Development of a Fitness Mobile Application for Achieving Fitness Goals and Tracking Fitness Progress: Strategym

By

Saood Aslam

Student № B626182

Final Year Project: 21COC251

Department Of Computer Science

Loughborough University

May 2021

Acknowledgements

I would like to give a special thank you to my friends and family for supporting me consistently from the start to the completion of this project. I would also like to give a special thank you to Jane Rogers, who has supported me and provided me with guidance throughout the entirety of this year. Finally, I would like to give a special thank you to my supervisor Dr. Walter Hussak for the great amount of help he has given throughout the development of this project. Thank you all, your efforts are greatly appreciated.

Abstract

The fitness mobile application industry has seen a significant increase since the beginnings of the COVID-19 pandemic, and the annual increase in market size is predicted to continue growing after the end of the pandemic from 2022 to 2025.

With more and more consumers entering the world of fitness applications, the industry is ripe with opportunities that can be exploited by a high-quality application that delivers the wants and needs of the consumer. This project aims to tackle the task of developing a well-rounded fitness application that exploits the opportunities within the fitness application industry by emphasising a focus on these wants and needs.

The project itself covers areas including domain research of the fitness application industry, technological research with regards to the technologies surrounding mobile application development, solution planning based on the research, specifying requirements, design, implementation, testing, and an evaluation of the project output.

Contents

1 Introduction 16

1.1 Background 16

1.2 Aims and Objectives 16

1.3 Project Scope and Feasibility Study: 17

2 Literature Review 18

2.1 Introduction 18

2.2 Problem Domain 18

2.3 Gap Analysis 20

2.4 Technological Research 23

2.4.1 Mobile Application Type 23

2.4.2 Mobile Application Platforms 25

2.4.3 Mobile Application Tools 27

2.4.4 Mobile Application Databases 29

3 Solution & Planning 30

3.1 Introduction: 30

3.2 Fitness Domain 30

3.2.1 Convenience/Ease of Use 32

3.2.1.1 Practical Application 32

3.2.2 Customisation 39

3.2.2.1 Practical Application 39

3.2.3 Fitness Tracking Capabilities 39

3.2.3.1 Practical Application: 39

3.2.4 Personalisation 41

3.2.4.1 Motivation 41

3.2.4.2 Practical Application: 41

3.2.5 Planning How to Monitor Health and Fitness Status 44

3.2.5.1 Practical Applications: 44

3.3 Technological Domain 45

3.3.1 Platform 45

3.3.2 Type 45
3.3.3 Developmental Tool 46
3.3.4 Database 46
4 Developmental Methodology 49
4.1 Linear Approaches 50
4.1.1 Waterfall Model 51
4.2 Non-linear Approaches 51
4.2.1 Iterative Model 51
4.2.2 Incremental Model 52
4.3 Conclusions 53
5 Requirements 54
5.1 Profile-Side Specifications 55
5.1.1 Authentication 55
5.1.1.1 Necessary Requirements 55
5.1.1.2 Extended Requirements 55
5.1.2 Profile Data 55
5.1.2.1 Necessary Requirements 55
5.1.2.2 Extended Requirements 56
5.2 Fitness-Side Specifications 56
5.2.1 Necessary Requirements: 56
5.2.2 Extended Requirements 57
5.3 Diet-Side Specifications. 57
5.3.1 Necessary Requirements 57
5.3.2 Extended Requirements 57
6 System Design 58
6.1 Navigation Maps 58
6.1.1 Application Launch Cases 58
6.1.2 Authentication Flow: 59
6.1.3 Overarching Navigational Structure 60
6.1.4 Profile Section 61
6.1.5 Diet Section 62

6.1.6 Fitness Section 64
6.2 Wireframes *66*
6.2.1 Onboarding Screens: 67
6.2.2 Authentication Screens: 68
6.2.3 Profile Side 69
6.2.4 Diet Side 72
6.2.5 Fitness Side 76
6.2.6 Global Components: 83
6.3 Database Design *86*
6.3.1 Firebase 86
6.3.2 SQLite 87
7 Implementation & Testing 87
7.1 Testing Strategy: *88*
7.2 Configuration & Setup *89*
7.2.1 Development Environment 89
7.2.2 Initializing the Application: 89
7.2.3 Initializing Firebase: 90
7.2.4 NPM Libraries 91
7.2.5 Initializing Local Read/Write Permissions for SQLite 94
7.2.6 Initializing the Default Food Database 94
7.2.7 Initializing the Default Exercise Database 95
7.2.8 Initializing the Generated Workouts Database 95
7.3 Execution *96*
7.3.1 Increment 1: User Interface Design Implementation 96
7.3.1.1 Static UI: 96
7.3.1.2 Form UI 99
Animated UI: 100
7.3.1.3 Testing, Issues & Resolutions 103
7.3.1.4 Results Display 104
7.3.2 Increment 2: Navigation Implementation and Login Functionality Implementation 114
7.3.2.1 Application Launch Cases Implementation 114
7.3.2.2 Login Functionality Implementation 119

7.3.2.3 Navigating Between Screens in the Same Stack 124
7.3.2.4 Testing, Issues & Resolutions 125
7.3.2.5 Results Display 127
7.3.3 Increment 3: Profile Side Implementation 129
7.3.3.1 Authentication Forms Functionality 130
7.3.3.2 Firestore Forms Functionality 132
7.3.3.3 Additional Feature: Neural Network Bodyfat Percentage Predictor: 135
7.3.3.4 Testing & Resolutions 146
7.3.3.5 Results Display 149
7.3.4 Increment 4: Diet Side Implementation 152
7.3.4.1 Access to the In-App Food Database: 152
7.3.4.2 Creating Food Items: 153
7.3.4.3 Displaying and Deleting User Created Food Items: 155
7.3.4.4 Adding Foods to Today's Intake 158
7.3.4.5 Viewing Today's Intake (Pie Chart) and Viewing Intake Over Time (Line Graph) 160
7.3.4.6 Tracking Today's Food Intake Against Goal Calories and Macronutrients 161
7.3.4.7 Testing & Resolutions 166
7.3.4.8 Results Display 167
7.3.5 Increment 5: Fitness Side Implementation 171
7.3.5.1 Generation of a Workout Plan 171
7.3.5.2 Creating a Workout 172
7.3.5.3 Creating an Exercise 176
7.3.5.4 Displaying User-Created Workouts 176
7.3.5.5 Performing a Workout 177
7.3.5.6 View Workout Progress and Exercise Progress 180
7.3.5.7 Testing & Resolutions 185
7.3.5.8 Results Display 187
8 Final Project Evaluation 193
8.1 Final Project Vs. Requirements 193
8.2 Final Project Vs. Dimensions 193
8.3 Final Project Vs. Aims and Scope 195
9 Appendices 196

9.1 Testing: *196*
9.1.1 Increment 1: User Interface Design Implementation 196
9.1.2 Increment 2: Navigation Implementation and Login Functionality 201
9.1.3 Increment 3: Profile Side Implementation 203
9.1.4 Increment 4: Diet Side Implementation 205
9.1.5 Increment 5: Fitness Side Implementation 211
9.2 Neural Network Test Runs and Results *219*
9.3 Libraries List *219*
9.4 User Manual *221*
9.4.1 Setup: 221
9.4.2 Guide: 221
9.4.2.1 Onboarding Screens: 221
9.4.2.2 Login/Registration Screens 221
9.4.2.3 Profile Section Guide 221
9.4.2.4 Diet Section Guide 222
9.4.2.5 Fitness Section Guide 222
9.5 Program Listings *223*
9.6 Other Essential Files Locations: *226*
10 References 227

List of Tables

Table 2.1: Gap Analysis Table 23
Table 2.2: Mobile Application Type Comparison 25
Table 2.3: Mobile Application Platforms 26
Table 2.4: Mobile Application Tools 29
Table 2.5: Mobile Application Databases Comparison 30
Table 3.1: Activity Factor Coefficient 34
Table 3.2: Workout Date VS. Sets Performed 40
Table 7.1: Visual Studio Code Extensions 89
Table 7.2: Descriptive List of Main Packages Used In-App 94

List of Figures

Figure 1.1.1: The initial Gantt Chart 18
Figure 3.1: Dimensions Scales 31
Figure 3.2: BMR and TDEE Functions Pseudocode 34
Figure 3.3: Calculation Of Target Calories and Macronutrients Pseudocode 37
Figure 3.4: Sets Performed Per Workout Graph 41
Figure 3.5: Healthy and Unhealthy Bodyfat Percentage Ranges (ACE Fit, 2022) 44
Figure 3.6: Flow Diagram of the Main Thread and Async Thread 47
Figure 3.7: Flow Diagram of Firebase Asynchronous Function to Retrieve Data from Firestore 48
Figure 4.1 (Tawfiq, 2020) 49
Figure 4.2: Linear SDLC 50
Figure 4.3: Iterative SDLC (Rastogi, 2015, Figure 2). 52
Figure 4.4: Incremental SDLC (Kyeremeh, 2019, Figure 1) 53
Figure 4.5: Revised Gantt Chart 54
Figure 6.1: Navigational Flow of Application Launch Cases 59
Figure 6.2: Navigational Flow of Authentication Instances 60
Figure 6.3: Overarching Navigational Flow Between Application's Main Sections 61
Figure 6.4: Navigational Flow: Profile Section 62
Figure 6.5: Navigational Flow: Diet Section 63
Figure 6.6: Navigational Flow: Fitness Section 66
Figure 6.7: Firebase Client/Server Architecture 86
Figure 6.8: SQLite Client/Server Architecture 87
Figure 7.1: Firebase Initialization 90
Figure 7.2: Firebase Authentication GUI 91
Figure 7.3: Firebase Cloud Firestore GUI 91
Figure 7.4: NPM Install 92
Figure 7.5: Android Read/Write Permissions 94
Figure 7.6: Sample Custom Header Component Code 97
Figure 7.7: Sample Custom Header with Back Code 98
Figure 7.8: Sample UI Code With 3rd Party; Custom; In-Built Components and Styles 99
Figure 7.9: Code Snippet: Changing Input Field Focus 100

Figure 7.10: Top Tab Buttons Code 101
Figure 7.11: Animation Effect: Code Snippet 1: Initializing Variables 102
Figure 7.12: Animation Effect Code Snippet 2: Animation in the Main Render 103
Figure 7.13: Initial Application Launch Case 116
Figure 7.14: Code Snippet from authProvidor Component 117
Figure 7.15: Code Snippet: Setting the Profile Stack 118
Figure 7.16: Code Snippet: Initial Stack Set and Fitness Tab Association with Nav Stack 118
Figure 7.17: Code Snippet from Routes Component 119
Figure 7.18: Code Snippet: Function for Checking Valid Email Input 120
Figure 7.19: Code Snippet: Function for Outputting Error Message on Invalidated Form 120
Figure 7.20: Code Snippet: Validate User Then Run Login Function 121
Figure 7.21: Code Snippet: Async Register; Await createUserWithEmailAndPassword API; .then() Adjunction 122
Figure 7.22: Code Snippet: .catch(error) 123
Figure 7.23: Code Snippet: Logout Function in AuthProvider 123
Figure 7.24: Code Snippet: onPress Method For Rendered Logout Component on Profile Home Screen 124
Figure 7.25: Code Snippet: useNavigation Function on Profile Settings Button 124
Figure 7.26: Code Snippet: Custom Header Component with Back onPress Prop Method 125
Figure 7.27: Code Snippet: TDEE Calculation Screen with Cusom Header Component, Passing Profile Home As The Previous Screen Name 125
Figure 7.28: Screenshot Example of the Bottom Nav Bar Implemented Across All Screens in AppStack Main Application 127
Figure 7.29: Screenshot Example of Invalid Login/Reg Form Error Message Displayed 128
Figure 7.30: Screenshot of catch.error Alert If The Email is Already Registered 128
Figure 7.31: Firestore GUI Displaying New Document for Newly Registered User 129
Figure 7.32: Code Snippet: Update User Password Function in AuthProvider Component 130
Figure 7.33: Code Snippet: Update Password onPress Method on the Rendered Submit Button Component on the Profile Home Screen 131
Figure 7.34: Code Snippet: Show Message Function Inside the Email Form Validation Function 132
Figure 7.35: Input Data Flow for Firestore-Updating Forms 133
Figure 7.36: Code Snippet: Update User TDEE Function in AuthProvider 134
Figure 7.37: Code Snippet: On Press Method on Submit Button Component for Calculating and Updating User TDEE 135
Figure 7.38: Standardisation Formula for Neural Network Training 137

Figure 7.39: MAX Operator on the Age Column 137
Figure 7.40: Min Operator on the Age Column 138
Figure 7.41: Standardisation Formula On a Value in The Age Column 138
Figure 7.42: Implementing the Men's Standardised Dataset 139
Figure 7.43: Implementing the Training Config of the Mens Bodyfat Neural Network and Outputting the Trained Neural Network as a .JSON File 141
Figure 7.44: Implementing the Training Config of the Women's Bodyfat Neural Network and Outputting the Trained Neural Network as a .JSON File 141
Figure 7.45: Standardisation and De-Standardisation Functions (Men's Example) 144
Figure 7.46: Running the Trained Neural Network on User Inputs Function (Men's Example) 144
Figure 7.47: On Press Method For Submit Form Button To Execute Running User's Input on Trained Neural Network 145
Figure 7.7.48: Dietary Goal Also Re-Calculating TDEE 149
Figure 7.49: Firestore Document From User Filled Forms 149
Figure 7.50: Invalid Form Notification Example Screenshot 150
Figure 7.51: Bodyfat Percentage Prediction Success Alert 150
Figure 7.52: Incorrect Password Alert Example Screenshot 151
Figure 7.53: Statistics Tab with Breakdown Data of Users Metrics Screenshot Example 151
Figure 7.54: Code Snippet: Rendered FlatList for In-App Food Database 152
Figure 7.55: Code Snippet: Food Calories and Macronutrients for Each Item in FlatList 153
Figure 7.56: Code Snippet: Grams TextInput Component for Each Item in FlatList 153
Figure 7.57: Code Snippet: useEffect for Creating Table to Store User-Created Foods 154
Figure 7.58: Code Snippet: Function for Inserting Created Food Item Into Database Table 155
Figure 7.59: On-Press Method For Inserting Created Food Item On Pressing Green Basket Icon 155
Figure 7.60: UseEffect for Getting User Created Foods From SQL Table 156
Figure 7.61: Code Snippet: Rendered FlatList Displaying User Created Foods 156
Figure 7.62: Code Snippet: Delete Function For Deleting Created Food Item From SQL Table 157
Figure 7.63: Rubbish Icon On-Press Method with Alert and Delete Function Braced Inside 157
Figure 7.64: Code Snippet: Formatting the Date for SQLite Table Name 158
Figure 7.65: Code Snippet: Insert Data Function for Adding Food Item to Today's Intake SQL Table 158
Figure 7.66: Code Snippet: On-Press Method For Inserting Food Item Into Today's Intake With Alert Confirmation 159
Figure 7.67: Code Snippet: useEffect for Getting Today's Intake SQL Table 160

Figure 7.68: Code Snippet: Delete Function for Deleting Food Item From Today's Intake 161
Figure 7.69: Code Snippet: useEffect to Get Calories Column Example 161
Figure 7.70: Function To Add Items In Array 162
Figure 7.71: Function to Add Numbers in a String 163
Figure 7.72: Firebase Function to Get Goal Calories and Macronutrients from User's Firestore 164
Figure 7.73: Code Snippet: Calculation for Calories Remaining And Calories Array For Pie Chart 164
Figure 7.74: Code Snippet: Rendered Pie Chart Component Example 164
Figure 7.75: Code Snippet: Conditional Set Text Colour Function 165
Figure 7.76: Code Snippet: Example of Line Graph: Calories History Against Date Line Graph 166
Figure 7.77: Users Created Foods SQLite Table Opened in Visual Studio Code 167
Figure 7.78: Screenshot: Diet Home Screen With Today's Food Items Listed 167
Figure 7.79: Screenshot: In-App Food Database Rendered 168
Figure 7.80: Screenshot: Users Created Food Items SQL Table Rendered 168
Figure 7.81: Screenshot: Confirmation Alert For Adding Food Item to Today's Intake 169
Figure 7.82: Screenshot: Error Message Example for Invalid Form Input 169
Figure 7.83: Screenshot: Today's Totals Pie Chart Display 170
Figure 7.84: Screenshot: Daily Progress Over Time Graph Display 170
Figure 7.85: Code Snippet: Generated Workout Plan Rendered on Screen 172
Figure 7.86: Code Snippet: Formatting Workout Table Name and Passing as a Prop 173
Figure 7.87: Code Snippet: Importing the In-App Exercise Database and Filtering Only For Shoulder Exercises on Shoulder Ex List Screen Example 173
Figure 7.88: Code Snippet: UseEffect for Executing SQL Query to Get User Created Shoulder Exercises 174
Figure 7.89: Insert Data Function for Inserting Exercises to Created Workout 175
Figure 7.90: Code Snippet: Delete Function for Deleting Exercise from Created Workout 176
Figure 7.91: Code Snippet: Function for Inserting Created Exercise into User Created Exercises SQL Table 176
Figure 7.92: Code Snippet: UseEffect for Getting Saved Workout Tables from SQL Database + Delete Function for Deleting a Workout Table from SQL Database 177
Figure 7.93: Code Snippet: Getting the Workout Table Name Prop From Previous Screen and UseEffect For Getting the Data in the SQL Table 178
Figure 7.94: Code Snippet: useEffect for Creating Table for Today's Workout; Insert Data Function for Inserting Exercise Sets Performed Into Today's Workout Table; On-Press Method to Insert Sets On Pressing the Green Circular Tick Icon 179
Figure 7.95: Stop Watch Configuration Algorithm 180

Figure 7.96: Code Snippet: UseEffect Appended Variable Snipped for Getting Date Column; Rendered Line Chart Example Showing Date against Target vs Actual Sets 181
Figure 7.97: Code Snippet: Function for Displaying Conditional Statements Analysing User Workout Performance 182
Figure 7.98: OnPress Method Passing Exercise User Pressed On as Prop For Display Exercise Progress Screen 182
Figure 7.99: Code Snippet: Data Manipulation Functions For Exercise Data To Be Displayed 183
Figure 7.100: Code Snippet: Data Manipulation for Exercise Data To Be Displayed Part 2 184
Figure 7.101: Code Snippet: Functions for Finding the Max Weight Lifted and Displaying it On Screen 185
Figure 7.102: Code Snippet: Original Code for Filtering Out Everything Except Shoulder Exercises 186
Figure 7.103: Code Snippet: Resolved Code for Filtering Out Everything Except Shoulder Exercises 186
Figure 7.104: Screenshot: List of Shoulder Exercises to Add to Created Workout 187
Figure 7.105: Screenshot: Error Message on Inserting Invalid Sets/Reps for Adding Exercise to Created Workout 188
Figure 7.106: Screenshot: Alert Message on Adding Exercise to Created Workout 188
Figure 7.107: Screenshot: Created Workout Overview Screen 189
Figure 7.108: Screenshot: List of User's Created Workouts 189
Figure 7.109: Screenshot: Generated Workout Plan for User 190
Figure 7.110: Screenshot: Error Message on Inserting Invalid Data On Perform Workout Screen 190
Figure 7.111: Screenshot: Success Message on Adding Valid Set To Performed Workout 191
Figure 7.112: Screenshot: Completed Workout Overview Screen 191
Figure 7.113: Screenshot: Exercise Progress Screen Shoulder Press Example 192
Figure 7.114: Screenshot: Workout History Progress Screen 192

List of Equations

Equation 3.1: Basic Metabolic Rate 33
Equation 3.2: Total Daily Energy Expenditure 33
Equation 3.3: Goal Calories When Gaining Weight 35
Equation 3.4: Goal Calories When Losing Weight 35
Equation 3.5: Goal Macronutrients 36
Equation 3.6: Total Volume of An Exercise Performed 42
Equation 7.1: Sigma Function for No. of Workouts Per Fitness Goal 95
Equation 7.2: De-standardisation Formula For Bodyfat Percentage Output 138

1 Introduction

1.1 Background

Throughout the COVID-19 pandemic, the UK Government enforced restrictions that opened and closed gyms on many occasions, such as gyms closing for 4 months at the start of the pandemic (Institute for Government, 2021). This coincided with the fitness application industry experiencing significant market growth, growing by 50.5% from 2019 to 2020, which superseded that of previous years 2017 to 2018 and 2018 to 2019 (Statista, 2022). This growth was driven by the main motivators behind why individuals use fitness applications, including: a higher control of privacy, and the convenience fitness applications offer as opposed to hiring personal trainers or travelling to the gym (Joseph et al, 2020). The increase in the fitness application industry is projected to continue growing from 2022 to 2023 by 24.1%, and 2023 to 2024 by 19.8% (Statista, 2022).

There are 2 main categories of mobile applications within the fitness industry: diet-focused applications and workout-focused applications. Research conducted on determining which type is preferred found that, whilst workout-focused applications are preferred over diet-focused applications, a combination of both supersede applications that focus on individual categories. In addition, it was also found that fitness applications that provide personalized features and an easy-to-use interface help users the most in achieving tangible results in relation to their fitness goals (Joseph et al, 2020).

1.2 Aims and Objectives

In correspondence with the research highlighted in the background (1.1), the main aims of this project are to produce a fitness mobile application with an intuitive interface, that integrates both workout and diet features providing users with a centralised, personalisable profile where they can set and track their fitness goals and diet goals. Listed below is a breakdown of the objectives subsequent to these key aims:

1. Carry out a comprehensive literature review in the fitness application domain and in the technological domains surrounding mobile application development in order to determine the best approach for the project's exectution. Key outcomes of the research include:
    a. Identification of user experience in the fitness application industry
    b. Identification of any gaps in the fitness application industry
    c. Comparative analysis of mobile application development frameworks
    d. Comparative analysis of software development life cycle approaches

    e. Comparative analysis of front-end design principles and practices
    f. Comparative analysis of back-end databases fitting the features of the application
2. Construct a requirements and specifications document breaking down the functionality of the application in light of the literature review carried out.
3. Utilise design methodologies to construct the front-end of the application, producing a well structured house-style, user interface, and navigational flow.
4. Develop a fitness mobile application that aligns with the requirements, aims and objectives identified.
5. Carry out testing strategies to identify and resolve any bugs within the system to ensure the robustness of the end application.
6. Evaluation of the final project along 3 dimesions:
    a. The successes of the project
    b. The failures of the project
    c. Future considerations and potential improvements of the end application

1.3 Project Scope and Feasibility Study:

An initial Gantt Chart had been developed in order to determine the scope of this project and give a clear, visual representation of the timeline (Figure 1.1.1). An informal project diary had also been produced in order to log the processes undertaken throughout the project, record the time taken to complete processes and determine any potential changes that would need to be made to the initial Gantt Chart in terms of:

1. The length of time required to complete a process
2. Any processes that weren't initially considered but required throughout the development.

Task	Estimated Dates	Oct	Nov	Dec	Jan	Feb	Mar	Apr	May
Project Diary	Oct 4th – May 4th								
Analysis	Oct 4th – Oct 31st								
Research	Oct 4th – Oct 8th								
Requirement Spec	Oct 9th – Oct 20th								
Planning	Oct 21st – Oct 31st								
Design	Nov 1st – Nov 15th								
Mock Screens	Nov 1st – Nov 15th								
Implementation	Nov 16th – Mar 9th								
Front-End Design	Nov 16th – Nov 26th								
Backend Database	Nov 27th – Dec 10th								
Profile Section	Dec 11th – Jan 10th								
Diet Section	Jan 11th – Feb 10th								
Fitness Section	Feb 11th – March 9th								
Testing	March 10th – May 1st								
Front-End Design	March 10th – April 25th								
Functionality	March 20th – May 1st								
Fixes/ Improvements	April 1st – April 20th								
Front-End Design	April 1st – April 10th								
Functionality	April 11th – April 20th								
Deployment/Submission	May 2nd – May 4th								

Figure 1.1.1: The initial Gantt Chart

2 Literature Review

2.1 Introduction

This literature review covers all research conducted with regards to the health and fitness application domain of the project and the technological domains of mobile application development. Throughout the literature a review, a comparative analysis has been executed in order to determine the appropriate approaches to all aspects of the project.

2.2 Problem Domain

Research conducted on individuals using diet tracking applications found that all respondents disliked applications that required inputting copious amounts of data for tracking their caloric intake, and preferred applications that provide a means of motivation to the user to continue their weight loss journey (Solbrig et al, 2017). Additionally, a main challenge faced by users of diet tracking applications is the time-consuming

task of searching for caloric or nutritional information on the foods they consume (Luhanga, Hippocrate A. AKPA, Suwa and Arakawa, 2018).

Amongst 4 systematic reviews carried out regarding the efficacy of fitness applications in terms of causing positive behavioural change, a commonality was that most mobile fitness applications tend to lack an inclusion of key aspects of behavioural theory, which consequently makes them less efficacious in producing tangible behavioural change. These features that promote behavioural change that fitness applications lack include: firstly, objective outcome measurements with precise tracking information; secondly, personalized, goal-setting features, and thirdly, personalized, real-time feedback where the user is given feedback regarding their current status in relation to their fitness goals (HUMANIZE, 2018).

Further emphasising this point is an academic journal published by Francis Academic Press, UK, which found by conducting an analysis on user experience that users tend to pay more attention to the features of the application that offer personalized feedback with regards to their health and fitness data, especially when this data is provided in a visualized format. Furthermore, providing the user with a training plan that aligns with their workout goals was the second most important factor in enhancing user experience (Shen, 2022). Joseph et al (2020) suggest as a result from their survey conducted on 70 fitness application users that motivating the user to continue progressing towards their workout goals tends to increase the likelihood of continual use of the applications.

Whilst supporting the findings highlighted above regardinging the importance of progress tracking on behaviour, a report published by the University College London (UCL Interaction Centre, 2018) also emphasised the efficacy of autmation in fitness applications. Utilizing automation where possible as an alternative to the user manually entering or calculating data has a direct positive impact on user experience due to increasing the application's ease-of-use. In doing so, it also inhibits the likelihood of the user manually inputting or calculating inaccruate data, which would consequently lead to an inaccurate representation of their fitness levels.

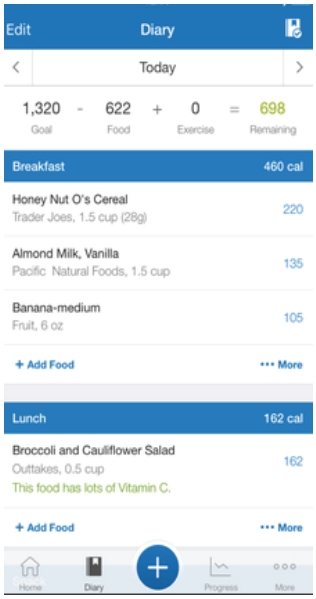
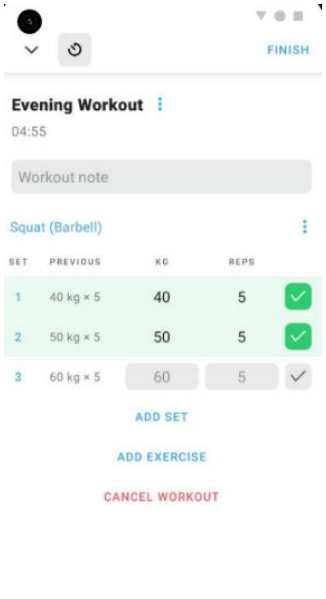
Many users have a negative perception of privacy violations due to fitness applications (Leijdekkers, 2015), which has the potentiality to sever the user-organization relationship. For example, Under Armour made headlining news when their diet tracking application MyFitnessPal experienced a data breach of 150 million users due to usage of an old encryption algorithm SHA-1 (Bonnington, 2018), and ultimately ended up selling the application in 2020 (Muoio, 2020).

2.3 Gap Analysis

The current market leaders in the fitness application industry that are direct competitors offering similar features include MyFitnessPal and Courtney Black Fitness (Great Britain: top Android health apps by revenue 2022 | Statista, 2022). A further application in the Google Play Store that would be in direct competition with Strategym is Strong Workout Tracker Gym Log.

An analysis has been performed on each of these 3 competitors to investigate their product offering, strengths, weaknesses, and any gaps in the market that can be exploited and captured. A month's worth subscription was purchased for each application in order to carry out the analysis on the full functionality available in each application (Table 2.1: Gap Analysis Table).

Company Name	**MyFitnessPal**	**Strong Workout Tracker Gym Log**	**Courtney Black Fitness**
Product Offering	Primarily a calorie and macronutrient tracker application with limited workout tracking functionality.	Workout logging and workout progress tracking application.	Workout logging, workout planner and calorie tracker application.

User Interface Design	 	 	 
Target Audience	Primarily individuals looking track their dietary progress and caloric intake.	Primarily gym goers looking to track their fitness progress.	Primarily individuals looking to track their fitness progress, dietary progress, and caloric intake.
Strengths	• Large food and nutritional database in-app	• Intuitive, highly responsive, and easy-to-use user interface	• Inclusion of features from both the fitness and diet side of fitness applications.

	• Stores user daily caloric intake history allowing the user to view their caloric, macronutrient and micronutrient intake to examine their progress. • Enhanced food searching abilities such as ability to scan a barcode and acquire nutritional information about a food source • Utilisation of Computer Vision to estimate nutritional and macronutrient composition of food items from photos of the food item. • Automated caloric and macronutrient intake calculations based on user measurements and user-set diet goal. • Includes functionality with regards to creating a workout routine and logging the workout routine	• Abundance of workout-centric functionality including ability to create custom workouts; create custom exercises; and perform workouts. • Includes functionality to view a visualised graph representation of exercise progress over time. • Vast database of hundreds of exercises provided in-app with demonstrative photos displaying how to perform the exercise as well as the muscle group being trained. • Motivational strategies used in-app such as notifications when the user performs a new strength personal record on an exercise.	• Large food and nutritional database in-app • Large exercise database in-app • Personalisation functionality: includes functionality to set a workout goal and also includes functionality to generate a personalised workout plan. • Aesthetically pleasing and intuitive user interface. • Visualised graph representation of workout progress and • Automated caloric intake and macronutrient calculations based on user measurements and user-set diet goal.
Weaknesses	• Security breach of 150 million users leading to a tarnished brand reputation as of 2018 (Bonnington, C, 2018). • Diet-centric → Only covers the diet and nutritional aspect of fitness, and has no features regarding the workout-side of fitness • Utilises a user-generated food database (rather than one from a professional body). This could lead to inaccurate	• Limited personalisation functionality: does not include functionality to set a workout goal or target sets/reps per workout. • Does not include functionality to generate a personalised workout plan based on a workout goal. • Workout-centric → Only covers the workout aspect of fitness and has no features or	• Female-centric: inclusion of motivational quotes and photos used throughout the app directed predominantly towards the female population. • Limitations on offline usage. Most in-app functionality requires a network connection, and the user is logged out when not connected to a network

	nutritional values of food items, thus leading to inaccurate caloric tracking. • Static workout section → no ability to perform workout; the user must edit the original workout routine and then log the edited version.	functionality regarding the diet-side of fitness. • Automated stopwatch as soon as the user begins a workout with no control to pause, stop, or reset the timer. • No representation of actual workout progress, only exercise progress.	• Lagging front-end when performing a workout that includes videos/photos, likely due to the retrieval of media over network. • Expensive: No free tier features and monthly price is £15.99 per month.

Table 2.1: Gap Analysis Table

2.4 Technological Research

This section is an overview of the research conducted regarding the factors to consider when deciding on the technologies to employ for mobile application development. There are 4 main decisions to make when deciding on the technologies to use for mobile application development which are highlighted in the subsections below.

2.4.1 Mobile Application Type

There exist 3 main types of mobile applications: native applications, hybrid applications, and progressive web applications. Each type utilizes a different set of tools, frameworks, and languages when building mobile applications which leads to different outcomes in terms of an application's design and functionality. The table below (Table 2.2: Mobile Application Type Comparison) displays what each type is, what they have to offer, and comparatively analyses their strengths and weaknesses.

Type:	**Native**	**Hybrid Web App**	**Progressive Web App (PWA)**
Definition	Applications that are tailored for use on a particular platform (Adelakun-Adeyemo and Olalekan, 2015). For example, an	Applications that are built using the web technologies JavaScript, HTML, and CSS with a viewport to adjust the size of each webpage to fit	Applications that are built using web technologies JavaScript, HTML, and CSS within a PWA framework that enables access to APIs

	application available on the Google Play Store that utilizes Android's native components.	the size of the mobile device (Khandeparkar, Gupta and B.Sindhya, 2015). The web application is then wrapped inside a native container that act as a browser inside an application such as Android's WebView component (Google, 2022).	on the native device (Carneiro, 2018)
Strengths	• High speed: when downloading a native application on a device, all of its native components are downloaded too which means they run instantly. • All features native to a particular device are accessible such as camera, GPS, geolocation. • Since features of the application are downloaded and installed onto the device, native applications can employ offline functionality.	• Inherently cross-platform. Web technologies can be functional in a web browser on a desktop, or in a WebView component of native devices. • Single codebase for all platforms. With the use of viewport, the web application adjusts its size for the device it's being accessed from, and the functionality is not device-specific, thus native codebases aren't required.	• Some native features accessible through APIs such as geolocation. • Inherently cross-platform. Web technologies can be utilized through a framework such as Ionic (Ionic, 2022) and accessed from a desktop web browser, or any device as a compiled application. • Single codebase for all platforms. Frameworks such as Ionic enable a single directory for all the code associated with the PWA whilst also providing APIs to access native components.
Weaknesses	• Not cross-platform. Developing an application using a native framework such as Android's SDK, would not be compatible	• All application functionality and components require a stable network connection. • Slower loading time compared to native applications due to	• Only functional on newer mobile devices that have built-in APIs that accept connections from PWAs. For example, old Android devices would not be

	for use on a device that runs iOS. • Can be costly to update an application that has multiple different versions native to each platform, since these updates would need to be applied to each version.	requirement of network connection. • No accessibility to native features such as the device's camera. The application is a pure web application with a viewport.	able to run PWAs (Kvist and Mathiasson, 2019) • Limited native features are accessible, not all. • Not supported by some web browsers. For example, PWAs are supported on Microsoft Edge but not Internet Explorer (MSEdgeTeam, 2022).

Table 2.2: Mobile Application Type Comparison

2.4.2 Mobile Application Platforms

There exist two major mobile applications that hold 99.8% of the mobile application market share; Android and iOS ((Masaad Alsaid et al., 2021). Mobile application developers have the option of either targeting one of the platforms or targeting both as a hybrid approach. The table below (Table 2.3: Mobile Application Platforms) displays what each platform has to offer and compares the strengths and weaknesses of each approach.

Platform:	**iOS**	**Android**	**Hybrid Platform Application**
Definition	Applications that are native to Apple's iOS operating system	Applications that are native to Google's Android operating system	Applications that have a native version for the Android OS as well as a native version for iOS.
Strengths	• Access to all native components of the platform device	• Access to all native components of the platform device	• Flexibility: These applications can be developed in separate codebases such as the

	• Inherits the advantages of native mobile applications.	• Inherits the advantages of native mobile applications. • Android application development can be performed on different desktop operating systems, including Linux, Windows, and Mac OSX.	iOS version developed in swift and the Android OS developed in Android's SDK, or they can be developed in a single codebase using a framework such as flutter (Flutter, 2019). • Inherits the advantages of native mobile applications.
Weaknesses	• An application developed for an Android device is not cross-compatible with other operating systems such as iOS. • Inherits the weaknesses of native applications. • Limited iOS development capabilities on anything other than a desktop OSX machine. For example, Xamarin is the only framework enabling iOS development on Windows (Microsoft, 2022) and it requires a physical iOS device, since iOS emulators are only available on Mac OSX.	• An application developed for an iOS device is not cross-compatible with other operating systems such as Android. • Inherits the weaknesses of native applications.	• Even in the case of a single code-base for an application native to different devices, configuration is still required to access different native components on different devices. For example, react-native requires platform-specific code in order for the application to be cross-platform (Meta Inc., 2022)

Table 2.3: Mobile Application Platforms

2.4.3 Mobile Application Tools

In terms of the tools, languages and frameworks on offer, the main native tool used to produce Android-specific applications is Android SDK released by the owners of Android, Google. The main native tool used to produce iOS applications is Swift, released by the owners on iOS, Apple. With regards to cross-platform developmental frameworks, the market leader is React Native, followed closely by Flutter (Sujay Vailshery, 2021). The table below (Table 2.4: Mobile Application Tools) covers what each tool is, what it offers, and what the strengths and weaknesses of it are.

Framework:	**React-Native**	**Flutter**	**Android SDK**	**Swift**
Definition	Cross-platform mobile application development framework (Meta Inc., 2022).	Cross-platform software development kit created by Google (Flutter, 2019).	Software development kit for Android application development (Google Developers, 2019).	Programming language specifically for iOS development (Apple, 2019).
Programming Language	JavaScript	Dart	Java; Kotlin	Swift
Platform	Cross-platform	Cross-platform	Android OS	iOS
Strengths	• Inherits all the strengths of hybrid mobile application platforms. • Due to Node.JS as the backend, react-native can take advantage of a large library of 3rd party libraries available in the node package manager (npm)	• Includes various 3rd party libraries to enhance development in the dart package library (Flutter, 2022). • Can also run on the browser as a web application and can be used to produce a desktop application (Flutter, 2022).	• Inherits the strength of the Android mobile application platform. • Programmable in multiple IDEs such as Visual Studio Code as well as the Android SDK IDE. • Java is a well-established object-oriented language	• Inherits the strength of the iOS mobile application platform. • Programmable in multiple IDEs such as Visual Studio Code as well as the XCode for Mac.

	registry (npm inc., 2022). • An extension of the React framework for web applications, thus react-native applications can also run in a browser on a desktop (Expo Cli, 2022). • Ability to use platform-specific UI components for the design of applications • Ability to scale up the application to another OS at any time.	• Ability to scale up the application to another OS at any time. • Includes it's own UI toolkit, thus no need to configure the UI components for each native device.	and is commonly used, thus has extensive documentation and guidance.	
Weaknesses	• Inherits the weaknesses of hybrid mobile application platforms i.e., configuration for platform-specific tools and access required.	• The dart package library is relatively small in comparison to the node package manager repository, which is reaching 2 million available packages (DeBill, 2022). • Less able to use platform-specific UI components for the design of applications. • Dart as a newer programming language that is less	• Inherits the weaknesses of the Android mobile application platform. • Since there are various devices released by various different companies that run on the Android OS, testing on all of these devices would need to be carried out in order to thoroughly ensure all features work on all models,	• Inherits the weaknesses of the iOS mobile application platform. • Swift is a newer, native programming language, that is less commonly used than languages such as Java used by Android SDK, thus it has less documentation and guidance.

		commonly used than languages such as JavaScript used by react-native, thus it has less documentation and guidance.	and this can be time consuming.	

Table 2.4: Mobile Application Tools

2.4.4 Mobile Application Databases

According to a survey conducted in 2022, Firebase, MongoDB's Realm, and SQLite are amongst the top mobile application databases in use (Mehta, 2022). In order to determine which database is most appropriate for storing data in the backend of Strategym, an analysis on each database has been conducted with regards to their nature, structure, offerings, security, and pricing (Table 2.5: Mobile Application Databases Comparison)

Database	**Firebase**	**Realm**	**SQLite**
Nature	Cloud Database Platform	Cloud Database	Local Database Engine
Structure (ie. SQL, noSQL)	JSON Document Store (stores data in JSON format in documents and collections)	Object Oriented model database (stores data as objects)	Standard SQL relational database structure
Offerings	• Backend-as-a-Service (BaaS). Firebase offers a backend stack, including services such as authentication, Firestore (Real-time Document Store), and Google storage buckets for storing multimedia files. • Offline support with asynchronous programming (Once connectivity is re-established after some	• Database-as-a-Service (DaaS). Offers objected oriented database with access control and role-based authentication • Offline support with asynchronous programming (Once connectivity is re-established after some interruption, the client device receives any changes it missed) (MongoDB, 2022)	• Offers a local SQL database engine tailored for local storage on mobile devices. • Exclusive for local use but can be synched to an external cloud database. • Authentication is optional (SQLite, 2022). • Lightweight and significantly faster than PostgreSQL, MySQL,

	interruption, the client device receives any changes it missed) (Google, 2022a) • Real-time Data synchronisation (Google, 2022c)	• Real-time data synchronisation (MongoDB, 2022)	and built-in file-system (SQLite, 2022a)
Security	• Authentication API • Data encryption complies with the 256-bit Encryption Standard (Google, 2019)	• Authentication API • Data encryption complies with the 256-bit Encryption Standard (uses AES-256 encryption). (MongoDB, 2022a).	• Authentication API • No intrinsic data encryption (Bricelam, 2021).
Pricing	Pay-as-you-go based on storage requirements. No cost up to 1GB with full functionality, then $0.108 per month per additional GB (Google, 2022d)	Monthly Subscription: first 30 days free then $30 a month for 2.5GB storage on the cloud. (MongoDB, 2022c).	Free. Open-source database engine developed in C (SQLite, 2022a).

Table 2.5: Mobile Application Databases Comparison

3 Solution & Planning

3.1 Introduction:

This section proposes a detailed solution in light of the research conducted in the literature review in section 2. The solution proposed provides a detailed plan for the requirements section and design section (section 5 and 6 respectively); a framework to evaluate the end-application (section 8); and identifies any technologies, equations, or pseudocode to be implemented in the implementation section (section 7).

3.2 Fitness Domain

This project attempts to resolve the issues mentioned in the problem domain and the weaknesses of similar applications mentioned in the gap analysis. There are 4 key dimensions that can be extracted from the research highlighted above by which the effectiveness of fitness applications can be measured upon. These key dimensions are convenience or ease-of-use; customisation; personalisation; and fitness tracking

capabilities. A scale has been constructed for each of these dimensions as a measure of the successfulness of the project (Figure 3.1: Dimensions Scales).

Figure 3.1: Dimensions Scales

3.2.1 Convenience/Ease of Use

There are 2 main factors that impact the ease-of-use of a fitness application, the first being the application's front-end and the second being the application's inclusion of functionality that automates processes and calculations.

With respect to the application's front-end, the important key factors that determine the success of the front-end across this dimension are the user interface design, and the navigational flow of the application. If the user interface design is well structured and clearly laid out minimising any potential confusion to the user, and if the navigational flow is intuitive, then this factor is rated as a 10 on the ease-of-use scale.

With respect to automating processes, the important key factor that determines its success across this dimension is its effectiveness at reducing data input from the user. If data input is limited only to where it is absolutely necessary, then this factor is rated as a 10 on the ease-of-use scale.

3.2.1.1 Practical Application

In terms of the diet-side of the application, automation involving the calculation of daily target calories and macronutrients; actual daily calories and macronutrients consumed; and current fitness levels can be implemented to replace data input.

There are two main formulas to calculate the maintenance calories for an individual: the Basic Metabolic Rate (BMR) formula and the Total Daily Energy Expenditure (TDEE) formula. BMR calculates the average amount of daily calories consumed by an individual's body to fuel essential functions at rest, and the equation used by clinicians to determine BMR is the Harris-Benedict equation (Frankenfield, Roth-Yousey and Compher, 2005). TDEE is an extension of the BMR calculation that accounts for an individual's activity levels in order to determine the average amount of daily calories required for an individual to maintain their weight given their lifestyle. For the Harris-Benedict equation, activity factor is categorized into 5 groups: sedentary; lightly active; moderately active; very active; and extra active.

The BMR equation (Equation 3.1: Basic Metabolic Rate), TDEE equation (Equation 3.2: Total Daily Energy Expenditure) and activity factor coefficient (Table 3.1: Activity Factor Coefficient) are given below (Tripathy and Saha, 2022).

$$\underline{\boldsymbol{BMR}}:$$

$$\boldsymbol{Mens}: 66.47 + \left(13.75 \times Weight(kg)\right) + \left(5 \times Height(cm)\right) - (6.75 \times Age)$$

$$\boldsymbol{Womens}: 655.1 + (9.563 \times Weight(kg)) + (1.85 \times Height(cm)) - (4.67 \times Age)$$

Equation 3.1: Basic Metabolic Rate

$$\underline{\boldsymbol{TDEE}}:$$

$$TDEE = BMR \times Actvity\ Factor$$

Equation 3.2: Total Daily Energy Expenditure

$$\underline{\boldsymbol{Activity\ Factor\ Coeffecient}}:$$

Label	**Coefficient**
Sedentary (little or no exercise)	1.2
Lightly Active (light exercise 1-3 days/week	1.375
Moderately Active (moderate exercise 3-5 days/week	1.55
Very Active (hard exercise 6-7 days per week	1.725
Extra Active (very hard exercise + physical job 6-7 days/week	1.9

Table 3.1: Activity Factor Coefficient

These equations require five inputs from the user: weight (in kilograms), height (in cm), age, biological sex, and activity level. The equations will be automated by implementing functions that takes these 4 inputs as parameters and performs the calculation internally. The pseudocode to illustrate this function is given below (Figure 3.2: BMR and TDEE Functions Pseudocode).

```
var BMR = null;
var TDEE = null;

function calculateBMR(weight, height, age, sex) {
    if sex is male then do {
        BMR = 66.5 + (13.75 x weight) + (5 x height) - (6.75 x age)
    }
    if sex is female then do {
        BMR = 655.1 + (9.563 x weight) + (1.85 x height) - (4.67 x age)
    }
}

function calculateTDEE(BMR, activityLevel) {
    TDEE = BMR x activityLevel;
}
```

Figure 3.2: BMR and TDEE Functions Pseudocode

Once the Total Daily Energy Expenditure has been calculated, the user's target calories and macronutrients can then be calculated. According to a paper published by the UK Government, 3500 calories is approximately equivalent to 1lb of bodyweight (UK Government, 2009), which is 7700 calories per kilogram of bodyweight. Thus, in order for an individual to gain 1kg of bodyweight, they would need to consume a total of 7700 calories above their TDEE, and in order for an individual to lose 1kg of bodyweight, they would need to consume calories totalling to 7700 below their TDEE. Given the ideal number of weeks the user would like to achieve their ideal weight, the total calories they would need to lose/gain to achieve that goal can be split per day in the timeframe given to calculate how many more/less calories they would need to consume to achieve their ideal weight by their target date. For example, person X is currently weighing 80kg, would like to weight 85kg in 10 weeks, and their TDEE has been calculated to be 2500 calories. For person X:

1. The total amount of weight they would like to gain is 5kg *(85 – 80)*
2. The total amount of calories equivalent to this weight is 38500 calories *(7700 x 5)*

3. The total amount of calories they would need to consume above their TDEE weekly is 3850 *(38500/10)*
4. Therefore, their daily caloric surplus is 550 *(3850 / 7)*
5. Thus, person X would need to consume 3050 calories *(TDEE + surplus)* per day in order to gain 5kg in 10 weeks.

The equations to calculate target calories have been constructed below (Equation 3.3: Goal Calories When Gaining Weight; Equation 3.4: Goal Calories When Losing Weight):

$$\underline{\textbf{\textit{Goal Calories: Gain Weight Equation:}}}$$

$$\textbf{\textit{Goal Calories}} = TDEE + (((weight\ to\ lose\ (kg)\ \times 7700) \div ideal\ number\ of\ weeks)\ \div 7)$$

Equation 3.3: Goal Calories When Gaining Weight

$$\underline{\textbf{\textit{Goal Calories: Lose Weight Equation:}}}$$

$$\textbf{\textit{Goal Calories}} = TDEE - (((weight\ to\ lose\ (kg)\ \times 7700) \div ideal\ number\ of\ weeks)\ \div 7)$$

Equation 3.4: Goal Calories When Losing Weight

The macronutrient intake of the user is dependent upon whether their dietary goal is to lose, gain, or maintain their weight. Instead of adding an extra input field for the user to define their dietary goal explicitly, this can be automated internally with a condition checking whether the ideal weight the user has set is higher or lower than their current weight. Following this, the target daily macronutritional intake can be calculated. A research paper investigating the ideal macronutritional intake for different dietary intakes in order to mitigate muscle catabolism (muscle waste) and promote a positive nitrogen balance for muscle anabolism (muscle gain) stated 3 key findings (Lambert, Frank and Evans, 2004):

1. When the dietary goal is to gain weight, the ideal amount of protein is 25% of the individuals ideal caloric intake, fats is 20%, and carobohydrates is 55%.
2. When the dietary goal is to lose weight, the ideal amount of protein is 30% of the individuals ideal caloric intake, fats is 15%, and carbohydrates is 55%.
3. When the dietary goal is to maintain weight, the ideal amount of protein is 25%, fats is 15% and carbohydrates is 60%.

According to the US Department of Agriculture (National Agricultural Library, 2022), in terms of the composition of macronutrients, protein has 4 calories per gram, carbohydrates have 4 calories per gram, and fats have 9 calories per gram. Thus, in order to calculate the required protein, carbohydrates and fats from the goal calories, this ratio of 4:4:9 (protein: carbohydrates: fats) would also need to be applied to get an accurate measure of the required target protein, fats, and carbohydrates (in grams) that would need to be consumed. The equation to calculate the target macronutrients have been constructed as (Equation 3.5: Goal Macronutrients).

$$\underline{\boldsymbol{Goal\ Macronutrients}}:$$

$$\boldsymbol{Goal\ Carbohydrates} = \frac{(goal\ calories\ (kcals)\ \times\ intake\ percentage\ (\%))}{4}$$

$$\boldsymbol{Goal\ Protein} = \frac{(goal\ calories\ (kcals)\ \times\ intake\ percentage\ (\%))}{4}$$

$$\boldsymbol{Goal\ Fats} = \frac{(goal\ calories\ (kcals)\ \times\ intake\ percentage\ (\%))}{9}$$

Equation 3.5: Goal Macronutrients

A function can be defined internally to automate this calculation and return the user's ideal caloric goal, protein, fats, and carbohydrates. This function would require 4 parameters: the user's current weight, ideal weight, the ideal number of weeks they desire to achieve their ideal weight, and their TDEE. The pseudocode to illustrate this function is given below (Figure 3.3: Calculation Of Target Calories and Macronutrients Pseudocode).

```
var goalCalories = null;
var goalProtein = null;
var goalCarbs = null;
var goalFats = null;

function calculateGoalCaloriesAndMacros(currentWeight, idealWeight, idealWeeks, TDEE) {
    if idealWeight is more than currentWeight then do {
        var weightToGain = idealWeight - currentWeight;
        var caloriesToGainWeight = weightToGain x 7700;
        var caloriesToGainWeekly = caloriesToGainWeight / idealWeeks;
        var dailyCaloriesSurplus = caloriesToGainWeekly / 7;
        goalCalories = TDEE + dailyCaloriesSurplus;
        goalProtein = (goalCalories x 0.25) / 4;
        goalFats = (goalCalories x 0.20) / 9;
        goalCarbs = (goalCalories x 0.55) / 4;
    }
    if idealWeight is less than currentWeight then do {
        var weightToLose = currentWeight - idealWeight;
        var caloriesToLoseWeight = weightToLose x 7700;
        var caloriesToLoseWeekly = caloriesToGainWeight / idealWeeks;
        var dailyCaloriesDeficit = caloriesToLoseWeekly / 7;
        goalCalories = TDEE - dailyCaloriesDeficit
        goalProtein = (goalCalories x 0.3) / 4;
        goalFats = (goalCalories x 0.15) / 9;
        goalCarbs = (goalCalories x 0.55) / 4;
    }
    else do {
        goalCalories = TDEE;
        goalProtein = (goalCalories x 0.25) / 4;
        goalCarbs = (goalCalories x 0.6) / 4;
        goalFats = (goalCalories x 0.15) / 9;
    }

}

```

Figure 3.3: Calculation Of Target Calories and Macronutrients Pseudocode

The inclusion of in-app databases is another aspect of the application that would improve the convenience and ease-of-use dimension. For example, rather than a user having to search the nutritional information of food items they have consumed and then inserting this data into their daily intake, it would be significantly more practical for the user to simply search for a food item in the application, select the food item they have consumed and then input the amount they have consumed. The US Department of Agriculture has an extensive database containing a variety of different food items and their associated nutritional information (FoodData Central, 2021), which is available for use by developers.

