0],), commit=True)

        delete_plan_query = "DELETE FROM workout_plans WHERE plan_id = %s;"
        self.execute_query(delete_plan_query, params=(plan_id[0],), commit=True)

def save_complete_workout_plan(self, user_id, plan_name, workout_plan):
    # Save a complete workout plan to the database
    self.start_transaction()
    try:
        # Check if the plan already exists and override it if necessary
        self.check_and_override_plan(user_id, plan_name)

        # Create the new plan
        plan_id = self.create_workout_plan(user_id, plan_name)

        for day_number, exercises in workout_plan.items():
            day_id = self.add_workout_day(plan_id, day_number)

            for exercise in exercises:
                exerciseid = self.get_or_create_exercise(exercise['name'])
                self.add_exercise_to_day(day_id, exerciseid, exercise['sets'], exercise['reps'])

        self.commit_transaction()
    except Exception as e:
        self.rollback_transaction()
        raise e

def retrieve_workout_plan(self, user_id, plan_name):
    # Retrieve a workout plan from the database
```

```
    workout_plan = {}
    plan_id = self.get_plan_id(user_id, plan_name)

    if plan_id:
        days_query = "SELECT day_id, day_number FROM workout_days WHERE plan_id = %s ORDER BY day_number;"

        days = self.execute_query(days_query, params=(plan_id,), fetch='all')

        for day_id, day_number in days:
            workout_plan[day_number] = self.get_exercises_for_day(day_id)

    return workout_plan

def get_exercises_for_day(self, day_id):
    # Get all exercises for a workout day from the database
    ex_query = """
    SELECT e.name, de.sets, de.reps
    FROM day_exercises de
    JOIN exercises e ON de.exerciseid = e.exerciseid
    WHERE de.day_id = %s;
    """
    exercises = self.execute_query(ex_query, params=(day_id,), fetch='all')
    return [{'name': name, 'sets': sets, 'reps': reps} for name, sets, reps in exercises]

def get_plan_id(self, user_id, plan_name):
    # Get the ID of a workout plan from the database
    query = "SELECT plan_id FROM workout_plans WHERE user_id = %s AND plan_name = %s;"
    result = self.execute_query(query, params=(user_id, plan_name), fetch='one')
    return result[0] if result else None

def get_plan_names(self, user_id):
    # Get all plan names for a user from the database
    query = "SELECT plan_name FROM workout_plans WHERE user_id = %s;"
    result = self.execute_query(query, params=(user_id,), fetch='all')
    return [row[0] for row in result] if result else None
```

3.4: Code Overview:

UserManager Class

- Managing a Single User:
    - The code is set up to work with just one user at a time. This design pattern (called "Singleton") is great for making sure my app doesn't get confused by managing multiple user profiles simultaneously.
- Creating Users:
    - Generates a unique ID for each new user.

- Stores this ID in the database.
- Sets up a user profile if one doesn't exist.
- Getting User Info:
- Pulls user data and preferences from the database.
- Updates or creates the user profile with this info.
- Lets you know if it can't find the user.

User Class

- Storing User Data:
  - Keeps track of things like user ID, email, gender, and other personal details.
- Loading from Database:
  - Populates the user profile with information pulled from the database.
- Setting Preferences:
  - Lets you tweak how the app works for that specific user.
- Making Updates:
  - Provides functions like update_email, update_name, etc. to let you easily change user info.
- Saving to Database:
  - Pushes any changes to the user's profile back to the database for secure storage.

DatabaseManager Class

- Database Connection: Handles the interactions to the SQL database.
- Running Queries: Executes the instructions (SQL queries) that let you add, change, or get data.
- Error Handling: Keeps an eye out for database problems and logs them so you can fix them.
- User Management Actions: Includes functions specifically for actions like update_user_profile, insert_user, and so on.

WindowManager Class:

- Purpose:
  - Responsible for switching between different parts of my app.

InitialPage Class

- Purpose:
  - The very first screen a new user sees in your app.
- What it does:
  - Automatically creates a user profile behind the scenes (using the UserManager).
  - Stores that user's ID for keeping track of them.
- Layout (.kv file):

- Simple layout with "Get Started" and "Sign In" buttons.
- Has a small terms of service notice at the bottom.

SignupPage Class

- Handles:
  - The whole process of creating a new user account.
- Checks for:
  - Duplicate usernames.
  - Make sure emails are in the correct format (like "[email address removed]").
  - Enforces some password rules so accounts are more secure.
- Success/Failure:
  - Lets the user know if the signup worked or what went wrong.
- .kv file: Contains fields for username, email, password, a "Sign Up" button, etc.

LoginPage Class

- Handles:
  - Letting existing users sign in.
- Checks For:
  - Valid username and password combination.
- If Success:
  - Grabs the user's preferences.
  - Send them to your app's dashboard (main area).
- .kv file: Similar look to the SignupPage, but set up for logging in instead.

SelectGender Screen

- Purpose:
  - Asks the user to pick their gender.
- What it does:
  - Updates the user's profile in the database with their selection.
  - Sends the user to the next screen in the setup process (asking for their age).

InputAge Screen

- Purpose:
  - Ask the user how old they are.
- Checks:
  - Make sure the age is a number.
  - Make sure the age is in a reasonable range
- If Valid:
  - Updates the user's profile.
  - Moves on to collecting height and weight information.

InputHeightWeight Screen

- Purpose:
  - Collects height and weight from the user.
- Flexibility:

    - Lets users switch between metric (metres/kilograms) and imperial (feet/pounds) units.
- Checks:
    - Make sure the entered height and weight make sense.
- If Valid:
    - Updates the user's profile.

ExperienceLevel Class

- Purpose:
    - Figures out how much fitness experience the user has (are they a gym newbie or a seasoned athlete?).
- How it works:
    - Provides buttons for "Novice," "Beginner," etc.
    - A border highlights the currently selected experience level.
    - Makes sure you select a level before moving on.
    - Database Tie-in: Stores your selected experience level in your profile.

InputBodyFat Class

- Purpose:
    - Lets you enter your body fat percentage (if you know it).
- Checks:
    - Makes sure the entered body fat percentage is a reasonable number.
- Database Tie-in: If valid, saves that number to your user profile.
- If you don't know you bodyfat percentage it redirects you to a page to calculate it.

CalculateBodyFat Class

- Purpose:
    - Estimates your body fat percentage. It offers two ways to do this:
- Measurements:
    - You enter things like waist, hip, and neck circumference.
- BMI & Age:
    - Uses your body mass index (BMI) and age for a less precise estimate.
- Gender Aware:
    - The layout of the screen changes depending on whether you're male or female adding an extra hip input if female.
- Checks:
    - Makes sure any measurements you enter seem reasonable.
- Database Tie-in:
    - Stores the calculated body fat percentage in your profile.

ActivityLevel Class

- Purpose:
    - Finds out how active you usually are (couch potato to super athlete).

  How it works:
    - Buttons for levels like "Sedentary," "Moderately Active," etc.
    - Visually highlights your selection.
    - Won't let you continue until you've picked an activity level.
    - Database Tie-in: Saves your activity level as part of your profile.

InputGoal Class

- Purpose:
  - Sets your overall fitness goal – lose weight, maintain, or gain muscle.
- Smart Recommendations: When you enter this screen:
  - It checks your gender, body fat, experience level, etc. from your profile.
  - Tries to suggest a goal that makes sense for you.
- Selection:
  - Buttons for "Lose," "Maintain," or "Gain".
  - Highlights your choice.
  - Database Tie-in: Stores your fitness goal in your profile.

WeightLossOrGainRate Class

- Purpose:
  - Lets you choose how fast you want to lose or gain weight.

How it Works:

- Displays buttons for different rates (e.g., slow and steady vs. aggressive).
- Highlights your currently selected rate.
- May display the recommended rate from the Goal screen (to help keep you on track).
- Database Tie-in: Saves that rate preference to the user profile.
-

Dashboard Class & UI

- Purpose:
  - This is the homepage of the app, where the user can switch to other pages and functionalities.
- How it works
  - Users can access all the functionalities of the app like selecting a workout (and log a workout), the AI chatbot and the nutrition page.
  - When it enters the screen a function to get nutrition details of the user gets triggered and through a circular progress bar it displays the data.
  - It displays several widgets to access all the features

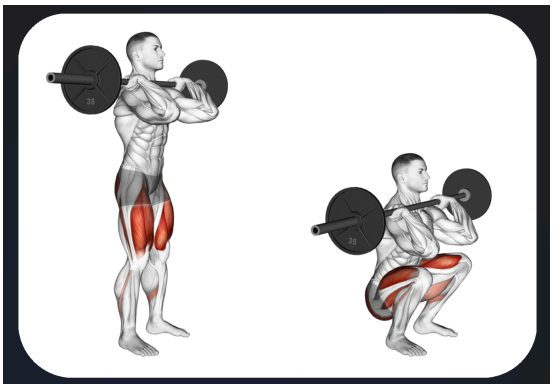
ExerciseList Class

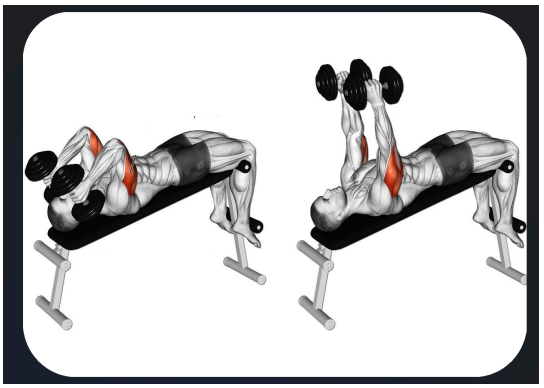
- Purpose:
  - Display and Search: Exercises are displayed in a user-friendly list; a search bar and delayed search logic help users quickly find what they need.
- How it works:
  - Exercise Details: Clicking on an exercise reveals instructions or details, aiding proper execution.
  - Plan Integration: The class handles selection of exercises with checkboxes; this feeds into other areas of your app (workout plan generation or logging).
- Key methods and attributes:
  - self.list_type: can be changed between “list” and “selectable”, if it’s list then it lets the user look though the exercises, search them and reading exercise

guidance, if it's "selectable" it adds a checkbox and you can add the selected exercises to a plan.

- populate_exercises(): – fetches exercise data from the exercises dictionary that i made and renders it on the screen.
- perform_search(): Implement a responsive search system that quickly narrows down results as the user types.
- on_checkbox_clicked(): actioned when the list turns into selectable and handles passing data between classes and changing ui.

I made the images manually and edited them one by one on photoshop to look coherent and have rounded corners.





GenerateWorkoutPlan Class

- User Details Retrieval:
  - Ensure secure and smooth retrieval of data stored during the initial setup.
- Workout Split Creation:
  - Make sure my logic accommodates common splits (full-body, upper/lower, push/pull/legs) while leaving room for future refinements.
- Workout Plan Generation:

    - It generates a workout plan based on a tree structure that I made iterating through each exercise of a given muscle group to find the best one prioritising exercises with muscle groups that the users chose to prioritise and then follows a plan scheme that I set up for each case (gender, experience, frequency etc).
- Dynamic Plan Modification:
    - If users don't like the exercise chosen by the algorithm they can always decide to switch to another one (making sure the next exercise is the next best one for that muscle group. It also allows users to delete an exercise if not wanted.
- Technical Underpinnings:
    - get_user_details(): Fetches user data (training frequency, experience, prioritised muscles, etc.) from the app's database.
    - divide_muscles_to_prioritize(): Takes the user's comma-separated list of prioritised muscle groups and breaks it down into individual muscles for targeted plan generation.
    - create_split(): Defines a list of possible workout splits based on the user's training frequency. Each split includes the workout days (e.g., a 3-day split might be ['chest', 'back', 'legs'])
    - create_plan(): The core logic. Selects an appropriate workout split, then creates a detailed plan (e.g., {'day_1': 'chest, chest, quads, …'}) based on user input and predefined rules.
    - extract_muscles_from_detailed_plan(): Takes the detailed plan and generates a list of muscle groups to be trained each day.
    - reorder_plan_based_on_priority(): Customises the order of exercises within the plan to emphasise the user's priority muscles. Uses a merge sort function (explained later).
    - parse_tree_to_exercises(): Processes a hierarchical 'tree' data structure to create a dictionary of exercises organised by muscle groups, divisions, and ratings.
    - generate_workout_plan(): Iterates through the detailed plan, finds the best-rated exercises (based on the tree) for each muscle group, marks them used, and builds the workout plan (a dictionary like: {'day_1': ['exercise1', 'exercise2', ...]}).
    - find_muscle_and_division(): Helper function to identify the muscle group and division of a given exercise within the exercise data structure.
    - map_muscle_groups_to_divisions(): Refines the detailed plan, replacing general muscle groups (e.g., 'chest') with specific divisions (e.g., 'upper chest'), keeping exercise selection balanced.
    - map_exercise_details(): Fetches detailed information (name, type, icon) for individual exercises to be used when displaying the plan to the user.
    - rewrite_workout_plan(): Merges exercise details into the workout plan structure for proper display.
    - format_workout_plan(): Prepares the final workout plan in a format ready for your app's UI, keeping track of which muscles are targeted each day.
- Supporting Functions

- merge_sort(): Implements the Merge Sort algorithm, utilised to sort exercises by rating and when reorganising the plan.
- switch_exercise_handler(): Provides the functionality for users to swap out exercises. It smartly suggests the next best-rated replacement within the same muscle group.
- save_plan(): Logic for saving the generated plan (interacts with the select_workout screen sending the data for the user to access the plan and log a workout)
- switch_days_left(), switch_days_right():These handle navigation (left/right arrows) between different days of the workout plan within the UI.

FoodSearch Class

- Purpose
  - This class provides the tools for your users to find foods, understand their nutritional content, and track them as part of their daily intake. Think of it as the digital version of reading food labels but with way more power.
- Key Features
  - Food Search and Data: The class can fetch detailed nutritional information about foods, using an API to a food database.
  - Logging and Tracking: Foods selected by the user can be saved into a log. The class keeps track of the total calories, protein, etc., consumed over the day.
  - Visual Feedback: Progress bars visually show the user how their food choices impact their daily nutritional goals
- Technical Underpinnings
  - Attributes: Internal variables store things like user's daily targets (daily_calories), their logged foods (logged_foodlist), and control search timing (search_trigger).
  - Methods: Specific functions carry out the key tasks:
  - get_daily_calories(): Fetches target values
  - calculate_nutrients(): Does the math on food items
  - log_food(), saved_foods(), remove_food(): Manage adding, changing, and removing foods from the logged list.
  - get_food_data(): Fetches data from the food database
  - perform_search(): Does the actual filtering and presentation of search results.

SelectWorkout class

- Purpose
  - It's the user's hub for workout management. It's where they:
    - Browse: View a list of existing workout plans – both predefined ones and any personalised plans associated with their account.
    - Select: Choose a workout plan to execute, either to follow its structure or initiate the workout logging process.
    - Generate: Optionally, trigger the creation of a brand new, personalised workout plan.

- Key Attributes
    - workout_plans: Acts as a container to hold a list of available workout plan data. Each plan has its own structure (name, days, exercises, etc.).
    - workout_name: Keeps track of which workout plan the user has currently selected.
    - opened: This flag signals whether the details panel for a selected plan is expanded or not.
    - plan: Stores the in-depth information of the workout plan that the user has chosen.
    - generated_plan: Temporarily holds a newly generated plan fetched from the GeneratePlan class.
- Key Methods
    - on_kv_post(base_widget): Called after the UI loads from your .kv file. Here, you'd initialise the screen content by fetching workout plans and displaying them.
    - get_plans_from_database(): Handles communication with the database to retrieve the user's saved workout plans.
    - populate_workout(): Takes workout plan data and populates the visible list on the screen.
    - edit_item(text): initiates editing an existing plan going into the EmptyPlan screen with the current workout to modify it.
    - choose_plan_panel(text): Manages the core interaction:
    - Updates the UI when the user selects a different workout plan.
    - Displays workout details in the expandable panel.
    - Trigger the "Generate New Plan" flow if needed.
    - on_generate_plan(generated_plan=None): Called when your plan generation system has produced a new plan. Stores it within the class, probably for later display or saving.
    - on_log_workout(day): Sends the user to the logging screen, along with info about the plan and specific day's workout. This initiates the actual tracking of exercise sets and repetitions.

EmptyPlan class

- Purpose
    - It allows the user to create their own plan selecting from the ExerciseList screen.
- Defining Structure:
    - Setting the number of days in their plan (like a 3-day split, 5-day split, etc.).
    - Filling the Days: Adding specific exercises to each day of their workout plan.
    - Saving the Result: When complete, saving their custom plan into your app's data for later use.
- Key Attributes
    - training_days: How many workout days this user's plan will have. Chosen before from an InputTrainingFrequency screen.
    - workout_plans_by_day: A dictionary to store exercises. Keys are day numbers (1, 2, etc. for each day), and values are lists of exercises for that day.

    - added_exercises: Helps the UI know if a given day has content yet.
    - current_day: Tracks which day (e.g., day 2) the user is actively editing.
    - workout_plans: Acts as an accumulator as exercises are added, building the complete plan structure for saving.
- Key Methods
    - update_plan(plan): Takes exercise data and populates the plan structure appropriately.
    - on_kv_post(base_widget): Runs after your UI (.kv) loads. Initialises screen visuals, sets up navigation, and probably fills in any "blank" data for a new plan.
    - select_day(day): Updates the UI when a day is selected – showing those exercises and highlighting in the navigation.
    - change_screen(day): Handles the actual switching between workout days within the UI.
    - added_exercises_to_plan(added_exercises): The core update: Handles taking selected exercises and adding them to the workout_plans_by_day structure.
    - populate_workout(): Prepares the selected day's exercises for UI display.
    - toggle_save_panel(): Controls the visibility of the "Save Plan" section.
    - save_plan(): Handles sending the constructed plan to your database (or wherever workout plans are stored).
- .kv File Highlights:
    - ScreenManager: For multi-day view switching.
    - MDFloatLayout: My main layout, likely using its background colour for theming.
    - RecycleView: To display day's exercises (note your custom 'SetExercisePlan' view).
    - MDTextButton: Back, save, and add exercise buttons.
    - DraggablePanel: Your popup/collapsible area for naming/saving the plan.
    - BoxLayout/AnchorLayout: For organising day labels and content.

LogWorkout class

- Purpose
    - Where the plan the user picked or created actually gets turned into real workout data – sets, reps, and weight can be logged.
- Key Attributes
    - last_clicked_item, type: For knowing which exercise they just interacted with and whether they were entering reps or weight.
    - workout_rows, current_row: workout_rows holds info for ALL sets. current_row helps know which one is active.
    - selected_row: When editing an existing set's data, this tells you which one.
    - sets, current_item_id, current_page: These track the workout plan position of the logged exercise and manage screen navigation.
    - plan: The structure of the chosen workout.
    - workout_rows_instances: Holds references to your custom objects (see below) that handle managing set data.

    - workout_data: This is where the set-by-set reps, weights, and calculations like 10-rep max estimates get stored.
- Key Methods
    - initialize_workout(): Takes the workout plan structure and sets up the UI's log entry sections accordingly.
    - add_plus_icon(), add_set(),remove_set(): Allow the user to change the plan on the fly – more sets, fewer sets, even new exercises.
    - update_sets(): Ensures UI reflects changes.
    - generate_rows(): Seems to create your custom WorkoutRow objects, which would handle logging single set data.
    - add_exercise(), remove_exercise(): The 'real-time' workout plan editing feature.
    - change_screen(), set_opacity(): Manage exercise-focused views during the workout.
    - choose_item(), chosen_item(), next_row(): The core sequence: Select set, log its data, go to the next.
    - calculate 10RM(): Gives users progress feedback based on their performance and allows them to estimate a 10 rep max if they want to.
    - save_workout(): Write a workout log to the database
    - on_back(), on_single_workout(): Handle navigation to other parts of my app and uses pickle module to dump all instances of the class in a file and fetch it back if the user decides to continue the workout later or wants to exit the app.

ChatBotScreen class:

- Purpose
    - The ChatBotScreen class creates a virtual "fitness coach" conversation for your users. It marries:
- AI Backend:
    - The connection to your chosen AI system (OpenAI), which provides the logic to drive the coach's responses.
- Key Attributes
    - text_input: The space where the user enters their messages.
    - chat_list: The core container in your UI where the messages (user and bot) are displayed.
    - chat: An object representing your AI backend. It houses the API key and handles actually turning user input into AI output.
    - chat_history: This acts as the session 'transcript' so the AI can refer back to what was said.
    - initial_prompt: The chatbot's intro – sets the tone and defines what it can help with.
    - memory: Keeps the conversation context active across the user's interaction.
- Key Methods
    - initial_prompt(): Starts the "conversation" by displaying the bot's introductory message, laying the groundwork for interaction.
    - get_response(user_input), get_response_thread(user_input): When the user sends a message:
        - Takes that user input.
        - Sends it to your AI backend for processing (chat object).

        - Stores the input/response in chat_history.
    - send(): Handles the user clicking send (or hitting enter, likely). Clears the input, shows the message they just typed, and kicks off get_response.
    - rebuild_chat_history(): For when the user returns to the chat. This probably pulls conversation data from your database.
    - update_chat_ui(response_content): Actually places the bot's response in the visual chat.
    - save_to_database(): Writes the session's chat_history to persistent storage, so the user can pick up where they left off later.
- How it Works
    - Screen Opens: initial_prompt greets the user, inviting questions.
    - Conversing: When the user enters text and hits 'send':
    - send() captures that input, displays it locally, and calls get_response.
    - get_response passes that input to your AI backend along with the existing chat_history.
    - The AI replies, the response is stored, and update_chat_ui is used to display it on screen.
    - Saving: When the user leaves, save_to_database makes sure the conversation isn't lost for future sessions.

3.5 Completeness of solution:

1.4.1 : Customizable Workout Plans:

Achieved in the GeneratePlan class:

```
class GeneratePlan(Screen):
    # Class representing the screen for generating a workout plan

    def get_user_details(self):
        # Method to retrieve user details from the database
        user_id = self.manager.get_screen('initialpage').user_id
        user = UserManager.get_user(user_id)
        trainingfrequency = int(user.training_frequency[0])
        experiencelevel = user.experience_level
        prioritizemusclegroups = user.prioritized_muscle_groups
        gender = user.gender
        self.trainingfrequency = trainingfrequency
        return trainingfrequency, experiencelevel, gender, prioritizemusclegroups

    muscle_divisions = {
        # Dictionary to divide muscles into their respective groups
        "chest": ["upper chest", "middle chest", "lower chest"],
        "upper back": ["rhomboids", "mid traps", "upper traps"],
        "lats": ["upper lats", "lumbar lats", "lower lats"],
        "shoulders": ["front delts", "side delts", "rear delts"],
        "quads": ["vastus medialis and lateralis", "rectus femoris"],
        "hamstrings": ["bicep femoris"],
        "glutes": ["gluteus medius", "gluteus maximus"],
        "calves": ["soleus", "gastroc"],
        "abs": ["upper abs", "lower abs"],
        "triceps": ["long head", "side and medial head"],
        "biceps": ["bicep brachii", "brachialis"],
        "forearms": ["brachioradialis", "wrist extensor", "wrist flexors"],
        "adductors": ["adductor magnus"]
    }

    @staticmethod
    def divide_muscles_to_prioritize(prioritizemusclegroups):
        # Static method to divide prioritized muscle groups into individual muscles
        muscles_to_prioritize = []
        for group in prioritizemusclegroups.split(','):
            group = group.strip()
            if group == 'shoulders':
                muscles_to_prioritize.extend(['front delts', 'side delts', 'rear delts'])
            else:
                muscles_to_prioritize.append(group)
```

```
        return muscles_to_prioritize


    @staticmethod
    def create_split():
        # Static method to create a list of workout splits
        # Each index in the list corresponds to a different workout plan
        # Each split is a tuple, where each element is a string representing a workout day
        split=[None]*15
        split[0]= ('full body') # 1 day split
        split[1]= ('upper body', 'lower body') # 2 day split
        split[2]= ('push', 'pull', 'legs') # 3 day split
        split[3]= ('chest', 'back', 'shoulders', 'arms', 'legs') # 5 day split
        split[4]=('biceps, chest, shoulders', 'triceps, back, core', 'quads, hamstrings, glutes, calves') # 3 day split
        split[5]=('chest, shoulders, core', 'quads, calves, adductors', 'back, core', 'arms, forearms', 'posterior chain, calves,
core',) # 5 day split, chest, back or arms focus
        split[6]= ('chest, forearms', 'quads, glutes, calves', 'shoulders, arms', 'hamstrings, glutes, core', 'back,core, forearms',
'quads, adductors, calves') # 6 day split, chest, leg focus
        split[7]=('full body', 'triceps, back', ' hamstrings, glutes', 'biceps, chest', 'back, shoulders', 'quads, glutes', 'calves,
core' ) # 7 day split, leg focus
        split[8]= ('chest, tricpes', 'back, biceps', 'legs, core', 'shoulders', 'chest, back', 'biceps, tricpes', 'legs, core') # 6 day
split, powerlifting focus, or powerbuilding
        split[9]= ('chest, shoulders, triceps', 'back, biceps, hamstrings', 'quads, glutes, calves, core' ) # 3 day split, powerlifting
focus, or powerbuilding
        split[10]=('upper body', 'lower body', 'upper body', 'lower body') # 4 day split
        split[11]=('chest, shoulders', 'back, core', 'arms, calves', 'quads, hamstrings, glutes') # 4 day split, powerlifting focus, or
powerbuilding
        split[12]=('biceps, chest, shoulders', 'triceps, back, core', 'quads, hamstrings, glutes, calves','biceps, chest, shoulders',
'triceps, back, core', 'quads, hamstrings, glutes, calves') # 6 day split, powerlifting focus, or powerbuilding
        split[13]= ('push', 'pull', 'legs', 'push', 'pull', 'legs') # 6 day split, push/pull/legs
        split[14]=('chest, triceps, forearms', 'shoulders, hamstrings, core', 'back, biceps, calves', 'quads, glutes, adductors, core')
# 4 day split, muscle group focus
        return split

    @staticmethod
    def create_plan(trainingfrequency, muscles_to_prioritize, experiencelevel, gender, split):
        # Static method to create a workout plan based on the user's training frequency, prioritized muscles, and experience level
        # The method uses the provided split to determine the workout plan
        # The plan and detailed plan are returned as a tuple

        # If the user trains once a week, only type of workout is possible
        if trainingfrequency== 1:
            plan = split[0]
            detailed_plan={'day_1': 'chest, chest, quads, upper back, hamstrings, lats, glutes, side delts, calves, triceps, biceps,
abs'}

        # If the user trains twice a week
        elif trainingfrequency== 2:
            plan = split[1]
            detailed_plan={'day_1': 'chest, upper back, chest, lats, side delts, tricpes, biceps', 'day_2': 'quads, hamstrings, glutes,
quads, glutes, calves, abs'}

        # If the user trains three times a week
        elif trainingfrequency== 3:
            if 'triceps' or 'biceps' in muscles_to_prioritize or experiencelevel=='advanced' or experiencelevel=='elite' or
experiencelevel=='intermediate':
                plan= split[4]
                detailed_plan={'day_1': 'biceps, biceps, chest, chest, front delts, side delts', 'day_2': 'triceps, tricpes, upper back,
lats, lats, rear delts', 'day_3': 'quads, hamstrings, glutes, quads, glutes, calves, abs'}
            else:
                plan = split[2]
                detailed_plan={'day_1': 'chest, chest, front delts, side delts, triceps, triceps', 'day_2': 'upper back, lats, lats,
biceps, biceps', 'day_3': 'quads, hamstrings, glutes, quads, glutes, calves, abs'}

        # If the user trains four times a week, there are three possible plans, based on the user's experience level and prioritized
muscles
        elif trainingfrequency== 4:
            if 'triceps' or 'biceps' or 'chest' or 'upper back' or 'lats' or 'calves' in muscles_to_prioritize or
experiencelevel=='advanced' or experiencelevel=='elite' or experiencelevel=='intermediate':
                plan= split[11]
                detailed_plan={'day_1': 'chest, chest, chest, front delts, side delts', 'day_2': 'upper back, lats, lats, rear delts,
abs, abs', 'day_3': 'calves, triceps, biceps, triceps, biceps, forearms', 'day_4': 'quads, hamstrings, glutes, quads, glutes,
adductors'}
            elif 'front delts' or 'side delts' or 'quads' in muscles_to_prioritize:
                plan=split[14]
                detailed_plan={'day_1': 'chest, chest, chest, triceps, triceps, foreamrs, foreamrs', 'day_2': 'side delts, front delts,
hamstrings, front delts, hamstrings, side delts'}
            else:
                plan = split[10]
                detailed_plan={'day_1': 'chest, upper back, chest, lats, side delts, tricpes, biceps', 'day_2': 'quads, hamstrings,
glutes, quads, adductors, calves, abs', 'day_3': 'lats, chest, upper back, chest, side delts, biceps, triceps', 'day_4': 'hamstrings,
glutes, quads, hamstrings, glutes, calves, abs'}

        # If the user trains five times a week, two possible plans are available, based on the user's experience level and prioritized
muscles
```

```
        elif trainingfrequency== 5:
            if 'triceps' or 'biceps' or 'chest' or 'upper back' or 'lats' or 'front delts' or 'side delts' or 'rear delts' in
muscles_to_prioritize or gender=='male':
                plan= split[3]
                detailed_plan={'day_1': 'chest, chest, chest, front delts, side delts', 'day_2': 'upper back, lats, lats, abs, abs',
'day_3': 'rear delts, front delts, side delts, rear delts, side delts, front delts', 'day_4': 'triceps, biceps, triceps. biceps,
forearms, forearms', 'day_4': 'quads, hamstrings, glutes, quads, glutes, adductors', 'day_5': 'quads, hamstrings, glutes, quads,
hamstrings, glutes, calves'}
            else:
                plan = split[5]
                detailed_plan={'day_1': 'chest, chest, front delts, side delts, abs, abs', 'day_2': 'quads, adductors, calves, quads,
calves, glutes', 'day_3': 'upper back, lats, lats, forearms, abs, abs', 'day_4': 'triceps, biceps, triceps, biceps, forearms, forearms',
'day_5': 'hamstrings, glutes, hamstrings, glutes, calves, abs'}

        # If the user trains six times a week, there are three possible plans, based on the user's experience level and prioritized
muscles
        elif trainingfrequency== 6:
            if 'triceps' or 'biceps' in muscles_to_prioritize or experiencelevel=='intermediate':
                plan= split[12]
                detailed_plan={'day_1': 'biceps, biceps, chest, chest, front delts, side delts', 'day_2': 'triceps, tricpes, upper back,
lats, lats, rear delts', 'day_3': 'quads, hamstrings, glutes, quads, glutes, calves, abs','day_4': 'biceps, biceps, chest, chest, side
delts, side delts', 'day_5': 'triceps, tricpes, upper back, upper back, lats, rear delts', 'day_6': 'hamstrings, quads, glutes, quads,
glutes, calves, abs'}
            elif experiencelevel=='beginner' or experiencelevel=='novice' or 'lats' or 'upper back' or 'chest' in muscles_to_prioritize:
                plan = split[13]
                detailed_plan={'day_1': 'chest, chest, front delts, side delts, triceps, triceps', 'day_2': 'upper back, lats, lats,
biceps, biceps', 'day_3': 'quads, hamstrings, glutes, quads, glutes, calves, abs', 'day_4': 'chest, chest, side delts, side delts,
triceps, triceps', 'day_5': 'upper back, lats, upper back, biceps, biceps', 'day_6': 'hamstrings, quads, glutes, quads, hamstrings,
calves, abs'}
            else:
                plan = split[6]
                detailed_plan={'day_1': 'chest, chest, chest, forearms, forearms','day_2':'quads, glutes, quads,glutes, calves',
'day_3': 'side delts, triceps, biceps, rear delts, triceps, biceps, front delts', 'day_4': 'hamstrings, glutes, hamstrings, glutes,
calves, abs', 'day_5': 'upper back, lats, lats, abs, abs, forearms', 'day_6': 'quads, adductors, calves, quads, adductors, glutes'}

        # If the user trains seven times a week, one possible plan is available, based on the user's experience level and prioritized
muscles
        elif trainingfrequency== 7:
            plan = split[7]
            detailed_plan={'day_1': 'chest, quads, biceps, glutes, side delts, forearms', 'day_2':'triceps, triceps, upper back, lats,
lats, rear delts', 'day_3':'hamstrings, glutes, hamstrings, glutes, abs', 'day_4':'biceps, biceps, chest, chest, chest', 'day_5':'upper
back, lats, front delts, side delts, lats, rear delts', 'day_6':'quads, glutes, quads, gutes, glutes', 'day_7':'calves, calves, abs,
abs'}
        return plan, detailed_plan

    @staticmethod
    def extract_muscles_from_detailed_plan(detailed_plan, training_frequency):
        # Static method to extract the muscles to be trained from the detailed plan
        # The method iterates over the days in the plan, splits the muscles for each day, and appends them to a list
        # The list of muscles to be trained is returned
        muscles_to_train = []
        for day in range(1, training_frequency + 1):
            muscles_for_day = detailed_plan[f'day_{day}'].split(', ')
            muscles_to_train.append(muscles_for_day)
        return muscles_to_train

    def reorder_plan_based_on_priority(self, detailed_plan, muscles_to_prioritize, muscles_to_train):
        # Method to reorder the workout plan based on the user's prioritized muscles
        reordered_plan = {}
        for day, muscles in detailed_plan.items():
            daily_muscles = muscles.split(', ')
            # Custom comparison function for prioritizing muscles
            compare_func = lambda x, y: (x in muscles_to_prioritize) and (y not in muscles_to_prioritize)
            prioritized_muscles = self.merge_sort(daily_muscles, compare=compare_func)
            reordered_plan[day] = ', '.join(prioritized_muscles)
        return reordered_plan

    from flattened_tree import tree

    def parse_tree_to_exercises(self, tree):
        # Method to parse a tree of exercises into a dictionary
        # The dictionary is structured as {muscle_group: {division: [(exercise_name, rating), ...], ...}, ...}
        # The tree is assumed to have a specific structure, with muscle groups at level 3, divisions at level 4, exercises at level 5,
and ratings at level 6

        exercise_map = {}
        muscle_group, division = None, None

        for node in tree:
            # Skip certain nodes
            if node['name'] in ['progression program', 'Pushups', 'Pull ups', 'Squats', 'Leg Raises', 'Bridges', 'Skills']:
                continue

            # Handle muscle group nodes
            if node['level'] == 3:
```

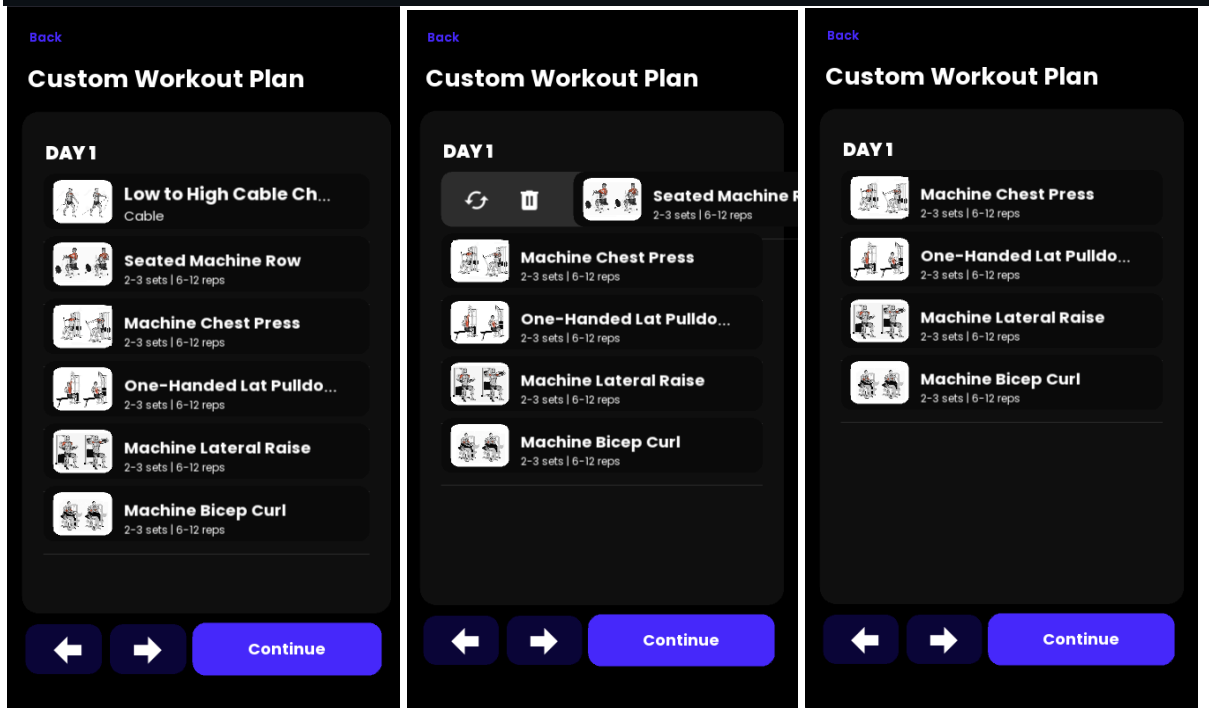
```
            muscle_group = node['name']
            exercise_map[muscle_group] = {}

        # Handle division nodes
        elif node['level'] == 4 and muscle_group:
            division = node['name']
            exercise_map[muscle_group][division] = []

        # Handle exercise nodes
        elif node['level'] == 5 and muscle_group and division:
            exercise_name = node['name']

        # Handle rating nodes
        elif node['level'] == 6 and muscle_group and division and exercise_name:
            try:
                rating = float(node['name'])
                exercise_map[muscle_group][division].append((exercise_name, rating))
            except ValueError:
                # Skip if rating is not a valid number
                continue
    return exercise_map


def generate_workout_plan(self, detailed_plan, exercise_map):
    # Method to generate a workout plan based on a detailed plan and an exercise map
    # The workout plan is a dictionary where each key is a day and each value is a list of exercises for that day
    # The method also keeps track of used exercises to avoid repetition

    # Initialize the workout plan and used exercises
    workout_plan = {}
    used_exercises = {category: [] for category in exercise_map.keys()}

    # Iterate over each day in the detailed plan
    for day, muscle_groups in detailed_plan.items():
        day_plan = []
        # Iterate over each muscle group for the day
        for muscle_group in muscle_groups.split(', '):
            highest_rated_exercise = None
            highest_rating = -1
            # Iterate over each category and subgroup in the exercise map
            for category, subgroups in exercise_map.items():
                for subgroup, exercises in subgroups.items():
                    # If the muscle group is in the subgroup and there are exercises
                    if muscle_group in subgroup and exercises:
                        # Iterate over each exercise and its rating
                        for exercise, rating in exercises:
                            # If the rating is higher than the current highest rating and the exercise has not been used
                            if rating > highest_rating and exercise not in used_exercises[category]:
                                # Update the highest rated exercise and its rating
                                highest_rated_exercise = exercise
                                highest_rating = rating
            # If a highest rated exercise was found
            if highest_rated_exercise:
                # Add the exercise to the day plan and mark it as used
                day_plan.append(highest_rated_exercise)
                for category, subgroups in exercise_map.items():
                    if muscle_group in subgroups:
                        used_exercises[category].append(highest_rated_exercise)
                        break

        # Add the day plan to the workout plan
        workout_plan[f'{day}'] = day_plan

    # Return the workout plan
    return workout_plan


def find_muscle_and_division(self, exercise_name):
    # Method to find the muscle group and division for a given exercise
    # The method iterates over the exercise map and returns the muscle group and division if the exercise is found
    # If the exercise is not found, the method returns None for both the muscle group and division
    for muscle_group, divisions in self.exercise_map.items():
        for division, exercises in divisions.items():
            for ex_name, _ in exercises:
                if ex_name == exercise_name:
                    return muscle_group, division
    return None, None

def map_muscle_groups_to_divisions(self, detailed_plan, muscle_divisions):
    # Method to map muscle groups to divisions in a detailed plan
    # The method iterates over the detailed plan and replaces each muscle group with a specific division from the muscle divisions
    # The method uses a division tracker to ensure that divisions are used evenly for each muscle group
    # The updated plan is returned
    updated_plan = {}
    for day, muscle_groups in detailed_plan.items():
```

```
            updated_muscle_groups = []
            division_tracker = {}

            for muscle_group in muscle_groups.split(', '):
                if muscle_group in muscle_divisions:
                    if muscle_group not in division_tracker:
                        division_tracker[muscle_group] = 0

                    division_index = division_tracker[muscle_group] % len(muscle_divisions[muscle_group])
                    division_tracker[muscle_group] += 1

                    specific_division = muscle_divisions[muscle_group][division_index]
                    updated_muscle_groups.append(specific_division)
                else:
                    updated_muscle_groups.append(muscle_group)

            updated_plan[day] = ', '.join(updated_muscle_groups)
        return updated_plan

    plan=[]

    def map_exercise_details(self, exercise_name):
        # Method to map exercise details for a given exercise name
        # The method iterates over the list of exercises and checks if the name matches the given exercise name
        # If a match is found, a dictionary with the exercise details is created and appended to the plan
        # The dictionary is also returned
        # If no match is found, the method returns None

        for exercise in exercises:
            if exercise['name'] == exercise_name:
                result = {
                    'icon': exercise['icon'],
                    'text': f"[size=20][font=Poppins-Bold.ttf][color=#FFFFFF]{exercise['name']}[/color][/font][/size]",
                    'secondary_text': f"[size=15][font=Poppins-Regular.ttf][color=#FFFFFF]{exercise['type']}[/color][/font][/size]",
                    'image_source': exercise['icon']
                }
                self.plan=list(self.plan)
                self.plan.append(result)
                return result
        return None

    def rewrite_workout_plan(self, workout_plan):
        # Method to rewrite a workout plan with detailed exercise information
        # The method iterates over each day in the workout plan and maps each exercise to its details
        # The detailed exercises are then added to the generated plan for the corresponding day
        # The generated plan is returned
        generated_plan = {}
        for day, plan_exercises in workout_plan.items():
            detailed_exercises = [self.map_exercise_details(exercise) for exercise in plan_exercises]
            generated_plan[day] = detailed_exercises
        return generated_plan


    formatted_plan = {}

    def format_workout_plan(self, workout_plan):
        # Method to format a workout plan with detailed exercise information
        # The method iterates over each day in the workout plan and extracts the list of exercises
        # For each exercise, the method finds the exercise details and adds them to the day's exercises
        # The method also keeps track of the muscles worked on each day
        # The formatted plan is returned

        formatted_plan = {}
        for day in range(1, self.trainingfrequency + 1):
            day_key = f'day_{day}'
            if day_key in workout_plan:
                exercises_list = workout_plan[day_key]
                day_exercises = []
                muscles_worked = set()

                for name in exercises_list:
                    exercise_detail = next((ex for ex in exercises if ex['name'] == name), None)
                    if exercise_detail:
                        day_exercises.append({
                            'name': name,
                            'type': exercise_detail['type'],
                            'icon': exercise_detail['icon']
                        })
                        muscles = exercise_detail['primary_muscle'].split(',') + exercise_detail.get('secondary_muscles', '').split(',')
                        muscles_worked.update(muscle.strip() for muscle in muscles if muscle)

                formatted_plan[day_key] = {
                    'muscles': list(muscles_worked),
                    'exercises': day_exercises
                }
```

