CMP-5012B Software Engineering II

Group Project Stage 2 – Report

https://bitbucket.org/zmf18gwu/cmp-5012b-part1/src/master/

Design and implementation of a health tracker application.

Group Number: 12

Group Members: Jonathan (Buzz) Embley-Riches 100237137

Linfeng (Chris) Chen 100254216

Ranhui (Harris) Zhang 100256949

James Burrell 100263300

Date: Last edited 10/06/2020

1 Stage 1 Feedback Address

1.1 Feedback Table

Feedback label	Feedback statement	Feedback address	Update
F1.1	*Explain how features in the table in section 1.1. translate to features in the similar systems matrix in section 1.3.*	Added a brief explanation how we produced the similar systems matrix from the feature table.	**1.2.1 F1.1**
F2.1	*Redirect alternative scenarios to the step in the successful scenario where they will return to following failure.*	Added redirects to appropriate locations in use case.	**1.2.2 F2.1**
F2.2	*You could write an alternative version of user stories by using your personas.*	Examples of user stories tailored to our personas added.	**1.2.3 F2.2**
F3.1	*You derived the classes from NLM on user stories. How did you decide what should be a class and what should be a class attribute. Also – in the OOA class diagram you have the following relationships: Association from Admin to User – should this not be inheritance? The composite relationships between User and other 'tracker' classes seems plausible but you should explain why these relationships are indeed composite.*	Explanation about class and class attribute decisions added. New OOA class diagram has updated relationships. Admin class is simply a utility class and not a user with control, however this wasn't clear in that section of the Stage 1 report. Explanation into composite relationship between User and trackers added.	**1.2.4 F3.1** **OOA diagram at 2.5.1**
F4.1	*Update relationship between User and AdminClass, the former being aggregate to the latter. In addition, the Email class should be connected to related classes.*	Updated OOA class diagram with corrected relationships and gave Email class related classes.	**OOA diagram at 2.5.1**
F4.2	*Enlarge class relationship symbols in model class diagram.*	We tried our best to find a way of enlarging them, but hopefully the neatened layout makes them easier to identify.	**OOA diagram at 2.5.1**
F4.3	*Various class relationships in the M part of the MVC are different to those in the model class diagram. You should make these two consistent.*	The relationships should now be consistent.	**MVC diagram at 2.5.2**

F4.4	*You should establish all C-V, C-M and M-Db relationships. Put each of these three in different diagrams.*	Separated into 3 diagrams.	**MVC diagram at 2.5.2**
F5.1	*This might be just a demo thing but make sure you use the same language across the GUI.*	Videos were recorded with consistent languages.	**See demo video.**
F5.2	*Labels (e.g. apple, milk, steak) are hard to see on bar plot.*	Updates to UI should hopefully address this.	**See A.2 in appendix.**
F5.3	*Perhaps it would be better to organise the source code in three M,V and C folders to make the MVC architecture more clear.*	Classes added to folders in source code.	**See source code.**
F5.4	*Do not set bounds (e.g. BMI in User.java) directly in the code. These value ranges should be stored in the database (or xml, csv file) with their corresponding label, i.e. 18.5, 24.9, Normal, etc.*	Added values to the database and are now pulled from it.	**See A.3 in appendix.**
F5.5	*Ditto in methods such as saveEditWeight, Height etc. the valid ranges should be derived from a database (or xml, csv file) and not hardcoded in the programming code.*	Addes values to the database and are now pulled + pushed to it.	**See A.4 in appendix.**
F6.1	*Put multiple screenshots in appendix instead of text body.*	Multiple screenshots now appear in appendix with labels.	**See appendix.**
F6.2	*Check for any wrong section numbering (e.g., 1.1 -> 1.2) and inconsistent font use (e.g., 'Analysis of similar system' part).*	Labelling double checked in this report and fixed in part 1.	**-**
F7.1	*For Stage 2, use a single video demo as it will improve consistency and clarity (i.e., not School of Computing Sciences R.J. Lapeer, 06/04/2020 multiple video clips). Don't forget to mention or display the name of speaker of a certain part.*	Video was edited into one, with labels indicating speaker shown.	**See demo video.**
F7.2	*The briefing document requests to demo the unit test – include this for the Stage 2 video.*	Demo included unit test by Chris.	**See demo video.**

1.2 Addressing stage 1 feedback.

The following sections contain responses to the feedback we received for the first stage of our application development. Only feedback points that can be appropriately addressed in this section will be presented here, ie. Parts 1 and 2 from Stage 1, the rest will be shown in more relevant sections, and can be found at the locations in the above table.

1.2.1 F1.1 Response

In the first stage of analysing similar systems, 1.1, we chose 3 popular fitness apps and identified every distinct feature from each one and noted them down. After we subdivided this list into 3 sections, choosing the features we believed are most relevant for our app, and other features that would also be possible but not essential.

After this was done, in section 1.3 we reduced this list further by choosing features we thought were the most important in terms of the final product, irrespective of if they appeared in every app. The bullet points in the Key Features column in 1.1 were grouped together into more generic terms, for example with MyFitnessPal, progress reports and exercise logging were combined under the feature Exercise Tracker in the feature matrix. These generic terms helped to compare the 3 applications, and in doing so decide which features would be included in our app.

1.2.2 F2.1 Response

USE CASE NAME	**User Account Create**	
Goal in Context	To allow a user to create an account, which will allow them to enter personal information and have a place to login and securely access all the features of the app.	
Scope & Level	Create an account system.	
Preconditions	System is running and the user account file is loaded.	
Success End Condition	A new user account is created.	
Failed End Condition	Invalid account creation.	
Primary Actor	User	
Trigger	Actor selecting “Sign up”.	
SUCCESS SCENARIO	**Step**	**Action**
	1	The system displays a window requesting log in or sign up.
	2	The actor selects the sign up option.
	3	The actor enters an email and password they would like to use for their account.

	4	The system verifies that the email is the correct format and passwords match.
	5	The actor enters all their personal information requested from the system.
	6	The system verifies the entered information is valid.
	7	The system stores the information and confirms, then displays the main window
ALTERNATIVE SCENARIO	**Step**	**Branching action**
	4a	If the information is invalid, the system displays an error message, and a helpful tip. The user is then redirected back to step 3.
	6a	If the information is invalid, the system displays an error message, and a helpful tip. The user is then redirected back to step 5.
RELATED INFORMATION		
Priority	Top priority	
Performance Target	-	
Frequency	Infrequent	
Subordinate Use Cases	-	
Channel to Primary Actor	User interface.	
Secondary Actors	-	
Channel to Secondary Actors	-	
OPEN ISSUES	Verifying that the email is live and valid?	
SCHEDULE	Due date is version 1.0 release	
AUTHOR	Linfeng Chen, Ranhui Zhang, James Burrell, Buzz Embley-Riches.	

1.2.3 F2.2 Response

Below are examples of user stories produced in the context of how they would be used by our personas. One user story has been given for each of the characters in Stage 1 report section 2.3.

Goal capture	#4	HIGH
As Emma Rutherford		
I want to add time data from my last 4 walkathlons.		
So that I can see if I am improving from event to event.		

Exercise input	#2	HIGH
As Ethan Geary		
I want to record any exercise I have just done at the gym and how long it took.		
So that I can be reminded in future workout routines when I return to the app.		

Group reports	#8	MED
As Dan Johnson		
I want to be able to see how much and what exercise my friends do.		
So that I can work on getting healthier with other people.		

1.2.4 F3.1 Response

"You derived the classes from NLM on user stories. How did you decide what should be a class and what should be a class attribute."

The main factor in deciding whether a noun we identified in section 3.1 would be considered to become a class or class attribute would be what they describe. Nouns that could be fully defined by just a single piece of information would be used as class attributes. Complex nouns that required more than one piece of information to fully define would become classes. For example, the class *User* would never be able to be used if it was formed of just a name in the context of application. It requires additional information, such as age, height and weight. These factors could only have a single value for each user, and so would be considered to be class attributes.

"Also – in the OOA class diagram you have the following relationships: Association from Admin to User – should this not be inheritance?"

Please see the updated OOA class diagram for the correction.

"The composite relationships between User and other 'tracker' classes seems plausible but you should explain why these relationships are indeed composite."

The main reason we defined the relationship between User and Tracker classes as composite is their strong dependency on each other. Our idea for implementation was to define each User with a blank tracker for each aspect of the app, Exercise, Food, Water etc. as part of their account. If a User object exists, it must have the ability to record information in trackers as values of zero or no information still constitute results when displayed in the user's reports. Similarly, trackers are useless without an associated user, and therefore should only be able to be created for a ser to use.

2 Stage 2 Additional Requirements and Updates

2.1 Revised MoSCoW

The first stage in expanding the functionality of the application was making a list of potential features that we could add. Below is our updated MoSCoW analysis from stage 1, with the potential new features or changes marked in blue. Features that were successfully implemented in are indicated with a red tick.

Must Have	Should Have	Could Have	Won't Have
User accounts ✔	Food database ✔	Macro counter	Barcode scanner
Personal information ✔	Calorie counter ✔	Sleep counter ✔	Heart rate counter
Exercise capture ✔	Water tracking ✔	Device connection ✔	Stress tracker
Diet capture ✔	Coaching ✔	Health quiz ✔	Glucose tracker
Goal capture ✔	Medication reminder ✔		Automatic workout tracking
Regular weight capture ✔	Appointment reminder ✔		
User groups ✔	Music player ✔		
User reports ✔			
Group reports ✔			

2.2 Functional requirements.

The following list is an explanation of additional functionality we have added during stage 2. Screenshots of these implementations are linked to the appendix.

- Added an appointment reminder tab that allows the user to add a start date and time, event title, and further notes, and will then receive email notifications for it.

- Added a medication reminder tab that allows the user to add a start date and time, medication name and further notes, and will then receive email notifications for it.

- Device connection has been implemented using the FitBit API, which users can view in the FitBit tab. The user is able to pair their FitBit with the application and have it's information displayed in the page.

- Added a music player that allows users to listen to music in the app. Music can be added and removed to/from a ‘Selected Tracks’ playlist. There are action buttons for pause, play, stop, as well as a volume slider and status indicator.

- A health quiz has been added which tests users on various medical points in order to educate them to be healthier.

2.3 Non-functional requirements.

The following list is an explanation of features that would be described as non-functional we have added during stage 2. These updates were not added to the MoSCoW analysis, as we did not view them as requirements specific to this application, but good bits of software in general. Screenshots of these implementations are linked to the appendix.

- Made changes to user interface on each tab to be more intuitive and easier to read, and replaced background image too. These changes are in response to the feedback point F5.2.

2.4 Object oriented analysis.

The following diagram is a simplified view of the updated class structure of our application. It can be viewed as a shared file in Google Drive.

OOA Model

2.5 Object oriented design.

2.5.1 Model class diagram

Below is the final iteration of the model class diagram used for our application. It can also be viewed as a shared file in Google Drive.

Full Model Class Diagram

2.5.2 MVC Diagram

Below is the updated model-view-controller diagram updated for stage 2. It has been split up as suggested in the report template. They are linked to shared files on Google Drive.