Additionally, an exercise database in-app would also improve the convenience of the application. The NHS categorise exercises into 7 muscle-groups: legs, back, abdomen, chest, shoulders, biceps, and triceps (NHS, 2022). Including exercises that cause the highest level of muscle activation in these muscle groups would be of most benefit to the user in achieving their workout goal. Studies have been conducted on participants where electrodes have been attached to the muscle group in question whilst performing a variety of different exercises for that muscle group. The findings are as follows:

- Shoulder exercises that cause the most muscle activation (Sweeney et al., 2014):
    - Shoulder press
    - 45-degree incline row
    - Seated rear lateral raise
- Chest exercises that cause the most muscle activation: (Schanke et al., 2012)
    - Barbell bench press
    - Pec deck machine
    - Cable crossovers
- Legs exercises that cause the most muscle activation:
    - Quads: Hack squat (lion, 2015)
    - Calves: Donkey calf raises (lion, 2015)
    - Glutes: Quadruped hip extensions (Anon, 2006)
- Back exercises that cause the most muscle activation: (Edelburg et al., 2018)
    - Pull-ups
    - Bent-over rows
    - I-Y-T raises
- Abs exercises that cause the most muscle activation: (SuppVersity EMG Series - Rectus Abdominis, Obliques and Erector Spinae, 2011)
    - Crunch
    - Hanging leg raises
    - Side bends
- Triceps exercises that cause the most muscle activation: (Boehler et al., 2011)
    - Triangle push ups
    - Triceps kickbacks
    - Dips
- Bicep exercises that cause the most muscle activation: (Young et al., 2014.)

- Concentration curls
- Cable curls
- Chin-ups

3.2.2 Customisation

Customisation refers to in-app functionality that enables the user to customise their complete profile. The inclusion of customisable features facilitates the personalisation dimension of the application since the user can personalise the application to their liking. Furthermore, restricting the user to only those exercises and foods that are provided in-app without the option to add or manipulative their own data would lead to an end-product that is inflexible and inadequate in comparison to similar applications in the fitness application industry that all offer customizable features.

3.2.2.1 Practical Application

Features that enhance the customisation of the application include: the ability to create and delete custom workouts, create, and delete custom exercises, create and delete custom foods; and create and manage profile details. Customisation also includes features that enable the user to customise the user interface of the application to their liking.

3.2.3 Fitness Tracking Capabilities

Fitness tracking capabilities are features that enable the user to view their fitness and dietary progress over time. As mentioned in the problem domain, objective outcome measurements with precise tracking information is considered the fundamental, underlying component of fitness applications that promotes behavioural change, especially when displayed in a visual format such as a graph.

3.2.3.1 Practical Application:

A database in the back-end of the application that stores a user's workout history and diet history can be utilised to analyse the user's workout and dietary progress over time. For example, a user would like to tack the number of sets they've completed per workout over time:

1. Every time the user performs a workout, the sets performed for that workout and the workout date are stored in a database along with all the previous workouts that the user has performed

2. When the user decides they want to track their progress, the data of all previous workouts is gathered from the database
3. The data is then split into 2 separate arrays (or data structures of the equivalent); the first storing the date of the workout and the second storing the number of sets completed.
4. The data is then displayed on a graph which takes the date array as the x axis and the sets array as the y axis

The database table stored in the back-end and the graph produced on the front-end for the example above is given below (Table 3.2 and Figure 3.4):

Workout Date	Sets Performed
01/01/2021	5
02/01/2021	8
03/01/2021	10
04/01/2021	4
05/01/2021	6
06/01/2021	5
07/01/2021	3
08/01/2021	7
09/01/2021	8

Table 3.2: Workout Date VS. Sets Performed

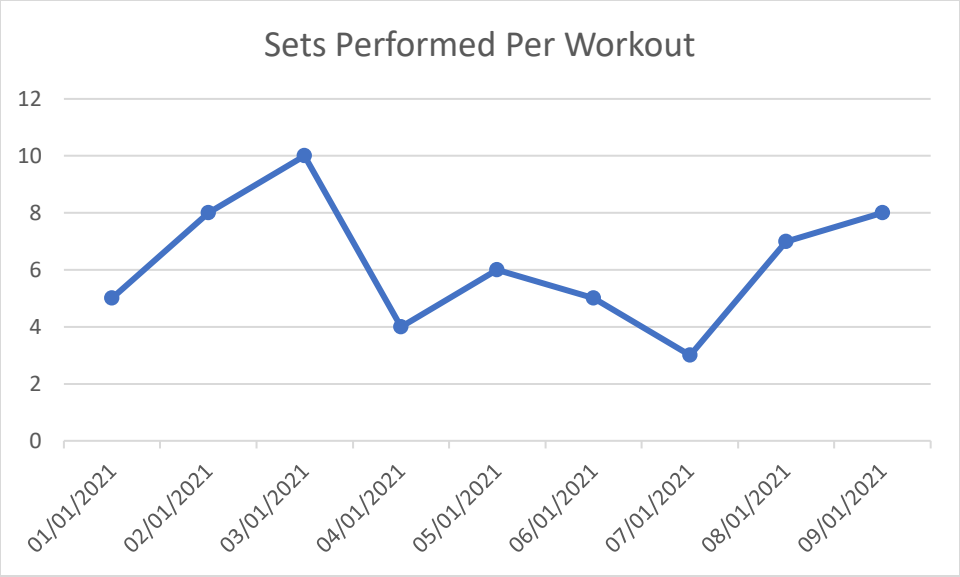


Figure 3.4: Sets Performed Per Workout Graph

3.2.4 Personalisation

Personalisation refers to functionality that personalises the application for each specific user. As mentioned in the problem domain, personalisation features are one of the most important aspects of fitness applications that promote behavioural change. Functionality such as generating workouts aligned with a user's workout-goal; generating target nutritional intakes aligned with the users' dietary goal; and providing personalised feedback in relation to the user's progress towards their goal are within this dimension.

3.2.4.1 Motivation

As mentioned in the problem domain, users of fitness applications tend to steer away from applications that are strictly statistical and instead have a strong preference for applications that provide a means of personalised, motivational feedback to the user to continue their fitness journey. Thus, motivation is an essential sub-category of personalisation. An example of a fitness application that utilizes motivational strategies is Strong Workout and Gym Log, which uses notifications to congratulate and inspire the user when they perform a new personal record for an exercise.

3.2.4.2 Practical Application:

There is an overlap between the convenience dimension and the personalisation dimension in that many personalisation features can be automated. As covered in the practical applications of automation to

enhance the ease-of-use dimension of the application (2.4.1.1), the fitness application can be personalised through functionality that calculates the specific user's BMR, TDEE, goal calories and goal macronutrients based on inputs regarding their physical measurements such as height, weight, and age, and based on their dietary goals.

Another feature to enhance the personalisation aspect of the application is through functionality that generates a workout plan for the user based on inputs regarding their workout goal and training frequency. For example, if a user sets their workout goal to build muscle and sets their training frequency to 3 workouts per week, then a workout plan can be generated which involves 3 separate workouts to perform with exercises and targets tailored to building muscle.

A meta-analysis on the subject of the optimal training frequency of muscle groups per weeks and hypertrophic (muscle-building) outcomes found that a frequency of training each muscle group twice a week promoted superior hypertrophic outcomes when compared to once a week (Schoenfeld, Ogborn and Krieger, 2016). Additionally, another meta-analysis with regards to hypertrophic outcomes and the optimal number of sets completed per week found that 10+ sets per muscle group per week led to superior hypertrophic outcomes when compared to a workout regimen consisting of less than 10 sets per muscle group per week (Schoenfeld, Ogborn and Krieger, 2017). In terms of the optimal number of reps per muscle group per week to promote muscle growth, current research suggests that a repetition range between 6-12 reps per set is optimal (Schoenfeld, 2010).

With regards to muscular strength, a meta-analysis released in the journal of Sports Medicine found that the main factor promoting muscular strength is the total volume performed per exercise per week (Grgic et al., 2018). The following equation is used to determine the total volume of an exercise performed:

$$Total\ Volume:$$

$$Volume = The\ Total\ Number\ of\ Sets\ \times Average\ Reps\ Per\ Set\ \times Average\ Weight\ Per\ Set$$

Equation 3.6: Total Volume of An Exercise Performed

For example, for the same exercise:

- Person X performs: 2 sets of 5 reps at 200kg and 1 additional set of 8 reps at 80kg

    - Volume = 3 x ((5 + 5 + 8)/3) x ((200 + 200 + 80)/3) = 7680
- Person Y performs: 2 sets of 3 reps at 200kg and 2 additional sets of 6 reps at 100kg
    - Volume = 4 x ((3 + 3 + 6 + 6)/4) x ((200 + 200 + 100 + 100)/4) = 8100
- In this scenario, person Y would achieve higher strength gains than person X due to a higher total volume.

In terms of the optimal number of sets per exercise per week to promote muscular strength, a meta-analysis found that medium (5-9 sets) or high (10 or more) sets per week improved strength significantly more than less than 5 sets performed per week (Ralston et al., 2017). Thus, the use of either medium-weekly-set training or high-weekly-set training would be appropriate to set when generating a workout for users who set their workout goal to muscular strength. Additionally, research has found that the optimum number of repetitions per set to promote muscular strength is between 3 and 9 (Berger, 1962).

Finally, for a user whose workout goal is muscular-endurance-focused, there is evidence supporting that 1-2 sets of low-resistance (meaning low weight) high-repetition exercises performed lead to a significantly higher improvement in muscular endurance, whereby the optimal number of reps lies between 30-40 reps if 2 sets are performed, or 100-150 reps on a single set performed (Anderson and Kearney, 1982).

Generating personalised workout plans using the optimal number of sets, reps, weight, frequency, and volume for each workout goal with the high muscle-activating exercises planned to be in-app, would lead to the optimal progress towards the user's set workout goal.

As mentioned previously, users of fitness applications value personalised, motivational feedback. This could be employed in a similar fashion to Strong Workout Log, whereby a user receives a personalised message when they achieve a personal record on an exercise. Furthermore, motivational, personalised feedback could also be employed with the fitness tracking capabilities of the application, whereby data is collected with regards to a user's workout history and is compared to the target number of sets, reps and weight of the exercises in the generated workout, and so using this the user receives a personalised message telling them if they are on track, falling behind or exceeding their targets in a motivational manner such as "You are currently on track, keep going!". The diet-side of the application could also employ motivational, personalised feedback, whereby the user's caloric/macronutrient daily intake value is compared to their targets and so using this data they receive a personalised message telling them if they are on track with their targets in a similar motivational manner.

3.2.5 Planning How to Monitor Health and Fitness Status

The NHS defines body mass index (BMI) as "the measure that uses your height and weight to work out if you're healthy" (NHS, 2019). However, there are issues when determining an individual's health status via their BMI calculation. The NHS give the example of heavyweight boxers, who are inaccurately classed as obese according to their BMI score, which is due to their body composition containing more muscle than fat. An alternative measure of an individual's health is their bodyfat percentage, which directly measures body composition and an individual's fat-free mass (Etchison et al., 2011).

The American Council on Exercise constructed the following table to determine the healthy and unhealthy bodyfat percentage ranges for men and women:

Description	Women	Men
Essential Fat	10-13%	2-5%
Athletes	14-20%	6-13%
Fitness	21-24%	14-17%
Acceptable	25-31%	18-24%
Obesity	>32%	>25%

Figure 3.5: Healthy and Unhealthy Bodyfat Percentage Ranges (ACE Fit, 2022)

3.2.5.1 Practical Applications:

Though bodyfat percentage is an accurate measure of an individual's health levels, the current methods to calculate it accurately are impractical. For example, a method to accurately calculate bodyfat percentage that is in current use is hydrostatic weighting which is weighing an individual's body density underwater (British Heart Foundation, 2020). However, conveniently there are datasets available that consist of people's bodyfat percentages (calculated through these accurate, impractical methods) against their physical measurements.

Fisher released a dataset consisting of the bodyfat percentage of 252 men in 1994 (through hydrostatic weighing) against various physical measurements such as age, height, weight, chest circumference, and waist

circumference (Johnson, 1996). Furthermore, Johnson (2021) released a dataset consisting of the bodyfat percentage of 184 women (also calculated through hydrostatic weighing) also against various physical measurements.

Both these datasets can be utilized by an artificial neural network for training purposes. This trained neural network can then be used to estimate a user's bodyfat percentage, based on inputs regarding the user's physical measurements. A well-trained neural network estimating a user's bodyfat percentage would give the user a better indication of their current health status without the inconvenience of the hydrostatic weighing method. This bodyfat percentage calculation can then be used to provide the user with personalised feedback regarding their current health status.

3.3 Technological Domain

3.3.1 Platform

In light of the technological research conducted in the literature review, the main two platforms that comprise the mobile application industry are Android and iOS. These platforms have a fairly similar number of users in the US and in the UK, with iOS being the market leader in the US by a less than 10% and Android being the market leader in the UK by less than 10% (Buchholz, 2020). Therefore, in order to reach as many fitness mobile application users as possible, it is best for Strategym to be a hybrid cross-platform application.

3.3.2 Type

In terms of the mobile application type, a hybrid web application is an impractical solution since access to any functionality in-app would require an active network connection, which may also lead to slower application performance since it would be dependent on the network connection speed. Furthermore, Progressive Web Apps are a newer technology that may not be supported on all iOS or Android mobile applications. Native applications on the other hand are tailored to support the operating systems they are made for, and don't require an active network connection since application components are downloaded and installed onto the user's device.

However, native iOS applications require Mac OSX for development, since Apple has made the developmental tools required only available on the OSX operating system. Since this developmental process is taking place on Windows 10 OS, for this project Strategym will be tailored towards Android OS with the

programmable ability to become cross-platform. In order to make this possible, a native, cross-platform mobile development framework is required for the implementation of the application.

3.3.3 Developmental Tool

In light of the mobile application tools researched, React-Native and Flutter are the two technological tools that support cross-platform development of native mobile applications. Both tools include the ability to scale up development to include another operating system at any time and various 3rd party libraries to enhance the developmental process. However, the dart package library used by Flutter is relatively small in comparison to node package manager used by react-native. Furthermore, Dart is a newer programming language with less documentation and guidance in comparison to JavaScript, therefore this may slow down the development process and result in a less optimal solution given the time-scale of this project. Therefore, the ideal framework to use for the implementation of Strategym is react-native.

3.3.4 Database

As covered in the fitness domain solution, Strategym requires a backend database to store user data such as a user's workout history, diet history and health metrics. It is essential to ensure that security protocols have been considered when selecting a database to reduce the likelihood of data breaches such as the one encountered by MyFitnessPal.

From the research conducted on the different databases available for use, Firebase has an authentication API as well as a data encryption protocol in place that complies with the 256-bit Encryption Standard. Therefore, Firebase is ideal for holding sensitive data collected from the user regarding their personal details and health metrics that may be deemed confidential such as the user's full name, weight, and age.

Firebase Firestore is a cloud database that utilises asynchronous programming (Google, 2022) whereby the asynchronous function returns a promise until resolved without causing a bottleneck that would prevent the rest of the code in the program thereafter to execute. This is beneficial as it means that in the case of a severed network connection, rather than the function failing when executed, it returns a promise and waits for a network connection to be re-established whilst also allowing the rest of the code in the program to execute. The diagram constructed below illustrates the flow of an asynchronous function from execution to resolution. The left-hand-side represents the main thread of the program, and the right-hand-side represents the asynchronous thread. As seen, when the asynchronous function is pending (known as await), the main thread is consisting of code outside of the asynchronous function is executed avoiding the bottleneck

problem. Then, when the async function resumes, the promise is resolved meaning that the function is complete.

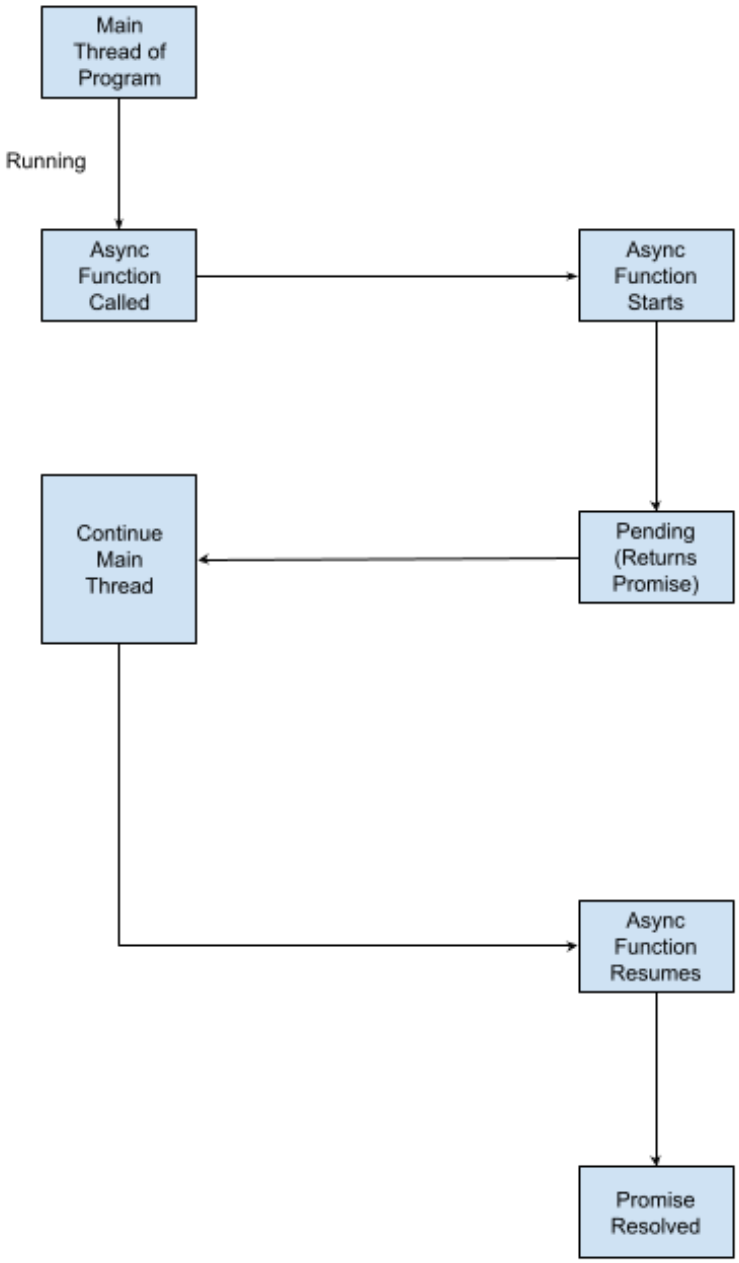


Figure 3.6: Flow Diagram of the Main Thread and Async Thread

Additionally, Firebase Firestore also supports offline persistence. Once data from the cloud database has been retrieved once, it is cached from the backend for offline access (Google, 2022). Then, if the user's

network connection is severed and they want to read their data stored in the cloud, this data is instead retrieved from the cache storage enabling the user to view it offline. The function to retrieve data from Firebase is also asynchronous thus when a network connection is re-established, the data function gets the actual data from the cloud database and updates the cache. The flow of this asynchronous function involving the cache is illustrated below (Figure 3.7).

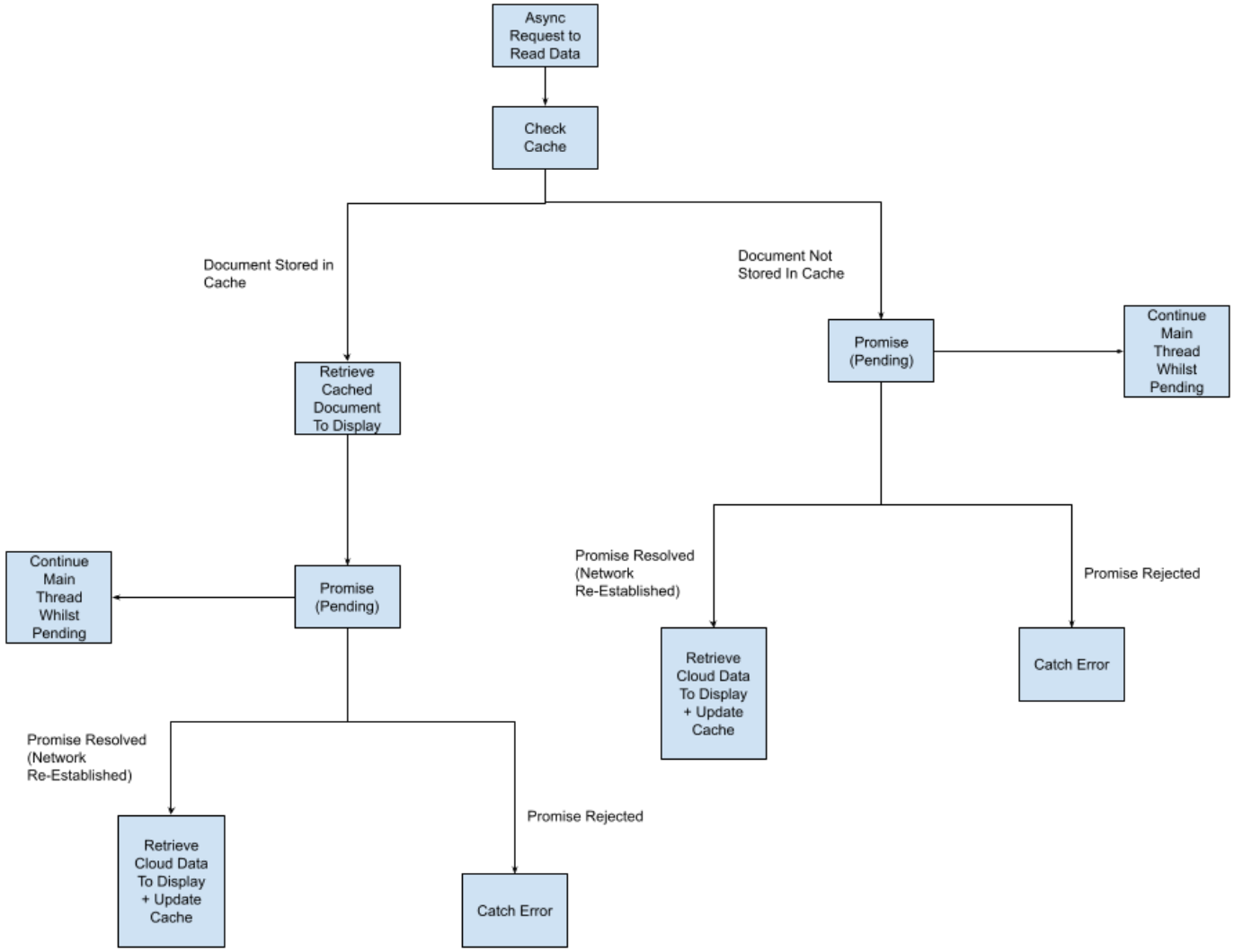


Figure 3.7: Flow Diagram of Firebase Asynchronous Function to Retrieve Data from Firestore

A drawback of Firebase's cloud platform is that it would be impossible to retrieve data offline if the data hasn't been cached. This could limit the functionality of in-app features that require the retrieval of data from a backend database. For example, if a user has their workout history stored in the cloud Firestore database

but does not have a copy of this data stored in cache, then they will not be able to access this data offline and so they will not be able to attain personalised feedback in relation to their workout progress. For use-cases as such where data is less sensitive (than a user's personal details), it would be beneficial to use a local database which uses the device's storage so that it's completely accessible offline. As covered in the literature review, SQLite is a local database engine tailored for mobile applications. It is a lightweight relational database that is faster than alternatives such as PostgreSQL and MySQL, and therefore doesn't compromise the user experience either. So, for use-cases where data will need to be stored locally, SQLite will be used.

4 Developmental Methodology

In the international journal of Computer Science and Information Technology, Rastogi defines a software development life cycle (SDLC) as "a well-defined and systematic approach, practiced for the development of a reliable high quality software system" (Rastogi, 2015). Essentially an SDLC defines the methodological approach to defining, planning, designing, implementing, and testing a software development project. The stages shared amongst all software development life cycles are as follows (Figure 4.1 (Tawfiq, 2020)):

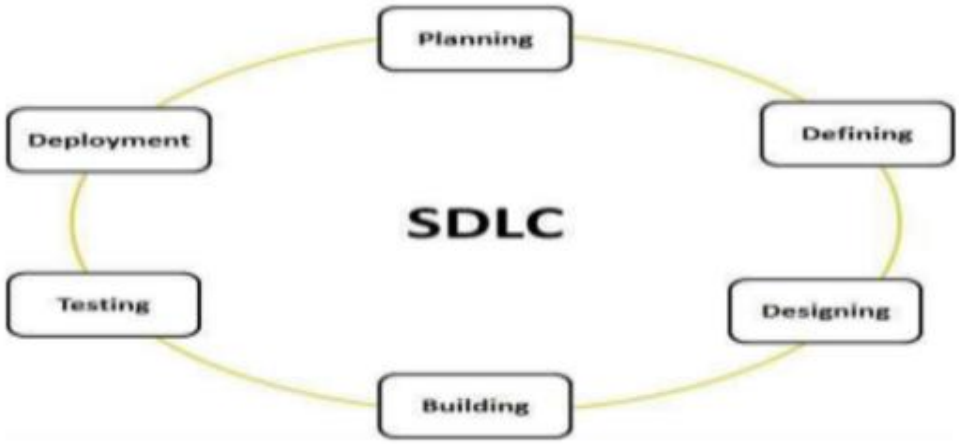


Figure 4.1 (Tawfiq, 2020)

The connections between these stages differ amongst different approaches. There are many different approaches for developing custom software such as mobile applications, and an analysis has been conducted on the approaches in order to determine the best fit for this project.

4.1 Linear Approaches

Linear software development models are sequential in nature, meaning that each next stage in the software development lifecycle occurs when the previous stage is completed in its entirety. For example, for the SDLC shown in Figure 4.1, a linear model would be (Figure 4.2: Linear SDLC):

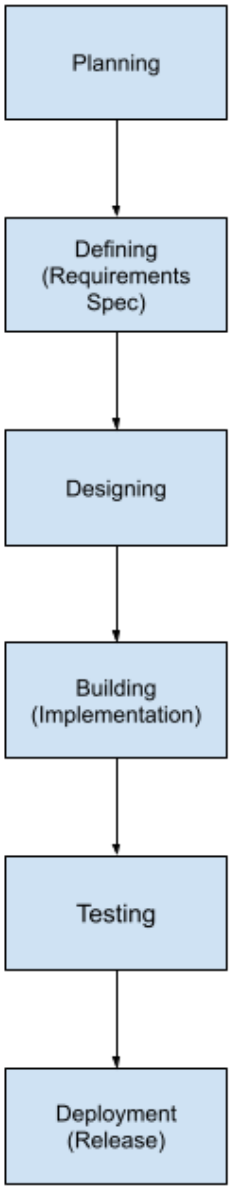


Figure 4.2: Linear SDLC

4.1.1 Waterfall Model

The waterfall model is a linear approach to software development whereby transition from one stage to the next occurs only when the prior stage is completed in its entirety, and the output of a previous, completed stage acts as the input for the next stage.

Due to the linear nature of the waterfall model, it does not allow for iterations or reflections to be made on previous stages. For example, once the design stage begins, the model leaves no room for re-visiting the requirements specification stage to make any necessary changes. This is impractical for real-life applications where the user requirements aren't known or well-defined from the inception of the project. Therefore, the waterfall model is suited for projects with well-defined, unchanging requirements.

4.2 Non-linear Approaches

Non-linear software developmental approaches often follow a cyclical flow, whereby a previous stage in the software development life cycle does not have to be completed in its entirety and can be revisited and updated at a later stage if needed. There are many models that employ a non-linear approach, amongst which include the iterative model and incremental mode.

4.2.1 Iterative Model

The iterative SDLC model is a non-linear approach to software development whereby the software is produced in iterations. A simplified build of the software is initially implemented, followed by enhanced builds that sequentially increase in complexity to meet the client requirements. An analogy that best describes this is considering each iteration from conception to release a mini-waterfall. This 'iterative builds' process is illustrated below (Figure 4.3: Iterative SDLC (Rastogi, 2015, Figure 2).).

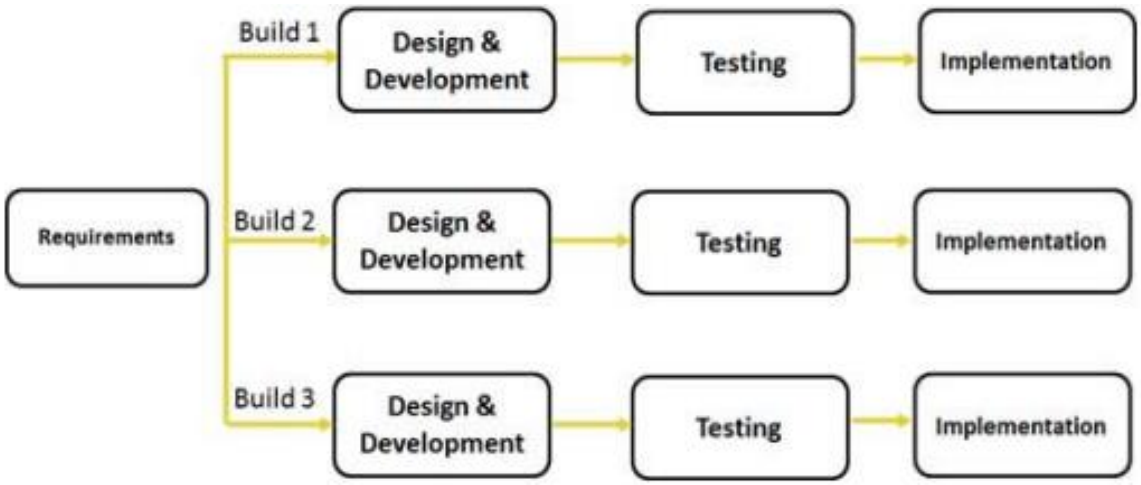


Figure 4.3: Iterative SDLC (Rastogi, 2015, Figure 2).

An advantage of the iterative model is that the client can provide feedback to developers early on in the process and the developers can make newer iterations to the software in response to the feedback received. This leads to an end-product that is likely to be more aligned with the client's requirements. However, a disadvantage of the iterative approach is that each iteration is a linear, sequential waterfall thus leaves no room for re-visiting a previous stage in an iteration, which could be costly since any changes that are needed to earlier stages can only be made in the next iteration after the implementation of the current iteration is completed.

4.2.2 Incremental Model

The incremental model is a non-linear approach to software development whereby components of a software are implemented in incremental cycles. Similar to the iterative model, these incremental cycles can also be considered as mini-waterfalls. The requirements specification is compartmentalised into components of the software, where each component can be implemented in an increment. After each increment, the client can analyse the current state of the software in terms of what has been implemented and give feedback to the developers with regards to any changes that need to be made. This incremental process is illustrated below (Figure 4.4: Incremental SDLC).

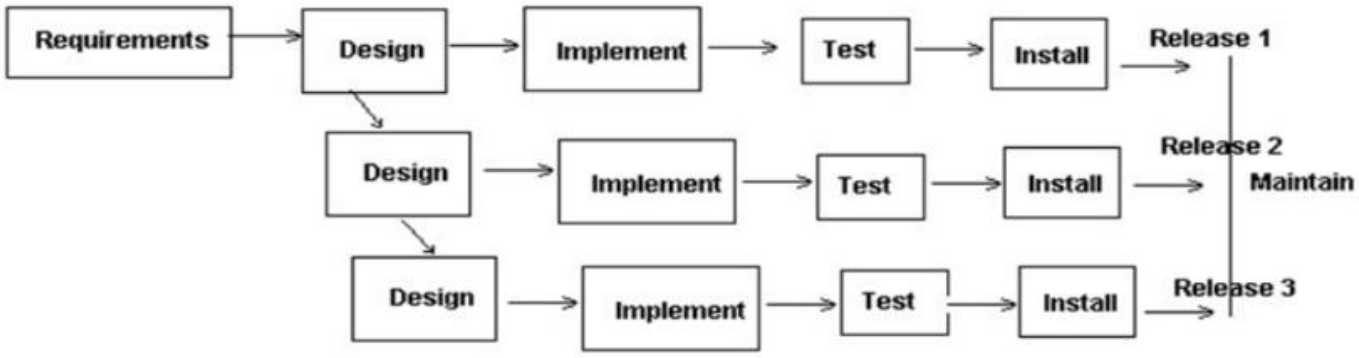


Figure 4.4: Incremental SDLC (Kyeremeh, 2019, Figure 1)

4.3 Conclusions

Given the nature and time frame of this project, the incremental SDLC is most appropriate for the developmental process. The requirements of the fitness application, though planned, are not well-defined as of yet and require flexibility to change when they are defined in section 5 and potentially during the implementation stage. Additionally, this fitness Strategym intrinsically consists of 3 main sections: a profile section, a fitness section, and a diet section. These 3 sections can therefore be compartmentalised into their own subsections in the requirements specification and can be incrementally implemented and tested.

The initial Gantt Chart has also been updated to coincide with the chosen software development life cycle as seen in Figure 4.5: Revised Gantt Chart. A testing stage has been added after each increment of the main sections of the application are implemented, and the estimated dates have been adjusted in order to fit this within the time frame of this project. The inclusion of multiple testing stages eases the process of debugging as, instead of testing and debugging the application in its entirety only at the end, each increment can be tested individually, which means that there is less code to debug during each testing stage, making it easier to locate and fix errors.

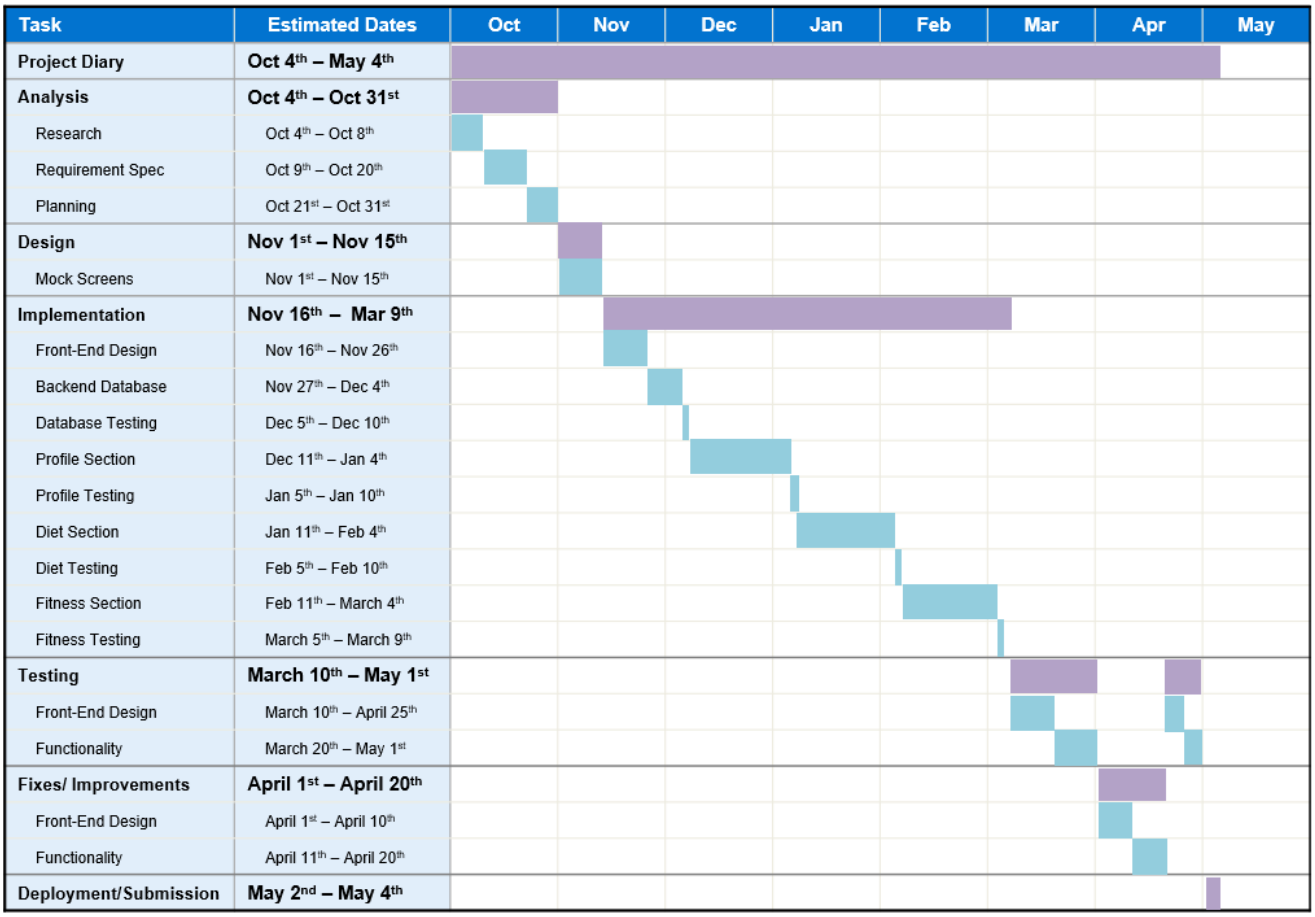


Figure 4.5: Revised Gantt Chart

5 Requirements

The following section covers the requirements for Strategym which have been established based on the research undertaken during the literature review and planning sections. These requirements are compartmentalised into the 3 sections of the application: the profile section, the diet section, and the fitness section, which are planned to be incrementally implemented. For each section the requirements are divided as either necessary requirements or extended requirements. Necessary requirements are those that *must be* implemented in the application, whereas extended requirements are those *should be* implemented if the project time frame enables to do so.

5.1 Profile-Side Specifications

5.1.1 Authentication

5.1.1.1 Necessary Requirements

- Users must be able to register a new account
- Users must be able to login to their existing account
- Users must be able to log out of their account
- User authentication credentials must be secured
- Only the user and no one else but the user must have access to their own profile details

5.1.1.2 Extended Requirements

- User should remain logged in after logging in initially, even when closing and re-opening the application
- Users should be able to change their login email
- Users should be able to change their login password
- Users should be able to troubleshoot their password if forgotten

5.1.2 Profile Data

5.1.2.1 Necessary Requirements

- Users must be able to input their physical attributes and retrieve calculations regarding their BMR and TDEE. The physical attributes required as input from the user must be:
    1) Age
    2) Height
    3) Weight
    4) Activity Level
- Users must be able to change their physical attributes and retrieve updated calculations regarding their BMR and TDEE in real time.
- Users must be able to choose a fitness goal they would like to achieve, with the following options:
    1) Muscular hypertrophy (muscle-growth)
    2) Muscular strength
    3) Muscular endurance

- Users must be able to input how often they train per week
- Users must be able to retrieve a personalised fitness plan generated based on their set fitness goal and training frequency inputs
- Users must be able to input an ideal weight they would like to achieve
- Users must be able to input an ideal date to achieve their ideal weight
- Users must be able to retrieve the daily caloric and macronutrient intake recommendations based on their ideal weight and ideal date to achieve their ideal weight inputs.
- Only the user and no one else but the user must have access to their own profile data.

5.1.2.2 Extended Requirements

- Users should be able to input their physical attributes and retrieve an estimation of their bodyfat percentage based on a neural network trained on the dataset of physical attributes against bodyfat percentage outcomes.
- Users should be recommended a dietary goal based on the results of their bodyfat percentage calculation.
- Users should be able to research their health outcomes using an NHS search API
- Users should be able to upload progress photos and view a timeline of their progress

5.2 Fitness-Side Specifications

5.2.1 Necessary Requirements:

- Users must be able to create, save and delete exercises
- Users must be able to create, save and delete workout plans
- Users must be able to perform workouts that are generated based on their fitness goal
- Users must be able to perform their own created workouts
- Users must be able to perform their own created exercises
- Users must be able to view the progress they are making in terms of the exercises they perform
- Users must be able to view their workout progress over time in a visual format
- Users must be told whether they are on track, falling behind, or exceeding their fitness goal.

5.2.2 Extended Requirements

- Users should be given positive feedback in the form of motivation with regards to their workout progress
- Users should be able to display the progress they have made for specific exercises on a graph57
- Users should be able to set strength goals for specific exercises
- Users should be notified with a motivational message when these strength goal for specific exercises are achieved

5.3 Diet-Side Specifications

5.3.1 Necessary Requirements

- Users must be able to view, add and delete food items from their daily intake
- Users must be able to create, save and delete food their own food items.
- Users must be able to view their daily caloric and macronutrient intake against their target daily caloric and macronutrient intake in a visual format
- Users must be told whether they are on track, falling behind, or exceeding their caloric and macronutrient intake
- Users must be able to view their food intake history and overall progress they are making towards their dietary goals in a visual format
- Users must be told whether they are on track, falling behind, or exceeding their dietary goal based on their food intake history.

5.3.2 Extended Requirements

- Users should be able to access a food database API from a professional body and add foods from this database into their daily intake.
- Users should be recommended food items based on their previous food intake.
- Users should be given positive feedback in the form of motivation in regard to their dietary progress

6 System Design

This section covers the front-end design, back-end design, and navigational flow of the fitness application, in light of the planning and requirements covered in sections 3 and 5.

6.1 Navigation Maps

The navigational structure of Strategym consists of 5 main parts: application launch cases, the authentication flow, the profile section, the fitness section, and the diet section. Each part encompasses its own navigational structure, as well as an overarching navigational structure illustrating the interconnections between each part.

6.1.1 Application Launch Cases

The application launch cases specifies the potential cases for when the user launches the application.

The first case is the first launch after the application has just been installed. Here, the user is first navigated to the onboarding screens where they will be given a brief summary of what the application has to offer.

The second case is any succeeding app launches where the user is not already logged into their account. Here, the user is navigated to the login/registration screen where they will first have to either log in or register in order to reach the profile section of the application.

The third case is any succeeding app launches where the user is already logged into their account. Here, the user is navigated directly to the main screens of the application.

The following diagram represents the navigational flow of each case (Figure 6.1: Navigational Flow of Application Launch Cases

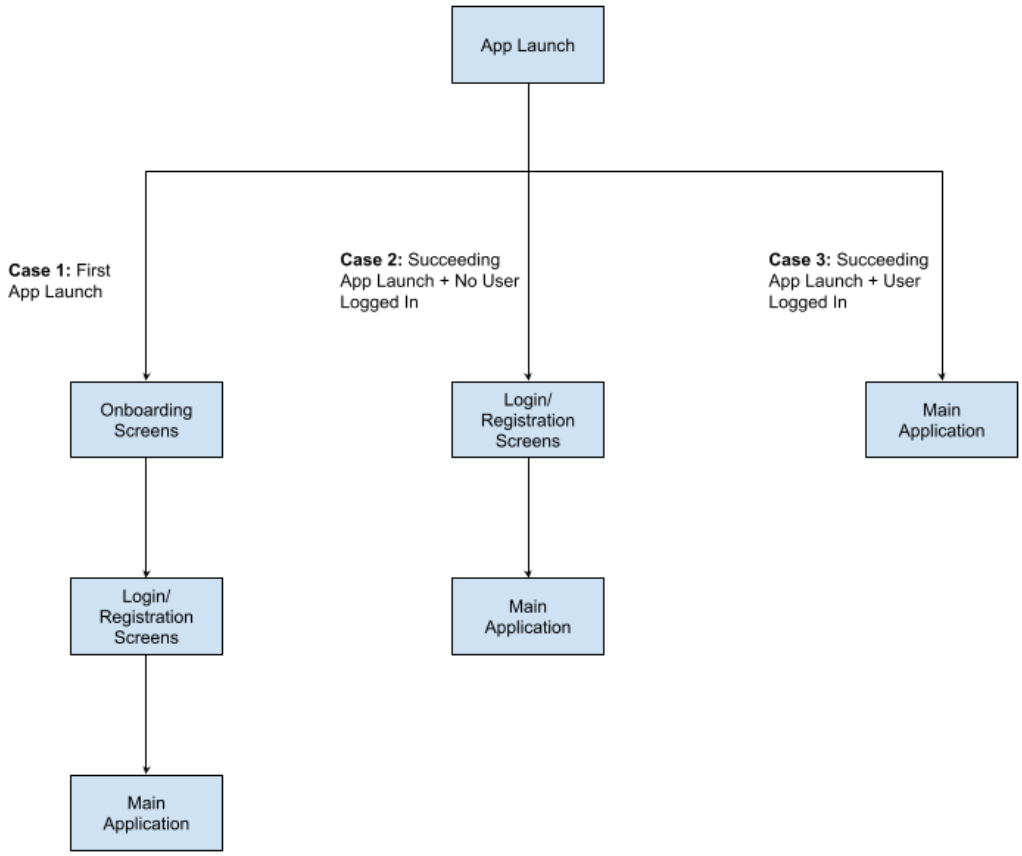


Figure 6.1: Navigational Flow of Application Launch Cases

6.1.2 Authentication Flow:

The authentication flow specifies the navigational mapping with regards to user authentication instances such as user login, user register, and user logout.

Once the user reaches the login/registration screen, they navigate between each form by swiping horizontally. Then, by pressing the corresponding button (login or register), they are logged into the application and have access to the main application. From here, they can then navigate to their profile screen where a logout button is placed, which, when pressed, logs the user out of the system and navigates them back to the login/registration screen (Figure 6.2: Navigational Flow of Authentication Instances).

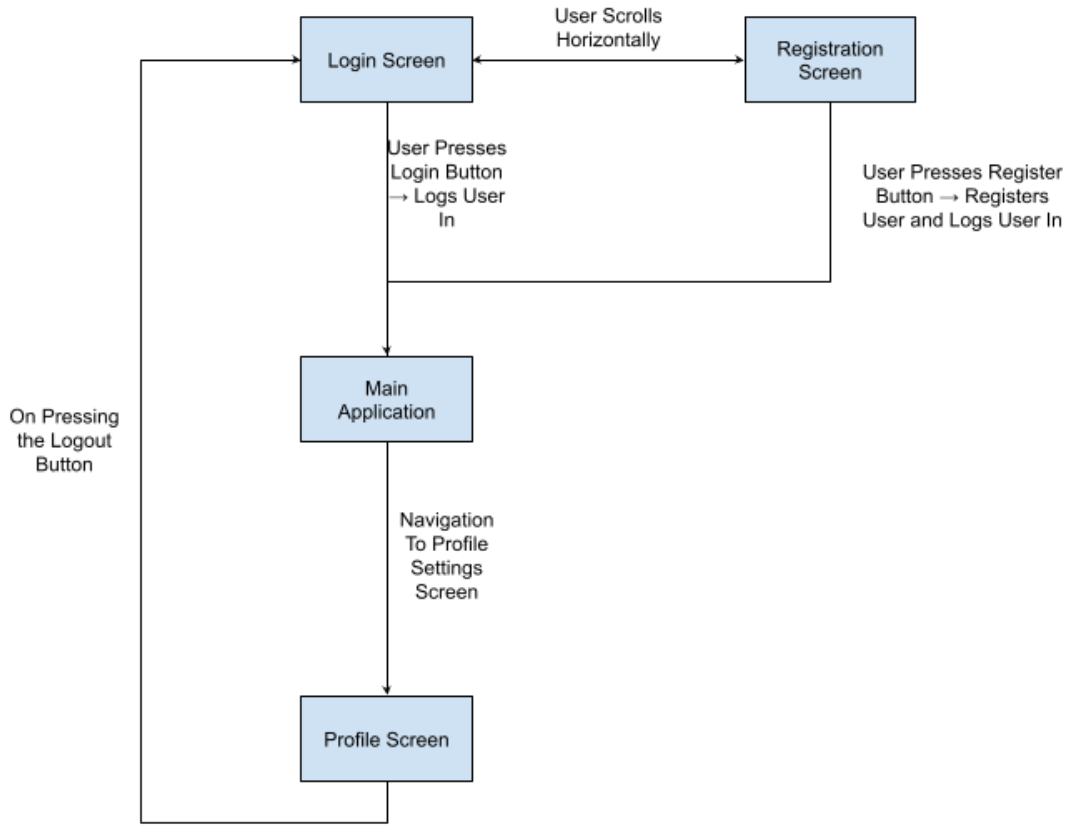


Figure 6.2: Navigational Flow of Authentication Instances

6.1.3 Overarching Navigational Structure

In order for the user to navigate between the main sections of the application (i.e., the profile section, diet section, and fitness section), they have access to a bottom navigation bar which displays icons representing each of the different main sections. When in one section, the user can navigate to another section by pressing on the corresponding icon in the bottom navigation bar.

The following diagram represents the overarching navigational structure between the different sections of the application. The top level of each tree is the section the user is currently on, and the bottom levels represent the sections the user can navigate to using the bottom navigation bar (Figure 6.3: Overarching Navigational Flow Between Application's Main Sections).