```
        self.formatted_plan = formatted_plan
        return formatted_plan

    def generate_plan(self):
        # Method to generate a workout plan based on user details and preferences
        # The method follows several steps to create a personalized workout plan

        # Step 1: Get user details
        trainingfrequency, experiencelevel, gender, prioritizemusclegroups = self.get_user_details()

        # Step 2: Divide prioritized muscle groups
        self.muscles_to_prioritize = self.divide_muscles_to_prioritize(prioritizemusclegroups)

        # Step 3: Create a workout split based on the user's training frequency
        split = self.create_split()

        # Step 4: Create a detailed plan based on the user's training frequency, prioritized muscles, experience level, and workout
split
        self.plan, self.detailed_plan = self.create_plan(trainingfrequency, self.muscles_to_prioritize, experiencelevel,gender, split)

        # Step 5: Extract the muscles to be trained from the detailed plan
        self.muscles_to_train = self.extract_muscles_from_detailed_plan(self.detailed_plan, trainingfrequency)

        # Step 6: Reorder the plan based on the user's prioritized muscles
        self.reordered_plan = self.reorder_plan_based_on_priority(self.detailed_plan, self.muscles_to_prioritize, trainingfrequency)

        # Step 7: Map muscle groups to divisions in the detailed plan
        self.detailed_plan = self.map_muscle_groups_to_divisions(self.detailed_plan, self.muscle_divisions)

        # Step 8: Parse the exercise tree to create an exercise map
        self.exercise_map = self.parse_tree_to_exercises(self.tree)

        # Step 9: Generate a workout plan based on the detailed plan and exercise map
        self.workout_plan = self.generate_workout_plan(self.detailed_plan, self.exercise_map)

        # Step 10: Format the workout plan for display
        self.formatted_plan = self.format_workout_plan(self.workout_plan)

        # Step 11: Rewrite the workout plan with detailed exercise information
        self.generated_plan = self.rewrite_workout_plan(self.workout_plan)

        self.populate_all_exercises()
        print(self.formatted_plan)
    def on_enter(self, *args):
        # Method that is called when the screen is entered
        # It generates a workout plan and populates the exercises for the day

        self.generate_plan()
        self.populate_exercises()

    def remove_item_handler(self, instance, value):
        # Method to handle the removal of an item from the exercise list
        # It removes the item from the list widget and updates the exercises_per_day dictionary

        self.ids.exercise_list.remove_widget(value)

        if self.day in self.exercises_per_day:
            if value in self.exercises_per_day[self.day]:
                self.exercises_per_day[self.day].remove(value)


    day = 1
    exercises_per_day = {}
    def populate_exercises(self, _=None):
        # Method to update the exercise list widget for the current day
        self.ids.day_plan.text = f"DAY {self.day}"
        self.ids.exercise_list.clear_widgets()
        if self.day in self.exercises_per_day:
            for item in self.exercises_per_day[self.day]:
                self.ids.exercise_list.add_widget(item)

    def populate_all_exercises(self):
        # Method to populate exercises for each day in the plan
        for day in range(1, self.trainingfrequency + 1):
            self.populate_exercises_for_day(day, self.formatted_plan[f'day_{day}']['exercises'])

    def populate_exercises_for_day(self, day, exercises):
        # Method to create SwipeToDeleteItem for each exercise and add it to exercises_per_day
        exercise_items = []
        for exercise in exercises:
            item = SwipeToDeleteItem()
            item.text = f"[size=18][font=Poppins-Bold.ttf][color=#FFFFFF]{exercise['name']}[/color][/font][/size]"
            item.secondary_text = f"[size=12][font=Poppins-Regular.ttf][color=#FFFFFF]2-3 sets | 6-12 reps [/color][/font][/size]"
            item.image_source = exercise['icon']
```

```
            item.bind(on_remove_item=self.remove_item_handler)
            item.bind(on_switch_exercise=lambda instance, x=item: self.switch_exercise_handler(x))
            exercise_items.append(item)
        self.exercises_per_day[day] = exercise_items

    already_selected_exercises = set()
    # Merge sort function
    def merge_sort(self, arr, compare=lambda x, y: x < y):
        # Custom merge sort that uses a comparison function
        if len(arr) > 1:
            mid = len(arr) // 2
            L = arr[:mid]
            R = arr[mid:]

            self.merge_sort(L, compare)
            self.merge_sort(R, compare)

            i = j = k = 0

            while i < len(L) and j < len(R):
                if compare(L[i], R[j]):
                    arr[k] = L[i]
                    i += 1
                else:
                    arr[k] = R[j]
                    j += 1
                k += 1

            while i < len(L):
                arr[k] = L[i]
                i += 1
                k += 1

            while j < len(R):
                arr[k] = R[j]
                j += 1
                k += 1

        return arr


    # Function to handle switching exercises
    def switch_exercise_handler(self, item):
        # Extract the exercise name from the markup text
        current_exercise = re.search(r'\[color=#FFFFFF](.*?)\[/color]', item.text).group(1)

        # Find the muscle group and division of the current exercise
        muscle_group, division = self.find_muscle_and_division(current_exercise)

        if division:
            # Get the exercises for the current muscle group and division
            division_exercises = self.exercise_map[muscle_group][division]
            # Sort the exercises by rating in descending order
            sorted_exercises = self.merge_sort(division_exercises, compare=lambda x, y: x[1] > y[1])

            # If all exercises have been selected once, clear the set
            if len(self.already_selected_exercises) == len(sorted_exercises):
                self.already_selected_exercises.clear()

            # Find the next highest-rated exercise that is not the current exercise and has not been selected before
            for next_exercise in sorted_exercises:
                if next_exercise[0] != current_exercise and next_exercise[0] not in self.already_selected_exercises:
                    new_exercise = next_exercise[0]
                    # Add the new exercise to the set of selected exercises
                    self.already_selected_exercises.add(new_exercise)
                    # Get the details of the new exercise
                    new_exercise_details = self.map_exercise_details(new_exercise)

                    # Update the UI with the new exercise details
                    item.text = new_exercise_details['text']
                    item.secondary_text = new_exercise_details['secondary_text']
                    item.image_source = new_exercise_details['icon']
                    return


    def save_plan(self):
        MDApp.get_running_app().root.get_screen('select_workout').on_generate_plan(generated_plan=self.formatted_plan)

    def switch_days_left(self):
        # Method to switch to the previous day in the workout plan
        # If the current day is the first day, the method disables the left arrow button
        # Otherwise, the method enables the left arrow button and decreases the day by 1
        # The method then populates the exercises for the new day

        self.ids.arrow_right.opacity = 1
```

```
        self.ids.arrow_right.disabled = False

        if self.day == 1:
            self.ids.arrow_left.opacity = 0.3
            self.ids.arrow_left.disabled = True
        else:
            self.ids.arrow_left.opacity = 1
            self.ids.arrow_left.disabled = False
            self.day -= 1
        self.populate_exercises()

    def switch_days_right(self):
        # Method to switch to the next day in the workout plan
        # If the current day is the last day, the method disables the right arrow button
        # Otherwise, the method enables the right arrow button and increases the day by 1
        # The method then populates the exercises for the new day

        self.ids.arrow_left.opacity = 1
        self.ids.arrow_left.disabled = False

        if self.day == self.trainingfrequency:
            self.ids.arrow_right.opacity = 0.3
            self.ids.arrow_right.disabled = True
        else:
            self.ids.arrow_right.opacity = 1
            self.ids.arrow_right.disabled = False
            self.day += 1
        self.populate_exercises()
```



1.4.2 Comprehensive food tracking integration, 1.4.3 Nutritional Breakdown of Meals
Achieved in FoodSearch class:

```
class FoodSearch(Screen):
    # Class representing the screen for displaying a list of food items
    search_delay = 0.05

    def __init__(self, **kwargs):
        # Initialize the screen and create a trigger for performing search with a delay
        super().__init__(**kwargs)
        self.current_unpacked_data = {}
        self.logged_foodlist=[]
        self.search_trigger = Clock.create_trigger(self.perform_search, self.search_delay)
        self.serving = 1
        self.editing_index = None
        self.total_calories = 0
        self.total_protein = 0
        self.total_fats = 0
```

```
        self.total_carbs = 0
        self.saved_unit_sequence = None
        self.unit_to_sequence = {
            'serving': (True, False, False, False, False, False),
            'oz': (False, True, False, False, False, False),
            'g': (False, False, True, False, False, False),
            'hg': (False, False, False, True, False, False),
            'lb': (False, False, False, False, True, False),
            'cup': (False, False, False, False, False, True),
        }

        #Clock.schedule_once(lambda dt: self.change_screen(screen_name='food_details_screen'), 1)
    def on_kv_post(self, base_widget):
        super().on_kv_post(base_widget)
        self.rebuild_food_list()
        self.get_daily_calories()

    def get_daily_calories(self):
        result = db_manager.get_daily_values('262efaa4-1a2d-484e-8de8-32966c8a6a82',
datetime.now().strftime("%Y-%m-%d"))
        print(result)
        try:
            daily_target = result[0][0]
            if daily_target is None:
                daily_target = '2000, 150, 200, 50'  # Default value
            daily_target = daily_target.split(', ')
            print(daily_target)
            self.daily_calories = int(daily_target[0])
            self.daily_protein = int(daily_target[1])
            self.daily_carbs = int(daily_target[2])
            self.daily_fats = int(daily_target[3])
            self.total_calories = result[0][1]
            self.total_protein = result[0][2]
            self.total_fats = result[0][3]
            self.total_carbs = result[0][4]
        except (IndexError, ValueError, AttributeError):
            # Handle the case when the result is empty or invalid
            self.daily_calories = 2800
            self.daily_protein = 150
            self.daily_carbs = 180
            self.daily_fats = 100
            self.total_calories = 0
            self.total_protein = 0
            self.total_fats = 0
            self.total_carbs = 0

        self.ids.calories_consumed.text = f"{self.total_calories}/{self.daily_calories}"
        #TODO add gui changes to calories when you enter the app

        return self.daily_calories, self.daily_protein, self.daily_carbs, self.daily_fats, self.total_calories,
self.total_protein, self.total_carbs, self.total_fats

    def calculate_nutrients(self, food_data,):
        portion_factor = round(float(food_data['portion_size']) * food_data.get('selected_weight', 100) / 100, 2)
        return {
            'kcal': round((food_data['original_macros']['calories'])*portion_factor),
            'protein': round((food_data['original_macros']['protein'])*portion_factor),
            'carbs': round((food_data['original_macros']['carbs'])*portion_factor),
            'fats': round((food_data['original_macros']['fats'])*portion_factor)
        }

    def create_food_text(self, food, nutrients):
        food_text = f"[size=18][font=Poppins-SemiBold.ttf][color=#FFFFFF]{food['label']}[/color][/font][/size]"
        weight = round((food['selected_weight']))
        secondary_text = f"[size=13][font=Poppins-Regular.ttf][color=#FFFFFF]{nutrients['kcal']}kcal
{nutrients['protein']}P {nutrients['carbs']}C {nutrients['fats']}F • {weight}g[/color][/font][/size]"
        return {'text': food_text, 'secondary_text': secondary_text, 'original_data': food}

    def calculate_totals(self, logged_foodlist):
        for food in logged_foodlist:
            selected_weight = food.get('selected_weight', 1)
            portion_factor = round((food['portion_size']) * float(selected_weight) / 100, 2)
            self.total_calories += food['original_macros']['calories'] * portion_factor
```

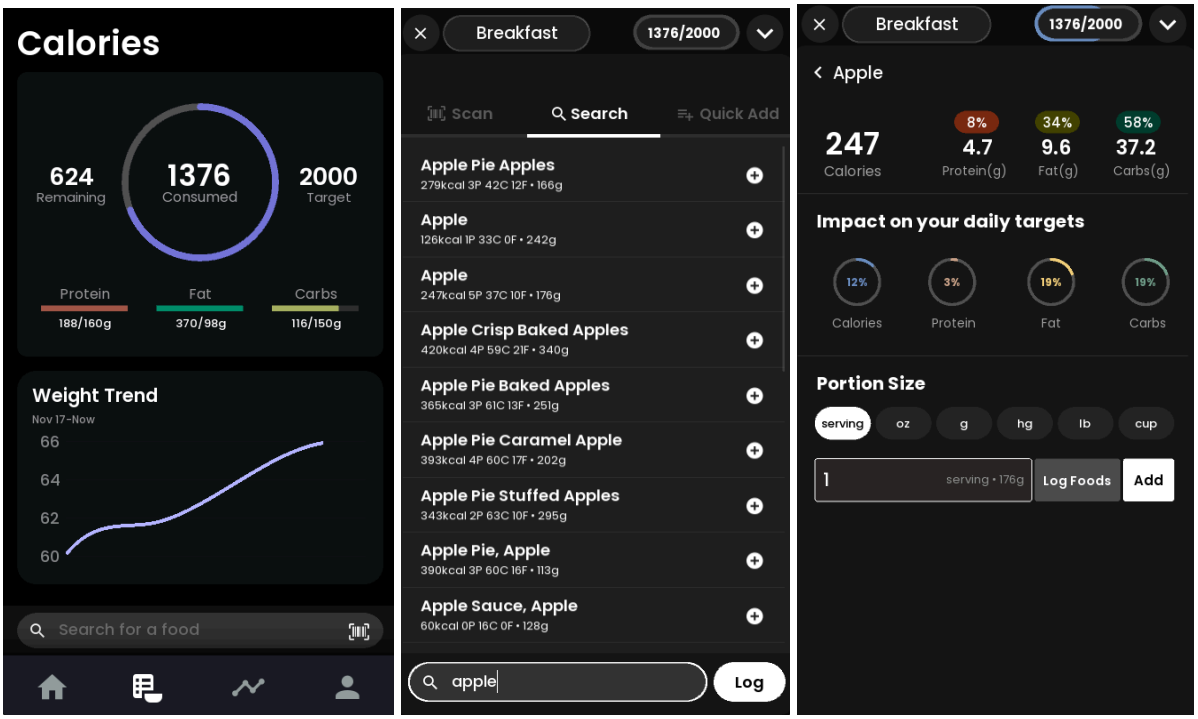
```
            self.total_protein += food['original_macros']['protein'] * portion_factor
            self.total_fats += food['original_macros']['fats'] * portion_factor
            self.total_carbs += food['original_macros']['carbs'] * portion_factor
        return self.total_calories

    def log_food(self):
        user_id = '262efaa4-1a2d-484e-8de8-32966c8a6a82'  # Replace with the actual user ID
        date = datetime.now().strftime("%Y-%m-%d")  # Current date

        for food in self.logged_foodlist:
            # Prepare data for database insertion
            label = food['label']
            kcal = food['calories']
            protein = food['protein']
            carbs = food['carbs']
            fats = food['fats']
            fiber = food['fiber']
            portion_size = food['portion_size']
            selected_weight = food.get('selected_weight', 1)
            unit = food['unit']


            db_manager.insert_logged_foods(user_id, label, kcal, protein, carbs, fats, fiber, portion_size,
selected_weight, unit, date)
        self.logged_foodlist.clear()

    def rebuild_food_list(self):
        user_id = '262efaa4-1a2d-484e-8de8-32966c8a6a82'  # Replace with the actual user ID
        date = datetime.now().strftime("%Y-%m-%d")  # Current date

        # Get the food data from the database
        food_items = db_manager.get_food_items(user_id, date)
        for index, item in enumerate(food_items):
            # Unpack the tuple
            # Assuming columns are in the order: id, label, kcal, protein, carbs, fats, fiber, portion_size,
selected_weight, unit, timestamp
            user_id, label, kcal, protein, carbs, fats, fiber, portion_size, selected_weight, unit, timestamp = item

            # Convert the values to the correct types and calculate the macros per 100g so that it can display them
and use them for calculations properly
            portion_size = float(portion_size)
            selected_weight = float(selected_weight)

            kcal = float(kcal)
            protein = float(protein)
            carbs = float(carbs)
            fats = float(fats)
            fiber = float(fiber)

            unit_sequence = self.unit_to_sequence[unit]

            food_data = {
                'index': index,
                'label': label,
                'calories': kcal,
                'protein': protein,
                'carbs': carbs,
                'fats': fats,
                'fiber': fiber,
                'portion_size': portion_size,
                'selected_weight': selected_weight,
                'unit': unit,
                'unit_sequence': unit_sequence,
                'original_macros': {
                    'calories': round(kcal/ portion_size / selected_weight * 100, 2),
                    'protein': round(protein/ portion_size / selected_weight * 100, 2),
                    'carbs': round(carbs/ portion_size / selected_weight * 100, 2),
                    'fats': round(fats/ portion_size / selected_weight * 100, 2)
                }
            }

            self.logged_foodlist.append(food_data)
            food_data['index'] = index
```

```
        # Recalculate the totals
        self.ids.added_food_rv.data = [self.create_food_text(food, self.calculate_nutrients(food)) for food in
self.logged_foodlist]
        self.total_calories = self.calculate_totals(self.logged_foodlist)

    def saved_foods(self):
        food_data = copy.deepcopy(self.current_unpacked_data)

        # Update or add the food item
        if self.editing_index is None:
            food_data['index'] = len(self.logged_foodlist)
            self.logged_foodlist.append(food_data)
        else:
            self.logged_foodlist[self.editing_index] = food_data
            self.editing_index = None

        # Update the RecycleView data and total calories
        self.ids.added_food_rv.data = [self.create_food_text(food, self.calculate_nutrients(food)) for food in
self.logged_foodlist]
        self.total_calories = self.calculate_totals(self.logged_foodlist)

        # Update UI elements
        self.update_ui_elements()

    def remove_food(self, original_data):
        index = next((i for i, food in enumerate(self.logged_foodlist) if food['index'] == original_data['index']),
None)
        if index is not None:
            self.logged_foodlist.pop(index)
        for i, food in enumerate(self.logged_foodlist):
            food['index'] = i

        self.ids.added_food_rv.data = [self.create_food_text(food, self.calculate_nutrients(food)) for food in
self.logged_foodlist]
        self.total_calories = self.calculate_totals(self.logged_foodlist)

        # Update UI elements
        self.update_ui_elements()

    def update_ui_elements(self):
        self.ids.calories_consumed.text = f"{round(self.total_calories)}/{self.daily_calories}"
        self.ids.daily_calories_consumed.progress = round((self.total_calories / self.daily_calories) * 100) if
self.daily_calories else 0

    def quick_actions(self, barcode, search, quickadd):
        states = {
            'barcode': (0, ('scan', 1), ('search', .3), ('quick_add', .3)),
            'search': (150, ('scan', .3), ('search', 1), ('quick_add', .3)),
            'quickadd': (300, ('scan', .3), ('search', .3), ('quick_add', 1))
        }

        state = 'barcode' if barcode else 'search' if search else 'quickadd'
        bar_position, *opacities = states[state]

        self.ids.horizontal_bar.bar_position = bar_position

        for name, opacity in opacities:
            getattr(self.ids, f'{name}_icon').opacity = opacity
            getattr(self.ids, f'{name}_text').opacity = opacity

    def get_food_data(self, query):
        app_id = 'a4541f6e'
        app_key = '06ad20316bcef2b9088b22533fbf840a'
        url =
f'https://api.edamam.com/api/food-database/v2/parser?app_id={app_id}&app_key={app_key}&ingr={query}&nutrition-type=lo
gging'

        response = requests.get(url)
        if response.status_code == 200:
            return response.json()
        else:
            return f"Error: Unable to fetch data, status code {response.status_code}"
```

```
    def unpack_food_data(self, original_data):
        nutrients = original_data['food']['nutrients']
        measures = original_data.get('measures', [])
        serving_size = next((measure for measure in measures if measure.get('label') == 'Serving'), 100)
        unpacked_data = {
            'label': original_data['food']['label'],
            'calories': round(nutrients.get('ENERC_KCAL', 0),2),
            'carbs': round(nutrients.get('CHOCDF', 0),2),
            'protein': round(nutrients.get('PROCNT', 0),2),
            'fats': round(nutrients.get('FAT', 0),2),
            'fiber': round(nutrients.get('FIBTG', 0),2),  # Include fiber
            'portion_size': 1,  # Initialize portion_size with a default value
            'unit': 'serving',  # Save the unit name instead of the boolean sequence
            'unit_sequence': (True, False, False, False, False, False),  # Initialize unit_sequence with a default
value (serving
            'selected_weight': int(serving_size['weight'])
        }
        original_macros = {
            'calories': unpacked_data['calories'],
            'protein': unpacked_data['protein'],
            'carbs': unpacked_data['carbs'],
            'fats': unpacked_data['fats']
        }
        unpacked_data['original_macros']=original_macros
        return unpacked_data

    def unit_to_sequence_func(self, unit):
        return self.unit_to_sequence.get(unit, (True, False, False, False, False, False))

    def perform_search(self, *args):
        try:
            query = self.ids.search_field.text
            data = self.get_food_data(query)
            filtered_foods = []
            if isinstance(data.get('hints'), list):
                for hint in data['hints']:
                    if isinstance(hint.get('food'), dict) and query.lower() in hint['food'].get('label', '').lower():
                        unpacked_data = self.unpack_food_data(hint)
                        food_text =
f"[size=18][font=Poppins-SemiBold.ttf][color=#FFFFFF]{unpacked_data['label']}[/color][/font][/size]"

                        weight = unpacked_data['selected_weight']

                        # Calculate the portion factor
                        portion_factor = weight / 100 if weight != 'N/A' else 1

                        # Calculate the kcal and other macros for the serving size
                        kcal = round(unpacked_data['calories'] * portion_factor)
                        protein = round(unpacked_data['protein'] * portion_factor)
                        carbs = round(unpacked_data['carbs'] * portion_factor)
                        fats = round(unpacked_data['fats'] * portion_factor)

                        secondary_text = f"[size=13][font=Poppins-Regular.ttf][color=#FFFFFF]{kcal}kcal {protein}P
{carbs}C {fats}F • {weight}g[/color][/font][/size]"
                        filtered_foods.append({'text': food_text, 'secondary_text': secondary_text, 'original_data':
unpacked_data})

            self.ids.rv.data = filtered_foods
        except Exception as e:
            print(f"An error occurred: {e}")

    def search_query(self, query):
        # Reset the countdown for the search trigger and adjust the opacity of the search label based on whether the
search field is empty
        if self.ids.search_field.text:
            self.search_trigger.cancel()
            self.ids.search_label.opacity=0
        else :
            self.ids.search_label.opacity=.3
        self.search_trigger.cancel()
        self.search_trigger()
```

```
    def change_screen(self, *args, screen_name):
        # Assuming this method is part of a class that has access to the screen manager
        self.ids.food_screen_manager.current = screen_name

    def food_details(self, unpacked_data, index):
        print(unpacked_data)
        self.editing_index = index
        Clock.schedule_once(lambda dt: self.change_screen(screen_name='food_details_screen'), 0.5)
        # unpack the data
        self.kcal = unpacked_data['calories']
        self.carbs = unpacked_data['carbs']
        self.protein = unpacked_data['protein']
        self.fats = unpacked_data['fats']

        fiber = unpacked_data['fiber']
        self.serving = unpacked_data['selected_weight']

        # update the portion size, unit, and macros based on the portion size
        self.current_unpacked_data = copy.deepcopy(unpacked_data)  # Update current_unpacked_data

        self.portion_size = unpacked_data['portion_size']
        self.ids.portion_size.text = str(self.portion_size)

        self.kcal *= self.portion_size
        self.protein *= self.portion_size
        self.carbs *= self.portion_size
        self.fats *= self.portion_size

        protein_ratio = self.protein * 4.1 / self.kcal
        fat_ratio = self.fats * 8.8 / self.kcal
        carbs_ratio = ((self.carbs - fiber) * 4.1 + fiber * 1.9) / self.kcal
        total_ratio = protein_ratio + fat_ratio + carbs_ratio

        self.ids.protein_percentage.text =
f"[size=15][font=Poppins-Medium.ttf]{round(protein_ratio/total_ratio*100)}%[/font][/size]"
        self.ids.fats_percentage.text =
f"[size=15][font=Poppins-Medium.ttf]{round(fat_ratio/total_ratio*100)}%[/font][/size]"
        self.ids.carbs_percentage.text =
f"[size=15][font=Poppins-Medium.ttf]{round(carbs_ratio/total_ratio*100)}%[/font][/size]"
        self.ids.food_details_label.text = unpacked_data['label']

        self.selected_unit(*self.unit_to_sequence_func(unpacked_data['unit']),)
        self.update_unpacked_data()

    def update_unpacked_data(self):
        # Get the current portion size and unit sequence
        try:
            portion_size = float(self.ids.portion_size.text)
        except ValueError:
            portion_size = 1  # default to 1 if the input is not a valid float

        # Update the current_unpacked_data
        self.current_unpacked_data['portion_size'] = portion_size

    def update_portion_values(self):
        print(self.current_unpacked_data)
        try:
            amount = round(float(self.ids.portion_size.text)* self.current_unpacked_data['selected_weight']/ 100, 2)
        except ValueError:
            amount = 1
        values = {key: round(self.current_unpacked_data['original_macros'][key] * amount, 2) for key in
self.current_unpacked_data['original_macros']}
        self.current_unpacked_data.update(values)

        original_font_sizes = {
            'calories': 34,
            'protein': 22,
            'fats': 22,
            'carbs': 22
        }

        daily_values = {
            'calories': self.daily_calories,
```

```
            'protein': self.daily_protein,
            'fats': self.daily_fats,
            'carbs': self.daily_carbs
        }

        for key, value in values.items():
            if key == 'calories':
                text = str(round(value))
            else:
                text = str(round(value, 1))
            self.ids[f'{key}_amount'].text = text

            # Calculate the percentage of the nutrient over the daily amount
            percentage = round((value / daily_values[key]) * 100)
            self.ids[f'{key}_bar'].custom_text = f"{percentage}%"
            self.ids[f'{key}_bar'].value = percentage

            self.ids.calories_consumed.text = f"{round(self.total_calories)}/{self.daily_calories}"
            if self.daily_calories != 0:
                self.ids.daily_calories_consumed.progress = round((self.total_calories ) / self.daily_calories * 100)
            else:
                self.ids.daily_calories_consumed.progress = 0  # or whatever value makes sense in this case

            # Adjust the font size based on the length of the text
            if len(text) > 4:
                if key == 'calories':
                    self.ids[f'{key}_amount'].font_size = original_font_sizes[key] - (len(text) - 4) * 5.5
                else:
                    self.ids[f'{key}_amount'].font_size = original_font_sizes[key] - (len(text) - 4) * 2
            else:
                self.ids[f'{key}_amount'].font_size = original_font_sizes[key]

    def selected_unit(self, serving, oz, g, hg, lb, cup):
        white = [1, 1, 1, 1]
        black = [0, 0, 0, 1]
        original_color = [35/255, 35/255, 35/255, 1]

        units = {
            'serving': {'condition': serving, 'weight': self.serving},
            'oz': {'condition': oz, 'weight': 28.3},
            'g': {'condition': g, 'weight': 1},
            'hg': {'condition': hg, 'weight': 100},
            'lb': {'condition': lb, 'weight': 453.6},
            'cup': {'condition': cup, 'weight': 200},
        }

        # Determine the selected unit
        self.current_unpacked_data['unit'] = next(unit for unit, details in units.items() if details['condition'])

        # Reset color of all buttons
        for unit_name in units:
            button = self.ids[f'portion_unit_{unit_name}']
            button.btn_color = button.btn_color_down = original_color
            button.color = white  # Reset text color to white

        # Set color and text of the selected button
        for unit_name, details in units.items():
            if details['condition']:
                button = self.ids[f'portion_unit_{unit_name}']
                button.btn_color = button.btn_color_down = white
                button.color = black
                self.ids.portion_hint_text.text = f"{unit_name} • {round(details['weight'])}g"
                self.portion_size = round(float(self.serving)/100* details['weight'], 2)
                self.current_unpacked_data['selected_weight'] = details['weight']
                break
        self.update_unpacked_data()
        self.update_portion_values()
```



1.4.4 User-Created Workout Plan Design:

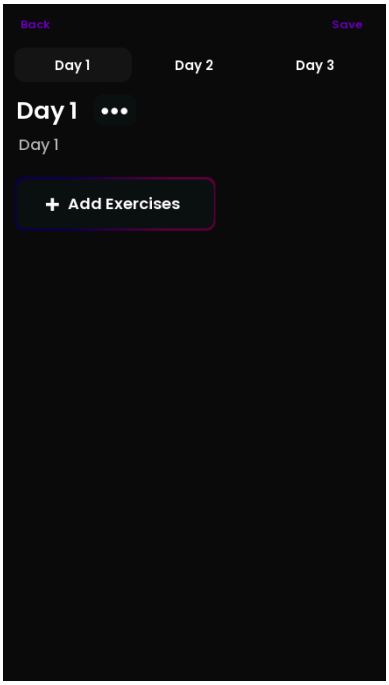
Achieved in the EmptyPlan class:

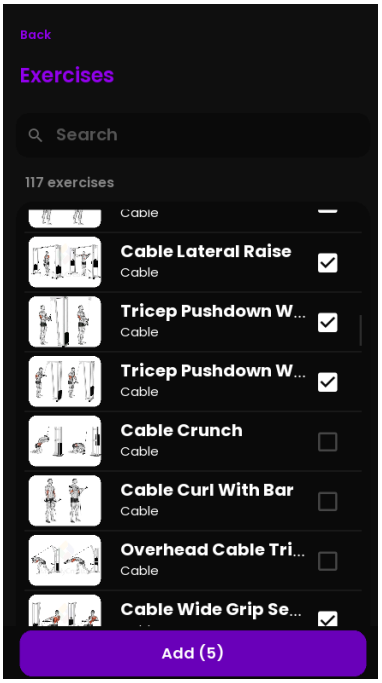
```
class EmptyPlan(Screen):
    """
    A screen in a fitness app that represents an empty workout plan.
    Allows the user to navigate between different days of the plan, add exercises to the plan,
    and display the workout details for each day.
    """
    def __init__(self, **kw):
        super().__init__(**kw)
        self.training_days = 3
        self.workout_plans_by_day = {day: [] for day in range(1, self.training_days + 1)}
        self.added_exercises = [False for _ in range(self.training_days)]
        self.current_day = 1
        self.workout_plans = []
        self.add_exercise_button_new= "add_exercises_post_1"
        self.opened= False

    def update_plan(self, plan):
        self.workout_plans_by_day = plan
        self.training_days = len(plan)
        self.added_exercises = [False for _ in range(self.training_days)]
        self.added_exercises = [True if plan[day] else False for day in plan]
        self.ids.days_in_plan.clear_widgets()
        self.ids.invisible_rect.clear_widgets()
        self.add_days_to_boxlayout()
        self.select_day(1)
        self.populate_workout()
        self.added_exercises_to_plan(plan[1])

    def on_kv_post(self, base_widget):
        """
        Called after the widget has been added to the widget tree.
        Adds the days of the workout plan to the layout, selects the first day, and populates the workout details.
        """
        self.add_days_to_boxlayout()
        self.select_day(1)
        self.populate_workout()
        return super().on_kv_post(base_widget)

    def add_days_to_boxlayout(self):
        """
        Adds the days of the workout plan to the layout as labels and invisible rectangles for navigation.
```

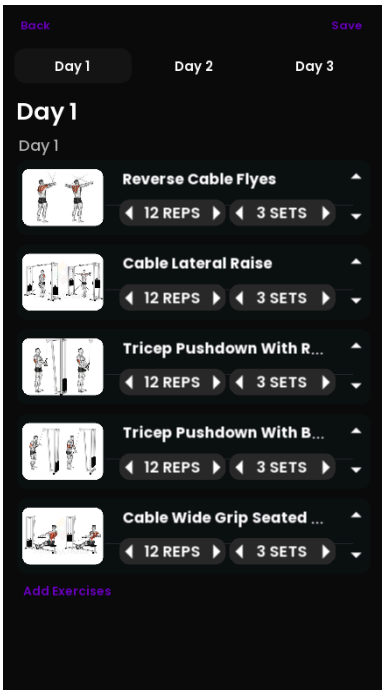
```
        """
        for day in range(1, self.training_days + 1):
            label = Label(
                text=f"Day {day}",
                color=(1, 1, 1, 1),
                font_name="Poppins-SemiBold",
                font_size=15,
                halign='center'
            )
            invisible_rect = RoundedRectangle2(
                size_hint=(1, .5),
                pos_hint={'center_x': 0.5, 'center_y': 0.5},
                opacity=0,
                on_release=partial(self.change_screen, day)
            )
            self.ids.days_in_plan.add_widget(label)
            self.ids.invisible_rect.add_widget(invisible_rect)

    def select_day(self, day, *args):
        """
        Updates the selected day and highlights it on the layout.
        """
        self.current_day = day
        self.ids.selected_day_rec.size_hint = (1 / self.training_days - .01, .5)
        base_center_x = (2 * day - 1) / (2 * self.training_days)
        adjusted_center_x = base_center_x
        self.ids.selected_day_rec.pos_hint = {'center_x': adjusted_center_x, 'center_y': 0.5}
        self.change_items()


    def change_screen(self, day, *args):
        exercise_plan_id=f"exercises_plan_{self.current_page}"
        self.ids[exercise_plan_id].data= [] #clear the data
        self.workout_plans= [] #clear the data
        """
        Changes the screen to the next or previous day of the workout plan based on the selected day.
        """
        if self.ids.empty_plan_scr.current == "empty_plan_day_1":
            self.ids.empty_plan_scr.current = "empty_plan_day_2"
        else:
            self.ids.empty_plan_scr.current = "empty_plan_day_1"
        if day > self.current_day:
            self.ids.empty_plan_scr.transition.direction = 'left'
        else:
            self.ids.empty_plan_scr.transition.direction = 'right'
        self.select_day(day, *args)

    def change_items(self):
        """
        Updates the title and subtitle of the current day's workout details.
        """
        self.current_page = self.ids.empty_plan_scr.current[-1]
        self.add_exercises_id = f"add_exercise_btn_{self.current_page}"
        self.exercise_list= f"exercises_list_{self.current_page}"
        title_id = f"title_{self.current_page}"
        subtitle_id = f"subtitle_{self.current_page}"
        self.ids[title_id].text = f"Day {self.current_day}"
        self.ids[subtitle_id].text = f"Day {self.current_day}"

        if self.added_exercises[self.current_day - 1]:
            self.added_exercises_to_plan(self.workout_plans_by_day[self.current_day])
        else:
            self.ids[self.add_exercise_button_new].pos_hint={"center_x": .18, "center_y": 2}
            self.ids[self.add_exercises_id].pos_hint= {'center_x': .5,'center_y': .5}
            self.ids[self.exercise_list].pos_hint= {'center_x': 2, 'center_y': 2}

    def added_exercises_to_plan(self, added_exercises):
        self.add_exercise_button_new= f"add_exercises_post_{self.current_page}"

        if len(added_exercises) > 5:
            y_coord= 0.105
        elif len(added_exercises) == 5:
            y_coord= 0.145
```

```
        elif len(added_exercises) == 4:
            y_coord= 0.26
        elif len(added_exercises) == 3:
            y_coord= 0.375
        elif len(added_exercises) == 2:
            y_coord= 0.51
        else:
            y_coord= 0.63

        if added_exercises:
            self.added_exercises[self.current_day - 1] = True
            self.ids[self.add_exercise_button_new].pos_hint={"center_x": .18, "center_y": y_coord}
            self.ids[self.add_exercises_id].pos_hint= {'center_x': 2, 'center_y': 2}
            self.ids[self.exercise_list].pos_hint= {'center_x': .5, 'center_y': 0.45}
            self.workout_plans_by_day[self.current_day] = copy.deepcopy(added_exercises)
            # Check if the exercises are already in the desired dictionary format
            if added_exercises and (not isinstance(added_exercises[0], dict)):
                # Convert list of exercise names to list of dictionaries only if they are not already dictionaries
                self.workout_plans_by_day[self.current_day] = [
                    {'name': exercise, 'sets': 3, 'reps': 12} for exercise in added_exercises
                ]
            else:
                # If they are already dictionaries, use them as is
                self.workout_plans_by_day[self.current_day] = copy.deepcopy(added_exercises)

            self.populate_workout()

    def populate_workout(self):
        """
        Populates the workout plan with exercise details.
        """
        if self.added_exercises[self.current_day - 1]:
            self.workout_plans.clear()  # Clear the list to repopulate it

            for exercise_dict in self.workout_plans_by_day[self.current_day]:
                exercise_name = exercise_dict['name']
                sets = exercise_dict['sets']
                reps = exercise_dict['reps']

                # Find the exercise in the exercises dictionary
                exercise = next((e for e in exercises if e['name'] == exercise_name), None)
                if exercise is not None:
                    item = {
                        'image_source': exercise['icon'],
                        'text':
f"[size=17][font=Poppins-Bold.ttf][color=#FFFFFF]{exercise_name}[/color][/font][/size]",
                        'sets_text': f"{sets} SETS",
                        'reps_text': f"{reps} REPS",
                    }
                    self.workout_plans.append(item)

            exercise_plan_id = f"exercises_plan_{self.current_page}"
            self.ids[exercise_plan_id].data = self.workout_plans

    def find_exercise_index(self, ex_name):
        for i, exercise in enumerate(self.workout_plans_by_day[self.current_day]):
            if exercise['name'] == ex_name:
                return i

    def update_sets_and_reps(self, ex_name, sets_text, reps_text, move_value):
        ex_name = re.search(r'\[color=#FFFFFF](.*?)\[/color]', ex_name).group(1)
        sets = int(sets_text.split()[0])  # Extract sets number
        reps = int(reps_text.split()[0])  # Extract reps number

        i = self.find_exercise_index(ex_name)
        if move_value is None:
            # Update the exercise in the workout plan for the current day
            self.workout_plans_by_day[self.current_day][i]['sets'] = sets
            self.workout_plans_by_day[self.current_day][i]['reps'] = reps
        else:
            # Change order of the exercises
            exercise = self.workout_plans_by_day[self.current_day].pop(i)
            if move_value == -1:  # Move the current exercise one position up
```

```
                self.workout_plans_by_day[self.current_day].insert(i - 1, exercise)
            else:  # Move the current exercise one position down
                self.workout_plans_by_day[self.current_day].insert(i + 1, exercise)

        # Debug print to check the structure after the update

    def toggle_save_panel(self):
        self.opened= not self.opened
        if self.opened:
            self.ids.behind_panel_rec.pos_hint= {'center_x': 0.5,'center_y': 0.5}
        else:
            self.ids.behind_panel_rec.pos_hint= {'center_x': 2,'center_y': 2}
            return
        day_number = [i for i in range(1,len(self.workout_plans_by_day)+1)]
        items_box = self.ids.items_box  # Access the BoxLayout within AnchorLayout
        items_box.clear_widgets()  # Clear the BoxLayout
        for day in day_number:
            save_plan_item = SavePlanItem(day=day,selected_exercises= self.workout_plans_by_day)
            items_box.height= dp(35)*len(day_number)
            items_box.add_widget(save_plan_item)  # Add SavePlanItem to the BoxLayout

    def save_plan(self):
        user_id = self.manager.get_screen('initialpage').user_id
        plan_name= self.ids.plan_name.text
        workout_plan=self.workout_plans_by_day
        db_manager.save_complete_workout_plan(user_id, plan_name, workout_plan)
```







1.4.5 Extensive Exercise Library and Technique Guidance:

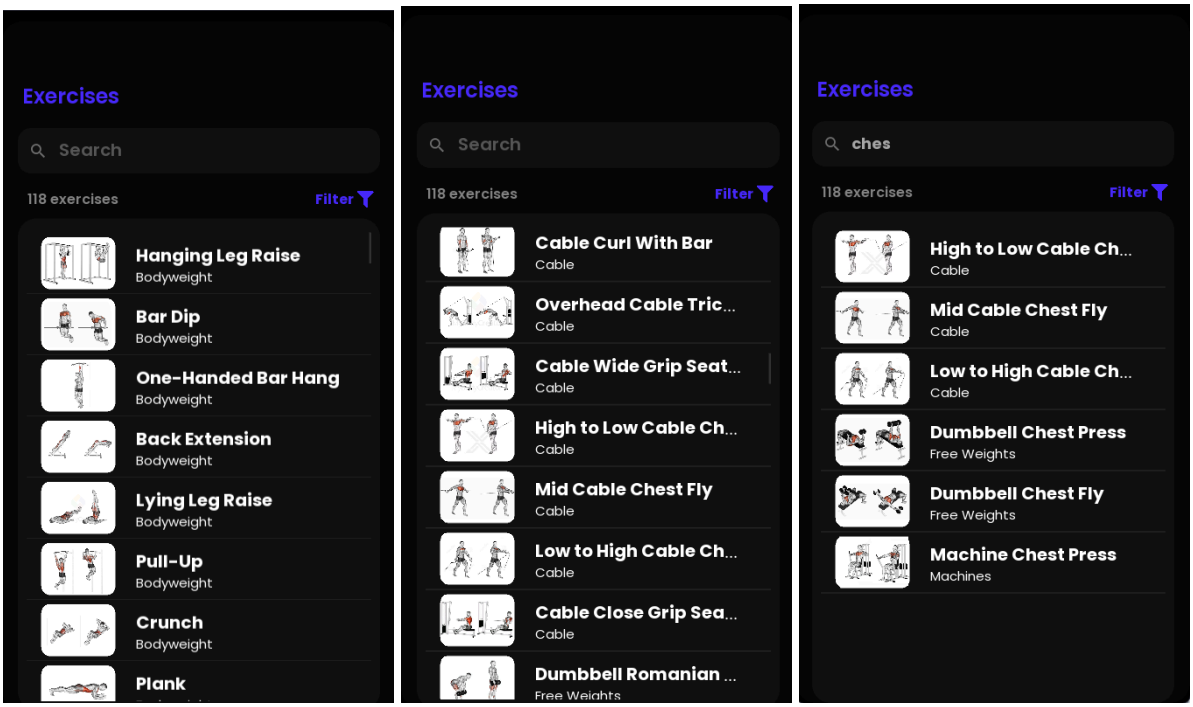
Achieved in the ExerciseList class:

```
class ExerciseList(Screen):
    # Class representing the screen for displaying a list of exercises
    search_delay = 0.3
    list_type = 'selectable'

    def __init__(self, **kwargs):
        # Initialize the screen and create a trigger for performing search with a delay
        super().__init__(**kwargs)
        self.search_trigger = Clock.create_trigger(self.perform_search, self.search_delay)
        self.item_clicked = False
        self.add_button_clicked = False
        self.selected_exercises = []
        self.guide= False
```

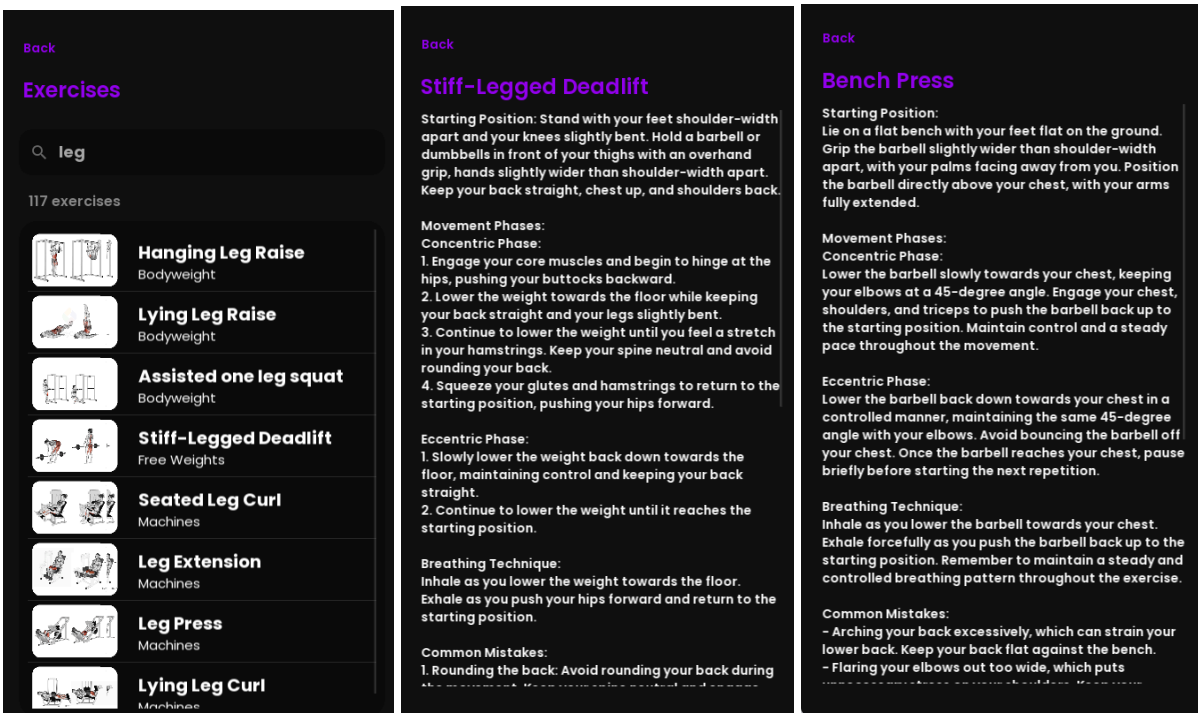
```
        self.pass_to_logworkout= False

    def on_kv_post(self, *args):
        # Populate the exercise list when entering the screen
        self.populate_exercises()
        self.on_list_type()

    def on_list_type(self, *args):
        # Adjust the UI based on the list type
        if self.list_type == 'list':
            self.ids.back_btn.text = 'Back'
            self.ids.add_exercises_layout.pos_hint = {'center_x': 2, 'center_y': 2}

    def populate_exercises(self, dt=None):
        # Populate the exercise list with data from the 'exercises' dictionary
        self.ids.rv.data = [{
            'text': f"[size=20][font=Poppins-Bold.ttf][color=#FFFFFF]{exercise['name']}[/color][/font][/size]",
            'secondary_text':
f"[size=15][font=Poppins-Regular.ttf][color=#FFFFFF]{exercise['type']}[/color][/font][/size]",
            'image_source': exercise['icon'],
            'is_checked': False,  # Add this line
        } for exercise in exercises]

    def perform_search(self, *args):
        # Perform a search in the exercise list based on the query from the search field
        query = self.ids.search_field.text
        filtered_exercises = [{
            'image_source': exercise['icon'],
            'text': f"[size=20][font=Poppins-Bold.ttf][color=#FFFFFF]{exercise['name']}[/color][/font][/size]",
            'secondary_text':
f"[size=15][font=Poppins-Regular.ttf][color=#FFFFFF]{exercise['type']}[/color][/font][/size]",
            'is_checked': False,  # Add this line
        } for exercise in exercises if query.lower() in exercise['name'].lower()]
        self.ids.rv.data = filtered_exercises

    def search_query(self, query):
        # Reset the countdown for the search trigger and adjust the opacity of the search label based on whether the
search field is empty
        if self.ids.search_field.text:
            self.search_trigger.cancel()
            self.ids.search_label.opacity=0
        else :
            self.ids.search_label.opacity=.3
        self.search_trigger.cancel()
        self.search_trigger()

    def on_back_btn(self):
        # Reset the UI to the initial state when the back button is clicked
        self.item_clicked = False
        self.ids.exercise_label.text = "Exercises"
        self.ids.exercise_guide_box.pos_hint = {'center_x': 2, 'center_y': 0.45}
        self.ids.rvcard.pos_hint = {'center_x': .5, 'center_y': 0.5}
        self.ids.search_btn.pos_hint= {'center_x': 0.5, 'center_y': 0.8}
        self.ids.search_box.pos_hint= {'center_x': 0.54, 'center_y': 0.803}
        self.ids.search_field.pos_hint= {'center_x': 0.5, 'center_y': 0.8}
        self.ids.exercise_count.pos_hint= {"center_x": 0.18, "top": .745}

    def show_exercise_guide(self, selected_exercise):
        # Display the exercise guide when the user clicks on the name of an exercise in the log workout screen
        self.list_type = 'list'
        self.on_list_type()
        self.on_item_click(selected_exercise)
        self.guide= True

    def on_item_click(self, clicked_exercise):
        # Update the UI to show the exercise guide when an exercise is clicked
        self.item_clicked= True
        self.ids.exercise_label.text = clicked_exercise
        self.ids.exercise_guide_text.text = exercise_technique[clicked_exercise]
        self.ids.exercise_guide_box.pos_hint = {'center_x': 0.5, 'center_y': 0.45}
        self.ids.rvcard.pos_hint = {'center_x': 2, 'center_y': 0.5}
        self.ids.search_btn.pos_hint = {'center_x': 0.5, 'center_y': 2}
        self.ids.search_box.pos_hint = {'center_x': 0.54, 'center_y': 2}
```

```
        self.ids.search_field.pos_hint = {'center_x': 0.5, 'center_y': 2}
        self.ids.exercise_count.pos_hint = {"center_x": 0.18, "top": 2}

    def empty_list(self):
        # Uncheck all the exercises in the list
        for instance in ExerciseItemPlan.instances:
            if isinstance(instance, ExerciseItemPlan):  # Check if the instance is of the ExercisePlanItem class
                instance.empty_plan = True
                instance.on_checkbox_active(instance, value=False)

    def on_checkbox_clicked(self, added_exercises):
        # Update the selected exercises and the add exercises button text when a checkbox is clicked
        self.selected_exercises = added_exercises
        self.ids.add_exercises_btn.text = f"Add ({len(self.selected_exercises)})"

    def on_exit_screen(self):
        # Uncheck all the exercises and clear the selected exercises when exiting the screen
        for item in MDApp.get_running_app().root.get_screen('exercise_list').ids.rv.data:
            item['is_checked'] = False
        for instance in ExerciseItemPlan.instances:
            instance.uncheck()
        self.empty_list()
        self.selected_exercises = []
        self.on_checkbox_clicked(added_exercises= [])

    def pass_exercises_to_logworkout(self):
        # Pass the selected exercises to the log workout screen
        if self.pass_to_logworkout:
            log_workout_screen = MDApp.get_running_app().root.get_screen('logworkout')
            log_workout_screen.add_exercise(self.selected_exercises)
            self.manager.current = "logworkout"
            self.pass_to_logworkout = False
            self.on_exit_screen()
        else:
            self.pass_to_logworkout = True

    def on_button_clicked(self):
        # Pass the selected exercises to the log workout screen or the empty plan screen when the button is clicked
        if self.pass_to_logworkout:
            self.pass_exercises_to_logworkout()
            return
        empty_plan_screen= MDApp.get_running_app().root.get_screen('empty_plan')
        empty_plan_screen.added_exercises_to_plan(self.selected_exercises)
        self.manager.current = "empty_plan"
        self.on_exit_screen()

    def change_screen(self):
        # Change the screen based on the current state
        if self.item_clicked:
            self.on_back_btn()
            if self.guide:
                self.manager.current = "logworkout"
        elif self.list_type == 'list':
            self.manager.current = "dashboard"
        else:
            self.manager.current= "empty_plan"
            self.on_exit_screen()
```

(before colour changes)



(after)



1.4.6 Workout Logging and History:

Achieved in LogWorkout class:

```
class LogWorkout(Screen):
    def __init__(self, **kwargs):
        super(LogWorkout, self).__init__(**kwargs)
        self.last_clicked_item = None  # Stores the last clicked item in the workout plan
        self.type = None  # Stores the type of the selected row (either "reps" or "weight")
        self.workout_rows = []  # Stores the rows of the current workout
        self.current_row = {}  # Stores the index of the current row in the workout
        self.selected_row = []  # Stores the index of the selected row in the workout
        self.sets= None  # Stores the number of sets for the current exercise
```

```
        self.current_item_id = 0  # Stores the id of the current item in the workout plan
        self.current_page = 1  # Stores the current page number in the workout plan
        self.plan= []  # Stores the workout plan
        self.workout_rows_instances = {}  # Stores the instances of WorkoutRow for each exercise in the plan
        self.workout_data = {}  # Stores the weight, reps, and tenrm for each set of each exercise

    def initialize_workout(self):
        # Initialize the workout_rows_instances dictionary with an empty list for each exercise in the plan
        self.current_row= {exercise : 0 for exercise in range(len(self.plan)+1)}
        # Initialize the workout_rows_instances dictionary with an empty list for each exercise in the plan
        self.workout_rows_instances= {exercise : [] for exercise in range(len(self.plan)+1)}
        # For each exercise in the plan, create a WorkoutImage and add it to the layout
        for i in range(len(self.plan)):
            item = WorkoutImage(source_image=((self.plan[i]['name']).lower().replace(" ", "")+".png"), item_id=i)
            self.ids.bl.add_widget(item)
        # Set the opacity of the first image for the exercise to 1 since all the other images are set to 0.5
        self.set_opacity(self.ids.bl.children[-1])
        self.add_plus_icon()
        # Set the type of the first workout row to "current" and update its icon
        self.workout_rows[0].type = "current"
        self.workout_rows[0].update_icon()
        # Store the current page number
        self.current_page= self.ids.logworkout_sm.current[-1]
        # Set the text of the exercise_name_1 label to the name of the first exercise in the plan
        self.ids.exercise_name_1.text = self.plan[0]['name']
        self.update_sets()

    def add_plus_icon(self):
        # Add a plus icon to the layout to allow the user to add new exercises to the plan
        item = WorkoutImage(source_image="plus_bg.png", item_id=len(self.plan), opacity=1)
        self.ids.bl.add_widget(item)

    def add_set(self):
        # Create a new instance of WorkoutRow
        new_row = WorkoutRow(
            pos_hint={'center_x': 0.5, 'center_y': 0.71 - 0.05 * self.plan[self.current_item_id]['sets']},  #
Position of the row
            set=str(self.plan[self.current_item_id]['sets'] + 1),  # Set number
        )
        # Add the new row to self.workout_rows
        self.workout_rows.append(new_row)
        # Add the new row to self.workout_rows_instances
        self.workout_rows_instances[self.current_item_id].append(new_row)
        # Add the new row to the appropriate layout in the UI
        log_sets_layout = f"log_sets_layout_{self.current_page}"
        self.ids[log_sets_layout].add_widget(new_row)
        # If the new set is the current set, update self.current_row and set the type of the new row to "current"
        if self.current_row[self.current_item_id]+1 == self.plan[self.current_item_id]['sets'] and
self.workout_rows[self.current_row[self.current_item_id]].type == "done":
            print("current row", self.current_row[self.current_item_id])
            self.current_row[self.current_item_id] += 1
            new_row.type = "current"
            new_row.update_icon()
        # Update self.plan to reflect the new number of sets
        self.plan[self.current_item_id]['sets'] += 1
        self.update_sets()

    def remove_set(self):
        # Check if there are more than one sets
        if len(self.workout_rows) > 1:
            # Remove the last row from self.workout_rows
            last_row = self.workout_rows.pop()
            # Remove the last row from self.workout_rows_instances
            if last_row in self.workout_rows_instances[self.current_item_id]:
                self.workout_rows_instances[self.current_item_id].remove(last_row)
            # Remove the last row from the appropriate layout in the UI
            log_sets_layout = f"log_sets_layout_{self.current_page}"
            self.ids[log_sets_layout].remove_widget(last_row)
            # Update self.plan to reflect the new number of sets
            self.plan[self.current_item_id]['sets'] -= 1
            # If the removed set was the current set, update self.current_row and set the type of the new last row to
"current"
            if self.current_row[self.current_item_id] == len(self.workout_rows):
```

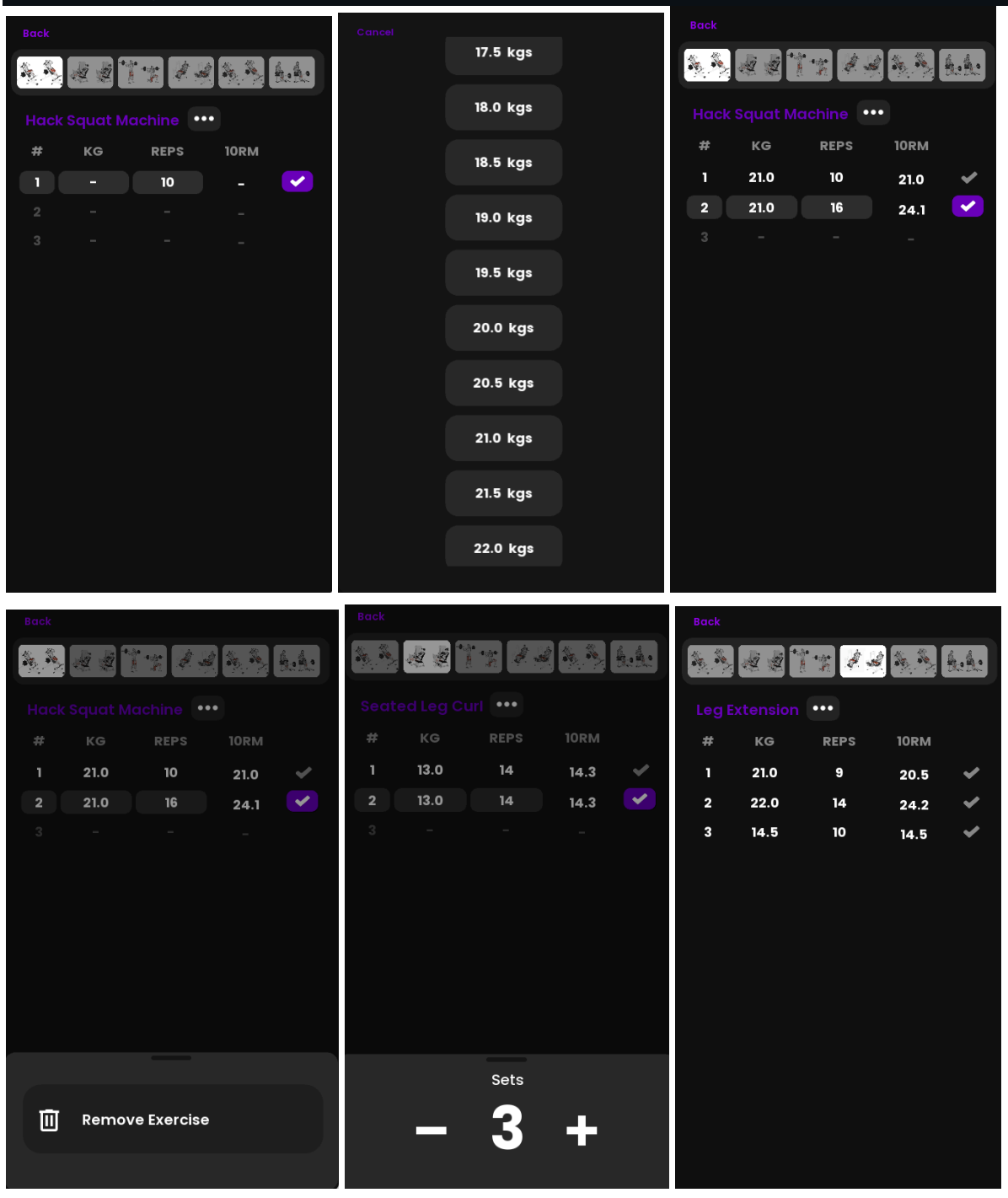
```
                self.current_row[self.current_item_id] -= 1
                self.workout_rows[-1].type = "current"
                self.workout_rows[-1].update_icon()
            # Update self.workout_data to remove the data for the removed set
            if self.current_item_id in self.workout_data and len(self.workout_rows) in
self.workout_data[self.current_item_id]:
                del self.workout_data[self.current_item_id][len(self.workout_rows)]
            self.update_sets()

    def update_sets(self):
        self.ids.set_label.text = str(self.plan[0]['sets'])

    def generate_rows(self):
        y = 0.71  # Initial y position for the first row
        log_sets_layout = f"log_sets_layout_{self.current_page}"  # Get a reference to log_sets_layout
        for i in range(1,(self.plan[self.current_item_id]['sets'])+1):  # For each set in the current exercise
            workout_row = WorkoutRow(
                pos_hint={'center_x': 0.5, 'center_y': y},  # Position of the row
                set=str(i),  # Set number
            )
            self.ids[log_sets_layout].add_widget(workout_row)  # Add the row to log_sets_layout
            self.workout_rows.append(workout_row)  # Add the row to the list of workout rows
            y -= 0.05  # Decrease the y position for the next row
        self.workout_rows[0].reps= str(self.plan[self.current_item_id]['reps'])  # Set the reps for the first row

    def add_exercise(self, exercise_names):
        # Remove the plus icon from the box layout
        if self.ids.bl.children:
            self.ids.bl.remove_widget(self.ids.bl.children[0])

        # Loop through the list of exercise names and create a new WorkoutImage for each exercise
        for i in range(len(exercise_names)):
            # The item_id is the current length of the plan plus the index of the exercise
            item = WorkoutImage(source_image=((exercise_names[i]).lower().replace(" ", "")+".png"),
item_id=(len(self.plan)+i))
            self.ids.bl.add_widget(item)

        self.add_plus_icon() # Add the plus icon back to the box layout
        # Loop through the list of exercise names again
        for exercise in exercise_names:
            # Add a new exercise to the plan with the default sets and reps
            self.plan.append({'name': exercise, 'sets': 3, 'reps': 10})
            # Set the current row for the new exercise to 0
            self.current_row[len(self.plan)-1] = 0
            # Initialize the workout rows instances for the new exercise to an empty list
            self.workout_rows_instances[len(self.plan)-1] = []

        # Add a new entry for the exercise in current_row and workout_rows_instances
        new_index = len(self.plan) - 1
        self.current_row[new_index] = 0
        self.workout_rows_instances[new_index] = []

        # Set the opacity of the first added image to 1 and change the screen to that
        self.set_opacity(self.ids.bl.children[(-(len(self.plan)-len(exercise_names)+1))])


    def remove_exercise(self):
        exercise_index = self.current_item_id

        # Remove the exercise from the plan, current_row, and workout_rows_instances
        del self.plan[exercise_index]
        del self.current_row[exercise_index]
        self.workout_rows.clear()
        if self.workout_rows_instances[exercise_index] != []:
            del self.workout_rows_instances[exercise_index]

        # Remove the exercise from workout_data
        if exercise_index in self.workout_data:
            del self.workout_data[exercise_index]

        # Re-index the remaining exercises
        for i in range(exercise_index, len(self.plan)):
            self.current_row[i] = self.current_row.pop(i + 1)
```

```
            self.workout_rows_instances[i] = self.workout_rows_instances.pop(i + 1)
            if i + 1 in self.workout_data:
                self.workout_data[i] = self.workout_data.pop(i + 1)

        # Update the UI
        self.ids.bl.clear_widgets()
        for i in range(len(self.plan)):
            item = WorkoutImage(source_image=((self.plan[i]['name']).lower().replace(" ", "")+".png"), item_id=i)
            self.ids.bl.add_widget(item)
        self.add_plus_icon()

        if self.current_item_id < len(self.plan) - 1:
            self.set_opacity(self.ids.bl.children[(-(exercise_index+1))])
        else:
            self.set_opacity(self.ids.bl.children[-1])


    def change_screen(self, item_id):
        print("screen changed")
        # Determine the direction of the transition based on the item_id
        if item_id > self.current_item_id:
            self.ids.logworkout_sm.transition.direction = 'left'
        elif item_id < self.current_item_id:
            self.ids.logworkout_sm.transition.direction = 'right'
        elif item_id == len(self.plan):
            return
        # Switch between the two pages to give the illusion of multiple pages
        if self.ids.logworkout_sm.current == "logworkout_scr_1":
            self.ids.logworkout_sm.current = "logworkout_scr_2"
        else:
            self.ids.logworkout_sm.current = "logworkout_scr_1"
        # Update the current page number
        self.current_page = self.ids.logworkout_sm.current[-1]
        # Store the current workout rows
        self.workout_rows_instances[self.current_item_id] = self.workout_rows
        # Update the current item id
        self.current_item_id = item_id
        # Update the exercise name label
        exercise_name_id = f"exercise_name_{self.current_page}"
        self.ids[exercise_name_id].text = self.plan[self.current_item_id]['name']
        #update position of the three dots icon
        dots_rec_id = f"dots_rec_{self.current_page}"
        dots_id= f"dots_{self.current_page}"
        #find the width of the exercise name based on a dictionary of character widths that i made
        characters = characters_width_dict.get_text_width(self.plan[self.current_item_id]['name'])
        #use a formula that i found through trial and error to calculate the distance the icon should move
        distance = round((characters - 31.8)*0.0113,2)
        self.ids[dots_rec_id].pos_hint= {'center_x': 0.48+distance,'center_y': 0.82}
        self.ids[dots_id].pos_hint = {'center_x': 0.48+distance,'center_y': 0.82}
        # Clear the current sets layout
        log_sets_layout = f"log_sets_layout_{self.current_page}"
        self.ids[log_sets_layout].clear_widgets()
        # If there are no workout rows for the current item, generate new rows
        if self.workout_rows_instances[self.current_item_id] == []:
            self.workout_rows = []
            self.generate_rows()
            self.workout_rows[0].type = "current"
            self.workout_rows[0].update_icon()

        else:
            # Otherwise, load the existing workout rows
            self.workout_rows = []
            self.workout_rows = self.workout_rows_instances[self.current_item_id]
            for i in range(len(self.workout_rows)):
                widget = self.workout_rows[i]
                if widget.parent:
                    widget.parent.remove_widget(widget)
                self.ids[log_sets_layout].add_widget(widget)

    def set_opacity(self, selected_item):
        # Set the opacity of the last clicked image for the exercise to 0.5
        if self.last_clicked_item:
            self.last_clicked_item.opacity = 0.5
```

```
        selected_item.opacity = 1 # Set the opacity of the selected image for the exercise to 1
        self.last_clicked_item = selected_item

        if selected_item.item_id == len(self.plan):
            self.manager.transition= NoTransition()
            MDApp.get_running_app().root.get_screen('exercise_list').pass_exercises_to_logworkout()
            self.manager.current = "exercise_list"
            return
        self.change_screen(selected_item.item_id)  # Change the screen to the selected exercise

    def on_cancel(self):
        #method to move the choose_item widget off the screen
        self.ids.choose_item.pos_hint = {'center_x': 2.5,'center_y': 0.5}

    def choose_item(self, row, type):
        # Store the index of the selected row
        self.selected_row = int(row) -1
        # Check if the selected row is a row that is not marked as done or current, if so, exit the method
        if self.workout_rows[self.selected_row].type == "todo":
            return
        # Move the choose_item widget to the center of the screen
        self.ids.choose_item.pos_hint = {'center_x': 0.5,'center_y': 0.5}
        # Store the type of the selected row (either "reps" or "weight")
        self.type= type
        # Depending on the type, populate the recycleview with the appropriate data
        if type == "reps":
            self.ids.choose_item.ids.choose_item_rv.data = [{'text': f'{i}  rep'} if i== 1 else {'text': f'{i}
reps'} for i in range(1, 100)]
        elif type == "weight":
            self.ids.choose_item.ids.choose_item_rv.data = [{'text': f'{i/2}  kgs'} for i in range(1, 300)]

    def chosen_item(self, number):
        # Depending on the type, set the reps or kg of the selected row to the chosen number
        if self.type == "reps":
            self.workout_rows[self.selected_row].reps = number
        elif self.type == "weight":
            self.workout_rows[self.selected_row].kg = number
        # Move the choose_item widget off the screen
        self.ids.choose_item.pos_hint = {'center_x': 2.5,'center_y': 0.5}
        # if an item is changed and it was previously marked as done, update the workout data
        if self.workout_rows[self.selected_row].type == "done":
            if self.current_item_id not in self.workout_data:
                self.workout_data[self.current_item_id] = {}
            self.workout_data[self.current_item_id][self.selected_row[self.current_item_id]] = {
                'weight': self.workout_rows[self.selected_row[self.current_item_id]].kg,
                'reps': self.workout_rows[self.selected_row[self.current_item_id]].reps,
                'tenrm': self.workout_rows[self.selected_row[self.current_item_id]].tenrm,
            }
        # Update the icon of the selected row
        self.workout_rows[self.selected_row].update_icon()
        # Calculate the 10 rep max for the selected row
        self.calculate10RM()

    def next_row(self):
        # Check if the current row is not empty
        if self.workout_rows[self.current_row[self.current_item_id]].type == "current" and
self.workout_rows[self.current_row[self.current_item_id]].reps != "-" and
self.workout_rows[self.current_row[self.current_item_id]].kg != "-":
            # Mark the current row as done
            self.workout_rows[self.current_row[self.current_item_id]].type = "done"
            # Update the icon of the current row
            self.workout_rows[self.current_row[self.current_item_id]].update_icon()
            # Update the workout data for the current row
            if self.current_item_id not in self.workout_data:
                self.workout_data[self.current_item_id] = {}
            self.workout_data[self.current_item_id][self.current_row[self.current_item_id] + 1] = {
                'weight': self.workout_rows[self.current_row[self.current_item_id]].kg,
                'reps': self.workout_rows[self.current_row[self.current_item_id]].reps,
                'tenrm': self.workout_rows[self.current_row[self.current_item_id]].tenrm,
            }
            # If there are more sets in the current exercise, move to the next row
            if self.current_row[self.current_item_id] < self.plan[self.current_item_id]['sets'] - 1:
               # Copy values from the current (now completed) row to the next row
```

```
                completed_row = self.workout_rows[self.current_row[self.current_item_id]]
                next_row = self.workout_rows[self.current_row[self.current_item_id] + 1]

                # C
                next_row.reps = completed_row.reps
                next_row.kg = completed_row.kg
                next_row.tenrm = completed_row.tenrm
                # Move to the next row and mark it as the current row
                self.current_row[self.current_item_id] += 1
                self.workout_rows[self.current_row[self.current_item_id]].type = "current"
                self.workout_rows[self.current_row[self.current_item_id]].update_icon()
            else:
                def check_and_switch():
                    for i in range(len(self.plan)):
                        if i not in self.workout_data or len(self.workout_data[i]) != self.plan[i]['sets']:
                            self.set_opacity(self.ids.bl.children[-(i+1)])
                            return

                # Check if the current exercise is the last one in the plan
                if self.current_item_id == len(self.plan) - 1:
                    # Check if all exercises are done
                    if len(self.workout_data) == len(self.plan):
                        check_and_switch()
                        self.ids.save_workout_panel.toggle()
                        self.ids.behind_panel_rec3.pos_hint = {'center_x': 0.5, 'center_y': 0.5}

                    else:
                        # If not all exercises are done check which ones are not done and switch to that screen
                        for i in range(len(self.plan)):
                            if i not in self.workout_data:
                                self.set_opacity(self.ids.bl.children[-(i+1)])
                                break
                else:
                    # Check if all sets in the next exercises are done
                    if self.current_item_id + 1 in self.workout_data and len(self.workout_data[self.current_item_id +
1]) == self.plan[self.current_item_id + 1]['sets']:
                        print("All sets in the next exercise are done")
                        check_and_switch()
                        self.ids.save_workout_panel.toggle()
                        self.ids.behind_panel_rec3.pos_hint = {'center_x': 0.5, 'center_y': 0.5}
                    else:
                        # If there are no more sets in the current exercise, move to the next exercise
                        self.set_opacity(self.ids.bl.children[-(self.current_item_id +2)])
                        return

                # If all exercises are done, print the workout data
                if self.current_item_id == len(self.plan)-1:
                    self.ids.save_workout_panel.toggle()
                    self.ids.behind_panel_rec3.pos_hint = {'center_x': 0.5, 'center_y': 0.5}
                else:
                    # Otherwise, update the opacity of the next exercise image
                    self.set_opacity(self.ids.bl.children[-(self.current_item_id)-1])
        else:
            # If the current row is empty, shake the row
            self.workout_rows[self.current_row[self.current_item_id]].shake_animation()

    def calculate10RM(self):
        # Check if the selected row has valid kg and reps values
        if self.workout_rows[self.selected_row].kg != "-" and self.workout_rows[self.selected_row].reps != "-":
            # Get the weight and reps from the selected row
            weight = self.workout_rows[self.selected_row].kg
            reps = self.workout_rows[self.selected_row].reps
            # Calculate the one rep max using the Epley formula
            onerm= float(weight) * (1 + (0.0333 * (float(reps))))
            # Calculate the ten rep max as 75% of the one rep max
            tenrm = round(onerm * 0.75,1)
            # Store the ten rep max in the selected row
            self.workout_rows[self.selected_row].tenrm = str(tenrm)

    def save_workout(self):
        named_workout_data = {}
        for i, exercise in enumerate(self.plan):
            if i in self.workout_data:
```

```
            named_workout_data[exercise['name']] = self.workout_data[i]

    def on_back(self):
        exercises_completed = 0
        #store the exercise as 1 if all the sets are done and 0 if not
        for i in range(len(self.plan)):
            if i in self.workout_data and len(self.workout_data[i]) == self.plan[i]['sets']:
                exercises_completed += 1
        MDApp.get_running_app().root.get_screen('dashboard').exercise_completed= exercises_completed

    def on_single_workout(self):
        #method that is called when the user wants to directly start a workout without selecting a plan
        self.plan=[] #clear the plan
        #get the exercises from the exercise list screen which passes to the add_exercise method
        self.manager.transition= NoTransition()
        MDApp.get_running_app().root.get_screen('exercise_list').pass_exercises_to_logworkout()
        self.manager.current = "exercise_list"
```



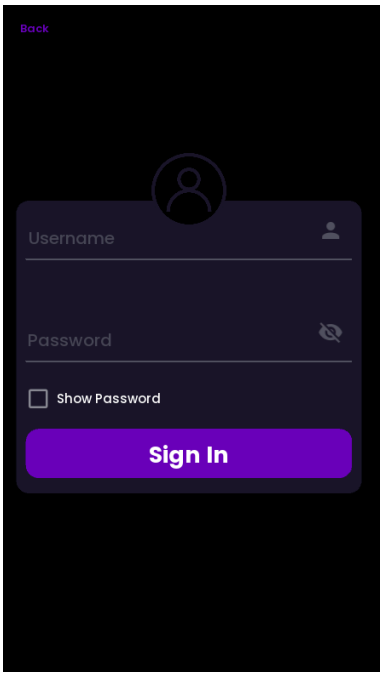
1.4.7 Secure User Authentication and Data Handling:

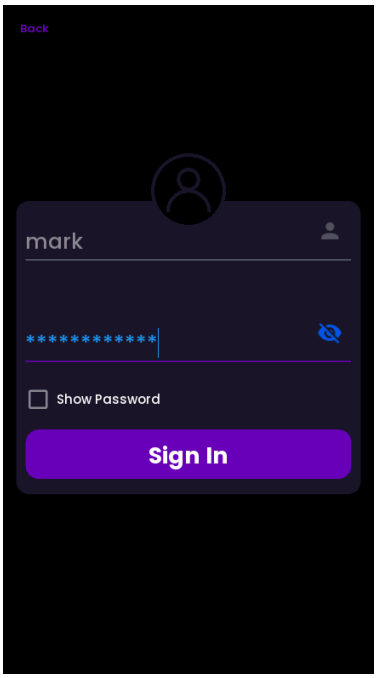
Achieved in Signup and Login classes:

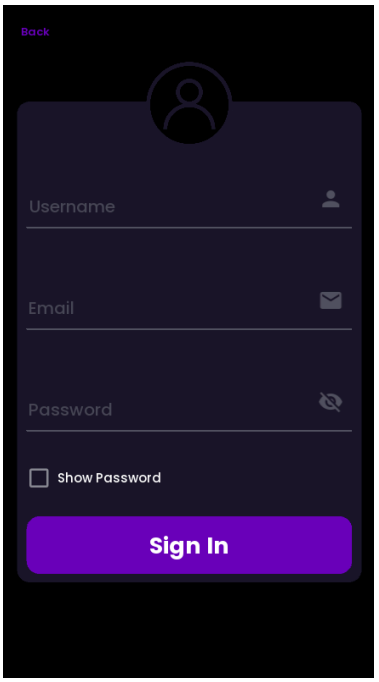
```
class SignupPage(Screen):
    # Initialize successful_signup to False
    successful_signup = False

    def signup(self):
        # Get user input from the signup form
        username = self.ids.signup_username.text
        email = self.ids.signup_email.text
        password = self.ids.signup_password.text

        # Validate the user input
        if not (username and email and password):
            self.ids.signup_error.text = "Please fill in all fields."
            self.signup_success = False
            return

        # Check if the username already exists in the database
        db_manager = DatabaseManager(db_config)
        if db_manager.get_user_by_username(username):
            self.ids.signup_error.text = "Username already exists."
            self.signup_success = False
            return

        # Validate the email address format
        if not re.match(r"[^@]+@[^@]+\.[^@]+", email):
            self.ids.signup_error.text = "Please enter a valid email address."
            self.successful_signup = False
            return

        # Validate the password complexity
        password_validation_result = self.validate_password_complexity(password)
        if password_validation_result is not True:
            self.ids.signup_error.text = password_validation_result
            self.successful_signup = False
            return

        # Attempt to create the user in the database
        try:
            self.successful_signup = True
            db_manager = DatabaseManager(db_config)
            user_id = self.manager.get_screen('initialpage').user_id
            user = UserManager.get_user(user_id)
            user.update_email(email)
            db_manager.insert_user(username, email, password, user_id)
            toast("Signup successful.")
            self.ids.signup_error.text = ""
        except Exception as e:
            self.ids.signup_error.text = str(e)
    def validate_password_complexity(self, password):
        # Validate the complexity of the password
        if len(password) < 8:
            return "Password must be at least 8 characters long."
        if not re.search(r"[A-Z]", password):
            return "Password must contain at least one uppercase letter."
        if not re.search(r"[a-z]", password):
            return "Password must contain at least one lowercase letter."
        if not re.search(r"[0-9]", password):
            return "Password must contain at least one number."
        if not re.search(r"[!@#$%^&*(),.?\":{}|<>]", password):
            return "Password must contain at least one special character."
        return True

    def on_pre_enter(self, *args):
        # Reset the form fields and error messages when the signup page is about to be entered
        self.ids.signup_username.text = ""
        self.ids.signup_email.text = ""
        self.ids.signup_password.text = ""
        self.ids.signup_error.text = ""
```

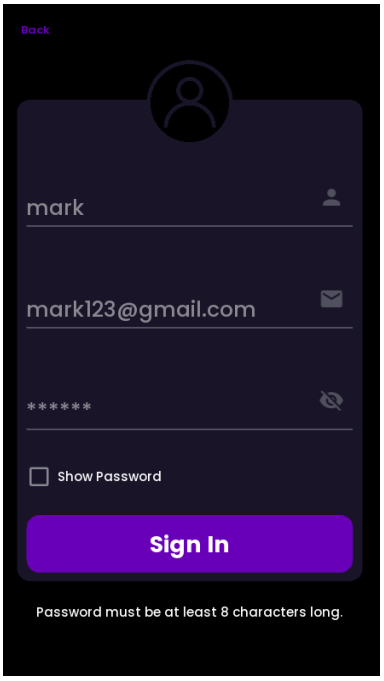
```
    def on_enter(self, *args):
        # Set the focus to the username field when the signup page is entered
        Clock.schedule_once(lambda dt: self.ids.signup_username.focus == True)

    def change_screen(self):
        # Attempt to signup the user and navigate to the selectgender screen if signup is successful
        self.signup()
        if self.successful_signup == True:
            self.manager.current = "dashboard"
class LoginPage(Screen):
    def on_pre_enter(self, *args):
        # Reset the error message when the login page is about to be entered
        self.ids.error.text = ''

    def login(self):
        # Get user input from the login form
        username = self.ids.username.text
        password = self.ids.password.text

        # Validate the user input
        if not (username and password):
            self.ids.error.text = 'Please enter both username and password'
            return

        # Attempt to login the user
        try:
            db_manager = DatabaseManager(db_config)
            user_data = db_manager.get_user_by_username(username)
            if user_data:
                user_id, username, hashed_password = user_data
                # Check if the entered password matches the stored password
                if bcrypt.checkpw(password.encode('utf-8'), hashed_password.encode('utf-8')):
                    # Get the user's preferences from the database
                    UserManager.get_user(user_id)
                    # Navigate to the dashboard screen
                    self.manager.current = 'dashboard'  # Assuming 'home_page' is the name of the next screen
                    self.ids.error.text = ''
                    toast("Login successful!")
                else:
                    # Show an error message if the login failed
                    self.ids.error.text = 'Incorrect username or password'
                    # Clear the password field for security reasons
                    self.ids.password.text = ''
        except Exception as e:
            # Show an error message if the login process failed
            self.ids.error.text = 'Login failed. Please try again.'
            logging.error("Error during login: {}".format(e))
```

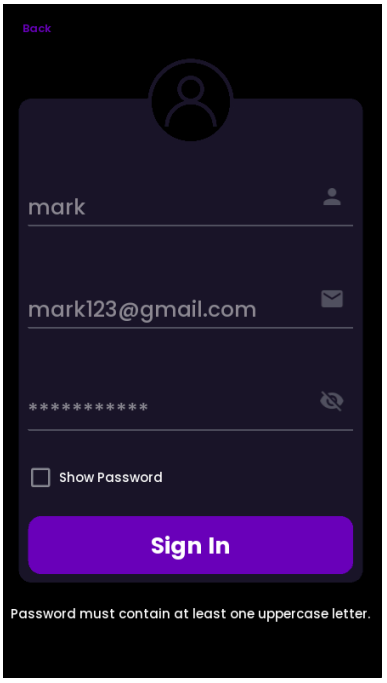
(after colour change)

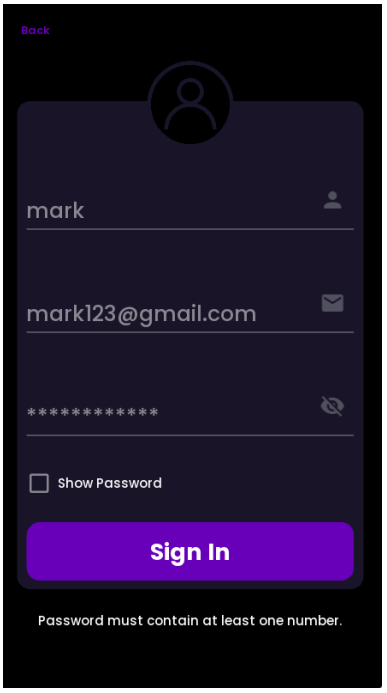












1.4.9 Comprehensive Tracking Features

Achieved in InputHeightWeight class, InputAge, InputBodyfat classes, LogWorkout, Foodsearch classes:

```
class InputAge(Screen):
    def validate_age(self, age):
        # Validate the age input by the user. If the age is a digit and within the valid range (10 to 100),
        # update it in the database, clear the validation message, and navigate to the next screen.
        # If the age is not valid, display a validation message.
        if age.isdigit() and 10 <= int(age) <= 100:
            self.update_age(age)
            self.ids.validage.text = ""
            self.manager.current = "inputheightweight"
        else:
```

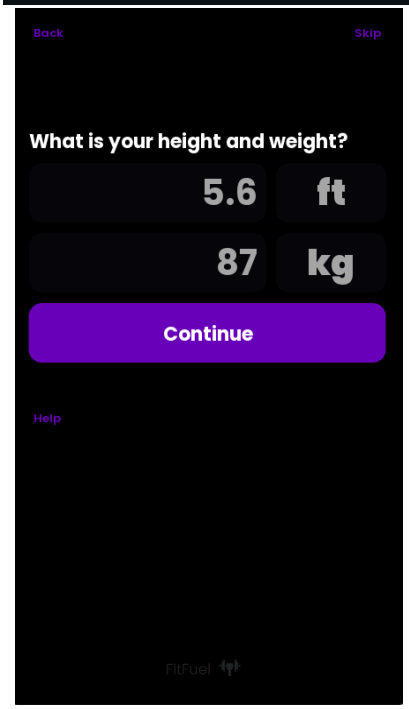
```
            self.ids.validage.text = "Please enter a valid age between 10 and 100"

    def update_age(self, age):
        # Convert the input age to float, retrieve the user_id from the 'initialpage' screen,
        # get the user object using this id, and update the user's age in the database.
        self.age = float(age)
        user_id = self.manager.get_screen('initialpage').user_id
        user = UserManager.get_user(user_id)
        user.update_age(age)
class InputHeightWeight(Screen): #this will be displayed after the InputAge screen, and it takes the User's Height
and Weight as input
    invalidhw = False  # Indicates whether the height and weight values are invalid
    height = None
    weight = None
    height_unit = StringProperty('m')
    weight_unit = StringProperty('kg')

    # Method to toggle height unit between meters and feet
    def toggle_height_unit(self):
        self.height_unit = 'ft' if self.height_unit == 'm' else 'm'

    # Method to toggle weight unit between kilograms and pounds
    def toggle_weight_unit(self):
        self.weight_unit = 'lbs' if self.weight_unit == 'kg' else 'kg'


    def convert_feet_to_meters(self, feet, inches):
        # Convert feet and inches to meters
        return round(feet * 0.3048 + inches * 0.0254, 2)

    def convert_pounds_to_kilograms(self, pounds):
        # Convert pounds to kilograms
        return round(pounds * 0.45359237,2)

    def validate_height(self, height):
        # Validate the height
        if 0.50 <= height <= 2.50:
            return True
        self.ids.validhw.text = "Please enter a valid height."
        self.ids.validhw.font_size = 16
        return False

    def validate_weight(self, weight):
        # Validate the weight
        if 15 <= weight <= 250:
            return True
        self.ids.validhw.text = "Please enter a valid weight."
        self.ids.validhw.font_size = 16
        return False

    def output(self, height_str, weight_str):
        self.invalidhw = False  # Reset the invalid flag

        # Check if both height and weight are empty.
        if not height_str or not weight_str:
            self.ids.validhw.text = "Please enter all the values for height and weight."
            self.ids.validhw.font_size = 12
            self.invalidhw = True
            return

        # Parse height and weight
        try:
            if self.height_unit == 'ft':
                # Assume the format for feet and inches is "ft.in"
                feet, inches = map(int, height_str.split('.'))
                height = self.convert_feet_to_meters(feet, inches)
            else:
                height = float(height_str)

            weight = float(weight_str)
            if self.weight_unit == 'lbs':
                weight = self.convert_pounds_to_kilograms(weight)
```

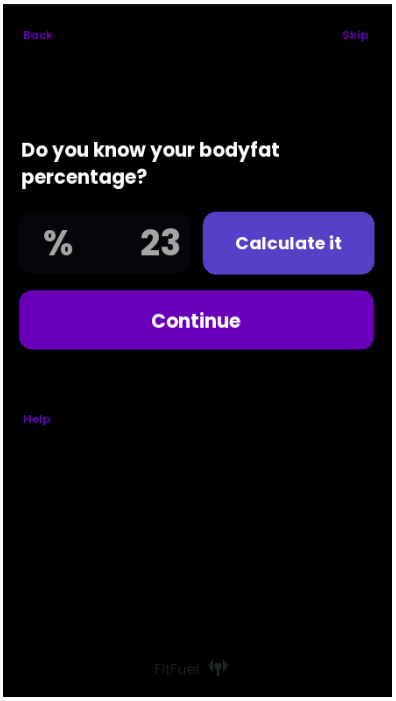
```
        except ValueError:
            self.ids.validhw.text = "Invalid format for height or weight."
            self.invalidhw = True
            return

        # Validate height and weight
        if not self.validate_height(height) or not self.validate_weight(weight):
            self.invalidhw = True
            return

        # If validation passes, update the database
        self.height = height
        self.weight = weight

        user_id = self.manager.get_screen('initialpage').user_id
        user = UserManager.get_user(user_id)
        user.update_height(height)
        user.update_weight(weight)

        # If no errors, move to the next screen
        if not self.invalidhw:
            self.manager.current = "experiencelevel"
class InputBodyfat(Screen):
    # Class representing the screen for inputting user's bodyfat percentage
    invalidbf = False
    bodyfat = ""

    def validbodyfat(self, bodyfat):
        # Method to validate the bodyfat input by the user
        # If the input is empty, a validation message is displayed
        if bodyfat == "":
            self.ids.validbf.text = "Please enter your bodyfat percentage"
            self.invalidbf = True
            self.bodyfat = ""
        else:
            # If the input is not empty, it is converted to float and checked if it's within the valid range (4 to
50)
            # If it's not within the range, a validation message is displayed
            # If it's within the range, the bodyfat percentage is updated in the database
            bodyfat = float(bodyfat)
            if bodyfat > 50 or bodyfat < 4:
                self.ids.validbf.text = "Please enter a valid bodyfat percentage"
                self.invalidbf = True
            else:
                self.ids.validbf.text = ""
                self.invalidbf = False
                self.bodyfat = bodyfat

                #update the bodyfat in the user class
                user_id = self.manager.get_screen('initialpage').user_id
                user = UserManager.get_user(user_id)
                user.update_bodyfat(bodyfat)
class CalculateBodyFat(Screen):
    # Class representing the screen for calculating user's bodyfat percentage
    invalidneck= False
    invalidwaist= False
    invalidhip= False
    no_measurement= False
    bodyfat=float()

    def calculatebf(self, pressed):
        # Method to calculate the bodyfat percentage based on the user's measurements
        # Retrieve the user's details from the database or user input
        user_id = self.manager.get_screen('initialpage').user_id
        user = UserManager.get_user(user_id)
        gender= user.gender
        height= user.height
        weight= user.weight
        age= int(user.age)
        neck= self.ids.neck.text
        waist= self.ids.waist.text
        hip= self.ids.hip.text
        if gender == "female":
```

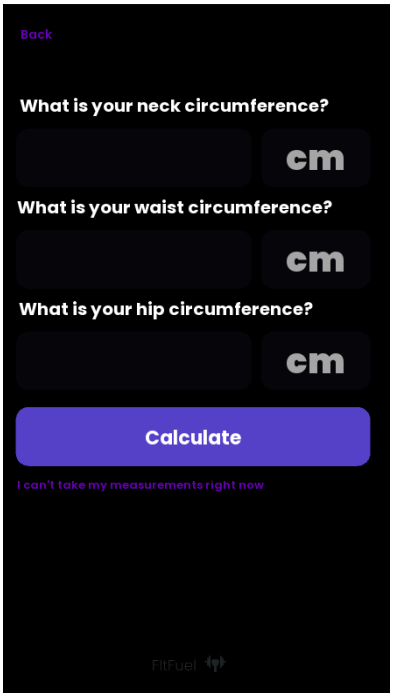
```
            # If the user is female, modify the layout of the input bodyfat screen to include hip measurement
            # ... layout modification code ...
            self.ids.neckcircumference.pos_hint = {"center_x": .452, "center_y": 0.855}
            self.ids.neckinputrectangle.pos_hint = {"center_x": 0.348, "center_y": 0.78}
            self.ids.neckunitrectangle.pos_hint = {"center_x": 0.81, "center_y": 0.78}
            self.ids.neckunit.pos_hint= {"center_x": 0.81, "center_y": 0.78}
            self.ids.neck.pos_hint= {"center_x": 0.348, "center_y": 0.78}
            self.ids.waistcircumference.pos_hint = {"center_x": .452, "center_y": 0.709}
            self.ids.waistinputrectangle.pos_hint = {"center_x": 0.348, "center_y": 0.634}
            self.ids.waistunitrectangle.pos_hint = {"center_x": 0.81, "center_y": 0.634}
            self.ids.waistunit.pos_hint= {"center_x": 0.81, "center_y": 0.634}
            self.ids.waist.pos_hint= {"center_x": 0.348, "center_y": 0.634}
            self.ids.hipcircumference.pos_hint = {"center_x": .43, "center_y": 0.563}
            self.ids.hipinputrectangle.pos_hint = {"center_x": 0.348, "center_y": 0.488}
            self.ids.hipunitrectangle.pos_hint = {"center_x": 0.81, "center_y": 0.488}
            self.ids.hipunit.pos_hint= {"center_x": 0.81, "center_y": 0.488}
            self.ids.hip.pos_hint= {"center_x": 0.348, "center_y": 0.488}
            self.ids.calculaterectangle.pos_hint = {"center_x": 0.498, "center_y": 0.379}
            self.ids.no_measurement.pos_hint = {"center_y" : 0.31, "center_x" : 0.365}
            self.ids.validnhw.pos_hint = {"center_x": 0.5, "center_y": 0.324}
            self.ids.calculate.pos_hint = {"center_x": 0.498, "center_y": 0.379}
            self.ids.bf_rectangle.size_hint= .9,.1
            self.ids.bf_rectangle.pos_hint= {'center_x': 0.498,'center_y': 0.268}
            self.ids.estimate_text.pos_hint= {'center_x': 0.498,'center_y': 0.298}

            # If no measurements are provided, calculate bodyfat using BMI and age, less accurate but still useful
            if self.no_measurement == True:
                pressed= False
                BMI= round(weight / (height * height), 1)
                bodyfat= round((1.20 * BMI) + (0.23 * age) - 5.4, 1)
                self.bodyfat= bodyfat

                #update the bodyfat in the user class
                user_id = self.manager.get_screen('initialpage').user_id
                user = UserManager.get_user(user_id)
                user.update_bodyfat(bodyfat)
            # If measurements are provided, calculate bodyfat using the US Navy method after the calculte button is
pressed
            if pressed== True:
                if neck == "" and waist == "" and hip == "":
                    self.ids.validnhw.text = "Please enter your neck, waist and hip circumference"
                    self.ids.validnhw.font_size = 10
                    self.invalidneck = True
                    self.invalidwaist = True
                    self.invalidhip = True
                    return
                # If only one or two measurements are provided, display a validation message
                elif neck == "" or waist == "" or hip == "":
                    self.ids.validnhw.text = "Please enter all the values for neck, waist and hip circumference"
                    self.ids.validnhw.font_size = 10
                    self.invalidneck = True
                    self.invalidwaist = True
                    self.invalidhip = True
                    return
                else:
                    # If all measurements are provided, convert them to float and check if they are within the valid
range

                    neck = float(neck)
                    waist = float(waist)
                    hip = float(hip)
                    if self.ids.neckunit.text == "in":
                        neck = round(neck * 2.54, 2)
                    if self.ids.waistunit.text == "in":
                        waist = round(waist * 2.54, 2)
                    if self.ids.hipunit.text == "in":
                        hip = round(hip * 2.54, 2)

                    # Display a validation message if any of the measurements are not within the valid range
                    if neck < 20 or neck > 60:
                        self.ids.validnhw.text = "Please enter a valid neck circumference"
                        self.ids.validnhw.font_size = 10
                        self.invalidneck = True
                        return
```

```
                    if waist < 50 or waist > 150:
                        self.ids.validnhw.text = "Please enter a valid waist circumference"
                        self.ids.validnhw.font_size = 10
                        self.invalidwaist = True
                        return
                    if hip < 50 or hip > 150:
                        self.ids.validnhw.text = "Please enter a valid hip circumference"
                        self.ids.validnhw.font_size = 10
                        self.invalidhip = True
                        return
                    else:
                        self.invalidneck = False
                        self.invalidwaist = False
                        self.invalidhip = False
                        self.ids.validnhw.text = ""
                        # If all measurements are valid, display the bodyfat percentage and update it in the database
                        if self.invalidneck == False and self.invalidwaist == False and self.invalidhip == False:
                            self.ids.bf_rectangle.opacity= 1
                            self.ids.estimate_text.opacity= 1
                            self.ids.no_measurement.pos_hint = {"center_y" : 2, "center_x" : 0.365}
                            self.ids.estimate_text.font_size= 14
                            self.ids.bf_continue.pos_hint= {'center_x': 0.498,'center_y': 0.16}
                            self.ids.bf_continue_text.pos_hint= {'center_x': 0.498,'center_y': 0.16}
                            self.ids.bf_continue_text.opacity= 1

                        # Calculate bodyfat percentage using the US Navy Body Fat Formula
                        bodyfat= round(495 / (1.29579 - 0.35004 * math.log10(waist + hip - neck) + 0.22100 *
math.log10(height*100)) - 450, 1)
                        self.bodyfat= bodyfat
                        self.ids.bodyfat_percent.font_size= 45
                        self.ids.bodyfat_percent.pos_hint= {'center_x': 0.498,'center_y': 0.255}
                        self.ids.bodyfat_percent.text = f"{bodyfat}%"

                        user_id = self.manager.get_screen('initialpage').user_id
                        user = UserManager.get_user(user_id)
                        user.update_bodyfat(bodyfat)
            else:
                # if the gender is male, modify the layout of the input bodyfat screen to exclude hip measurement
                if self.no_measurement == True:
                    # If no measurements are provided, calculate bodyfat using BMI and age, less accurate but still
useful, the formula is different for men and women
                    BMI= round(weight / (height * height), 1)
                    bodyfat= round((1.20 * BMI) + (0.23 * age) - 16.2, 1)
                    self.bodyfat= bodyfat

                    user_id = self.manager.get_screen('initialpage').user_id
                    user = UserManager.get_user(user_id)
                    user.update_bodyfat(bodyfat)

                # If measurements are provided, calculate bodyfat using the US Navy method after the calculte button is
pressed
                if pressed== True:
                    #validation for the measurements
                    if neck == "" and waist == "":
                        self.ids.validnhw.text = "Please enter your neck and waist circumference"
                        self.ids.validnhw.font_size = 10
                        self.invalidneck = True
                        self.invalidwaist = True
                        return
                    elif neck == "" or waist == "":
                        self.ids.validnhw.text = "Please enter all the values for neck and waist circumference"
                        self.ids.validnhw.font_size = 10
                        self.invalidneck = True
                        self.invalidwaist = True
                        return
                    else:
                        # Convert the measurements to float and check if they are within the valid range
                        neck = float(neck)
                        waist = float(waist)
                        if self.ids.neckunit.text == "in":
                            neck = round(neck * 2.54, 2)
                        if self.ids.waistunit.text == "in":
                            waist = round(waist * 2.54, 2)
```

```
                    if neck < 20 or neck > 60:
                        self.ids.validnhw.text = "Please enter a valid neck circumference"
                        self.ids.validnhw.font_size = 10
                        self.invalidneck = True
                        return
                    if waist < 50 or waist > 150:
                        self.ids.validnhw.text = "Please enter a valid waist circumference"
                        self.ids.validnhw.font_size = 10
                        self.invalidwaist = True
                        return
                    else:
                        # If all measurements are valid, display the bodyfat percentage and update it in the database
                        self.invalidneck = False
                        self.invalidwaist = False
                        self.ids.validnhw.text = ""
                        if self.invalidneck == False and self.invalidwaist == False:
                            self.ids.bf_rectangle.opacity= 1
                            self.ids.no_measurement.pos_hint = {"center_y" : 2, "center_x" : 0.365}
                            self.ids.estimate_text.opacity= 1
                            self.ids.bf_continue.pos_hint= {'center_x': 0.498,'center_y': 0.14}
                            self.ids.bf_continue.opacity= 1
                            self.ids.bf_continue_text.opacity= 1

                        # Calculate bodyfat percentage using the US Navy Body Fat Formula
                        bodyfat = round(495 / (1.0324 - 0.19077 * math.log10(waist - neck) + 0.15456 *
math.log10(height*100)) - 450, 1)
                        self.ids.bodyfat_percent.text = f"{bodyfat}%"
                        self.bodyfat= bodyfat

                        user_id = self.manager.get_screen('initialpage').user_id
                        user = UserManager.get_user(user_id)
                        user.update_bodyfat(bodyfat)
```



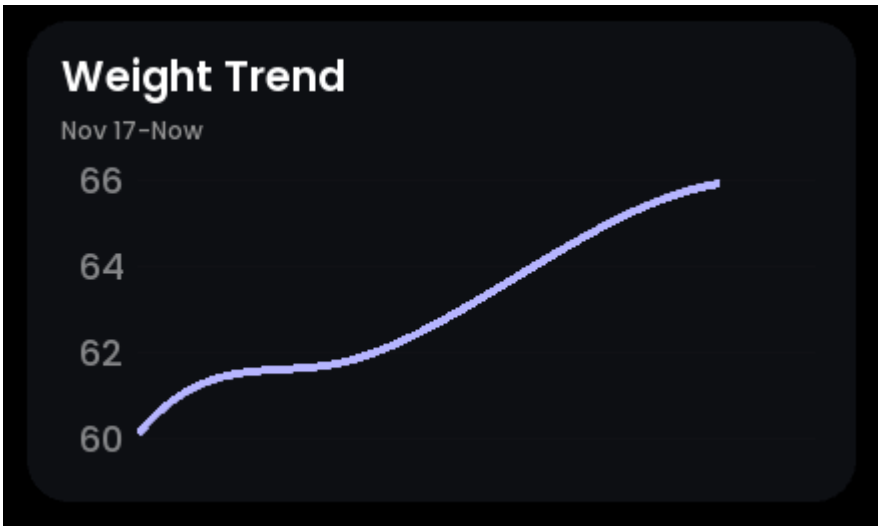




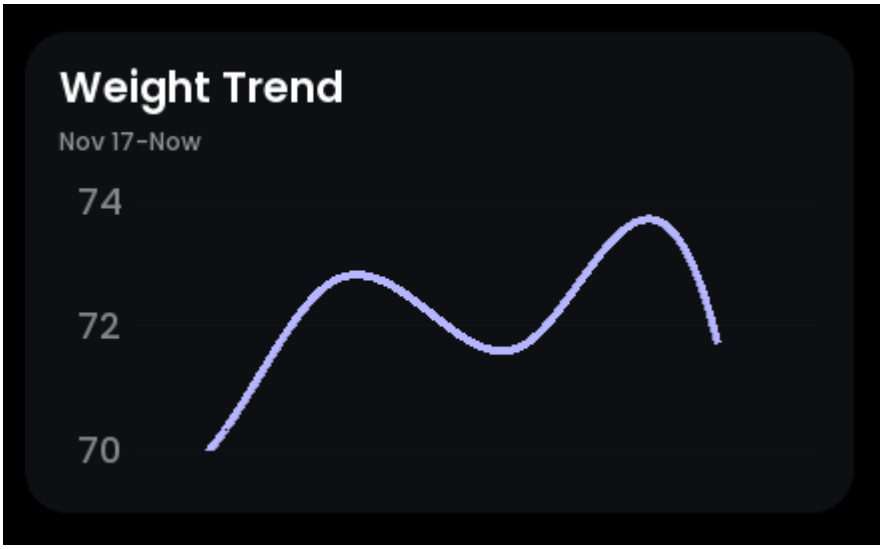
Bodyweight graph (with example data)

```
class BodyweightGraph(BoxLayout):
    highest_weight= NumericProperty(0)
    lowest_weight= NumericProperty(0)
    elements= NumericProperty(0)
    weight_data = [60, 62, 61, 63, 64, 65, 66,] #example data
    highest_weight= max(weight_data)
    lowest_weight= min(weight_data)
```

```
    elements= len(weight_data)

    def __init__(self, **kwargs):
        super(BodyweightGraph, self).__init__(**kwargs)

    def on_kv_post(self, base_widget):
        self.update_graph(self.ids.weight_graph, self.weight_data )
        return super(BodyweightGraph, self).on_kv_post(base_widget)

    def update_graph(self, graph, weight_data):
        graph.background_color = [0, 0, 0, 0]
        self.ids.weight_graph.ymin = min(self.weight_data)
        self.ids.weight_graph.ymax = max(self.weight_data) + .5
        if max(self.weight_data) - min(self.weight_data) < 4:
            self.ids.weight_graph.y_ticks_major = .5
        elif max(self.weight_data) - min(self.weight_data) < 8:
            self.ids.weight_graph.y_ticks_major = 2
        # Interpolating using a spline
        x = np.arange(len(weight_data))
        y = np.array(weight_data)
        spline = UnivariateSpline(x, y, s=1)
        x_smooth = np.linspace(0, len(weight_data) - 1, 300)
        y_smooth = spline(x_smooth)

        plot = SmoothLinePlot(color=[182/255, 181/255, 1, 1]) #smoothen line
        plot.points = [(i, weight) for i, weight in zip(x_smooth, y_smooth)]
        graph.add_plot(plot)


<BodyweightGraph>:
    MDCard:
        orientation: "vertical"
        size_hint: .92, 1
        radius: [21]  # This will give the rounded corners
        padding: "8dp", "40dp", "8dp", "20dp"
        md_bg_color: get_color_from_hex("0D0F13")
        BoxLayout:
            orientation: 'vertical'
            size_hint: .95, 1
            pos_hint: {'center_x': 0.5,'center_y': 0.5}
            Graph:
                id: weight_graph
                x_ticks_minor: 0
                y_grid_label: True
                x_grid_label: True
                label_options: {'color': [1, 1, 1, 0.5], 'bold': True, 'font_name': 'Poppins-Medium.ttf'}
                x_grid: True
                y_grid: True
                xmin: 0
                xmax: root.elements
                ymin: root.lowest_weight
                ymax: root.highest_weight
                border_color: [1, 1, 1, 0]
                tick_color: [1, 1, 1, 0.15
```



(different values)



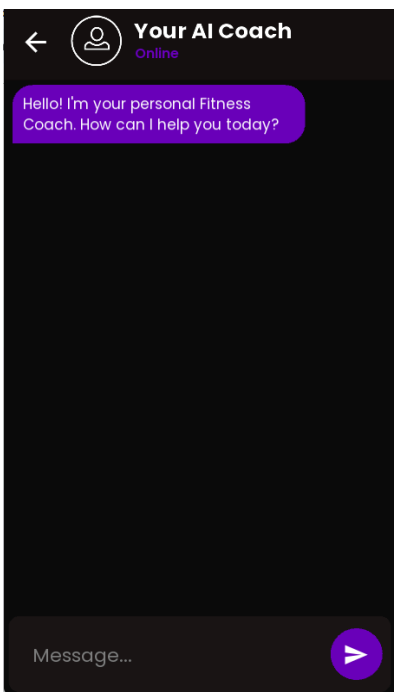
1.4.10 AI-powered Conversational Coach:

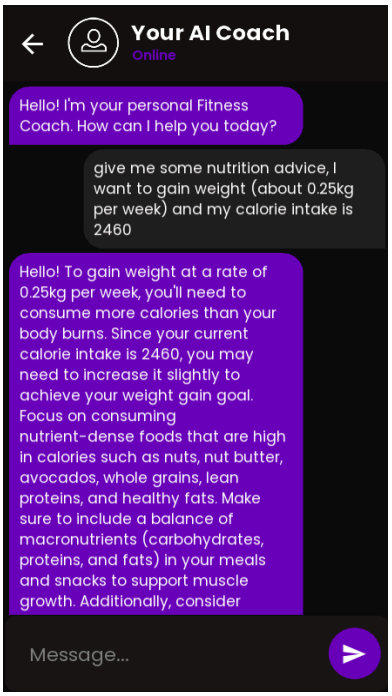
Achieved in ChatBotScreen class:

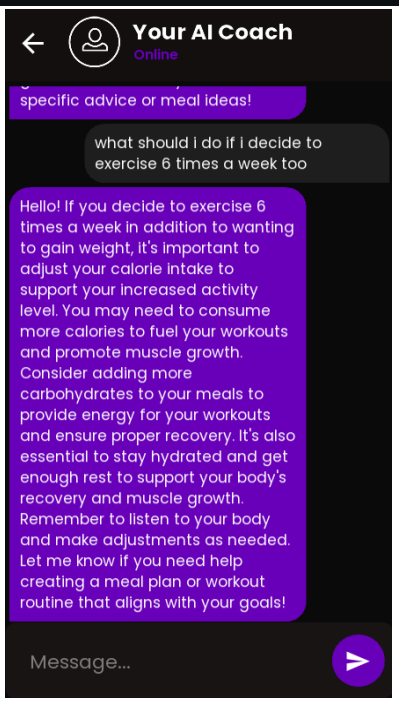
```
class ChatBotScreen(Screen):
    def __init__ (self, **kwargs):
        super(ChatBotScreen, self).__init__(**kwargs)
    def on_kv_post(self, base_widget):
        super().on_kv_post(base_widget)
        self.initial_prompt()

    def initial_prompt(self):
        # Initialize the chatbot with the OpenAI API key and other parameters
        # Initialize the chat history and the initial prompt
        # Get the initial response from the chatbot and add it to the chat UI

        self.chat = ChatOpenAI(
            openai_api_key="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",
            streaming=True,  # Assuming non-streaming for simplicity
            temperature=0.5
        )
        self.chat_history = []
        self.initial_prompt ="Your role is to be a personal Fitness Coach, ready to support me on my health and
fitness journey. As an expert in nutrition and exercise, you're here to offer me the best advice on creating a
balanced diet, mastering exercise techniques, and maintaining a healthy lifestyle. Whether you need guidance on meal
planning, tips for staying motivated, or assistance with workout routines. You need to act enthusiastic, supportive
and as a friend, and you need to be able to answer any questions I have. This is my data:'gender: male, age: 18,
height: 1.75, weight: 65.00, experience_level: Beginner, 6 months to 1.5 years of lifting, bodyfat: 13.40,
activity_level: Moderately Active, exercise 4-5 times a week, goal: Gain Weight, calories: 2566' Answer this with
```

```
just 'Hello! I'm your personal Fitness Coach. How can I help you today?' and nothing else, and then I will ask you
other questions."
        self.memory = ConversationBufferMemory()
        self.memory.chat_memory.add_user_message(self.initial_prompt)
        initial_response = self.get_response(self.initial_prompt)

        self.ids.chat_list.add_widget(Response(text=initial_response, size_hint_x=.75))
        return
    def get_response(self, user_input):
        # Load the chat history
        # Add the user input to the chat memory
        # Get the response from the chatbot and add it to the chat memory
        # Return the full response
        history = self.memory.load_memory_variables({})['history']
        full_input = user_input
        self.memory.chat_memory.add_user_message(user_input)
        full_response = ""
        for response in self.chat.stream([HumanMessage(content=history), HumanMessage(content=full_input)]):
            full_response += response.content.replace('\n', ' ')
        self.memory.chat_memory.add_ai_message(full_response)
        return full_response

    def get_response_thread(self, user_input):
        # Get the full response from the chatbot
        # Add the response to the chat history
        # Schedule the UI update to be run on the main thread
        full_response = self.get_response(user_input)
        self.chat_history.append(('e87601ef-eda4-4e3e-bca5-b6bb32bc483f', full_response, 'AI'))
        Clock.schedule_once(lambda dt: self.update_chat_ui(full_response))
        # Save to database here if desired, or after certain intervals/conditions

    def send(self):
        # Check if the text input is not empty
        if self.ids.text_input != "":
            # Get the text from the input field
            self.value = self.ids.text_input.text
            # Define a mapping of message length to size
            size_map = {range(0, 3): .15, range(3, 6): .22, range(6, 9): .28, range(9, 12): .34,
                        range(12, 15): .40, range(15, 18): .46, range(18, 21): .52, range(21, 24): .58,
                        range(24, 27): .64, range(27, 30): .70, range(30, 1000000): .77}
            # Determine the size of the message based on its length
            self.size = (next((size for range_, size in size_map.items() if len(self.value) in range_), .77),
self.size[1])
            # Determine the alignment of the message based on its length
            self.halign = "center" if len(self.value) < 30 else "left"
            # Get the status label
            self.status_label = self.ids.status
            # Add the message to the chat UI
            self.ids.chat_list.add_widget(Command(text=self.value, size_hint_x=self.size[0], halign=self.halign))
            # Update the status label to "Typing..."
            self.status_label.text = "Typing..."
            # Start a new thread to get the response from the chatbot
            threading.Thread(target=self.get_response_thread, args=(self.value,)).start()
            # Clear the text input
            self.ids.text_input.text = ""
            # Add the message to the chat history
            self.chat_history.append(('e87601ef-eda4-4e3e-bca5-b6bb32bc483f', self.value, 'Human'))

    def rebuild_chat_history(self):
        # Get the user ID
        user_id = 'e87601ef-eda4-4e3e-bca5-b6bb32bc483f'  # Or dynamically set this
        # Get the chat messages from the database
        chat_messages = db_manager.get_chats_by_user_id(user_id)
        # Iterate over the chat messages
        for message_tuple in chat_messages:  # Iterate in the original order
            # Unpack the tuple
            chat_id, user_id, text, timestamp, sender = message_tuple
            # Depending on the sender, use different UI elements
            if sender == 'AI':
                # Display the message as an AI response
                self.ids.chat_list.add_widget(Response(text=text, size_hint_x=.75))
            else:
                # Display the message as a user command
```

```
            self.ids.chat_list.add_widget(Command(text=text, size_hint_x=.75))

    def update_chat_ui(self, response_content):
        # Add the response to the chat UI
        self.ids.chat_list.add_widget(Response(text=response_content, size_hint_x=.75))
        # Update the status back to "Online"
        self.status_label.text = "Online"

    def save_to_database(self):
        # Iterate over the chat history
        for message in self.chat_history:
            # Unpack the message
            user_id, text, sender = message