Controller - Model

Controller - View

Model - Database

Full MVC Diagram

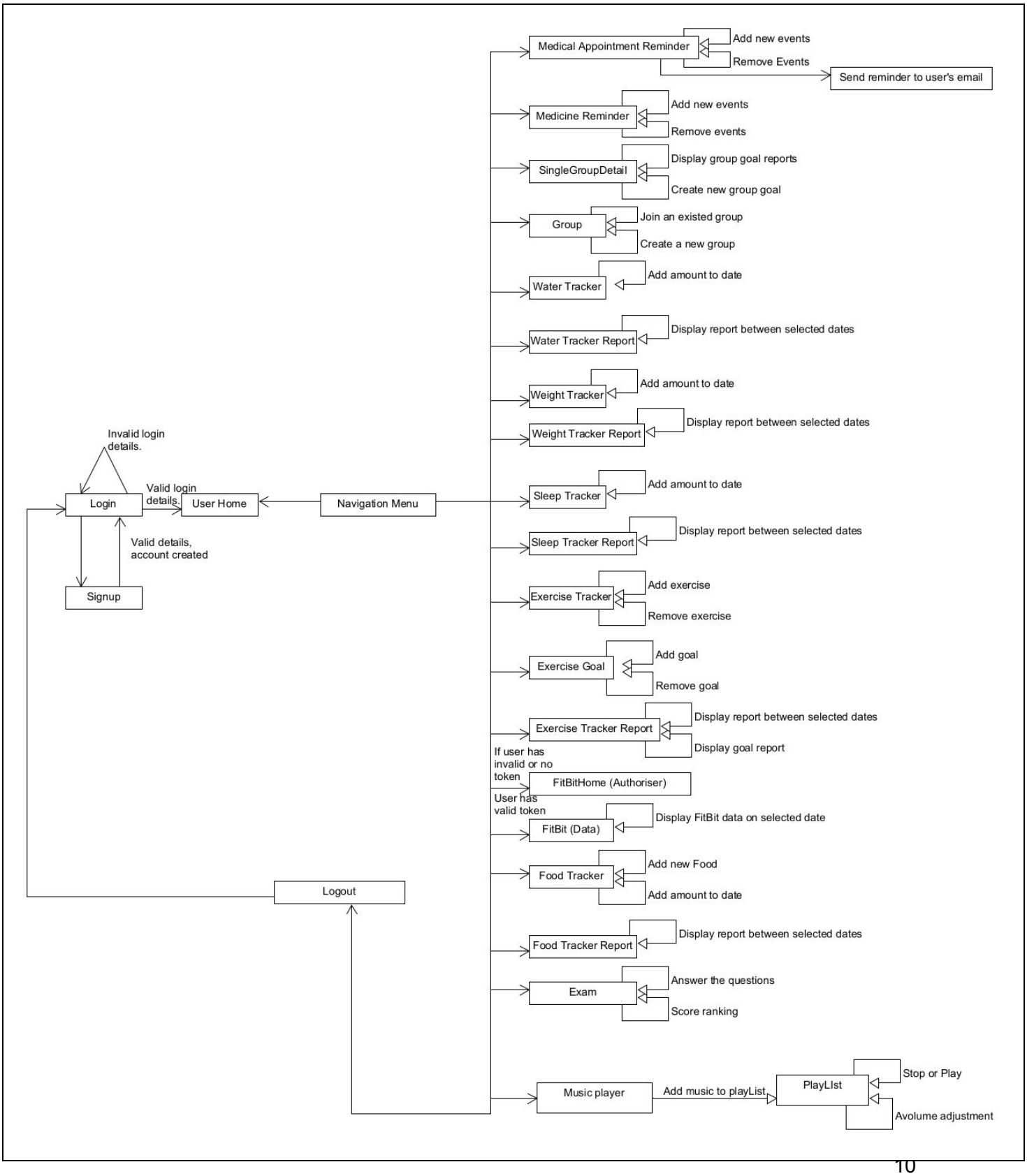
2.6.1 Details

For stage 2, we continued to use Java as our programming language of choice, as changes in the coursework requirements meant we would not have to port into a web application. If this were to have been needed, our code would have been converted to Python and we would have then used the Django framework for web development. We continued to use JavaFX for making the user interface, as feedback from the stage 1 was largely positive. JUnit was used for our test framework. New API's used were the FitBit Web API[1] for device connection and JavaMail API[2] for sending medication and appointment reminders. PostgreSQL and pgAdmin 4 continued to be used to create and manage the application's database.

[1] https://dev.fitbit.com/build/reference/web-api/
[2] https://javaee.github.io/javamail/

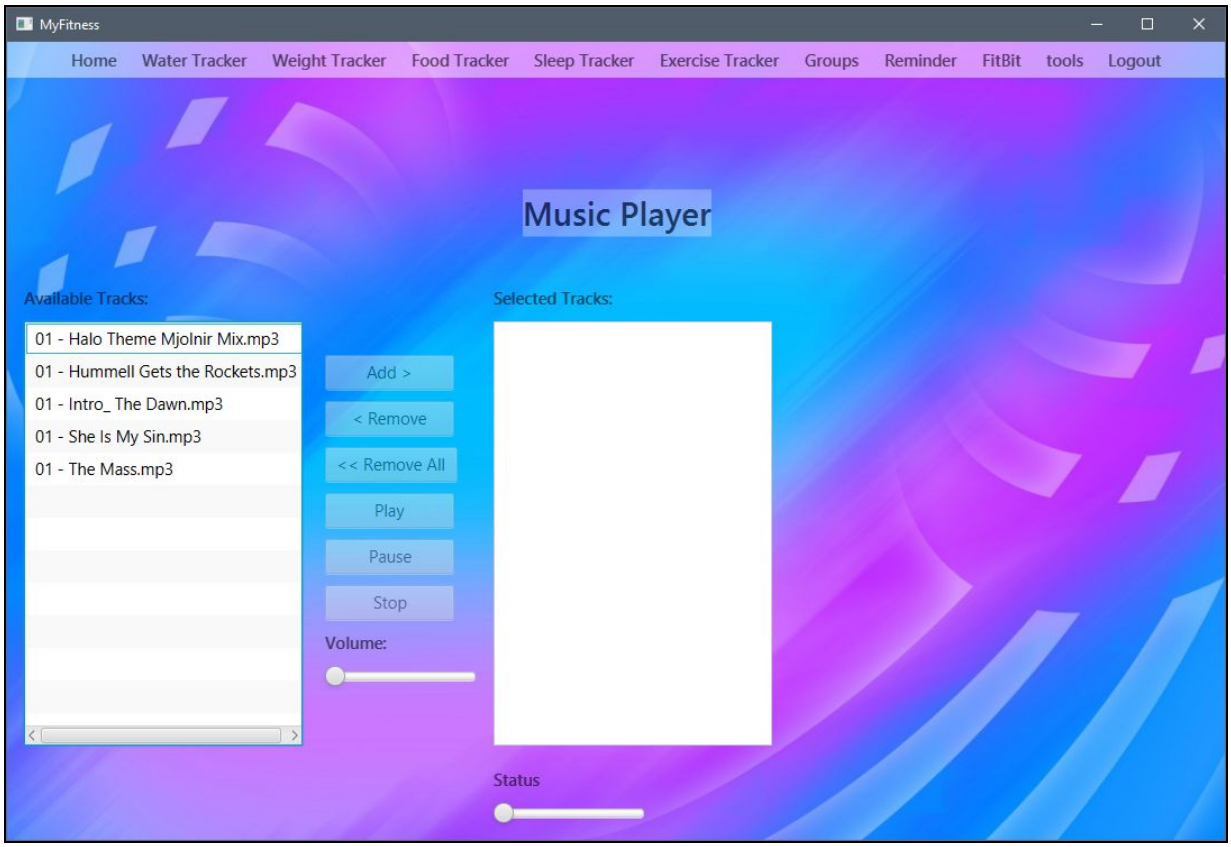
2.6.2 Front end (UI/View) state diagram.

[The UI's state transition diagram is viewable as a pdf here as well.]



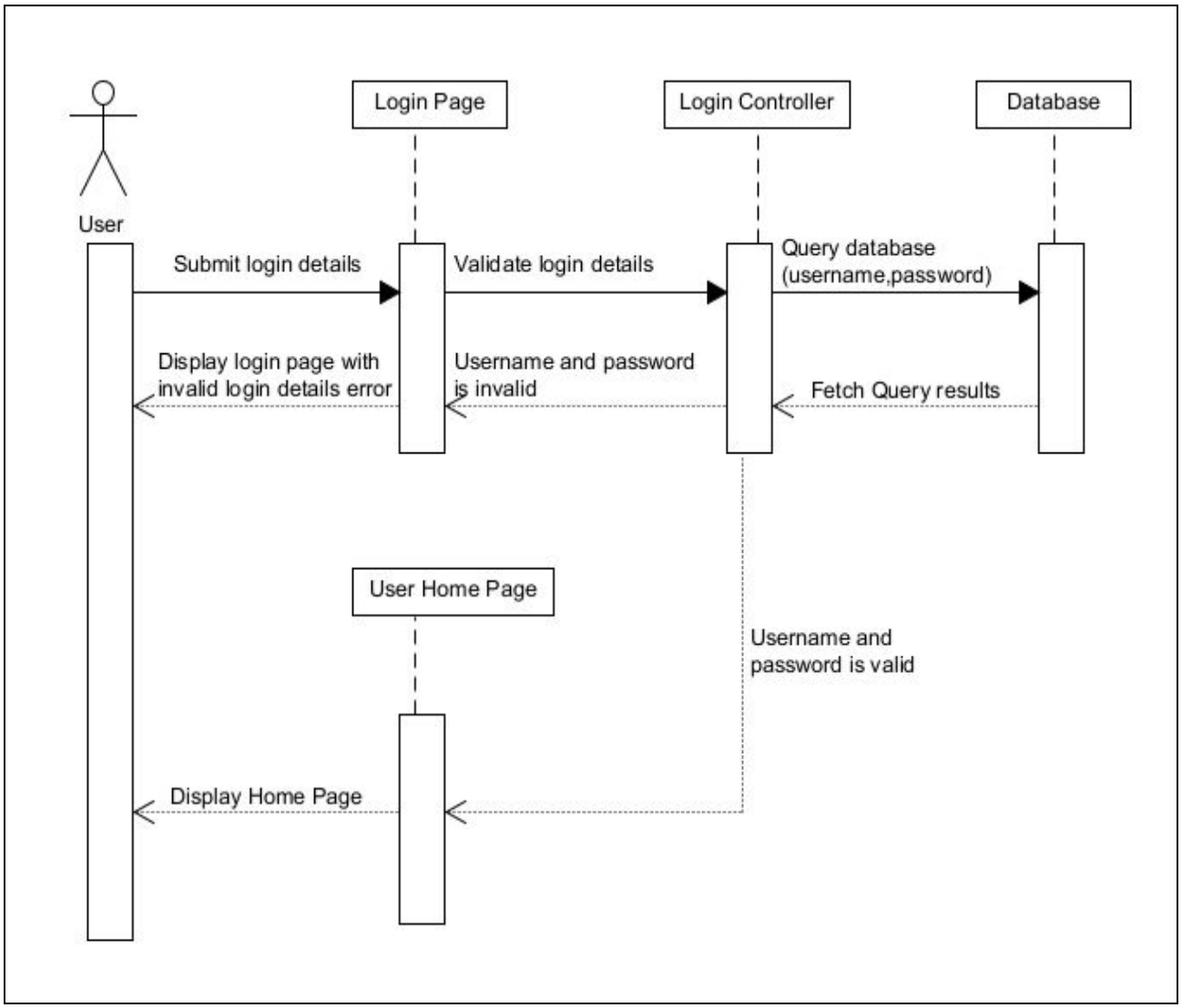
2.6.3 Front end design.

From the state's in the above diagram, the below is a screenshot of an example state of the UI, the Music Player. More screenshots of the UI can be seen at **A.1** in the appendix.



2.6.4 Sequence diagram.

The following shows the final iteration of a full stack sequence diagram for the login procedure in our application.



2.6.5 Application specific code.

Whilst our application functionality does not rely on many complex algorithms or calculations, an example of a simple, inherent calculation would be the calculation of a users BMI based on their weight and height. Below is the code fragment.