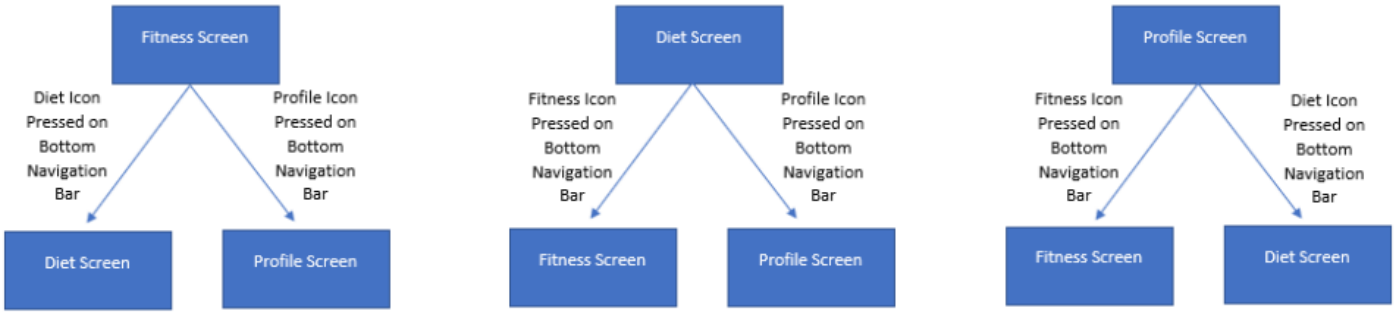


Figure 6.3: Overarching Navigational Flow Between Application's Main Sections

6.1.4 Profile Section

In light of the necessary requirements for the profile section defined in section 5 of the report, the profile section consists of the following screens:

Level 0 Screens:

- Profile home screen → The main screen of the profile section enabling the user to navigate to all the level 1 screens in the section.
- Profile statistical overview screen → Displays the users health and fitness metrics including their TDEE, BMR, dietary goal and fitness goal

Level 1 Screens:

- Profile settings screen → Where the user is able to update their profile settings such as their name and email.
- TDEE calculation screen → Where the user can input, save, and update data required for calculating their TDEE
- Fitness goal screen → Where the user can input, save, and update their fitness goal
- Dietary goal screen → Where the user can input, save, and update their dietary goals

The profile home screen is the start point of this section. It is where the user is navigated to after pressing on the profile icon in the bottom navigation tab. From the profile home screen, the user navigates to all other screens aforementioned in the profile section. The navigation tree for this section is displayed in Figure 6.4:

Navigational Flow: Profile Section. The screens at the lower levels of the navigation tree include a back button in their header to access the previous screen at the higher level of the navigation tree.

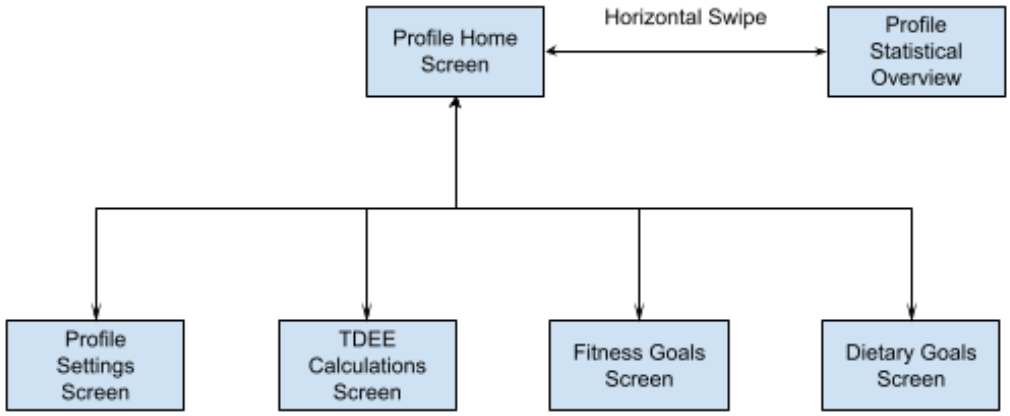


Figure 6.4: Navigational Flow: Profile Section

6.1.5 Diet Section

In light of the necessary requirements for the diet section defined in section 5 of the report, the diet section consists of the following screens:

Level 0 Screens:

- Diet home screen → The main screen of the diet section which shows the user the foods they've added to their daily intake today, and also enables the user to navigate to:
    - The 'overall dietary progress' screen
    - The 'dietary progress today' (i.e. the day they are using the application) screen
    - The 'add foods' screen

Level 1 Screens:

- Add foods screen → This screen displays a list of the default foods database included in-app, and also enables the user to navigate to:

    - The create foods screen
    - The users saved foods screen
- Overall dietary progress screen → Displays a visual representation of the users daily caloric and macronutrients intake history against their target calories and macronutrients
- Dietary progress today screen → Displays a visual representation of the users caloric and macronutrient intake today against their target daily calories and macronutrients

Level 2 Screens:

- Create foods screen → Where the user can input and save data for a food item they wish to create
- Users saved foods screen → Where the user can view a list of the food items that they have created and saved and add any food items in the list to their food intake for today.

The diet home screen is the start point of this section. It is where the user is navigated to after pressing on the diet icon in the bottom navigation tab. The navigation tree for this section is displayed in Figure 6.5: Navigational Flow: Diet Section. The screens at the lower levels of the navigation tree include a back button in their header to access the previous screen at the higher level of the navigation tree.

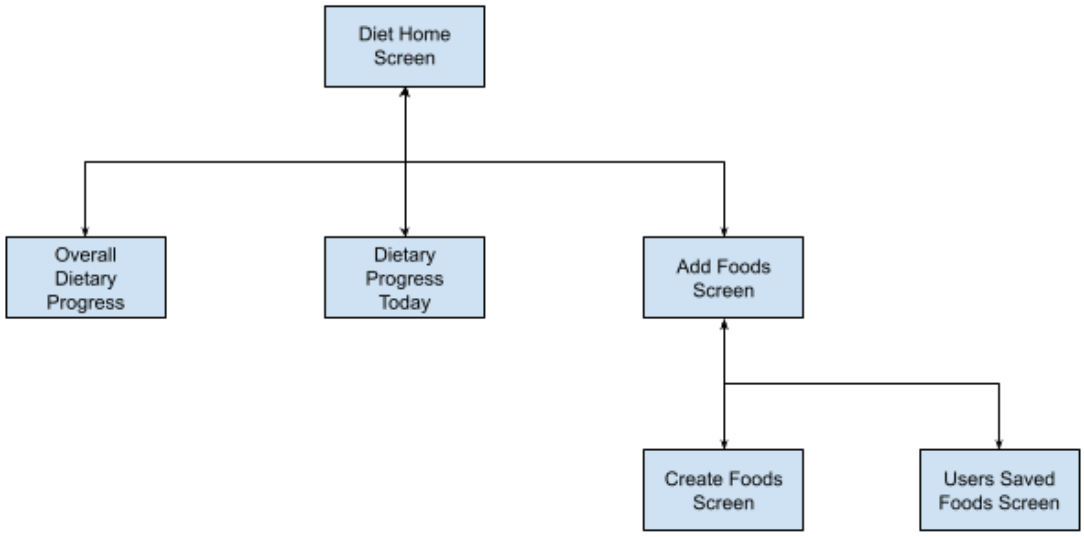


Figure 6.5: Navigational Flow: Diet Section

6.1.6 Fitness Section

In light of the necessary requirements for the fitness section defined in section 5 of the report, the fitness section consists of the following screens:

Level 0 Screens:

- Fitness home screen → The main screen of the fitness section enabling the user to navigate to the following screens in this section:
    - The create a workout screen
    - The begin a workout screen
    - The show overall workout progress screen
    - The search exercises for progress screen

Level 1 Screens:

- Create a workout screen → Where the user initialises the workout they wish to create by setting a name for the workout. The user can then navigate to the following screen:
    - Search exercises screen
- Begin a workout screen → Displays a list of workouts generated from the user's set fitness goals as well as a list of the user's created workouts. The user can select a workout to perform from the list and is then navigated to the:
    - Perform a workout screen
- Show overall workout progress screen → Displays a visual representation of the user's workout history progress against their target workout goals.
- Search exercises for progress screen → Displays a list of muscle groups that are pressable. In correspondence with the muscle group the user presses, they are then navigated to one of the:
    - Screens for each of the muscle groups (exercise progress route)

Level 2 Screens:

- Search exercises screen (create a workout route) → Displays a list of muscle groups that are pressable. In correspondence with the muscle group the user presses, they are then navigated to the screen for that muscle group. From this screen they can also navigate to the created workout overview screen or create an exercise screen. These next-level screens are listed as:

    - Screens for each of the muscle groups (create a workout route)
    - Created workout overview screen
    - Create an exercise screen
- Perform a workout screen → Displays a list of the exercises in the workout that the user has chosen to perform. The user is able to set a timer when they begin their workout, input the number of reps and the weight they have performed for the exercise, and once they have completed their workout, they can navigate to the:
    - Performed workout overview screen
- Screens for each of the muscle groups (exercise progress route) → Displays a list of the exercises corresponding to the muscle group the user pressed on the previous screen. The list of default exercises in-app are concatenated with the user-created exercises to form one list. After selecting an exercise in the list, the user is navigated to the:
    - Displayed exercise progress screen

Level 3 Screens:

- Screens for each of the muscle groups (create a workout route) → Displays a list of the exercises corresponding to the muscle group the user pressed on the previous screen. The list of default exercises in-app are concatenated with the user-created exercises to form one list. The user can select an exercise in the list to add to the workout they are creating.
- Created workout overview screen → Displays a list of all of the exercises, target sets and target reps the user has added to their created workout. Here they can delete any exercises they added to their created workout and save their created workout.
- Create an exercise screen → Where the user can input and save data for an exercise they wish to create
- Performed workout overview screen → Displays a list of the exercises, reps, sets, and weights the user has performed in the workout. Here they can delete any exercises they added to the list of exercises the performed in the workout.
- Displayed exercise progress screen → Displays a visual representation of the reps, sets, and weights the user performed against the target sets, reps, and weights for the corresponding exercise.

The fitness home screen is the start point of this section. It is where the user is navigated to after pressing on the fitness icon in the bottom navigation tab. The navigation tree for this section is displayed in Figure 6.6:

Navigational Flow: Fitness Section. The screens at the lower levels of the navigation tree include a back button in their header to access the previous screen at the higher level of the navigation tree.

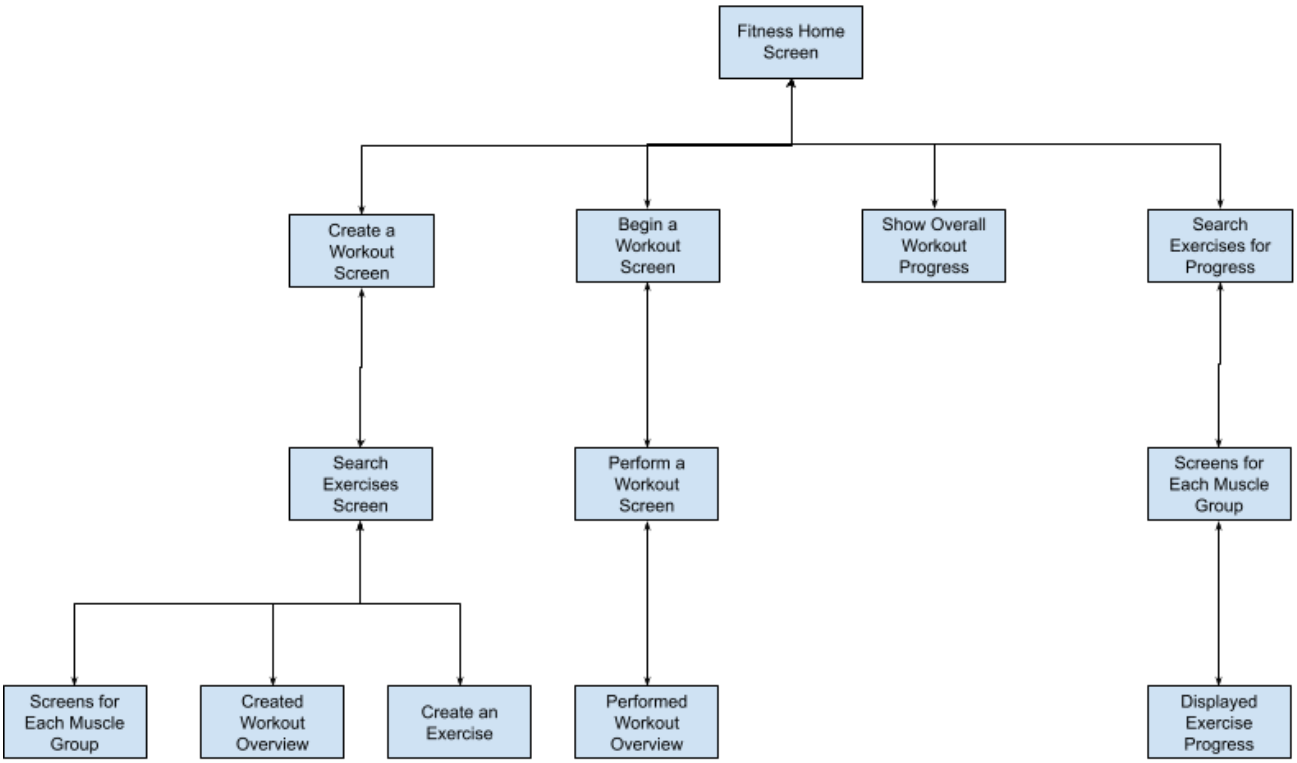


Figure 6.6: Navigational Flow: Fitness Section

6.2 Wireframes

Wireframing is as essential aspect of prototyping the user interface design of a software application. The US Department of Health & Human Services define wireframes as a "basic visual interface guide that suggests the structure of an interface and the relationships between its pages", "without the distraction of colour". The department also define 5 key elements to define for each page: navigation; company logo; content area sections, search function, user log in areas (U.S. Department of Health & Human Services, 2013). The following subsections display the wireframes of each individual screen within each section, as well as providing a description of each element displayed in the wireframe and the navigational routes associated with any buttons or icons.

6.2.1 Onboarding Screens:

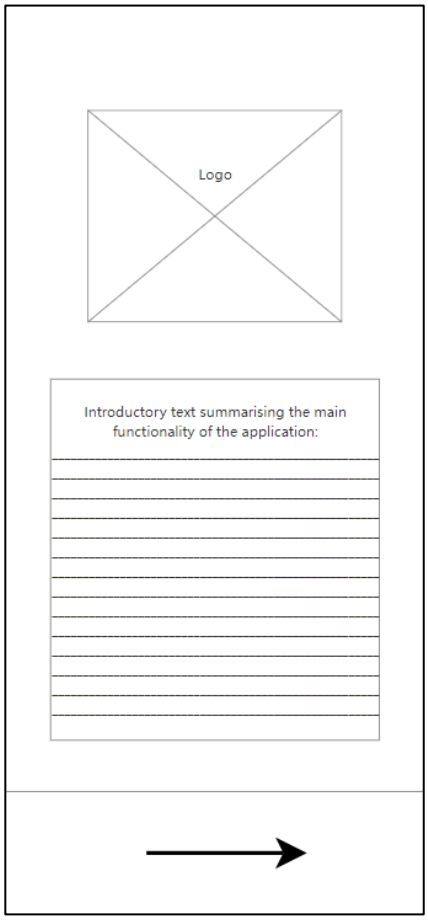


Figure 6.7: Onboarding

<u>Onboarding (Figure 6.7):</u>

Components:

- Logo Image
- Text Box Introducing the application to the user, giving a description of the main functionality regarding the fitness capabilities, dietary capabilities, and profile capabilities of the application.
- Next Arrow Icon button: On press → navigate to the login/registration authentication screens

6.2.2 Authentication Screens:

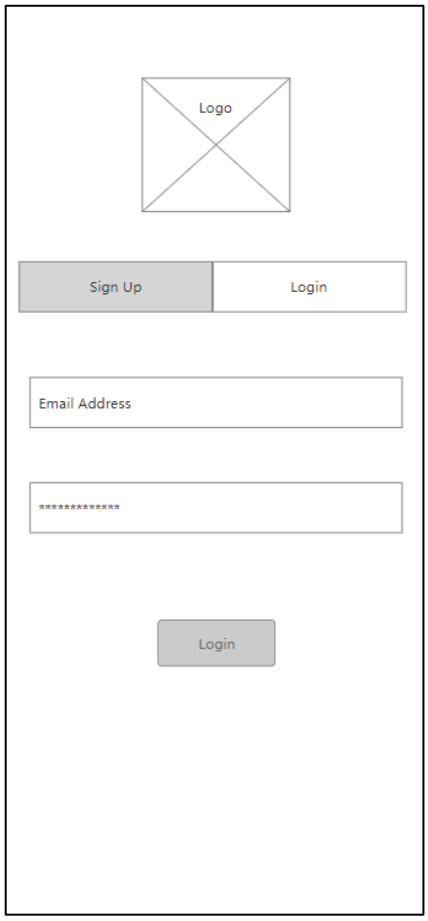


Figure 6.8: Login

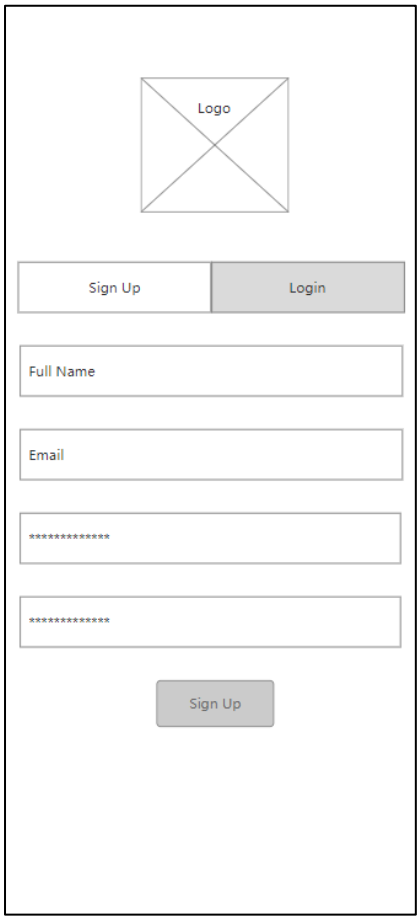


Figure 6.9: Sign Up

Login (Figure 6.8):

Components:

- Logo Image
- Sign Up/Login Tabs: On Press Sign Up → Navigate to the Sign-Up Form
- Email; password input fields for user login details
- Login button: On press → Validate user login details; navigate user to main application.

Sign Up (Figure 6.9):

- Logo Image
- Sign Up/Login Tabs: On Press Login → Navigate to the Login Form
- Name; email; password; confirm password input fields for user registration details
- Sign Up button: On press→ Validate user registration details; navigate user to main application.

Animations:

- Scroll Right → Navigate to the Sign-Up form
- Scroll Left → Navigate to the Login Form

6.2.3 Profile Side

Level 0 Screens:

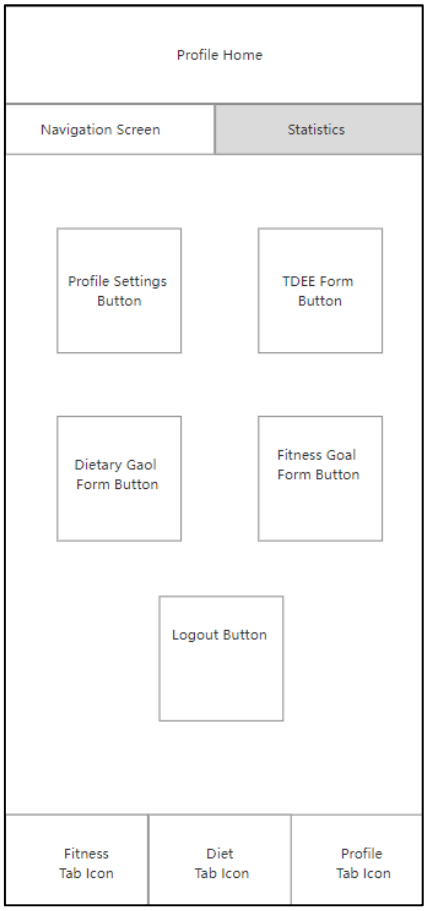


Figure 6.10: Profile Home: Navigation

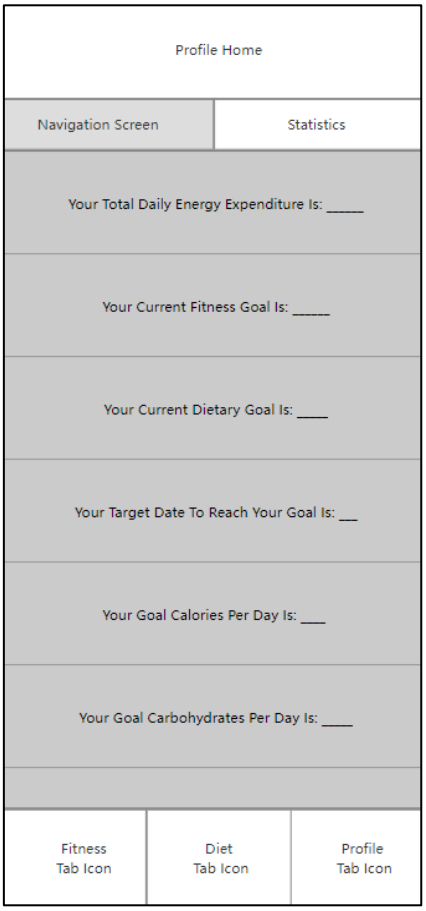


Figure 6.11: Profile Home: Statistics

Profile Home: Navigation (Figure 6.10):

Components:

- Navigation/Statistics Tabs: on Press Statistics → Navigate to the statistics screen
- Profile Settings button: on press → navigate to the profile settings screen
- TDEE form button: on press → navigate to the TDEE form
- Dietary Goal button: on press → navigate to the dietary goal form
- Fitness Goal button → on press → navigate to the fitness form

Profile Home: Statistics (Figure 6.11):

- Navigation/Statistics Tabs: On Press Navigation → Navigate to the navigation screen
- Scrollable list displaying all the user's calculations and goals such as their TDEE and fitness/diet goals.

Animations:

- Scroll Right → Navigate to the Sign-Up form
- Scroll Left → Navigate to the Login Form

Level 1 Screens:

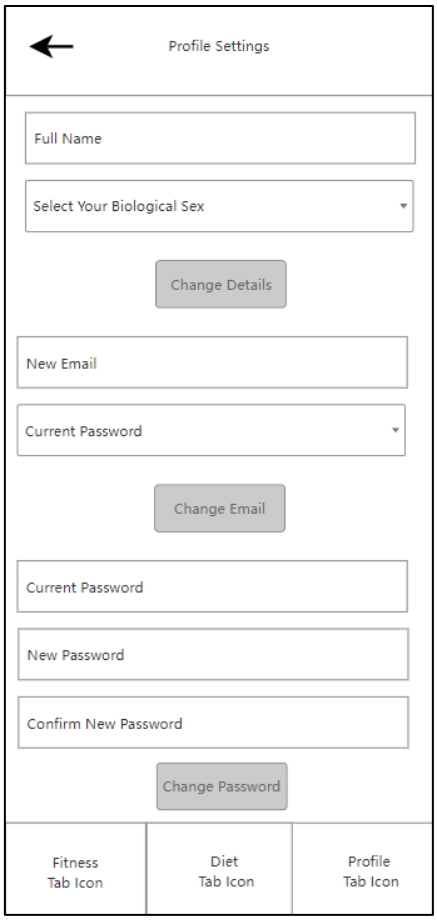


Figure 6.12: Profile Settings

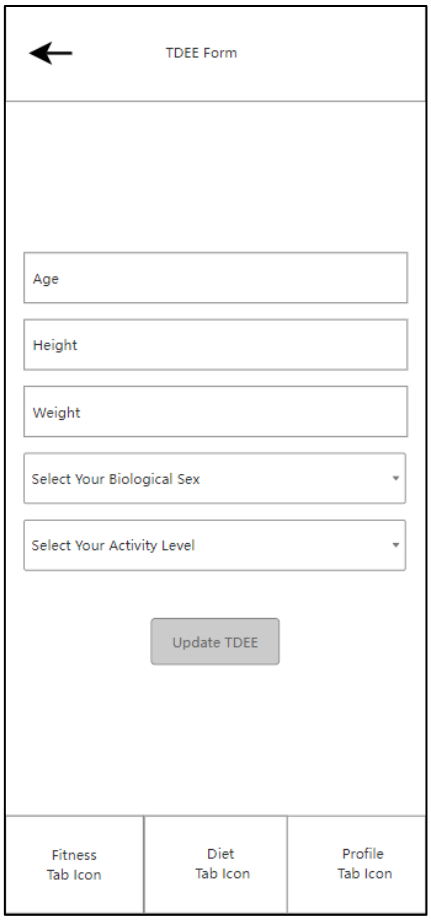


Figure 6.13: TDEE Form

Profile Settings (Figure 6.12):

Previous Screen = Profile Home: Navigation.

Components:

- Full Name input and biological sex dropdown input fields for user update profile
- Change Details button: on press → validate input fields; update user profile details
- New Email; Current Password input fields for user update email
- Change Email button: on press → validate input fields; update user email
- Current Password; New Password; Confirm New Password input fields for user update password
- Change Details button: on press → validate input fields; update user password

TDEE Form (Figure 6.13):

Previous Screen = Profile Home: Navigation.

Components:

- Age; Height; Weight; Biological Sex; Activity Level input fields for calculating/updating the users TDEE
- Update TDEE button: on press → validate input fields; calculate + update user TDEE.

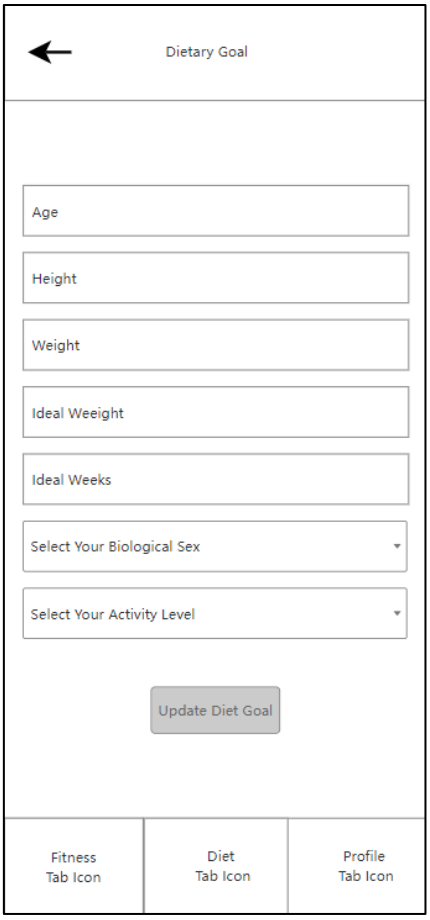


Figure 6.14: Dietary Goal Form

Figure 6.15: Fitness Goal Form

<u>Dietary Goal Form (Figure 6.14):</u>

Previous Screen = Profile Home: Navigation.

Components:

- Age; Height; Weight; Ideal Weight, Ideal Weeks; Biological Sex; Activity Level input fields for updating the users dietary goal and goal calories/macronutrients
- Update Diet Goal button: on press → validate input fields; update user diet goal; calculate goal calories, protein, fats, and carbs.

<u>Fitness Goal Form (Figure 6.15):</u>

Previous Screen = Profile Home: Navigation.

Components:

- Fitness icon image
- Fitness Goal; Training Frequency dropdown input fields for updating the users fitness goal
- **Update Fitness Goal button: on press** → validate dropdown input fields; update user fitness goal.

6.2.4 Diet Side

Level 0 Screens:

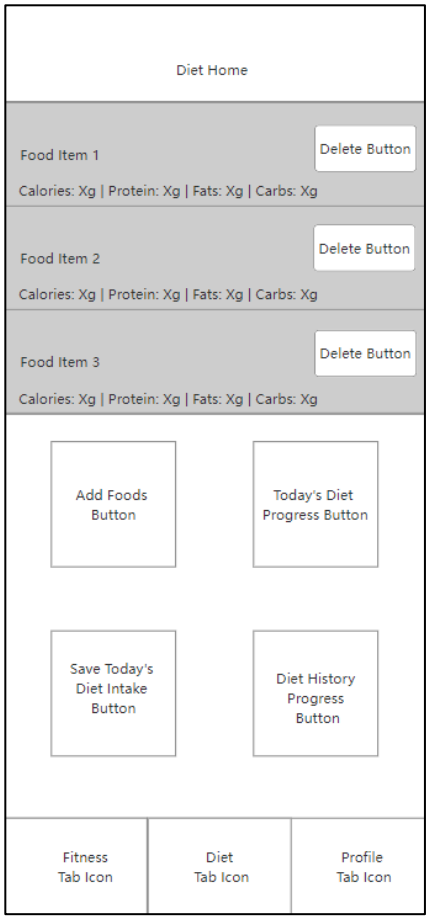


Figure 6.16: Diet Home

<u>Diet Home (Figure 6.16):</u>

Components:

- Scrollable list of food items consumed today, along with their calories and macronutrients
- Delete button next to each item in list: on press → delete food item from foods consumed today
- Add Foods button: on press → navigate to the Add Foods screen
- Today's Diet Progress button: on press → navigate to the Today's Intake Totals screen
- Save Today's Intake button: on press → save today's food intake totals to the back-end local SQLite database
- Diet History Progress button: on press → navigate to the Diet History Progress Screen

Level 1 Screens:

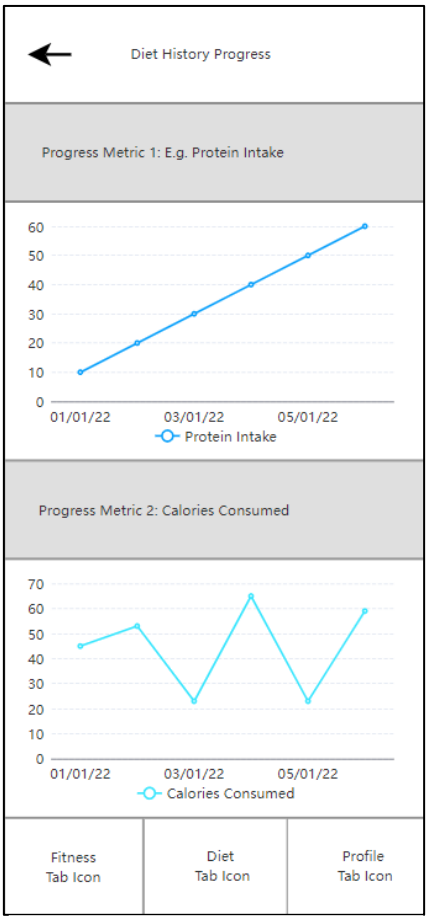


Figure 6.17: Diet History Progress

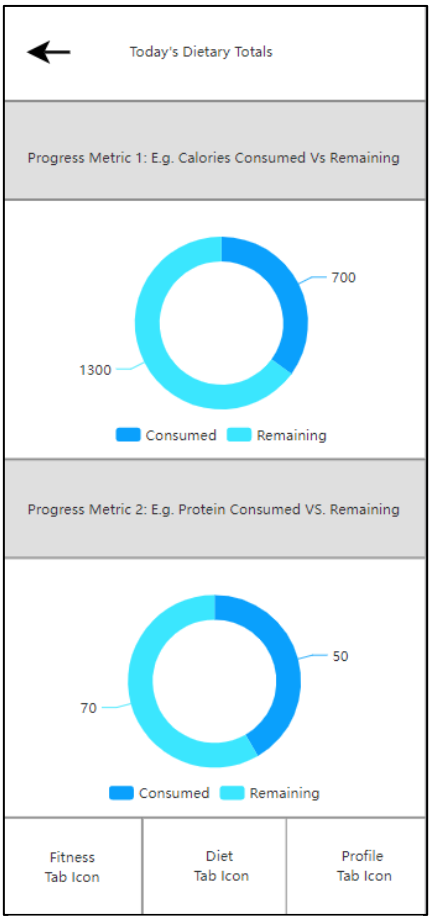


Figure 6.18: Today's Intake Totals

<u>Diet History Progress (Figure 6.17):</u>

Previous Screen = Diet Home

Components:

- All screen components in a scrollable list
- Progress Metric 1; Progress Metric 2 etc. subheadings
- Visual graphs depicting user dietary progress over time along different metrics

<u>Today's Intake Totals (Figure 6.18):</u>

Previous Screen = Diet Home

Components:

- All screen components in a scrollable list
- Calories Consumes VS Remaining; Protein Consumed VS. Remaining; Carbohydrates Consumed VS. Remaining; Fats Consumed VS. Remaining subheadings followed by corresponding visual pie charts depicting user depicting daily food intake analysis.

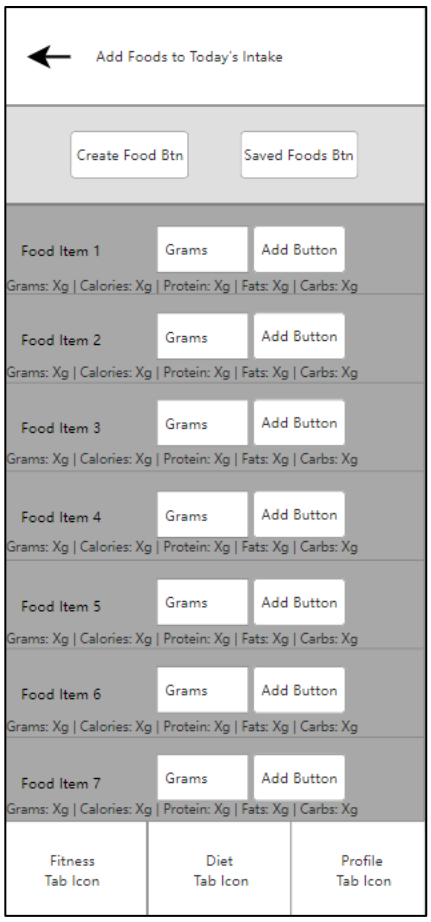


Figure 6.19: Add Foods

<u>Add Foods (Figure 6.19):</u>

Previous Screen = Diet Home

Components:

- Scrollable list of default food items, along with their calories and macronutrients
- Grams input field for user to add a food to today's intake
- Add button: on press → validate grams input field; add food item to user's today's intake.
- Create Food button: on press → navigate to Create Foods Form.
- Saved Foods button: on press → navigate to Saved Foods List

Level 2 Screens:

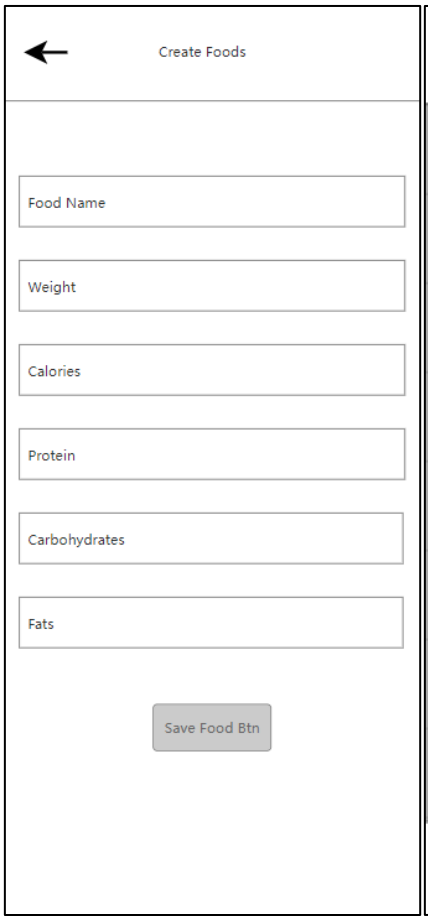


Figure 6.20: Create Food Form

Figure 6.21: Saved Foods List

<u>Create Food Form (Figure 6.20):</u>

Previous Screen = Add Foods

Components:

- Food Name; Weight; Calories, Protein; Carbohydrates; Fats input fields for creating a food item
- Save Food button: on press → validate input fields; create and save food item.

<u>Saved Foods (Figure 6.21):</u>

Previous Screen = Add Foods

Components:

- Scrollable list of user-created food items, along with their calories and macronutrients
- Grams input field for user to add a food to today's intake
- Add button: on press → validate grams input field; add food item to user's today's intake.

6.2.5 Fitness Side

Level 0 Screens:

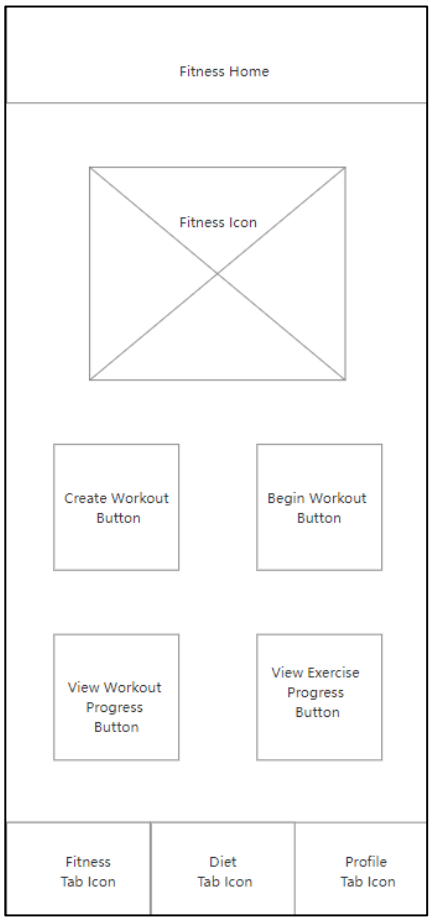


Figure 6.22: Fitness Home

<u>Fitness Home (Figure 6.22):</u>

Components:

- Fitness icon image
- Create Workout button: on press → navigate to the Workout Progress screen
- Begin Workout button: on press → navigate to the Begin Workout screen
- View Workout Progress button: on press → navigate to the Workout Progress screen
- View Exercise Progress button: on press → navigate to the Search Exercises screen

Level 1 Screens:

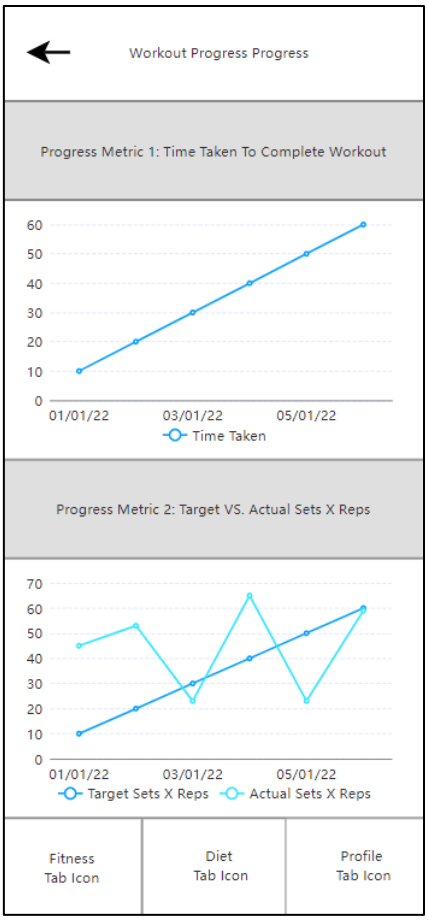


Figure 6.23: Workout Progress

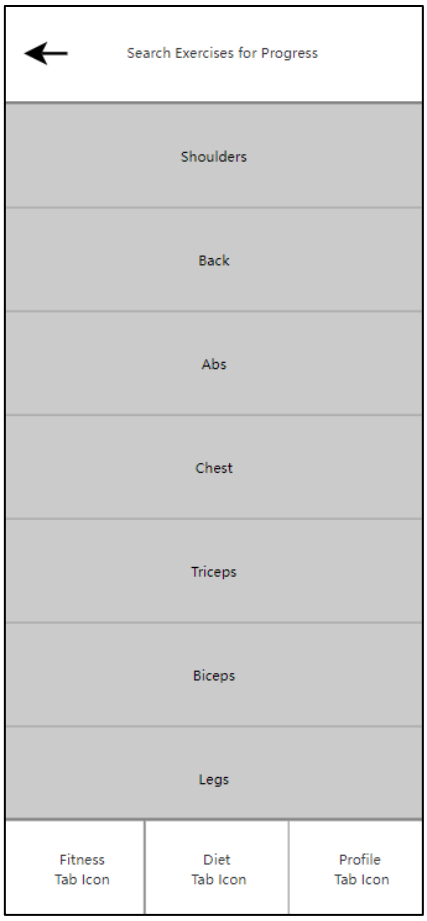


Figure 6.24: Search Exercises (Progress Route)

<u>Workout Progress (Figure 6.23):</u>

Previous Screen = Fitness Home

Components:

- All screen components in a scrollable list
- Progress Metric 1; Progress Metric 2 etc. subheadings
- Visual graphs depicting user workout progress over time along different metrics (for example, target vs. actual sets x reps).

<u>Search Exercises (Figure 6.24):</u>

Previous Screen = Fitness Home

Components:

- Scrollable list of muscle group names
- All muscle group names a pressable. On press → navigate to the Muscle Group Exercises screen (corresponding to the muscle group pressed)

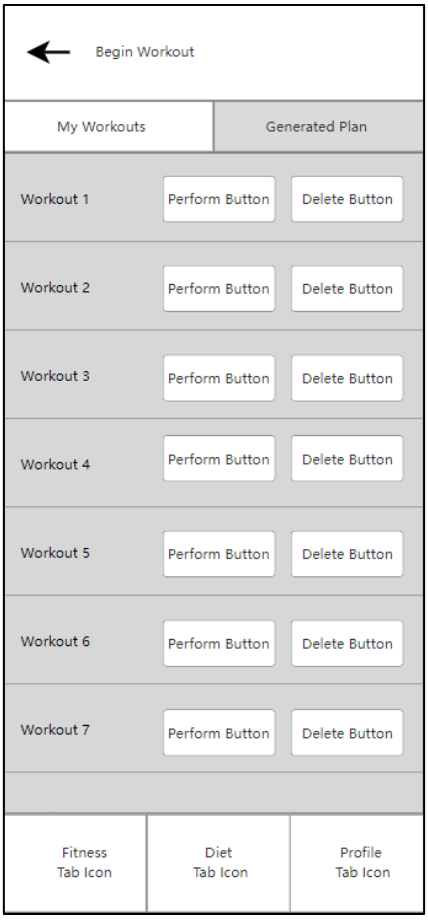


Figure 6.25: Begin Workout

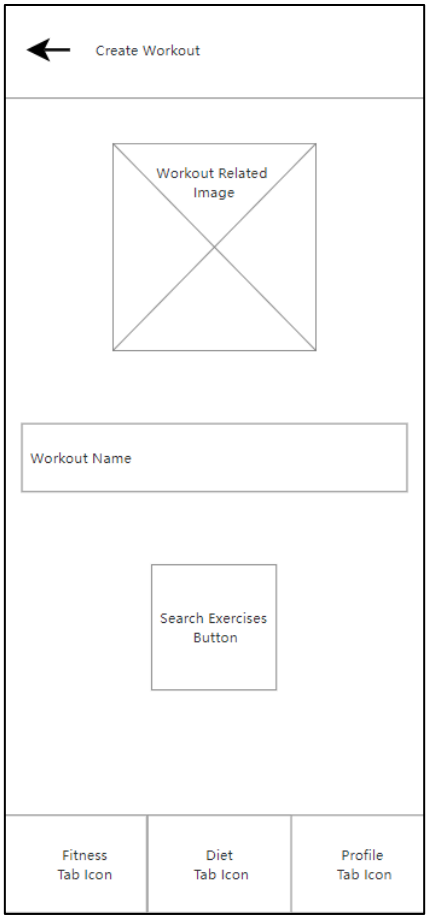


Figure 6.26: Create Workout

<u>Begin Workout (Figure 6.25):</u>

Previous Screen = Fitness Home

Components:

- Scrollable list of user-created workouts in My Workouts tab, and scrollable list of generated workout plan in Generated Plan tab
- Perform button: on press → navigate to the Perform Workout screen
- Delete button: on press → delete created workout from backend database

<u>Create Workout (Figure 6.26):</u>

Previous Screen = Fitness Home

Components:

- Workout icon image
- Workout Name input field for creating a workout
- Search Exercises button: on press → navigate to the Search Exercises screen

Level 2 Screens:

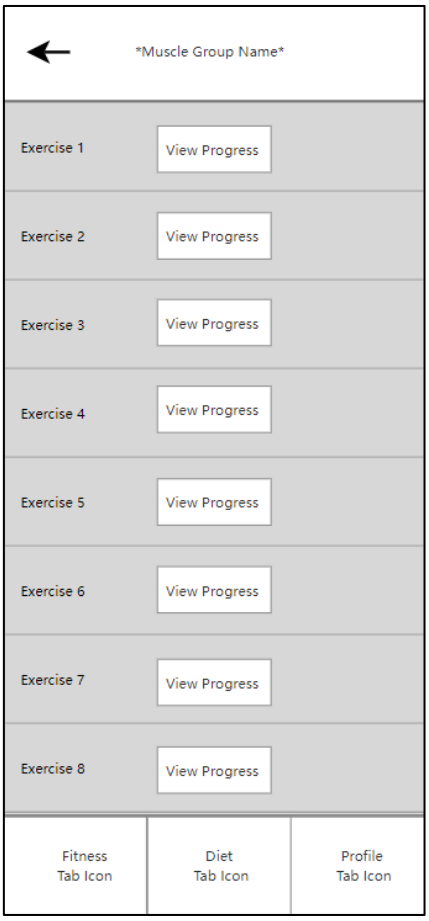


Figure 6.27: Muscle Group Exercises (Progress Route)

Figure 6.28: Perform Workout

<u>Muscle Group Exercises (Figure 6.27):</u>

Previous Screen = Search Exercises (Progress Route)

Components:

- Scrollable list of muscle group exercises names corresponding to the user's selected muscle group from previous screen
- View Progress buttons: on press → navigate to the View Selected Exercise's Progress screen

<u>Perform Workout (Figure 6.28):</u>

Previous Screen = Begin Workout

Components:

- Timer displaying the amount of time the user is spending on performing the workout
- Start button: on press → start the timer
- Stop button: on press → stop the timer
- Scrollable list of exercises corresponding to selected workout
- Reps and Weight input fields to input the exercise, reps, and weight the user has performed in the workout
- Add button: on press → validate the input fields; add the exercise, reps and weight to the performed workout.

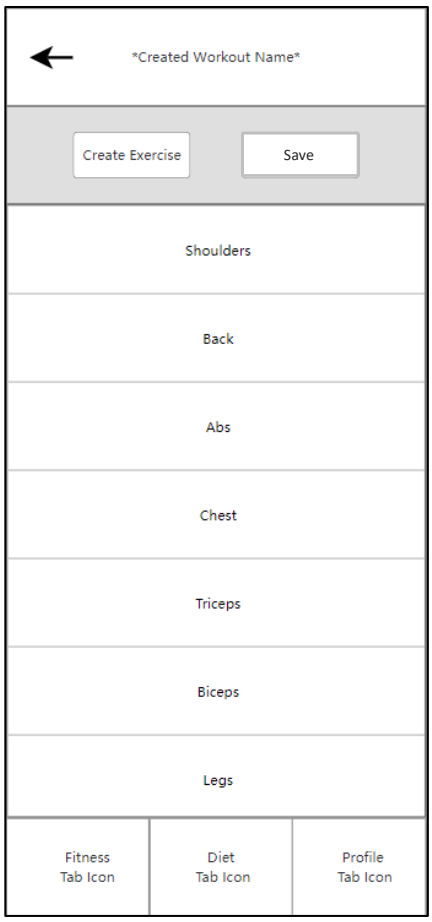


Figure 6.29: Search Exercises (Create Workout Route)

<u>Search Exercises (Figure 6.29):</u>

Previous Screen = Create Workout

Components:

- Create Exercise button: on press → navigate to the Create Exercise form.
- Save button → on press → navigate to Performed Workout Overview
- Scrollable list of muscle group names
- All muscle group names a pressable. On press → navigate to the Muscle Group Exercises screen (corresponding to the muscle group pressed)

Level 3 Screens:

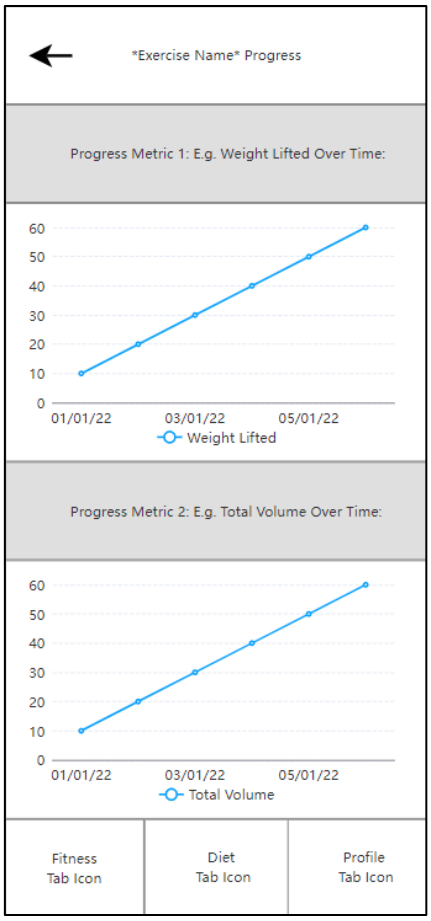


Figure 6.30: View Selected Exercise's Progress

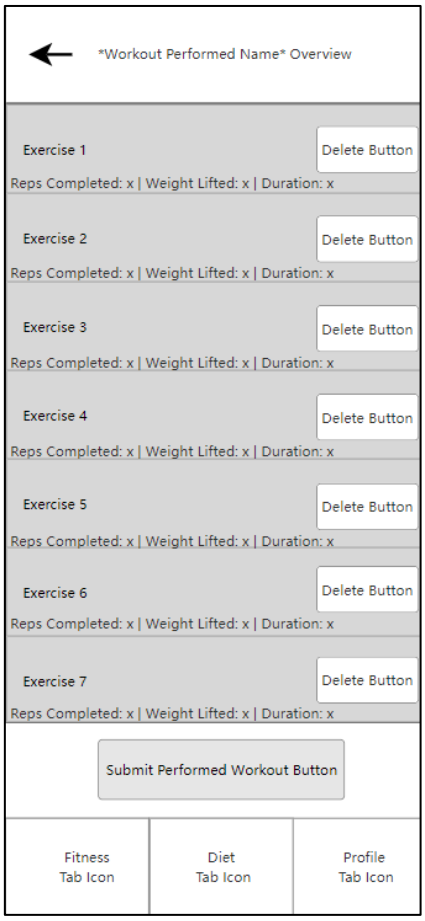


Figure 6.31: Performed Workout Overview

<u>View Selected Exercise's Progress (Figure 6.30):</u>

Previous Screen = Muscle Group Exercises

Components:

- All screen components in a scrollable list
- Progress Metric 1; Progress Metric 2 etc. subheadings
- Visual graphs depicting user exercise progress over time along different metrics (exercise volume completed over time, weight lifted over time).