            # Save the message to the database
            db_manager.insert_chat(user_id=user_id, message=text, sender=sender)

        # Clear the history after saving
        self.chat_history.clear()
```

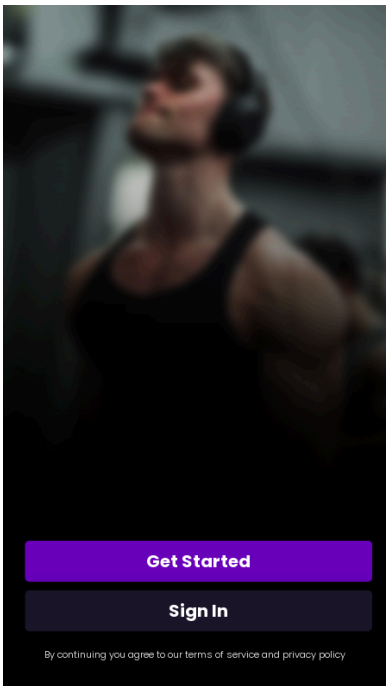


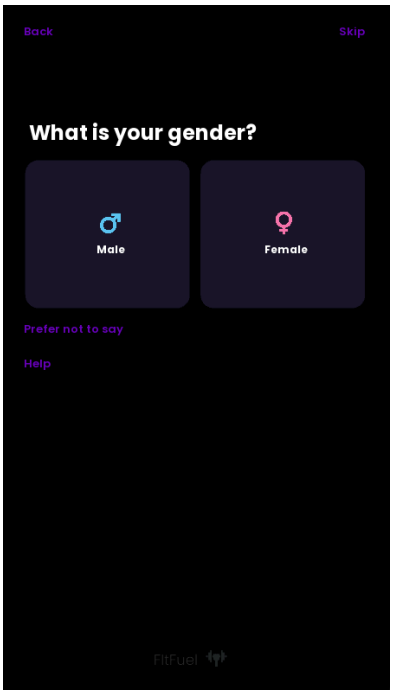


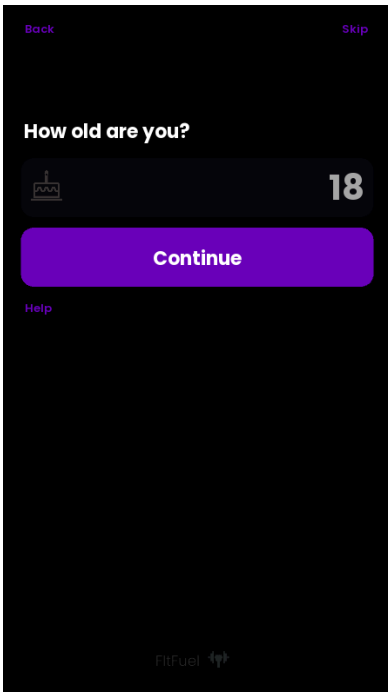


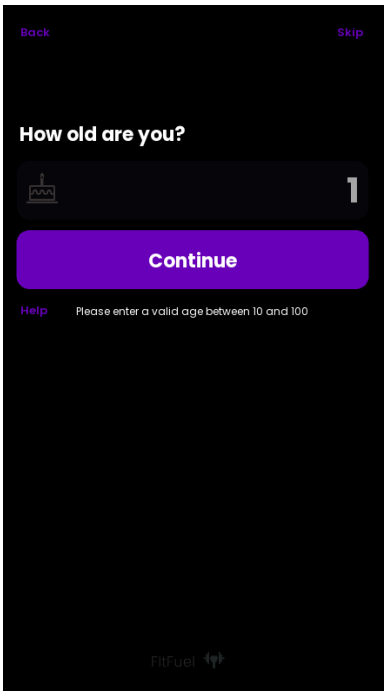
1.4.8 Interactive & User friendly interface

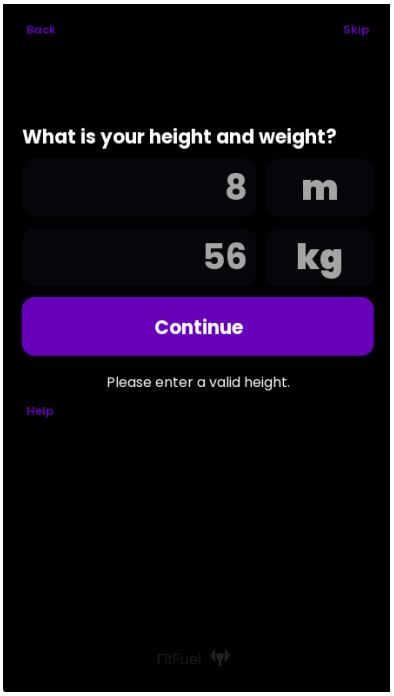
I believe that I achieved it throughout the project.

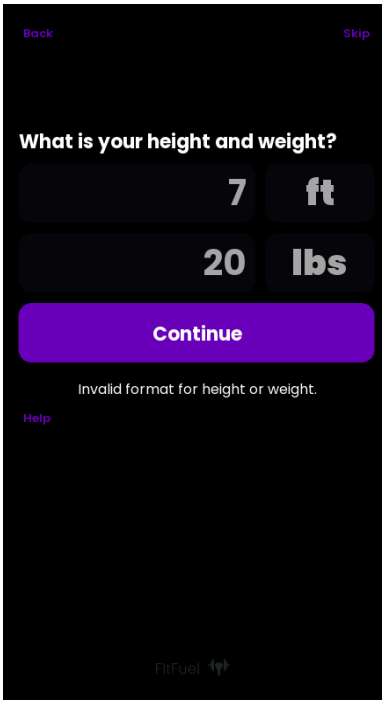

Get Started
Sign In
By continuing you agree to our terms of service and privacy policy

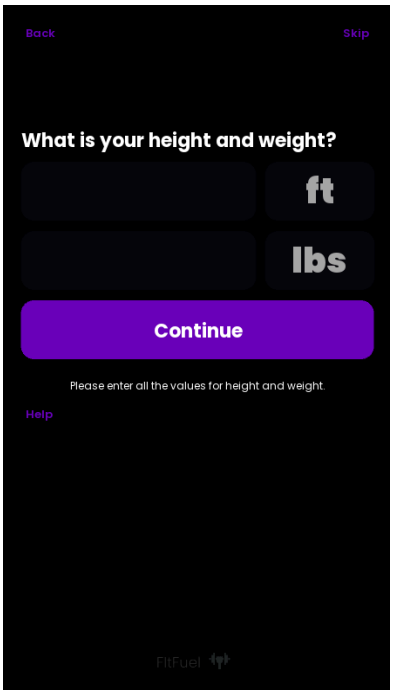

Back
Skip
What is your gender?
Male
Female
Prefer not to say
Help
FitFuel

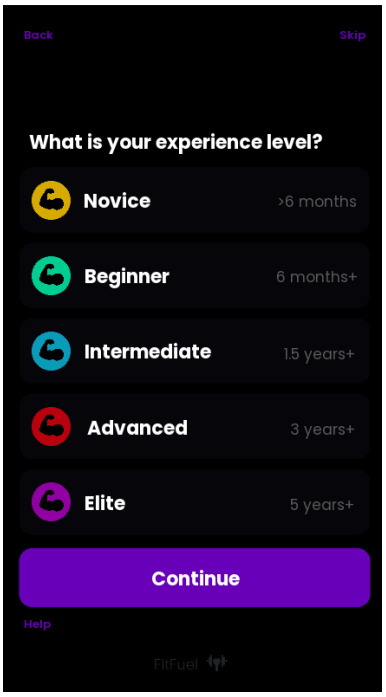

Back
Skip
How old are you?
18
Continue
Help
FitFuel

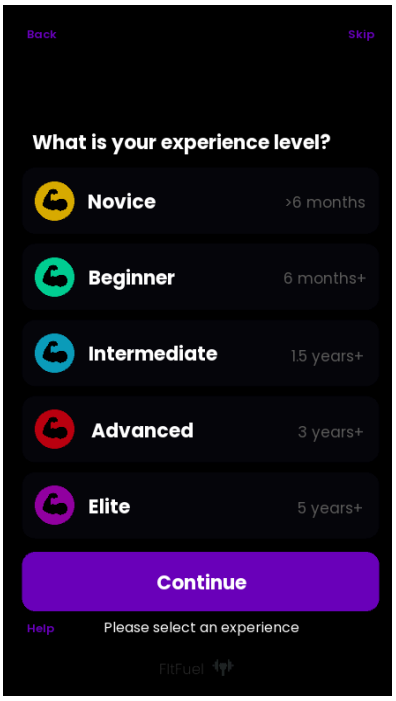

Back
Skip
How old are you?
1
Continue
Help
Please enter a valid age between 10 and 100
FitFuel

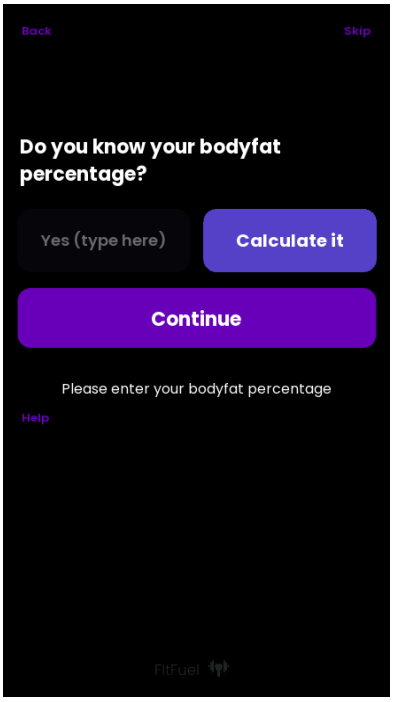

Back
Skip
What is your height and weight?
8
m
56
kg
Continue
Please enter a valid height.
Help
FitFuel

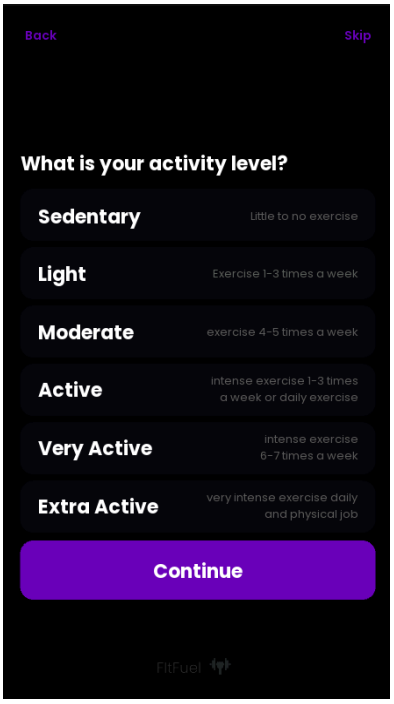

Back
Skip
What is your height and weight?
7
ft
20
lbs
Continue
Invalid format for height or weight.
Help
FitFuel

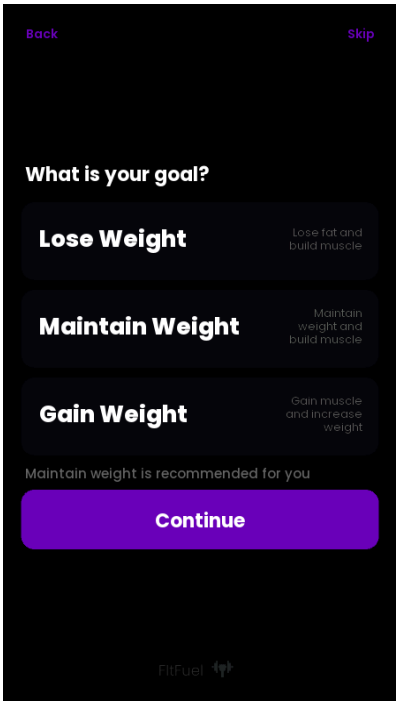

Back
Skip
What is your height and weight?
ft
lbs
Continue
Please enter all the values for height and weight.
Help
FitFuel

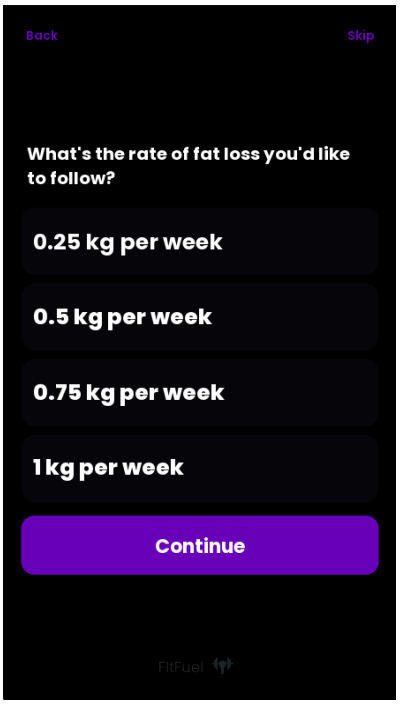

Back
Skip
What is your experience level?
Novice >6 months
Beginner 6 months+
Intermediate 1.5 years+
Advanced 3 years+
Elite 5 years+
Continue
Help
FitFuel

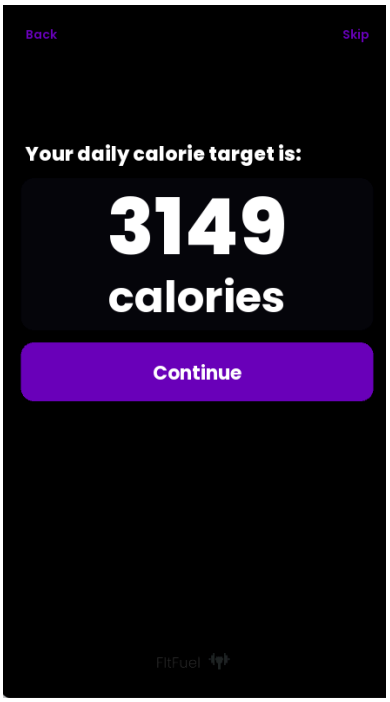

Back
Skip
What is your experience level?
Novice >6 months
Beginner 6 months+
Intermediate 1.5 years+
Advanced 3 years+
Elite 5 years+
Continue
Help
Please select an experience
FitFuel

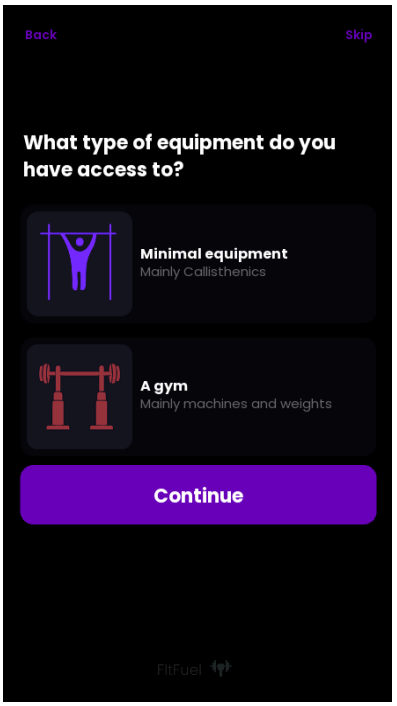

Back
Skip
Do you know your bodyfat percentage?
Yes (type here)
Calculate it
Continue
Please enter your bodyfat percentage
Help
FitFuel

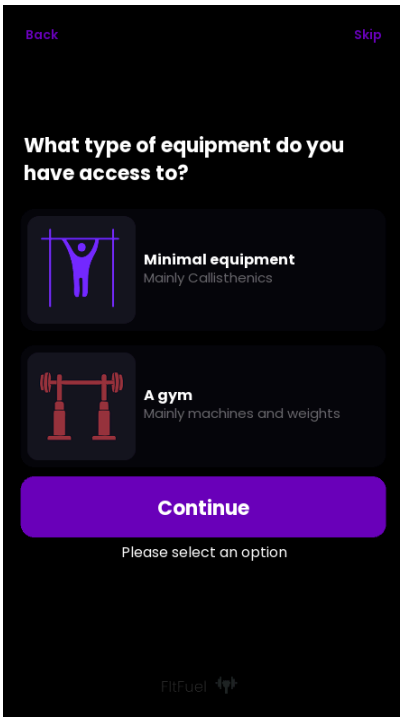

Back
Skip
What is your activity level?
Sedentary Little to no exercise
Light Exercise 1-3 times a week
Moderate exercise 4-5 times a week
Active intense exercise 1-3 times a week or daily exercise
Very Active intense exercise 6-7 times a week
Extra Active very intense exercise daily and physical job
Continue
FitFuel


Back
Skip
What is your goal?
Lose Weight Lose fat and build muscle
Maintain Weight Maintain weight and build muscle
Gain Weight Gain muscle and increase weight
Maintain weight is recommended for you
Continue
FitFuel

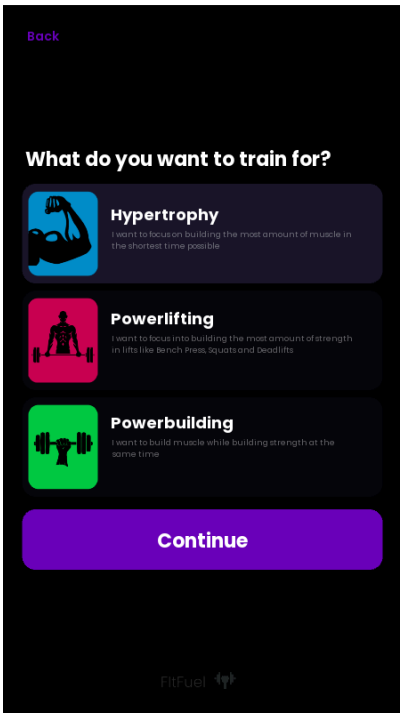

Back
Skip
What's the rate of fat loss you'd like to follow?
0.25 kg per week
0.5 kg per week
0.75 kg per week
1 kg per week
Continue
FitFuel

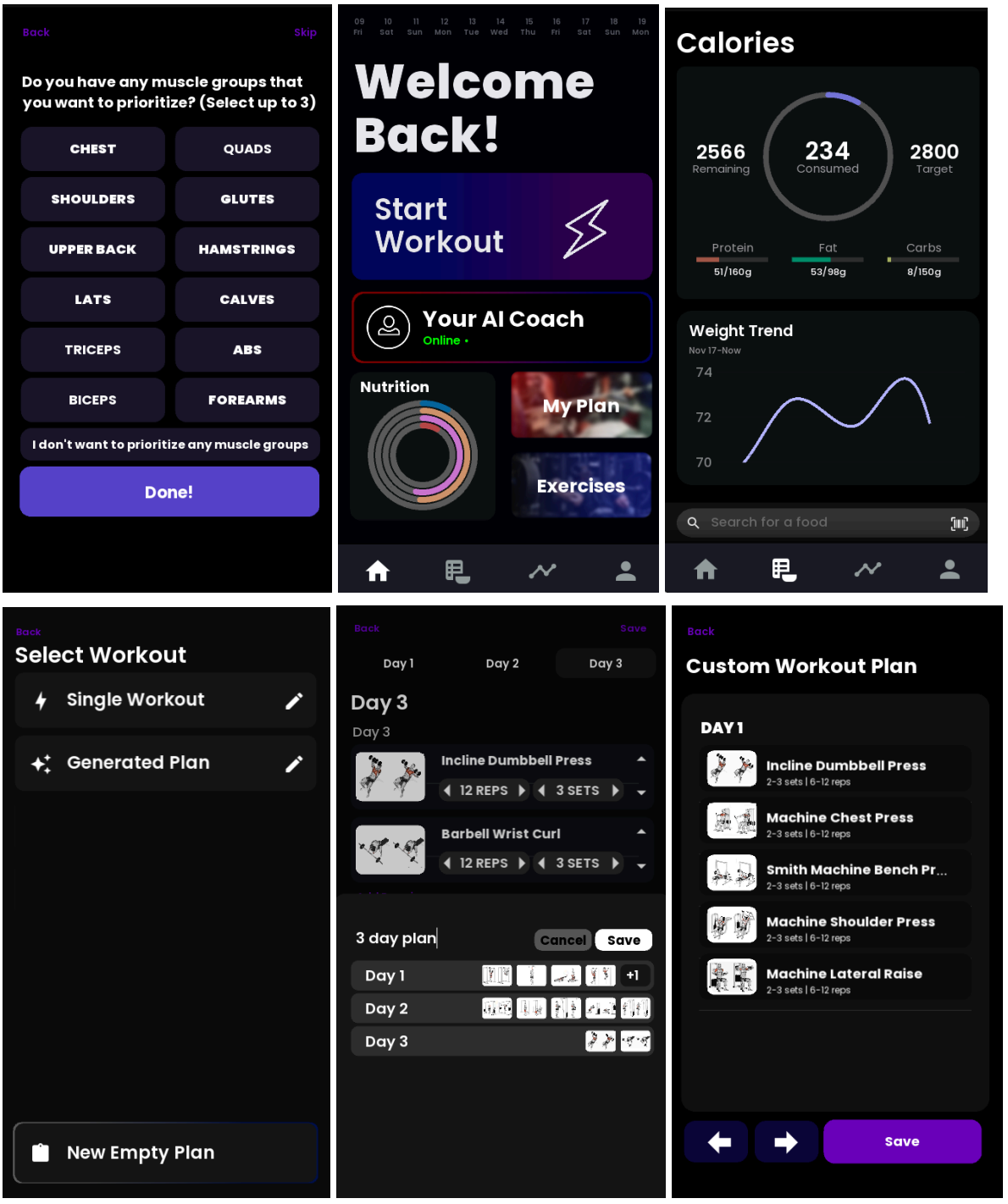

Back
Skip
Your daily calorie target is:
3149
calories
Continue
FitFuel

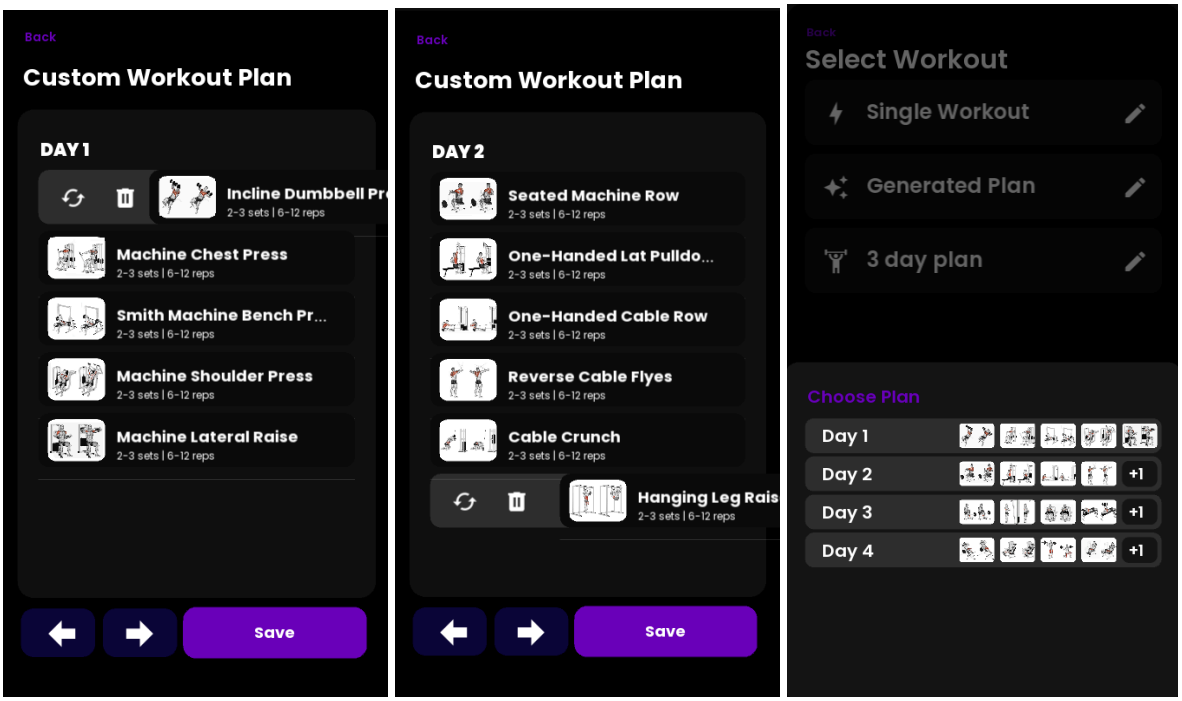

Back
Skip
What type of equipment do you have access to?
Minimal equipment
Mainly Callisthenics
A gym
Mainly machines and weights
Continue
FitFuel


Back
Skip
What type of equipment do you have access to?
Minimal equipment
Mainly Callisthenics
A gym
Mainly machines and weights
Continue
Please select an option
FitFuel


Back
Skip
How many days per week do you want to train?
1x week Not Recommended
2x week Time Efficient
3x week Good
4x week Recommended
5x week Optimal
6x week Optimal
7x week Time Consuming
Continue
FitFuel


Back
What do you want to train for?
Hypertrophy
I want to focus on building the most amount of muscle in the shortest time possible
Powerlifting
I want to focus into building the most amount of strength in lifts like Bench Press, Squats and Deadlifts
Powerbuilding
I want to build muscle while building strength at the same time
Continue
FitFuel


Back
Skip
Do you have any muscle groups that you want to prioritize? (Select up to 3)
CHEST
QUADS
SHOULDERS
GLUTES
UPPER BACK
HAMSTRINGS
LATS
CALVES
TRICEPS
ABS
BICEPS
FOREARMS
I don't want to prioritize any muscle groups
Done!
Welcome Back!
Start Workout
Your AI Coach
Online
Nutrition
My Plan
Exercises
Calories
2566
Remaining
234
Consumed
2800
Target
Protein
51/160g
Fat
53/98g
Carbs
8/150g
Weight Trend
Nov 17-Now
74
72
70
Search for a food
Back
Select Workout
Single Workout
Generated Plan
New Empty Plan
Back
Save
Day 1
Day 2
Day 3
Day 3
Day 3
Incline Dumbbell Press
12 REPS
3 SETS
Barbell Wrist Curl
12 REPS
3 SETS
3 day plan
Cancel
Save
Day 1
+1
Day 2
Day 3
Back
Custom Workout Plan
DAY 1
Incline Dumbbell Press
2-3 sets | 6-12 reps
Machine Chest Press
2-3 sets | 6-12 reps
Smith Machine Bench Pr...
2-3 sets | 6-12 reps
Machine Shoulder Press
2-3 sets | 6-12 reps
Machine Lateral Raise
2-3 sets | 6-12 reps
Save



3.4: Techniques Used:

Technique	Location
Hash Map (Data structure)	GeneratePlan() class, attribute muscle_division, attribute detailed_plan, attribute exercises_per_day, attribute reordered_plan in reorder_plan_based_on_priority() method,exercise_map in parse_tree_to_exercises() method, workout_plan, used_exercises in generate_workout_plan() method etc. FoodSearch() class, attributes unit_to_sequence(), current_unpacked_data(), attribute food_data in rebuild_food_list() method,attribute unpacked_data in unpack_food_data() method, attribute units in selected_unit() method. SelectWorkout() class, attributes workout_plans, plan EmptyPlan() class, attribute workout_plans_by_day Logworkout() class, attributes current_row,

	workout_rows_instances, workout_data exercise_techniques in exercise_guide.py file, exercises in exercises.py file, tree in flattened_tree.py
Trees (Data structure)	GeneratePlan() class. method parse_tree_to_exercises()
Hashing	SignupPage() class, signup() method, LoginPage() class, login() method
Complex data model with many related tables	DatabaseManager() class
Cross-table Parameterized SQL Queries	DatabaseManager class, execute_query method and other methods performing JOIN operations
Aggregate SQL Functions	DatabaseManager class, insert_logged_foods method
Merge Sort Algorithm	GeneratePlan() class, merge_sort() method
Calling Parameterized Web Service APIs	FoodSearch() class, get_food_data() method, ChatBotScreen() class
Regular Expressions	SignupPage class, signup and validate_password_complexity methods, EmptyPlan class, update_sets_and_reps method

3.4.1 Hash Maps

Muscle Categorization and Workout Planning

I used a dictionary (muscle_division) to group muscles ("chest", "upper back", etc.), streamlining workout plan generation by targeting specific areas.
detailed_plan, exercises_per_day, and reordered_plan dictionaries structured my plans, provided daily workout guidance, and allowed customization based on user priorities.

Exercise Selection and Recommendation

My exercise_map was a nested dictionary linking muscle groups to exercises (and ratings). This was key for suggesting optimal exercises to build the workout_plan.
The used_exercises dictionary prevented repetition, promoting workout variety.

Food Tracking and User Interface

I created the unit_to_sequence() function to map units to boolean sequences, for dynamic unit conversion UIs.
Attributes like current_unpacked_data, food_data, unpacked_data, and units tracked food details and user selections, keeping the food information display accurate and responsive.

Workout Management and User Guidance
In various classes, dictionaries like workout_plans, plan, workout_plans_by_day, current_row, workout_rows_instance, and workout_data manage workout options, details, and in-progress workout tracking for seamless user experiences.

Other:
The exercise_techniques dictionary gave detailed exercise instructions, for safe and effective training.
An exercises list acted as a comprehensive exercise repository for the application.
The tree structure organised my workout plans hierarchically.
character_widths facilitated text sizing and alignment for a polished user interface.

How I created the hash maps:

(Dictionary to store exercise names and details)

Started with an excel sheet with a list of exercises, I wrote the targeted muscles and specific targeted division, then gave a rating of the exercises on a scale from 1 to 10 in terms of how good the exercise is based on several factors like stability, overloading capacity etc.. :

name	type	primary_muscle	secondary_muscles	bias	hypertrophy_scale
Machine Chest Press	Machines	chest	front delts, triceps	middle chest	9.5
Machine Shoulder Press	Machines	front delts	traps, side delts	front delts	9.5
Machine Lateral Raise	Machines	side delts	traps	side delts	9.5
Seated Smith Machine Shoulder Press	Machines	front delts	upper back, triceps	front delts	9.2
Reverse Cable Flyes	Cable	rear delts	upper back, triceps	rear delts	9
Cable Lateral Raise	Cable	side delts	traps, front delts	side delts	9
Tricep Pushdown With Rope	Cable	triceps		long head triceps	9
Seated Leg Curl	Machines	hamstrings		bicep femoris	9
Leg Extension	Machines	quadriceps		rectus femoris	9
Hack Squat Machine	Machines	quadriceps, glutes	adductors	vastus medialis and lateralis, gluteus maximus	9
One-Handed Cable Row	Machines, Cable	lats	biceps, forearms, upper back, rear delts	lumbar lats	9
Tricep Pushdown With Bar	Cable	triceps	front delts	long head triceps	8.7
Dumbbell Romanian Deadlift	Free Weights	glutes, hamstrings	forearms, erectors	gluteus maximus, bicep femoris	8.7
Machine Bicep Curl	Machines	biceps	forearms	bicep brachii	8.7
Lat Pulldown With Pronated Grip	Machines	lats	biceps, forearms, upper back, rear delts	lower lats, lumbar lats	8.7
Belt Squat	Machines	quadriceps, glutes	adductors	vastus medialis and lateralis, gluteus maximus	8.7
Cable Curl With Bar	Cable	biceps	forearms	bicep brachii	8.6
Stiff-Legged Deadlift	Free Weights	glutes, hamstrings	erectors, forearms, upper back, lats	gluteus maximus, bicep femoris	8.6
Dumbbell Curl	Free Weights	biceps	forearms	bicep brachii	8.6
Hip Thrust	Free Weights	glutes	hamstrings	gluteus maximus	8.6
Pec Deck	Machine	chest	front delts, triceps	middle chest	8.6
Leg Press	Machines	quadriceps, glutes	adductors	vastus medialis and lateralis, gluteus maximus	8.6
Cable Curl With Rope	Machines, Cable	biceps	forearms	brachialis, brachioradialis	8.6
Hanging Leg Raise	Bodyweight	abs	obliques	lower abs	8.5
Sissy Squat	Bodyweight	quadriceps		rectus femoris	8.5
Cable Crunch	Cable	abs	obliques	upper abs	8.5

I used the following code in python to transform this into a dictionary:

```python
import pandas as pd

# Function to create the icon filename from the exercise name
def create_icon_name(name):
    return name.lower().replace(' ', '') + '.png'

# Load the Excel file
df = pd.read_excel('exercise_details.xlsx')

# Add an 'icon' column to the DataFrame
df['icon'] = df['name'].apply(create_icon_name)

# Convert the DataFrame to a list of dictionaries
exercises = df.to_dict('records')

# Print the result to verify
for exercise in exercises:
    print(exercise)


#output (part of it):
exercises = [{'name': 'Hanging Leg Raise', 'type': 'Bodyweight', 'icon': 'hanginglegraise.png', 'primary_muscle': 'abs', 'secondary_muscles':
'obliques', 'bias': 'Lower abs', 'hypertrophy_scale': 8.5}, {'name': 'Bar Dip', 'type': 'Bodyweight', 'icon': 'bardip.png', 'primary_muscle':
'chest,triceps', 'secondary_muscles': 'front delts', 'bias': 'Lower chest, side and medial head', 'hypertrophy_scale': 8.0}, {'name': 'One-Handed Bar
Hang', 'type': 'Bodyweight', 'icon': 'one-handedbarhang.png', 'primary_muscle': 'forearms', 'secondary_muscles': '', 'bias': 'wrist flexors',
'hypertrophy_scale': 8.0}, {'name': 'Back Extension', 'type': 'Bodyweight', 'icon': 'backextension.png', 'primary_muscle': 'hamstrings, glutes',
'secondary_muscles': 'erectors', 'bias': 'gluteus maximus, bicep femoris', 'hypertrophy_scale': 7.8}, {'name': 'Lying Leg Raise', 'type':
'Bodyweight', 'icon': 'lyinglegraise.png', 'primary_muscle': 'abs', 'secondary_muscles': 'obliques', 'bias': 'lower abs', 'hypertrophy_scale': 7.5},
{'name': 'Pull-Up', 'type': 'Bodyweight', 'icon': 'pull-up.png', 'primary_muscle': 'lats, upper back, rear delts', 'secondary_muscles': 'biceps,
forearms', 'bias': 'lower lats, rear delts, rhomboids', 'hypertrophy_scale': 7.5}, {'name': 'Crunch', 'type': 'Bodyweight', 'icon': 'crunch.png',
'primary_muscle': 'abs', 'secondary_muscles': 'obliques', 'bias': 'upper abs', 'hypertrophy_scale': 7.0}, {'name': 'Plank', 'type': 'Bodyweight',
'icon': 'plank.png', 'primary_muscle': 'abs', 'secondary_muscles': 'obliques', 'bias': 'upper abs, lower abs', 'hypertrophy_scale': 6.5}, {'name':
'Side Plank', 'type': 'Bodyweight', 'icon': 'sideplank.png', 'primary_muscle': 'abs', 'secondary_muscles': 'obliques', 'bias': 'obliques',
'hypertrophy_scale': 6.5}, {'name': 'Chin-Up', 'type': 'Bodyweight', 'icon': 'chin-up.png', 'primary_muscle': 'lats, biceps', 'secondary_muscles':
'upper back', 'bias': 'lumbar lats, lower lats, bicep femoris', 'hypertrophy_scale': 6.5}, {'name': 'Kneeling Plank', 'type': 'Bodyweight', 'icon':
'kneelingplank.png', 'primary_muscle': 'abs', 'secondary_muscles': 'obliques', 'bias': 'obliques', 'hypertrophy_scale': ''}, {'name': 'Kneeling Side
Plank', 'type': 'Bodyweight', 'icon': 'kneelingsideplank.png', 'primary_muscle': 'abs', 'secondary_muscles': 'obliques', 'bias': 'obliques',
'hypertrophy_scale': ''}, {'name': 'Incline Push-Up', 'type': 'Bodyweight', 'icon': 'inclinepush-up.png', 'primary_muscle': 'chest',
'secondary_muscles': 'front delts, triceps', 'bias': 'upper chest', 'hypertrophy_scale': ''}, {'name': 'Kneeling Push-Up', 'type': 'Bodyweight',
'icon': 'kneelingpush-up.png', 'primary_muscle': 'chest', 'secondary_muscles': 'front delts, triceps', 'bias': 'middle chest', 'hypertrophy_scale':
''}, {'name': 'Push-Up', 'type': 'Bodyweight', 'icon': 'push-up.png', 'primary_muscle': 'chest', 'secondary_muscles': 'front delts, triceps', 'bias':
'middle chest', 'hypertrophy_scale': ''}, {'name': 'Glute Bridge', 'type': 'Bodyweight', 'icon': 'glutebridge.png', 'primary_muscle': 'glutes',
'secondary_muscles': 'hamstrings', 'bias': 'gluteus maximus', 'hypertrophy_scale': ''}, {'name': 'Bodyweight Squat', 'type': 'Bodyweight', 'icon':
'bodyweightsquat.png', 'primary_muscle': 'quadriceps, glutes', 'secondary_muscles': 'adductors', 'bias': 'vastus medialis and lateralis, gluteus
maximus', 'hypertrophy_scale': ''}, {'name': 'BodyWeight Lunge', 'type': 'Bodyweight', 'icon': 'bodyweightlunge.png', 'primary_muscle': 'quadriceps,
glutes', 'secondary_muscles': '', 'bias': 'gluteus medius, rectus femoris', 'hypertrophy_scale': ''}, {'name': 'Bench Dip', 'type': 'Bodyweight',
'icon': 'benchdip.png', 'primary_muscle': 'triceps', 'secondary_muscles': 'front delts, chest', 'bias': 'side and medial head', 'hypertrophy_scale':
''}, {'name': 'Close-Grip Push-Up', 'type': 'Bodyweight', 'icon': 'close-grippush-up.png', 'primary_muscle': 'triceps', 'secondary_muscles': 'chest,
front delts', 'bias': 'side and medial head', 'hypertrophy_scale': ''}, {'name': 'Tricep Bodyweight Extension', 'type': 'Bodyweight', 'icon':
'tricepbodyweightextension.png', 'primary_muscle': 'triceps', 'secondary_muscles': '', 'bias': 'side and medial head', 'hypertrophy_scale': ''},
{'name': 'Inverted Row', 'type': 'Bodyweight', 'icon': 'invertedrow.png', 'primary_muscle': 'upper back, lats, rear delts', 'secondary_muscles':
'biceps, forearms', 'bias': 'lumbar lats, mid traps, rhomboids, rear delts', 'hypertrophy_scale': ''}, {'name': 'Decline Push-up', 'type':
'Bodyweight', 'icon': 'declinepush-up.png', 'primary_muscle': 'chest, front delts', 'secondary_muscles': 'triceps', 'bias': '', 'hypertrophy_scale':
''}, {'name': 'Assisted one Leg squat', 'type': 'Bodyweight', 'icon': 'assistedonelegsquat.png', 'primary_muscle': 'quadriceps, glutes',
'secondary_muscles': '', 'bias': '', 'hypertrophy_scale': ''}, {'name': 'Pistol Squat', 'type': 'Bodyweight', 'icon': 'pistolsquat.png',
'primary_muscle': 'quadriceps, glutes', 'secondary_muscles': '', 'bias': '', 'hypertrophy_scale': ''}, {'name': 'Sissy Squat', 'type': 'Bodyweight',
'icon': 'sissysquat.png', 'primary_muscle': 'quadriceps', 'secondary_muscles': '', 'bias': 'rectus femoris', 'hypertrophy_scale': 8.5}, {'name':
'Reverse Cable Flyes', 'type': 'Cable', 'icon': 'reversecableflyes.png', 'primary_muscle': 'rear delts', 'secondary_muscles': 'upper back, triceps',
'bias': 'rear delts', 'hypertrophy_scale': 9.0}, {'name': 'Cable Lateral Raise', 'type': 'Cable', 'icon': 'cablelateralraise.png', 'primary_muscle':
'side delts', 'secondary_muscles': 'traps, front delts', 'bias': 'side delts', 'hypertrophy_scale': 9.0}, {'name': 'Tricep Pushdown With Rope',
'type': 'Cable', 'icon': 'triceppushdownwithrope.png', 'primary_muscle': 'triceps', 'secondary_muscles': '', 'bias': 'long head',
'hypertrophy_scale': 9.0}, {'name': 'Tricep Pushdown With Bar', 'type': 'Cable', 'icon': 'triceppushdownwithbar.png', 'primary_muscle': 'triceps',
'secondary_muscles': 'front delts', 'bias': 'long head', 'hypertrophy_scale': 8.7}, {'name': 'Cable Crunch', 'type': 'Cable', 'icon':
'cablecrunch.png', 'primary_muscle': 'abs', 'secondary_muscles': 'obliques', 'bias': 'upper abs', 'hypertrophy_scale': 8.5}, {'name': 'Cable Curl
With Bar', 'type': 'Cable', 'icon': 'cablecurlwithbar.png', 'primary_muscle': 'biceps', 'secondary_muscles': 'forearms', 'bias': 'bicep brachii',
'hypertrophy_scale': 8.6}, {'name': 'Overhead Cable Triceps Extension', 'type': 'Cable', 'icon': 'overheadcabletricepsextension.png',
'primary_muscle': 'triceps', 'secondary_muscles': '', 'bias': 'side and medial head', 'hypertrophy_scale': 8.4}, {'name': 'Cable Wide Grip Seated
Row', 'type': 'Cable', 'icon': 'cablewidegripseatedrow.png', 'primary_muscle': 'upper back, rear delts', 'secondary_muscles': 'biceps, forearms,
Lats', 'bias': 'mid traps, rhomboids, rear delts', 'hypertrophy_scale': 7.8}, {'name': 'High to Low Cable Chest Fly', 'type': 'Cable', 'icon':
'hightolowcablechestfly.png', 'primary_muscle': 'chest', 'secondary_muscles': 'front delts, biceps', 'bias': 'lower chest', 'hypertrophy_scale':
7.5}, {'name': 'Mid Cable Chest Fly', 'type': 'Cable', 'icon': 'midcablechestfly.png', 'primary_muscle': 'chest', 'secondary_muscles': 'front delts,
biceps', 'bias': 'middle chest', 'hypertrophy_scale': 7.5}, {'name': 'Low to High Cable Chest Fly', 'type': 'Cable', 'icon':
'lowtohighcablechestfly.png', 'primary_muscle': 'chest', 'secondary_muscles': 'front delts, biceps', 'bias': 'upper chest', 'hypertrophy_scale':
7.3}, {'name': 'Cable Close Grip Seated Row', 'type': 'Cable', 'icon': 'cableclosegripseatedrow.png', 'primary_muscle': 'lats,rear delts, upper
back', 'secondary_muscles': 'biceps, rear delts', 'bias': 'mid traps, rhomboids, rear delts, lumbar lats', 'hypertrophy_scale': 6.0}, {'name':
```

```
'Dumbbell Romanian Deadlift', 'type': 'Free Weights', 'icon': 'dumbbellromaniandeadlift.png', 'primary_muscle': 'glutes, hamstrings',
'secondary_muscles': 'forearms, erectors', 'bias': 'gluteus maximus, bicep femoris', 'hypertrophy_scale': 8.7}, {'name': 'Stiff-Legged Deadlift',
'type': 'Free Weights', 'icon': 'stiff-leggeddeadlift.png', 'primary_muscle': 'glutes, hamstrings', 'secondary_muscles': 'erectors, forearms, upper
back, lats', 'bias': 'gluteus maximus, bicep femoris', 'hypertrophy_scale': 8.6}, {'name': 'Dumbbell Curl', 'type': 'Free Weights', 'icon':
'dumbbellcurl.png', 'primary_muscle': 'biceps', 'secondary_muscles': 'forearms', 'bias': 'bicep brachii', 'hypertrophy_scale': 8.6}, {'name':
'Dumbbell Preacher Curl', 'type': 'Free Weights', 'icon': 'dumbbellpreachercurl.png', 'primary_muscle': 'biceps', 'secondary_muscles': 'forearms',
'bias': 'bicep brachii', 'hypertrophy_scale': 8.5},
```

(Dictionary to store exercise details)
Manually creating comprehensive exercise descriptions is incredibly time-consuming. I wanted a faster way to fill my app with quality instructional content. This is why I used LLMs (Large Language Models) to help me do this in a time efficient manner. So I chose ChatOpenAI from the Langchain library because it has great flexibility and the responses are consistent. The prompt outlines the required sections, ensuring the LLM provides well-organised information.

Here's how my code worked:

- Feeding the LLM: My exercises list supplied the names. The loop fed each exercise name into the formatted exercise_prompt and sent it to the LLM.
- Retrieving and Storing: The LLM's response streamed in, and I assembled the exercise_guide dictionary.
- Saving Progress: The export_to_txt function let me save guides as they were generated, preventing data loss.
- Rate Limiting Respect: That time.sleep(21) is key – it avoids overwhelming the OpenAI API, ensuring compliance with their usage limits.
- The Output: LLMs aren't perfect, but they did a good job with the structure I provided.

Here is the code:

```python
from dotenv import load_dotenv
import os
import time
import openai
from langchain.chat_models import ChatOpenAI
from langchain.memory import ConversationBufferMemory
from langchain.schema import HumanMessage
from exercises import exercises
import json
load_dotenv()
openai_api_key = os.getenv("OPENAI_API_KEY")

class ChatBot:
    def __init__(self):
        self.chat = ChatOpenAI(
            openai_api_key=openai_api_key,
            streaming=True,
            temperature=0.5
        )
        self.initial_prompt = "Directly provide a step-by-step guide for {exercise_name}, without any introductory or concluding remarks
such as 'Certainly, here's how you do it'. The guide should strictly include: 1. **Starting Position**: Describe the initial posture and
grip. 2. **Movement Phases**:   - *Concentric Phase*: Detail the muscle engagement and movement path.    - *Eccentric Phase*: Explain
the controlled return to the starting position. 3. **Breathing Technique**: When to inhale and exhale during the exercise. 4. **Common
Mistakes**: Highlight typical errors to avoid. 5. **Safety Tips**: Mention any precautions to prevent injury. 6. **Variations**: Briefly
note any alternative forms of the exercise for different difficulty levels or target muscles. Emphasize that the response should begin
immediately with the 'Starting Position' and conclude with 'Variations', omitting any additional commentary."
        self.memory = ConversationBufferMemory()
        self.memory.chat_memory.add_user_message(self.initial_prompt)
        self.get_responses()

    def get_response(self, user_input):
        response_content = ""
        for response in self.chat.stream([HumanMessage(content=user_input)]):
            response_content += response.content
        return response_content

    def get_responses(self):
        self.exercise_guide = {}
```

```
        for exercise_dict in exercises:
            exercise_name = exercise_dict['name']
            exercise_prompt = self.initial_prompt.replace("{exercise_name}", exercise_name)
            self.memory.chat_memory.add_user_message(exercise_prompt)
            response = self.get_response(exercise_prompt)
            if response is not None:
                self.exercise_guide[exercise_name] = response
                print(f"Successfully generated exercise guide for {exercise_name}")
                self.export_to_txt()  # Export to text file after each successful response
            time.sleep(21)  # Wait for 21 seconds before continuing to the next iteration

    def export_to_txt(self):
        with open('exercise_guide.txt', 'w') as file:
            file.write(json.dumps(self.exercise_guide, indent=4))
# Create a new ChatBot instance
bot = ChatBot()
bot.export_to_txt()

#Output(portion of it)

exercise_technique={
    "Hanging Leg Raise": "Starting Position: Hang from a pull-up bar with your arms fully extended and shoulder-width apart. Keep your legs together and straight, and allow your body to hang freely.\n\nMovement Phases:\nConcentric Phase: Engage your core muscles and slowly raise your legs by flexing your hips and bending your knees. Continue the movement until your thighs are parallel to the ground or slightly higher. Maintain control throughout the entire range of motion.\n\nEccentric Phase: Lower your legs back down to the starting position in a controlled manner. Extend your hips and straighten your knees as you descend, maintaining control and avoiding swinging or momentum.\n\nBreathing Technique: Exhale as you lift your legs, and inhale as you lower them back down.\n\nCommon Mistakes:\n1. Using momentum: Avoid swinging or using momentum to lift your legs. Focus on using your core muscles to control the movement.\n2. Arching the back: Keep your back straight and avoid arching or excessively curving your spine during the exercise.\n3. Raising legs too high: Do not lift your legs above parallel to the ground, as this can strain your lower back and reduce the effectiveness of the exercise.\n\nSafety Tips:\n1. Start with a comfortable range of motion and gradually increase it as your strength improves.\n2. If you experience any pain or discomfort in your lower back, modify the exercise or consult a fitness professional.\n3. Avoid jerky or uncontrolled movements to prevent injury.\n\nVariations:\n1. Bent knee raise: Perform the exercise with your knees bent instead of keeping your legs straight, which can make it easier for beginners.\n2. Weighted leg raise: Hold a dumbbell or ankle weights between your feet to increase the resistance and intensity of the exercise.\n3. Hanging knee tucks: Instead of raising your legs straight, bend your knees and bring them towards your chest, targeting your lower abs.",
    "Bar Dip": "Starting Position: Stand between parallel bars with your arms fully extended and your hands gripping the bars. Your palms should be facing inward, and your hands should be slightly wider than shoulder-width apart.\n\nMovement Phases:\n- Concentric Phase: Lower your body by bending your elbows and leaning forward slightly. Keep your torso upright and your legs straight as you descend. Continue until your shoulders are below your elbows.\n- Eccentric Phase: Push yourself back up by straightening your arms. Maintain control and avoid swinging or jerking motions. Return to the starting position with your arms fully extended.\n\nBreathing Technique: Inhale as you lower your body during the concentric phase. Exhale as you push yourself back up during the eccentric phase.\n\nCommon Mistakes:\n1. Using momentum: Avoid using your legs or swinging your body to lift yourself up. Focus on using your upper body strength.\n2. Rounded shoulders: Keep your shoulders pulled back and down throughout the exercise to maintain proper form.\n3. Elbows flaring out: Keep your elbows close to your body to engage the triceps and prevent unnecessary strain on the shoulders.\n\nSafety Tips:\n1. Warm up before performing bar dips to prepare your muscles.\n2. Start with assistance or use resistance bands if you find the exercise too challenging.\n3. Avoid locking your elbows at the top of the movement to prevent joint strain.\n4. If you experience any pain or discomfort, stop the exercise and consult a fitness professional.\n\nVariations:\n1. Assisted Bar Dips: Use a resistance band or an assisted dip machine to provide support as you build strength.\n2. Weighted Bar Dips: Hold a dumbbell or wear a weighted vest to increase the intensity of the exercise.\n3. Parallel Bar Dips: Perform the exercise on parallel bars to target the triceps and chest muscles differently.\n4. L-sit Dips: Extend your legs forward and keep them parallel to the ground throughout the exercise to engage the core muscles.",
```

(Dictionary to set characters width for UI changes)

I built a dictionary called character_widths. I noted down how wide each letter (and some symbols) would be when displayed. I used a mix of manual estimation and a fancy font measuring library (which I commented out).

Calculating Text Width: I wrote a little function called get_text_width. It takes a string of text and does some quick maths:

It looks up each character's width from my map.
It adds all the individual widths together for the total text width.