File - C:\Users\James\Downloads\MyFitnessApp\src\Model\user\User.java

```
209 public double calculateBMI(){
210         BMIValue = weight/((height/100)*(height/100
    ));
211         System.out.println("Controller.weight: " +
    weight);
212         System.out.println("height: " + height);
213         System.out.println("bmvalue : " + BMIValue);
214         System.out.println(BMI_UNDERWEIGHT);
215         if( BMIValue < BMI_UNDERWEIGHT){
216             bmiCategory = BMICategory.UNDERWEIGHT;
217         }
218         else if( BMIValue >= BMI_NORMAL_lower &&
    BMIValue <= BMI_NORMAL_upper){
219             bmiCategory = BMICategory.NORMAL;
220
221
222         }
223         else if( BMIValue >= BMI_OVERWEIGHT_lower
     && BMIValue <= BMI_OVERWEIGHT_upper){
224             bmiCategory = BMICategory.OVERWEIGHT;
225
226
227         }
228         else if( BMIValue >= BMI_OBESE_lower &&
    BMIValue <= BMI_OBESE_upper){
229             bmiCategory = BMICategory.OBESE;
230
231
232         }
233         else if(BMIValue >= BMI_MORBIDLY_OBESE_upper
    ){
234             bmiCategory = BMICategory.MORBIDLY_OBESE
    ;
235
236
237         }
238         return BMIValue;
239     }
```

2.6.6 Testing

2.6.6.1 Unit tests (dynamic white box)

When doing white box testing we used JUnit unit testing framework to produce a set of test results that represent a good sample of the methods used in our application. The tests we ran are as follows:

- calculating a user's BMI value.
- adding multiple users to a created group.
- sending email reminders to a specified email.

The source code, test result page and screenshots are available in the appendix at **A.5**, **A.6** and **A.7.**

2.6.6.2 User test (dynamic black box)

For our black box testing, we designed a set of questions for a sample user to answer whilst using the app. We carried out two tests, one at the start of the stage 2 development where any changes were yet to be made, and one at the end of the development process with our final product. We did this to get a better indication on what features to implement and gain more opinionated feedback on what the user interface was like and how it could be improved.

This table shows the original test, carried out on 18/04/2020.

Before stage 2 development.

Test	How	Why	Pass/Fail	Additional Notes
Is the UI aesthetically pleasing?	Asked the tester their opinion of the overall look of the app.	We want to gauge the testers to see if the app is appealing to look at and to see if it will hold their interest.	Neither pass or fail.	Improve the background image. Rearrange the ui elements, and make them fit the app's theme
Is text and all items easily visible and readable?	Asked the tester if they think text and items are easily readable and visible.	To test to see if placement, size and overall position of text and items fit with the app and look correct in their location.	Pass.	Could look a bit better, everything is a bit placed randomly but it is still readable and visible.
Is it easily traversable?	Asked the tester to navigate round the app and get their opinion if they think it easy and reliable to navigate through the app.	To test to see if using the functions in the pages are easy to use and the tester is able to interact fully and easily with the app.	Pass.	No navigation problems.

If you made an error, is it easy to correct?	Asked the user to make a mistake with data entry and do some actions incorrectly and asked them if they were easily able to correct their mistakes.	To make sure the app handles incorrect usage properly and allows the user to not feel locked into any certain feature at any time.	Pass.	App has back buttons/ cancel buttons for any inputs, and can move around with the menu bar.
Can you leave the app easily?	Asked the user to exit the app by clicking the exit window button.	To ensure that the app is easily closable.	Pass.	No comment.
Can you leave any page easily?	Asked the user to start performing a task on a page, then leave it.	To test if leaving a page during navigation breaks or ruins the users experience.	Pass.	No comment.
Is the app efficient in its navigation?	Asked the user to perform tasks and generally use the app to see if the navigation is efficient in its routing to certain pages.	To gauge how intuitive our UI and navigation menu is.	Pass.	Navigation menu is simple and intuitive to use.
Is the app useful?	Asked the user their personal thoughts on the app and if they find it useful and if they would want to use it.	To find out if the app is actually a useful end product to the user and any alterations that can be made to improve the app.	Pass.	Yes the app is useful but could have more features that relate to health/lifestyle such as a music section.The app has no way to get information from any devices, so something like this could be added to make it more useful.

These comments were then used to update the requirements for stage 2 developments, as seen in the MoSCoW analysis in section 2.1.

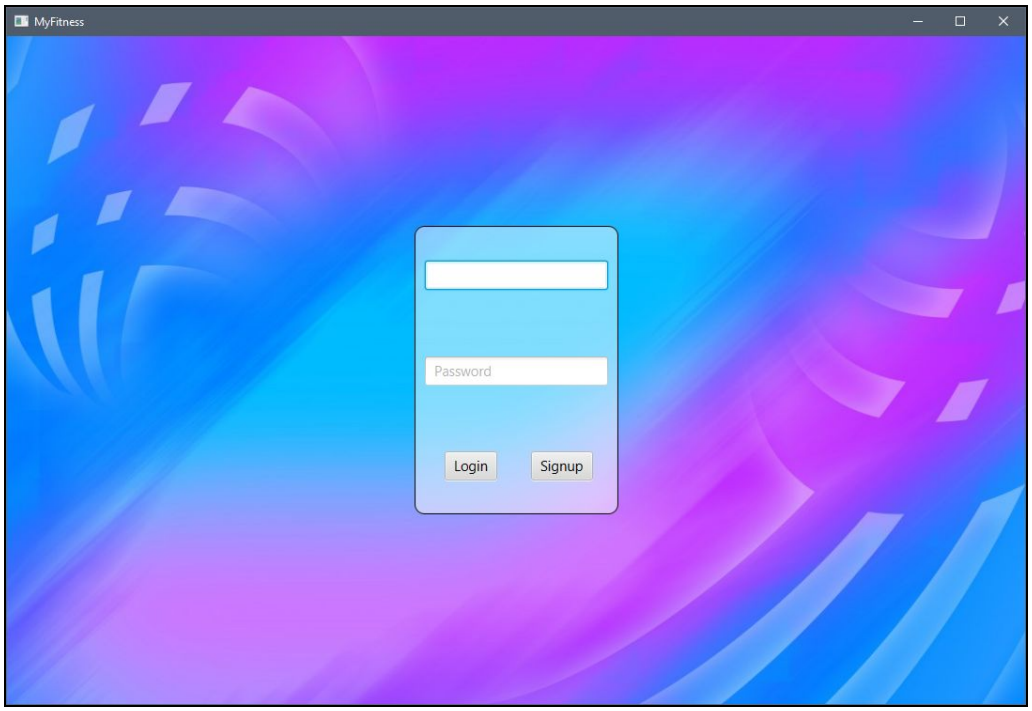
After stage 2 development.

Test	How	Why	Pass/Fail	Additional Notes
Is the UI aesthetically pleasing?	Asked the tester their opinion of the overall look of the app.	We want to gauge the testers to see if the app is appealing to look at and to see if it will hold their interest.	Pass	IThe background has been improved, the ui elements have been moved around and now everything seems to fit nicely with the entire theme of the app.
Is text and all items easily visible and readable?	Asked the tester if they think text and items are easily readable and visible.	To test to see if placement, size and overall position of text and items fit with the app and look correct in their location.	Pass.	Everything is now properly readable and fits where the items are placed.
Is it easily traversable?	Asked the tester to navigate round the app and get their opinion if they think it easy and reliable to navigate through the app.	To test to see if using the functions in the pages are easy to use and the tester is able to interact fully and easily with the app.	Pass.	No navigation problems.
If you made an error, is it easy to correct?	Asked the user to make a mistake with data entry and do some actions incorrectly and asked them if they were easily able to correct their mistakes.	To make sure the app handles incorrect usage properly and allows the user to not feel locked into any certain feature at any time.	Pass.	App still has good navigation and able to go back from any input or cancel at any time.
Can you leave the app easily?	Asked the user to exit the app by clicking the exit window button.	To ensure that the app is easily closable.	Pass.	No comment.
Can you leave any page easily?	Asked the user to start performing a task on a page, then leave it.	To test if leaving a page during navigation breaks or ruins the users experience.	Pass	No comment.

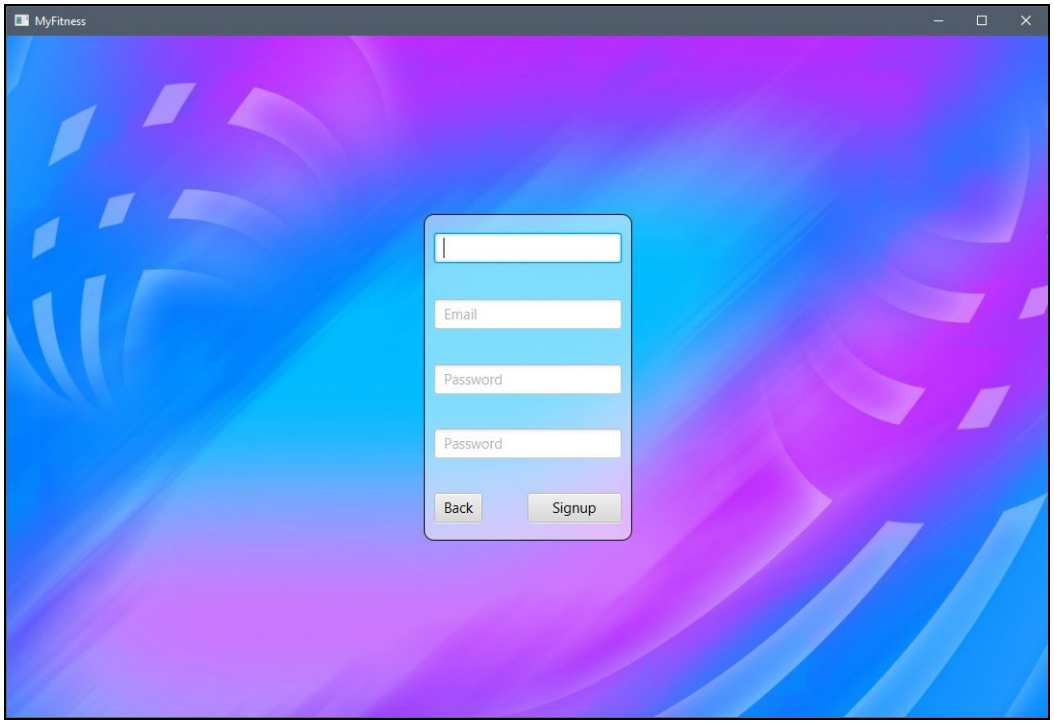
Is the app efficient in its navigation?	Asked the user to perform tasks and generally use the app to see if the navigation is efficient in its routing to certain pages.	To gauge how intuitive our UI and navigation menu is.	Pass	Menu bar is still easy to use.
Is the app useful?	Asked the user their personal thoughts on the app if they find it useful and if they would want to use it.	To find out if the app is actually a useful end product to the user and any alterations that can be made to improve the app.	Pass.	The extra features improve the app and make it much more useful as now you are able to listen to music from the app during a workout, take health quizzes, have reminders for appointments or medication and view information that is automatically taken from a fitbit watch or device.