<u>Performed Workout Overview (Figure 6.31):</u>

Previous Screen = Perform Workout

Components:

- Scrollable list of exercises, reps completed, weight lifted, and duration from the workout the user performed on the previous screen.
- Delete buttons: on press → delete exercise from list of exercises user performed in workout
- Submit button: on press → save the performed workout to database; navigate to the fitness home screen

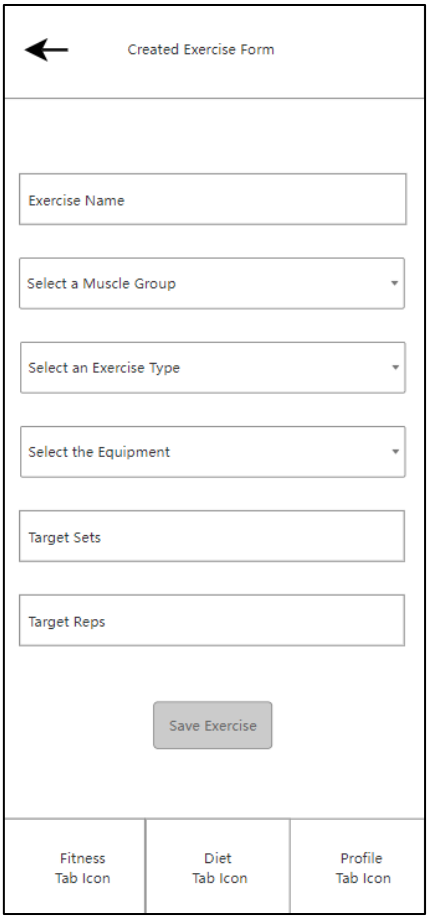


Figure 6.32: Create Exercise Form

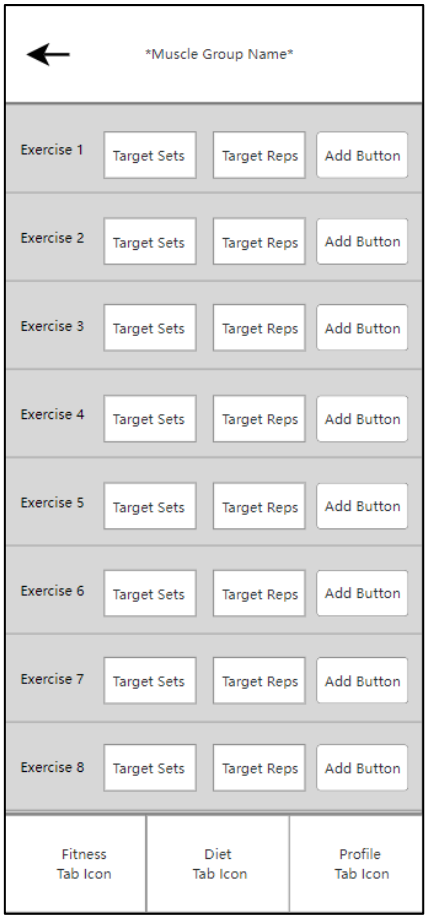


Figure 6.33: Muscle Group Exercises (Create Workout Route)

Create Exercise Form (Figure 6.32):

Previous Screen = Search Exercises

Components:

- Exercise Name; Muscle Group dropdown; Exercise Type dropdown; Equipment dropdown; Target Sets, Target Reps input fields for creating a exercise
- Save Exercise button: on press → validate input fields; create exercise and save; add exercise to workout user is creating

Muscle Group Exercises (Figure 6.33):

Previous Screen = Search Exercises

Components:

- Scrollable list of exercises corresponding to user-selected muscle group from previous screen
- Target Sets, Target Reps input fields to input the exercise, target reps, and target sets the user wants to add to the workout being created
- Add button: on press → validate the input fields; add the exercise, target reps and target sets to the workout being created

Figure 6.34: Created Workout Overview

<u>Created Workout Overview (Figure 6.34):</u>

Previous Screen = Search Exercises

Components:

- Scrollable list of exercises, muscle groups, equipment, target sets and target reps the user has added to the workout they are creating
- Delete buttons: on press → delete exercise from list of exercises in the workout the user is creating
- Save Workout button: on press → save the workout the user has created to database; navigate to the fitness home screen
- Delete Created Workout button: on press → delete the workout the user has created; navigate to the fitness home screen

6.2.6 Global Components:

Throughout the fitness application, there is a need for global components to be utilized in order to handle use-cases that are common throughout the fitness application. 3 potential use cases have been defined:

1. Validation: When the user inputs invalid entries in forms and attempts to submit the form.
2. Confirmation: When the user makes changes to their data through submitting a form
3. Loading: When the user accesses a screen that requires initial loading time to gather and manipulate data from a backend before displaying its contents.

In order to handle the validation use-case, an error message is displayed on-screen informing the user of the invalid form input. The design of this error message is shown in Figure 6.35, along with an example of what the error message tells the user.

In order to handle the confirmation use-case, an alert box is displayed on the screen confirming the data entry the user is requesting to make. This alert box provides the user with the option to cancel the request, of confirm the request. An example of this alert box is shown if Figure 6.36.

In order to handle loading use-cases, a loading animation is displayed to the user when navigating to the screen that requires it. It is displayed for one or less than one second before it is hidden and the contents of the screen are displayed. The planned design for this loading animation is shown in Figure 6.37.

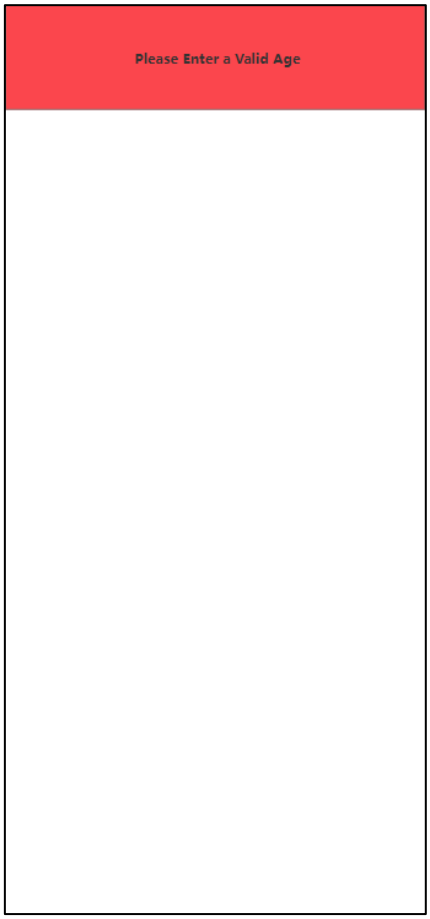


Figure 6.35: Invalid Form Message Example

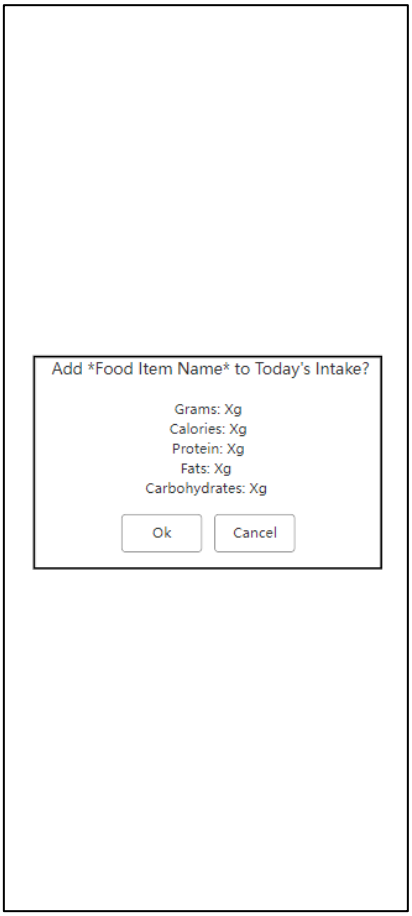


Figure 6.36: Confirmation Alert Box Example

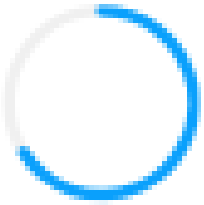


Figure 6.37: Loading Animation Design

Invalid Form Message (Figure 6.35 for example):

For each form, when submit is pressed:

1. Execute function to check valid input entry
2. If function returns invalid form, show the red invalid form error message at the top of the screen for 3 seconds
3. Else: execute the function that the form is intended for

Confirmation Alert Box (Figure 6.36 for example):

For each button that requires confirmation from the user, on press:

1. Display an alert box informing the user of what they are about to do
2. On press Ok → execute the function that the initial button was intended for
3. On press Cancel → break the function; close the alert box

Loading Animation Design (Figure 6.37 for example):

For screens that are reading and manipulating data from a backend database (for example view workout progress graphs over time):

1. Hide the components of the screen and show the loading animation for <= one second
2. Hide the animation and show the screen components after <= one second

6.3 Database Design

Strategym is planned to utilize a cloud database for the user's profile data and a local database for all other instances of data storage. As mentioned, the reason for this is the secureness of Google's Firebase cloud database server, as well as enabling the user the ability to switch devices and login to their account on the other device with the ability to access the core profile data of their account. The reason for the inclusion of a local database is to provide reliable offline functionality within the application without the need of a network connection or the need to cache the data first from a cloud database.

6.3.1 Firebase

The client-server architecture for Firebase's Firestore has been constructed in Figure 6.7: Firebase Client/Server Architecture. When the client executes a read/write request function to the cloud database, they first pass an authentication API which validates that the user is authenticated before granting any permissions to Firestore. If the validation is true, then the user is granted access to Firestore and is sent an instance of the document they requested to read/write. If validation is false, then the user is denied access to the Firestore database and the Firebase API returns an error message which notifies the user that they do not have access to the database.

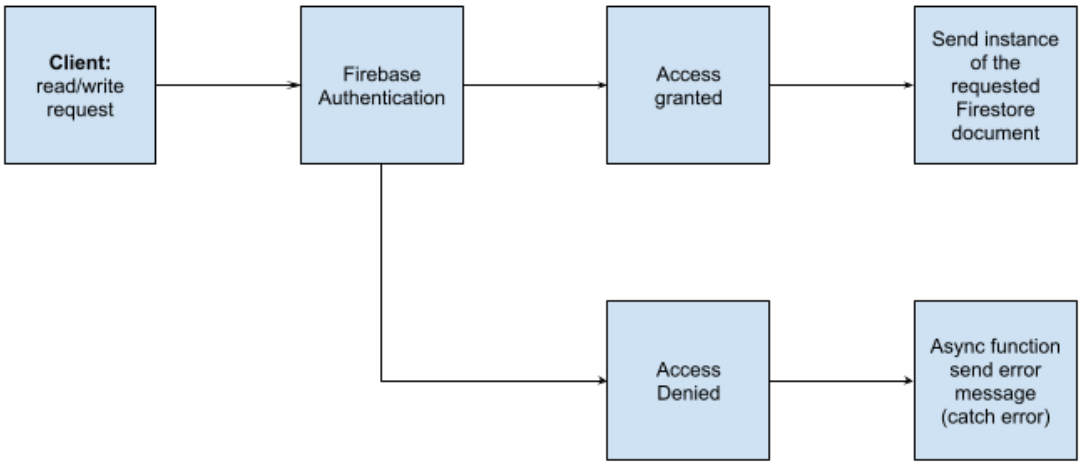


Figure 6.7: Firebase Client/Server Architecture

6.3.2 SQLite

The client/server architecture for SQLite has been constructed in Figure 6.8: SQLite Client/Server Architecture When the client executes a read/write request function to a local database, the Android operating system first examines whether the permissions of the application have been set to allow reading/writing local storage. If the application has been granted permission, then the SQL query defined in the function is executed, and the user has access to the database and tables stored in the device's local storage. If the application is not granted permission, then the SQL query in the function is not executed and the user has no access to the local databases stored on the device.

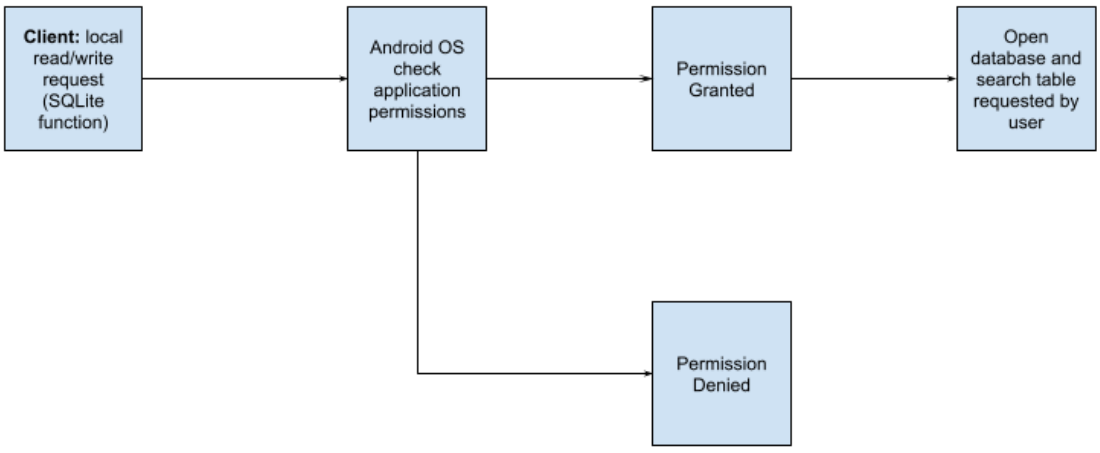


Figure 6.8: SQLite Client/Server Architecture

7 Implementation & Testing

This section covers the implementation of Strategym. It provides a breakdown of the procedures that have taken place when developing the fitness application. There are two main stages of the implementation of Strategym:

1. Configuration: this is the initial setup of the development environment and the configuration of the technological tools required for the application's development
2. Execution and Testing: this stage walks through the actual implementation of the components of the application, and the testing phases after each increment.

This section also covers the testing of Strategym. This is because, in the incremental software development approach chosen for this project, testing occurs after each increment as opposed to after the entire implementation stage. Therefore, it makes more sense to cover the testing in concurrence with implementation. See Appendix 9.1 for the full testing tables of each increment.

The entirety of the implementation and testing that is covered in this section is in chronological order, including the subsections within the two main stages mentioned above.

7.1 Testing Strategy:

There are four levels of software testing: (Albarka Umar, 2020)

1. Unit testing: the testing of individual 'unit' in isolation. A unit can be defined as a single element in the code such as a function or UI component.
2. Integration testing: the testing of two or more combined units that work together, to ensure that the flow of functions, components or data within the software is bug-free.
3. System testing: the testing of the entirety of a completed software to measure its outcome against the functional and non-functional requirements.
4. Acceptance testing: the testing of the entirety of a completed software to measure its outcome against the user's/clients' requirements.

Unit testing is a time-consuming approach to testing since it involves writing test cases for every single individual component of the software. Given the time frame and scope of this project, unit testing would be an impractical testing strategy to utilize. Furthermore, unit testing would have to be strategized with another level of testing since it does not ensure that individual units working together are working correctly, only units in isolation. Much of the functionality of Strategym involves the integration of individual units. For example, the user inserting their dietary goal into a form integrates two components: the first being calculating the goal calories and macronutrients and inserting this into the database, and the second being displaying this data on the user statistics tab of the profile home screen. Therefore, integration testing to handle such instances is the optimal testing strategy to employ after each increment is implemented.

7.2 Configuration & Setup

7.2.1 Development Environment

The development environment chosen for the implementation of the application is Visual Studio Code, since it includes extensions tailored towards the react-native framework and the technological stack planned to be used throughout development such as Node.js and SQLite. A list of the extensions and their purpose are given in Table 7.1: Visual Studio Code Extensions.

Extension	Description
React Native Tools (Microsoft, 2022)	Provides command-line operations and debugging for React Native projects
Node.js Extension Pack (Anderson, 2022)	Includes support for Node.js terminal commands and supports the latest JavaScript version (ECMAScript 2018).
SQLite (Alexcvzz, 2022)	Enables database visualization: opening and querying SQLite files with the .db extension
Prettier – Code formatter (Prettier, 2022)	Formats JavaScript code in order to keep the whitespace and indentations well structured.

Table 7.1: Visual Studio Code Extensions

7.2.2 Initializing the Application:

React-Native has extensive documentation regarding initializing the application in the development environment, which, once initialized, produces an initial project folder with the boilerplate required to run

the application (Meta Inc, 2022). The project generated is essentially a blank canvas on which developers can build. For the initialization of Strategym, React Native CLI is the command line application that has been used since there are no limitations on the API's and libraries that can be installed to enhance in-app features, unlike the alternative Expo CLI (Expo, 2022). This in turn prevents any knock-on effects that could limit the functionality of the application.

7.2.3 Initializing Firebase:

In order to initialize Firebase, the Android package name of the application is required. This package name is set in the AndroidManifest.xml file generated by the initialization of the application. This is why the initialization of the application has occurred first. In this case, the package has been named "com.strategym" (Figure 7.1: Firebase Initialization).

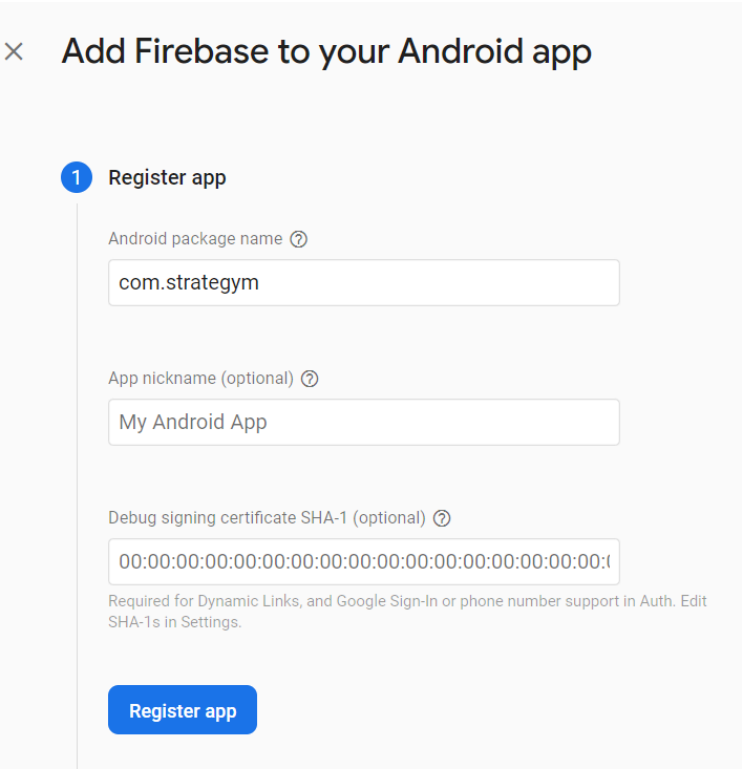


Figure 7.1: Firebase Initialization

Google then provide the next steps for initialising Firebase with the application, which includes copying the generated configuration file "google-services.json" into the project directory which is used to connect the project to the Firebase API client, and then adding the necessary dependencies to the application's build.

A display of the resulting Authentication and Firestore GUI that is used in this project is given in Figures Figure 7.2: Firebase Authentication GUI and Figure 7.3: Firebase Cloud Firestore GUI respectively.

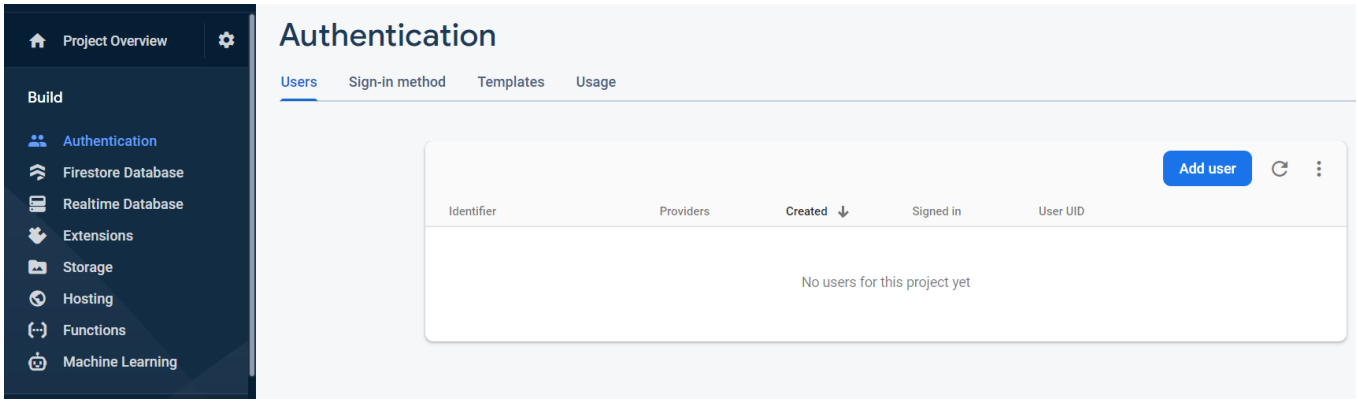


Figure 7.2: Firebase Authentication GUI

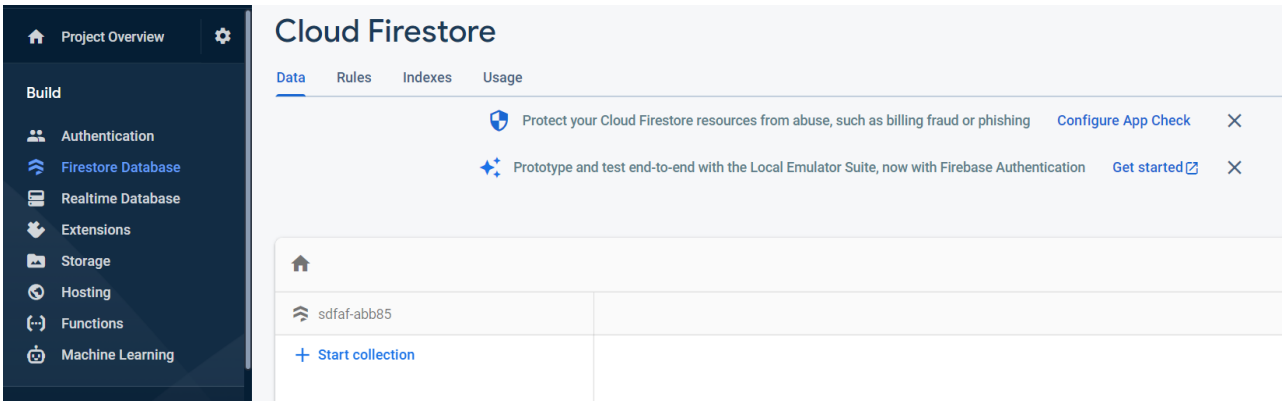


Figure 7.3: Firebase Cloud Firestore GUI

7.2.4 NPM Libraries

React-Native CLI comes with the “npm” terminal command which enables packages from the node package manager to be installed and saved into the project directory for use. The syntax to install packages using this command line is straightforward and an example is given below (Figure 7.4: NPM Install):

```
npm install package-name --save
```

Figure 7.4: NPM Install

A descriptive list of the main node packages installed is given below (Table 7.2: Descriptive List of Main Packages Used In-App). Appendix 9.3 contains a full list of all core and third-party packages, dependency packages and any specific modules within packages that have been installed.

Package Name	Description
react-native-firebase (Invertase Limited, 2022)	The official recommended collection of supporting modules for connecting react-native mobile applications with Firebase services.
react-native-sqlite-storage (Andpor, 2022)	A library of modules and functions that provide support for using SQLite3 in react-native mobile applications. All SQL-based querying syntax is eligible with this library.
react-navigation (Meta Inc, 2022)	A library that is made up of core utilities to create and handle the navigational structure of a react-native application. Also contains UI components to complement the navigation stack.
react-native-paper (Callstack, 2022)	A high-quality UI/UX library that includes design components such as buttons, text inputs, themes and icons that can be utilized in a react-native application.
react-native-vector-icons (Arvidsson, 2022)	Contains an extensive library of material icons optimized for mobile applications with the

	additional ability to change the size and colour of icons.
react-native-picker-select (LawnStarter Organisation, 2022)	A dropdown input picker UI package which emulates the native picker interfaces for Android.
react-native-flash-message (Ferreira, 2022)	A UI package which displays a message on the screen for a set amount of time, with the additional ability to format the design and placement of the message.
react-native-chart-kit (Indiespirit, 2021)	A library that includes utilities for displaying graphs and charts on screen given arrays of data.
react-native-pie-chart (Xu, 2022)	A library that includes utilities for displaying pie charts and donut charts on screen given arrays of data.
react-native-onboarding-swiper (Johannes Filter, 2022).	A library that includes utilities for displaying onboarding screens for the initial application launch
brain.js (Open-Source Collective, 2022)	An artificial intelligence library used to configure, train, and use neural networks on given data inputs. Enables the configuration of the neural network whilst hiding the mathematical complexity of matrix manipulation. This package has been installed for the use of implementing a bodyfat percentage predictor in-app as an additional feature if there is enough time to do so.
fs (Dahl, 2022)	Node.js filesystem package which enables reading and writing files in the project directory. This

	package will be used in conjunction with brain.js to save a trained neural network (if implemented) to the local project directory.

Table 7.2: Descriptive List of Main Packages Used In-App

7.2.5 Initializing Local Read/Write Permissions for SQLite

In order for SQLite to read and write to an Android device's local storage, these permissions must be added to the AndroidManifest.xml file located at root → android → app → src → main → AndroidManifest.xml. This storage is considered internal to the device, but external to the application. Thus, the following 2 lines have been added to this file (Figure 7.5: Android Read/Write Permissions):

```
2
3     <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
4     <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
5
```

Figure 7.5: Android Read/Write Permissions

7.2.6 Initializing the Default Food Database

Since JSON file storage follows the same structure as JavaScript's array of objects, and since JavaScript ES6 includes the function JSON.stringify to exchange and manipulate data within a JavaScript program, all internal databases within this application are stored in JSON files structured as arrays of objects with properties.

With respect to the internal food database that comes with the application, the USDA food nutrition database contains over 2000 food items and includes the micronutrients of each food item too. The inclusion of the entire database would be impractical since it would drastically increase the application size, and micronutrients are outside of the requirements of Strategym. Therefore, this database has been pruned to include 28 popular fruits, vegetables, grains, and meats, with the properties: name, calories, protein, fats, carbohydrates, and grams. This file is called "nutritionalDatabase.json" and is located in the dietScreens folder of the project directory (see Appendix 9.5 for program listing).

7.2.7 Initializing the Default Exercise Database

With respect to the internal exercise database that comes with the application, 3 exercises per muscle group that cause the highest muscle activation were identified in problem solution section 3. The database has been stored as a JSON file "exerciseDatabase.json" as an array of objects with the properties: exercise name, muscle group, type, equipment, and is located in the fitnessScreens folder of the project directory (see Appendix 9.5 for program listing).

7.2.8 Initializing the Generated Workouts Database

With respect to the internal workouts database that is planned to be utilized for generating a workout plan for the user based on their set fitness goal, there are 3 main fitness goals to generate workouts for: endurance, hypertrophy, and strength. As well as this, the user must set a training frequency ranging from 1 time per week to 7 times per week. Since the research conducted had measured the optimal number of sets per muscle group as weekly totals, the target sets and reps property of each exercise varies depending on the training frequency set by the user as well as by the fitness goal set by the user. The following sigma function determines how many workouts to generate in total per fitness goal:

$$Number\ of\ Workout\ Plans\ Per\ Fitness\ Goal: \sum_{n=1}^{7} n$$

Equation 7.1: Sigma Function for No. of Workouts Per Fitness Goal

This output of this function is 26, which means that per fitness goal, 26 different workouts must be generated per fitness goal to generate a workout plan that provides the user with the optimal number of sets and reps per workout for them to achieve their fitness goal. This totals to 26 x 3 = 78 workouts in total to be included internally in the application.

These workouts have been stored in a .js file named after the fitness goal they correspond to (for example hypertrophy.js), as an array of objects labelled the training frequency they correspond to (for example hypertrophyx1 where x1 means a training frequency of once a week) with the properties: exercise, muscle group, type, equipment, target sets and target reps. The 3 files for the workout plans of the different fitness

goals are called endurancePlans.js, hypertrophyPlans.js, and strengthPlans.js, and are located in the workoutPlans folder of the project directory (see Appendix 9.5 for program listing).

7.3 Execution

This section outlines the incremental implementation of the features of Strategym defined in the requirements. At the end of each increment screenshots have been included that display the resulting components, followed by a description of the integration testing phase along with any resolutions that had been made. For the full testing tables of each increment, see Appendix 9.1.

7.3.1 Increment 1: User Interface Design Implementation

The first increment regards producing the user interface for the screens designed in the system design section. This would ease the navigational setup of the application in the next increment since most of the screens to include in the navigational stack would already be defined and able to import.

7.3.1.1 Static UI:

All screens of this application except from the login/registration screen and profile home screen encompass a static user interface design, where no animation-based effects are used. These screens have been produced first, using the in-built react-native UI components. These in-built components have equivalence to HTML tags. For example, the <View> component in react-native is equivalent to the tag in HTML. Furthermore, the structure of these components are identical to HTML tags too. For example, <View></View> is wrapped around all the content within a section of the screen the same way is wrapped around all the content within a section of a webpage. An essential react-native component utilized for the UI of each screen is the <TouchableOpacity> component, which is the equivalent of HTML's <a> link tag. The TouchableOpacity component takes an OnPress method, which will be configured in the next increment. It is mainly wrapped around View components styled as tiles that would navigate the user to another screen in the application.

All built-ins react-native components include the prop "styles" which is the equivalent of CSS styles sheets on the web, enabling the developer to set the design of a component. This prop has been utilized for most components to set the structure, layout and colour of each screen and its constituents.

In order to have fine control over the display and functionality of the headers on each screen, custom header components have been produced which are imported into each screen. One is a standard custom header component which simply takes the screen name as a prop and places it in the centre of the header. The second is a custom header component designed the same way, but also takes the previous screen name as a prop and includes a back icon at the top left, which, during the navigation increment will be configured to navigate the user to the previous screen in the stack.

```
const CustomHeaderComponent = props => {
  return (
    <View
      style={{
        flexDirection: 'row',
        justifyContent: 'center',
        padding: 10,
        borderBottomWidth: 1,
        borderBottomColor: 'black',
        backgroundColor: '#CBC3E3',
      }}>
      <View>
        <Text style={{fontSize: 20, color: 'black'}}>{props.pageName}</Text>
      </View>
    </View>
  );
};

export default CustomHeaderComponent;
```

Figure 7.6: Sample Custom Header Component Code

```
const CustomHeaderWithBack = props => {
  return (
    <View
      style={{
        flexDirection: 'row',
        justifyContent: 'center',
        padding: 10,
        borderBottomWidth: 1,
        borderBottomColor: 'black',
        backgroundColor: '#CBC3E3',
      }}>
        <View
          style={{
          flexDirection: 'row',
          justifyContent: 'flex-start',
          padding: 5,
        }}>
          <TouchableOpacity
            onPress={() => }> // previous screen navigation in onPress method during next increment
          <MaterialCommunityIcons
            name="arrow-left"
            color="black"
            size={30}></MaterialCommunityIcons>
        </TouchableOpacity>
      </View>
      <View>
        <Text style={{fontSize: 20, color: 'black'}}>{props.pageName}</Text>
      </View>
    </View>
  );
};

export default CustomHeaderWithBack;
```

Figure 7.7: Sample Custom Header with Back Code

Components from 3rd party libraries have also been utilized where necessary to meet the design specifications of section 6. For example, the fitness home screen includes a fitness logo to be displayed on the screen, thus the component <MaterialCommunityIcons> has been utilized to acquire a fitness-related image from the library to display on screen. For forms and buttons, the 3rd party library react-native-paper has been utilized to implement the input fields and the buttons (the <TextInput> and <Button> components from this library). The library react-native-picker-select has been utilized to implement input fields with a dropdown selection (the <RNPickerSelect> component from this library). Sample code demonstrating the use of in-built react-native components, CSS styles, 3rd party components, and the custom header component is shown below (Figure 7.8: Sample UI Code With 3rd Party; Custom; In-Built Components and Styles.

```
const FitnessHomeScreen = () => {
  return (
    <View style={{backgroundColor: '#cbc3e3', height: '100%'}}>

      <CustomHeaderComponent pageName="Fitness Home" />

      <View style={{alignItems: 'center'}}>
        <MaterialCommunityIcons
          name="weight-lifter"
          color="#171717"
          size={250}
        />
      </View>

      <View
        style={{flexDirection: 'row', justifyContent: 'center', padding: 25}}>
        <TouchableOpacity
          onPress={() => {}}> // insert onPress navigation to create workout screen
          <View
            style={{
              backgroundColor: '#301934',
              height: 140,
              width: 140,
              elevation: 400,
              borderRadius: 20,
            }}>
            <Text>
              Create Workout
            </Text>
          </View>
        </TouchableOpacity>
        <TouchableOpacity
          onPress={() => {}}> // insert onPress navigation to begin workout screen
          <View
            style={{
              backgroundColor: '#301934',
              height: 140,
              width: 140,
              elevation: 400,
              borderRadius: 20,
            }}>
            <Text>
              Begin Workouts
            </Text>
          </View>
        </TouchableOpacity>
      </View>
    </View>
  );
};

export default FitnessHomeScreen;
```

Figure 7.8: Sample UI Code With 3rd Party; Custom; In-Built Components and Styles

7.3.1.2 Form UI

In order to improve the usability of forms throughout the application, functionality has been implemented that changes the focus to the next input field when the user presses enter on a previous input. In order to implement this, the useRef react hook has been utilized to reference each text input field. This reference can

then be used by the previous input component on its onSubmitEditing property to change the focus to the referenced (the next) input field. The onSubmitEditing property is an property that comes with the <TextInput> component of react-native-paper, that determines the function that is executed when the user presses on the enter button on the keyboard. The following snipped code is an example of how this has been implemented (Figure Figure 7.9: Code Snippet: Changing Input Field Focus):

```
const heightRef = useRef();
const weightRef = useRef();

 return (

     <TextInput
         label="Height (cm)"
         ref={heightRef}
         returnKeyType="next"
         onSubmitEditing={() => {
           weightRef.current.focus();
         }}
     />

     <TextInput
         label="Weight (kg)"
         ref={weightRef}
     />

)
```

Figure 7.9: Code Snippet: Changing Input Field Focus

Animated UI:

The two three screens in the wireframe design that utilize animation in the form of top tabs and a horizontal swipe are the login/registration screen; the begin workout screen and profile home screen. React-native comes with an in-built “Animated” component, which includes various functions to animate a screen, such as changing the colour of other components it’s wrapped around as well as interpolating a screen horizontally on the user swiping left or right.

To achieve the animation design, firstly a custom component for the top tab buttons has been made which defines the design, style, and placement of the top tab buttons. It takes 3 props: title, backgroundColor, and onPress. The title prop displays the text displayed in the component; backgroundColor defines the background colour of the tab before, during and after the horizontal swipe; and the onPress prop which

defines what happens to the style/shading of the top tabs when pressing on the top tab button. The component is wrapped around the in-built <TouchableWithoutFeedback> component, which is similar to the <TouchableOpacity> component however doesn't change colour shade of a component wrapped inside it when pressed on. This is used as opposed to <TouchableOpacity> since the shade of the top tab button will be dependent on the animation rather than the pressing of it. <TouchableWithoutFeedback> includes the onPress prop which is passed from the main screen. Wrapped in this component there exists an <Animated.View> component which is the <Animated> component applied to the <View> component in react-native, and has the additional feature of being able to animate the View given a reference for the animation. In this case, the animation is the change in the background colour of the tab, and the reference prop for the background colours to animate from and to is passed from the main screen. Wrapped within the <Animated.View> component is a <Text> component which takes the prop title (passed from the main screen component) and displays it on the tab. The use of a separate component for the top tab buttons that takes these props is useful since it is reusable amongst all screens that require top tab buttons with this animation effect. The code for the top tabs component is (Figure 7.10: Top Tab Buttons Code):

```
const TopTabBtns = ({title, backgroundColor, onPress}) => {
  return (
    <TouchableWithoutFeedback onPress={onPress}>
      <Animated.View style={[styles.container, {backgroundColor}]}>
        <Text style={styles.title}>{title}</Text>
      </Animated.View>
    </TouchableWithoutFeedback>
  );
};

const styles = StyleSheet.create({
  container: {
    backgroundColor: '#301934',
    height: 45,
    width: '50%',
    justifyContent: 'center',
    alignItems: 'center',
    borderTopWidth: 1,
    borderBottomWidth: 1,
    borderColor: 'black,',
  },
  title: {color: 'white', fontSize: 20},
});

export default TopTabBtns;
```

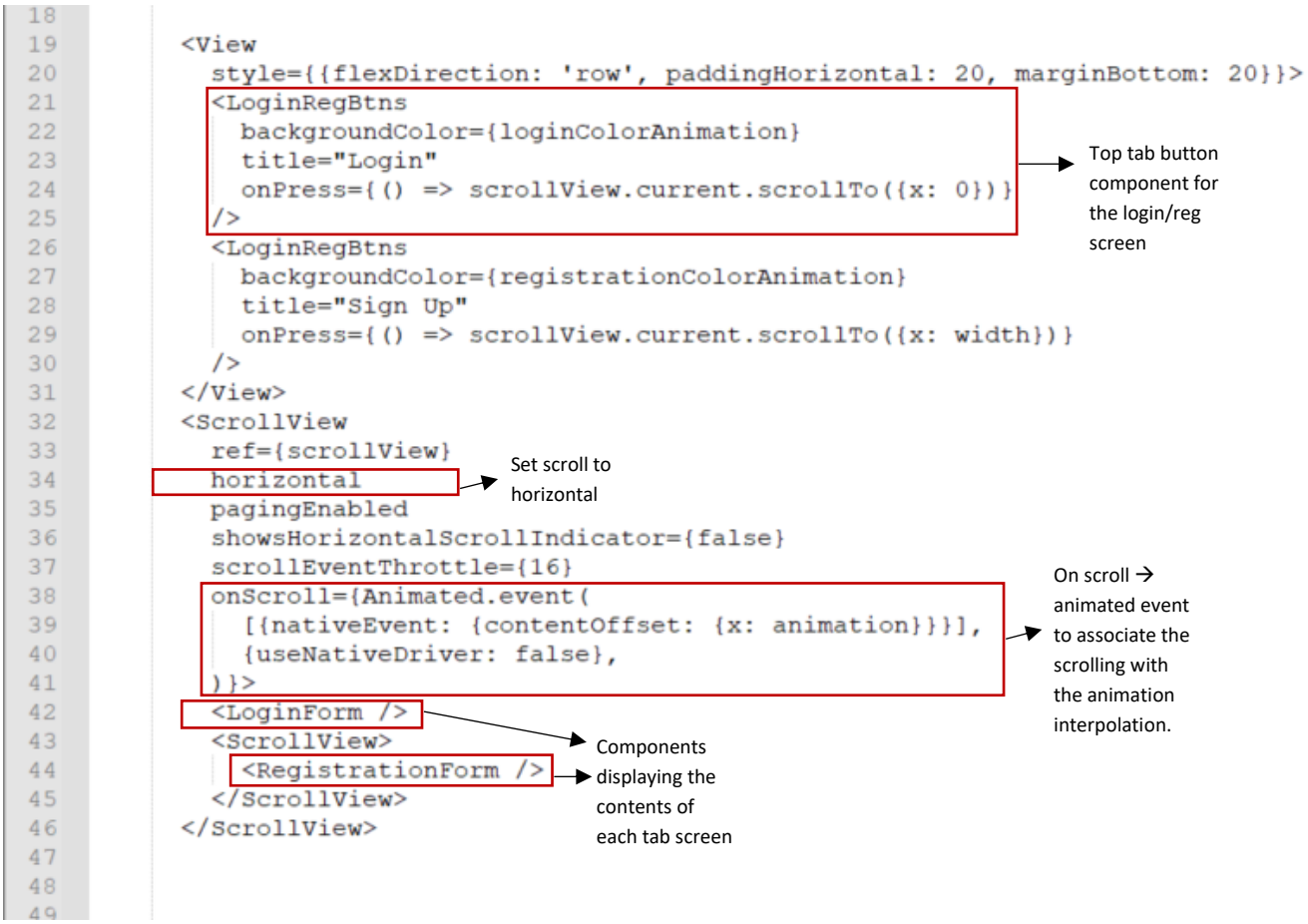
Figure 7.10: Top Tab Buttons Code

The second stage of achieving the animation effect is by configuring horizontal scrolls of the tab screens on the main component and the props to pass through to the top tabs' components. In order for the screen to

achieve a horizontal scroll, the in-built <ScrollView> component is utilized and configured to scroll to the end of the width of the screen, where the second tab's contents are placed. The useRef react hook is used on the <ScrollView> in order to get the current value of the scroll (i.e. the current position of where the user has scrolled horizontally to). This useRef is made associative with a defined 'animation' variable which holds the current value of the scroll as an animated value, meaning that the animation is correlated with the current state of the scroll performed by the user (i.e. as the user swipes left/right, the animation changes). In order to achieve this, the onScroll prop is defined in the <ScrollView> component which executes the animated event (Animated.Event) of interpolating between the set colours on scrolling horizontally between the screens. The colours that the animation interpolates between during the horizontal scroll are dark purple and chalk. The top tab button's background colour is dark purple when that tab's contents are displayed on screen and interpolate to chalk when that tab's contents are not displayed on screen. The following code snippets show how this animation effect has been implemented:

```
1   const animation = useRef(new Animated.Value(0)).current;
2   const scrollView = useRef();
3
4   const loginColorAnimation = animation.interpolate({
5     inputRange: [0, width],
6     outputRange: ['#301934', '#171717'],
7   });
8
9   const registrationColorAnimation = animation.interpolate({
10    inputRange: [0, width],
11    outputRange: ['#171717', '#301934'],
12  });
13
```

Figure 7.11: Animation Effect: Code Snippet 1: Initializing Variables




```
<View
  style={{flexDirection: 'row', paddingHorizontal: 20, marginBottom: 20}}>
  <LoginRegBtns
    backgroundColor={loginColorAnimation}
    title="Login"
    onPress={() => scrollView.current.scrollTo({x: 0})}
  />
  <LoginRegBtns
    backgroundColor={registrationColorAnimation}
    title="Sign Up"
    onPress={() => scrollView.current.scrollTo({x: width})}
  />
</View>
<ScrollView
  ref={scrollView}
  horizontal
  pagingEnabled
  showsHorizontalScrollIndicator={false}
  scrollEventThrottle={16}
  onScroll={Animated.event(
    [{nativeEvent: {contentOffset: {x: animation}}}],
    {useNativeDriver: false},
  )}>
  <LoginForm />
  <ScrollView>
    <RegistrationForm />
  </ScrollView>
</ScrollView>
```

Figure 7.12: Animation Effect Code Snippet 2: Animation in the Main Render

7.3.1.3 Testing, Issues & Resolutions

See Appendix 9.1.1 for the full testing table of this increment.

The testing stage of this increment consisted of 3 main aspects: testing the design, layout and structure of the application; testing the animation effects on the login/registration screen and on the profile home screen; and testing pressing enter on each form's input fields to ensure that the focus is directed to the next input field after pressing.

All tests had passed with the exception of the design, layout, and structure of the profile settings screen, which resulted in a user interface that was too cramped and cluttered, making it difficult for the user to view which forms are associated with which input fields and buttons. In order to resolve this issue, the animated horizontal scroll view approach has been taken, identical to that of the login/registration screen, the profile home screen, and the begin workout screen, with the same code snippet structure as displayed in the

animation UI subsection above. Here, the forms are split by profile details and login (authentication based) details. The profile details tab contains the profile details input form regarding the setting and updating of the users profile data such as their name, age, and height. The login details tab contains the forms associated with updating the user's authentication details, which includes the form for changing the email associated with the user's account, and the form for changing the user's password associated with their account. Since this tab's content includes two forms, a vertical ScrollView component has also been implemented to space out the form components and improve their usability.

7.3.1.4 Results Display

Note #1: in the case of screens that would need to display dynamic text returned by a function or lists of any sort that would need to display data from a database, placeholders or dummy text have been wrapped inside a vertical react-native <ScrollView> component, which enables vertical scrolling. The placeholder is to be replaced when said implementation begins during a later increment. As for graphs and any other forms of diagrams that require arrays of data, the 3rd party library components for these graphs have been implemented, and empty arrays have been created, to be filled with data when that implementation begins during a later increment.

Note #2: in the case of forms, many forms also utilize the vertical <ScrollView> component in order to fit the input fields and submit button onto the screen without cluttering the components. Thus, in the following screenshots, some forms may have more input fields than what is shown, and all forms do have a submit button but may not be shown in the screenshots since they are at the bottom of the form thus at the bottom of the vertical scroll.