```
import freetype

FONT_POINTS = 16
DISPLAY_DPI = 96
face = freetype.Face("Poppins-SemiBold.ttf")

face.set_char_size(FONT_POINTS << 6, FONT_POINTS << 6, DISPLAY_DPI, DISPLAY_DPI)

character_widths = {}
for char in "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ -":
    pen_x = 0
    previous_glyph = 0
    glyph_index = face.get_char_index(char)
    if face.has_kerning and previous_glyph and glyph_index:
        delta = face.get_kerning(previous_glyph, glyph_index)
        pen_x += delta.x >> 6
    face.load_glyph(glyph_index, freetype.FT_LOAD_FLAGS['FT_LOAD_RENDER'])
    pen_x += face.glyph.advance.x >> 6
    character_widths[char] = pen_x

# Normalize the widths so they're relative to the width of a space
space_width = character_widths[' ']
```

```
character_widths = {char: width / space_width for char, width in character_widths.items()}
print(character_widths)
```

3.4.2 Trees:

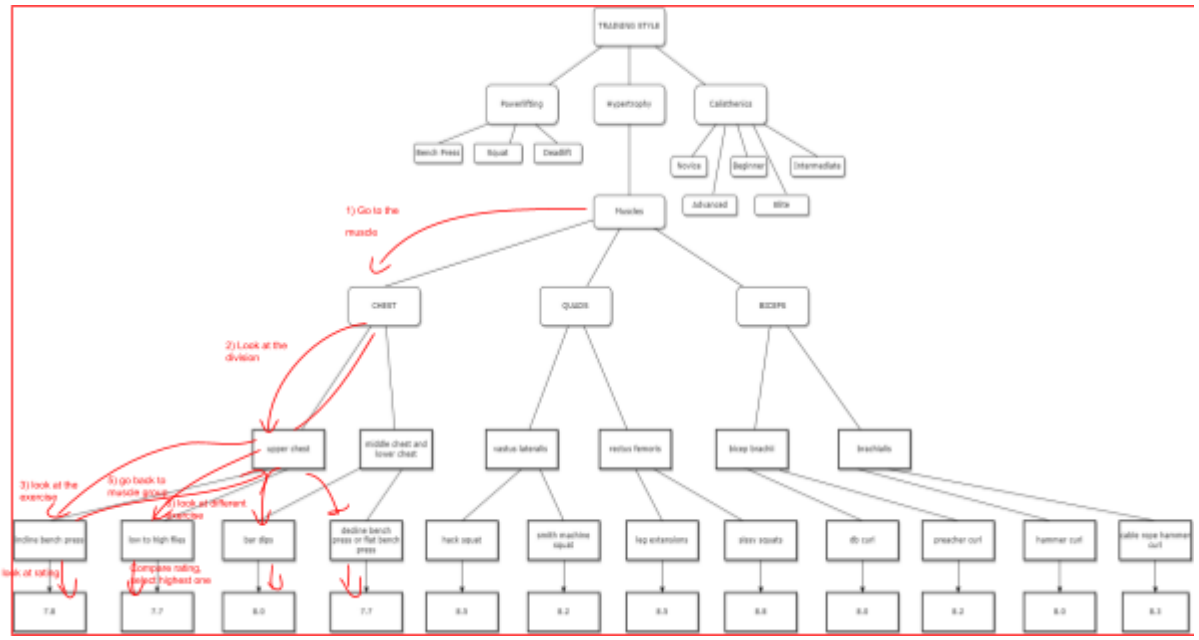
Complex Tree Structure:I constructed a multi-level tree with various nodes representing different aspects of workout plans, such as workout types, experience levels, muscle groups, and specific exercises. This structure is complex and well-suited to the hierarchical nature of the data I am dealing with.
Dynamic Tree Manipulation:I added dynamic manipulation of the tree, such as adding children nodes based on different criteria (e.g., experience levels, muscle groups).
Traversal and Data Extraction:Functions like flatten_tree and print_tree are effectively traversing the tree. This is crucial in a tree data structure to access and utilise the stored data.
The parse_tree_to_exercises function shows further use of tree traversal to extract and organise data for practical application.

How I mapped the Tree:

To generate a workout plan for each user we need to take in consideration different factors such as training frequency, the style of preferred training, the gender of the user, the muscle that the user wants to priotize etc.

In the same session it is recommended to have at least two exercises for the same muscle group with two sets each in order to grow muscle, these exercises have to be slightly different in order for the exercise to not be redundant. For example is not useful to do a flat dumbbell bench press and a flat barbell bench press in the same session as it's basically the same exact movement performed with different types of resistance, a much more suitable pair for the chest would be a flat dumbbell bench press and an incline dumbbell bench press as one targets the lower portion of the pecs and the other targets more the upper portion of the pecs.

Thus, I decided to use a tree data structure to try to solve this problem and find the best plan for each user.



This is part of the type of tree I'm gonna use.
The algorithm is gonna follow the red arrow.

Function to print me a tree for my data:

```
from exercises import exercises
import pydot
import os
class Node:
    def __init__(self, name):
        self.name = name
        self.children = {}

    def add_child(self, name, child):
        self.children[name] = child

    def get_child(self, name):
        return self.children.get(name)

def build_workout_plan_tree():
    # Create root of the tree
    root = Node('workout plan')

    # Level 1 nodes
    calisthenics = Node('calisthenics')
    powerlifting = Node('powerlifting')
    hypertrophy = Node('hypertrophy')

    # Adding level 1 nodes to root
    root.add_child('calisthenics', calisthenics)
    root.add_child('powerlifting', powerlifting)
    root.add_child('hypertrophy', hypertrophy)

    # Calisthenics level 2 nodes
    experience_levels = ['novice', 'beginner', 'intermediate', 'advanced', 'elite']
    for level in experience_levels:
        calisthenics.add_child(level, Node(level))

    # Powerlifting level 2 nodes
    powerlifting_exercises = ['bench press', 'deadlift', 'squats']
```

```
    for exercise in powerlifting_exercises:
        powerlifting.add_child(exercise, Node(exercise))

    # Hypertrophy level 2 node
    muscle_groups = Node('muscle groups')
    hypertrophy.add_child('muscle groups', muscle_groups)

    # Calisthenics level 3 nodes for novice, beginner, intermediate
    for level in experience_levels[:3]:
        calisthenics.children[level].add_child('progression program', Node('progression program'))

    # Advanced and elite levels point to muscle groups
    for level in experience_levels[3:]:
        calisthenics.children[level].children = muscle_groups.children

    # Muscle groups level 4 nodes
    muscles = ['chest', 'upper back', 'lats', 'shoulders', 'quads',
               'hamstrings', 'glutes', 'calves', 'abs', 'triceps',
               'biceps', 'forearms', 'adductors']
    for muscle in muscles:
        muscle_groups.add_child(muscle, Node(muscle))

    # Progression program level 4 nodes
    progression_exercises = ['Pushups', 'Pull ups', 'Squats', 'Leg Raises',
                             'Bridges', 'Skills']
    for exercise in progression_exercises:
        calisthenics.children['novice'].children['progression program'].add_child(exercise, Node(exercise))
        calisthenics.children['beginner'].children['progression program'].add_child(exercise, Node(exercise))
        calisthenics.children['intermediate'].children['progression program'].add_child(exercise, Node(exercise))

    # Define muscle group divisions, for level 5 nodes
    muscle_group_divisions = {
        'chest': ['upper chest', 'middle chest', 'lower chest'],
        'shoulders': ['front delts', 'side delts', 'rear delts'],
        'quads': ['vastus medialis and lateralis', 'rectus femoris'],
        'lats': ['upper lats', 'lumbar lats', 'lower lats'],
        'glutes': ['gluteus medius', 'gluteus maximus'],
        'hamstrings': ['bicep femoris'],
        'calves': ['soleus', 'gastroc'],
        'upper back': ['upper traps','mid traps', 'rhomboids'],
        'triceps': ['long head', 'side and medial head'],
        'biceps': ['bicep brachii', 'brachialis'],
        'abs': ['upper abs', 'lower abs'],
        'forearms': ['brachioradialis', 'wrist extensor', 'wrist flexors'],
        'adductors': ['adductor magnus']
    }

    # Add muscle group divisions to the muscle_groups node
    for muscle, divisions in muscle_group_divisions.items():
        muscle_node = muscle_groups.get_child(muscle) if muscle_groups.get_child(muscle) else Node(muscle)
        muscle_groups.add_child(muscle, muscle_node)
        for division in divisions:
            division_node = Node(division)
            muscle_node.add_child(division, division_node)

    return root

def add_exercises_to_tree(node, exercises):
    for exercise in exercises:
        biases = exercise['bias'].split(', ')
        for bias in biases:
            # Navigate through the tree to find the correct muscle group and bias
            for muscle_group_node in node.get_child('hypertrophy').get_child('muscle groups').children.values():
                division_node = muscle_group_node.get_child(bias)
                if division_node:
                    # Add the exercise to the division node as a new node
                    exercise_node = Node(exercise['name'])
```

```
                    division_node.add_child(exercise['name'], exercise_node)
                    # Now add the hypertrophy scale as a child of this exercise node
                    rating_node = Node(f"{exercise['hypertrophy_scale']}")
                    exercise_node.add_child(f"{exercise['hypertrophy_scale']}", rating_node)


workout_plan_root = build_workout_plan_tree()

# Add exercises to the tree
add_exercises_to_tree(workout_plan_root, exercises)

def print_tree(node, level=0):
    print('|-------->' * level + node.name)
    for child in node.children.values():
        print_tree(child, level + 1)
print_tree(workout_plan_root)


def flatten_tree(node, parent_name=None, level=0, flat_list=[]):
    node_info = {
        'name': node.name,
        'parent': parent_name,
        'level': level,
        'children': [child.name for child in node.children.values()]
    }

    # Include the hypertrophy_scale if it exists on the node
    if hasattr(node, 'hypertrophy_scale'):
        node_info['hypertrophy_scale'] = node.hypertrophy_scale

    flat_list.append(node_info)

    for child in node.children.values():
        flatten_tree(child, node.name, level + 1, flat_list)

    return flat_list


Print the flattened list.
print(flatten_tree)



#Output: (portion of the tree)
|-------->|-------->|-------->|-------->|-------->|-------->7.5
|-------->|-------->|-------->|-------->|-------->|-------->Smith Machine Bench Press
|-------->|-------->|-------->|-------->|-------->|-------->8.5
|-------->|-------->|-------->upper back
|-------->|-------->|-------->|-------->upper traps
|-------->|-------->|-------->|-------->|-------->Dumbbell Shrug
|-------->|-------->|-------->|-------->|-------->|-------->8.0
|-------->|-------->|-------->|-------->|-------->|-------->Barbell Shrug
|-------->|-------->|-------->|-------->|-------->|-------->7.8
|-------->|-------->|-------->|-------->mid traps
|-------->|-------->|-------->|-------->|-------->|-------->Inverted Row
|-------->|-------->|-------->|-------->|-------->|-------->
|-------->|-------->|-------->|-------->|-------->|-------->Cable Wide Grip Seated Row
|-------->|-------->|-------->|-------->|-------->|-------->|-------->7.8
|-------->|-------->|-------->|-------->|-------->|-------->Cable Close Grip Seated Row
|-------->|-------->|-------->|-------->|-------->|-------->|-------->6.0
|-------->|-------->|-------->|-------->|-------->|-------->Barbell Row
|-------->|-------->|-------->|-------->|-------->|-------->|-------->6.5
|-------->|-------->|-------->|-------->|-------->|-------->Seated Machine Row
|-------->|-------->|-------->|-------->|-------->|-------->|-------->8.5
|-------->|-------->|-------->|-------->rhomboids
|-------->|-------->|-------->|-------->|-------->Pull-Up
|-------->|-------->|-------->|-------->|-------->|-------->7.5
|-------->|-------->|-------->|-------->|-------->|-------->Inverted Row
|-------->|-------->|-------->|-------->|-------->|-------->
|-------->|-------->|-------->|-------->|-------->|-------->Cable Wide Grip Seated Row
|-------->|-------->|-------->|-------->|-------->|-------->|-------->7.8
|-------->|-------->|-------->|-------->|-------->|-------->Cable Close Grip Seated Row
|-------->|-------->|-------->|-------->|-------->|-------->|-------->6.0
|-------->|-------->|-------->|-------->|-------->|-------->Barbell Row
|-------->|-------->|-------->|-------->|-------->|-------->|-------->6.5
|-------->|-------->|-------->|-------->|-------->|-------->Seated Machine Row
|-------->|-------->|-------->|-------->|-------->|-------->|-------->8.5
|-------->|-------->|-------->Lats
```

```
|--------|--------|--------|--------|--------upper lats
|--------|--------|--------|--------|--------Dumbbell Row
|--------|--------|--------|--------|--------|--------7.0
|--------|--------|--------|--------|--------One-Handed Lat Pulldown
|--------|--------|--------|--------|--------|--------8.5
|--------|--------|--------|--------|--------Seated Machine Row
|--------|--------|--------|--------|--------|--------8.5
|--------|--------|--------|--------lumbar Lats


#Flattened tree (portion of it):
tree= [{'name': 'workout plan', 'parent': None, 'Level': 0, 'children': ['calisthenics', 'powerlifting', 'hypertrophy']}, {'name': 'calisthenics',
'parent': 'workout plan', 'Level': 1, 'children': ['novice', 'beginner', 'intermediate', 'advanced', 'elite']}, {'name': 'novice', 'parent':
'calisthenics', 'level': 2, 'children': ['progression program']}, {'name': 'progression program', 'parent': 'novice', 'Level': 3, 'children':
['Pushups', 'Pull ups', 'Squats', 'Leg Raises', 'Bridges', 'Skills']}, {'name': 'Pushups', 'parent': 'progression program', 'Level': 4, 'children':
[]}, {'name': 'Pull ups', 'parent': 'progression program', 'Level': 4, 'children': []}, {'name': 'Squats', 'parent': 'progression program', 'Level':
4, 'children': []}, {'name': 'Leg Raises', 'parent': 'progression program', 'Level': 4, 'children': []}, {'name': 'Bridges', 'parent': 'progression
program', 'Level': 4, 'children': []}, {'name': 'Skills', 'parent': 'progression program', 'Level': 4, 'children': []}, {'name': 'beginner',
'parent': 'calisthenics', 'Level': 2, 'children': ['progression program']}, {'name': 'progression program', 'parent': 'beginner', 'Level': 3,
'children': ['Pushups', 'Pull ups', 'Squats', 'Leg Raises', 'Bridges', 'Skills']}, {'name': 'Pushups', 'parent': 'progression program', 'level': 4,
'children': []}, {'name': 'Pull ups', 'parent': 'progression program', 'Level': 4, 'children': []}, {'name': 'Squats', 'parent': 'progression
program', 'Level': 4, 'children': []}, {'name': 'Leg Raises', 'parent': 'progression program', 'Level': 4, 'children': []}, {'name': 'Bridges',
'parent': 'progression program', 'level': 4, 'children': []}, {'name': 'Skills', 'parent': 'progression program', 'Level': 4, 'children': []},
{'name': 'intermediate', 'parent': 'calisthenics', 'Level': 2, 'children': ['progression program']}, {'name': 'progression program', 'parent':
'intermediate', 'level': 3, 'children': ['Pushups', 'Pull ups', 'Squats', 'Leg Raises', 'Bridges', 'Skills']}, {'name': 'Pushups', 'parent':
'progression program', 'level': 4, 'children': []}, {'name': 'Pull ups', 'parent': 'progression program', 'Level': 4, 'children': []}, {'name':
'Squats', 'parent': 'progression program', 'Level': 4, 'children': []}, {'name': 'Leg Raises', 'parent': 'progression program', 'Level': 4,
'children': []}, {'name': 'Bridges', 'parent': 'progression program', 'level': 4, 'children': []}, {'name': 'Skills', 'parent': 'progression
program', 'Level': 4, 'children': []}, {'name': 'advanced', 'parent': 'calisthenics', 'Level': 2, 'children': ['chest', 'upper back', 'lats',
'shoulders', 'quads', 'hamstrings', 'glutes', 'calves', 'abs', 'triceps', 'biceps', 'forearms', 'adductors']}, {'name': 'chest', 'parent':
'advanced', 'level': 3, 'children': ['upper chest', 'middle chest', 'Lower chest']}, {'name': 'upper chest', 'parent': 'chest', 'Level': 4,
'children': ['Incline Push-Up', 'Low to High Cable Chest Fly', 'Incline Dumbbell Press', 'Incline Bench Press', 'Dumbbell Pullover']}, {'name':
'Incline Push-Up', 'parent': 'upper chest', 'Level': 5, 'children': ['']}, {'name': '', 'parent': 'Incline Push-Up', 'Level': 6, 'children': []},
{'name': 'Low to High Cable Chest Fly', 'parent': 'upper chest', 'Level': 5, 'children': ['7.3']}, {'name': '7.3', 'parent': 'Low to High Cable Chest
Fly', 'Level': 6, 'children': []}, {'name': 'Incline Dumbbell Press', 'parent': 'upper chest', 'Level': 5, 'children': ['8.2']}, {'name': '8.2',
'parent': 'Incline Dumbbell Press', 'Level': 6, 'children': []}, {'name': 'Incline Bench Press', 'parent': 'upper chest', 'Level': 5, 'children':
['7.6']}, {'name': '7.6', 'parent': 'Incline Bench Press', 'Level': 6, 'children': []}, {'name': 'Dumbbell Pullover', 'parent': 'upper chest',
'level': 5, 'children': ['6.0']}, {'name': '6.0', 'parent': 'Dumbbell Pullover', 'Level': 6, 'children': []}, {'name': 'middle chest', 'parent':
'chest', 'Level': 4, 'children': ['Kneeling Push-Up', 'Push-Up', 'Mid Cable Chest Fly',...
```

3.4.3 Hashing:

I am using bcrypt to hash user passwords. This is a good practice in handling user authentication as it significantly increases security. Storing hashed passwords instead of plain text ensures that even if my database is compromised, the actual passwords remain protected.

The use of bcrypt.gensalt() generates a unique 'salt' for each password, which adds an additional layer of security by ensuring that the hash output is unique even for identical passwords.

Password Verification:The process of verifying the user's password during login (bcrypt.checkpw) is a key security feature. By comparing the hashed version of the input password with the stored hashed password, I maintain security while validating user credentials.This method is effective in preventing common security vulnerabilities such as brute force or dictionary attacks.

Example:

```
if bcrypt.checkpw(password.encode('utf-8'), hashed_password.encode('utf-8')):
```

3.4.4 Complex Data Model

I designed a complex data model with multiple interlinked tables to represent various aspects of nutritional intake and workout routines. This model included tables for food items, daily totals, workout plans, workout days, and exercises, each with specific attributes to store relevant data. The relationships between these tables were carefully crafted to ensure data integrity and facilitate efficient data retrieval and manipulation. (Look at 3.3.4 Section)

Field	Type	Null	Key
id	char(36)	NO	PRI
username	varchar(255)	YES	
email	varchar(255)	YES	
password	varchar(255)	YES	
gender	varchar(6)	YES	
age	int	YES	
height	decimal(5,2)	YES	
weight	decimal(5,2)	YES	
experience_level	varchar(50)	YES	
bodyfat	decimal(4,2)	YES	
activity_level	varchar(255)	YES	
goal	varchar(100)	YES	
calories	int	YES	
equipment	varchar(255)	YES	
training_style	varchar(100)	YES	
training_frequency	varchar(100)	YES	
timestamp	datetime	YES	
prioritized_muscle_groups	varchar(255)	YES	

Field	Type	Null	Key
user_id	char(36)	YES	MUL
total_calories	int	YES	
total_protein	int	YES	
total_fats	int	YES	
total_carbs	int	YES	
date	date	YES	
daily_target	varchar(20)	YES	

Field	Type	Null	Key
chat_id	int	NO	PRI
user_id	char(36)	YES	MUL
AI	varchar(255)	NO	
timestamp	datetime	YES	
sender	text	YES	

Field	Type	Null	Key
user_id	char(36)	YES	MUL
label	varchar(255)	YES	
kcal	int	YES	
protein	decimal(5,2)	YES	
carbs	decimal(5,2)	YES	
fats	decimal(5,2)	YES	
fiber	decimal(5,2)	YES	
portion_size	decimal(5,2)	YES	
weight	decimal(6,2)	YES	
unit	varchar(10)	YES	
timestamp	datetime	YES	

Field	Type	Null	Key
day_exercise_id	int	NO	PRI
day_id	int	YES	MUL
ExerciseID	int	YES	MUL
sets	int	YES	
reps	int	YES	

Field	Type	Null	Key
ExerciseID	int	NO	PRI
Name	varchar(255)	NO	

Field	Type	Null	Key
plan_id	int	NO	PRI
user_id	char(36)	YES	MUL
plan_name	varchar(255)	YES	

Field	Type	Null	Key
day_id	int	NO	PRI
plan_id	int	YES	MUL
day_number	int	YES	

- chats table is linked to userdata through user_id. Showing that chats are associated with users and a user can have multiple chat entries.
- daily_totals table is also linked to userdata through user_id. This indicates that daily nutritional totals are tracked per user.
- day_exercises table is linked to two tables: workout_days and exercises through ExerciseID. Each day has associated exercises with details like sets and reps.
- exercises table is a standalone reference table which lists exercises by ExerciseID and their names.
- food_items table is linked to userdata through user_id. This table probably stores information about food intake per user.

- ➢ userdata is the central table that likely stores all the user information, and several other tables reference it by user_id.

3.4.5 Cross-table Parameterized SQL Queries

My approach included the use of cross-table parameterized SQL queries to insert and update data across multiple tables based on user interactions. For instance, when logging food items, I inserted data into the food_items table and then used a parameterized query to update the daily_totals table to reflect the new totals for calories, protein, fats, and carbs for the given day. This ensured data consistency and integrity across the system.

- ➢ In update_exercise_plan(), there's a query that conditionally performs either an UPDATE or INSERT based on whether a plan already exists. This involves passing user_id and plan data as parameters.
- ➢ In save_complete_workout_plan(), there's a sequence of operations that interacts with multiple tables (workout_plans, workout_days, day_exercises, exercises). This method performs checks, inserts, and updates across these tables, using user_id, plan_name, and workout_plan as parameters.
- ➢ retrieve_workout_plan() executes a series of queries to reconstruct a user's workout plan from workout_plans, workout_days, and day_exercises tables by joining them on their respective identifiers.

3.4.6 Aggregate SQL Functions

I used aggregate SQL functions to calculate sums and totals. By using functions like SUM(), I was able to aggregate data from multiple records, such as the total calories consumed in a day or the total repetitions performed in a workout session. This allowed for dynamic generation of user insights and progress tracking.

- ➢ In insert_logged_foods(), the query with REPLACE INTO daily_totals uses aggregate functions (SUM()) to calculate daily totals of nutritional values from the food_items table, grouping by user_id and date.

3.4.7 Merge Sort Algorithm:

The function merge_sort divides the array into halves, recursively sorts them, and then merges them back together in sorted order.The ability to sort based on a specified key and the option to reverse the sorting order add flexibility and utility to the function, making it more adaptable to different scenarios in my application.

Application in Exercise Sorting:I used merge sort to organise exercises based on their ratings. This is a practical application of the algorithm, ensuring that exercises are sorted efficiently, which is especially important since the list of exercises is large.

3.4.8 Calling Parameterized Web Service APIs:

I integrated external food database APIs into my application. This feature allows users to search for foods and instantly access comprehensive nutritional breakdowns. To do this I connected the application to a free food database API called Edamam. This allows users to directly search for the foods they've eaten, and the application retrieves detailed nutritional information. The application parses the JSON response from the API to isolate the most pertinent nutritional details. This data is then displayed to the user in a clear format and can be optionally stored for future reference

The get_food_data() function dynamically crafts API request URLs. It securely embeds essential parameters like your application ID, API key, and the user's food query.
Upon receipt, the JSON response from get_food_data() is relayed to the unpack_food_data() function. This function decodes the JSON, pinpoints nutritional metrics (calories, carbs, protein, fats), and rounds the values for user-friendliness.

In the ChatBotScreen class I used the ChatOpenAI object from the Langchain library that acts as the foundation for the application's AI-powered fitness coach. The class establishes a secure link to OpenAI's advanced language models. Functions like chat.stream conveniently package up conversational history, the user's latest input, and any tailored instructions from your initial prompt. The API expertly analyses the provided data and produces text responses that emulate a knowledgeable and supportive fitness coach.

3.4.9 Regular Expressions

In my code, I used regular expressions primarily for validation and parsing purposes. Within the SignupPage class, I apply regular expressions in several ways:

Email Validation: I ensure that the email address entered by the user during the signup process adheres to a standard email format. The regular expression [^@]+@[^@]+\.[^@]+ checks for the presence of an @ symbol and a period in the appropriate positions to validate the structure of an email address.

```
        if not re.match(r"[^@]+@[^@]+\.[^@]+", email):
            self.ids.signup_error.text = "Please enter a valid email address."
            self.successful_signup = False
            return
```

Password Complexity Validation: I verify the complexity of the user's password by using different regular expressions to search for uppercase letters, lowercase letters, numbers, and special characters. This ensures that the password meets the security standards before it is accepted.

```
    def validate_password_complexity(self, password):
        # Validate the complexity of the password
        if len(password) < 8:
            return "Password must be at least 8 characters long."
        if not re.search(r"[A-Z]", password):
```

```
            return "Password must contain at least one uppercase letter."
        if not re.search(r"[a-z]", password):
            return "Password must contain at least one lowercase letter."
        if not re.search(r"[0-9]", password):
            return "Password must contain at least one number."
        if not re.search(r"[!@#$%^&*(),.?\":{}|<>]", password):
            return "Password must contain at least one special character."
        return True
```

In addition, within the EmptyPlan class, I use a regular expression to extract the exercise name from a string of text formatted for display. The regular expression \[color=#FFFFFF](.*?)\[/color] is used to locate and retrieve the exercise name encapsulated within the color tags, which is part of the markup used in the Kivy language for styling the application's user interface.

```
ex_name = re.search(r'\[color=#FFFFFF](.*?)\[/color]', ex_name).group(1)
```

3.4.10 Other techniques:

- Linear search:For example, in the find_muscle_and_division method within the GeneratePlan class, I used a linear search to find the muscle group and division for a given exercise name (in map_exercise_details() method and switch_exercise_handler() method)
- Mathematical calculations: Like BMI, daily calorie intake, height, weight and other conversions, body fat calculations with different formulas (in DisplayCalories() class, calories() method, CalculateBodyFat() class, calculatebf() method)
- 1D Lists and 2D Lists:.For example, when I was creating workout splits in the create_split method, I used multi-dimensional arrays. Each split in the array represents a workout plan for a specific day, and each day's plan contains a list of exercises or muscle groups. Similarly, in the muscle_divisions dictionary, the values associated with each muscle group key are lists, and some of these lists contain further subdivisions. This could be interpreted as a form of a multidimensional array, where each element of the outer array (the list associated with a key in the dictionary) is itself an array.

3.5: Coding style:

- **Excellent Practices:**
  - Modules with Appropriate Interfaces: I organised the code into classes and methods (akin to modules and subroutines) with clear interfaces. For example, the GeneratePlan class and its methods have specific responsibilities and communicate with other parts of the program through their interfaces.

    - Loosely Coupled Modules: My design ensures that modules interact primarily through their interfaces. This loose coupling allows for easier maintenance and scalability.
    - Cohesive Modules: Each module in my project is designed to perform a specific task. For instance, the DatabaseManager handles all database interactions, maintaining cohesion.
    - Grouped Subroutines: Methods related to specific functionalities are grouped within appropriate classes, enhancing the modularity and organisation of the code.
    - Defensive Programming: I have employed defensive programming practices, particularly in handling potential errors or unexpected inputs in database operations.
    - Good Exception Handling: The code includes exception handling mechanisms, particularly in database interactions and file operations, to manage runtime errors effectively.
- **Good Practices:**
    - Well-designed User Interface: The use of Kivy for the UI design demonstrates a focus on creating an intuitive and user-friendly interface.
    - Modularization of Code: The code is modularized into classes and methods, enhancing readability and maintainability.
    - Effective Use of Local and Minimal Global Variables: Local variables are used within methods and classes, with minimal reliance on global variables.
    - Managed Casting of Types: Where necessary, type casting is used, for example, converting string inputs to integers for database queries.
    - Use of Constants: Constants, particularly for configuration settings, are used effectively.
    - Appropriate Indentation and Consistent Style: The code follows consistent indentation and style guidelines, making it more readable and maintainable.
    - Self-documenting Code: The code is self-explanatory with meaningful variable names and method names that clearly indicate their purpose.
    - Parameterized File Paths: File paths, especially for configuration files like JSON, are parameterized, enhancing flexibility and security.
- **Basic Practices:**
    - Meaningful Identifier Names: Variables and method names are chosen to be descriptive and meaningful, enhancing the readability of the code.
    - Effective Annotation: The code is well-commented, explaining the functionality and logic where necessary, which aids in understanding and maintaining the codebase.

Section 4:Testing:

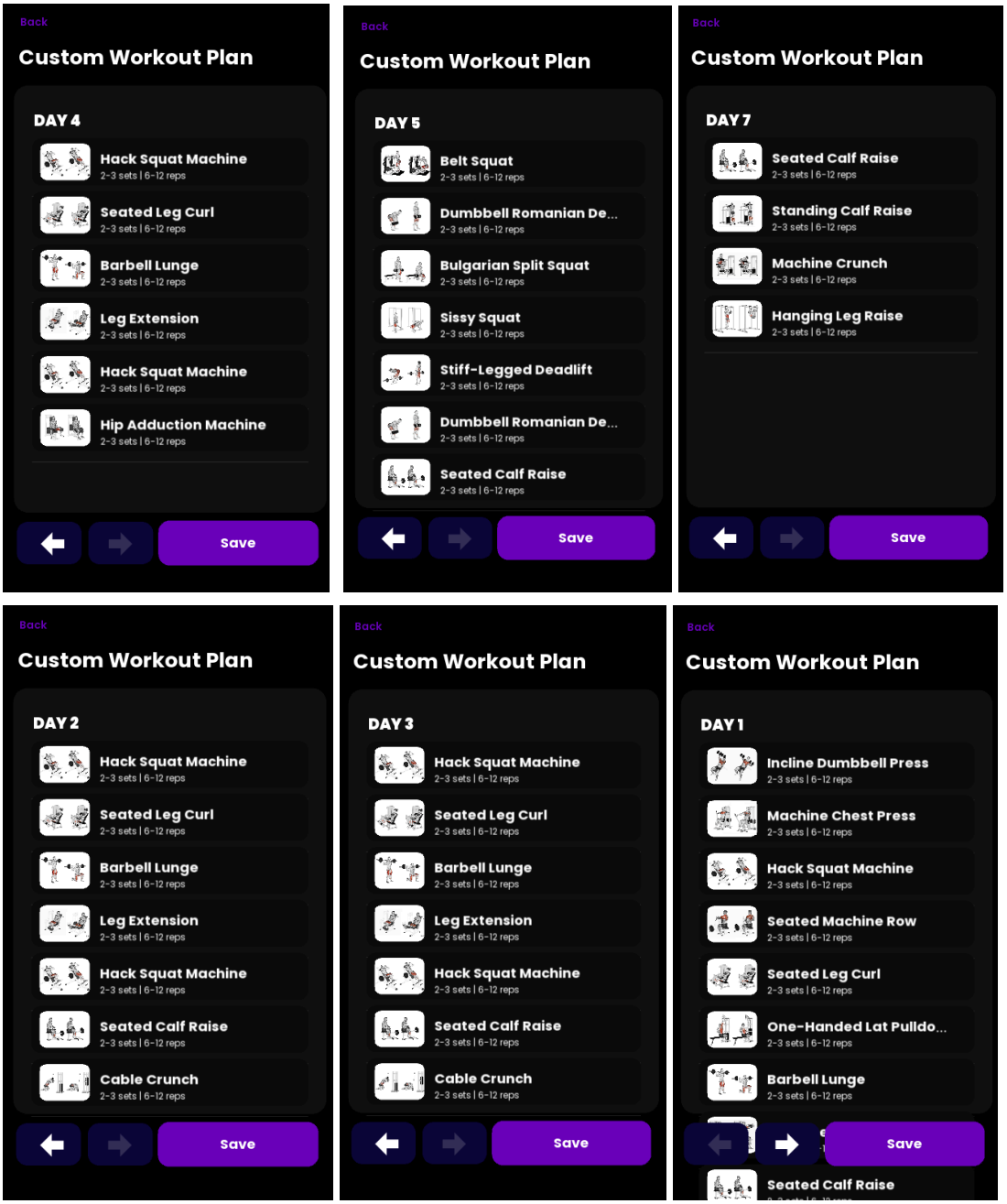
4. 1 Final Test Table

Test	Objective	Description	Expected Result	Actual result	Notes
1	1.4.1	User profiles for each plan: 5-day, 7-day, 2-day, 3-day, 1-day. All match correctly	Appropriate plans for each frequency, respecting user goals and levels.	Plan generated successfully with appropriate lengths	Day 1 had UI issues with list overlap.
2	1.4.1	User tests for male (day 4 and day 3), female (day 4 day 3), all different	Different plans for different genders.	All plans were generated correctly, and differently.	
3	1.4.1, 1.4.8	Test for user ease of use, understandability, if the user likes the workout or not	Approves workout	User approves generated plan	
4	1.4.2	search for apple, chicken, beef, mayo to check if they match	Accurate display of caloric and macronutrient data.	Database recognized all entries, nutritional info accurate.	Test diverse food items to check database comprehensiveness.
5	1.4.3	Check different portion sizes for a certain food and if the conversion looks right.	Nutritional info adjusts correctly for portion size and unit changes.	Portion size and unit changes are reflected accurately in the app.	User confirms correct adjustments
6	1.4.3, 1.4.2	Test log data feature and see if the values match.	Values should match from the logged values and the values that are used	It gave errors at first but it was fixed.	
7	1.4.2, 1.4.8	Verify ease of use and satisfaction with result from the user	Ease of use and every output looks right	Positive, no specific negatives noted	May want follow-up questions if time allows
8	1.4.6	Searches for exercises like "squat," "press," and "curl."	Accurate results with detailed technique guides and images.	Search function performed well; guides and images were helpful.	
9	1.4.7	Attempted unauthorised logins and password retrieval.	System resists login attempts; stored passwords are hashed.	it correctly uses hashing to encrypt and protect passwords, all the passwords	Regularly update security protocols and test for new threats.

				are also difficult to guess given the complexity of the requirement for it.	
10	1.4.8	Usability testing for app interface.	Users find navigation intuitive and straightforward.	Users have no confusion, positive reaction	Can still gather targeted UI suggestions after test
11	1.4.9	Track features for accuracy and clarity.	Accurate, clear visual representations of progress.	Test clarity of progress over time in graphs.	User Acceptance
12	1.4.10	Test AI conversational coach responses.	Relevant, accurate, motivational AI responses.	Assess natural language processing capabilities.	Says "hello" before each message. User seem happy about the feature.
13	1.4.4	Test app capacity for users to create workout plans with varied complexity.and retrieved correctly	Plans are saved and retrieved accurately without any data loss.	All user-created plans were stored and fetched correctly.	
14	1.4.6	Log multiple workouts, including different exercises, sets, and reps and check if all the functionalities like add set, remove set, remove exercise etc works well	The tracking feature works well and it tracks everything that is necessary.	Integration tested successfully; workout history was consistent and reliable.	

4.2 Testing Evidence:

Test 1: Generate a 4 day program. Generate a 5 day plan, a 7 day plan, a 2 day plan and a 3 day plan and a 1 day plan.
Other variables must be the same:
Output:



Frequency looks right, the right arrow's opacity is low which means it's the last screen. However the 1 day plan doesn't look right as it exceeds the space given for a page. Also, when testing, I noticed a drop down in performance while generating these, the issue might be due the use of a MDList item for each exercise and it generates the whole plan first and then it renders it, even for pages I didn't go through yet. To fix this I decided to use a Recyclerview that has a fixed size and elements inside it can't overlap it.

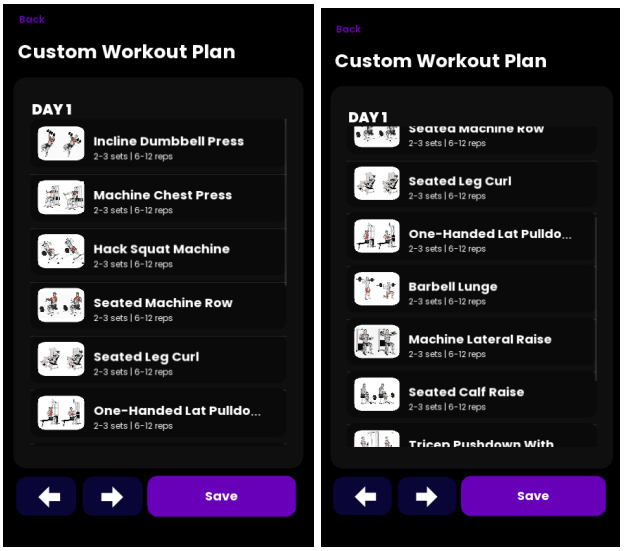
Code changed:

```
    def remove_item_handler(self, instance, value):
        # Method to handle the removal of an item from the exercise list
        # It removes the item from the list widget and updates the exercises_per_day dictionary

        # Find the index of the item to remove
        index_to_remove = None
```

```
        for i, item in enumerate(self.ids.exercise_list.data):
            if item['text'] == value['text']:  # Adjust this condition based on your data structure
                index_to_remove = i
                break
        # Remove the item from the RecycleView data
        if index_to_remove is not None:
            self.ids.exercise_list.data.pop(index_to_remove)

        if self.day in self.exercises_per_day:
            if value in self.exercises_per_day[self.day]:
                self.exercises_per_day[self.day].remove(value)

    day = 1
    exercises_per_day = {}
    def populate_exercises(self, _=None):
        # Method to update the exercise list widget for the current day
        self.ids.day_plan.text = f"DAY {self.day}"
        self.ids.exercise_list.data = []  # Clear the RecycleView data
        if self.day in self.exercises_per_day:
            for item in self.exercises_per_day[self.day]:
                # Add the item to the RecycleView data
                self.ids.exercise_list.data.append(item)


    def populate_all_exercises(self):
        # Method to populate exercises for each day in the plan
        for day in range(1, self.trainingfrequency + 1):
            self.populate_exercises_for_day(day, self.formatted_plan[f'day_{day}']['exercises'])