3. Appendix

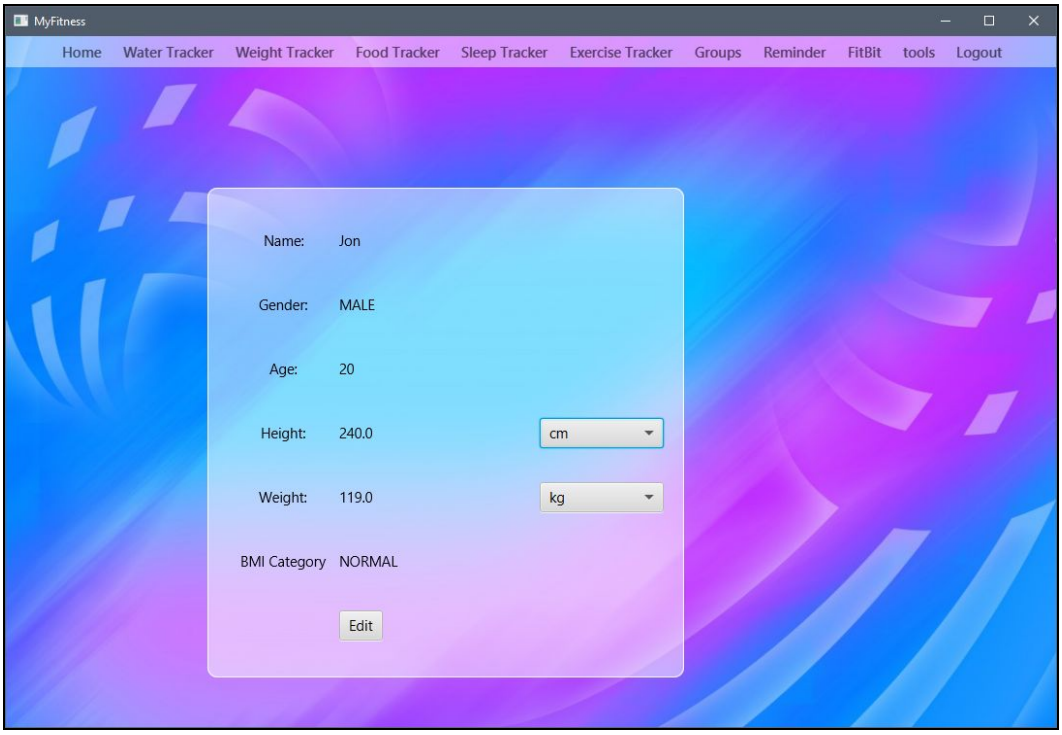
A.1 User interface screenshots showing implemented features and states in the state transition table.



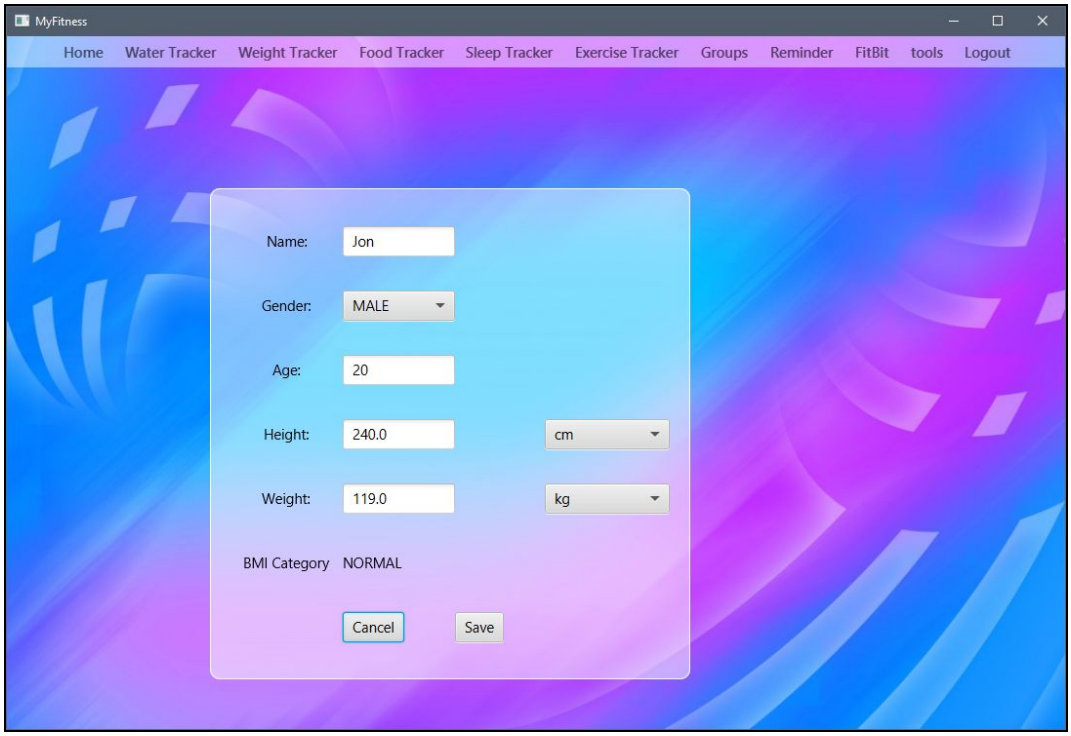
Login page.



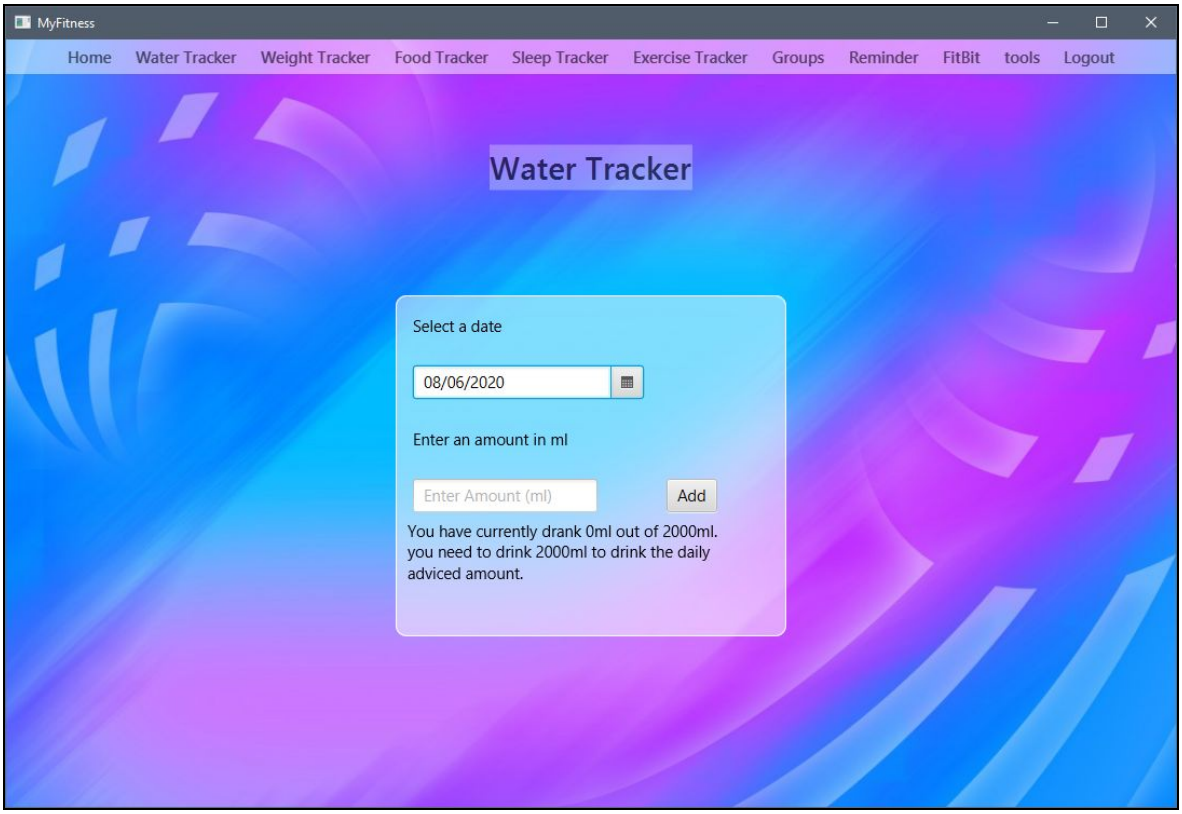
Signup page.



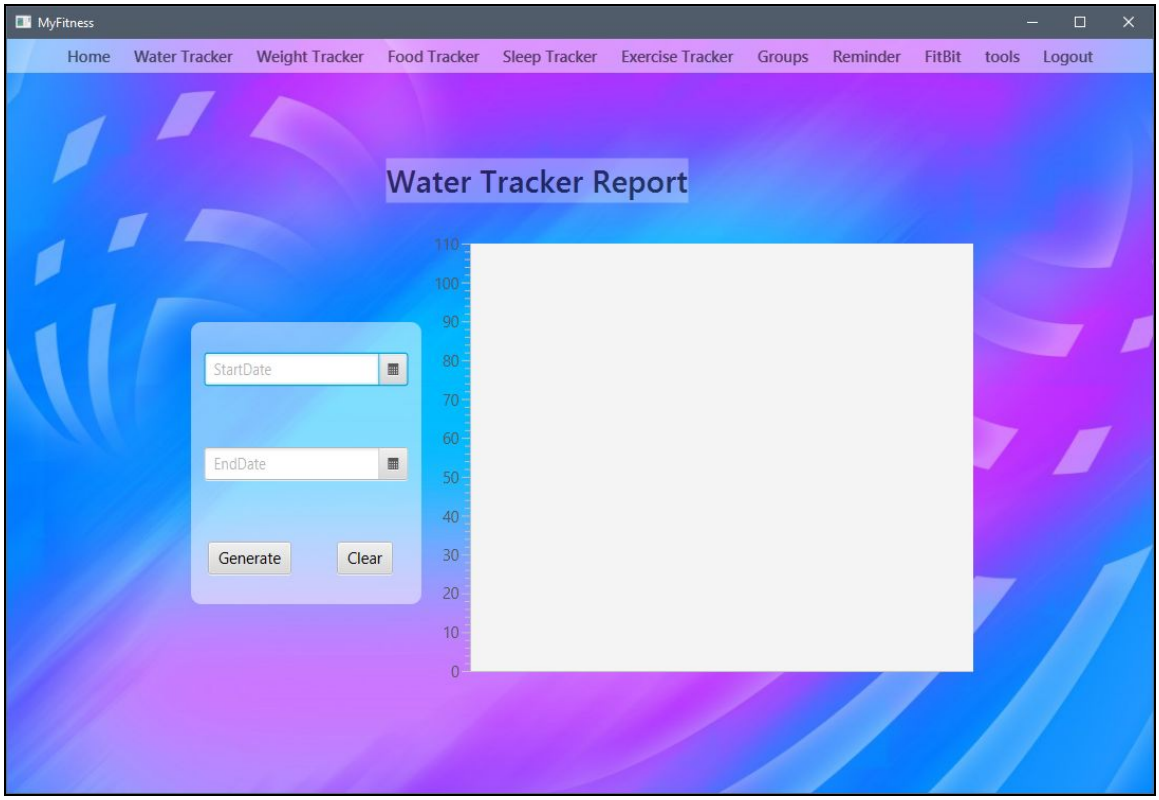
User information page.



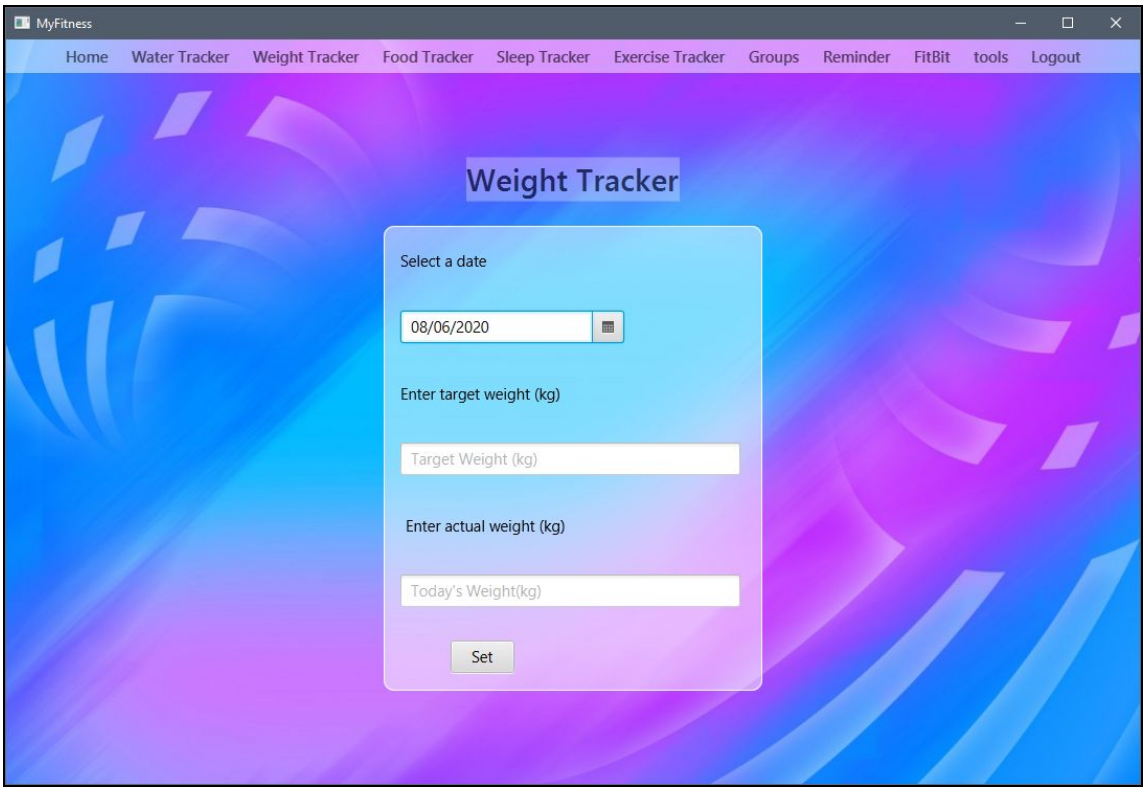
Editing user information.