Onboarding Screens UI Screenshots:

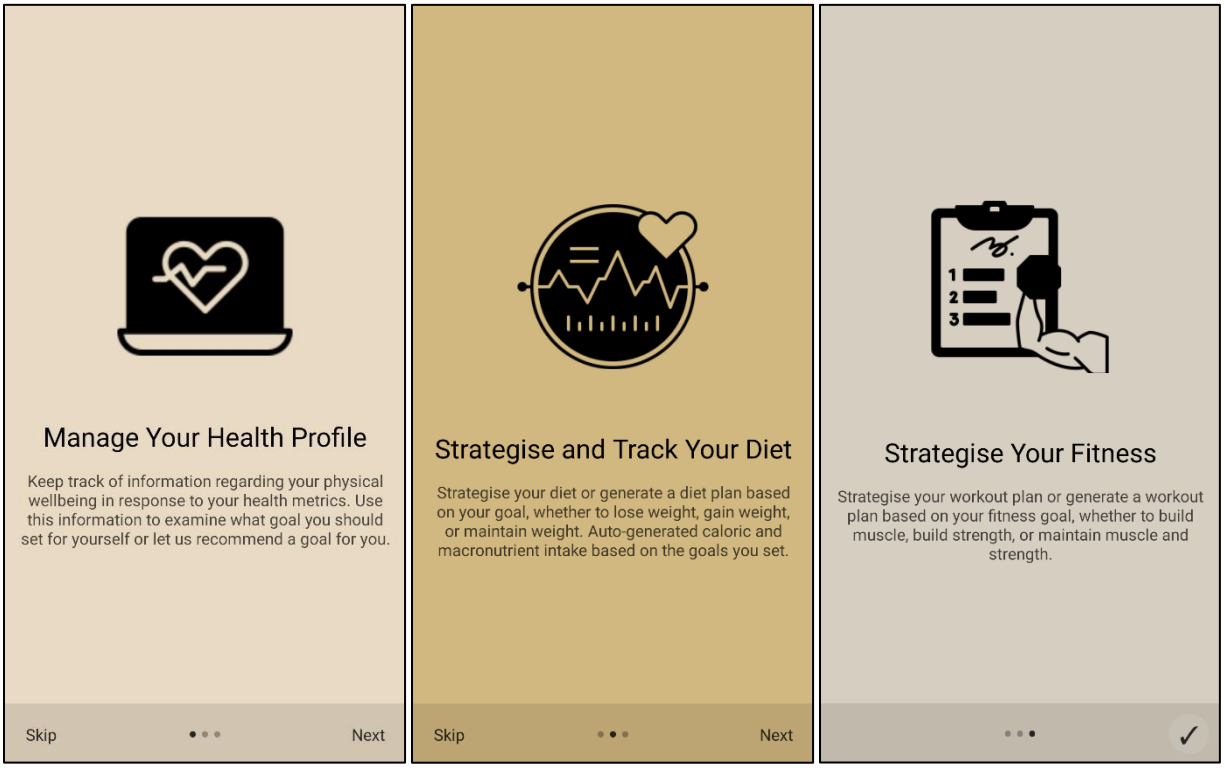


Figure: Onboarding Screens UI

Login/Registration Screen UI Screenshots:

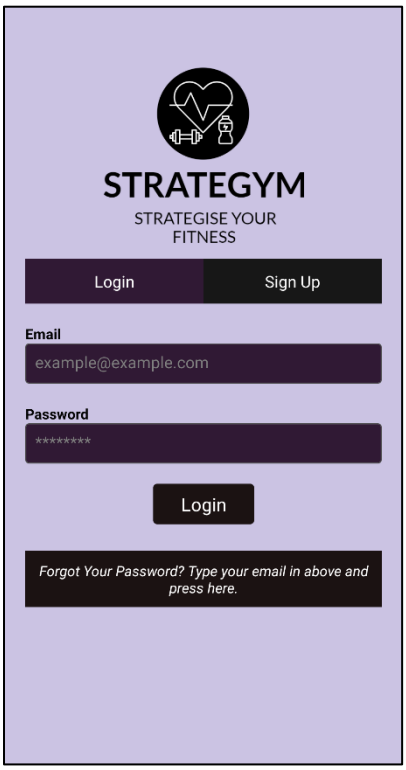


Figure: Login/Reg Screen: Login Tab UI

Figure: Login/Reg Screen: Sign Up Tab UI

Profile Screens UI Screenshots:

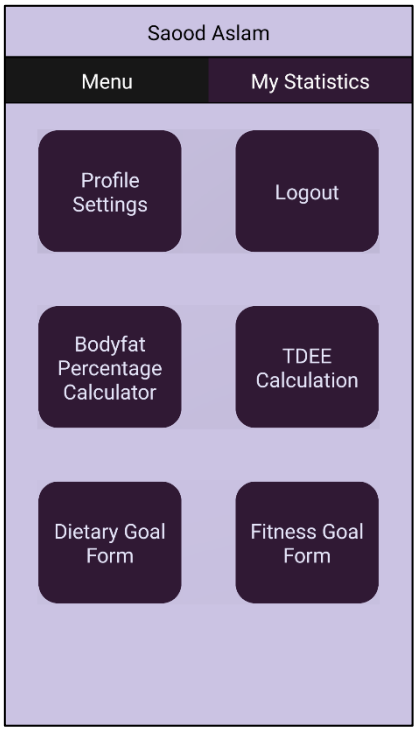


Figure: Profile Home Screen: Menu Tab UI

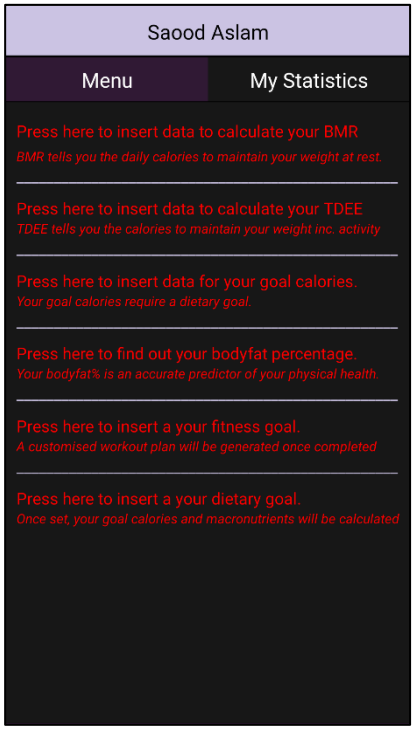


Figure: Profile Home Screen: Statistics Tab UI

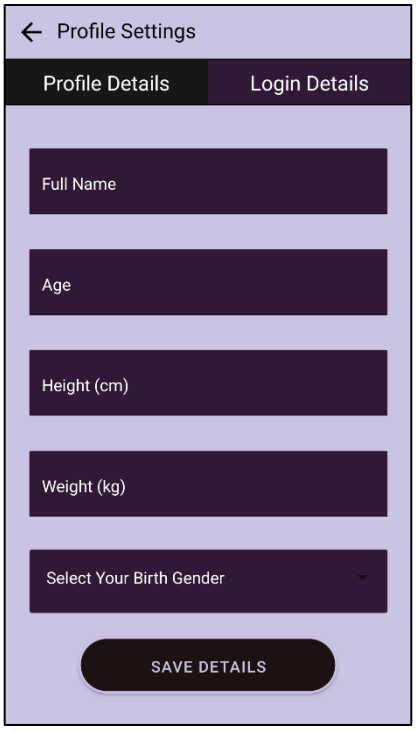


Figure: Profile Settings Screen: Profile Details Tab UI

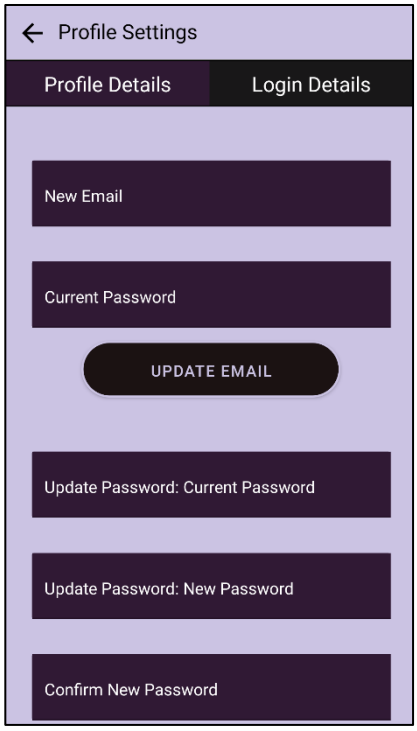


Figure: Login Details Tab UI

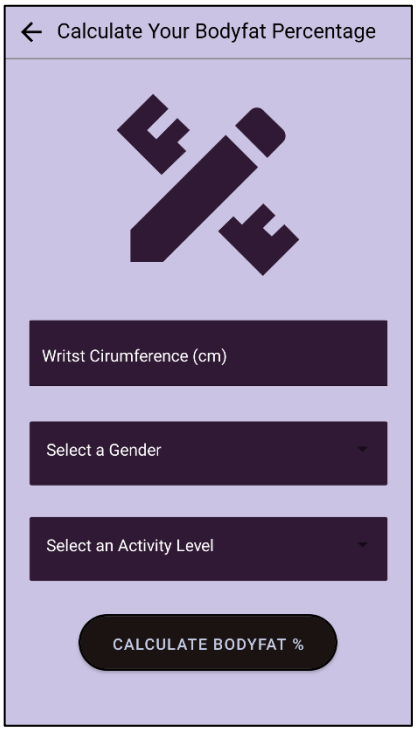


Figure: Bodyfat Percentage Calculation Screen UI

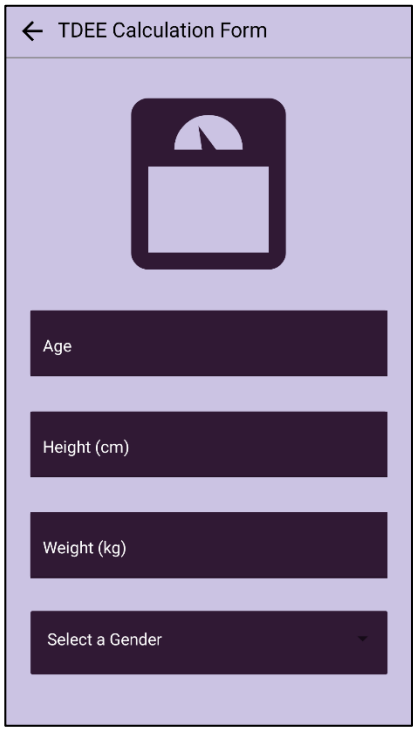


Figure: TDEE Calculation Screen UI

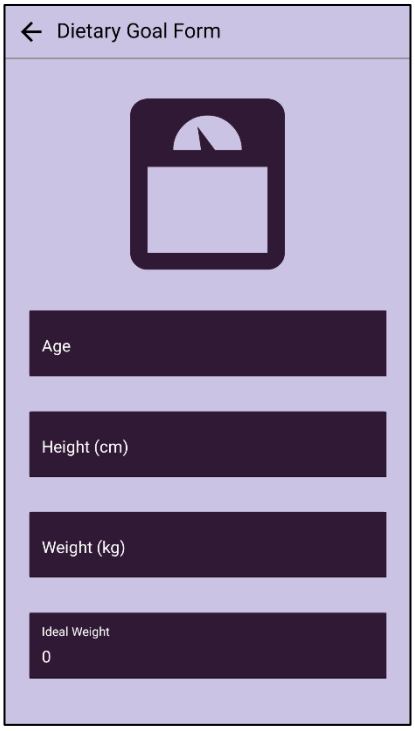


Figure: Dietary Goal Screen UI

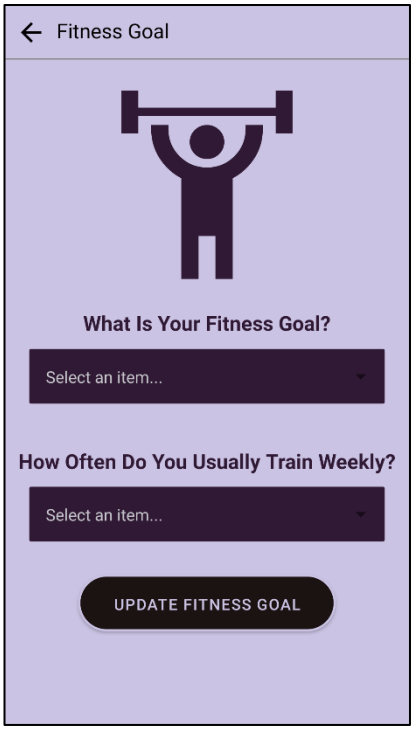


Figure: Fitness Goal Screen UI

Diet Screens UI Screenshots:

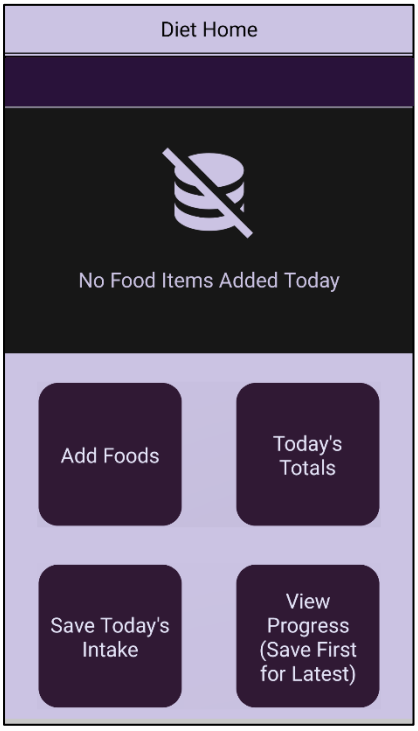


Figure: Diet Home Screen UI

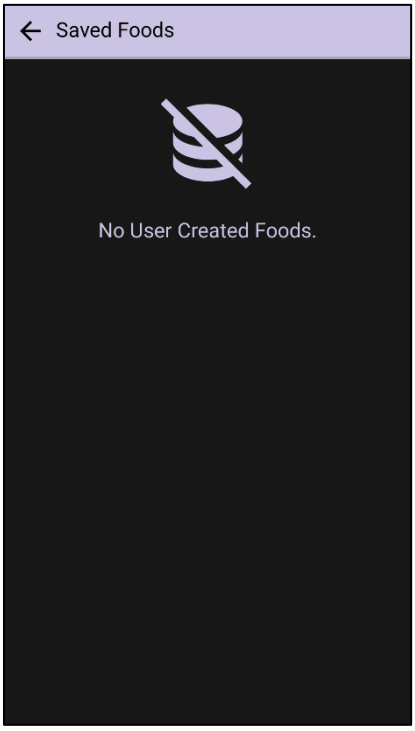


Figure: Saved Foods Screen UI

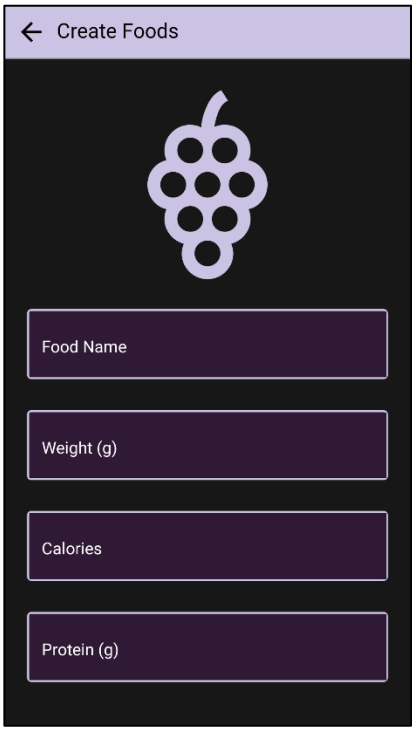


Figure: Create Foods Screen UI

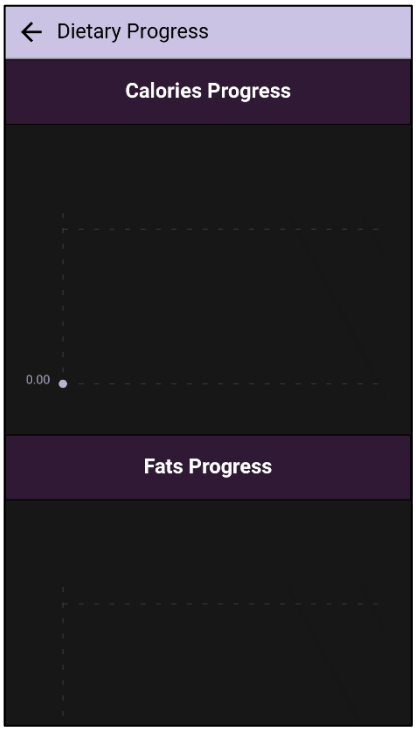


Figure: Dietary Progress Screen UI

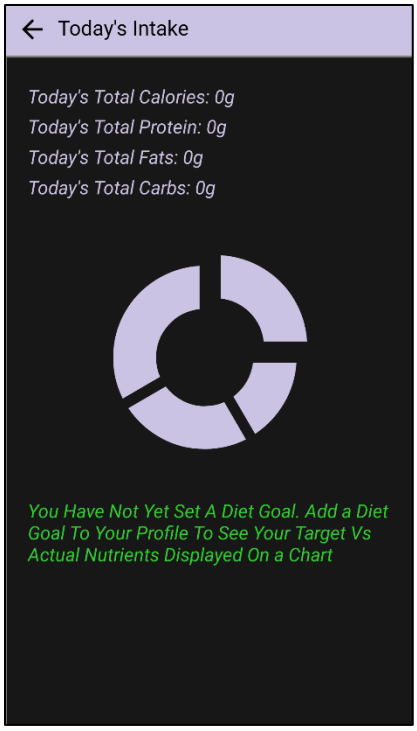


Figure: Today's Intake Screen UI

Fitness Screens UI Screenshots:

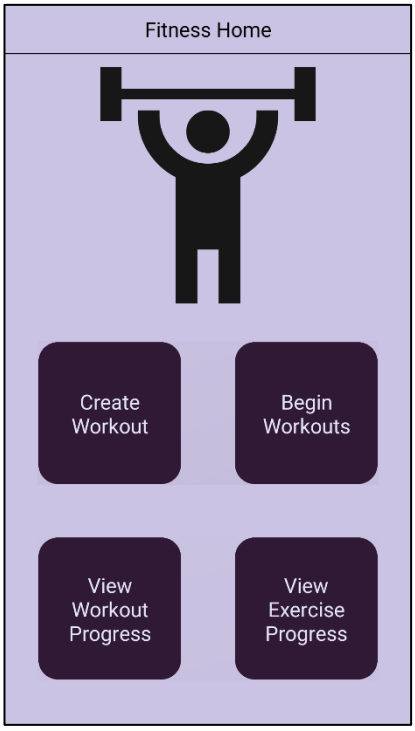


Figure: Fitness Home Screen UI

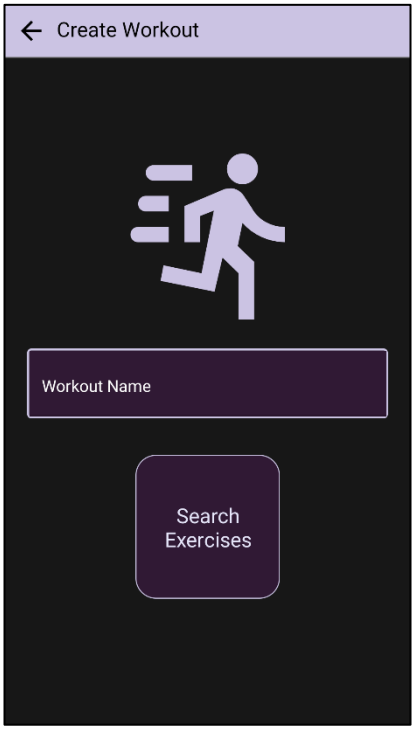


Figure: Create Workout Screen UI

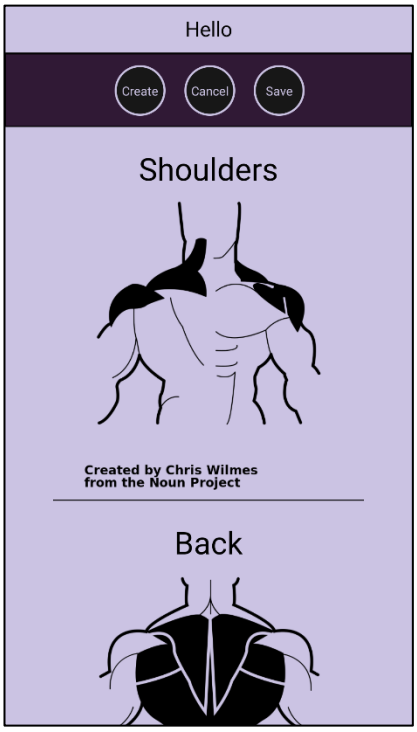


Figure: Search Exercises Screen UI

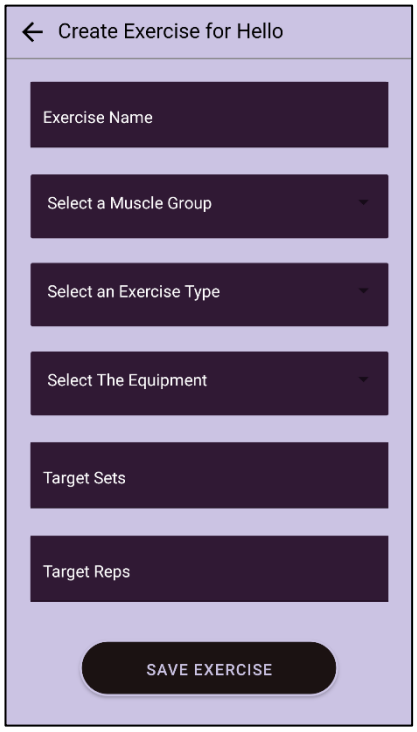


Figure: Create Exercises Screen UI

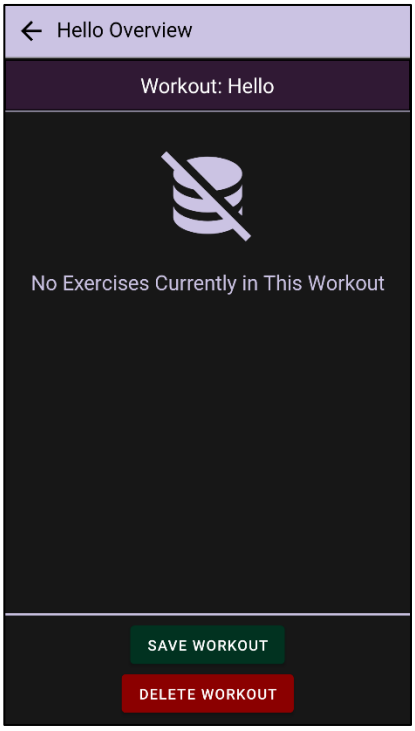


Figure: Created Workout Overview Screen UI

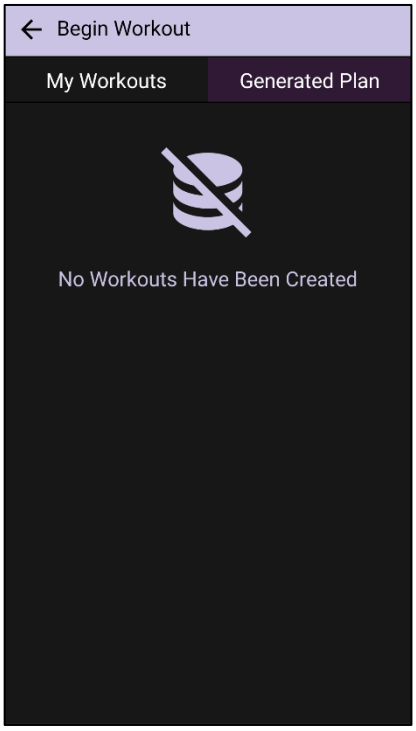


Figure: Begin Workout Screen: My Workouts Tab UI

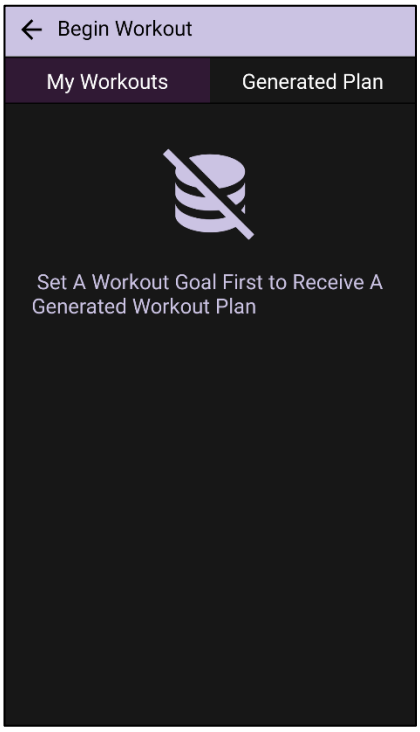


Figure: Generated Workout Plan Screen UI

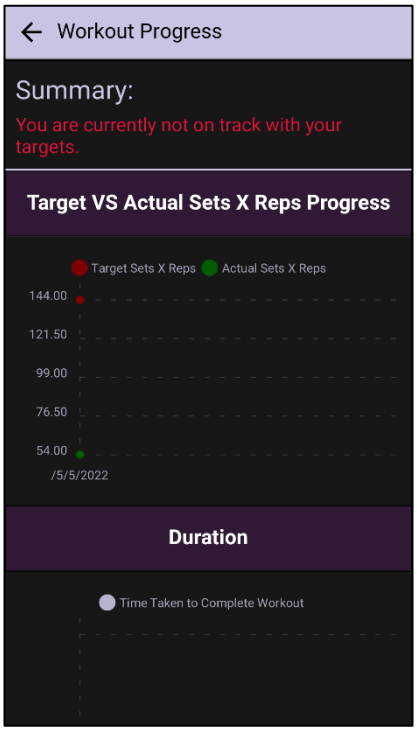


Figure: Workout Progress Screen UI

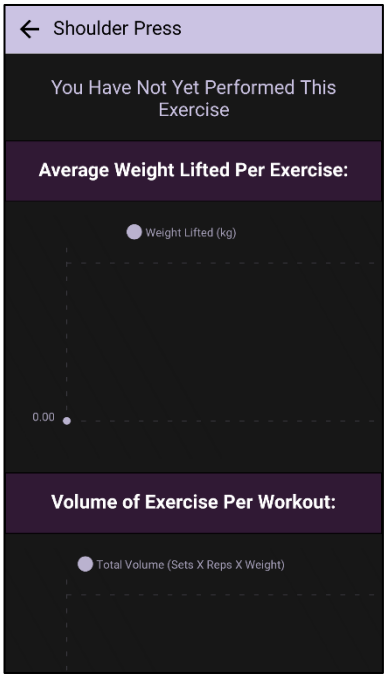


Figure: View Exercise's Progress Screen UI

7.3.2 Increment 2: Navigation Implementation and Login Functionality Implementation

This increment regards the navigational between the screens of the application, in correspondence with the navigational trees constructed in the system design.

7.3.2.1 Application Launch Cases Implementation

The first navigation component to implement is the three application launch cases, which are:

1. Initial application launch: onboarding screens → login/registration screens → main application
2. Succeeding application launch + user is not logged in: login/registration screens → main application
3. Succeeding application launch + user is logged in: directly to the main application

7.3.2.1.1 Initial Application Launch Case

In order to handle the initial application launch case, the <Stack.Navigator> component has been used to wrap the onboarding screens and login/registration screen within the same navigational stack. Then, the application checks in the local storage to see whether the current launch is the initial launch or if the

application has been launched before. If it is the initial launch, then the application is routed to the onboarding screens in the stack, and if it is not then the application is routed directly to the login/registration screens in the stack.

To implement this, a component named AuthStack has been produced which handles the main functionality within this navigational stack and is to be wrapped around the entire application so that it is the first component that is executed when the application launches.

In this component, the useState react hook is used to manage the state of the AuthStack component in terms of whether the current launch is the first launch or not (useState is applied to the set the state of the variable isFirstLaunch). useState is an in-built state management react-native hook which stores the state in memory and enables setting the state of a variable or property in a component.

To store whether the application has been launched before or not, the local storage AsyncStorage package that comes with react-native is used. When the user launches the application for the first time, it checks whether a variable called 'alreadyLaunched' which is stored in AsyncStorage is either null (meaning that the application has not been launched before) or true (meaning that the application has been launched before). If the alreadyLaunched variable is null, then:

- alreadyLaunched is set to true and stored in AsyncStorage. This is so that, for every launch thereafter the condition for it not being the application's initial launch case is executed only
- The useState variable isFirstLaunch is set to true so that the application routes to the onboarding screen in the stack

On launch, if the alreadyLaunch variable in AsyncStorage is already true, then:

- The useState variable isFirstLaunch is set to false so that the application routes to the login/registration screen in the stack.

The code regarding the checking of the alreadyLaunched variable in AsyncStorage and the setting of the useState variable is executed within a react useEffect hook. This hook runs the code inside it automatically any time the component re-renders. In this instance, anytime the user re-launches the application (causing the AuthStack component to re-render), the useEffect hook is executed to check the conditional for alreadyLaunched in AsyncStorage and to set isFirstLaunched.

```
2
3  const AuthStack = () => {
4    const [isFirstLaunch, setIsFirstLaunch] = useState(null);
5    let routeName;
6
7    useEffect(() => {
8      AsyncStorage.getItem('alreadyLaunched').then(value => {
9        if (value == null) {
10         AsyncStorage.setItem('alreadyLaunched', 'true');
11         setIsFirstLaunch(true);
12       } else {
13         setIsFirstLaunch(false);
14       }
15     });
16   }, []);
17
18   if (isFirstLaunch === null) {
19     return null;
20   } else if (isFirstLaunch === true) {
21     routeName = 'Onboarding';
22   } else {
23     routeName = 'LoginRegistrationScreen';
24   }
25
26   return (
27     <Stack.Navigator
28       initialRouteName={routeName}
29       screenOptions={{headerShown: false}}>
30       <Stack.Screen
31         screenOptions={{headerShown: false}}
32         name="Onboarding"
33         component={OnboardingScreen}
34       />
35       <Stack.Screen
36         screenOptions={{headerShown: false}}
37         name="LoginRegistrationScreen"
38         component={LoginRegistrationScreen}
39       />
40     </Stack.Navigator>
41   );
42 };
```

Figure 7.13: Initial Application Launch Case

7.3.2.1.2 Succeeding Application Launch Case:

In order to handle the second and third case where the user has launched the application before and is not logged in/is logged in, an authProvider component has been constructed that sets the global context of the application. This component is contains the functions from the Firebase auth() API, specifically the functions to log the user in (auth().singInWithEmailAndPassword) and register the user (auth().createUserWithEmailAndPassword). Here, the useState hook is also used to set the state of the variable 'user' to the logged in user if the user is logged in, or to a default value null if the user is not logged

in. This is a global state since it is within the global context authProvidor component, which therefore means that functions related to manipulating the specific user's data such as their name and age can be defined in this component and called from other components in the main application.

The Firebase functions within this component are set asynchronously, whereby the asynchronous function returns a promise until resolved without causing a bottleneck that would prevent the rest of the code in the program thereafter to execute.

```
3
4   const [user, setUser] = useState(null);
5   return (
6     <AuthContext.Provider
7       value={{
8         user,
9         setUser,
10        // FUNCTION FOR THE USER TO LOGIN TO THE APPLICATION USING FIREBASE AUTHENTICATION
11        login: async (email, password) => {
12          try {
13            await auth()
14              .signInWithEmailAndPassword(email, password)
15              .then(() => {})
16              .catch(error => {
17                if (error.code === 'auth/wrong-password') {
18                  Alert.alert(
19                    'Incorrect Password',
20                    'The password you have entered is incorrect. Please try again. ',
21                    [{text: 'OK', onPress: () => console.log('OK Pressed')}],
22                  );
23                }
```

Figure 7.14: Code Snippet from authProvidor Component

An appStack component has also been constructed to handle succeeding App launch cases. This component contains the navigational mappings of all the screens in the main application. The <Stack.Navigator> component has been used to wrap all the diet screens within their own stack, the profile screens within their own stack, and the fitness screens within their own stack. For example, Figure 6.15 shows the code for producing the stack for the profile section of the application. The initial screen the user reaches when navigating to the profile stack is the profile home screen since this is the first screen in the list of <Stack.Screen> components.

```
const ProfileStack = ({navigation}) => (
  <Stack.Navigator screenOptions={{headerShown: false}}>
    <Stack.Screen name="Profile Home" component={ProfileHomeScreen} />

    <Stack.Screen name="Profile Settings" component={ProfileSettingsScreen} />
    <Stack.Screen name="Measurements" component={MeasurementsScreen} />
    <Stack.Screen
      name="Health and Fitness Metrics"
      component={HealthFitnessMetrics}
    />
    <Stack.Screen name="TDEE Calculation" component={TDEECalcForm} />
    <Stack.Screen name="Dietary Goal" component={DietaryGoalForm} />
    <Stack.Screen name="Fitness Goal" component={FitnessGoalForm} />
  </Stack.Navigator>
);
```

Figure 7.15: Code Snippet: Setting the Profile Stack

Within this component the bottom tab navigator has also been initialized to navigate between the different sections (navigational stacks) of the main application. By using the <Tab.Navigator> and <Tab.Screen> react navigation components, the initial stack the user is first navigated to when reaching the main application can be set, and the different stacks/sections that the user reaches when pressing on a tab in the bottom navigator can be set. For example, Figure 7.16: Code Snippet: Initial Stack Set and Fitness Tab Association with Nav Stack shows the initial stack the user is navigated to when reaching the main application, and also shows the component configuration for setting the fitness tab icon in the bottom navigator with the fitness navigation stack.

```
    <Tab.Navigator
      initialRouteName="FitnessTab"
      activeColor="white"
      barStyle={{backgroundColor: '#301935', width: fullScreenWidth}}>
      <Tab.Screen
        name="FitnessTab"
        component={FitnessStack}
        options={{
          tabBarLabel: 'Fitness',
          tabBarIcon: ({color}) => (
            <MaterialCommunityIcons
              name="weight-lifter"
              color={color}
              size={26}
            />
          ),
        }}
      />
```

Figure 7.16: Code Snippet: Initial Stack Set and Fitness Tab Association with Nav Stack

The next step involves handling the cases for whether the user is logged in or not, and the actions to take thereafter. In this step, a Routes component has been constructed which is used to examine if the user is logged in by checking the 'user' state variable in the global context component AuthProvider. Once this is done, the main application component (AppStack) is executed if the user is logged is, otherwise the AuthStack component is executed which navigates the user to the login/registration screen (rather than the onboarding screen since 'alreadyLaunched' in AsyncStorage is set to true). In Figure 7.17: Code Snippet from Routes Component below, AuthContext is the name of the global context created in AuthProvider.

```
2
3   const {user, setUser} = useContext(AuthContext);
4
5     const onAuthStateChanged = user => {
6       setUser(user);
7     };
8
9       return (
10      <NavigationContainer theme={AppTheme}>
11        {user ? <AppStack /> : <AuthStack />}
12      </NavigationContainer>
13    );
14
```

Figure 7.17: Code Snippet from Routes Component

7.3.2.2 Login Functionality Implementation

The application launch case components and functions are integrated with the login/registration functionality of the login/registration screen, meaning that in order to test the application launch cases, the login/registration functionality must also be implemented. For this reason, both the application launch cases and login functionality are implemented in the same increment.

Initially, functions have been created in the login and registration form components that check the validity of the users' inputs. For example, a function called validEmailCheck has been constructed which takes the email input from the user as a parameter and checks it against the email regular expression to ensure a valid email has been inserted. If a valid email is not inserted, the function returns false (Figure x.x). Functions for checking if the user has inserted a valid string as their name, a password that is longer than 8 characters, and whether the confirm password input is the same as the password input have all been constructed too in the same manner, returning false if these conditions are not met.

```
const validEmailCheck = value => {
  const checkEmail =
    /^([A-Za-z0-9_\-\.])+\@([A-Za-z0-9_\-\.])+\.([A-Za-z]{2,4})$/; //regular expression for email validation
  return checkEmail.test(value);
};

```

Figure 7.18: Code Snippet: Function for Checking Valid Email Input

If any of these conditions are not met, a function is executed which sets the state of an error variable, which is a string containing the corresponding error message (for example “Please enter a valid email”). This error message is set to display on the screen for 2500 milliseconds after which it is hidden, and every time the user presses on submit and the form is invalidated, the error variable state is updated with the corresponding string and shown on the screen again for 2500 milliseconds (Figure 7.19: Code Snippet: Function for Outputting Error Message on Invalidated Form).

```
const [error, setError] = useState('');

// FUNCTION FOR DISPLAYING AN ERROR MESSAGE TO THE USER IF THE USERS FORM ISN'T VALIDATED
const outputError = (error, updateState) => {
  updateState(error);
  setTimeout(() => {
    updateState('');
  }, 2500);
};
```

Figure 7.19: Code Snippet: Function for Outputting Error Message on Invalidated Form

The corresponding Firebase function to login or register the user has been imported on both the login form component and registration form component from the global authentication context AuthProvider component. This function has then been called in the onPress method of the login and sign-up buttons. The onPress methods are constructed such that the login/registration functions only run if the input validation functions are met. For example, the code snippet below shows the onPress method on the login button. Here, a validation function validLoginForm has been created which amalgamates all the other validation functions regarding the users input to check if any of them return false. If the amalgamated validLoginForm function returns true, then the login function is executed (Figure 7.20: Code Snippet: Validate User Then Run Login Func

```
// AMALGAMATED VALIDATION FUNC
 const validLoginForm = () => {
   if (!validLoginObject(userInfo))
     return outputError(
       'please insert your email address and password',
       setError,
     );

   if (!validEmailCheck(email))
     return outputError('Please insert a valid email address', setError);

   if (!password.trim || password.length < 8)
     return outputError('Please insert a valid password', setError);

   return true;
 };

//ON PRESS METHOD IN RENDER:
   onPress={() => {
     if (validLoginForm()) {
       login(userInfo.email, userInfo.password);
     }
   }}
```

Figure 7.20: Code Snippet: Validate User Then Run Login Function

The login and registration functions in the AuthProvider component, along with all other functions that are to be defined in this component, are asynchronous functions (see Figures Figure 3.6: Flow Diagram of the Main Thread and Async Thread and Figure 3.7: Flow Diagram of Firebase Asynchronous Function to Retrieve Data from Firestore). The await part of the function is attached to the main Firebase function of authenticating the user with their login or registration details. This prevents the rest of the asynchronous function from executing until the awaited function is complete. The .then() adjunction defines what happens after the await authentication function is complete. In the case of the user registration function, the user's Firestore document would need to be initialized to contain all the profile data that is to be stored in the users unique Firestore document ID in the .then() adjunction after the registration authentication is complete. In order to ensure the users document ID in Firestore is unique and accessible by other screens, the name of the document has been set to the users UID (unique identification).

```
register: async (fullName, email, password) => {
  try {
    await auth()
      .createUserWithEmailAndPassword(email, password)
      .then(() => {
        auth().currentUser.updateProfile({displayName: fullName});
      })
      .then(() => {
        firestore()
          .collection('users')
          .doc(auth().currentUser.uid)
          .set({
            fullName: fullName,
            email: email,
            age: '',
            weight: '',
            height: '',
            neckCircum: '',
            chestCircum: '',
            abdomenCircum: '',
            hipCircum: '',
            thighCircum: '',
            kneeCircum: '',
            ankleCircum: '',
            bicepCircum: '',
            forearmCircum: '',
            wristCircum: '',
            gender: '',
            bodyfatPercentage: '',
            idealWeight: '',
            BMR: '',
            gender: '',
            activityLevel: '',
            bodyfatPercentage: '',
            TDEE: '0',
            idealWeeks: '',
            goalCalories: '',
            goalProtein: '',
            goalCarbs: '',
            goalFats: '',
            trainingFrequency: '',
          });
      })
```

Figure 7.21: Code Snippet: Async Register; Await createUserWithEmailAndPassword API; .then() Adjunction

The .catch(error) adjunction is the error handling aspect of an asynchronous function which is required by all functions that run asynchronously. It handles any errors that may occur from the Firebase API functions above (authentication function and adding data to firebase function). Google has listed the error code "auth/email already-in-use" for the Firebase authentication API (Google, 2022). For this error, the react-native <Alert> component has been implemented which is displayed if this error code occurs and informs the user that the email is already registered, and to try logging in on the login screen. In all other error instances, the errors are logged in the console for testing and debugging purposes if they occur. The code snippet below is the .catch(error) of the registration function, (Figure 7.22: Code Snippet: .catch(error)). The asynchronous function for logging in follows the same structure as this entire registration function but with the login Firebase API instead and without the initialization of the user's Firestore.

```
            trainingFrequency: '',
          });
      })

      .catch(error => {
        if (error.code === 'auth/email already-in-use') {
          Alert.alert(
            'Email Already Registered',
            'The email you are trying to register with has already been registered, try signing in with it or signing up with another email.',
            [{text: 'OK', onPress: () => console.log('OK Pressed')}],
          );
        }
      });
  } catch (e) {
    console.log(e);
  }
},
```

Figure 7.22: Code Snippet: .catch(error)

Once the user is logged in or registered through these functions, the 'user' state variable in the global context is updated to the user that is logged in, which then changes the routing in the Route component from to the AppStack which is contains the navigation stack of the main application.

Additionally, the logout Firebase API has also been implemented in the AuthProvider global context component, following the same structure as the login and registration functions. The logout API logs the device out of Firebase, which sets the state of 'user' in authProvider to null, routing the user back to the login/registration screen of the AuthStack.

```
1
2
3
4
5
6
7
8
    logout: async () => {
      try {
        await auth().signOut();
      } catch (e) {
        console.log(e);
      }
    },
```

Figure 7.23: Code Snippet: Logout Function in AuthProvider

This logout function has been called in the profile home screen component from the AuthProvider component on the onPress method of the logout button displayed on this screen.

```
const {logout} = useContext(AuthContext);
return (
<TouchableOpacity style={{}} onPress={() => logout()}>
                    <View>
                      <Text>
                        Logout
                      </Text>
                    </View>
</TouchableOpacity>
    )
```

Figure 7.24: Code Snippet: onPress Method For Rendered Logout Component on Profile Home Screen

7.3.2.3 Navigating Between Screens in the Same Stack

The react-navigation library includes a useNavigation function which enables navigating between screens within the same stack. This function has been utilized on the onPress methods of all buttons and touchable components that are associated with navigating around the stack. An example use case of this is shown in the code snippet below. This shows the navigation function applied to the onPress method of the profile settings rendered component on the profile home screen. When the user presses this touchable component, they are navigated to the screen with the title “Profile Settings” which has been defined in the AppStack component.

```
const navigation = useNavigation();
return (
        <TouchableOpacity
           onPress={() => navigation.replace('Profile Settings')}>
           <View>
             <Text>
               Profile Settings
             </Text>
            </View>
         </TouchableOpacity>

        )
```

Figure 7.25: Code Snippet: useNavigation Function on Profile Settings Button

Also, the useNavigation function has been implemented on the custom header with back button component that was produced in increment 1. Now when this component is implemented on another screen, it takes the title of the previous screen as a prop and navigates to that screen on the onPress method for the back button. Figure 7.26: Code Snippet: Custom Header Component with Back onPress Prop Method6 shows the

code snippet implemented to the custom header with back component, whilst Figure 7.27: Code Snippet: TDEE Calculation Screen with Cusom Header Component, Passing Profile Home As The Previous Screen Name7 shows an example of this component being used of the TDEE calculations screen, with the prop "Profile Home" being passed so that the back button on this screen's header navigates back to the profile home screen.

```
const navigation = useNavigation();
return (
    <View>
      <TouchableOpacity
        onPress={() => navigation.replace(props.backNavScreen)}>
        <MaterialCommunityIcons
          name="arrow-left"
          color="black"
          size={30}>
          </MaterialCommunityIcons>
      </TouchableOpacity>
    </View>
      )
```

Figure 7.26: Code Snippet: Custom Header Component with Back onPress Prop Method

```
return (
    <View style={{backgroundColor: '#CBC3E3', flexGrow: 1,}}>
        <CustomHeaderWithBack
            pageName="TDEE Calculation Form"
            backNavScreen="Profile Home"
    />
       )
```

Figure 7.27: Code Snippet: TDEE Calculation Screen with Cusom Header Component, Passing Profile Home As The Previous Screen Name

7.3.2.4 Testing, Issues & Resolutions

The full testing table for this increment is available in Appendix 9.1.2.

The testing phase of this increment consisted of testing the following:

- The three application launch cases
- The error messages displayed for the login/registration form validation
- The alert message displayed when the user tries to register with a taken email

- The buttons on screens that are associated with navigating up and down the navigation stacks of the main application
- The logout button on the profile home screen

All tests in this testing phase had passed with the exception of one, being the alert message displaying when the user tries to register with a taken email. In order to address this issue, the code that had been implemented was examined against the error code provided by Google, and it was found that there was a typo in the conditional statement that the Alert component was braced within. The code had the if statement as:

```
if (error.code === 'auth/email already-in-use') { …
```

But should instead be:

```
if (error.code === 'auth/email-already-in-use') { …
```

This issue has been addressed, and the alert box has since been displayed when the user registers with a taken email, as expected.

7.3.2.5 Results Display

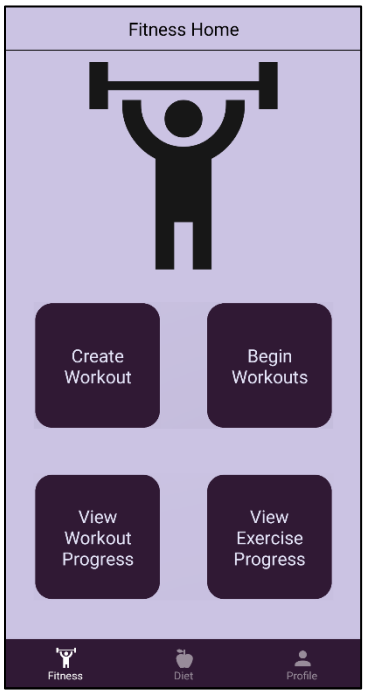


Figure 7.28: Screenshot Example of the Bottom Nav Bar Implemented Across All Screens in AppStack Main Application

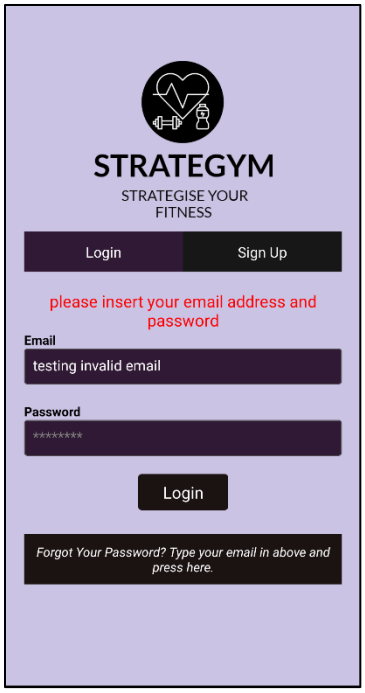


Figure 7.29: Screenshot Example of Invalid Login/Reg Form Error Message Displayed

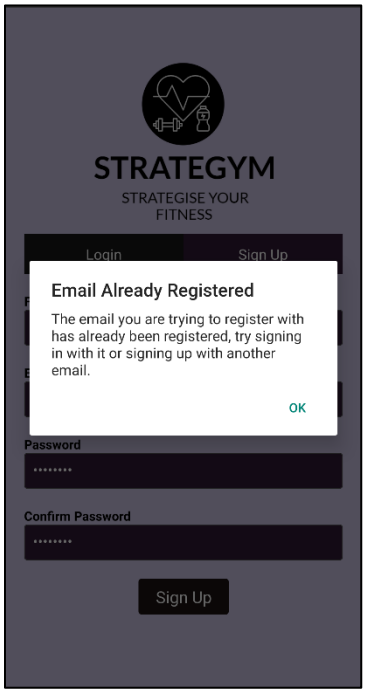


Figure 7.30: Screenshot of catch.error Alert If The Email is Already Registered

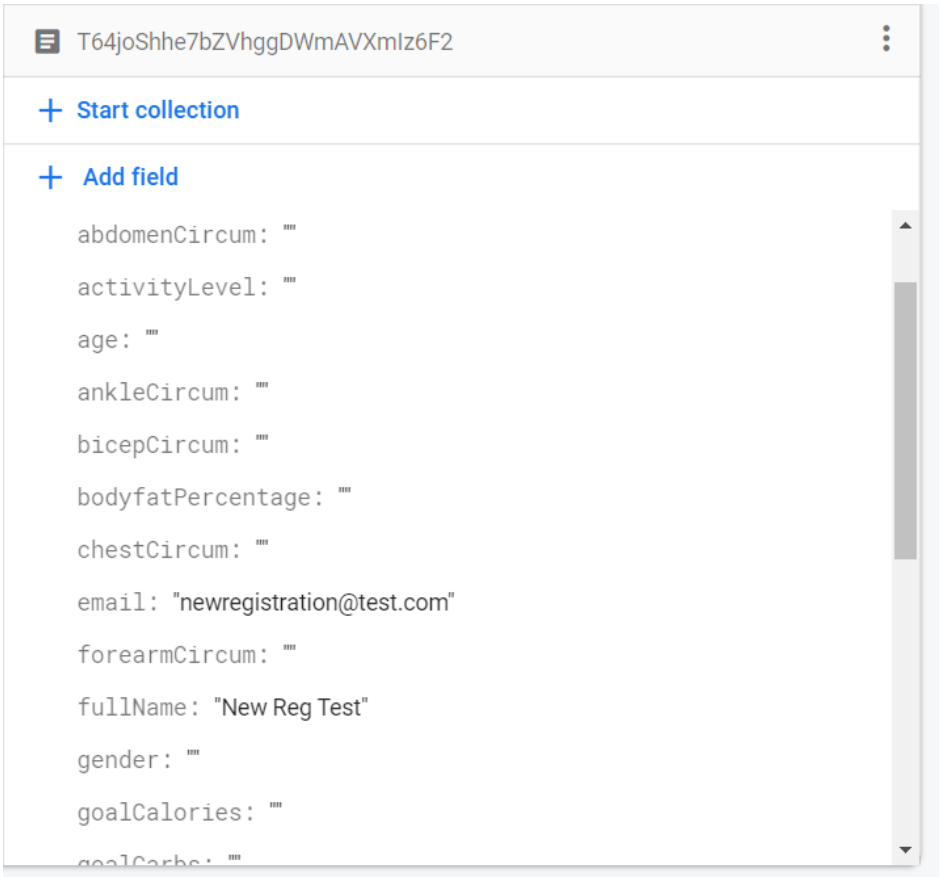


Figure 7.31: Firestore GUI Displaying New Document for Newly Registered User

7.3.3 Increment 3: Profile Side Implementation

The profile side functionality includes updating and saving all the data related to the profile of the user, as displayed in the Figure 7.31: Firestore GUI Displaying New Document for Newly Registered User. This section includes the TDEE calculation form; bodyfat percentage prediction form; profile settings form; dietary goal form and fitness goal form. All profile data is stored in Firebase, so that if the user decides to log into the application on another device, their profile data is accessible. Furthermore, as mentioned in section 3 (the planning solution section), Firebase's authentication API meets the encryption standards which ensures that user data is secured. All forms in this section update the data saved in the users unique Firestore document, with the exception of the login details form displayed on the profile settings screen, which updates the users authentication details.

7.3.3.1 Authentication Forms Functionality

There are two authentication forms on the profile settings screen, where one is to update the users email and the other is to update the users password. Firebase provides a function to handle instances where the user would like to update their email address (.updateEmail) and another function to handle instances where the user would like to update their password (.updatePassword). The update email function requires the user to provide their login details to log in again, and then executes the function, taking the new email as a parameter. The update password function requires the user to log in with their login details again and then executes the function, taking the new password as a parameter. Requiring the login details again provides an extra layer of security as it prevents cases where the user is already logged into their account on their device but someone else has access to their device and is trying to change their login details. A code snippet displaying the structure of these functions implemented in the AuthProvider global context component is displayed below (Figure 7.32: Code Snippet: Update User Password Function in AuthProvider Component)

```
updateUserPassword: async (oldPassword, newPassword) => {
  try {
    await auth()
      .signInWithEmailAndPassword(auth().currentUser.email, oldPassword)
      .then(() => {
        auth().currentUser.updatePassword(newPassword);
      })
      .then(() => {
        Alert.alert(
          'Password Successfullu Updated',
          'You have successfully updated your password',
          [{text: 'OK', onPress: () => console.log('OK Pressed')}],
        );
      })
      .catch(error => { //Error Code From Google Firebase Website
        if (error.code === 'auth/network-request-failed') {
          Alert.alert(
            'No Network Connection',
            'Your device currently has no network connection. Please establish a connection and try again ',
            [{text: 'OK', onPress: () => console.log('OK Pressed')}],
          );
        }
        if (error.code === 'auth/wrong-password') { //Error Code From Google Firebase Website
          Alert.alert(
            'Incorrect Password',
            'The password you have entered as your current password is incorrect. Please try again. ',
            [{text: 'OK', onPress: () => console.log('OK Pressed')}],
          );
        }
      });
  } catch (e) { // console log any other errors for debugging
    console.log(e.code, e.message);
  }
},
```

Figure 7.32: Code Snippet: Update User Password Function in AuthProvider Component

On the profile settings screen where the forms are displayed, the update email and update password functions are imported from AuthProvider. These functions are the placed inside the braces of a conditional validation function which checks if the inputs in the text fields of the form are valid and are then executed if the validation condition is met. The current password is sent as parameters to the function as well as the new email or password, so that the initial step of logging the user in again with the details they've inputted can be executed, followed by the new email or new password function. The following code snippet shows this implementation for the onPress method of the update password button as an example (Figure 7.33: Code Snippet: Update Password onPress Method on the Rendered Submit Button Component on the Profile Home Screen):

```
<Button
  onPress={() => {
    if (
      validPasswordInput(
        authData.password,
        authData.newPassword,
        authData.confirmNewPwd,
      )
    ) {
      updateUserPassword(password, newPassword);
    }
  }}>
  Update Password
</Button>
```

Figure 7.33: Code Snippet: Update Password onPress Method on the Rendered Submit Button Component on the Profile Home Screen

On forms throughout the entire main application, the valid form function which checks if there are valid inputs in each input field is set up to output the error message with the same design as the wireframe design produced in section 6. In order to achieve this, the library react-native-flash-message has been imported to each screen, which contains the function showMessage. This showMessage function displays the error message at the top of the screen styled as a notification, and appears for (RESEARCH) seconds. An example of the implementation of this function displaying an error message when the update email form is invalid is shown below (Figure 7.34: Code Snippet: Show Message Function Inside the Email Form Validation Function).

```
function validUpdateEmailForm(passwordForEmailReset) {
  if (!validEmailCheck(email)) {
    return showMessage({
      message: 'Please Enter a Valid Email Address',
      type: 'info',
      color: 'black',
      backgroundColor: 'red',
    });
  }

  if (passwordForEmailReset.trim() == '') {
    return showMessage({
      message: 'Please Enter Your Current Password',
      type: 'info',
      color: 'black',
      backgroundColor: 'red',
    });
  } else {
    return true;
  }
}
```

Figure 7.34: Code Snippet: Show Message Function Inside the Email Form Validation Function

7.3.3.2 Firestore Forms Functionality

The rest of the forms in the profile side of the application regard manipulating data inputs from the user and updating the data to the user's unique Firestore document. These forms are the:

- Profile Details form
- TDEE Calculation Form
- Dietary Goal Form
- Fitness Goal Form
- The Bodyfat Percentage Form

Before the data is uploaded to Firestore, the data is first used to make calculations that are related to the nature of the form. For example, the TDEE calculations form first uses the Harris-Benedict equation, identified in the problem solution, to calculate the users BMR based on their age, height and weight and gender inputs and then multiplies this equation by the activity level input by the user. After this is complete, the function to update all the corresponding data in Firestore is then executed. A full diagrammatic

representation of the flow from the user pressing submit on these Firestore-updating forms to the function to update the Firestore data is shown below (Figure 7.35: Input Data Flow for Firestore-Updating Forms

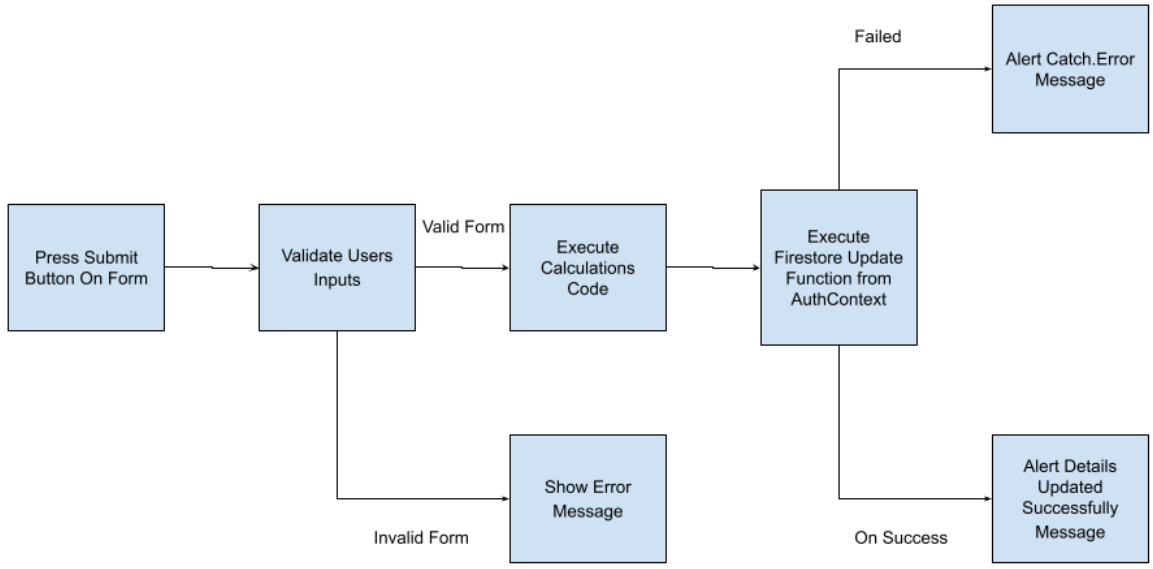


Figure 7.35: Input Data Flow for Firestore-Updating Forms

Each form follows the same process as demonstrated in the diagram above, with the corresponding equations defined in the problem solution, meaning the diet goal form implements the equations for calculating the goal calories and goal macronutrients; the TDEE form implements the Harris-Benedict equation etcetera. An example of this process is shown in the code snippets below. Here, firstly the Firestore update function is defined with the catch.error messages in the AuthProvider component and is imported into the TDEE calculations screen component (Figure 7.36: Code Snippet: Update User TDEE Function in AuthProvider); then the onPress method for the button on the TDEE calculations form is defined to check the validation for the form → execute the equations to calculate the BMR and TDEE → and then execute the Firestore update function that has been imported from AuthProvider, taking all of the related data as parameters and updating these in the Firestore document (Figure 7.37: Code Snippet: On Press Method on Submit Button Component for Calculating and Updating User TDEE).

```
updateUserTDEE: async (
  ageInput,
  heightInput,
  weightInput,
  genderInput,
  activityLevelInput,
  BMR,
  TDEE,
) => {
  try {
    await firestore()
      .collection('users')
      .doc(auth().currentUser.uid)
      .update({
        age: ageInput,
        height: heightInput,
        weight: weightInput,
        gender: genderInput,
        activityLevel: activityLevelInput,
        BMR: BMR,
        TDEE: TDEE,
      })
      .then(() => {
        Alert.alert(
          'Details Updated Successfully',
          'You have successfully updated your profile details',
          [{text: 'OK', onPress: () => console.log('OK Pressed')}],
        );
      })
      .catch(error => {
        console.log(error.code, error.message);
      });
  } catch (e) {
    console.log(e.code, e.message);
  }
},
```

Figure 7.36: Code Snippet: Update User TDEE Function in AuthProvider

```
onPress={() => {
  if (
    validMetricsForm(
      userMetrics.age.trim(),
      userMetrics.height.trim(),
      userMetrics.weight.trim(),
      userMetrics.gender,
    )
  ) {
    if (userMetrics.gender.trim() === 'Male') {
      userMetrics.BMR =
        66.47 +
        13.75 * userMetrics.weight.trim() +
        5.003 * userMetrics.height.trim() -
        6.75 * userMetrics.age.trim();
    }
    if (userMetrics.gender.trim() === 'Female') {
      userMetrics.BMR =
        655.1 +
        9.563 * userMetrics.weight.trim() +
        1.85 * userMetrics.height.trim() -
        4.676 * userMetrics.age.trim();
    }

    userMetrics.TDEE = userMetrics.BMR * userMetrics.activityLevel;

    updateUserTDEE(
      userMetrics.age.trim(),
      userMetrics.height.trim(),
      userMetrics.weight.trim(),
      userMetrics.gender.trim(),
      userMetrics.activityLevel,
      userMetrics.BMR,
      userMetrics.TDEE,
    );
    navigation.replace("Profile Home")
  }
}}
```

Figure 7.37: Code Snippet: On Press Method on Submit Button Component for Calculating and Updating User TDEE

7.3.3.3 Additional Feature: Neural Network Bodyfat Percentage Predictor:

An additional feature that has been implemented in the application in this increment is a trained neural network which takes the inputs from the bodyfat percentage form, runs it through a trained neural network and outputs an estimated bodyfat percentage based on the user's measurements. As mentioned during the planning and solutions section (section 3) of this project report, the incentive behind implementing this feature is to improve the accuracy of determining the user's current health and fitness status, since other methods of measuring health and fitness such as BMI are inaccurate.