    def populate_exercises_for_day(self, day, exercises):
        # Method to create SwipeToDeleteItem for each exercise and add it to exercises_per_day
        exercise_items = []
        for exercise in exercises:
            item = {
                'text': f"[size=18][font=Poppins-Bold.ttf][color=#FFFFFF]{exercise['name']}[/color][/font][/size]",
                'secondary_text': f"[size=12][font=Poppins-Regular.ttf][color=#FFFFFF]2-3 sets | 6-12 reps
[/color][/font][/size]",
                'image_source': exercise['icon'],
                # Add any other properties you need
            }
            exercise_items.append(item)
        self.exercises_per_day[day] = exercise_items
    already_selected_exercises = set()
```

(Also changed the code in the kv file)

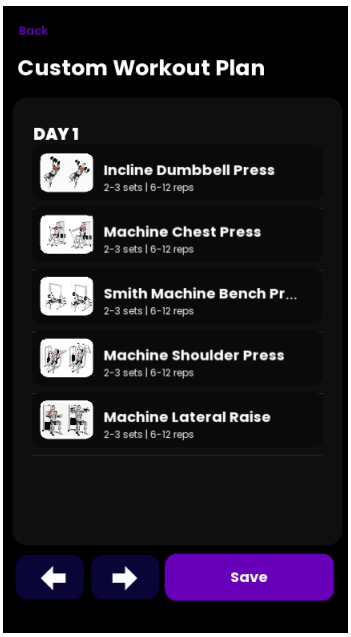
Output after changes (for 1 day frequency):

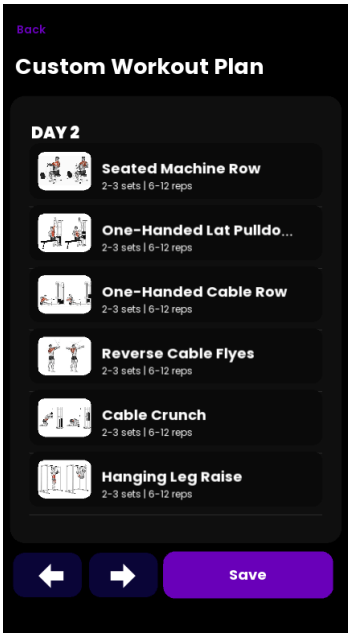


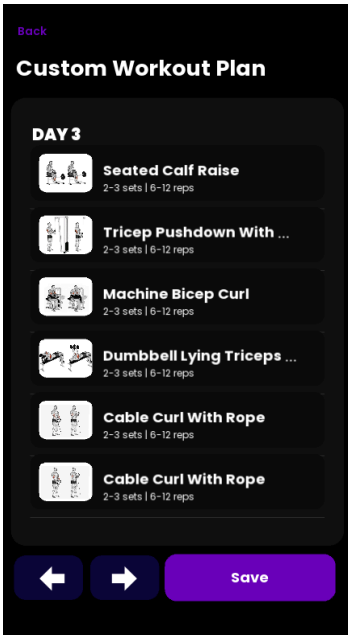
It's scrollable which fixes the issue, and the whole function feels way more reactive.

Test 2: Generate 2 different workouts with the same training frequency of 3 times and 4 times and compare results:

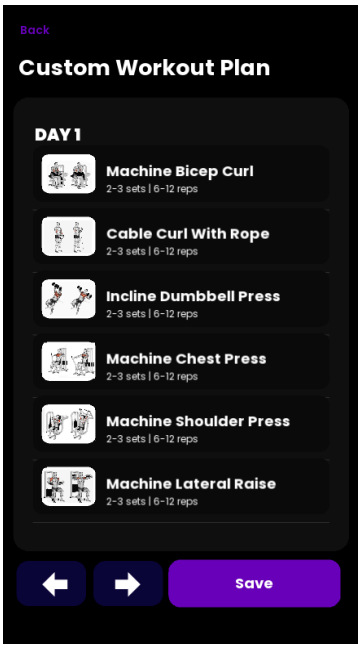
Output:
Male (4 day):

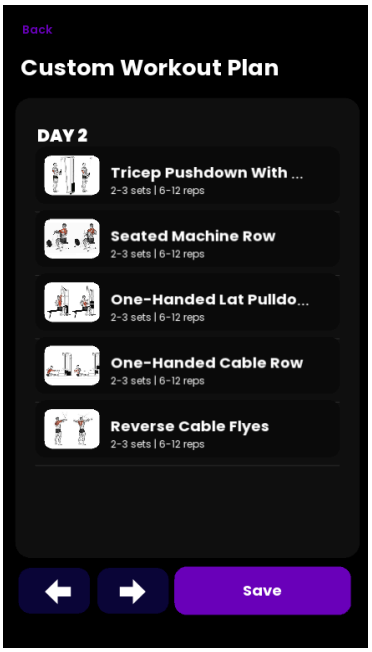


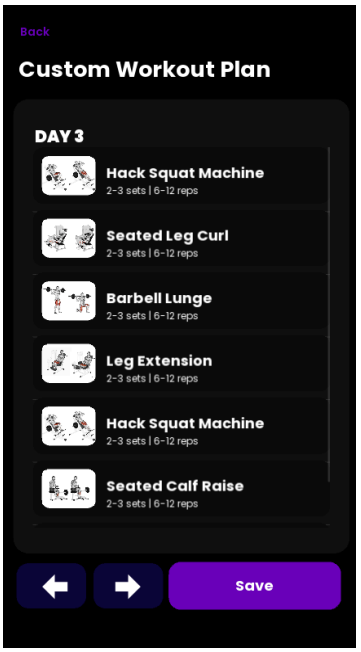




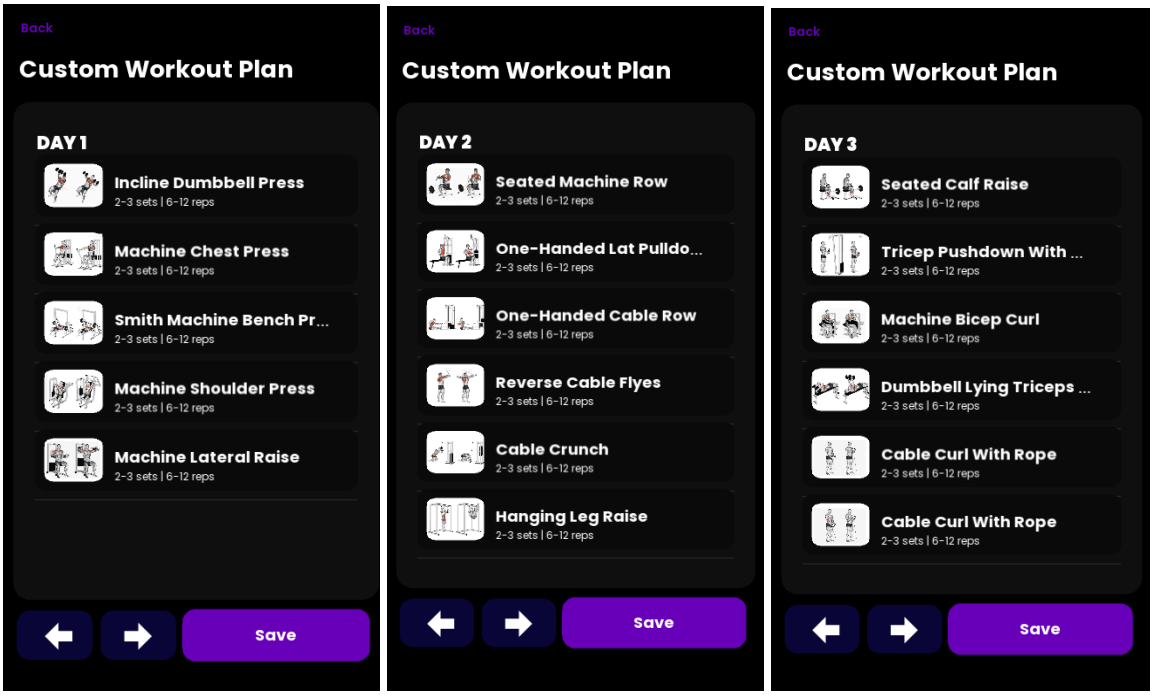
Male(3 day)



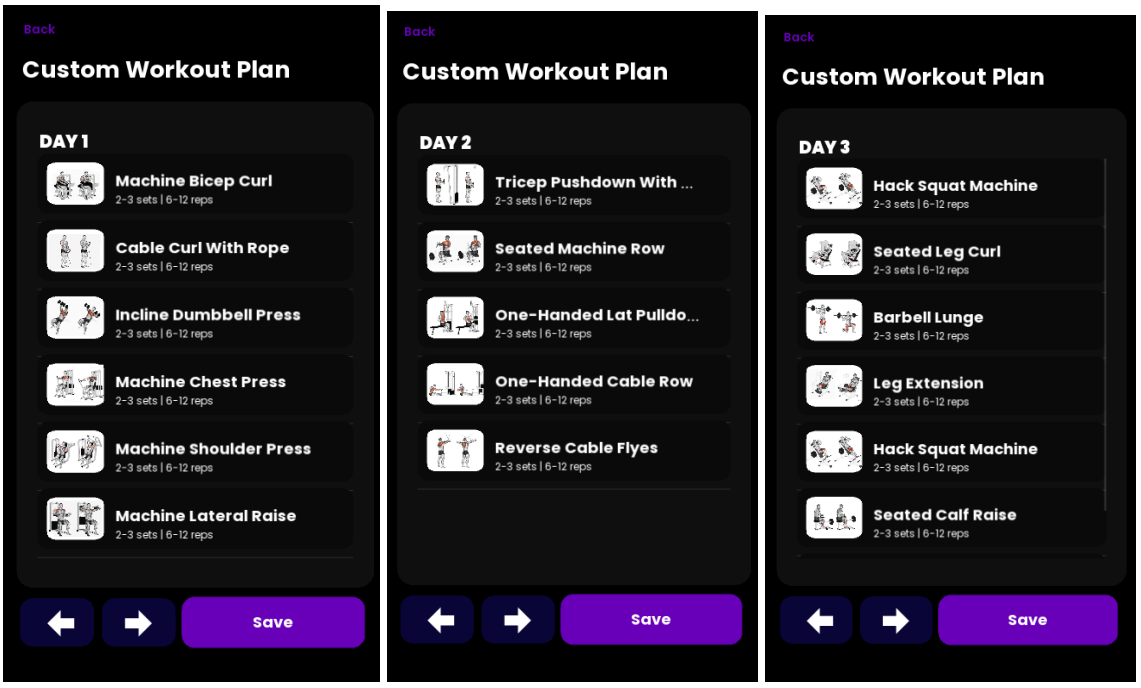




Female (4 days)

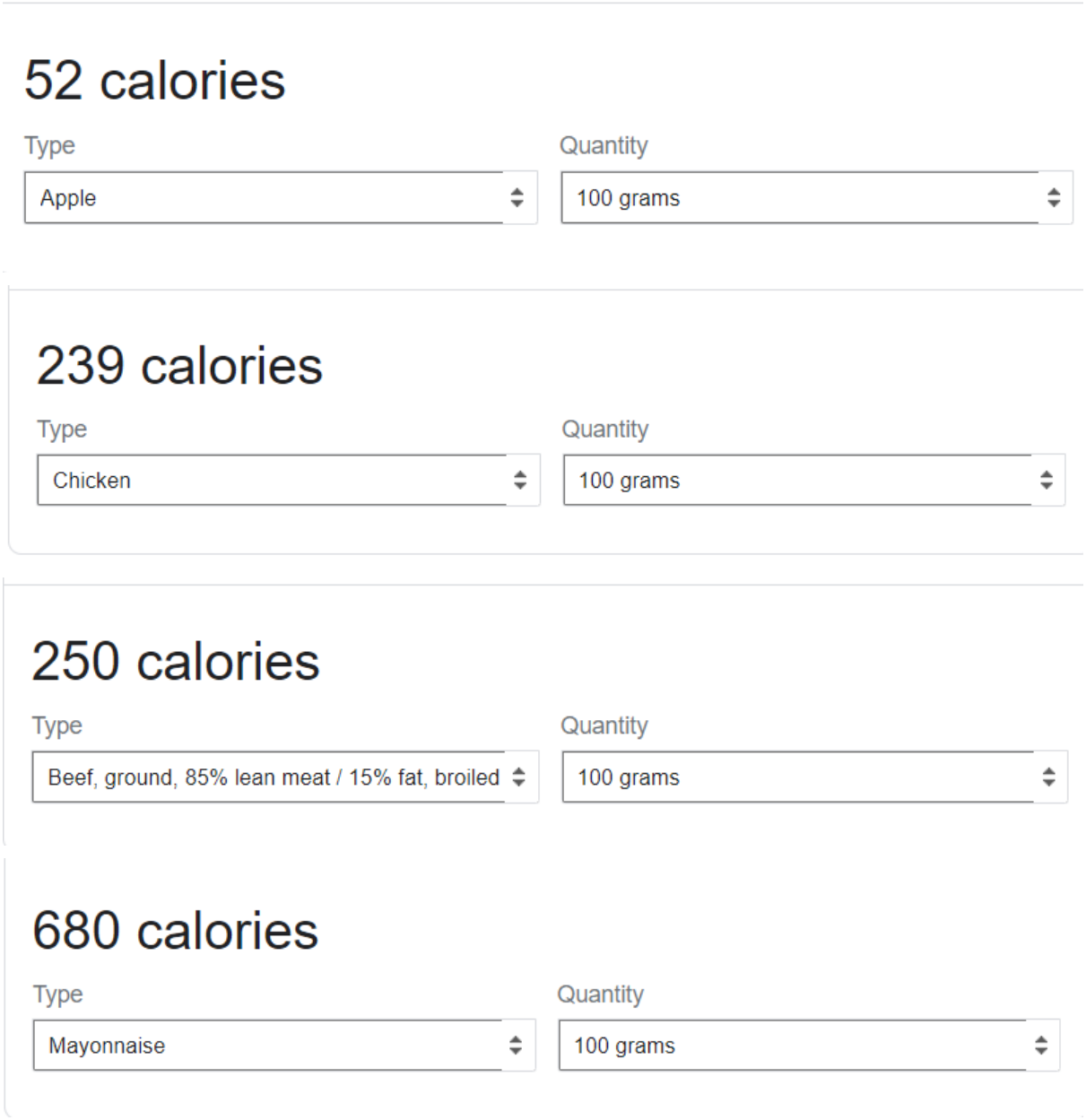


Female(3 days)

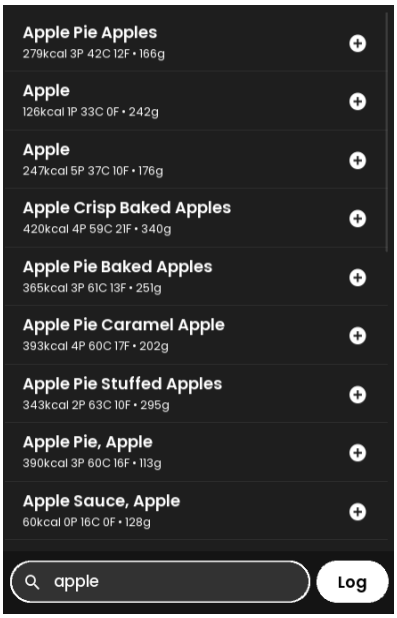


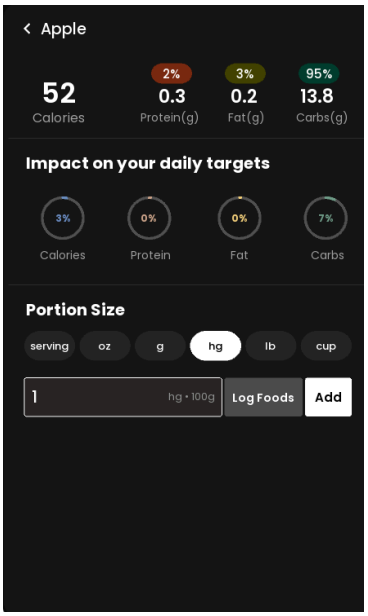
Observations:
It uses different plans, giving a little bit more leg and back focus for the female plans which is as expected

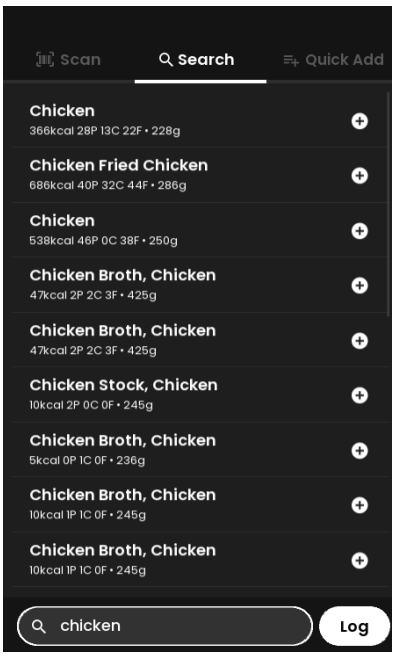
Test 4: verify food item recognition database: search for apple, chicken, beef, mayo
Expected output:

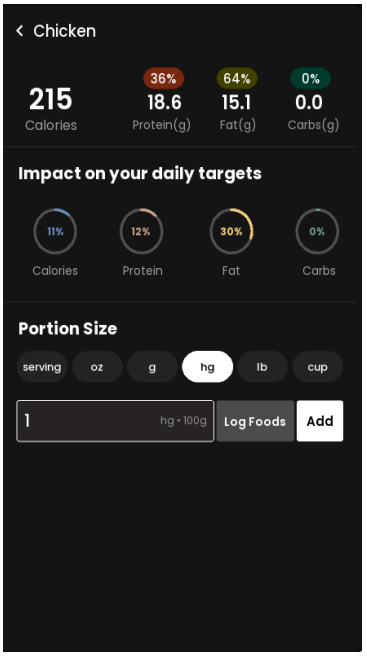

52 calories
Type
Apple
Quantity
100 grams
239 calories
Type
Chicken
Quantity
100 grams
250 calories
Type
Beef, ground, 85% lean meat / 15% fat, broiled
Quantity
100 grams
680 calories
Type
Mayonnaise
Quantity
100 grams

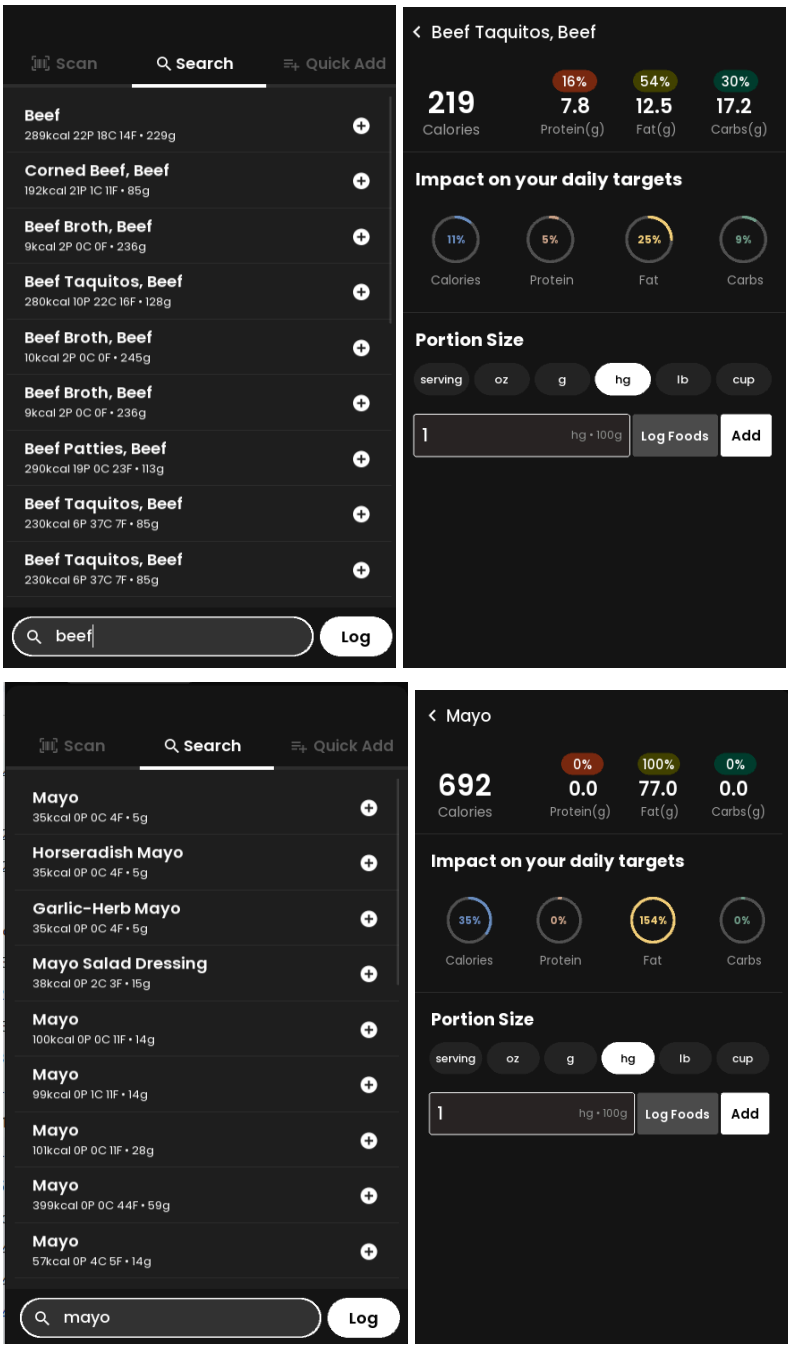
Output:


Apple Pie Apples
279kcal 3P 42C 12F • 166g
Apple
126kcal 1P 33C 0F • 242g
Apple
247kcal 5P 37C 10F • 176g
Apple Crisp Baked Apples
420kcal 4P 59C 21F • 340g
Apple Pie Baked Apples
365kcal 3P 61C 13F • 251g
Apple Pie Caramel Apple
393kcal 4P 60C 17F • 202g
Apple Pie Stuffed Apples
343kcal 2P 63C 10F • 295g
Apple Pie, Apple
390kcal 3P 60C 16F • 113g
Apple Sauce, Apple
60kcal 0P 16C 0F • 128g
apple
Log


Apple
52
Calories
2%
0.3
Protein(g)
3%
0.2
Fat(g)
95%
13.8
Carbs(g)
Impact on your daily targets
3%
Calories
0%
Protein
0%
Fat
7%
Carbs
Portion Size
serving
oz
g
hg
lb
cup
1
hg • 100g
Log Foods
Add

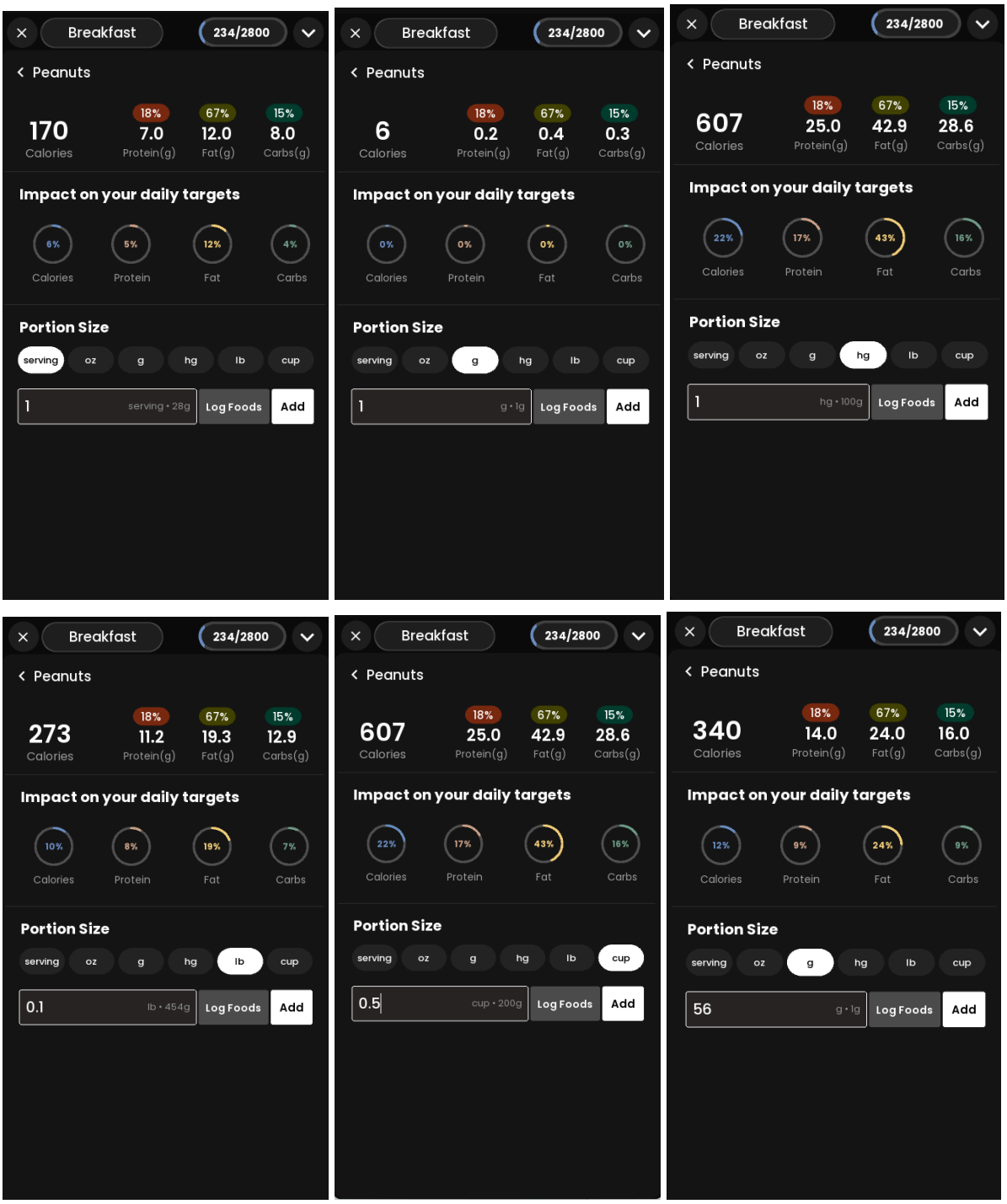

Scan
Search
Quick Add
Chicken
366kcal 28P 13C 22F • 228g
Chicken Fried Chicken
686kcal 40P 32C 44F • 286g
Chicken
538kcal 46P 0C 38F • 250g
Chicken Broth, Chicken
47kcal 2P 2C 3F • 425g
Chicken Broth, Chicken
47kcal 2P 2C 3F • 425g
Chicken Stock, Chicken
10kcal 2P 0C 0F • 245g
Chicken Broth, Chicken
5kcal 0P 1C 0F • 236g
Chicken Broth, Chicken
10kcal 1P 1C 0F • 245g
Chicken Broth, Chicken
10kcal 1P 1C 0F • 245g
chicken
Log


Chicken
215
Calories
36%
18.6
Protein(g)
64%
15.1
Fat(g)
0%
0.0
Carbs(g)
Impact on your daily targets
11%
Calories
12%
Protein
30%
Fat
0%
Carbs
Portion Size
serving
oz
g
hg
lb
cup
1
hg • 100g
Log Foods
Add



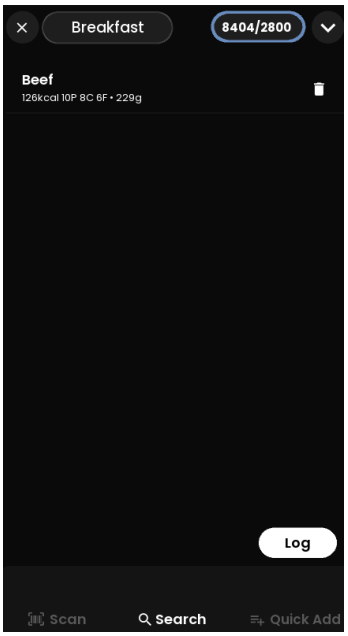
Observations: the output seems right for all of them, the values are close enough to the original values to be considered acceptable.

Test 5: Check different portion sizes for a certain food and if the conversion looks right.
Output:

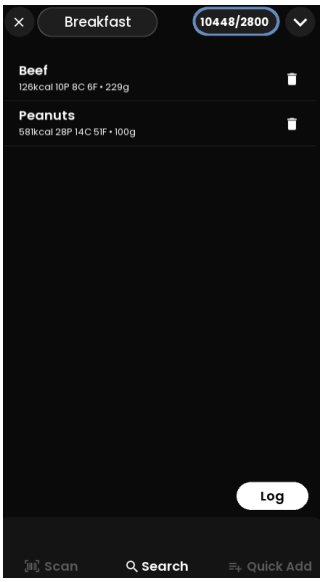


Observations: output looks right for all unit sizes and portion size, also the impact on daily target looks right as well.

Test 6: Test log food into the database feature
Output:

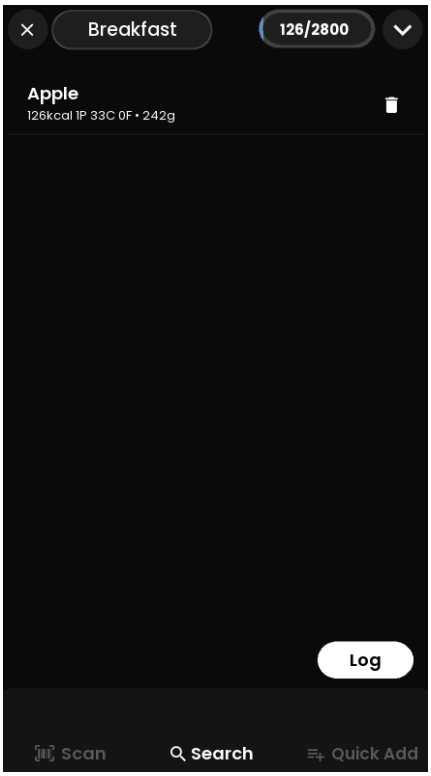


The output doesn't look right, let’s try with a different food.

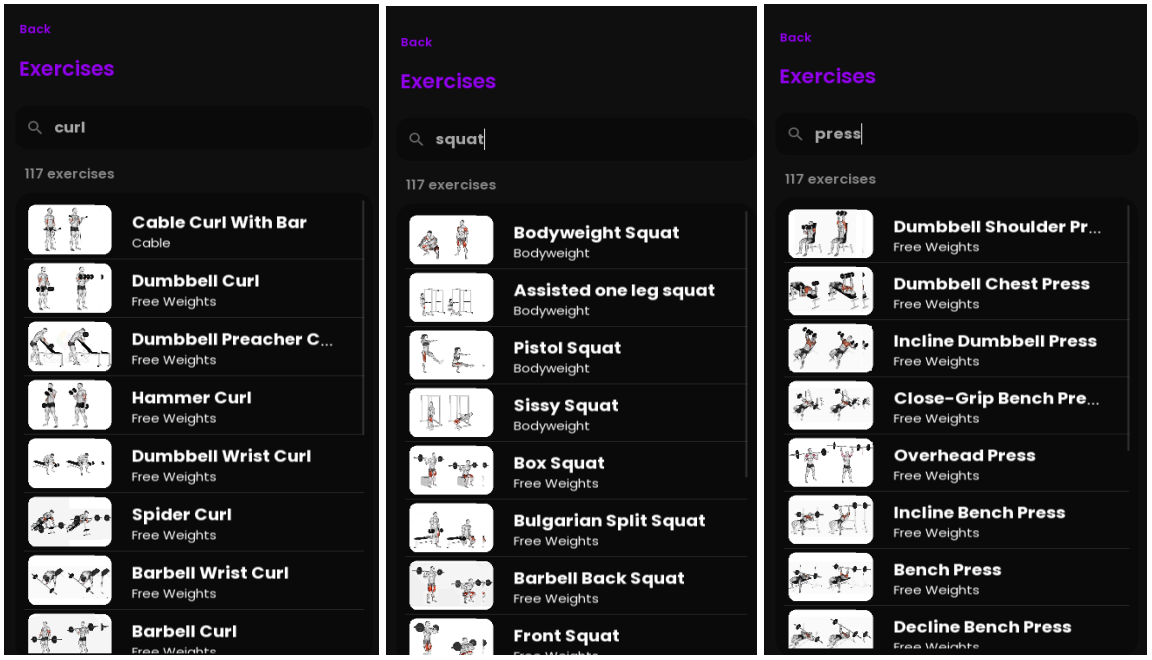


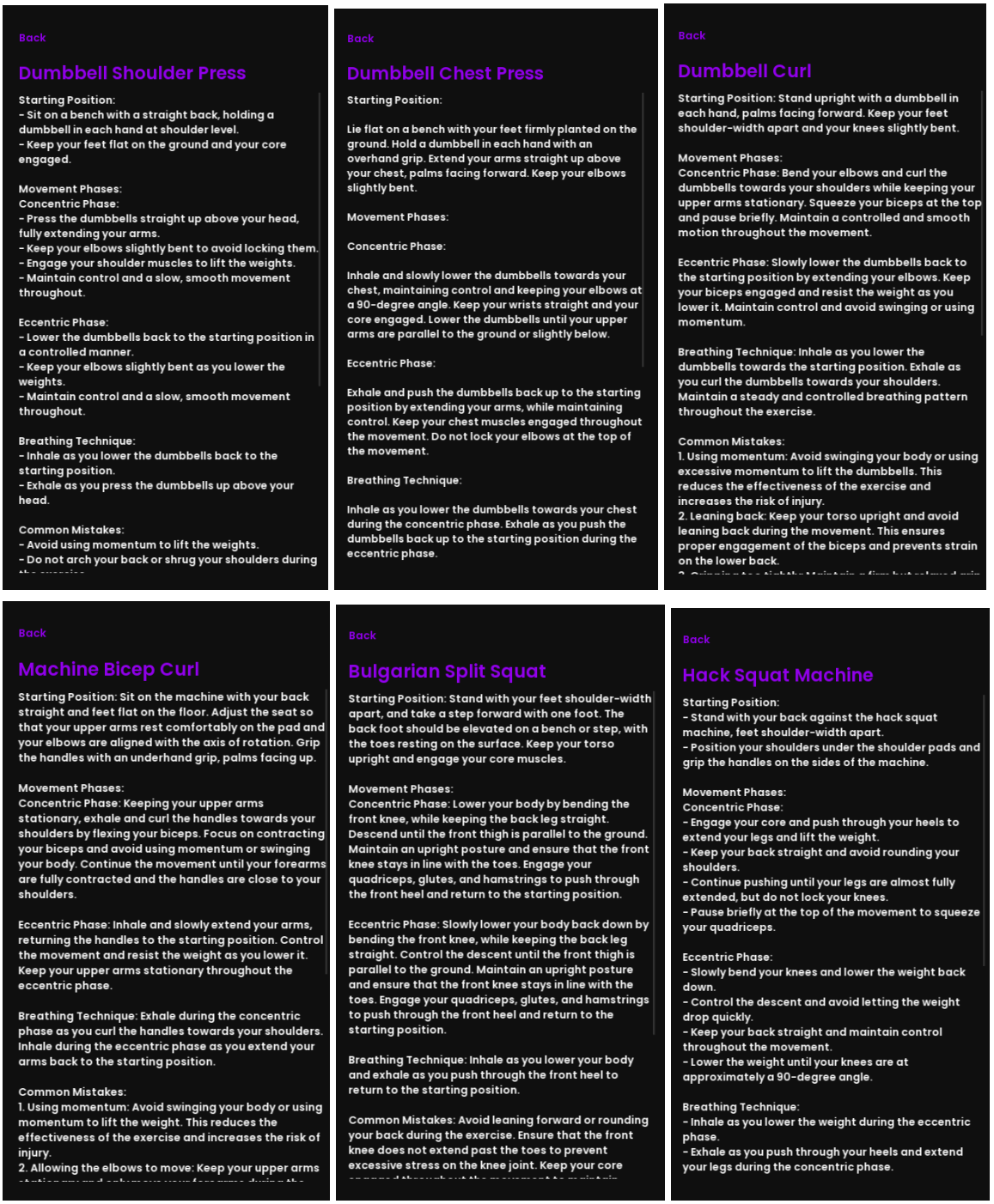
Same issue, let’s see if we can pinpoint the origin of the problem.

Found it, it was a database issue where it would get old data, sum it and displays it over the calories amount, it should be fixed now.



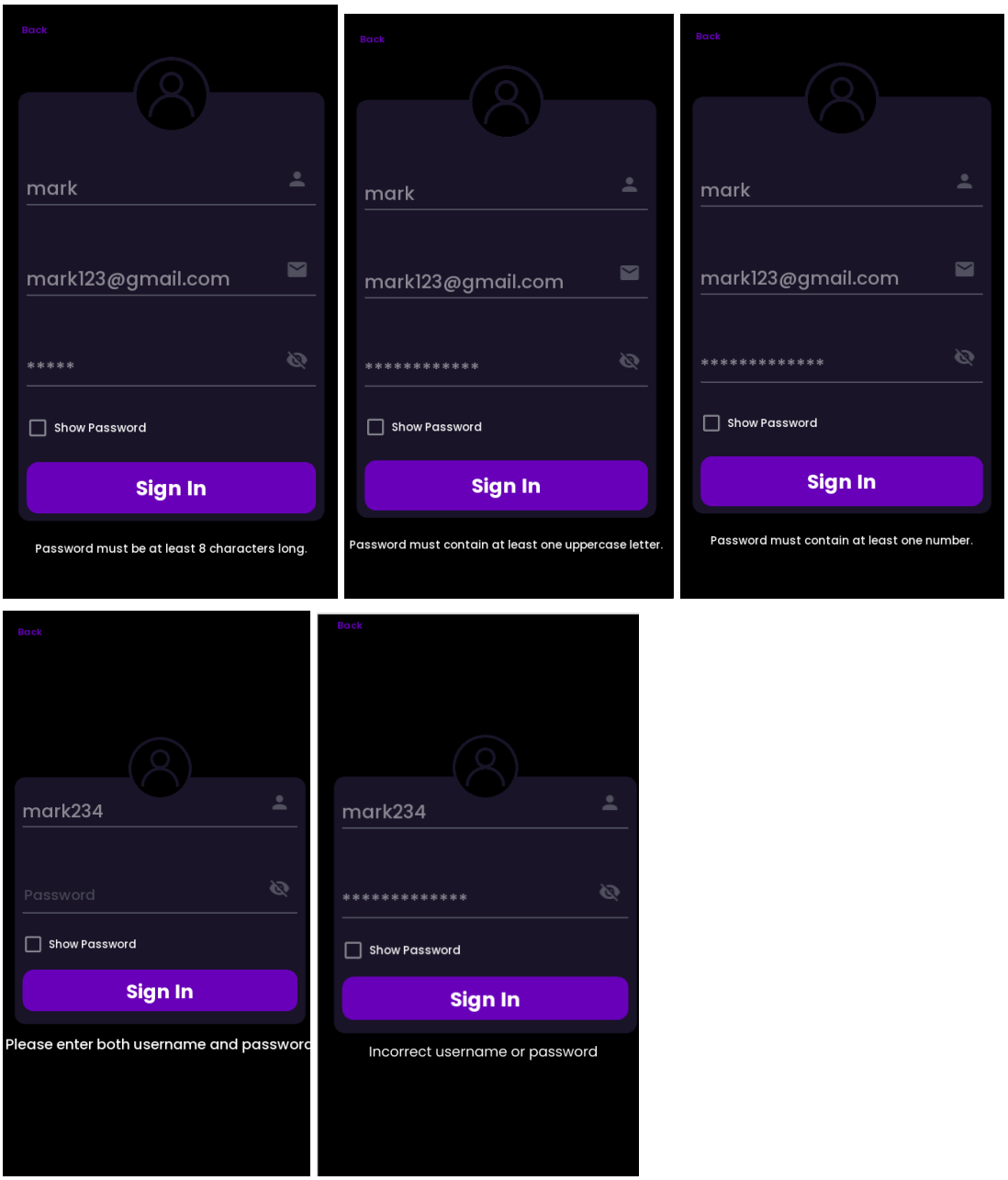
Test 8: Searches for exercises like "squat," "press," and "curl."
Output:





Observations: output seems to match up with the expected result.

Test 9: Attempted unauthorised logins and password retrieval and password complexity test.

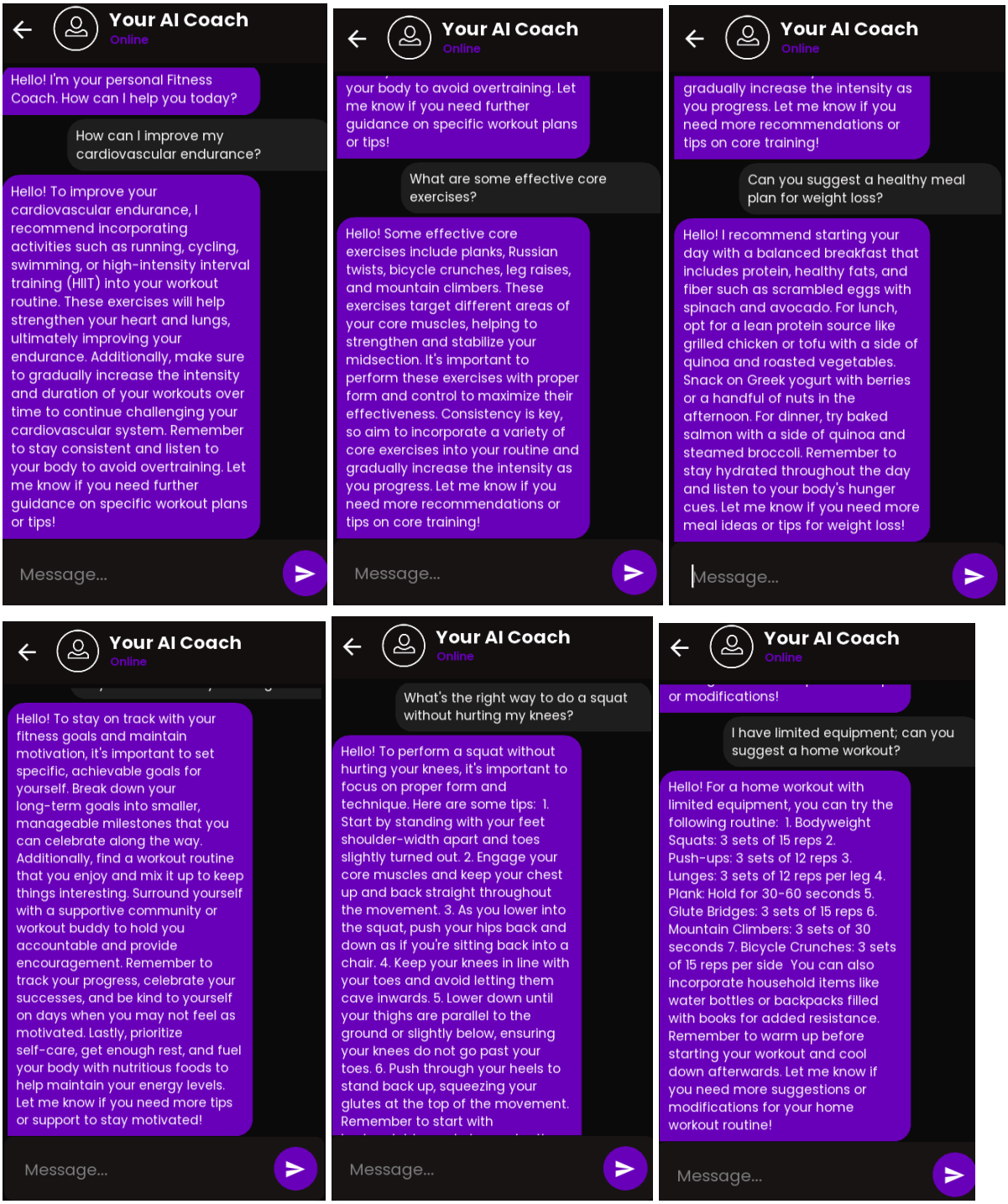


(with the actual password and username it works properly.)
Observations: it correctly uses hashing to encrypt and protect passwords, all the passwords are also difficult to guess given the complexity of the requirement for it.

Test 12: Ask AI fitness questions
Example questions:

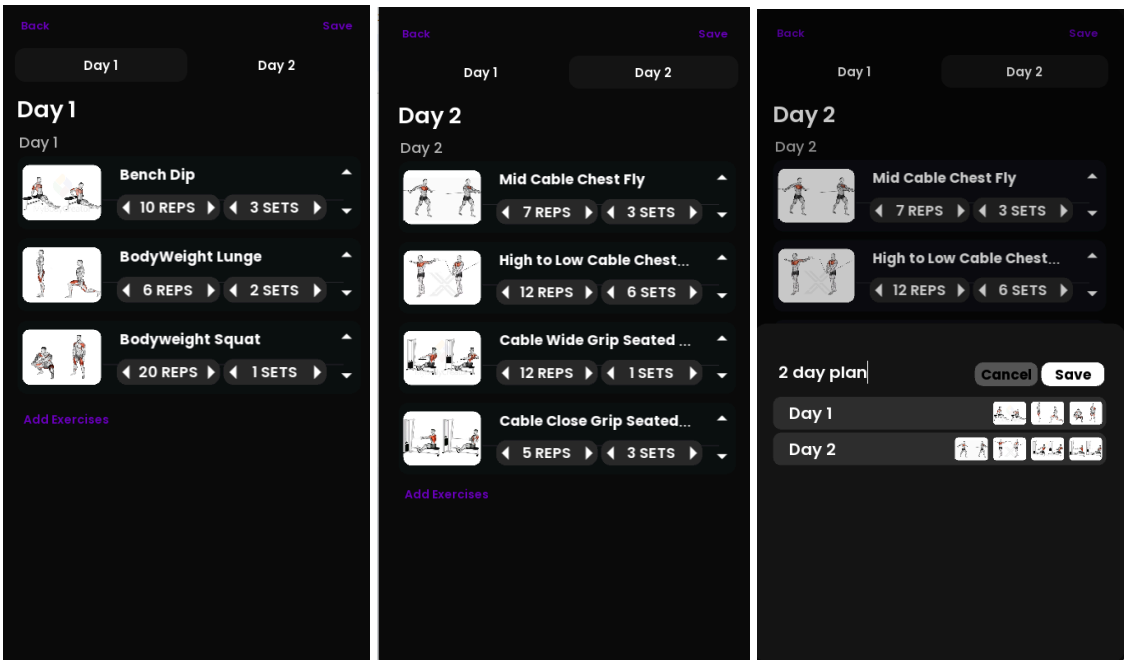
- "How can I improve my cardiovascular endurance?"
- "What are some effective core exercises?"
- "Can you suggest a healthy meal plan for weight loss?"
- "I'm feeling demotivated, how can I stay on track with my fitness goals?"
- "What's the right way to do a squat without hurting my knees?"
- "I have limited equipment; can you suggest a home workout?"

Output:

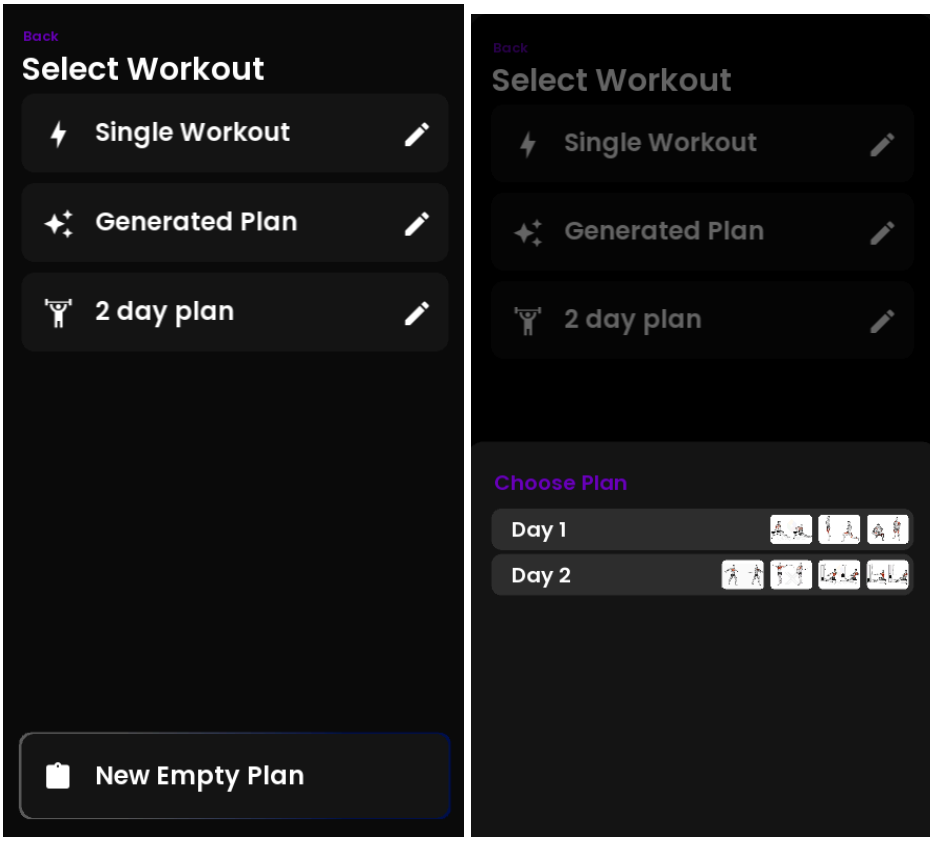


Observations: the output seems coherent, understandable and easy to follow. It seems to repeat "hello" all the time even if the conversation is still ongoing.

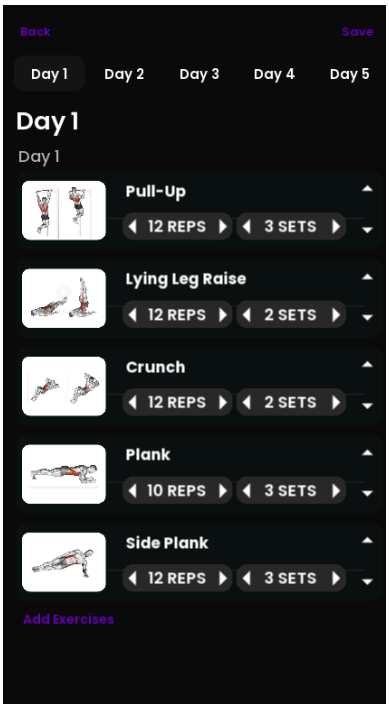
Test 13:
Test app capacity for users to create workout plans with varied complexity.and retrieved correctly

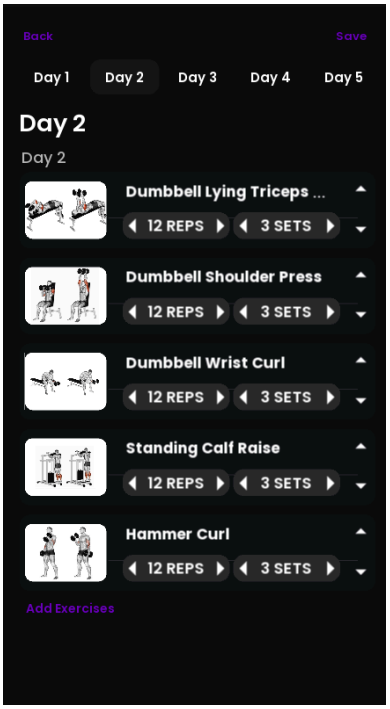


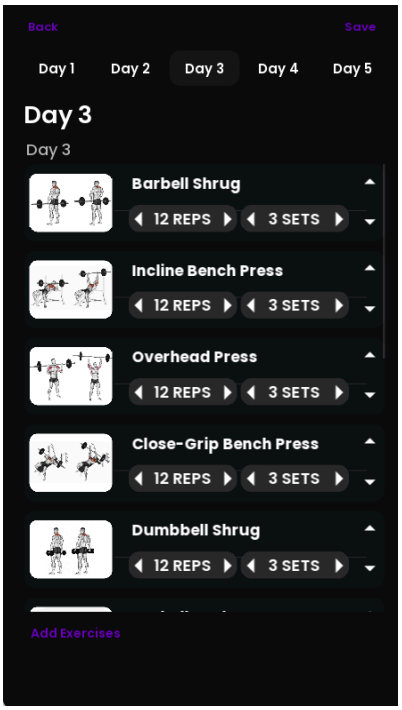
Simple 2 day plan

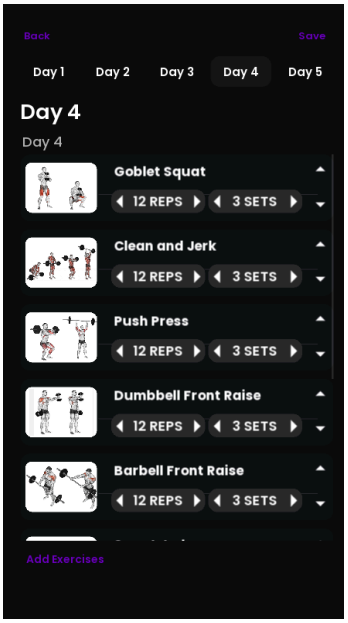


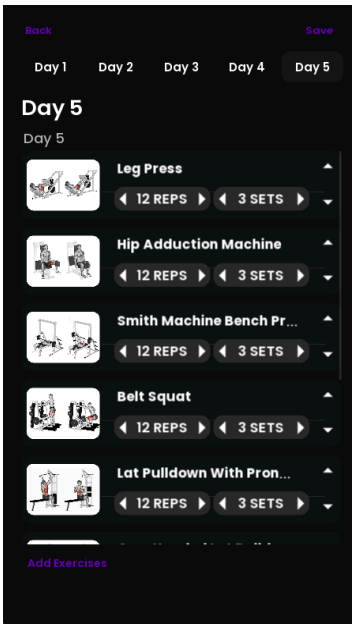
Retrieved correctly from database.

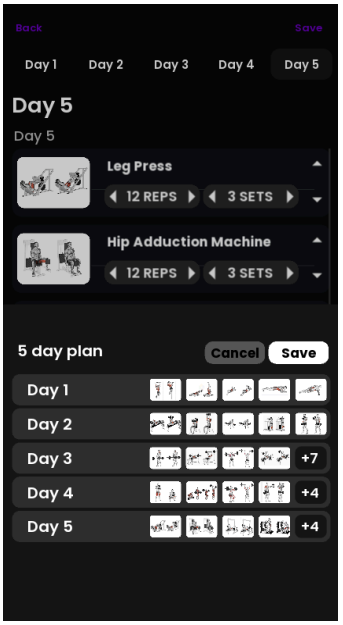

Back
Save
Day 1
Day 2
Day 3
Day 4
Day 5
Day 1
Day 1
Pull-Up
12 REPS
3 SETS
Lying Leg Raise
12 REPS
2 SETS
Crunch
12 REPS
2 SETS
Plank
10 REPS
3 SETS
Side Plank
12 REPS
3 SETS
Add Exercises

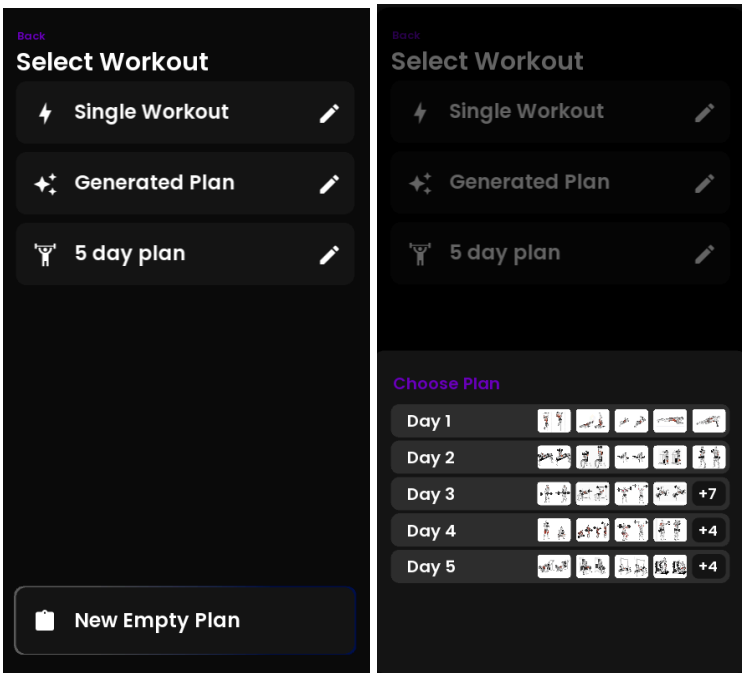

Back
Save
Day 1
Day 2
Day 3
Day 4
Day 5
Day 2
Day 2
Dumbbell Lying Triceps ...
12 REPS
3 SETS
Dumbbell Shoulder Press
12 REPS
3 SETS
Dumbbell Wrist Curl
12 REPS
3 SETS
Standing Calf Raise
12 REPS
3 SETS
Hammer Curl
12 REPS
3 SETS
Add Exercises


Back
Save
Day 1
Day 2
Day 3
Day 4
Day 5
Day 3
Day 3
Barbell Shrug
12 REPS
3 SETS
Incline Bench Press
12 REPS
3 SETS
Overhead Press
12 REPS
3 SETS
Close-Grip Bench Press
12 REPS
3 SETS
Dumbbell Shrug
12 REPS
3 SETS
Add Exercises


Back
Save
Day 1
Day 2
Day 3
Day 4
Day 5
Day 4
Day 4
Goblet Squat
12 REPS
3 SETS
Clean and Jerk
12 REPS
3 SETS
Push Press
12 REPS
3 SETS
Dumbbell Front Raise
12 REPS
3 SETS
Barbell Front Raise
12 REPS
3 SETS
Add Exercises


Back
Save
Day 1
Day 2
Day 3
Day 4
Day 5
Day 5
Day 5
Leg Press
12 REPS
3 SETS
Hip Adduction Machine
12 REPS
3 SETS
Smith Machine Bench Pr...
12 REPS
3 SETS
Belt Squat
12 REPS
3 SETS
Lat Pulldown With Pron...
12 REPS
3 SETS
Add Exercises


Back
Save
Day 1
Day 2
Day 3
Day 4
Day 5
Day 5
Day 5
Leg Press
12 REPS
3 SETS
Hip Adduction Machine
12 REPS
3 SETS
5 day plan
Cancel
Save
Day 1
Day 2
Day 3
+7
Day 4
+4
Day 5
+4

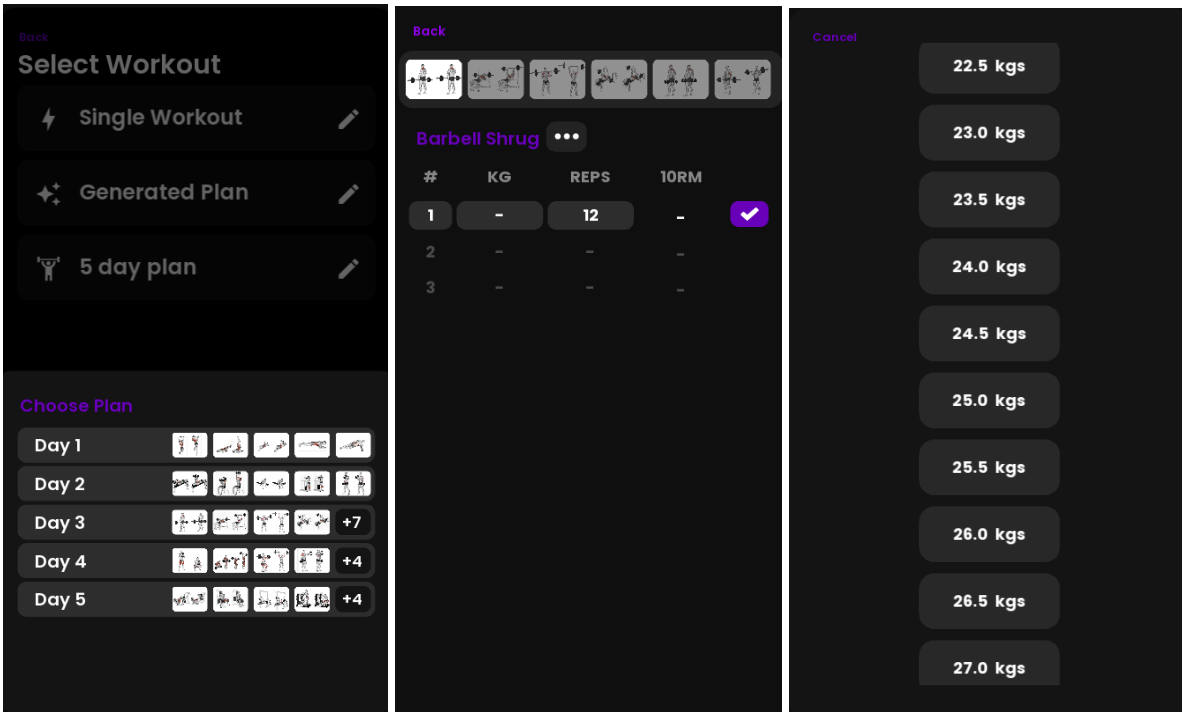


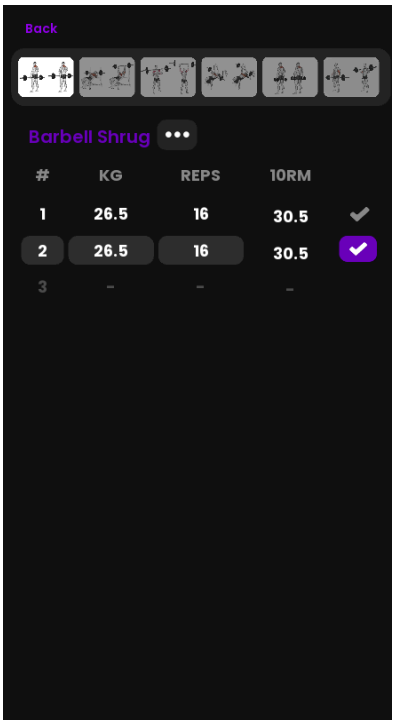
Retrieved workout correctly.

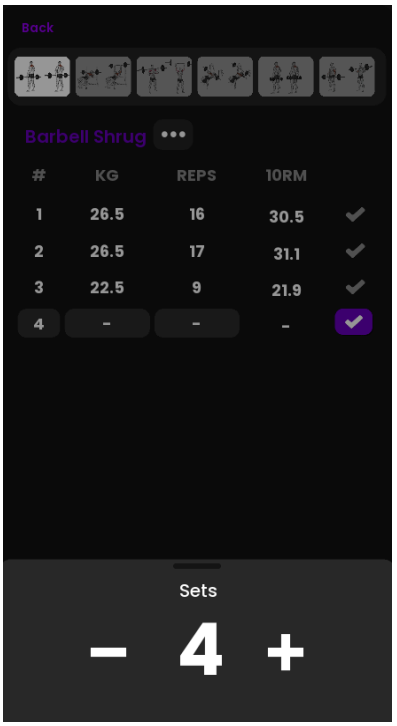
Observations: everything seems to be working seamlessly. It stores the workout in the database and it retrieves it back.

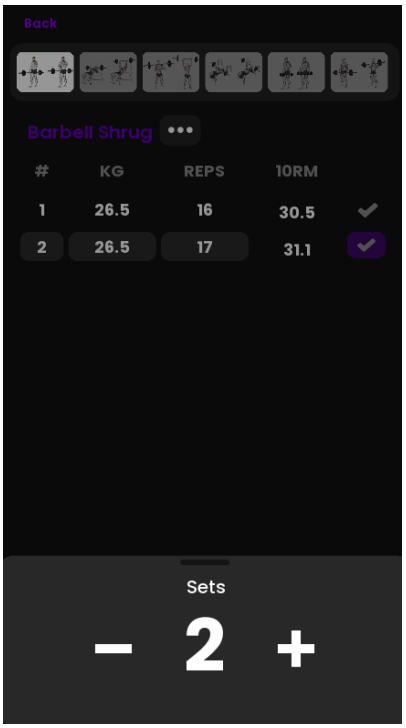
Test 14:
Log multiple workouts, including different exercises, sets, and reps and check if all the functionalities like add set, remove set, remove exercise etc works well.

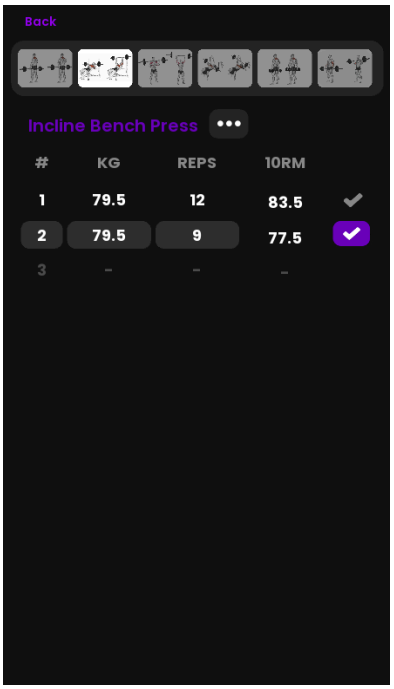
(using 5 day plan made earlier)

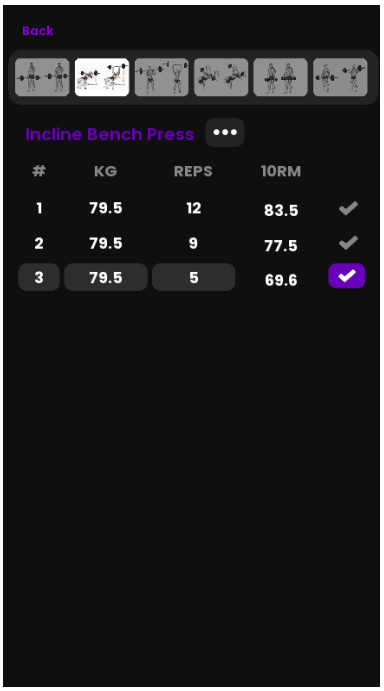


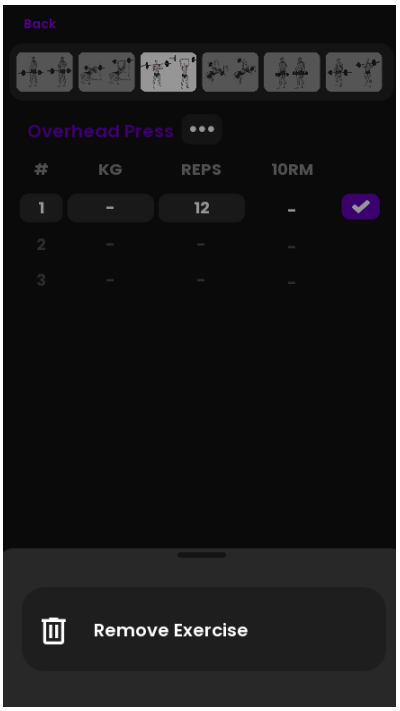

Back
Barbell Shrug
KG REPS 10RM
1 26.5 16 30.5
2 26.5 16 30.5
3 - - -

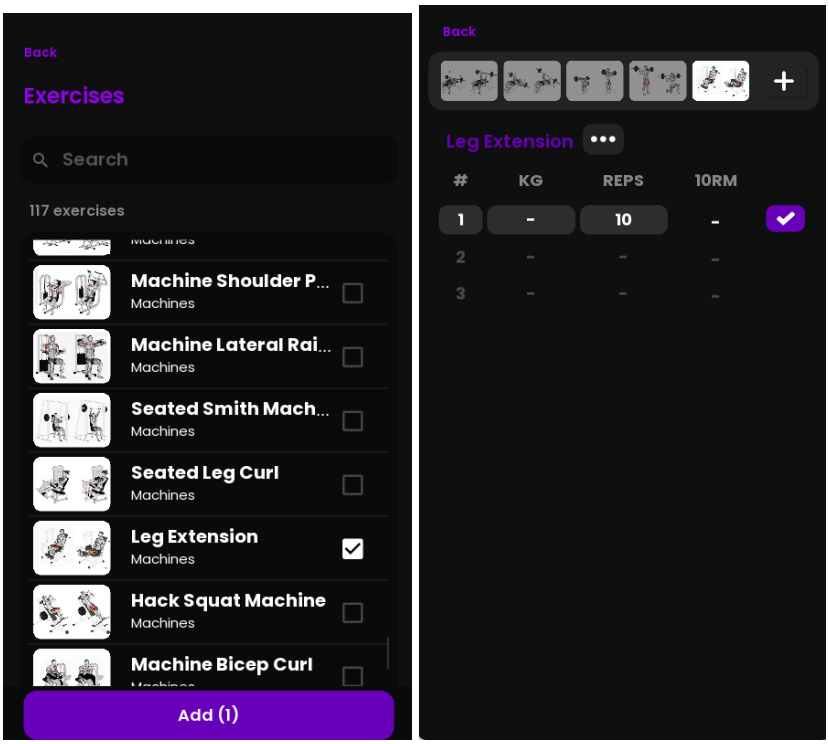

Back
Barbell Shrug
KG REPS 10RM
1 26.5 16 30.5
2 26.5 17 31.1
3 22.5 9 21.9
4 - - -
Sets
- 4 +


Back
Barbell Shrug
KG REPS 10RM
1 26.5 16 30.5
2 26.5 17 31.1
Sets
- 2 +


Back
Incline Bench Press
KG REPS 10RM
1 79.5 12 83.5
2 79.5 9 77.5
3 - - -


Back
Incline Bench Press
KG REPS 10RM
1 79.5 12 83.5
2 79.5 9 77.5
3 79.5 5 69.6


Back
Overhead Press
KG REPS 10RM
1 - 12 -
2 - - -
3 - - -
Remove Exercise



Observations: all the features seems to be working correctly
Test 3, 7, 10 (User feedback)

Feedback from Misha, "Wow, I like how I can choose a plan based on my goals and how much time I have. It's nice that there are options for beginners, intermediates, and advanced users. But I found the exercise library a bit overwhelming at first. Maybe some filters or categories would help narrow it down? The generated plan was great, looks like i could try that! The food part was nice, it was easy to understand how it works and navigate through it. I tried the plan making and logging a workout part too and it looks very user friendly and modern. I liked the little shaky animation you added if you clicked on the tick button and you still didn't put on weight or reps. The AI chatbot looks like a nice add too for a fitness app. And while the plan customization is excellent, consider improving exercise discoverability. Implement filters by muscle group, equipment needed, or difficulty level to streamline exercise selection, especially for beginners. But this was amazing as a nice side project so well done!"

Feedback from Bashir: "The food database is cool, I easily found everything I ate yesterday, even ì my favourite protein powder haha. Logging portion sizes and macros looks well structured. But it would be fantastic if I could set custom daily goals for each macronutrient, like some other fitness apps do. Also the other features like logging a workout, and making a plan is nice, i like the generated plan that it gives me, i'll try it and let you know if i like it. And I Like the UI of the app, it's well designed. Oh and the AI coach is cool, later you'll tell me how you added it"

Feedback from Hamed: "The AI coach is a nice feature. It gives me motivational tips and answers my questions about form and nutrition in a clear way. But , sometimes the

responses feel a bit generic and long. Maybe the coach could personalise its advice based on my workout history and goals?
I really liked the logging system though,it's very smooth and easy to use, the generated plan feature is not bad but I think I'll stick to my workout plan for these few weeks and then maybe I'll give it a shot. The food logging part it's nice but I don't think I'd use it since I don't like counting calories. I like all the graphs and bars you added though. And what do I think about the UI? I think it's impressive, you're good at graphic design"

Section 5: Evaluation

5.1 Evaluating Objectives

Objective 1.4.1 (Customizable Workout Plans)

- **Objective Description:**
  - The goal was to develop a dynamic algorithm capable of generating customizable workout plans based on user-specific inputs such as age, weight, goals, and experience level, ensuring a broad engagement with all major muscle groups and catering to various experience levels.
- **Implementation Methods:**
  - Implemented a flexible algorithm within the GenerateWorkoutPlan class that considers user inputs to tailor workout plans. Utilised a tree structure for exercise selection to prioritise user preferences and ensure balanced muscle group engagement.
- **Testing and Results:**
  - Conducted black box testing and asked users to assess the algorithm's adaptability and accuracy in creating personalised plans. Results indicated a high degree of customization, with all plans accurately reflecting user inputs.
- **User Feedback and Observations:**
  - Feedback was generally positive, with users appreciating the level of personalization and variety in their workout plans.
- **Challenges:**
  - Faced challenges in UI design, particularly in managing all the data required to make the tree structure, categorising each exercise neatly and transforming the data into a dictionary and then into a hierarchical tree structure. Since this was the first time for me dealing with this kind of data structure it was hard to understand how to do what and which steps to take to achieve what I wanted, so I had to watch several youtube videos explaining how they can be used and how I can implement them using python.

- **Conclusion and Improvements:**
- I believe that this objective was achieved and I am happy with how it turned out,I didn't really put exercises to do with callisthenics since they require a different type of progressive overload, also I put the choice for powerlifting and i didn't really do anything with it since I had to follow a different approach than regular workout plans,

all things I could have done if i had more time.Additionally, I think I learned a lot about python, tree data structures while completing this objective.

Objective 1.4.2: Comprehensive Food Tracking Integration

- **Objective Description:**
  - The aim was to integrate a comprehensive food database that includes a wide range of foods, including various cuisines, brands, and whole foods, enabling users to log foods easily into a database and have graphs displayed for calories and macronutrients consumed.

- **Implementation Methods:**
  - Integrated a food database API that provides extensive food item data. Developed a user-friendly interface within the FoodSearch class for easy food item logging, coupled with graphical representations of nutritional intake.

- **Testing and Results:**
  - Utilised black box testing to ensure the food database covered a diverse range of food items. Testing involved searching for common and uncommon food items, resulting in successful recognition and accurate nutritional logging in all tested cases.
- **User Feedback and Observations:**
  - Users appreciated the extensive coverage of the food database, finding it easy to log daily food intake. However, some users suggested implementing custom daily goals for macronutrients for enhanced personalization.
- **Challenges:**
  - The main challenge was ensuring that all the functionalities and functions would work correctly and would interact nicely with each other. I struggled with the design of it and making every aspect of the UI work, it was also very hard to find a good free food database to make API calls to and figuring out how to deal with the JSON files that were fetched.
- **Conclusion and Improvements:**
  - This objective was effectively met, with the food tracking integration offering broad food item coverage and user-friendly logging. Future improvements may add things like a quick manual entry and a barcode scanning feature that I would've implemented if I had more time. Also, while making this class I managed to figure out how to make a draggable panel custom widget which wasn't really something you could do directly within the Kivy framework and I have to admit that I wish figured this out sooner. I learned about how graphs in Kivy work too.

Objective 1.4.3: Nutritional Breakdown of Meals

- **Objective Description:**
  - The objective focused on providing a detailed nutritional breakdown for each logged meal, including calories and macronutrients, with the flexibility for users to adjust portion sizes and units of measurement.
- **Implementation Methods:**

- Implemented functionality within the FoodSearch class that allows users to view and adjust the nutritional content of meals based on portion sizes. The system dynamically updates the nutritional breakdown as users modify portions.

- **Testing and Results:**
  - Conducted usability testing to assess the accuracy and user-friendliness of the nutritional breakdown feature. Tests involved logging various meals with different portion sizes, confirming that the app accurately recalculated nutritional values.

- **User Feedback and Observations:**
  - Users found the nutritional breakdown feature helpful for understanding meal composition. The ability to adjust portion sizes was particularly appreciated. However, some users requested more intuitive controls for portion size adjustments.

- **Challenges:**
  - Encountered challenges in designing an intuitive UI for portion size adjustments.I struggled a lot with updating all the variables and attributes correctly and ensuring that the original values don't change when getting data from the database, calculating the portion sizes and all the different units, displaying everything correctly with nice circular graphs and implementing a method for saving, removing, the food item added.

- **Conclusion and improvements:**
  - I think this objective was successfully achieved, providing users with detailed and adjustable nutritional breakdowns of meals. Things I could improve on and implement is to use these data to show graphs about calories and macros spending patterns over time, another feature that I would've added if I had more time.

Objective 1.4.4: User-Created Workout Plan Design

- **Objective Description:**
  - This objective aimed to provide an interface for users to design their own workout plans from an extensive exercise library, including selecting days, exercises, sets, reps, and rest periods.

- **Implementation Methods:**
  - Developed the EmptyPlan class, which offers users a platform to create, customise, and save their workout plans. The interface includes a comprehensive exercise library accessed through the ExerciseList class for exercise selection.

- **Testing and Results:**
  - Functional testing was conducted to ensure that all features worked as intended, allowing for the creation of workout plans with varied complexity.

Users were able to save their plans, which were accurately retrievable for later use.

- **User Feedback and Observations:**
  - Feedback from user testing was positive, with users praising the flexibility and comprehensiveness of the workout plan design feature

- **Challenges and Improvements:**
  - The first challenge was a very small thing which was the calculation of the size of the rectangle that would go underneath the "Day 1" or "Day 2" text whenever it got clicked, I tried different formulas because it wasn't very straightforward as to where to put the text labels. The second hardest thing to implement was the navigation between screens that i wanted to implement, it needed to stay in the same screen but having a slide transition while switching to different days, so I tried using two screens, and it took me very long to find out how to implement the logic that would make switching between these two screens even though there were multiple days. After I managed to sort this I kept having issues while using the checkboxes in the ExerciseList class to select the exercises, since I used a recycle view which generates items to display in a list procedurally to improve performance it would also remove the already marked checkboxes. It took me a while to figure out how to fix this, I struggled with how I wanted to save the plan dictionary too and how to process this in an sql table after normalising it and selection within the extensive exercise library.