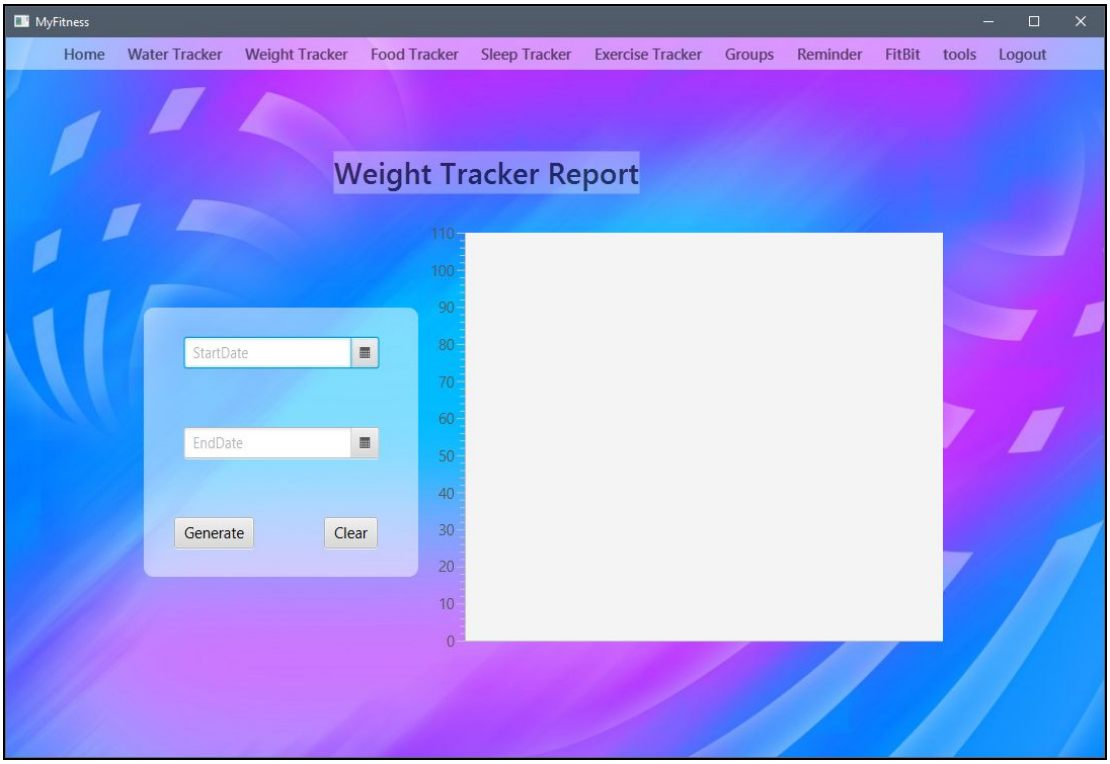
Water tracker page.



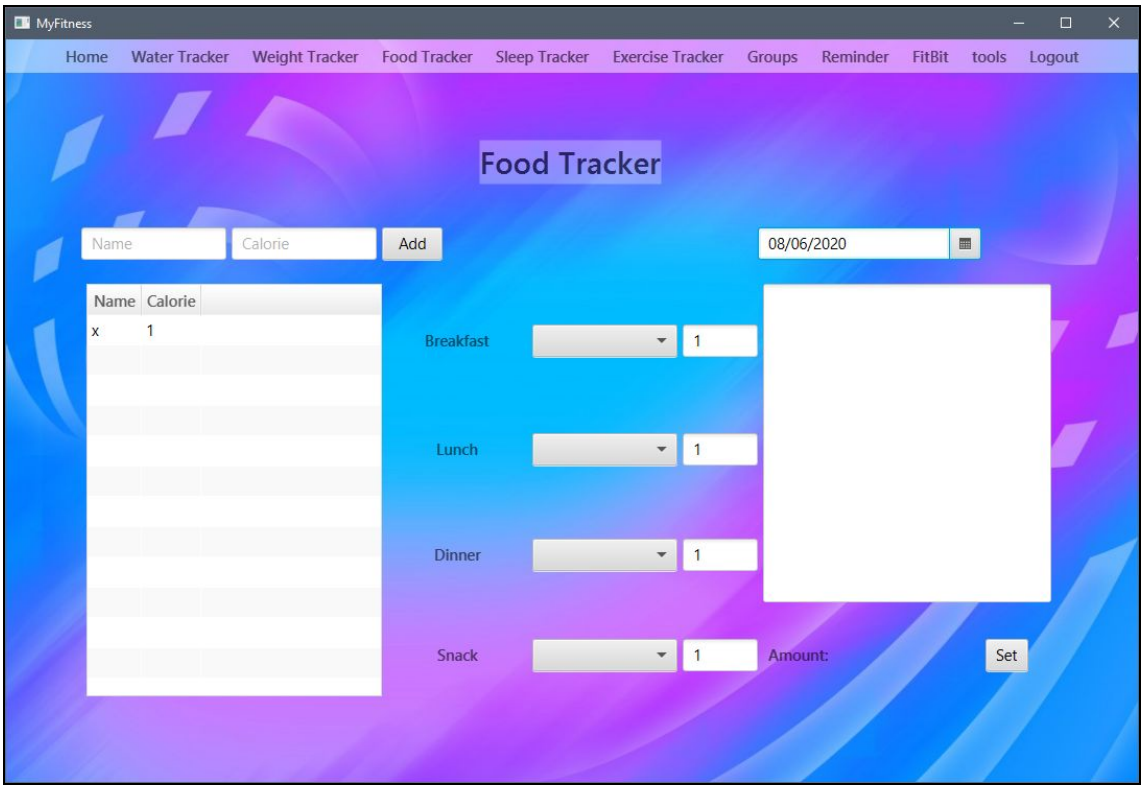
Water tracker report page.



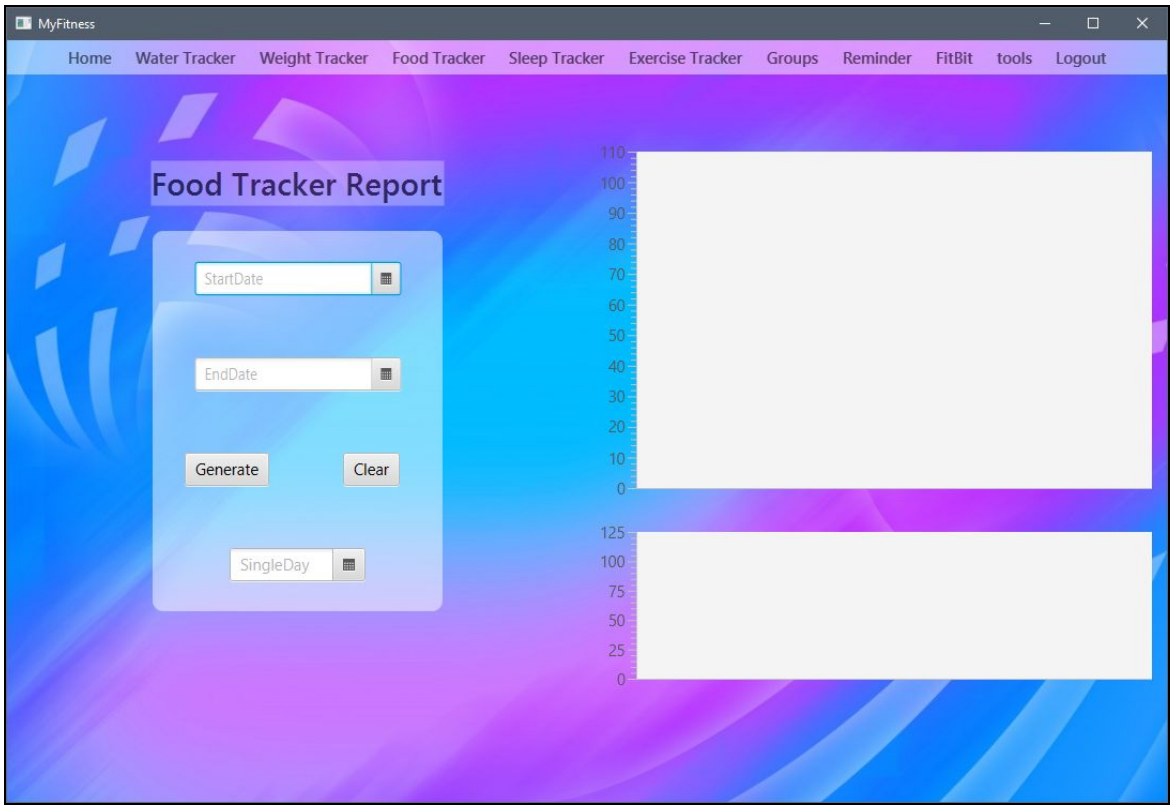
Weight tracker page.



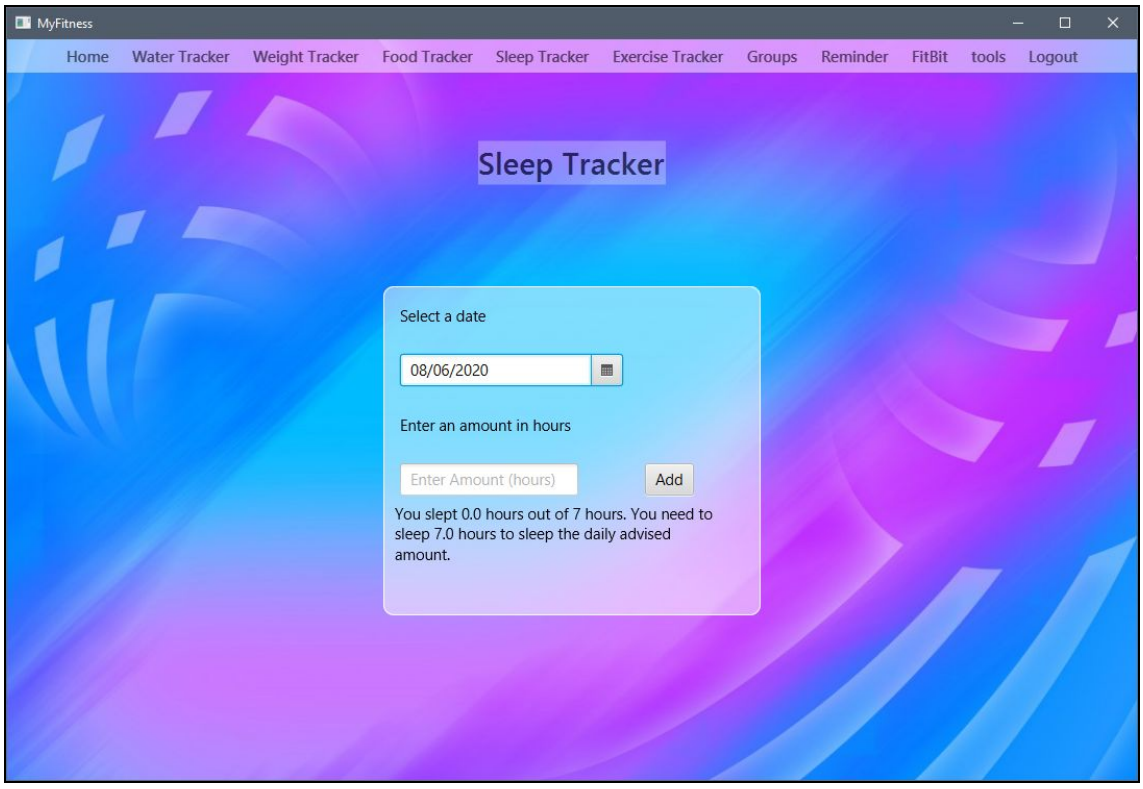
Weight tracker report page.