Two neural networks have been trained in this implementation, one on the male's bodyfat dataset and the other on the women's bodyfat dataset, which were both discovered in the solution planning section of this report. All relevant files and data related to the neural network are placed in a neural network folder in the root directory of the project (see Appendix 9.5 for program listings). As covered by Chris Dawson in part b of the Computer Science course at Loughborough University, there are 3 main stages to training a neural network:

1. Data cleansing
2. Data standardisation
3. Configuration and training of the neural network on the cleansed and standardised dataset.

7.3.3.3.1 Data Cleansing:

The first step is data cleansing. This involves getting rid of columns in the dataset that are not of use as an input or output in the neural network. In this case, the column in the men's bodyfat dataset that is not of use is the underwater density column, since this is not an input that will be required from the user, and the only output node of the neural network is the bodyfat percentage column. The column in the women's bodyfat dataset that is not of use is the BMI column and identifier column, which leaves the rest of the columns in this dataset identical to that of the cleansed men's bodyfat percentage dataset, thus requiring the same inputs from both male and female users using the application, which is practical as then only one bodyfat percentage form is required in the application.

7.3.3.3.2 Data Standardisation:

The next step is data standardisation, which involves standardising the datasets to values between 0 and 1, as the nature of the sigmoid activation function used only outputs a value between 0 and 1. The output of the neural network can then be de-standardised through re-arranging the chosen standardisation formula to find the real value. The standardisation formula presented by Chris Dawson is the following (Figure 7.38: Standardisation Formula for Neural Network Training):

$$S_i = \frac{R_i - Min}{Max - Min}$$

S_i : Standardised value

R_i : Raw value

Figure 7.38: Standardisation Formula for Neural Network Training

In order to find the minimum and maximum value of each column, the MIN and MAX operators in Excel have been used, followed by applying the above formula on each value in the columns with the minimum/maximum value in that column. The following figures are an example of the MIN and MAX operators being applied to the Age column, followed by the standardisation formula being applied to a value in the Age column (Figures: Figure 7.39: MAX Operator on the Age ColumnFigure 7.40: Min Operator on the Age Column,Figure 7.41: Standardisation Formula On a Value in The Age Column).

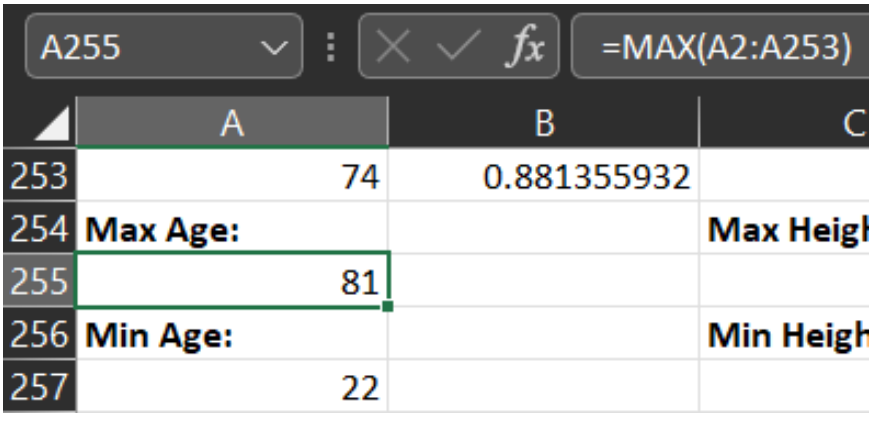


Figure 7.39: MAX Operator on the Age Column

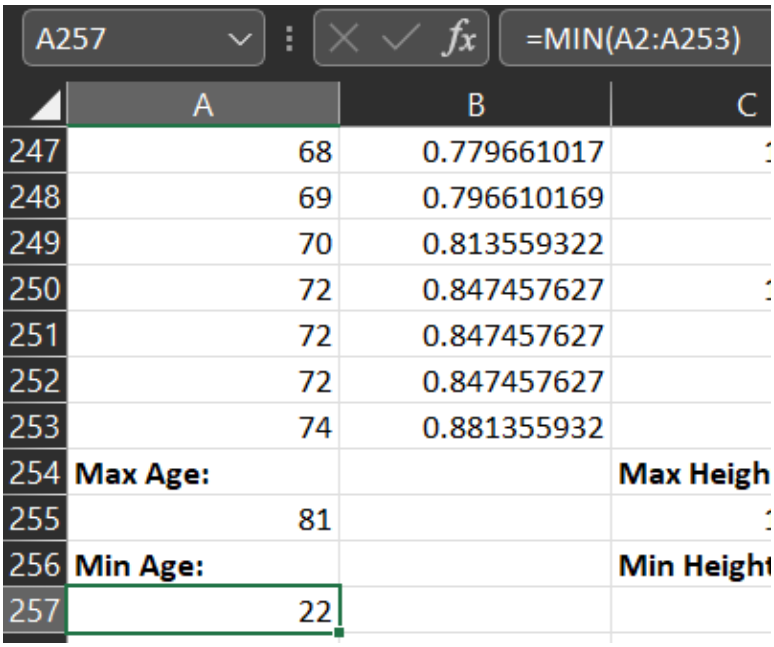


Figure 7.40: Min Operator on the Age Column

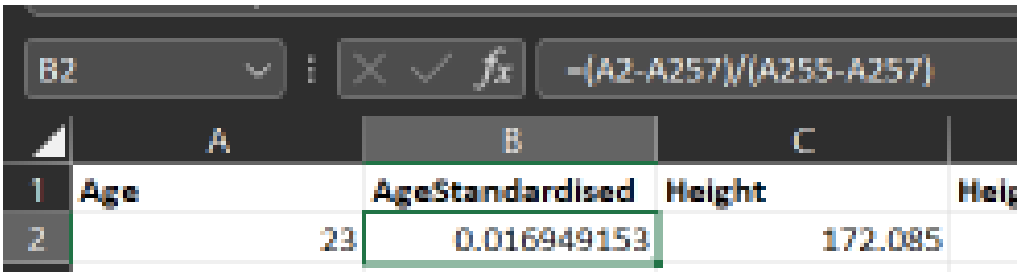


Figure 7.41: Standardisation Formula On a Value in The Age Column

In order to de-standardise the output bodyfat percentage prediction, the standardisation formula must be re-arranged to find Ri (raw value). The following de-standardised formula has been constructed:

$$Ri = Si \times (Max - Min) + Min$$

Equation 7.2: De-standardisation Formula For Bodyfat Percentage Output

7.3.3.3.3 Configuration and Training

The next step is the configuration and training of the cleansed and standardised dataset. The NPM library includes a neural network package called brain.js, which handles the mathematical complexity of the feed-forward neural network with the backpropogation algorithm, whilst also enabling the developer to set the configurations of the neural network. Furthermore, the fs (file system) NPM package enables the developer to save the trained neural network in the local directory as a JSON file, which can then be used later for the actual implementation of the trained neural network in the application. Brain.js also comes with a cross validation function, which internally splits the dataset it is trained on into a training set and a test set, without the developer having to program this function manually.

In order to implement the code for training the neural network, firstly the cleansed and standardised datasets had been converted from an Excel Spreadsheet file to a .JSON file, so that the properties can be imported as inputs and outputs in the training data JavaScript array. The example below shows this for the men's bodyfat dataset (Figure 7.42: Implementing the Men's Standardised Dataset).

```
// GETTING THE BRAIN.JS LIBRARY
  const brain = require('brain.js')



//THE MENS TRAINING DATA:
const data = require('./mensBFStandardisedData.json');
const trainingData = data.map(item => ({
  input: [
    parseFloat(item.AgeStandardised),
    parseFloat(item.HeightStandardised),
    parseFloat(item.WeightStandardised),
    parseFloat(item.NeckCircumStandardised),
    parseFloat(item.ChestCircumStandardised),
    parseFloat(item.AbdomenCircumStandardised),
    parseFloat(item.HipCircumStandardised),
    parseFloat(item.ThighCircumStandardised),
    parseFloat(item.KneeCircumStandardised),
    parseFloat(item.AnkleCircumStandardised),
    parseFloat(item.BicepCircumStandardised),
    parseFloat(item.ForearmCircumStandardised),
    parseFloat(item.WirstCirumStandardised),
  ],
  output: [parseFloat(item.BodyFatStandardised)],
}));

```

Figure 7.42: Implementing the Men's Standardised Dataset

Following this, the configuration of the neural network had been implemented, which involves setting the number of hidden nodes, the activation function to be used, the number of input nodes, the number of output nodes, the number of iterations to train the neural network on, and the learning rate of the neural network.

Though there is no absolute method, a rule of thumb commonly used to determine the number of hidden nodes in a neural network is for it to be 2/3 the size of the input layer + the size of the output layer. Thus 2/3 of 13 plus 1 is between 9 and 10, therefore 9 hidden nodes have been used for the neural network. The activation function used is the sigmoid activation function, which was covered by Chris Dawson extensively in part b of this course. Since brain.js takes care of the mathematical complexity of the neural network, the need for the sigmoid equation to be implemented manually isn't necessary. The learning rate and iterations set are 150000 iterations with a learning rate of 0.01 for the female neural network, and 200000 iterations with a learning rate of 0.005 for the male neural network. If there is enough time left-over when this project is completed, these two properties can be treated as dependent variables changing on each run of training the neural networks, to determine the number of iterations and learning rate that leads to the most accurate predictions. All-in-all the implementation of training the neural network with backpropogation and cross-validation, and the configuration is shown below (Figure 7.43: Implementing the Training Config of the Mens Bodyfat Neural Network and Outputting the Trained Neural Network as a .JSON File Figure 7.44: Implementing the Training Config of the Women's Bodyfat Neural Network and Outputting the Trained Neural Network as a .JSON File).

```
// CONFIGURATION FOR THE NERUAL NETWORK AND TRAINING CONFIGURATION:
const netOptions = {
  hiddenLayers: [9],
  activation: 'sigmoid',
  inputSize: 13,
  outputSize: 1,
};

const trainingOptions = {
  iterations: 200000,
  log: (details) => console.log(details),
  learningRate: 0.005,
};

// CREATING THE TRAINING BACKPROP NEURAL NETWORK WITH THE CROSS VALIDATION
const crossValidate = new brain.CrossValidate(() => new brain.NeuralNetwork(netOptions));

// TRAINING THE NEURAL NETWORK ON THE DATASET AND USING K FOLDS CROSS VALIDATION TECHNIQUE
const stats = crossValidate.train(trainingData, trainingOptions, );

// CONSOLE LOGGING THE RESULTS OF TRAINING TO EXAMINE THE ERROR AND PERFORMANCE OF THE NN
console.log(stats);

// CONVERTING THE CROSS VALIDATION NETWORK TO A STANDARD NEURAL NETWORK:
const net = crossValidate.toNeuralNetwork();

// SAVING THE TRAINED NEURAL NETWORK INTO A JSON FILE FOR USE IN THE STRATEGYM APPLICATION [NOTE - THIS SAVES IN THE STRATEGYM ROOT
  const fs = require("fs"); //fs NPM FILESYSTEM LIBRARY ALLOWS TO READ AND WRITE LOCAL DATA
  const trainedNN = net.toJSON();
  fs.writeFileSync("maleTrainedNN.json", JSON.stringify(trainedNN), 'utf-8');
```

Figure 7.43: Implementing the Training Config of the Mens Bodyfat Neural Network and Outputting the Trained Neural Network as a .JSON File

```
// CONFIGURATION FOR THE NERUAL NETWORK AND TRAINING CONFIGURATION:
const netOptions = {
  hiddenLayers: [9],
  activation: 'sigmoid',
  inputSize: 13,
  outputSize: 1,
};

const trainingOptions = {
  iterations: 150000,
  log: (details) => console.log(details),
  learningRate: 0.01,
};

// CREATING THE TRAINING BACKPROP NEURAL NETWORK WITH THE CROSS VALIDATION
const crossValidate = new brain.CrossValidate(() => new brain.NeuralNetwork(netOptions));


// TRAINING THE NEURAL NETWORK ON THE DATASET AND USING CROSS VALIDATION TECHNIQUE
const stats = crossValidate.train(trainingData, trainingOptions, );

// CONSOLE LOGGING THE RESULTS OF TRAINING TO EXAMINE THE ERROR AND PERFORMANCE OF THE NN
console.log(stats);

// CONVERTING THE CROSS VALIDATION NETWORK TO A STANDARD NEURAL NETWORK:
const net = crossValidate.toNeuralNetwork();

// SAVING THE TRAINED NEURAL NETWORK INTO A JSON FILE FOR USE IN THE STRATEGYM APPLICATION: [NOTE - THIS SAVES IN THE STRATEGYM ROOT
  const fs = require("fs"); //fs NPM FILESYSTEM LIBRARY ALLOWS TO READ AND WRITE LOCAL DATA
  const trainedNN = net.toJSON();
  fs.writeFileSync("femaleTrainedNN.json", JSON.stringify(trainedNN), 'utf-8');
```

Figure 7.44: Implementing the Training Config of the Women's Bodyfat Neural Network and Outputting the Trained Neural Network as a .JSON File

7.3.3.3.4 Implementation of the Trained and Saved Neural Networks in The Application

In order to implement the trained neural networks into the application, firstly standardisation functions have been constructed for each of the users inputs in the bodyfat percentage form, which follow the corresponding minimum and maximum values of each column in the dataset, as well as the de-standardised function for retrieving the raw bodyfat percentage output value from the neural network.

```
function standardiseInputAge(value) {
  return (value - 22) / (81 - 22);
}

function standardiseInputHeight(value) {
  return (value - 74.93) / (197.485 - 74.93);
}
function standardiseInputWeight(value) {
  return (value - 52.7507) / (164.7221 - 53.7507);
}
function standardiseInputNeck(value) {
  return (value - 31.1) / (51.2 - 31.1);
}

function standardiseInputChest(value) {
  return (value - 79.3) / (136.2 - 79.3);
}

function standardiseInputChest(value) {
  return (value - 79.3) / (136.2 - 79.3);
}

function standardisedInputAbs(value) {
  return (value - 69.4) / (148.1 - 69.4);
}

function standardisedInputHips(value) {
  return (value - 85) / (147.7 - 85);
}
function standardisedInputThigh(value) {
  return (value - 47.2) / (87.3 - 47.2);
}

function standardisedInputKnee(value) {
  return (value - 33) / (49.1 - 33);
}

function standardisedInputAnkle(value) {
  return (value - 19.1) / (33.9 - 19.1);
}

function StandardisedInputBicep(value) {
  return (value - 24.8) / (45 - 24.8);
}

function StandardisedInputForearm(value) {
  return (value - 21) / (34.9 - 21);
}

function StandardisedInputWrist(value) {
  return (value - 15.8) / (21.4 - 15.8);
}

function destandardisedBFOutput(value) {
  return value * (47.5 - 0) + 0;
}
```

Figure 7.45: Standardisation and De-Standardisation Functions (Men's Example)

Next, a function had been constructed for taking the inputs from the form and running it through the trained neural network.

```
function MaleUsersBodyfatPercentage(
  userAge,
  userHeight,
  userWeight,
  userNeck,
  userChest,
  userAbs,
  userHips,
  userThigh,
  userKnee,
  userAnkle,
  userBicep,
  userForearm,
  userWrist,
) {
  // LOADING THE TRAINED MENS BF% NEURAL NETWORK
  const brain = require('brain.js');
  //CONFIGURING THE NEURAL NETWORK'S ACTIVAITION
  var mensBfNN = new brain.NeuralNetwork({
    activation: 'sigmoid', // activation function
    hiddenLayers: [9], //1 hidden layer 2/3rds of the size of the input
  });
  const mensTrainedNN = require('./maleTrainedNN.json');
  mensBfNN.fromJSON(mensTrainedNN);

  var mBodyfatOutput = destandardisedBFOutput(
    mensBfNN.run([
      standardiseInputAge(userAge),
      standardiseInputHeight(userHeight),
      standardiseInputWeight(userWeight),
      standardiseInputNeck(userNeck),
      standardiseInputChest(userChest),
      standardisedInputAbs(userAbs),
      standardisedInputHips(userHips),
      standardisedInputThigh(userThigh),
      standardisedInputKnee(userKnee),
      standardisedInputAnkle(userAnkle),
      StandardisedInputBicep(userBicep),
      StandardisedInputForearm(userForearm),
      StandardisedInputWrist(userWrist),
    ]),
  );
  return mBodyfatOutput;
}
```

Figure 7.46: Running the Trained Neural Network on User Inputs Function (Men's Example)

Finally, the on-press method for the submit button on the bodyfat percentage form had been constructed, which runs the men's bodyfat percentage trained neural network if the user is male, and the women's bodyfat percentage trained neural network if the user is female.

```
  if (validMeasurementsForm()) {
    //RUN THE mensNN ON MALE USER'S DATA AND THEN UPDATE MEASUREMENTS
    if (userMeasurements.gender.trim() == 'Male') {
      userMeasurements.bodyfatPercentage =
        MaleUsersBodyfatPercentage(
          userMeasurements.age.trim(),
          userMeasurements.height.trim(),
          userMeasurements.weight.trim(),
          userMeasurements.neckCircum.trim(),
          userMeasurements.chestCircum.trim(),
          userMeasurements.abdomenCircum.trim(),
          userMeasurements.hipCircum.trim(),
          userMeasurements.thighCircum.trim(),
          userMeasurements.kneeCircum.trim(),
          userMeasurements.ankleCircum.trim(),
          userMeasurements.bicepCircum.trim(),
          userMeasurements.forearmCircum.trim(),
          userMeasurements.wristCircum.trim(),
        );
    }

    //RUN THE womensNN ON FEMALE USER'S DATA AND THEN UPDATE MEASUREMENTS....
    if (userMeasurements.gender.trim() == 'Female') {
      userMeasurements.bodyfatPercentage =
        FemaleUsersBodyfatPercentage(
          userMeasurements.age.trim(),
          userMeasurements.height.trim(),
          userMeasurements.weight.trim(),
          userMeasurements.neckCircum.trim(),
          userMeasurements.chestCircum.trim(),
          userMeasurements.hipCircum.trim(),
          userMeasurements.thighCircum.trim(),
          userMeasurements.kneeCircum.trim(),
          userMeasurements.ankleCircum.trim(),
          userMeasurements.bicepCircum.trim(),
          userMeasurements.forearmCircum.trim(),
          userMeasurements.wristCircum.trim(),
        );
    }
    updateUserMeasurements(
      userMeasurements.bodyfatPercentage,
    );
  }
}}
```

Figure 7.47: On Press Method For Submit Form Button To Execute Running User's Input on Trained Neural Network

7.3.3.4 Testing & Resolutions

The full testing table for this increment is available in Appendix 9.1.3.

The testing phase of this increment consisted of testing the following:

- The error messages displayed for the input forms on pressing submit for invalid data input
- The alert message displayed when the user tries to update their email or password with their wrong current password input
- Testing if the user can log in with their new authentication details when they submit a valid change email or change password form
- Testing whether the updates have been made to the user's Firestore document when they submit a form
- Testing if the trained neural network runs on the user's data input and outputs a de-standardised bodyfat percentage prediction that is updated in their Firestore document
- Testing the statistics tab of the home screen to see if the it dynamically displays all of the user's health and fitness metrics from their Firestore document.

All tests in this testing phase had passed, with no changes required to the core functionality of each component. However, one issue that had been noticed was that some forms across the profile section require the same data inputs, which therefore would require updating all calculations requiring that input. For example, the dietary form contains the inputs age, height, and weight, all of which should change the users TDEE too if they were to update these values in the dietary form. Therefore, for forms where this is the case, all necessary updates to calculations impacted by a user changing a value on one of the forms has been implemented. For example,, FIGURE XXX shows the dietary goal form now also updating the user's BMR and TDEE.

```
onPress={() => {
   if (
     validMetricsForm(
       userMetrics.age.trim(),
       userMetrics.height.trim(),
       userMetrics.weight.trim(),
       userMetrics.activityLevel,
       userMetrics.gender,
       userMetrics.idealWeight,
       userMetrics.idealWeeks,
     ) ) {
```

```
const userCurrentWeight = userMetrics.weight;
const userIdealWeight = userMetrics.idealWeight;
const userIdealWeeks = userMetrics.idealWeeks;
if (userMetrics.gender.trim() === 'Male') {
  userMetrics.BMR =
    66.5 +
    13.75 * userMetrics.weight.trim() +
    5.003 * userMetrics.height.trim() -
    6.75 * userMetrics.age.trim();

  userMetrics.TDEE =
    userMetrics.BMR * userMetrics.activityLevel;
}
if (userMetrics.gender.trim() === 'Female') {
  userMetrics.BMR =
    655.1 +
    9.563 * userMetrics.weight.trim() +
    1.85 * userMetrics.height.trim() -
    4.676 * userMetrics.age.trim();

  userMetrics.TDEE =
    userMetrics.BMR * userMetrics.activityLevel;
}
if (userIdealWeight > userCurrentWeight) {
  const weightToGain = userIdealWeight - userCurrentWeight;
  const caloriesToGainTotal = 7700 * weightToGain;
  const caloriesToGainWeekly =
    caloriesToGainTotal / userIdealWeeks;
  const caloriesSurplusDaily = caloriesToGainWeekly / 7;
  userMetrics.goalCalories = Math.round(
    TDEE + caloriesSurplusDaily,
  );
  userMetrics.goalProtein = Math.round(
    (userMetrics.goalCalories * 0.25) / 4,
  );
  userMetrics.goalFats = Math.round(
    (userMetrics.goalCalories * 0.2) / 9,
  );
  userMetrics.goalCarbs = Math.round(
    (userMetrics.goalCalories * 0.55) / 4,
  );
}
if (userIdealWeight < userCurrentWeight) {
  const weightToLose = userCurrentWeight - userIdealWeight;
  const caloriesToLoseTotal = 7700 * weightToLose;
```

```
    const caloriesToLoseWeekly =
      caloriesToLoseTotal / userIdealWeeks;
    const caloriesDeficitDaily = caloriesToLoseWeekly / 7;
    userMetrics.goalCalories = Math.round(
      TDEE - caloriesDeficitDaily,
    );
    //Macros Split for losing weight = 30% protein, 55% carbs, 15% fats
    //Using the Harris-Benedict Equation:
    userMetrics.goalProtein = Math.round(
      (userMetrics.goalCalories * 0.3) / 4,
    );
    userMetrics.goalFats = Math.round(
      (userMetrics.goalCalories * 0.15) / 9,
    );
    userMetrics.goalCarbs = Math.round(
      (userMetrics.goalCalories * 0.55) / 4,
    );
  }
  if (userIdealWeight == userCurrentWeight) {
    // if user wants to maintain
    userMetrics.goalCalories = Math.round(TDEE);
    // Macros Split for Maintenance = 25% protein, 60% carbs, 15% fats
    userMetrics.goalProtein = Math.round(
      (userMetrics.goalCalories * 0.25) / 4,
    );
    userMetrics.goalCarbs = Math.round(
      (userMetrics.goalCalories * 0.6) / 4,
    );
    userMetrics.goalFats = Math.round(
      (userMetrics.goalCalories * 0.15) / 9,
    );
  }
  updateUserDietaryGoal(
    userMetrics.age.trim(),
    userMetrics.height.trim(),
    userMetrics.weight.trim(),
    userMetrics.gender.trim(),
    userMetrics.activityLevel,
    userMetrics.BMR,
    userMetrics.TDEE,
    userMetrics.idealWeight,
    userMetrics.idealWeeks,
    userMetrics.goalCalories,
    userMetrics.goalCarbs,
    userMetrics.goalProtein,
```

```
            userMetrics.goalFats,
          );
          navigation.replace("Profile Home")
        }
      }}
```

Figure 7.7.48: Dietary Goal Also Re-Calculating TDEE

7.3.3.5 Results Display

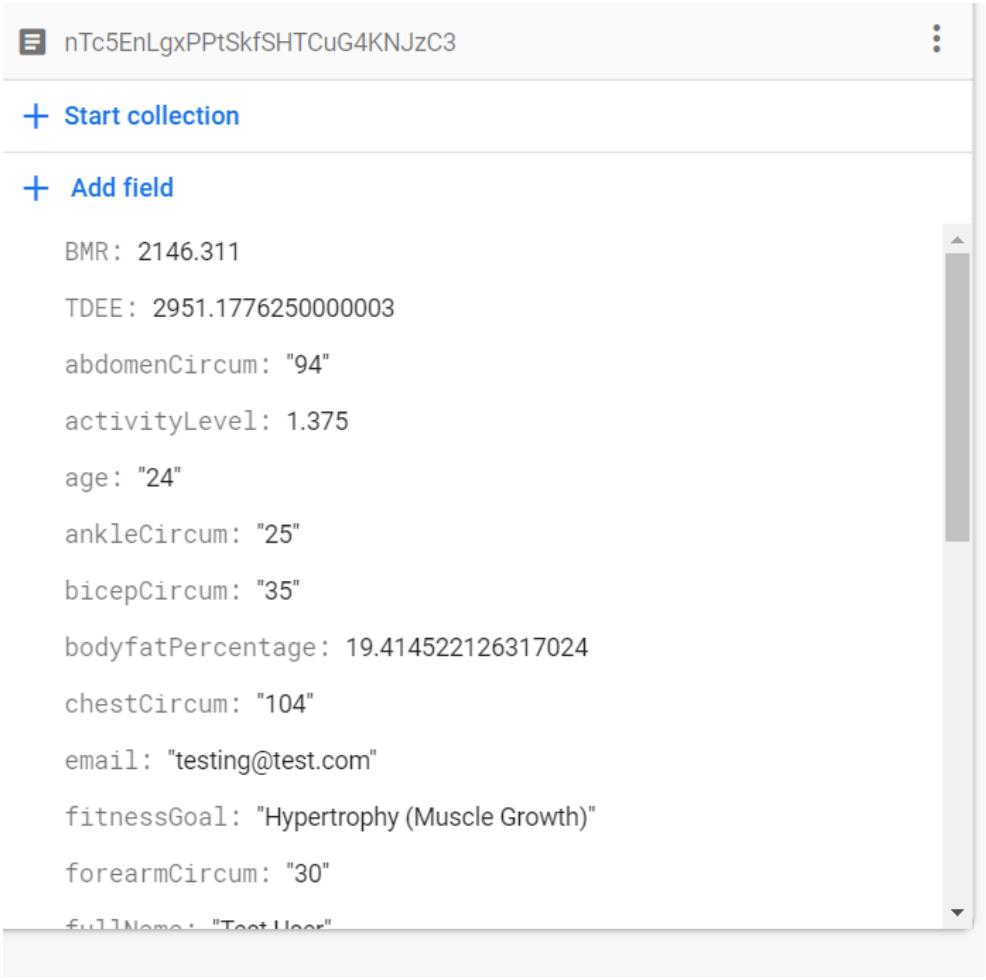


Figure 7.49: Firestore Document From User Filled Forms

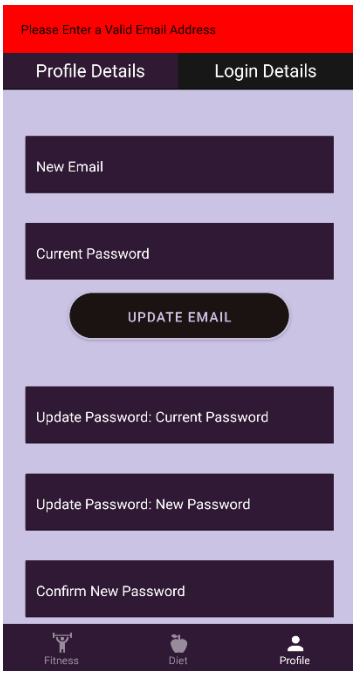


Figure 7.50: Invalid Form Notification Example Screenshot

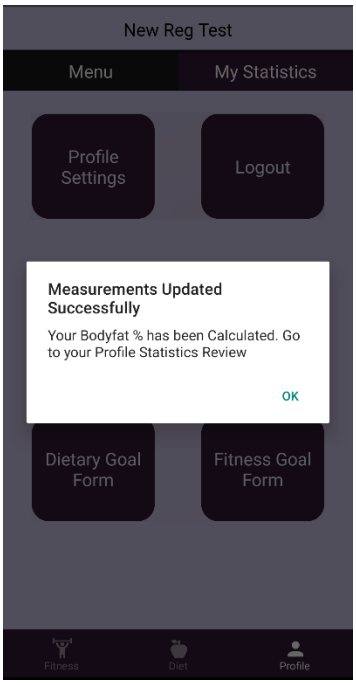


Figure 7.51: Bodyfat Percentage Prediction Success Alert

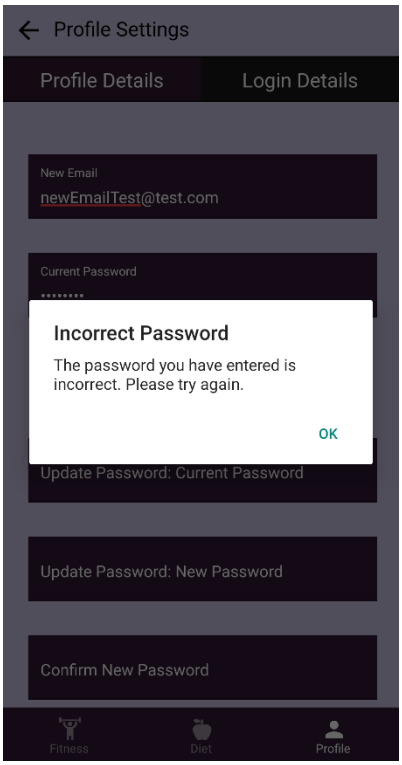


Figure 7.52: Incorrect Password Alert Example Screenshot

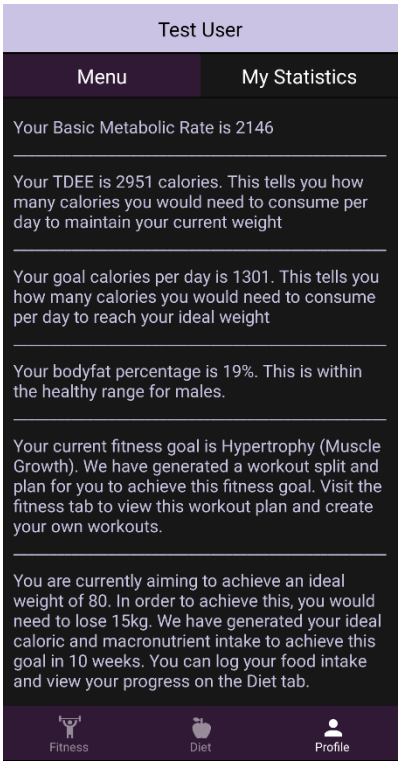


Figure 7.53: Statistics Tab with Breakdown Data of Users Metrics Screenshot Example

7.3.4 Increment 4: Diet Side Implementation

The diet side functionality involves all functionality related to the nutritional intake and dietary progress of the user. This includes:

- Access to the in-app food database
- Creating Food Items
- Displaying and Deleting User Created Food Items
- Adding Foods to Today's Intake
- Viewing Today's Intake (Pie Chart) and Viewing Intake Over Time (Line Graph)

7.3.4.1 Access to the In-App Food Database:

The in-app food database has already been created as a JSON file in the project directory. Therefore, to implement this feature, the JSON file would simply need to be imported in the Add Foods screen component, and rendered in the react-native <FlatList> component, which automatically lists the objects in the JSON file as a scrollable vertical list.

```
<FlatList
  data={nutritionalData}
  keyExtractor={nutritionalData => nutritionalData.Key}
  contentContainerStyle={{alignContent: 'center'}}
  ItemSeparatorComponent={() => (
    <View
      style={{
        height: 1,
        width: '100%',
        backgroundColor: '#CBC3E3',
        alignContent: 'center',
        alignItems: 'center',
      }}
    />
  )}
  renderItem={({item}) => (
   ..................
```

Figure 7.54: Code Snippet: Rendered FlatList for In-App Food Database

The <FlatList> component also enables the ability to access and display properties from objects in the JSON file by simply calling the property name. In this case, the calories, protein, carbohydrates and fats have all

been listed with the name of the food item in the list so that the user can view the nutritional information of a food item and then determine whether they would like to add it to today's intake or not.

```
<Text
  style={{
    fontSize: 13,
    color: '#CBC3E3',
    marginBottom: 10,
    marginTop: 10,
    marginLeft: 10,
    marginRight: 10,
  }}>
  Per 100g: Calories: {item.Calories} | Protein: {item.Protein}g
  | Fats: {item.Fats}g | Carbs: {item.Carbs}g
</Text>
```

Figure 7.55: Code Snippet: Food Calories and Macronutrients for Each Item in FlatList

An amount (in grams) text input has also been added to each rendered item in the <FlatList> so that the user can set the amount of a food item they have consumed before adding it to today's daily intake.

```
<TextInput
  placeholder="Amount (g)"
  onChangeText={gramsInput => setUserGramsInput(gramsInput)}
  defaultValue={gramsInput}
  backgroundColor="#301934"
  style={{
    width: '70%',
    height: 40,
    marginBottom: 5,
    fontSize: 15,
    mode: 'contained',
  }}
  theme={{
    colors: {
      text: 'white',
      placeholder: 'white',
      primary: 'white',
    },
  }}
/>
```

Figure 7.56: Code Snippet: Grams TextInput Component for Each Item in FlatList

7.3.4.2 Creating Food Items:

The SQLite local database storage is used to store the user's own created foods. In order to implement this feature, the react-native-sqlite-storage library has been used. The data is stored in a database called

userFoods.db, in a table called demo_food. The function to create this database has been implemented inside of react's useEffect hook so that when the create foods screen renders, it automatically searches to see if the table exists inside of the userFoods.db database, and if it doesn't then it creates the table. In order to test whether this useEffect is working in the testing phase of this increment, a console.log statement which outputs the result of the useEffect has also been implemented into the useEffect hook.

```
var db = openDatabase({name: 'userFoods.db'});

  useEffect(() => {
    db.transaction(function (txn) {
      txn.executeSql(
        "SELECT name FROM sqlite_master WHERE type='table' AND name='demo_food'",
        [],
        function (tx, res) {
          console.log('item:', res.rows.length);
          if (res.rows.length == 0) {
            txn.executeSql('DROP TABLE IF EXISTS demo_food', []);
            txn.executeSql(
              'CREATE TABLE IF NOT EXISTS demo_food(food_id INTEGER PRIMARY KEY AUTOINCREMENT, food_name VARCHAR(30), food_grams INT(15), food_calories INT(1
              [],
            );
          }
        },
      );
    });
  }, []);
```

Figure 7.57: Code Snippet: useEffect for Creating Table to Store User-Created Foods

Following this, an insert data function has been constructed which takes the inputs from the created food form as parameters and executes the SQL query to insert this data as a row into the table. In order to test and ensure that this data has been inserted successfully as well as inform the user that they have successfully created a food item, a react alert component has also been implemented in this function.

Furthermore, the SQLite query to insert the data into the table generates an auto-increment primary key column which is treated as a unique identifier for the rows in the table. This is so that SQL search queries that are executed can be efficiently done so using the identifier, and also so that items in the table that the user would like to delete can be addressed and located by their primary key. This auto-incremental primary key column is planned to be implemented for all SQL insert functions.

```
const InsertData = (foodName, grams, calories, protein, fats, carbs) => {
  db.transaction(function (tx) {
    tx.executeSql(
      'INSERT INTO demo_food (food_name, food_grams, food_calories, food_protein, food_fats, food_carbs) VALUES (?,?,?,?,?,?)',
      [foodName, grams, calories, protein, fats, carbs],
      (tx, results) => {
        console.log('Results', results.rowsAffected);
        if (results.rowsAffected > 0) {
          Alert.alert('Data Inserted Successfully....');
        } else Alert.alert('failed....');
      },
    );
  });
};
```

Figure 7.58: Code Snippet: Function for Inserting Created Food Item Into Database Table

This insert function has then been added to the on press method of the submit button rendered on the create food screen.

```
onPress={() => {
    if (validCreatedFoodForm()) {
      createdFood.calories = convertCalsTo100Grams();
      createdFood.protein = convertProteinTo100Grams();
      createdFood.carbs = convertCarbsTo100Grams();
      createdFood.fats = convertFatsTo100Grams();
      createdFood.grams = setGramsto100();

      InsertData(
        createdFood.foodName,
        String(createdFood.grams),
        String(createdFood.calories),
        String(createdFood.protein),
        String(createdFood.fats),
        String(createdFood.carbs),
      );
      navigation.replace('Saved Foods');
      //updateUserSavedFoodsDatabase();
    }
  }}
```

Figure 7.59: On-Press Method For Inserting Created Food Item On Pressing Green Basket Icon

7.3.4.3 Displaying and Deleting User Created Food Items:

The user's created and saved food items are displayed on the saved foods screen of the application. In order to display the food items here, the database userFoods.db has been opened in this screen's component code and a useEffect react hook has been implemented that executes the SQL query to get the demo_food table from the database and append it to an empty array called setFlatListItems every time the saved foods screen renders.

```
useEffect(() => {
  db.transaction(tx => {
    tx.executeSql(
      'SELECT * FROM demo_food',
      [],
      (tx, results) => {
        var temp = [];
        for (let i = 0; i < results.rows.length; ++i)
          temp.push(results.rows.item(i));
        setFlatListItems(temp);
      },
    );
  });
}, []);
```

Figure 7.60: UseEffect for Getting User Created Foods From SQL Table

This array is then displayed on the screen using the react-native <FlatList> component, in the same manner that the in-app foods database is displayed.

```
<FlatList
  data={flatListItems}
  keyExtractor={(item, index) => index.toString()}
  contentContainerStyle={{alignContent: 'center'}}
  //
  ItemSeparatorComponent={() => (
    <View
      style={{
        height: 1,
        width: '100%',
        backgroundColor: '#CBC3E3',
        alignContent: 'center',
        alignItems: 'center',
      }}
    />
  )}
  renderItem={({item}) => ( ..................
```

Figure 7.61: Code Snippet: Rendered FlatList Displaying User Created Foods

In order to handle the user deleting a food item they have created, a delete function has been constructed which takes the food ID from the FlatList as a parameter, and executes the SQL query to delete the food item that matches the ID in the database table. A react-native alert component has also been implemented into this function to alert the user that they have successfully deleted the food item from their database, and for

testing purposes a console.log statement has been implemented to debug any cases that may arise where the function fails.

```
const deleteData = foodID => {
  db.transaction(function (tx) {
    tx.executeSql(
      'DELETE FROM demo_food where food_id=?',
      [foodID],
      (tx, results) => {
        console.log('Results', results.rowsAffected);
        if (results.rowsAffected > 0) {
          alert('Data Deleted Successfully....');
        } else console.log('Failed...');
      },
    );
  });
};
```

Figure 7.62: Code Snippet: Delete Function For Deleting Created Food Item From SQL Table

The delete function has then been called in the on-press method for the bin icon which is rendered for every item on the list. In order to confirm the deletion with the user, the on press initially renders an alert message, which informs the user about the food item they are about to delete and if they are sure to delete the food item. The delete function is placed on the yes option of this alert message.

```
onPress={() => {
  Alert.alert('Delete' + item.food_name, '?', [
    {
      text: 'YES',
      onPress: () => {
        //DELETE ITEM FROM SQL FUNCTION GOES HERE
        deleteData(item.food_id);
        navigation.replace('Add Foods');
      },
    },
  ]);
}}
```

Figure 7.63: Rubbish Icon On-Press Method with Alert and Delete Function Braced Inside

7.3.4.4 Adding Foods to Today's Intake

In the diet section of the application the user can add foods to today's intake either on the saved foods screen or on the add foods screen. The implementation of this feature is identical among both screens. The user's food intake history is saved in an SQLite database called foodsIntakeHistory.db, and the name of the table that the food item is added to is named after today's date.

In order to implement this feature, firstly the in-built JavaScript ES6 object "Date" has been utilized in order to retrieve today's date, month and year. These are then concatenated into one string, formatted so that it is eligible as an SQL table name.

```
var date = new Date().getDate();
var month = new Date().getMonth() + 1;
var year = new Date().getFullYear();
const tableNameDate = '_' + date + '_' + month + '_' + year;
```

Figure 7.64: Code Snippet: Formatting the Date for SQLite Table Name

Following this, a UseEffect hook has been constructed which searches the database to see if a table named after the concatenated string of today's date exists and creates the table if it does not.

The insert data function takes the food data from the list item in the rendered FlatList component as parameters, and then executes an SQL query to insert the food data into the table named after today's date.

```
const InsertData = (foodName, grams, calories, protein, fats, carbs) => {
  db.transaction(function (tx) {
    tx.executeSql(
      'INSERT INTO ' +
        tableNameDate +
        ' (food_name, food_grams, food_calories, food_protein, food_fats, food_carbs) VALUES (?,?,?,?,?,?)',
      [foodName, grams, calories, protein, fats, carbs],
      (tx, results) => {
        console.log('Results', results.rowsAffected);
        if (results.rowsAffected > 0) {
          alert(foodName + "Has Been Added To Today's Intake");
        } else alert('failed....');
      },
    );
  });
};
```

Figure 7.65: Code Snippet: Insert Data Function for Adding Food Item to Today's Intake SQL Table

This insert data function is then called in the on-press method of the green basket icon displayed on the screen which represents adding food to today's intake. Prior to the insert function being executed, the user is prompted by a react alert component which informs the user of what they are about to add to their daily intake with the option to cancel the execution.

```
onPress={() => {
  if (validateGramsInput(gramsInput)) {
    const updatedCalories = Math.round(
      (parseFloat(item.Calories) / 100) * gramsInput,
    ); //Over 100 because default grams is 100
    const updatedProtein = Math.round(
      (parseFloat(item.Protein) / 100) * gramsInput,
    );
    const updatedCarbs = Math.round(
      (parseFloat(item.Carbs) / 100) * gramsInput,
    );
    const updatedFats = Math.round(
      (parseFloat(item.Fats) / 100) * gramsInput,
    );
    const foodName = item.FoodName;
    console.log(foodName);

    Alert.alert(
      'Would You Like To Add ' +
        item.FoodName +
        "To Today's Intake?",
      'Grams: ' +
        gramsInput +
        'g' +
        ' Calories: ' +
        updatedCalories +
        'g' +
        ' Protein: ' +
        updatedProtein +
        'g' +
        ' Fats: ' +
        updatedFats +
        'g' +
        ' Carbs: ' +
        updatedCarbs +
        'g',
      [
        {
          text: 'YES',
          onPress: () => {
            InsertData(
              item.FoodName,
              String(gramsInput),
              String(updatedCalories),
              String(updatedProtein),
              String(updatedFats),
              String(updatedCarbs),
            );
          },
        },
      ],
    );
  }
}}
```

Figure 7.66: Code Snippet: On-Press Method For Inserting Food Item Into Today's Intake With Alert Confirmation

7.3.4.5 Viewing Today's Intake (Pie Chart) and Viewing Intake Over Time (Line Graph)

The user is able to view the SQLite database table containing their food intake for today's date on the diet home screen, once again rendered in react-native's <FlatList> component.

Firstly in the code for the diet home screen component, the date, month, and year are again retrieved from the JavaScript in-built Date object. These are then concatenated into a string in the same way the SQL table name is, for use when searching for the table logging the food intake for today's date.

A useEffect hook has been constructed which executes the SQL query to search for the table containing the food items of today's date, and once found the table contents are then appended to an empty array.

```
useEffect(() => {
  db.transaction(function (txn) {
    txn.executeSql(
      "SELECT name FROM sqlite_master WHERE type='table' AND name=" +
        "'" +
        tableNameDate +
        "'",
      [],
      function (tx, res) {
        console.log('item:', res.rows.length);
        if (res.rows.length == 0) {
          txn.executeSql('DROP TABLE IF EXISTS ' + tableNameDate, []);
          txn.executeSql(
            'CREATE TABLE IF NOT EXISTS ' +
              tableNameDate +
              '(food_id INTEGER PRIMARY KEY AUTOINCREMENT, food_name VARCHAR(30), food_grams INT(15), food_calories INT(1
            [],
          );
        }
      },
    );
  });
}, []);
```

Figure 7.67: Code Snippet: useEffect for Getting Today's Intake SQL Table

The array is then displayed in the rendered FlatList component on the screen for the user to see. Similarly to how the delete function has been implemented for deleting the user's created foods, a delete function has also been constructed which executes an SQL query to delete a food item from today's intake table on the user pressing the bin icon in each row of the rendered FlatList component.

```
const deleteData = foodID => {
  db.transaction(function (tx) {
    tx.executeSql(
      'DELETE FROM ' + tableNameDate + ' where food_id=?',
      [foodID],
      (tx, results) => {
        console.log('Results', results.rowsAffected);
        if (results.rowsAffected > 0) {
          console.log('Food Item Deleted');
        } else console.log('Failed');
      },
    );
  });
};
```

Figure 7.68: Code Snippet: Delete Function for Deleting Food Item From Today's Intake

7.3.4.6 Tracking Today's Food Intake Against Goal Calories and Macronutrients

With regards to implementing the tracking of today's food intake, firstly each column of the table storing today's intake is selected and stored in it's own array within a useEffect hook. For example, an SQL query is executed to get the calories column from the table and append it to an array called calories.

```
useEffect(() => {
  db.transaction(tx => {
    tx.executeSql(
      'SELECT food_calories FROM ' + tableNameDate,
      [],
      (tx, results) => {
        var temp = [];
        for (let i = 0; i < results.rows.length; ++i)
          temp.push(results.rows.item(i));
        setCalories(temp);
      },
    );
  });
}, []);
```

Figure 7.69: Code Snippet: useEffect to Get Calories Column Example

Following this, a function has been constructed which adds up all the values within the array. The JavaScript ES6 script includes a .reduce operator on arrays in order to achieve this, which is constructed as follows (TutorialRepublic, 2022):

```
var sum = array.reduce(function(a, b){ return a + b; }, 0);
```

Figure 7.70: Function To Add Items In Array

However, due to this function not working as intended, another method to add all the items in the array has been implemented. The function is as follows:

1. Convert the array to a string using JSON.stringify operator in JavaScript
2. Add all the numbers in the string and treat numbers that are next to each other as one number rather than separate numbers (for example, treat 22 as twenty-two rather than two two's).