- **Conclusion and Improvements:**
  - I believe that this objective was met, with the platform successfully enabling users to create personalised workout plans. An improvement would be to introduce smart suggestions for exercises based on the user's goals and previous selections to guide more balanced plan creation. I think that adding this section taught me a lot on how python and how instances of classes work, it also allowed me to use the same dynamic changing screen technique for the Logworkout class which saved a lot of hard work and time.

Objective 1.4.5: Extensive Exercise Library and Technique Guidance

- **Objective Description:**
  - The objective was to provide an extensive library of exercises, complete with images illustrating each exercise. The aim was to implement a search feature to enhance the library's accessibility and include detailed technique guides for executing exercises safely and effectively.

- **Implementation Methods:**
  - Developed the ExerciseList class to house the exercise library, offering detailed views for each exercise that include images, instructions, and common mistakes to avoid. A search function was implemented to facilitate easy navigation through the list of exercises.

- **Testing and Results:**
    - Usability testing was conducted to evaluate the effectiveness of the search feature and the quality of exercise technique guidance provided. The features were all very easy to use and navigate though.

- **User Feedback and Observations:**
    - Users expressed satisfaction with the depth of the exercise library and found the technique guides beneficial for learning new exercises or correcting form. Suggestions for improvement included adding filters for equipment availability and muscle groups targeted to implement

- **Challenges:**
    - The main challenge was to manually make every single exercise image, making sure they're the same size and show the exercise clearly. Then it took me a while to figure out how to add the search feature and generate the whole list without errors.
- **Conclusion and Improvements:**
    - The extensive exercise library and technique guidance objective were successfully met, enhancing the user experience by providing valuable exercise-related information. Future improvements could include implementing advanced filtering options and expanding the library to cover more niche exercises, as well as adding a video example of how the exercise is performed.

Objective 1.4.6: Workout Logging and History

- **Objective Description:**
    - This objective aimed to enable users to log various aspects of their workouts, including the type of workout (generated plan, custom plan, single workout), sets, reps, and weights for strength-based workouts, ensuring a comprehensive tracking system for user progress.
- **Implementation Methods:**
    - The LogWorkout class was created to facilitate workout logging, allowing users to enter details of their workouts, including exercises performed, sets, reps, and weights. The system was designed to store this data effectively, creating a historical record of user workouts for progress tracking.

- **Testing and Results:**
    - System testing was employed to ensure the seamless integration of the logging feature with other app components. Testing was also used to verify that data was stored correctly in the database. All tests confirmed the reliability and accuracy of the workout logging feature.

- **User Feedback and Observations:**
    - Users appreciated the straightforward and user-friendly interface for logging workouts. The feature was well-received for its comprehensiveness and the ease with which users could track their progress over time.

- **Challenges:**
    - One challenge was designing an intuitive UI that could accommodate the diverse range of information required for the workout logging. Ensuring data accuracy and consistency across different workout types also presented difficulties.I struggled a lot with screen displaying and adding methods to add an exercise and remove an exercise, as well as adding a set and removing a set.

- **Conclusion and Improvements:**
    - The workout logging and history objective were successfully achieved, providing users with a robust tool for tracking their fitness progress. I would've liked to add a way to display older sets of an exercise if it was found in the database as well if I had more time. Also, since I implemented this class towards the end of the project after coding with Kivy for months I noticed that I was able to write this much faster.

Objective 1.4.7: Secure User Authentication and Data Handling

- **Objective Description:**
    - The goal was to implement a robust and secure login system with hashed password storage to ensure user privacy and data security. Additionally, the objective included encrypting and securely storing all personal user data.
- **Implementation Methods:**
    - A secure authentication system was developed, incorporating password hashing and encryption techniques to safeguard user credentials and personal information. Testing was conducted to look for security flaws.
- **Testing and Results:**
    - Security testing was conducted to challenge the authentication system and data handling practices. The tests confirmed that the login system was secure, with hashed passwords and encrypted personal data providing a high level of security.
- **User Feedback and Observations:**
    - Users reported feeling confident in the security of their personal information within the app. There were no reported incidents of unauthorised access or data breaches, affirming the effectiveness of the implemented security measures.
- **Challenges**:
    - Understanding how the bcrypt module works and using it though mysql was the hardest part in this whole process.

- **Conclusion and Improvements:**
    - The secure user authentication and data handling objective was met, ensuring the protection of user data and maintaining trust in the app's security measures. I could improve this by adding the functionality that would allow users to sign in with their google, apple or microsoft account.

Objective 1.4.8: Interactive and User-Friendly Interface

- **Objective Description:**

    - This objective focused on leveraging the Kivy framework to develop an engaging, interactive, and user-friendly interface for the app. The aim was to design responsive elements and clear navigation pathways to enhance the overall user experience.
- **Implementation Methods:**
    - Utilised the Kivy framework's capabilities to construct a dynamic UI with responsive design elements tailored to various screen sizes and devices. Special attention was given to the intuitive layout of navigation elements, ensuring users could easily access all features of the app.
- **Testing and Results:**
    - Usability testing was conducted with a group of test users to identify any navigation issues or design flaws. The feedback was overwhelmingly positive, with users finding the interface engaging and easy to navigate. Minor suggestions for improvements were noted and addressed in subsequent updates.
- **User Feedback and Observations:**
    - Test users praised the aesthetic appeal and functionality of the interface, highlighting the smooth transitions and logical layout of information. Some users suggested adding customizable themes and modes to further personalise the user experience.
- **Challenges:**
    - The main challenge for this objective was to just learn the framework, as a beginner I didn't know how to even use python correctly which made this process painfully slow, I had to watch countless tutorials and look at hundreds of stackoverflow pages about people explaining how to do something really specific that i wanted to do, it took me really long to code the first simple classes like Input height and weight, Input body fat, prioritise muscle groups etc. I spent a lot of time actually designing how the app would look and trying my best to follow the design as much as I could.
- **Conclusion and Improvements:**
    - The objective of creating an interactive and user-friendly interface was successfully achieved, significantly enhancing user engagement and satisfaction. Future improvements could include more customization options for users and continuous refinement of UI elements based on user feedback. I definitely believe that if I had to do this project all over again I'd be much better prepared.

Objective 1.4.9: Comprehensive Tracking Features

- **Objective Description:**
    - The aim was to implement comprehensive tracking features that enable users to monitor their body measurements, workout progress, and dietary intake. The objective included the development of visual progress graphs and statistical overviews to provide insightful feedback.
- **Implementation Methods:**
    - Developed a suite of tracking features within the app that allowed users to input and monitor various health and fitness metrics. Advanced data visualisation techniques were employed to present this information in an accessible and meaningful way through progress graphs and statistics.

- **Testing and Results:**
    - Test data was entered to assess the accuracy and clarity of the tracking features and visual representations. The features performed well, providing accurate tracking and clear visual feedback on user progress. The graphs and statistics offered valuable insights into users' health and fitness journeys.
- **User Feedback and Observations:**
    - Users appreciated the comprehensive nature of the tracking features, noting that the visual progress graphs were particularly motivating. Some users requested additional customization options for the graphs and the ability to track more specific metrics.
- **Challenges**:
    - A significant challenge was ensuring the accuracy and reliability of the tracking features, particularly given the variety of data points and the complexity of visualising this information effectively. Balancing detail with simplicity to cater to both novice and advanced users was also a key consideration.

- **Conclusion and Improvements:**
    - The comprehensive tracking features objective was met even though I would've added more if I had more time providing users with valuable tools to monitor their progress. I wanted to implement graphs for caloric trends, workout strength progression and much more. Future improvements will try to finish this properly.

Objective 1.4.10: AI-powered Conversational Coach

- **Objective Description:**
    - This objective entailed developing an AI-powered conversational coach capable of providing workout tips, nutrition advice, and general motivation. The coach was designed to handle a variety of user queries with accurate and helpful responses.
- **Implementation Methods:**
    - Integrated an AI chatbot system using OpenAI's API, enabling dynamic interactions based on user queries. The chatbot was programmed to understand and respond to questions related to fitness and nutrition, offering personalised advice and support.
- **Testing and Results:**
    - A series of predefined queries were used to test the AI chatbot's response accuracy and helpfulness. The chatbot demonstrated a high level of understanding and provided relevant, motivational responses, although some responses were noted to be generic at times.
- **User Feedback and Observations:**
    - Users were impressed with the AI coach's ability to provide instant feedback and support, making the fitness journey more interactive and engaging. Suggestions for improvement included enhancing the personalization of responses and reducing response times.

- **Challenges:**

- Using the right framework and making the Kivy UI work was challenging, I wanted to add 'streaming' as well which is showing the text as it generates, while it was possible to do it in the terminal, I wasn't quite sure how to implement it in the app. Therefore I stuck with a traditional approach more and displayed a simple "Typing…" whenever the AI was answering.

- **Conclusion and Improvements:**
  - The AI-powered conversational coach was successfully implemented, adding significant value to the app by enhancing user engagement and providing personalised support. Future improvements could focus on increasing the depth of personalization, expanding the range of topics covered, and continuously refining the AI's understanding of user intent to improve response relevance and accuracy.

5.2 Final Thoughts

I think that even though I wasn't able to finish this app as a whole, this project gave Valuable insights on how to make a full stack application, backend and frontend, it taught me how to use SQL language, how to use different libraries like Kivy, Requests, Langchain, bcrypt, read through documentation, how to use OOP properly, it taught me also a lot about functional programming since I started using many smaller functions rather than a big one with a lot of if statements for example, it taught me how to use complex data structures like trees, hash tables, how to manage data in and out of a database and many other things, how important testing is… But most importantly it helped me develop my problem solving skills a lot. Going through my code again now I noticed that I could've done things differently, in a cleaner, and more efficient way and this project taught me that as well.

Appendix A

1. Results of online survey conducted on Instagram stories:

Question	Options	Percentage	N. of Respondents	Total Respondents
1. Do you use a fitness app?	-Yes, to track my nutrition and workouts	27%	182	674
	-Yes, just to track my nutrition	34%	229	
	-Yes, just to track my workouts	22%	148	
	-I don't use fitness apps	17%	115	
2. What is your primary goal in using a fitness app?	-Weight loss	29%	189	651
	-Muscle building	21%	137	
	-To build muscle and lose weight at the	14%	143	

	same time			
	-General fitness and wellness	24%	156	
	-Training for a specific event (e.g., marathon)	4%	26	
3. What features are most important to you in a fitness app? (Select all that apply)	-Customised workout plans	56%	359	641
	-Nutritional guidance and meal tracking	42%	269	
	-Progress tracking (e.g., weight, muscle gain)	58%	372	
	-Integration with wearable devices	37%	237	
	-Social sharing and community features	20%	128	
	-Gamification and motivational incentives	23%	147	
	-Educational content on exercise and nutrition	26%	167	
4. What is your biggest challenge or frustration with current fitness apps? (Select all that apply)	-Lack of personalization	48%	302	629
	-Difficulty in understanding or following workout routines	31%	195	
	-Inaccurate or unreliable tracking	26%	164	
	-Limited variety of exercises	13%	82	
	-Poor app interface or user experience	17%	107	
5. Would you prefer an app that offers...	-Only workout plans	12%	73	610
	-Only nutritional plans	26%	159	
	-Both workout and nutritional plans	62%	378	
6. What type of exercise content do you prefer?	-Video demonstrations	13%	79	605
	-Written descriptions	9%	54	

	-Interactive, with a coach	32%	194	
	-A mix of video and written content	46%	278	
7. What's your preferred method for receiving fitness advice and tips?	-Written articles or blog posts	18%	105	584
	-Video tutorials	35%	204	
	-Live sessions with a trainer	19%	111	
	-Using AI like ChatGPT	28%	164	
8. How would you rate your current level of fitness expertise?	-Beginner	32%	180	564
	-Intermediate	37%	209	
	-Advanced	21%	118	
	-Expert	9%	51	

2. Full Interview conducted in local gym:

(Abdullah, 35-year-old professional)

"Hi Abdullah. To start, what brings you to fitness apps?"

"Honestly, it's desperation [chuckles]. Between work and kids, I feel like a human hamster wheel. I'm tired, gaining weight, and that old knee injury keeps flaring up. I need something I can actually squeeze into my life and that won't get me sidelined again."

"Understandable. Tell me about your past workout experience."

"Sporadic at best. I used to lift weights back in college, but that feels like a lifetime ago. I want results, but safely. Those "no pain, no gain" days are behind me."

Makes sense. What's holding you back right now?

"Time is the biggest one. After work, I'm often on kid duty, not gym duty. Plus, with online stuff, it's hard to know if I'm even doing things right to avoid making my knee worse."

—----------

(Abby, 20-year-old college student)

"Abby, what sparked your interest in getting fitter?"

"Honestly? Social media. Seeing all those before-and-afters is so motivating, but also, like, overwhelming. I want to look AND feel better, but there's just SO much information out there, I end up paralyzed."

"Yeah, that's real. Have you tried anything before?"

"Dabbled a bit. Campus gym is kinda intimidating, and workout videos online get boring fast. I need, like, a personal cheerleader in my pocket who tells me exactly what to do, ya know?"

"Absolutely. What kind of things frustrate you most?"

"Not seeing results! Sometimes I go hard, then slack off for weeks. And I want to build a bit of muscle, but those bodybuilder programs seem intense...I just want something for me."

(Zak, 17-year-old college student)

"Hey Zak! What kind of fitness goals are you chasing?"

Dude, I just want to get bigger and stronger, plain and simple. My buddies are hitting the gym, and I wanna keep up, but also, like, have fun with it, not get stuck on the same treadmill every time.

"Right on. What have you tried so far?"

"Couple of workouts from, like, YouTube guys. But it's hard for me to stick with anything...gets kinda repetitive. And sometimes I'm crushing it, other days I just wanna mess around with those free weights, but not sure what to do exactly."

"Fair enough. What's the main thing getting in your way?"

"Probably just... knowing I'm doing it right. If I know I can make progress, I'll totally stick with it. Need something that, like, grows with me, so I don't get bored."

Appendix B

Program Code

The full program code can be found in my Github repository at https://github.com/Silmoon18/FitnessApp/blob/main/fitness%20app.py (main python file) and https://github.com/Silmoon18/FitnessApp/blob/main/fitness_app.kv (UI file in kvlang)