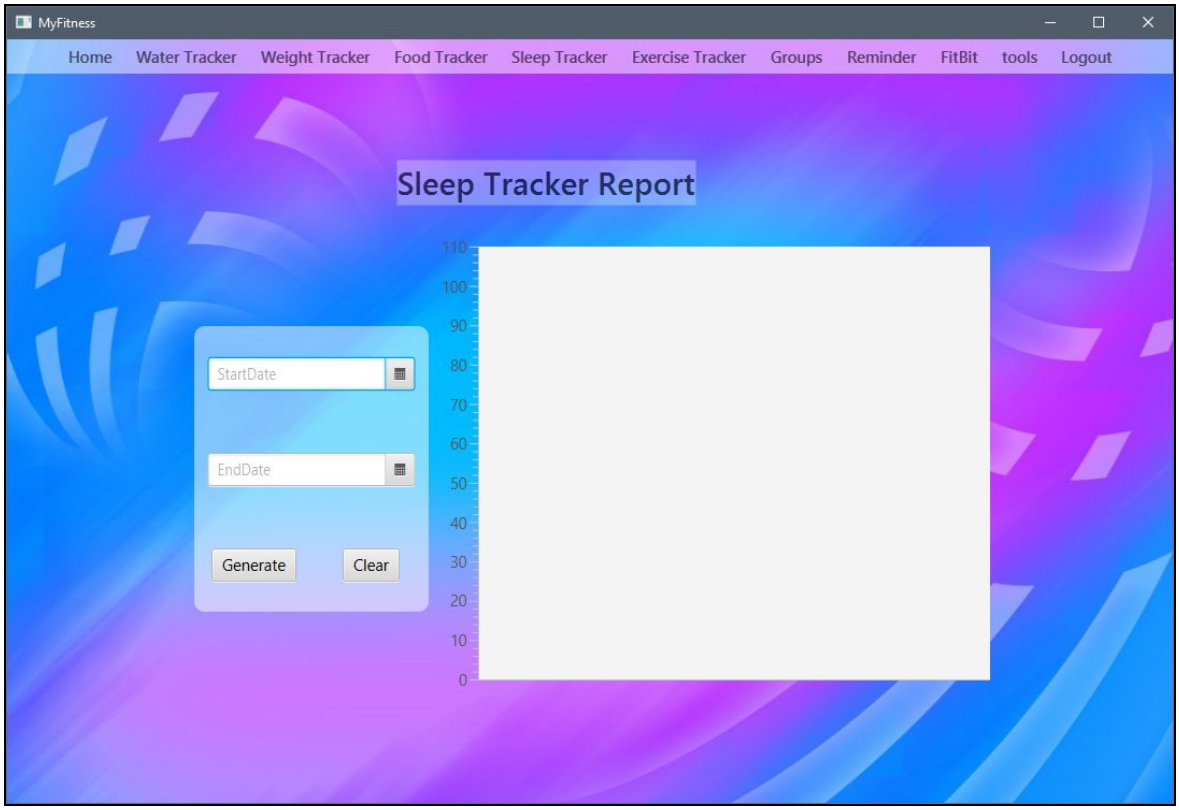
Food tracker page.



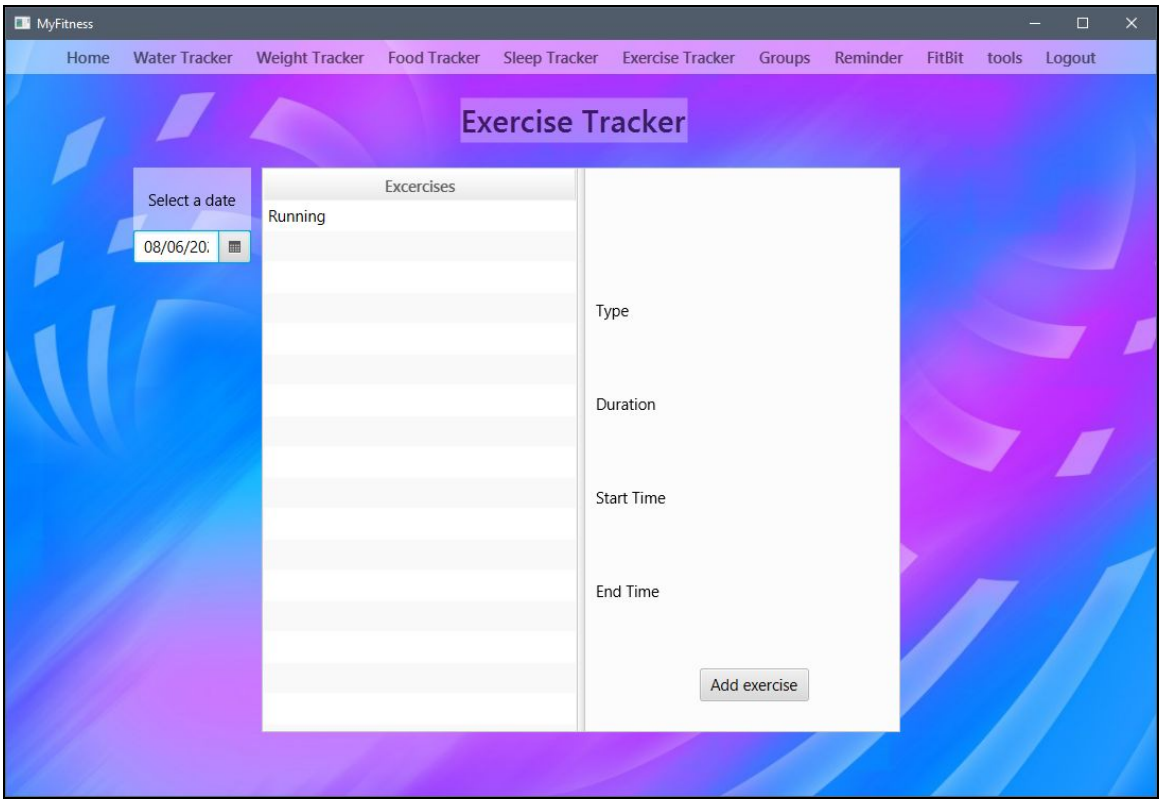
Food tracker report page.



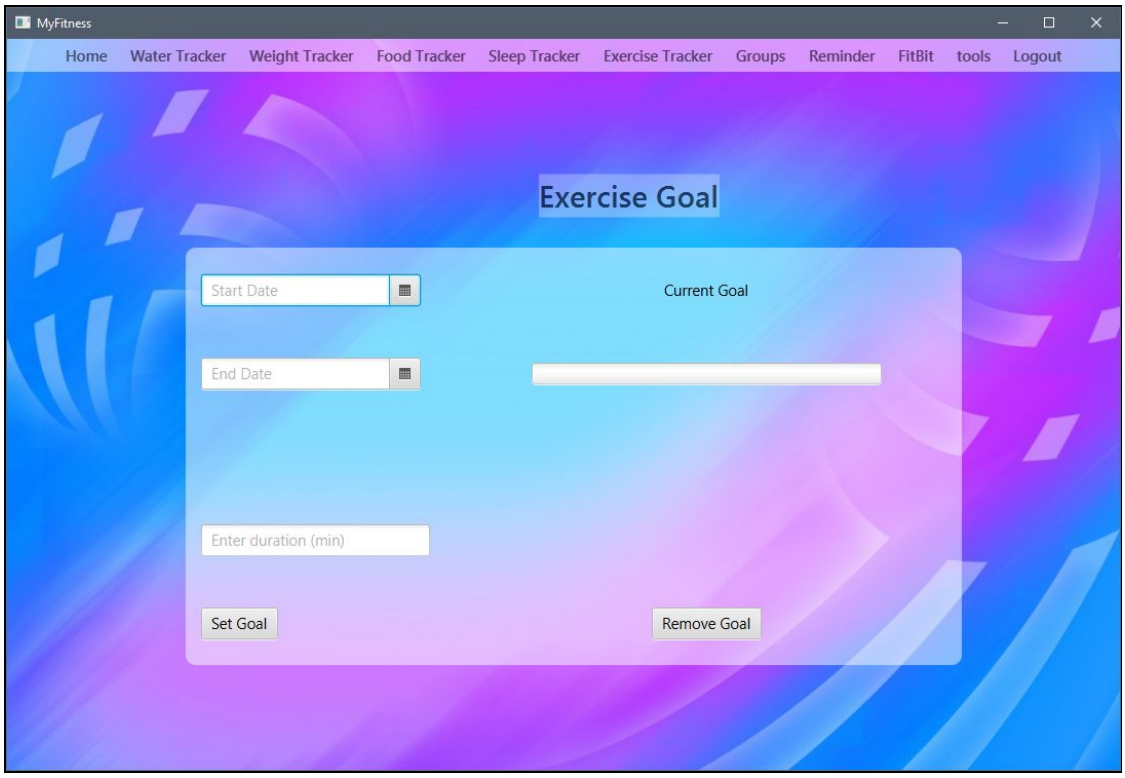
Sleep tracker page.



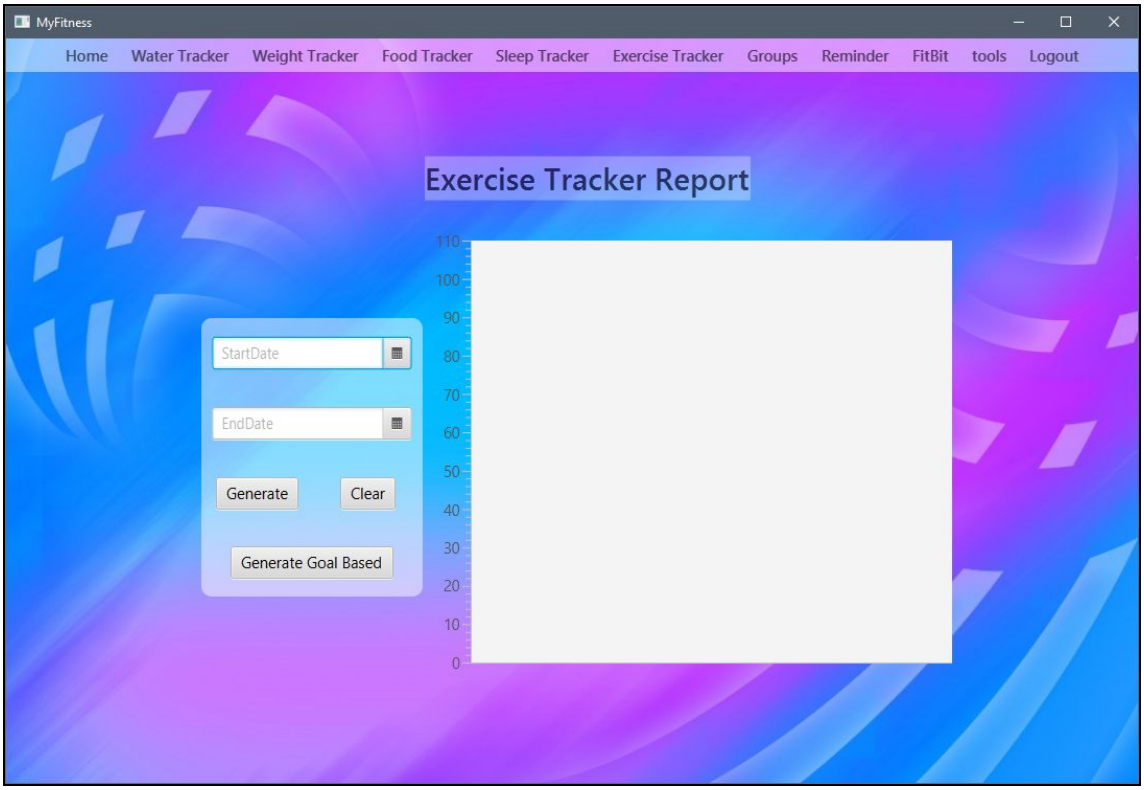
Sleep tracker report page.