The algorithm to execute this process had been found on stack overflow as the following:

```
// REFERENCE: https://stackoverflow.com/questions/53897373/js-how-to-got-the-sum-of-numb
function sumNumbers(string) {
  let pos = 1;
  let numArray = [];
  let numString = null;

  for (let num of string) {
    let isParsed = isNaN(parseInt(num));
    if (!numString && !isParsed && pos === string.length) {
      numArray.push(num);
    } else if (!numString && !isParsed && pos !== string.length) {
      numString = num;
    } else if (numString && !isParsed && pos === string.length) {
      numString += num;
      numArray.push(numString);
    } else if (numString && isParsed && pos === string.length) {
      numArray.push(numString);
    } else if (numString && !isParsed) {
      numString += num;
    } else if (numString && isParsed && pos !== string.length) {
      numArray.push(numString);
      numString = null;
    } else if (!numString && isParsed && pos === string.length) {
      numString += num;
      numArray.push(numString);
    }
    pos++;
  }
  console.log('numAray:', numArray);
  let result = null;

  for (let num of numArray) {
    let value = parseInt(num);
    result += value;
  }

  return NaNCheck(result);
}

```

Figure 7.71: Function to Add Numbers in a String

This function has been executed on all the column arrays of the table to get the daily total amounts of: calories, protein, fats and carbohydrates.

Next, on the todaysTotals screen component, a function has been constructed to get the document ID of the user, which contains their goal calories and goal macronutrients. Firebase's .collection.doc.get function is used to get the Firestore document data, followed by setting the goal calories and macronutrient variables to the data read from Firebase.

```
const getUserFirestoreMetrics = async () => {
  await firestore()
    .collection('users')
    .doc(auth().currentUser.uid)
    .get()
    .then(documentSnapshot => {
      if (documentSnapshot.exists) {
        console.log('User Data', documentSnapshot.data());
        setUserMetrics(documentSnapshot.data());
      }
    });
};
```

Figure 7.72: Firebase Function to Get Goal Calories and Macronutrients from User's Firestore

In order to calculate the number of calories and macronutrients remaining, the daily caloric/macronutrient goals retrieved from Firebase are subtracted by the total calories/macronutrients consumed. Then, arrays have been formed which hold the total calories/macronutrients consumed and the calories/macronutrients remaining. These arrays have been formed so that they can be displayed on a pie chart on the screen.

```
const caloriesDailyGoal = userMetrics.goalCalories;
const caloriesConsumed = route.params.caloriesTotalPass;
const caloriesRemaining = caloriesDailyGoal - caloriesConsumed;
const caloriesSeriesChart = [caloriesConsumed, caloriesRemaining];
```

Figure 7.73: Code Snippet: Calculation for Calories Remaining And Calories Array For Pie Chart

The react-native-pie-chart library is imported on the todays totals screen, which takes the calories/macronutrients arrays and displays the array as a pie chart on the screen.

```
<PieChart
  widthAndHeight={chartswidthAndHeight}
  series={caloriesSeriesChart}
  sliceColor={setPieChartColor(caloriesConsumed, caloriesDailyGoal)}
  doughnut={true}
/>
```

Figure 7.74: Code Snippet: Rendered Pie Chart Component Example

Finally, a function has been constructed which changes the colour of the text informing the user of how many calories/macronutrients they have remaining depending on its value. If the remaining amount is more than 0, then the text is green; if it is between 0 and -20, then it is amber, and if it is less than -20, then it is red. This gives the user an indication of whether they are on track or not using a traffic light system.

```
function setTextColor(remainingAmount) {
  if (remainingAmount >= 0){
    return '#013220';
  }
  if((remainingAmount < 0) && (remainingAmount >= -20)) {
    return '#d53600';
  }
  if(remainingAmount < -20) {
    return '#8b0000';
  }

}
```

Figure 7.75: Code Snippet: Conditional Set Text Colour Function

This entire process has been repeated for the tracking of the user's daily food intake history over time as well with the SQL table storing the total calories and macronutrients that the user has consumed per day, however with tracking, the data is displayed on a line graph using the LineChart component from the react-native-chart-kit package and is displayed on the daily intake progress screen. Here, an array storing the dates the user has inputted their food intake is plotted against the total calories/macronutrients that they have consumed in that day.

```
// CALORIES VS DATE DATA TO BE SHOWN ON THE LINE GRAPH:
const caloriesData = {
  labels: formatDateHistoryArray(dateHistory),
  datasets: [
    {
      data: formatCalsHistoryArray(caloriesHistory),
    },
  ],
};

return (
    <LineChart
      data={caloriesData}
      width={Dimensions.get('window').width - 16}
      height={200}
      chartConfig={chartConfigCalories}
     />
)
```

Figure 7.76: Code Snippet: Example of Line Graph: Calories History Against Date Line Graph

7.3.4.7 Testing & Resolutions

See Appendix 9.1.4 for the full testing table for this increment.

The testing phase of this increment primarily consisted of testing the backend SQLite databases to ensure that they had been generated and contained the relevant data that the user would input from the front-end; then testing the front-end of the application to ensure that the data from the back-end SQLite database tables had been acquired and displayed appropriately. The front-end was tested using the Android emulator similarly to how all other front-end testing phases had occurred throughout the implementation of this application. In terms of testing the back-end of the application, the SQLite browser extension had been utilized which displays tables within a .db file so the developer can examine if the table has been generated and the correct data has been appended to the database. This method of testing the back-end was utilized for all SQLite query executions throughout this increment. For example, the figure below is a screenshot of the table that had been generated from the user creating food items in the application. All-in-all, there were no issues or failures that had risen when testing this increment.

SQL ▾ 1 / 1 1 - 8 of 8

food_id	food_name	food_grams	food_calories	food_protein	food_fats
1	Pasta	100	131	5	1
2	Cornflakes	100	378	7	0
3	Beef	100	250	35	10
4	Watermelon	100	30	0.6	0.2
5	Whey Protein Shake	100	379	70	6.1
6	Noodles	100	137	4.51	2.06
7	Whipped Cream	100	340	2.8	36
8	Vegan Protein Blend	100	377	87	0.8

Figure 7.77: Users Created Foods SQLite Table Opened in Visual Studio Code

7.3.4.8 Results Display

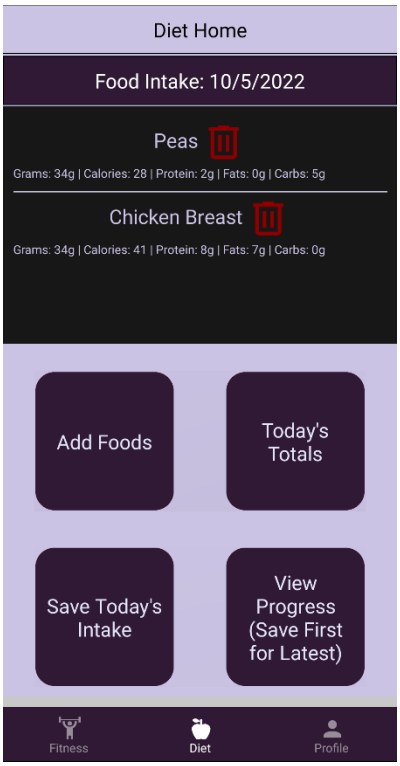


Figure 7.78: Screenshot: Diet Home Screen With Today's Food Items Listed

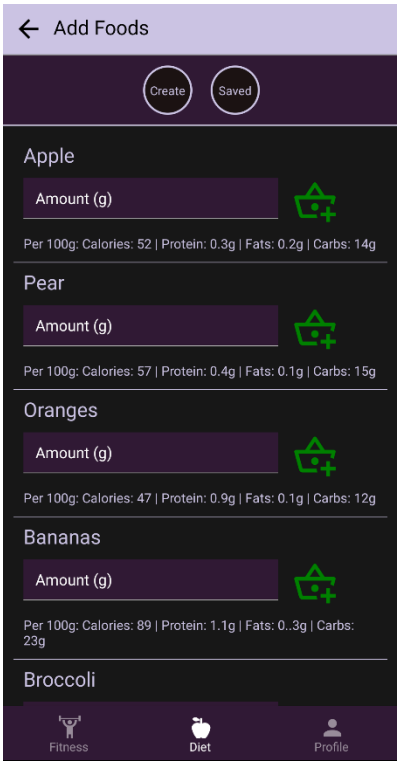


Figure 7.79: Screenshot: In-App Food Database Rendered

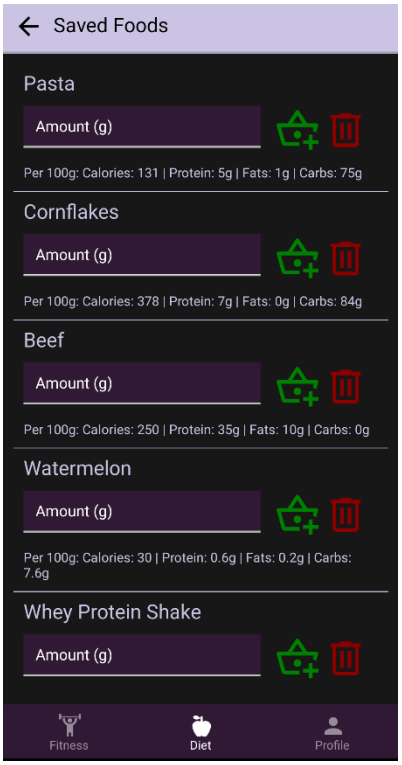


Figure 7.80: Screenshot: Users Created Food Items SQL Table Rendered

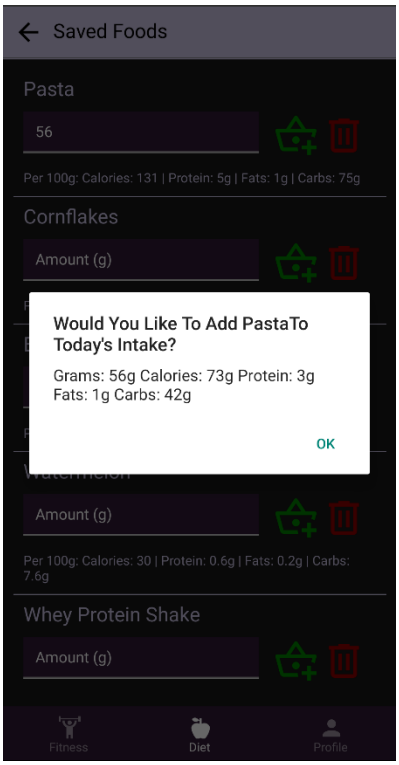


Figure 7.81: Screenshot: Confirmation Alert For Adding Food Item to Today's Intake

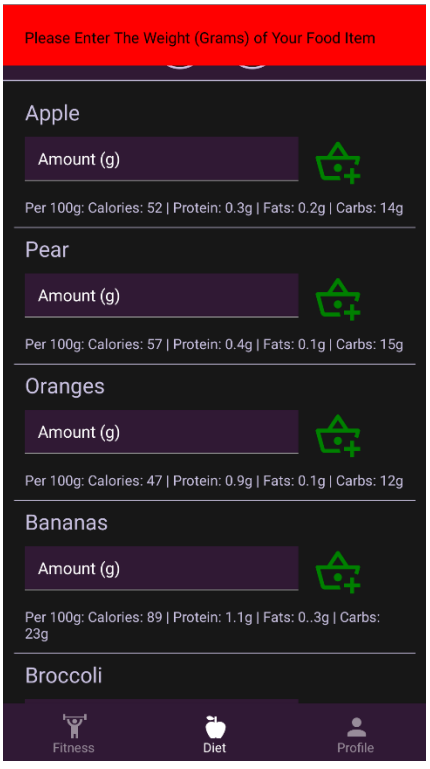


Figure 7.82: Screenshot: Error Message Example for Invalid Form Input

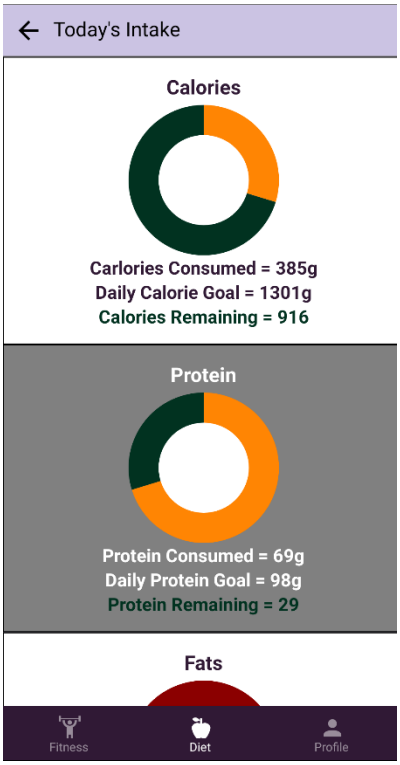


Figure 7.83: Screenshot: Today's Totals Pie Chart Display

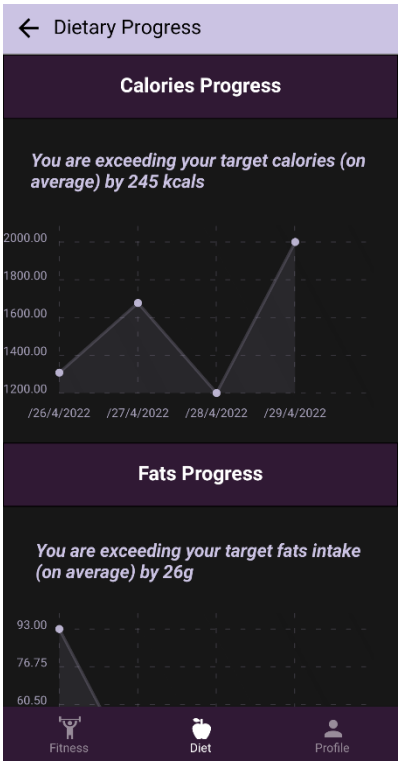


Figure 7.84: Screenshot: Daily Progress Over Time Graph Display

7.3.5 Increment 5: Fitness Side Implementation

The fitness side functionality involves all the functionality related to workouts and exercises. This includes:

- The generation of a workout plan based on the users set fitness goal and training frequency in the profile section
- Creating a workout
- Creating exercises
- Displaying and deleting user created workouts
- Performing a workout and saving the results of the performed workout
- Viewing the workout progress made over time
- Viewing the progress of selected exercises over time

The functions related to SQL database table manipulation such as creating a table, adding items, and deleting items all have the same structure as that of the functions in the diet section. Furthermore, the invalid data input errors and the confirmation alerts have also been implemented similarly to that of the diet section.

7.3.5.1 Generation of a Workout Plan

Similarly, to how the goal calories and macronutrients were acquired from the user's Firestore document in the diet section implementation, a function of the same structure has been constructed to acquire the user's fitness goal and their training frequency. Furthermore, similarly to how the in-app foods database was imported and listed on the add foods screen, the databases developed that hold the different workout plans for different fitness goals and training frequencies has also been imported in the fitness side of the application. Here, conditional statements have been constructed which take the training frequency and fitness goal set by the user as parameters and display the associated workout plan in a flat list on the generated plan tab of the begin workout screen.

```
const hypertrophyPlans = require('./workoutPlans/hypertrophyPlans');
const strengthPlans = require('./workoutPlans/strengthPlans');
const endurancePlans = require('./workoutPlans/endurancePlans');

function generatedUserPlan(trainingFreq, fitnessGoal) {

    if (trainingFreq == 1 && fitnessGoal == 'Hypertrophy (Muscle Growth)') {
     hypertrophyD1 = hypertrophyPlans.hypertrophyx1;
     return (
       <ScrollView style={{}}>
         <View>
           <Text>
             Day 1: Full Body
           </Text>
           <View>
             <Button
               onPress={() => {
                 navigation.replace('Perform Generated Workout', {
                   workoutName: 'Day 1: Full Body',
                   workoutArray: hypertrophyPlans.hypertrophyx1,
                 });
               }}>
               Perform
             </Button>
           </View>
           <View>
             <Text>
               Total Sets:{' '}
               {sumSetsInWorkoutArray(hypertrophyPlans.hypertrophyx1)} | Total
               Reps: {sumRepsInWorkoutArray(hypertrophyPlans.hypertrophyx1)} |
               Number of Exercises: {hypertrophyPlans.hypertrophyx1.length}
             </Text>
           </View>
         </View>
       </ScrollView>
     );
    }
    -------------
```

Figure 7.85: Code Snippet: Generated Workout Plan Rendered on Screen

Additionally, the total number of sets, reps and exercises are also calculated in the same way that the total calories and macronutrients are calculated in the diet section, and they are displayed for each workout in the generated workout plan list.

7.3.5.2 Creating a Workout

In order for the user to create a workout, they first set a workout name and then begin searching for exercises to include in their workout. The workout's content is to be saved in an SQLite database with the table name named after the workout name. In order to achieve this, the workout name that the user inputs is formatted so that it is eligible as an SQL table name. Throughout the entire process of the user creating a workout, the workout name in passed as a prop to all screens so that any exercise is added to that specific workout's SQLite table.

```
  function getWorkoutTableName(workoutName) {
    const workoutNameTable1 = workoutName.toLocaleLowerCase();
    const workoutNameTable2 = workoutNameTable1.replace(/\s/g, '_');
    return workoutNameTable2;
  }

return (
.............................
          onPress={() => {
            if (workoutNameValidation(createdWorkout.workoutName)) {
              const workoutTableName = getWorkoutTableName(
                createdWorkout.workoutName,
              );

              navigation.replace('Search Exercises', {
                workoutNamePassed: workoutTableName,
                actualWorkoutName: workoutName,
              });
            }
          }}


  )
```

Figure 7.86: Code Snippet: Formatting Workout Table Name and Passing as a Prop

The exercises search screen has been set up to display a list of touchable images that show each muscle group. On pressing a muscle group image, the user is navigated to a screen displaying a rendered FlatList of both the in-app exercises and user's own created exercises for that specific muscle group. In order to achieve this, the in-app database containing all the exercises is filtered to only include the exercises of that particular muscle group

```
const exercisesdb = require('./exercisesDatabase.json');

  function filterOutRows(database) {
  const result = database.filter(item => item.MuscleGroup == 'Shoulders');
  return result;
}

const shoulderExercises = filterOutRows(exercisesdb);
```

Figure 7.87: Code Snippet: Importing the In-App Exercise Database and Filtering Only For Shoulder Exercises on Shoulder Ex List Screen Example

The react useEffect hook is also used to execute an SQL query to retrieve all exercises from the user's created exercises table, where the muscle group is specified to correspond with the muscle group image the user pressed on in the previous screen. Following this, the two lists are then concatenated into one list of exercises for that muscle group

```
var db = openDatabase({name: 'userExercises.db'});

  useEffect(() => {
    db.transaction(tx => {
      tx.executeSql(
        'SELECT * FROM user_exercises WHERE exercise_musclegroup = "Shoulders"',
        [],
        (tx, results) => {
          var temp = [];
          for (let i = 0; i < results.rows.length; ++i)
            temp.push(results.rows.item(i));
          setSavedExFlatListItems(temp);
        },
      );
    });
  }, []);


    const concatDefaultNSavedExercises = shoulderExercises.concat(savedExFlatListItems);
```

Figure 7.88: Code Snippet: UseEffect for Executing SQL Query to Get User Created Shoulder Exercises

For each exercise displayed in the flat list, there are two inputs: one for the user to input their target sets and the other for the user to input their target reps. This is implemented so that if the user performs the exercise at any point, the target sets and reps can be compared to the actual sets and reps they have performed. Along with this, there is a plus icon which, when pressed, adds the exercise to the user's created workout SQL table (the name of the workout SQL table being passed as a prop to this screen).

```
var db2 = openDatabase({name: 'userWorkouts.db'});

const InsertData = (
  exerciseName,
  muscleGroup,
  type,
  equipment,
  setsInput,
  repsInput,
) => {
  db2.transaction(function (tx) {
    tx.executeSql(
      'INSERT INTO ' +
        workoutNamePassed +
        ' (exercise_name, exercise_musclegroup, exercise_type, exercise_equipment,
      [exerciseName, muscleGroup, type, equipment, setsInput, repsInput],
      (tx, results) => {
        console.log('Results', results.rowsAffected);
        if (results.rowsAffected > 0) {
          alert('Exercise Has Been Added To The Workout');
        } else console.log('failed....');
      },
    );
  });
};

  return (
  ..................

  onPress={() => {
                if (setsAndRepsValidation(setsInput, repsInput)) {
                  Alert.alert(
                    'Add ' +
                      item.Exercise +
                      'To ' +
                      actualWorkoutName +
                      '?',
                    ' Sets: ' + setsInput + ' Reps: ' + repsInput,
                    [
                      {
                        text: 'OK',
                        onPress: () => {
                          InsertData(
                            item.Exercise,
                            item.MuscleGroup,
                            item.Type,
                            item.Equipment,
                            setsInput,
                            repsInput,
                          );
                        },
                      },
                    ],
                  );
                }
              }}
  )
```

Figure 7.89: Insert Data Function for Inserting Exercises to Created Workout

Once the user has finished creating their workout, they can then view all of the exercises, target reps and target sets they have added to the workout on the workout overview screen. Each exercise displayed in this list includes a delete icon, which executes the SQL query to delete the exercise from the workout if the user presses on it.

```
const deleteData = exerciseID => {
  db.transaction(function (tx) {
    tx.executeSql(
      'DELETE FROM ' + workoutNamePassed + ' where exercise_id=?',
      [exerciseID],
      (tx, results) => {
        console.log('Results', results.rowsAffected);
        if (results.rowsAffected > 0) {
          console.log('Data Deleted Successfully....');
        } else console.log('Failed...');
      },
    );
  });
};
```

Figure 7.90: Code Snippet: Delete Function for Deleting Exercise from Created Workout

7.3.5.3 Creating an Exercise

The create an exercise form is similar in terms of functionality to the create a food form in the diet section. Here, an SQLite table is created when the screen renders called 'user_exercises', and an insert data function is constructed which takes the form inputs as parameters and executes the SQL query to append it into the user_exercises table. An alert message is displayed on the success of this function, and a console.log message on the failure for debugging purposes if issues arise.

```
var db = openDatabase({name: 'userExercises.db'});

  const InsertData = (exerciseName, muscleGroup, type, equipment) => {
    db.transaction(function (tx) {
      tx.executeSql(
        'INSERT INTO user_exercises (exercise_name, exercise_musclegroup, exercise_type, exercise_equipment) VALUES (?,?,?,?)',
        [exerciseName, muscleGroup, type, equipment],
        (tx, results) => {
          console.log('Results', results.rowsAffected);
          console.log(results);
          if (results.rowsAffected > 0) {
            Alert.alert('Exercise Created And Saved To Workout');
          } else console.log('failed....');
        },
      );
    });
  };
```

Figure 7.91: Code Snippet: Function for Inserting Created Exercise into User Created Exercises SQL Table

7.3.5.4 Displaying User-Created Workouts

The My Workouts tab of the begin workouts screen is where the user-created workouts are displayed. Similarly to how the user's saved foods are displayed in the diet section, the user-created workouts are

acquired from the user workouts SQL database in a useEffect hook so that they appear when the screen renders, and each workout listed has a delete button which, when pressed, executes the SQL query to delete the table for that workout in the database.

```
var db = openDatabase({name: 'userWorkouts.db'});

  useEffect(() => {
    db.transaction(tx => {
      tx.executeSql(
        'SELECT name FROM sqlite_master WHERE type = "table"',
        [],
        (tx, results) => {
          var temp = [];
          for (let i = 0; i < results.rows.length; ++i)
            temp.push(results.rows.item(i));
          setSavedWorkutFlatListItems(temp);
        },
      );
    });
  }, []);

    const deleteTable = workoutNamePassed => {
    db.transaction(function (tx) {
      tx.executeSql(
        'DROP TABLE IF EXISTS ' + workoutNamePassed,
        [],
        (tx, results) => {
          alert('Workout Deleted Successfully');
        },
        (tx, error) => {
          console.log(error);
        },
      );
    });
  };
```

Figure 7.92: Code Snippet: UseEffect for Getting Saved Workout Tables from SQL Database + Delete Function for Deleting a Workout Table from SQL Database

7.3.5.5 Performing a Workout

For each user-created and generated workouts displayed in the rendered FlatLists, there exists a perform workout button which, once pressed, passes the name of the workout that was pressed as a prop to the perform workout screen. The perform workout screen then uses the workout name which has been passed as a prop from the previous screen to search for the associated workout table in the database and then display the exercises of the workout in a FlatList component on this screen.

```
const userCreatedWorkoutTableName = route.params.workoutNamePassed;

  useEffect(() => {
    db.transaction(tx => {
      tx.executeSql(
        'SELECT * FROM ' + userCreatedWorkoutTableName,
        [],
        (tx, results) => {
          var temp = [];
          for (let i = 0; i < results.rows.length; ++i)
            temp.push(results.rows.item(i));
          setFlatListItems(temp);
        },
      );
    });
  }, []);
```

Figure 7.93: Code Snippet: Getting the Workout Table Name Prop From Previous Screen and UseEffect For Getting the Data in the SQL Table

For each exercise displayed in the list, the target sets and target reps are displayed as well as the previous records that the user has achieved when performing that exercise in the past. Furthermore, there exists a reps and weight input and a green tick icon. This is so that the user can log the sets they perform. In order to log their performance, a database has been constructed which holds the user's workout history. The tables in this database are named after today's date and formatted identically to the formatting of the date tables in the diet section storing the user's caloric/macronutrient intake history. On pressing the green tick icon, the exercise, reps, and weight completed is appended into the table logging today's workout performed.

```
var db2 = openDatabase({name: 'userWorkoutHistory.db'});

  useEffect(() => {
    db2.transaction(function (txn) {
      txn.executeSql(
        "SELECT name FROM sqlite_master WHERE type='table' AND name=" +
          "'" +
          tableNameDate +
          "'",
        [],
        function (tx, res) {
          console.log('item:', res.rows.length);
          if (res.rows.length == 0) {
            txn.executeSql('DROP TABLE IF EXISTS ' + tableNameDate, []);
            txn.executeSql(
              'CREATE TABLE IF NOT EXISTS ' +
                tableNameDate +
                '(exercise_id INTEGER PRIMARY KEY AUTOINCREMENT, exercise_name VARCHAR(30), exercise_reps INT(15),
              [],
            );
          }
        },
      );
    });
  }, []);

    const InsertData = (exerciseName, reps, weight, duration) => {
    db2.transaction(function (tx) {
      tx.executeSql(
        'INSERT INTO ' +
          tableNameDate +
          ' (exercise_name, exercise_reps, exercise_weight, exercise_duration) VALUES (?,?,?,?)',
        [exerciseName, reps, weight, duration],
        (tx, results) => {
          console.log('Results', results.rowsAffected);
          if (results.rowsAffected > 0) {
            console.log('Set Added');
            showMessage({
              message: 'Set Added',
              type: 'info',
              color: 'black',
              backgroundColor: 'green',
            });
          } else console.log('failed....');
        },
      );
    });
  };

  return (
  ..................

            onPress={() => {
                    if (validateRepsWeightInput(repsInput, weightInput)) {

                      InsertData(
                        item.exercise_name,
                        String(repsInput),
                        String(weightInput),
                        String(remainingSecs),
                      );
                    }
                  }}
```

Figure 7.94: Code Snippet: useEffect for Creating Table for Today's Workout; Insert Data Function for Inserting Exercise Sets Performed Into Today's Workout Table; On-Press Method to Insert Sets On Pressing the Green Circular Tick Icon

An additional feature added to this screen is a stopwatch, so that the user can record the duration of a workout they performed. This stopwatch uses the useState react hook to: set the state of the current time displayed on the stopwatch (remainingSecs); set the state of whether the stop watch is activated of not; and to get the state of the current time displayed on the stop watch which is then used to add to the database table of today's workout in the insert function. These states are implemented into a useEffect hook, which is

used to re-render the stopwatch component every second so that its state updates and displays an increment of 1 second after every second. The algorithm to implement this is based on a blog posted by Nabendu Biswas (2019).

```
const formatNumber = number => `0${number}`.slice(-2);

const getRemaining = time => {
  const mins = Math.floor(time / 60);
  const secs = time - mins * 60;
  return {mins: formatNumber(mins), secs: formatNumber(secs)};
};

const PerformWorkout = ({route}, props) => {
  // CONFIGURING THE STOPWATCH: REFERENCE: https://thewebdev.tech/reactnative-simple-timer-app https://www.youtube.com/watch?v=zpSwD5qoVQs
  const [remainingSecs, setRemainingSecs] = useState(0);
  const [isActive, setIsActive] = useState(false);
  const {mins, secs} = getRemaining(remainingSecs);

  var toggle = () => {
    setIsActive(!isActive);
  };

  var reset = () => {
    setRemainingSecs(0);
    setIsActive(false);
  };

  useEffect(() => {
    let interval = null;
    if (isActive) {
      interval = setInterval(() => {
        setRemainingSecs(remainingSecs => remainingSecs + 1);
      }, 1000);
    } else if (!isActive && remainingSecs !== 0) {
      clearInterval(interval);
    }
    return () => clearInterval(interval);
  }, [isActive, remainingSecs]);
```

Figure 7.95: Stop Watch Configuration Algorithm

Once the user finishes performing a workout, they are navigated to a screen displaying an overview of the workout they have performed. This screen is set up identical to how the create a workout over screen is set up, whereby the table logging the performed workout is displayed in a FlatList on the screen, and a delete icon is present for each item on the screen which enables the user to delete an exercise from the list of exercises they performed in the workout.

7.3.5.6 View Workout Progress and Exercise Progress

The view workout progress screen displays graphs showing the workout progress the user has made from their performed workouts over time. Similarly to how the diet section implements graphs showing the daily intake over time, this screen uses the useEffect hook to get all the columns from the total_daily_metrics table which stores the date, total reps, actual reps x sets, and target reps x sets amongst many other analysis metrics, and displays the analysis metric (each column being stored in it's own array) against the date array on a graph using the LineChart component from the react-native-chart-kit library.

```
        var temp = [];
        for (let i = 0; i < results.rows.length; ++i)
          temp.push(results.rows.item(i));
        setDateHistory(temp);
      },
    );
  });
}, []);


......

return (
          <LineChart
          data={{
            labels: dateHistory,
            datasets: [
              {
                data: targetVsActualSets,
              },
            ],
            legend: ['Actual Sets - Target Sets'],
          }}
          width={Dimensions.get('window').width - 16}
          height={200}
          chartConfig={{}}

        />
        )
```

Figure 7.96: Code Snippet: UseEffect Appended Variable Snipped for Getting Date Column; Rendered Line Chart Example Showing Date against Target vs Actual Sets

Conditional statements have been constructed which analyse the user's actual reps and sets completed per workout against their target reps and sets, and displays a motivational message if the user is on track telling them they are on track, or if the user is exceeding their targets, or if the user is falling behind their targets.

```
function displayUserProgressMsg(targetRepsXSets, actualRepsXSets) {
  const repsXSetsTargetLatest = repsXSetsTarget.slice(-1);
  const repsXSetsActualLatest = repsXSetsActual.slice(-1);

    if (parseInt(repsXSetsTargetLatest) < parseInt(repsXSetsActualLatest)) {
      return (
        <Text style={{color: '#90ee90', fontSize: 18}}>
          You are currently exceeding your targets! Keep Going!{' '}
        </Text>
      );
    }

    if (parseInt(repsXSetsTargetLatest) == parseInt(repsXSetsActualLatest)) {
      return (
        <Text style={{color: 'white', fontSize: 18}}>
          You are currently on track with your targets! Keep it Up!{' '}
        </Text>
      );
    }
  } else {
    return (
      <Text style={{color: 'white', fontSize: 18}}>
        No Data To Analyse, Begin a Workout!{' '}
      </Text>
    );
  }
}
```

Figure 7.97: Code Snippet: Function for Displaying Conditional Statements Analysing User Workout Performance

A similar implementation has been constructed for the exercise progress feature. When the user searches for an exercise for a particular muscle group, a view progress button is rendered for each exercise displayed in the list. On pressing the view progress button, the associated exercise name is passed as a prop to the next screen, being the display exercise progress screen.

```
<Button
   style={{
     alignSelf: 'center',
   }}
   mode="contained"
   color="#CBC3E3"
   onPress={() => {
     navigation.navigate('Display Progress', {
       nameOfPreviousScrn: 'Shoulders Progress',
       exerciseName: item.Exercise,
     });
   }}>
   View Progress
 </Button>
```

Figure 7.98: OnPress Method Passing Exercise User Pressed On as Prop For Display Exercise Progress Screen

This screen executes the SQL query inside the useEffect hook (so that it is executed on the screen rendering) to search for the table name that is identical to the exercise name passed as a prop (which contains the full history of reps and weights performed for that particular exercise) and append it to an array.

Using the .reduce() JavaScript in-built function, all of the reps which have the same date in their row are added together to calculate the total number of reps performed on each date.

```
// THIS PRODUCES AN ARRAY WHERE EACH DATE OCCUPIES ONE ROW AND EACH ROW HAS A COLUMN THA
const newResults = flatListItems.reduce(
  (acc, item) => ({
    ...acc,
    [item.today_date]: (acc[item.today_date] || 0) + item.exercise_reps,
  }),
  {},
);

const finalResult = Object.keys(newResults).map(key => ({
  date: key,
  reps: newResults[key],
}));

// SEPERATING THE DATE ARRAY AND REPS ARRAY FOR USE IN THE LINE GRAPH COMPONENT:
let dateArray = finalResult.map((item, index, arr) => {
  return item.date;
});

  let repsArray = finalResult.map((item, index, arr) => {
  return item.reps;
});
```

Figure 7.99: Code Snippet: Data Manipulation Functions For Exercise Data To Be Displayed

The same process is then repeated in order to find the sum of the weight lifted on each date. In order to find out the average weight, the number of sets per date must first be calculated. In this case, the number of sets are the number of rows that contain the same date. Using the .reduce() JavaScript function, this has been calculated by increasing the count by +1 every time a row is of the same date. Then, the average weight lifted function has been constructed which uses the .map JavaScript function to divide each total weight on each date by the number of sets completed on that date.

The volume completed of an exercise in a workout is the total sets x reps x average weight across sets. Total sets x reps is equivalent to the total number of reps completed across all sets, which has been calculated in the above function. Average weight across sets has also been calculated. Thus, in order to calculate the volume, the .map() JavaScript function has been used to multiply the average weight array by the total reps array. The resulting volume array is then plotted against the dates array and displayed of the rendered LineChart component.

```
const setsPerWorkout = flatListItems.reduce(function (r, a) {
  r[a.today_date] = (r[a.today_date] || 0) + 1;
  return r;
}, {});

const setsVals = Object.keys(setsPerWorkout).map(key => setsPerWorkout[key]);

// DOING THE SAME THING AS DONE FOR REPS BUT FOR WEIGHT LIFTED:
const newResults2 = flatListItems.reduce(
  (acc, item) => ({
    ...acc,
    [item.today_date]: (acc[item.today_date] || 0) + item.exercise_weight,
  }),
  {},
);

const finalResult2 = Object.keys(newResults2).map(key => ({
  date: key,
  weight: newResults2[key],
}));

let weightArray = finalResult2.map((item, index, arr) => {
  return item.weight;
});

//FINDING THE AVERAGE WEIGHT LIFTED BY DIVIDING THE SUMMED WEIGHTS BY THE NUMBER OF SETS COMPLETED
var weightAvgArray = weightArray.map(function (n, i) {
  return n / setsVals[i];
});

// CALCULATE THE VOLUME BY: REPS X SETS (WHICH IS ALREADY DONE BY SUMMING UP ALL THE REPS OF THAT EXERCISE
var volume = weightAvgArray.map(function (n, i) {
  return n * repsArray[i];
});
```

Figure 7.100: Code Snippet: Data Manipulation for Exercise Data To Be Displayed Part 2

Additionally, a message is rendered at the top of this screen which tells the user their personal record for the exercise. In order to display this message, a function has been constructed which applies the Math.max JavaScript function to the exercise_weight column in the exercises SQL table.

```
var maxWeightVal = Math.max.apply(
  Math,
  flatListItems.map(function (o) {
    return o.exercise_weight;
  }),
);

  function displayWeightPR() {
  if (maxWeightVal == 0) {
    return (
      <Text>You Have Not Yet Performed This Exercise</Text>
    )
  }
  else {
    return (
      <Text>Your Current Personal Record For This Exercise Is {maxWeightVal}kg!</Text>
    )
  }
}
```

Figure 7.101: Code Snippet: Functions for Finding the Max Weight Lifted and Displaying it On Screen

7.3.5.7 Testing & Resolutions

See Appendix 9.1.5 for the full testing table for this increment.

The testing phase of this increment consisted of the same tests as that of the diet section but applied to the features of the fitness section. All-in-all, there were no issues or failures that had risen when testing this increment, with the exception of one issue that had been encountered when testing the rendered list of exercises that were supposed to be displayed on the muscle group exercises list screen. The issue was that the .filter in-built JavaScript function did not filter the in-built exercises database array to that of the muscle group exercise it was programmed to.

In order to resolve this issue, the negation operator was used to negate all muscle groups except the muscle group the screen was intended to show the exercises for. For example, the code snippets below show how the change made to the code for the screen that displays all exercises of the Shoulder muscle group. The original filter function has been replaced by a negation on the filter function to filter out all exercises not associated with Shoulder exercises. This change then correctly displayed only shoulder exercises in the FlatList component on the screen.

```
const exercisesdb = require('./exercisesDatabase.json');

  function filterOutRows(database) {
  const result = database.filter(item => item.MuscleGroup == 'Shoulders');
  return result;
}

const shoulderExercises = filterOutRows(exercisesdb);
```

Figure 7.102: Code Snippet: Original Code for Filtering Out Everything Except Shoulder Exercises

```
const exercisesdb = require('./exercisesDatabase.json');

  function filterOutRows(database) {
  const result = database.filter(item => item.MuscleGroup != 'Chest');
  const result2 = result.filter(item => item.MuscleGroup != 'Biceps');
  const result3 = result2.filter(item => item.MuscleGroup != 'Back');
  const result4 = result3.filter(item => item.MuscleGroup != 'Abs');
  const result5 = result4.filter(item => item.MuscleGroup != 'Triceps');
  const result6 = result5.filter(item => item.MuscleGroup != 'Legs');
  const result7 = result6.filter(item => item.MuscleGroup != 'Full Body');

  return result7;
}
const shoulderExercises = filterOutRows(exercisesdb);
```

Figure 7.103: Code Snippet: Resolved Code for Filtering Out Everything Except Shoulder Exercises

7.3.5.8 Results Display

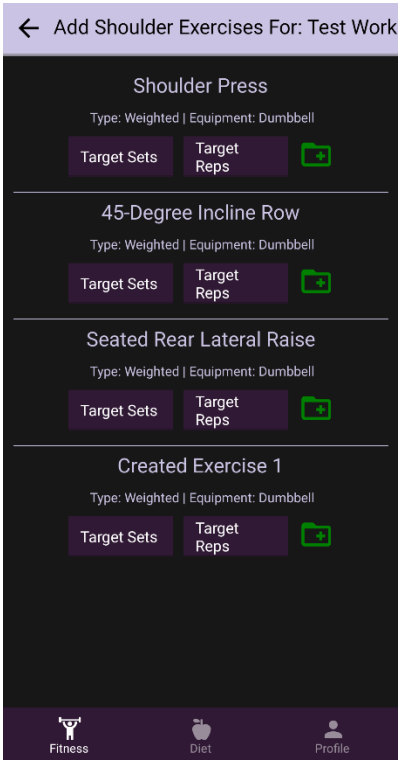


Figure 7.104: Screenshot: List of Shoulder Exercises to Add to Created Workout

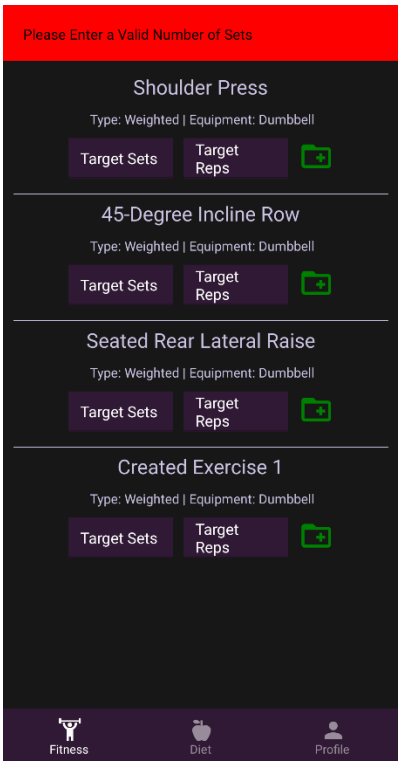


Figure 7.105: Screenshot: Error Message on Inserting Invalid Sets/Reps for Adding Exercise to Created Workout

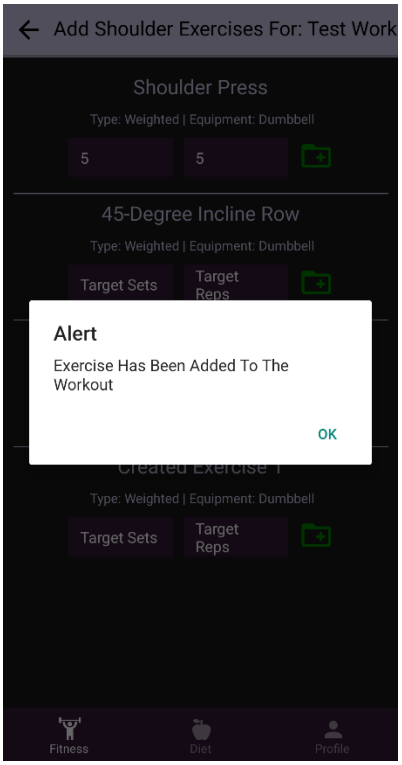


Figure 7.106: Screenshot: Alert Message on Adding Exercise to Created Workout

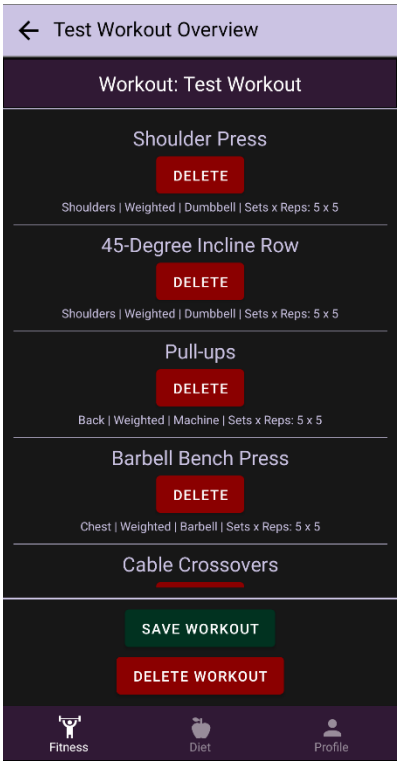


Figure 7.107: Screenshot: Created Workout Overview Screen

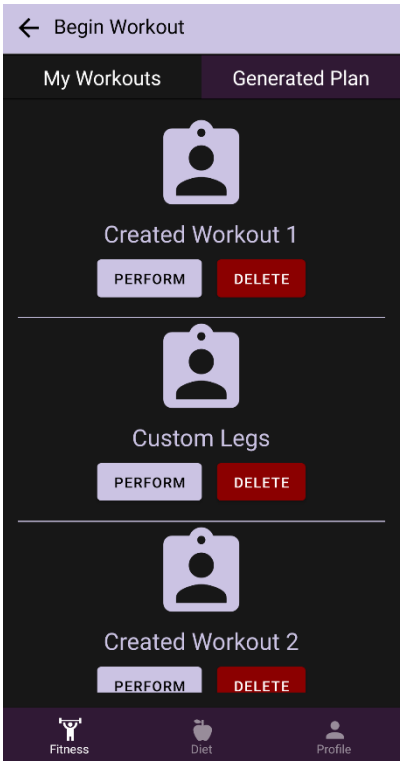


Figure 7.108: Screenshot: List of User's Created Workouts

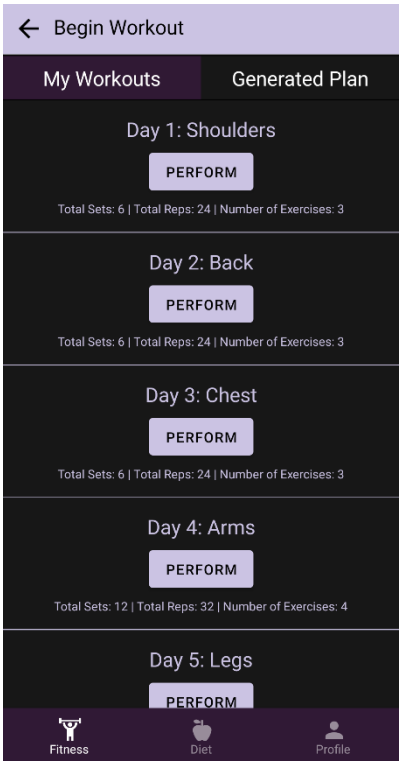


Figure 7.109: Screenshot: Generated Workout Plan for User

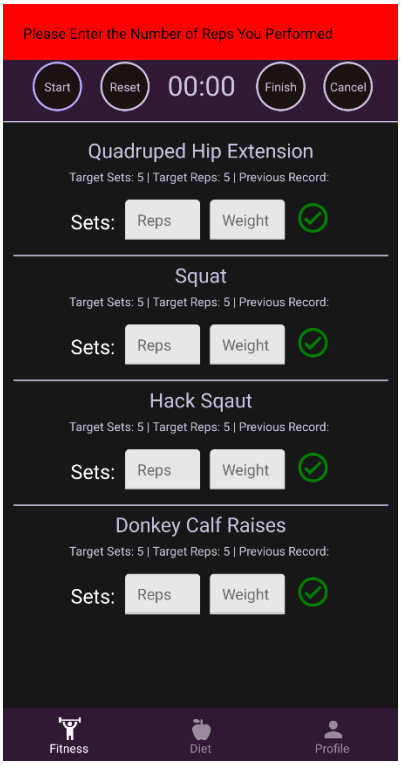


Figure 7.110: Screenshot: Error Message on Inserting Invalid Data On Perform Workout Screen

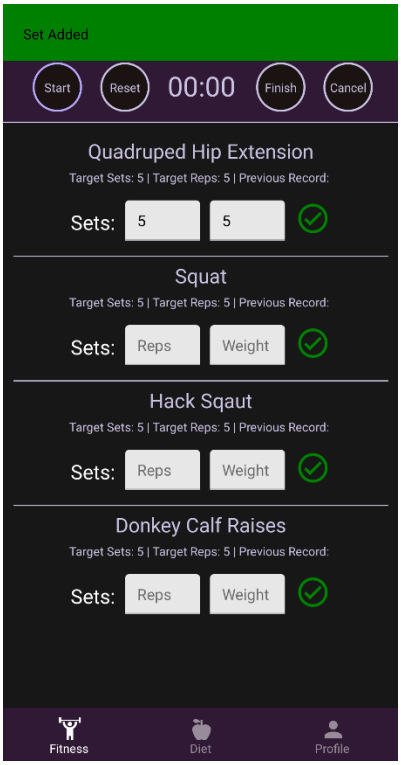


Figure 7.111: Screenshot: Success Message on Adding Valid Set To Performed Workout

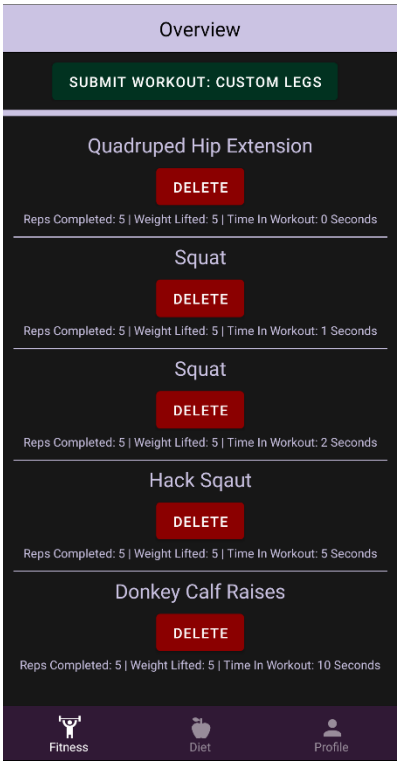


Figure 7.112: Screenshot: Completed Workout Overview Screen

Figure 7.113: Screenshot: Exercise Progress Screen Shoulder Press Example

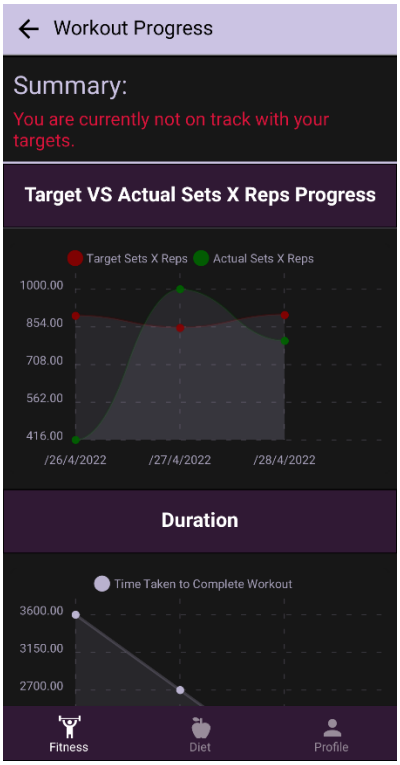


Figure 7.114: Screenshot: Workout History Progress Screen

8 Final Project Evaluation

This section is an evaluation of the output of this project. The quality of the final project is measured with respect to the dimensions identified in section 3 and the requirements of section 5. The overall project is evaluated against the aims that were set out at the beginning, and how well the project reached its aims, given the time frame of the project scope.

8.1 Final Project Vs. Requirements

In evaluating the output of this project against the defined requirements, since a huge amount of focus was devoted to ensuring the core functionality had been implemented, tested, and debugged, Strategym meets all the necessary requirements that had been laid out. Additionally, Strategym meets many extended requirements too, including:

- The user remaining logged in after logging in initially, even when closing and re-opening the application
- The user being able to change their login email
- The user being able to change their login password
- The user being able to input their physical attributes and retrieve an estimation of their bodyfat percentage based on a neural network trained on the dataset of physical attributes against bodyfat percentage outcomes
- The user being recommended a dietary goal based on the results of their bodyfat percentage calculations
- The user being given positive feedback in the form of motivation with regards to their workout/dietary progress
- The user being told whether they are on track, falling behind, or exceeding their fitness/dietary goals
- The user being able to view a visual display of the progress they have made for workouts, exercises and for their diet

8.2 Final Project Vs. Dimensions

During the literature review of this project, 4 scales had been identified in terms of what fitness application users value.

The convenience/ease-of-use scale is scored at 7/10. A lot of emphasis was put on automating the process of calculating fitness and dietary metrics so that the user would not have to do so themselves. This involved the automation of generating the user a workout plan, calculating their workout progress, and calculating their daily caloric and macronutrient goals amongst many others. However, along this metric there is still room for improvement. Firstly, Strategym does not include a search bar for the user to search through the exercises database or foods database, which can be a hassle for the user to search for the correct food to add to their daily intake. Furthermore, the bodyfat percentage form includes 13 inputs required from the user for the bodyfat percentage prediction to be executed. This is also inconvenient for the user as it does not only involve inputting this data, but also physically checking these measurements too. This could be improved by implementing Computer Vision based artificial intelligence which uses the devices camera to scan the body of the user and output estimated values with regards to their physical measurements.