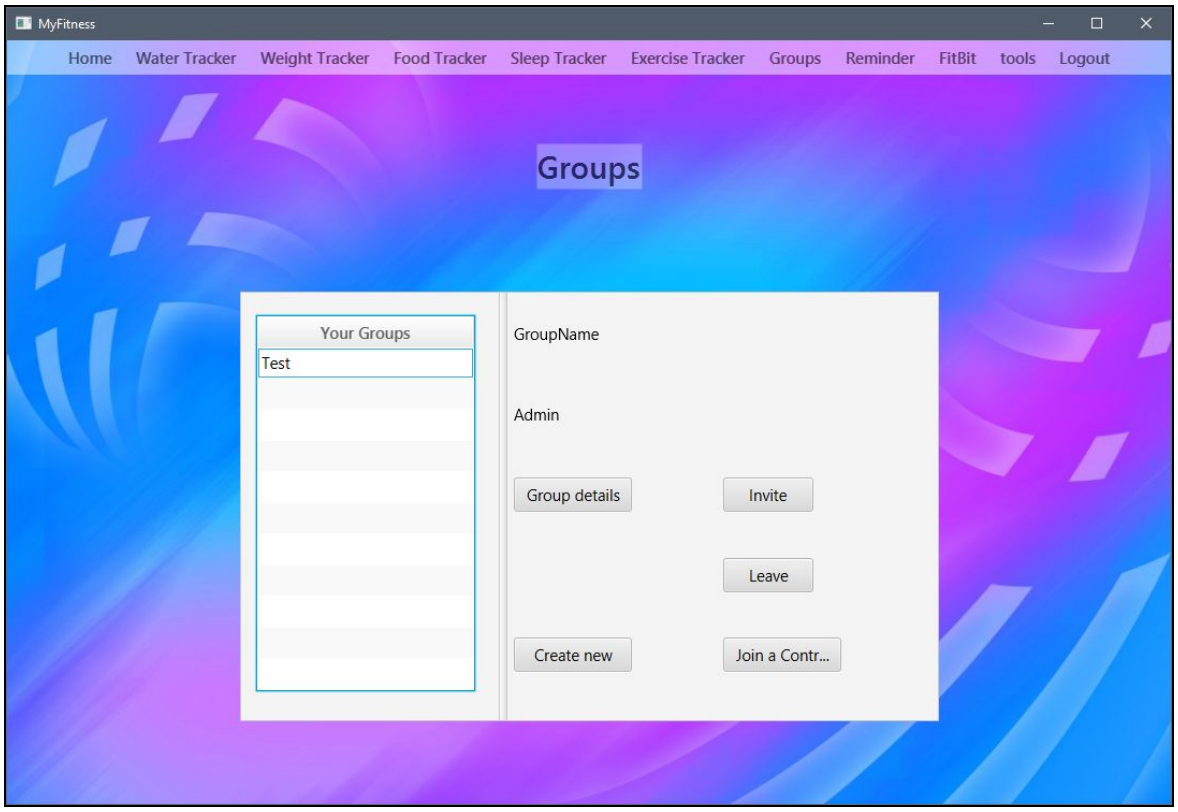
Exercise tracker page.



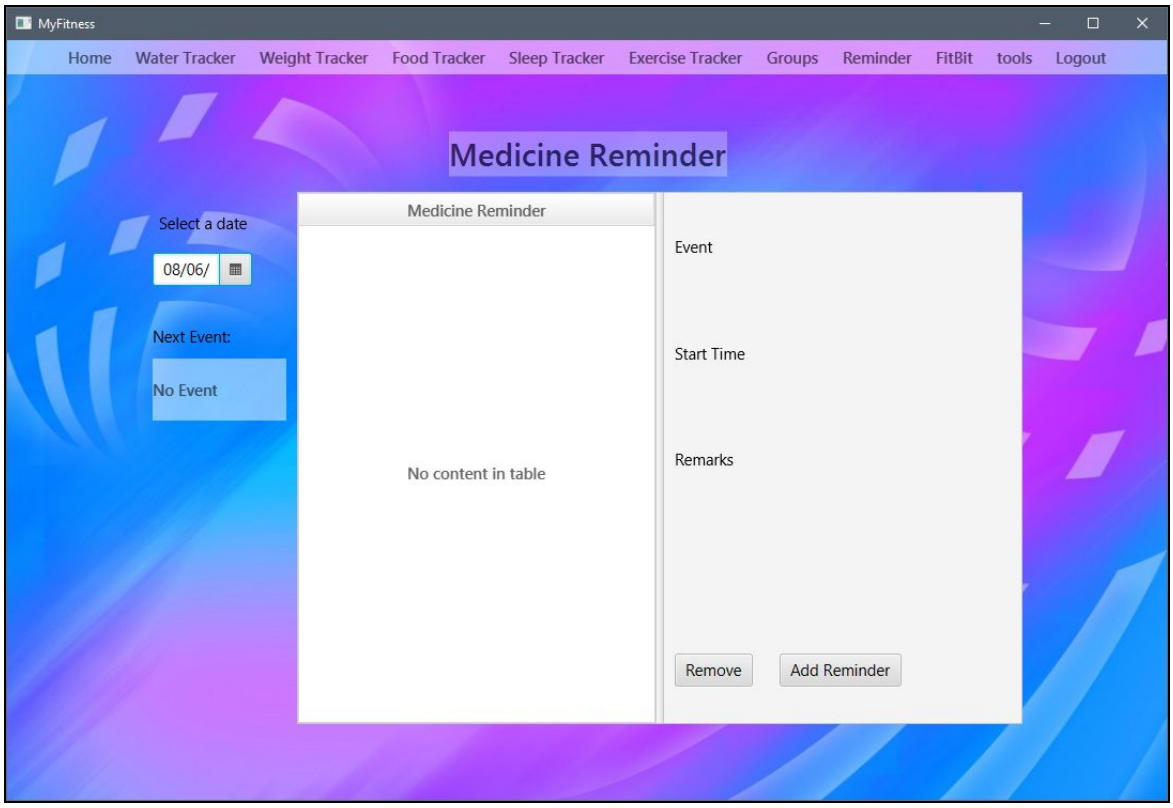
Exercise goal page.



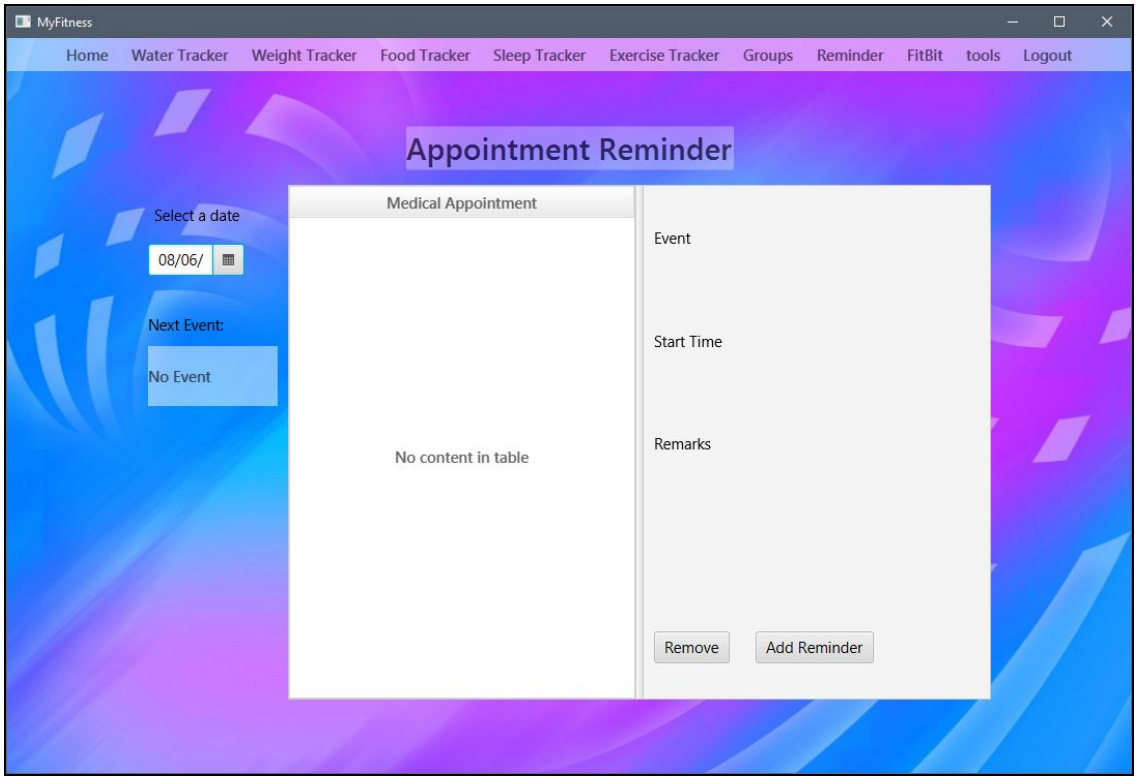
Exercise tracker report page.



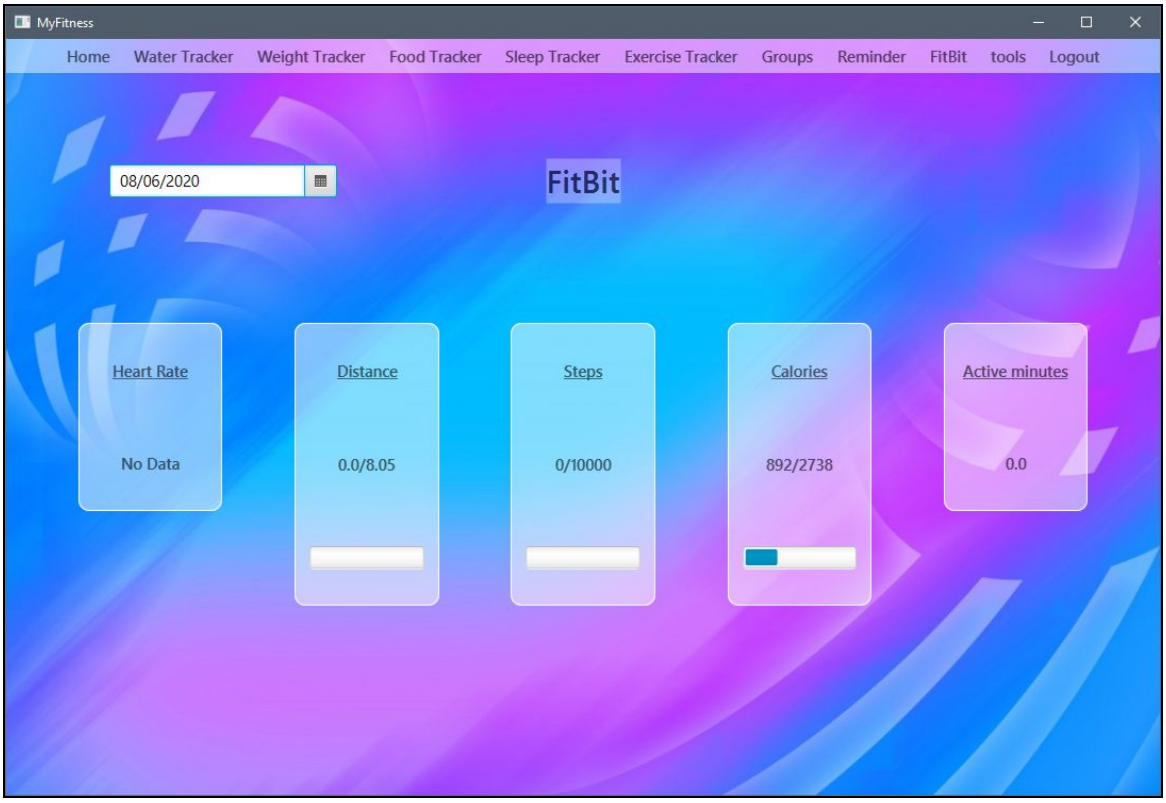
Groups page.