The customisation scale is scored at 6/10. Strategym does include features that enhance the customisation of the application. For example, the user can create and delete custom workouts and exercises, create and delete custom foods, and manage their profile details. However, there are not any features in the application that allow the user to customise the user interface to their liking in any way. This could be improved by adding a night and day mode to the application's theme, where the user can select whether they want the UI to be of a dark theme or light theme. Furthermore, extra features such as the user being able to upload and update a profile photo associated with their Strategym account would also add to the customisation dimension of this application.

The fitness tracking capabilities is scored at 8/10. Much of the back-end architecture of the application revolved around enabling the user to view the workout progress, exercise progress, and dietary progress. Thus, the application is packed with features that enable the user to track their fitness under a microscope so to speak. Just one example of this is that daily caloric and macronutrient intake tracking capability; whereby the user can track how many calories they have remaining for the day, how much protein they have remaining, how much carbohydrates they have remaining, and how many fats they have remaining, all visually displayed on their own pie charts. However, there is still room for improvement in this scale. Strategym does not include any features where a user can specifically see what food they had eaten on a prior date, nor a list of exercises they had performed in a workout on a previous date. The addition of these two features would improve the fitness tracking capabilities of Strategym.

The personalisation scale is scored at 7/10. the personalisation aspect of the application includes functionality that generates a workout plan for the user based on inputs regarding their workout goal and

training frequency; functionality that calculates the specific user's profile details BMR, TDEE, goal calories and goal macronutrients based on inputs regarding their physical measurements such as height, weight, and age, and based on their dietary goals; and features such as the motivational message directed towards the user informing them of whether they are on track or falling behind their fitness and dietary goals, inspiring them to keep going. However, there is still room for improvement along this dimension. For example, there are no features in the application that recommend the user any food items based on the history of foods they have consumed, nor is there any features related to recommending the user any exercises they should include in a workout based on how frequently they have performed specific exercises in the past. Adding these features to the application would improve its functionality on the personalisation scale.

8.3 Final Project Vs. Aims and Scope

The scope of this project was considerably large for the timeframe that was given for it to be completed in, since this application was closely knit to addressing the gaps in the fitness application industry. The application was intended to be the 'complete package' for fitness users, effectively addressing all the dimensions of a fitness application that fitness users value. As a result of this, concessions had to be made to some functions constructed in its code, which were still successful in realizing the intended outcomes however not optimal.

For example, the code included functions such as filtering out seven muscle groups line-by-line in order to output a list of exercises from one muscle group; the function worked as intended but was not the optimal solution. Another example of this is the implemented neural network, where concessions had to be made to the configuration of the iterations and learning rate. However, this time was spent more positively on areas regarding the core functionality of the application, resulting in an application that met the necessary requirements laid out in section 5.

A higher emphasis was placed on the back-end architecture of the application than the front-end design. The intention throughout the project, as discussed with project supervisor Walter, was to get a basic system with its core functionality working first, and then focus on improving the user interface. Having an adequate level of proficiency in SQLite database implementation as well as Firebase implementation from part b of this Computer Science course, the expected time to set these up was underestimated in comparison to the actual time it took. The learning curve of learning the react-native framework and implementing the backend databases in this framework was significantly bigger than expected, thus took a lot longer than expected.

Also, as already mentioned, the scope of this project was very large for it's time frame and included core-functionality that is considered too audacious in terms of amount. This, concatenated with the steep learning curve of react-native, resulted in not much time left-over to improve the front-end to a more desirable level. Therefore, in analysing the final project in its entirety, the most significant improvements that could be made if completing projects of a similar nature within a similar time-frame, would be a smaller project scope with a more detailed focus on less features, and better time-management in assessing how long certain tasks or will take and how much time should be dedicated to completing them.

9 Appendices

9.1 Testing:

9.1.1 Increment 1: User Interface Design Implementation

Action	Expected Outcome	Result	Issues	Executed Solution
Examine UI of all static screens	• Components all placed where intended • Design, layout, structure all set how intended	• All screens passed except profile settings screen	• Change profile details form; change email form; and change password forms all cluttered on profile details screen.	• Create top tab buttons "profile details" and "login details" for the profile settings screen • Place the profile details form on the profile details tab screen • Place the change email and change password forms on the login details tab screen • Set up the animation between the two tabs the same way it has been set up for the profile home screen

				and login/registration screen.
For each form: **1 – press top input field** **2 – press enter on keyboard** **3 – repeat steps 1 and 2 till you reach the last input field**	• On pressing enter, the focus should shift to the next input field in the form for the user to input data into the next input field	• Passed		
Scroll right on the login/registration screen	• Registration form is displayed • Sign Up top tap button interpolates to chalk background colour • Login top tab button interpolates to dark purple background colour	• Passed • Passed • Passed		
Scroll left on the login/registration screen	• Login form is displayed • Login top tap button interpolates to chalk background colour • Sign Up top tab button interpolates to dark purple background colour	• Passed • Passed • Passed		

Press the Sign Up top tab button on the login/registration screen	• ScrollView scrolls to the registration form • Sign Up top tap button interpolates to chalk background colour • Login top tab button interpolates to dark purple background colour	• Passed • Passed • Passed		
Press the Login top tab button on the login/registration screen	• ScrollView scrolls to the login form • Login top tab button interpolates to chalk background colour • Sign Up top tab button interpolates to dark purple background colour	• Passed • Passed • Passed		
Scroll right on the profile home screen	• My Statistics screen contents is displayed • My Statistics top tap button interpolates to chalk background colour • Menu top tab button interpolates to dark purple background colour	• Passed • Passed • Passed		

Scroll left on the profile home screen	• Menu screen contents is displayed • Menu top tap button interpolates to chalk background colour • My Statistics top tab button interpolates to dark purple background colour	• Passed • Passed • Passed		
Press the My Statistics top tab button on the profile home screen	• ScrollView scrolls to the My Statistics screen contents • My Statistics top tab button interpolates to chalk background colour • Menu top tab button interpolates to dark purple background colour	• Passed • Passed • Passed		
Press the Menu top tab button on the profile home screen	• ScrollView scrolls to the Menu screen contents • Menu top tab button interpolates to chalk background colour • My Statistics top tab button interpolates to dark purple background colour	• Passed • Passed • Passed		

Scroll right on the begin workout screen	• Generated Plan screen contents is displayed • Generated Plan top tab button interpolates to chalk background colour • My Workouts top tab button interpolates to dark purple background colour	• Passed • Passed • Passed		
Scroll left on the begin workout screen	• My Workouts screen contents is displayed • My Workouts top tab button interpolates to chalk background colour • Generated Plan top tab button interpolates to dark purple background colour	• Passed • Passed • Passed		
Press the My Workouts top tab button on the begin workouts screen	• ScrollView scrolls to the My Workouts screen contents • My Workouts top tab button interpolates to chalk background colour • Generated Plans top tab button interpolates to dark	• Passed • Passed • Passed		

	purple background colour			
Press the Generated Plans top tab button on the begin workouts screen	• ScrollView scrolls to the Generated Plans screen contents • Generated Plans top tab button interpolates to chalk background colour • My Workouts top tab button interpolates to dark purple background colour	• Passed • Passed • Passed		

9.1.2 Increment 2: Navigation Implementation and Login Functionality

Action	Expected Outcome	Result	Issues	Executed Solution
1 - Re-install application **2 - Launch application**	• On boarding screens displayed	• Passed		
1 - Fill in register form with invalid details **2 - Press sign up**	• Displays invalid form error message	• Passed		
1 – Fill in register form with valid details	• Navigates to the fitness tab of the main application • User Firestore document created	• Passed • Passed (Test output in Firestore GUI displayed in *Figure 7.31: Firestore GUI Displaying New*		

2 – Press sign up	and initialized, viewable in Firestore GUI	*Document for Newly Registered User)*		
1 – Fill register form with already registered email + valid details **2 – Press sign up**	• Alert box displaying message that the email is already taken	• Failed	• Typo in error code "auth/email already-in-use".	• Typo addressed and changed to "auth/email-already-in-use"
1 - Fill in login form with invalid details **2 - Press sign up**	• Displays invalid form error message	• Passed		
1 – Fill in login form with valid details **2 – Press sign up**	• Navigates to the fitness tab of the main application	• Passed		
1 – Logout of application	• Navigates back to the login/registration screen of the AuthStack	• Passed		
1 – Logout of the application **2 – Close the application** **3 – Relaunch the application**	• Routes to the AuthStack login/registration screen	• Passed		

1 – Login to the application **2 – Close the application** **3 – Relaunch the application**	• Routes directly to the AppStack fitness home screen	• Passed		
For each screen of the main application, press any buttons related to navigating around the screens of the section	• Navigates to the screen associated with the button's onPress method	• Passed		
For each screen of the main application higher than level 0 of the navigation tree, press the back button in the header	• Navigates to the previous screen in the navigation stack	• Passed		

9.1.3 Increment 3: Profile Side Implementation

Action	Expected Outcome	Result	Issues	Executed Solution
For all forms, enter invalid data and press submit	Red notification appears at the top of the screen informing the user of the invalid form	• Passed		
Input wrong current password into the	Alert message telling the user they have	• Passed		

update email form and press submit button	entered the wrong current password			
Input wrong current password into the change password form and press submit button	Alert message telling the user they have entered the wrong current password	• Passed		
1 - Enter valid details into the update email form and press submit **2 - Logout of account** **3 - Log in using new auth details**	Successful login with new authentication details	• Passed		
1 - Enter valid details into the update password form and press submit **2 - Logout of account** **3 - Login using new auth details**	Successful login with new authentication details	• Passed		
1 – For all Firestore related forms, enter valid details and press submit **2 – Open Firebase console in browser**	All details corresponding to the form calculated and updated in Firestore document	• Passed		

and check user's Firestore document				
1 – Enter valid details in bodyfat percentage form and press submit **2 – Open Firebase console in browser and check if user's bodyfat percentage is updated in Firestore document**	Bodyfat percentage prediction calculated and updated in user Firestore document	• Passed		
1 – Fill in valid details in all Firestore forms and press submit **2 – navigate to the statistics tab on profile home screen**	Statistics screen shows breakdown of all user statistics based on their data input submissions	• Passed		

9.1.4 Increment 4: Diet Side Implementation

Action	Expected Outcome	Result	Issues	Executed Solution
Navigate to the Add Foods Screen	• In-App food database listed on screen	• Passed		
On the Add Foods screen, leave the amount input empty and press the basket icon	• Error message displayed regarding invalid amount input on adding	• Passed		

	food item to today's intake			
On the Add Foods screen, type in a number in the amount input and press the basket icon	• Alert message displayed confirming the food item the user is requesting to add to today's intake	• Passed		
1 - On the Add Foods screen, type in a number in the amount input and press the basket icon **2 - Press Yes on the Alert message** **3 - Open up the generated SQLite database in VS Code using the SQLite browser extension**	• User's today's intake SQLite database table created • Food item added to user's today's intake database	• Passed • Passed		
1 - On the Add Foods screen, type in a number in the amount input and press the basket icon **2 - Press Yes on the Alert message** **3 - Navigate to the diet home sreen**	• Today's intake database table displayed on diet home screen with food item added by the user	• Passed		
On the diet home screen, press the bin	• Alert message displayed	• Passed		

icon on a food item listed in the FlatList	confirming the food item the user is requesting to delete from today's intake			
1 - On the diet home screen, press the bin icon on a food item listed in the FlatList **2 - Press Yes on the Alert Message** **3 - Open up the today's intake SQLite database in VS Code using the SQLite browser extension**	• Food item deleted from FlatList displaying today's intake • Food item deleted from SQL table regarding today's intake	• Passed • Passed		
1 - Navigate to the create foods screen **2 - Press the submit button on invalid input form**	• Error message displayed regarding invalid data input for create food item	• Passed		
1 - Navigate to the create foods screen **2 - Press the submit button on valid input form** **3 - Open up the demo_food SQLite database table in VS Code using the SQLite browser extension**	• SQLite table created for storing user saved foods • User created food item added SQLite user created food items table	• Passed • Passed		

1 - Navigate to the diet home screen **2 - Press the "Save Today's Intake" button** **3 - Open up the daily totals SQLite database table in VS Code using the SQLite browser extension**	• SQLite total calories and macronutrients table created • Today's total calories and macronutrients appended to totals SQL table	• Passed • Passed		
1 - Navigate to the create foods screen **2 - Press the submit button on valid input form** **3 - Navigate to the saved foods screen**	• User saved foods listed on screen	• Passed		
1 - Navigate to the saved foods screen **2 - Press the bin icon next to a food item in the displayed list** **3 - Open up the demo_food SQLite database table in VS Code using the SQLite browser extension**	• Food item deleted from user created foods SQL table • Food item deleted from user saved foods displayed on screen	• Passed • Passed		

1 - Navigate to the Today's Totals Screen **2 - Open up the react-native console output in VS Code**	• Console logging user's goal calories, protein, fats, and carbohydrates acquired from Firestore document	• Passed		
1 - On the Add Foods Screen, type in a number in the amount text input → press the green basked icon → press Yes on the Alert message **2 - Navigate to the Today's Totals screen**	Pie chart displaying: • Calories consumed VS. Calories remaining • Protein consumed VS. Protein remaining • Carbohydrates consumed VS. carbohydrates remaining • Fats consumed VS. Fats remaining	• Passed • Passed • Passed • Passed		
1 - On the Add Foods Screen, type in a number in the amount text input → press the green basked icon → press Yes on the Alert message **2 - Navigate to the Today's Totals screen**	• Calories remaining text is green • Calories remaining text is amber • Calories remaining text is red	• Passed • Passed • Passed		
1 - On the Add Foods Screen, type in a number in the amount text input →	• Protein remaining text is green • Protein remaining text is amber	• Passed • Passed • Passed		

press the green basked icon → press Yes on the Alert message **2 - Navigate to the Today's Totals screen**	• Protein remaining text is red			
1 - On the Add Foods Screen, type in a number in the amount text input → press the green basked icon → press Yes on the Alert message **2 - Navigate to the Today's Totals screen**	• Carbohydrates remaining text is green • Carbodhydrates remaining text is amber • Carbohydrates remaining text is red	• Passed • Passed • Passed		
1 - On the Add Foods Screen, type in a number in the amount text input → press the green basked icon → press Yes on the Alert message **2 - Navigate to the Today's Totals screen**	• Fats remaining text is green • Fats remaining text is amber • Fats remaining text is red	• Passed • Passed • Passed		
1 - On the Add Foods Screen, type in a number in the amount text input → press the green basked icon → press	Line graphs displaying: • Calories consumed over time	• Passed • Passed • Passed • Passed		

Yes on the Alert message **2 - On the diet home screen, press Save Today's Intake and press Yes on the Alert message** **3 - Navigate to the view daily progress over time screen**	• Protein consumed over time • Carbohydrates consumed over time • Fats consumed over time			
1 - On the Add Foods Screen, type in a number in the amount text input → press the green basked icon → press Yes on the Alert message **2 - On the diet home screen, press Save Today's Intake and press Yes on the Alert message** **3 - Navigate to the view daily progress over time screen**	Text displaying: • Average progress of user's caloric intake over time • Average progress of user's fats intake over time • Average progress of user's protein intake over time • Average progress of user's carbohydrates intake overtime	• Passed • Passed • Passed • Passed		

9.1.5 Increment 5: Fitness Side Implementation

Action	Expected Outcome	Result	Issues	Executed Solution

On the create workout screen, press search exercises on empty workout name input	• Error message displayed that informs the user of invalid data input	• Passed		
On the create workout screen, press search exercises on valid workout name input	• Navigate user to the search exercises screen • Display the set workout name on search exercises screen header	• Passed • Passed		
On the search exercises screen, press each muscle group image	• Navigate to the muscle group exercise list screen • List of muscle group exercises displayed on screen	• Passed • Failed	List displaying all exercises from the database, with all the exercise muscle group names being changed to the muscle group that was pressed on in the previous screen. Not displaying only the exercises of the muscle group that was pressed on in the previous screen.	Function to filter out items changed from = [muscle group name] to != [all muscle groups that are not to be displayed on the screen].
On the create exercises screen, press submit button on invalid form input	• Error message informing user of invalid form input	• Passed		
On the create exercises screen,	• Alert message informing the user that the exercise	• Passed • Passed		

press submit button on valid form input	has been created successfully • Created exercise appended to user created workouts SQL table			
Navigate to the shoulders muscle group exercises screen **Press the green plus icon on invalid sets and reps input**	• Error message informing the user of invalid data input	• Passed		
Navigate to the shoulders muscle group exercises screen **Press the green plus icon on valid sets and reps input**	• Alert message telling the user that the exercise has been added to the workout successfully • Exercise appended to the workout name table in SQLite	• Passed • Passed		
Navigate to the shoulders muscle group exercises screen **Press the green plus icon on valid sets and reps input** **Navigate to the created workout overview screen**	• Shoulder exercise listed in the created workout overview screen	• Passed		

Navigate to the shoulders muscle group exercises screen **Press the green plus icon on valid sets and reps input** **Press Yes on Alert Message** **Navigate to the created workout overview screen** **Press the red delete icon on the exercise displayed in list**	• Deletes exercise from list • Deletes exercise from user create workout SQL table	• Passed • Passed		
Press the delete button on the created workout overview screen	• Deletes created workout SQL table • Navigates to the fitness home screen	• Passed • Passed		
Press the save workout button on the created workout overview screen	• Created workout SQL table contains exercises added to created workout • Navigates to the fitness home screen	• Passed • Passed		
Press the cancel button on the search exercises screen	• Deletes created workout SQL table • Navigates to the fitness home screen	• Passed • Passed		
Navigate to the shoulders muscle	• Lists the created workout's name on	• Passed		

group exercises screen **Press the green plus icon on valid sets and reps input** **Navigate to the created workout overview screen and press save workout** **Navigate to the my workouts tab on the begin workouts screen**	the my workouts tab			
Press the delete button on a listed workout on the my workouts tab of the begin workouts screen	• Deletes the created workout from the list of created workouts • Deletes the SQLite created workout table	• Passed • Passed		
Navigate to the generated workout plan tab on the begin workout screen	• Lists the workouts of the workout plan in accordance with the user's fitness goal and training frequency	• Passed		
Press the perform button on a listed workout on the begin workouts screen	• Displays a list of exercises in the pressed workout, with their target sets and reps	• Passed		

Press the start button on the perform workout screen	• Begins the timer displayed at the top of the screen	• Passed		
Press the pause button on the perform workout screen	• Pauses the timer displayed at the top of the screen	• Passed		
Press the reset button on the perform workout screen	• Resets the timer to 00:00 displayed at the top of the screen	• Passed		
Press the cancel button on the perform workout screen	• Deletes the performed workout history SQL table • Navigates back to the fitness home screen	• Passed • Passed		
Press the green tick icon on invalid reps and weight data input on the perform workout screen	• Error message informing user of invalid data input	• Passed		
Press the green tick icon on valid reps and weight data input on the perform workout screen	• Alert message informing the user that the exercise performed has been logged	• Passed		
Press the green tick icon on valid reps and weight data input on	• Performed exercises from the workout listed on screen	• Passed		

the perform workout screen **Press the finish button on the perform workout screen**				
Press the green tick icon on valid reps and weight data input on the perform workout screen **Press the finish button on the perform workout screen** **Press the delete icon on listed exercises on the completed workout overview screen**	• Exercise deleted from the list of performed exercises • Exercise deleted from performed workout SQL table	• Passed • Passed		
Press the green tick icon on valid reps and weight data input on the perform workout screen **Press the finish button on the perform workout screen** **Press the submit button on the**	• Performed workout SQLite table logging the exercises, weights and reps performed • Navigate to fitness home screen	• Passed • Passed		

completed workout overview screen				
Press the green tick icon on valid reps and weight data input on the perform workout screen **Press the finish button on the perform workout screen** **Press the submit button on the completed workout overview screen** **Navigate to the view workout progress screen**	• Performed workout history with correctly calculated metrics displayed on all graphs	• Passed		
Press the green tick icon on valid reps and weight data input on Shoulder Press on the perform workout screen **Press the finish button on the perform workout screen** **Press the submit button on the**	• Performed shoulder press exercise history with correctly calculated metrics displayed on graph • Shoulder press max weight lifted from performed workout displayed at the top of the screen.	• Passed • Passed		

completed workout overview screen **Navigate to the view exercise progress screen → press view progress on Shoulder Press on the listed shoulder exercises screen**				

9.2 Neural Network Test Runs and Results

9.3 Libraries List

Package, Modules and Dependencies List:
react
react-native
react-native-firebase
react-native-sqlite-storage
react-navigation
react-native-paper
react-native-vector-icons
react-native-flash-message
react-native-chart-kit

react-native-picker-select
react-native svg
react-native-picker-select
react-native-pie-chart
react-native-onboarding-swiper
brain.js
fs
@react-native-async-storage/async-storage
@react-native-community/art
@react-native-community/progress-view
@react-native-firebase/app
@react-native-firebase/auth
@react-native-firebase/firestore
@react-navigation/bottom-tabs
@react-navigation/material-bottom-tabs
@react-navigation/native
@react-navigation/stack

@react-native-vector-icons/MaterialCommunityIcons

9.4 User Manual

9.4.1 Setup:

1. Setup the development environment for react-native with Android Studio following the guide on their website: https://reactnative.dev/docs/environment-setup
2. Download the project folder "Strategym" onto your desktop machine
3. Open up terminal → cd to the Strategym directory → input the command:
    a. npx react-native start
4. Open up terminal again → cd to the Strategym directory → input the command:
    a. npx react-native run-android
5. ***Or: run an emulator in Android Studio: https://developer.android.com/studio/run/emulator → locate the app-release.apk file in the Strategym folder → drag and drop the file onto the emulator screen to install it onto the emulator.***

9.4.2 Guide:

9.4.2.1 Onboarding Screens:

1. When launching the application, the initial screens you will reach are the onboarding screens.
2. Press the next icon at the bottom right corner of the onboarding screens until you reach the login/registration screen.

9.4.2.2 Login/Registration Screens

Test User Login:

- ***Email = testing@test.com***
- ***Password = test1234***

1. Login either using the test user's details provided or sign up on the registration screen with your own details (swipe left to access the sign up form)
2. Once the details are entered, press the submit button. You will be navigated to the main application.

9.4.2.3 Profile Section Guide

1. To access the profile section, press the profile icon at the right of the bottom navigator
2. To update your profile and authentication data, press on any of the following tiles:
    a. Profile Settings
    b. Bodyfat Percentage Calculator
    c. TDEE Calculation

    d. Dietary Goal Form
    e. Fitness Goal Form
3. Alternatively, if you have signed up as a new user, you can navigate to the My Statistics tab on the profile home screen by swiping left → then press on the listed texts to access the forms to update your profile
4. Once you have updated your profile, navigate back to the profile home screen by using the back button present in the header → navigate to the My Statistics tab by swiping left → here an analysis will be displayed presenting you with your health and fitness metrics corresponding to your profile
5. In order to logout of the application: swipe right to access the Menu tab and press the Logout tile

9.4.2.4 Diet Section Guide

This guide will take you through a full tutorial on how to use the diet section of the application. Instructions are set out from the diet home screen being the start screen.

1. To access the diet section, press the diet icon (the middle icon in the bottom navigation bar). This is the diet home screen
2. To add a food item to your today's intake: press the Add Foods tile → enter an amount of grams for a food item you want to add → press the green basket icon
3. To create a food item: press the Add Foods tile → press the circular Create button → enter the details of the food item you want to add → press the submit button
4. To access your created foods: press the Add Foods tile → press the circular Saved button
5. To add a food item you created to your today's intake: press the Add Foods tile → press the circular Saved button → enter an amount of grams for a food item you want to add → press the green basket icon
6. To delete a food item you created: press the Add foods tile → press the circular Saved button → press the red bin icon next to the food item you want to delete
7. To view food items you have added to your today's intake: follow '2.' Or '5.' → navigate back to the diet home screen by pressing the back icon on the top left of the screen → your food items for today's intake is displayed at the top of the diet home screen.
8. To delete food items you have added to your today's intake: press the red bin icon next to the food item you want to delete on the diet home screen
9. To view your calories/macronutrients ingested today against your targets as displayed on a pie chart: press the Today's Totals tile on the diet home screen
10. To track the progress of your daily caloric/macronutrient history: press the View Progress tile on the diet home screen
11. To save your intake for today, press the Save Today's Intake tile on the diet home screen

9.4.2.5 Fitness Section Guide

This guide will take you through a full tutorial on how to use the fitness section of the application. Instructions are set out from the fitness home screen being the start screen.

1. To access the fitness section, press the fitness icon (the left icon in the bottom navigation bar). This is the fitness home screen
2. To create a workout; create exercises; delete exercises from the created workout; and save the created workout (full walkthrough):
    a. Press the Create Workout tile → enter a name for your workout and press Search Exercises → press on one of the images displaying muscle groups → Enter a target number of sets and reps for that exercise to include in your workout → navigate back by pressing the back arrow at the top left of the screen → press the circular create button to create an exercise → fill in the form and press the submit button → press the circular save button → press the red bin icon next to a listed exercise to delete it from the workout → press the save button to save the workout you have created
3. To view your created workouts: press the Begin Workouts tile → swipe right to navigate to the My Workouts tab if not on that tab already
4. To delete your created workouts: press the Begin Workouts tile → swipe right to navigate to the My Workouts tab → press the delete button next to the workout name you would like to delete
5. To view your generated workout plan: press the Begin Workouts tile → swipe left to navigate to the Generated Plan tab
6. To perform a workout; set the timer for the workout; and add sets to the performed workout; and save the performed workout:
    a. Press the Begin Workouts tile → press the Perform button on one of the workouts listed → press the start timer to start the timer for the workout → press the pause button to pause the timer → press the reset button to reset the timer → enter a number of reps and weight for an exercise in the list and press the green tick button to add it to the workout (each time you press the green button it is another set added to the performed workout) → press the Finish circular button → press the red bin icon next to an exercise listed to delete it from the performed workout → press the Submit button to save the performed workout data.
7. To view the workout progress of all the workouts you have performed in the past against your target goals: press the View Workout Progress tile on the fitness home screen
8. To view the progress of a specific exercise you have performed: press the View Exercise Progress tile on the fitness home screen → press the muscle group which the exercise corresponds to → press the View Progress button next to the exercise name listed

9.5 Program Listings

This program listing lists all the files produced during the development of this project and their associated directories. All files include comments in-code guiding the user on what the program is used for, and comments above each function explaining the role of the function.

- root folder → App.js
- root folder → index.js
- src → navigation folder

    - appStack.js
    - authProvider.js
    - authStack.js
    - index.js
    - routes.js
- src → screens → dietScreens folder
    - addFoods.js
    - createFoodScreen.js
    - dailyIntakeProgressScreen.js
    - dietHomeScreen.js
    - nutritionalDatabase.json
    - savedFoodsScreen.js
    - todaysTotals.js
- src → screens → fitnessScreens folder
    - abs.js
    - absProgress.js
    - back.js
    - backProgress.js
    - beginWorkout.js
    - biceps.js
    - bicepProgress.js
    - cardio.js
    - cardioProgress.js
    - chest.js
    - chestProgress.js
    - completedGeneratedWOverview.js
    - completedWorkoutOverview.js
    - createdWorkoutOverview.js
    - createExercises.js
    - createWorkout.js
    - displayExProgress.js
    - exerciseProgress.js
    - exercisesDatabase.json

    - fitnessHomeScreen.js
    - legs.js
    - legsProgress.js
    - performGeneratedWorkout.js
    - performWorkout.js
    - searchExercises.js
    - shoulders.js
    - shouldersProgress.js
    - triceps.js
    - tricepsProgress.js
    - viewProgress.js
- src → screens → fitnessScreens → workoutPlans folder
    - endurancePlans.js
    - hypertrophyPlans.js
    - strengthPlans.js
- src → screens → globalComponent folder
    - customerHeaderWithBack.js
    - customHeaderComponent.js
- src → screens → loginRegistrationScreens folder
    - loginRegistrationScreen.js
- src → screens → loginRegistrationScreens → components folder
    - containerForLoginForm.js
    - loginForm.js
    - loginFormInput.js
    - loginRegBtns.js
    - loginRegFormHeader.js
    - registrationForm.js
    - submitLogin.js
- src → screens → onboardingScreens
    - onboardingScreen.js
- src → screens → profileScreens folder
    - dietaryGoalForm.js
    - femaleTrainedNN.json

    - fitnessGoalForm.js
    - healthFitnessMetrics.js
    - maleTrainedNNs.json
    - mensBFStandardisedData.json
    - profileHomeScreen.js
    - profileSettingsScreen.js
    - tdeeCalcForm.js
    - womenBFStandardisedData.json
- src → screens → profileScreens → components folder
    - containerForProfileTabs.js
    - menuBtnsStyling.js
    - saveSettingsbtn.js
    - settingsFormInput.js
    - topTabBtns.js
- root → neural networks folder
    - FemaleBFTrainingNN.js
    - MaleBFTrainingNN.js
    - mensBFStandardisedData.json
    - womenBFStandardisedData.json

9.6 Other Essential Files Locations:

This files listing lists all the other files produced (along with their directories) that are not code-based.

- root folder → neural networks → Original Datasets → Data Cleansed Standardised Datasets folder
    - Mens BF Standardised Dataset.xlsx
    - Womens BF Standardised Dataset.xlsx
- root folder → sqlite dummy data folder
    - foodsIntakeHistory.db
    - userExercises.db
    - userFoods.db
    - userWorkoutHistory.db
    - userWorkouts.db

10 References

ACE Fit (2022). Percent Body Fat Calculator: Skinfold Method. [online] American Council on Exercise. Available at: https://www.acefitness.org/resources/everyone/tools-calculators/percent-body-fat-calculator/?irclickid=UkpV55z3TxyIWx9x9uVMuSBVUkBSibS8tzZIwA0&irgwc=1&utm_source=Affiliate&utm_campaign=12960&clickid=UkpV55z3TxyIWx9x9uVMuSBVUkBSibS8tzZIwA0&utm_content=Online%20Tracking%20Link_984595&utm_medium=Impact&utm_channel=Affiliate_Marketing.

Adelakun-Adeyemo, O. and Olalekan, S. (2015). Issues in Native Mobile Application Programming. In: ACM International Conference on Computer Science Research and Innovations. Nigeria: University of Ibadan.

Albarka Umar, M. (2020). (PDF) Comprehensive study of software testing: Categories, levels, techniques, and types. [online] ResearchGate. Available at: https://www.researchgate.net/publication/342538504_Comprehensive_study_of_software_testing_Categories_levels_techniques_and_types.

Alexcvzz (2022). SQLite - Visual Studio Marketplace. [online] marketplace.visualstudio.com. Available at: https://marketplace.visualstudio.com/items?itemName=alexcvzz.vscode-sqlite.

Anderson, T. and Kearney, J.T. (1982). Effects of Three Resistance Training Programs on Muscular Strength And Absolute and Relative Endurance. Research Quarterly for Exercise and Sport, 53(1), pp.1–7.

Andpor (2022). react-native-sqlite-storage. [online] Node Package Manager. Available at: https://www.npmjs.com/package/react-native-sqlite-storage [Accessed 6 May 2022].

Anon, (2006). [online] Available at: https://www.acefitness.org/getfit/GlutesStudy2006.pdf [Accessed 1 May 2022].

Apple (2019). Swift - Apple Developer. [online] Apple.com. Available at: https://developer.apple.com/swift/.

Arvidsson, J. (2022). Multi-style fonts. [online] GitHub. Available at: https://github.com/oblador/react-native-vector-icons [Accessed 6 May 2022].

Berger, R.A. (1962). Optimum Repetitions for the Development of Strength. Research Quarterly. American Association for Health, Physical Education and Recreation, 33(3), pp.334–338.

Biswas, N. (2019). *The Web Dev*. [online] thewebdev.tech. Available at: https://thewebdev.tech/reactnative-simple-timer-app [Accessed 10 May 2022].

Boehler, B., Porcari, J., Kline, D., Hendrix, C., Foster, C. and Andrews, M. (2018). Terrific Triceps [online] ACE Fitness. Available at: https://acewebcontent.azureedge.net/certifiednews/images/article/pdfs/ACETricepsStudy.pdf [Accessed 1 May 2022].

Bonnington, C., 2018. The MyFitnessPal Hack Affects 150 Million Users. It Could've Been Even Worse.. [online] Slate Magazine. Available at: <https://slate.com/technology/2018/03/myfitnesspal-hack-under-armour-data-breach.html> [Accessed 29 April 2022].

bricelam (2021). Encryption - Microsoft.Data.Sqlite. [online] docs.microsoft.com. Available at: https://docs.microsoft.com/en-us/dotnet/standard/data/sqlite/encryption?tabs=netcore-cli [Accessed 2 May 2022].

Buchholz, K. (2020). Infographic: Apple or Android Nation? Operating System Popularity Across Countries. [online] Statista Infographics. Available at: https://www.statista.com/chart/22702/andoid-ios-market-share-selected-countries/.

Callstack (2022). react-native-paper. [online] npm. Available at: https://www.npmjs.com/package/react-native-paper [Accessed 6 May 2022].

Carneiro, J. (2018). Progressive Web Apps: Concepts and Features. [online] kriativ.tech, pp.1–4. Available at: http://www.kriativ-tech.com/wp-content/uploads/2018/02/JCarneiro-ProgressiveWebApps_en.pdf [Accessed 8 Mar. 2022].

Dahl, R. (2022). File system | Node.js v18.1.0 Documentation. [online] nodejs.org. Available at: https://nodejs.org/api/fs.html [Accessed 6 May 2022].

DeBill, E. (2022). Modulecounts. [online] www.modulecounts.com. Available at: http://www.modulecounts.com/.

Edelburg, H., Porcari, J., Camic, C., Kovacs, A., Foster, C. and Green, D. (2018). What Is the Best Back Exercise? [online] ACE Fitness. Available at: https://acefitnessmediastorage.blob.core.windows.net/webcontent/April2018/ACE_BackExerStudy.pdf [Accessed 1 May 2022].

Etchison, W.C., Bloodgood, E.A., Minton, C.P., Thompson, N.J., Collins, M.A., Hunter, S.C. and Dai, H. (2011). Body Mass Index and Percentage of Body Fat as Indicators for Obesity in an Adolescent Athletic Population. Sports Health: A Multidisciplinary Approach, [online] 3(3), pp.249–252. doi:10.1177/1941738111404655.

Expo (2022). Limitations. [online] Expo Documentation. Available at: https://docs.expo.dev/introduction/why-not-expo/ [Accessed 6 May 2022].

Expo Cli (2022). Running in the Browser. [online] Expo Documentation. Available at: https://docs.expo.dev/guides/running-in-the-browser/ [Accessed 2 May 2022].

Ferreira, L. (2022). react-native-flash-message. [online] GitHub. Available at: https://github.com/lucasferreira/react-native-flash-message [Accessed 6 May 2022].

Flutter (2019). Flutter - Beautiful native apps in record time. [online] Flutter.dev. Available at: https://flutter.dev/.

Flutter (2022). Flutter on the Web. [online] flutter.dev. Available at: https://flutter.dev/multi-platform/web [Accessed 2 May 2022].

Flutter (2022). Using packages. [online] docs.flutter.dev. Available at: https://docs.flutter.dev/development/packages-and-plugins/using-packages [Accessed 2 May 2022].

Frankenfield, D., Roth-Yousey, L. and Compher, C., 2005. Comparison of Predictive Equations For Resting Metabolic Rate in Healthy, Nonobease and Obese Adults: A Systematic Review. American Dietetic Association, [online] pp.775 - 789. Available at: <https://www.andeal.org/files/Docs/Frankenfield_et_al_2005%5B1%5D.pdf> [Accessed 30 April 2022].

Google (2019). Server-side encryption | Cloud Firestore | Google Cloud. [online] Google Cloud. Available at: https://cloud.google.com/firestore/docs/server-side-encryption.

Google (2022). Access data offline. [online] Firebase. Available at: https://firebase.google.com/docs/firestore/manage-data/enable-offline.

Google (2022). Access data offline. [online] Firebase. Available at: https://firebase.google.com/docs/firestore/manage-data/enable-offline.

Google (2022). Admin Authentication API Errors | Firebase Documentation. [online] Firebase. Available at: https://firebase.google.com/docs/auth/admin/errors [Accessed 8 May 2022].

Google (2022). Building web apps in WebView. [online] Android Developers. Available at: https://developer.android.com/guide/webapps/webview.

Google (2022). Cloud Firestore | Store and sync app data at global scale. [online] Firebase. Available at: https://firebase.google.com/products/firestore#:~:text=With%20Cloud%20Firestore%2C%20you%20can [Accessed 2 May 2022].

Google (2022). Sync, async, and promises | Firebase Documentation. [online] Firebase. Available at: https://firebase.google.com/docs/functions/terminate-functions [Accessed 3 May 2022].

Google (2022d). Firebase Pricing | Firebase. [online] Firebase. Available at: https://firebase.google.com/pricing.

Google Developers (2019). Meet Android Studio | Android Developers. [online] Android Developers. Available at: https://developer.android.com/studio/intro.

Grgic, J., Schoenfeld, B.J., Davies, T.B., Lazinica, B., Krieger, J.W. and Pedisic, Z. (2018). Effect of Resistance Training Frequency on Gains in Muscular Strength: A Systematic Review and Meta-Analysis. Sports Medicine, 48(5), pp.1207–1220.

HUMANIZE, 2018. Personalizing Mobile Fitness Apps using Reinforcement Learning. [online] Tokyo: HUMANIZE. Available at: <http://ceur-ws.org/Vol-2068/humanize7.pdf> [Accessed 29 April 2022].

Inc., M. (2022). Platform Specific Code · React Native. [online] reactnative.dev. Available at: https://reactnative.dev/docs/platform-specific-code [Accessed 2 May 2022].

Institute for Government (2021). Timeline of UK Government Coronavirus Lockdowns. [online] www.instituteforgovernment.org.uk. Available at: https://www.instituteforgovernment.org.uk/charts/uk-government-coronavirus-lockdowns.

Ionic (2022). Progressive Web App (PWA) Mobile Development. [online] Ionic. Available at: https://ionic.io/pwa [Accessed 2 May 2022].

Johannes Filter (2022). GitHub - jfilter/react-native-onboarding-swiper: Delightful onboarding for your React-Native app. [online] GitHub. Available at: https://github.com/jfilter/react-native-onboarding-swiper [Accessed 6 May 2022].

Johnson, R.W. (1996). Fitting Percentage of Body Fat to Simple Body Measurements. Journal of Statistics Education, 4(1). doi:10.1080/10691898.1996.11910505.

Johnson, R.W. (2021). Fitting Percentage of Body Fat to Simple Body Measurements: College Women. Journal of Statistics and Data Science Education, 29(3).

Joseph, S.A. (2020). User's Perspective About Mobile Fitness Applications. International Journal of Recent Technology and Engineering, 8(6), pp.3368–3373.

Khandeparkar, A., Gupta, R. and B.Sindhya, B.Sindhya. (2015). An Introduction to Hybrid Platform Mobile Application Development. International Journal of Computer Applications, 118(15), pp.31–33.

Kvist, J. and Mathiasson, P. (2019). Progressive Web Apps and other mobile developing techniques: a comparison Progressiva Webbappar och andra mobila utvecklingstekniker: en jämförelse. [online] Available at: https://www.diva-portal.org/smash/get/diva2:1480326/FULLTEXT01.pdf.

Kyeremeh, K. (2019). Overview of System Development Life Cycle Models. SSRN Electronic Journal. doi:10.2139/ssrn.3448536.

Lambert, C., Frank, L. and Evans, W., 2004. Macronutrient Considerations for the Sport of Bodybuilding. Sports Medicine, 34(5), pp.317-327.

LawnStarter Organisation (2022). react-native-picker-select. [online] GitHub. Available at: https://github.com/lawnstarter/react-native-picker-select [Accessed 6 May 2022].

Leijdekkers, R., 2015. The Use of Privacy Data by Fitness Applications,. [online] Essay.utwente.nl. Available at: <https://essay.utwente.nl/67332/1/Leijdekkers.BA.MB.pdf> [Accessed 29 April 2022].

lion, J. the (2015). What Are the Best Calf Exercises? [online] GymLion. Available at: http://gymlion.com/what-are-the-best-calf-exercises/ [Accessed 1 May 2022].

lion, J. the (2015). What Are the Best Quad Exercises? [online] GymLion. Available at: http://gymlion.com/what-are-the-best-quad-exercises/ [Accessed 1 May 2022].

Luhanga, E., Hippocrate A. AKPA, E., Suwa, H. and Arakawa, Y., 2018. Identifying and Evaluating User Requirements for Smartphone Group Fitness Applications. [online] Available at: <https://www.researchgate.net/publication/322513103_Identifying_and_Evaluating_User_Requirements_for_Smartphone_Group_Fitness_Applications> [Accessed 29 April 2022].

Masaad Alsaid, M.A.M., Ahmed, T.M., Jan, S., Khan, F.Q., Mohammad and Khattak, A.U. (2021). A Comparative Analysis of Mobile Application Development Approaches. Proceedings of the Pakistan Academy of Sciences: A. Physical and Computational Sciences, 58(1), pp.35–45.

Mehta, A. (2022). Top 15 Databases to Use in 2021 and Beyond. [online] Appinventiv. Available at: https://appinventiv.com/blog/top-web-app-database-list/.

Meta Inc (2022). Navigating Between Screens · React Native. [online] reactnative.dev. Available at: https://reactnative.dev/docs/navigation.

Meta Inc (2022). Setting up the development environment · React Native. [online] reactnative.dev. Available at: https://reactnative.dev/docs/environment-setup.

Microsoft (2022). React Native Tools - Visual Studio Marketplace. [online] marketplace.visualstudio.com. Available at: https://marketplace.visualstudio.com/items?itemName=msjsdiag.vscode-react-native [Accessed 6 May 2022].

Microsoft (2022). Xamarin | Open-source mobile app platform for .NET. [online] Microsoft. Available at: https://dotnet.microsoft.com/en-us/apps/xamarin.

MongoDB (2022). Encrypt a Realm - Swift SDK — MongoDB Realm. [online] www.mongodb.com. Available at: https://www.mongodb.com/docs/realm/sdk/swift/advanced-guides/encrypt-a-realm/#:~:text=Realm%20transparently%20encrypts%20and%20decrypts.

MongoDB (2022). Offline first. [online] www.mongodb.com. Available at: https://www.mongodb.com/docs/realm-legacy/solutions/offline-first.html [Accessed 2 May 2022].

MongoDB (2022). Realm Pricing. [online] www.mongodb.com. Available at: https://www.mongodb.com/docs/realm-legacy/pricing.html [Accessed 2 May 2022].

MSEdgeTeam (2022). Overview of Progressive Web Apps (PWAs) - Microsoft Edge Development. [online] docs.microsoft.com. Available at: https://docs.microsoft.com/en-us/microsoft-edge/progressive-web-apps-chromium/.

Muoio, D., 2020. Under Armour sells off MyFitnessPal for $345M, will shut down Endomondo by 2021. [online] MobiHealthNews. Available at: <https://www.mobihealthnews.com/news/under-armour-sells-myfitnesspal-345m-will-shut-down-endomondo-2021#:~:text=Under%20Armour%20announced%20late%20last,and%20platform%20back%20in%202015.> [Accessed 29 April 2022].

National Agricultural Library. 2022. How many calories are in one gram of fat, carbohydrate, or protein?. [online] Available at: <https://www.nal.usda.gov/legacy/fnic/how-many-calories-are-one-gram-fat-carbohydrate-or-protein> [Accessed 30 April 2022].

National Agricultural Library. 2022. How many calories are in one gram of fat, carbohydrate, or protein?. [online] Available at: <https://www.nal.usda.gov/legacy/fnic/how-many-calories-are-one-gram-fat-carbohydrate-or-protein> [Accessed 30 April 2022].

NHS. 2022. Physical activity guidelines for adults aged 19 to 64. [online] Available at: <https://www.nhs.uk/live-well/exercise/exercise-guidelines/physical-activity-guidelines-for-adults-aged-19-to-64/> [Accessed 1 May 2022].

Open Source Collective (2022). brain.js. [online] GitHub. Available at: https://github.com/BrainJS [Accessed 6 May 2022].

Prettier (2022). Prettier - Code formatter - Visual Studio Marketplace. [online] marketplace.visualstudio.com. Available at: https://marketplace.visualstudio.com/items?itemName=esbenp.prettier-vscode.

Ralston, G.W., Kilgore, L., Wyatt, F.B. and Baker, J.S. (2017). The Effect of Weekly Set Volume on Strength Gain: A Meta-Analysis. Sports Medicine, 47(12), pp.2585–2601.

Rastogi, V. (2015). Software Development Life Cycle Models- Comparison, Consequences. [online] Available at: http://www.ijcsit.com/docs/Volume%206/vol6issue01/ijcsit2015060137.pdf [Accessed 4 May 2022].

Schanke, W., Porcari, J., Felix, E., Hendrix, C. and Foster, C. (2012). Top 3 Most Effective Chest Exercises ACE-Sponsored Research. [online] Available at: https://acewebcontent.azureedge.net/certifiednews/images/article/pdfs/ACE_BestChestExercises.pdf.

Schoenfeld, B.J. (2010). The mechanisms of muscle hypertrophy and their application to resistance training. Journal of strength and conditioning research, [online] 24(10), pp.2857–72. Available at: https://journals.lww.com/nsca-jscr/fulltext/2010/10000/the_mechanisms_of_muscle_hypertrophy_and_their.40.aspx.

Schoenfeld, B.J., Ogborn, D. and Krieger, J.W. (2016). Effects of Resistance Training Frequency on Measures of Muscle Hypertrophy: A Systematic Review and Meta-Analysis. Sports Medicine, [online] 46(11), pp.1689–1697. Available at: https://pubmed.ncbi.nlm.nih.gov/27102172/.

Schoenfeld, B.J., Ogborn, D. and Krieger, J.W. (2017). Dose-response relationship between weekly resistance training volume and increases in muscle mass: A systematic review and meta-analysis. Journal of sports sciences, [online] 35(11), pp.1073–1082. Available at: https://www.ncbi.nlm.nih.gov/pubmed/27433992.

Shen, P., 2022. Analysis of User Needs of Sports and Fitness Apps from the Perspective of “Healthy China”. Academic Journal of Business & Management, [online] Vol. 2(Issue 5), pp.82-89. Available at: <https://francis-press.com/uploads/papers/Fi2azziVgOSRPT2yorYynruMAhvLI7cPKF6B8ZMd.pdf> [Accessed 29 April 2022].

Solbrig, L. et al. (2017) “People trying to lose weight dislike calorie counting apps and want motivational support to help them achieve their goals,” Internet interventions, 7(C), pp. 23–31. doi: 10.1016/j.invent.2016.12.003.

SQLite (2022). SQLite Copyright. [online] www.sqlite.org. Available at: https://www.sqlite.org/copyright.html#:~:text=SQLite%20is%20open%2Dsource%2C%20meaning [Accessed 2 May 2022].

SQLite (2022). SQLite Database Speed Comparison. [online] www.sqlite.org. Available at: https://www.sqlite.org/speed.html [Accessed 2 May 2022].

SQLite (2022). SQLite: Documentation. [online] www.sqlite.org. Available at: https://www.sqlite.org/src/doc/trunk/ext/userauth/user-auth.txt [Accessed 2 May 2022].

Statista. 2022. Fitness Apps - United Kingdom | Statista Market Forecast. [online] Available at: <https://www.statista.com/outlook/dmo/digital-health/digital-fitness-well-being/digital-fitness-well-being-apps/fitness-apps/united-kingdom?currency=GBP> [Accessed 14 March 2022].

Statista. 2022. Great Britain: top Android health apps by revenue 2022 | Statista. [online] Available at: <https://www.statista.com/statistics/699346/leading-android-health-apps-in-great-britain-by-revenue/> [Accessed 29 April 2022].

Sujay Vailshery, L. (2021). Cross-platform mobile frameworks used by global developers 2020. [online] Statista. Available at: https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/.

SuppVersity EMG Series - Rectus Abdominis, Obliques and Erector Spinae. (2011). SuppVersity EMG Series - Rectus Abdominis, Obliques and Erector Spinae: The Very Best Exercises For Sixpack Abs and a Powerful Midsection - SuppVersity: Nutrition and Exercise Science for Everyone. [online] Available at: https://suppversity.blogspot.com/2011/07/suppversity-emg-series-rectus-abdominis.html [Accessed 1 May 2022].

Sweeney, S., Porcari, J., Camic, C., Kovacs, A. and Foster, C., 2014. Shoulders. Above The Rest?. [online] ACE SHOULDER STUDY. Available at: <https://acewebcontent.azureedge.net/certifiednews/images/article/pdfs/ACEShoulderStudy.pdf> [Accessed 1 May 2022].

Tawfiq, F. (2020). (PDF) SDLC (system development life cycle). [online] ResearchGate. Available at: https://www.researchgate.net/publication/341883828_SDLC_system_development_life_cycle.

Tripathy, S. and Saha, D., 2022. Essentials of Evidence-Based Practice of Neuroanesthesia and Neurocritical Care. Academic Press, pp.375 - 387.

TutorialRepublic (2022). How to Find the Sum of an Array of Numbers in JavaScript. [online] www.tutorialrepublic.com. Available at: https://www.tutorialrepublic.com/faq/how-to-find-the-sum-of-an-array-of-numbers-in-javascript.php.

U.S. Department of Agriculture. 2021. FoodData Central. [online] Available at: <https://fdc.nal.usda.gov/download-datasets.html> [Accessed 1 May 2022].

U.S. Department of Health & Human Services (2013). Wireframes Design Wireframes. [online] U.S. Department of Health & Human Services. Available at: https://www.usability.gov/sites/default/files/creating-wireframes.pdf.

UCL Interaction Centre, 2018. "Keep Going!": Understanding the Implications of Coaching through Fitness Apps to Support Physical Training. [online] London: University College London, pp.1-15. Available at: <https://uclic.ucl.ac.uk/content/2-study/4-current-taught-course/1-distinction-projects/12-18/stoica_ralucaalexandra_2018.pdf> [Accessed 29 April 2022].

UK Government, 2009. Statement of the Calorie Reduction Expert Group1. pp.1-11. [online] Available at: < https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/215561/dh_127554.pdf>

Xu, G. (2022). react-native-pie-chart. [online] GitHub. Available at: https://github.com/genexu/react-native-pie-chart [Accessed 6 May 2022].

Young, S., Porcari, J., Camic, C., Kovacs, A. and Foster, C. (2014). ACE PROSOURCE | August 2014 ACE STUDY REVEALS BEST BICEPS EXERCISES EXCLUSIVE ACE-SPONSORED RESEARCH. [online] Available at: https://acewebcontent.azureedge.net/certifiednews/images/article/pdfs/ACE%20BicepsStudy.pdf