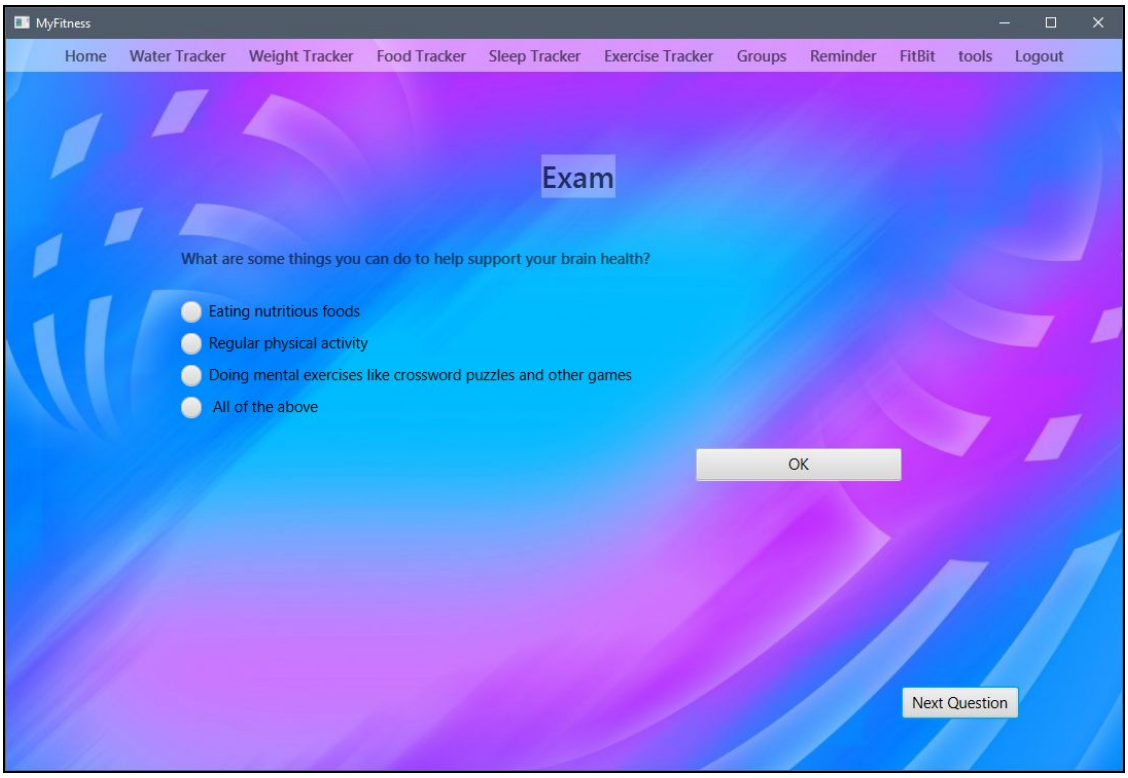
Medicine reminder page.



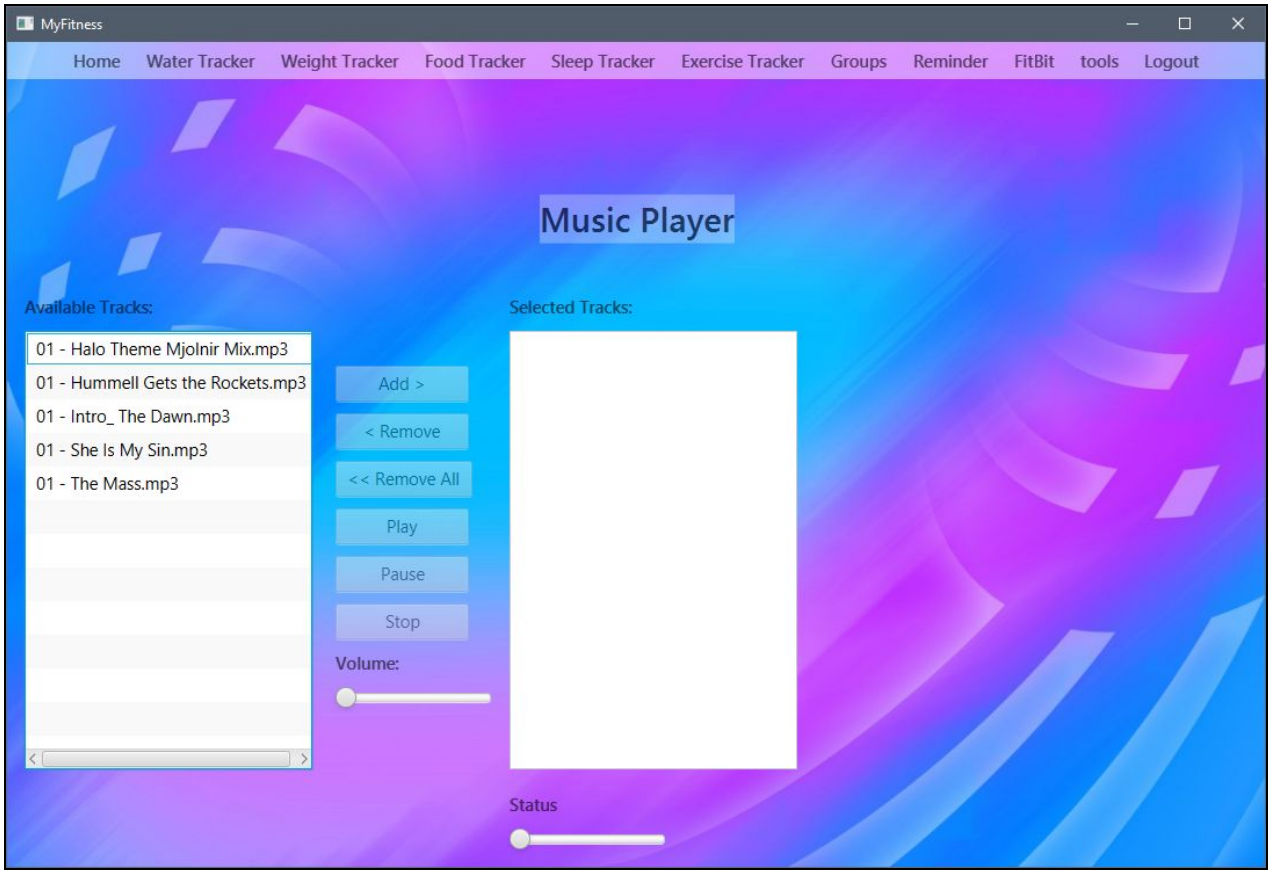
Appointment reminder page.



FitBit page.

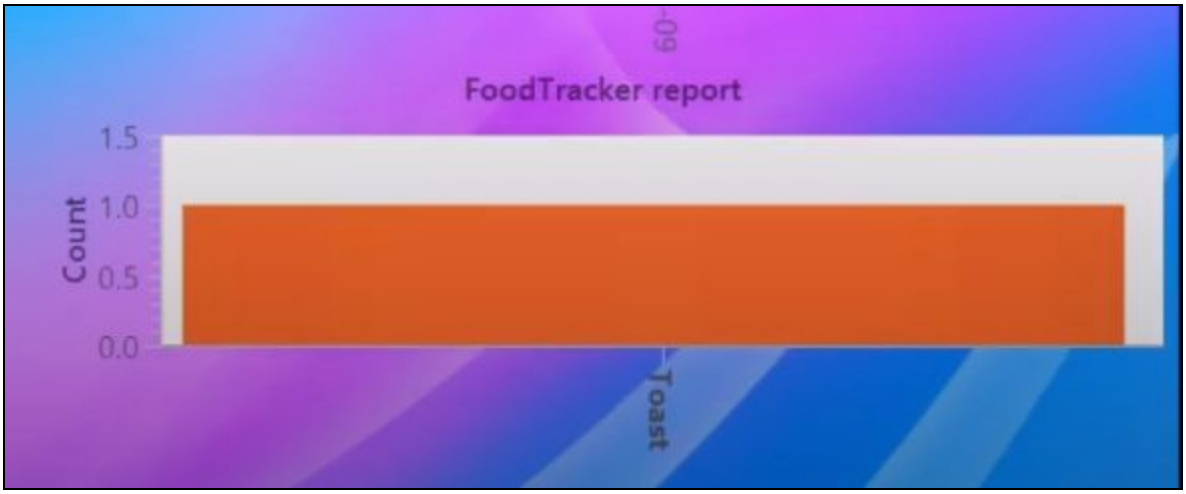


Health quiz page.



Music player page.

A.2 Feedback on labels screenshot.



A.3 Feedback on not hardcoding values into code I.

```
/*
 *
 * Description: This method calculates the users BMI based on their height and Controller.weight
 * Param: void
 * Returns: double
 *
 */
public double calculateBMI(){
    BMIValue = weight/((height/100)*(height/100));
    System.out.println("Controller.weight: " + weight);
    System.out.println("height: " + height);
    System.out.println("bmvalue : " + BMIValue);
    System.out.println(BMI_UNDERWEIGHT);
    if( BMIValue < BMI_UNDERWEIGHT){
        bmiCategory = BMICategory.UNDERWEIGHT;
    }
    else if( BMIValue >= BMI_NORMAL_lower && BMIValue <= BMI_NORMAL_upper){
        bmiCategory = BMICategory.NORMAL;


    }
    else if( BMIValue >= BMI_OVERWEIGHT_lower && BMIValue <= BMI_OVERWEIGHT_upper){
        bmiCategory = BMICategory.OVERWEIGHT;


    }
    else if( BMIValue >= BMI_OBESE_lower && BMIValue <= BMI_OBESE_upper){
        bmiCategory = BMICategory.OBESE;


    }
    else if(BMIValue >= BMI_MORBIDLY_OBESE_upper){
        bmiCategory = BMICategory.MORBIDLY_OBESE;


    }
    return BMIValue;
}
```

A.4 Feedback on not hardcoding values into code II.

```
/*
 *
 * Description: This method saves and verifies Controller.user inputted Controller.weight
 * Param:
 * Returns:
 *
 */
public boolean saveEditWeight(){
    String weightIn =user_enter_weight.getText();
    boolean valid = false;
    try {
        double ageConversion = Double.parseDouble(weightIn);
        valid = true;
    }
    catch (NumberFormatException e) {
        valid = false;
    }

    System.out.println(weightIn);
    System.out.println(valid);
    System.out.println(user.getWeightUnitPreference());
    if(valid &&  Double.parseDouble(weightIn) >0 &&  Double.parseDouble(weightIn) < user.getWEIGHT_MAX_KG() && user.getWeightUnitPreference().equals("kg")){//100
        user.setWeight( Double.parseDouble(weightIn));
        weightError.setVisible(false);
        return true;

    }
    else if(valid &&  Double.parseDouble(weightIn) >0 &&  Double.parseDouble(weightIn) < user.getWEIGHT_MAX_LB() && user.getWeightUnitPreference().equals("lb")){//300
        user.setWeight( Double.parseDouble(weightIn)/2.205);
        weightError.setVisible(false);
        return true;

    }
    else{
        weightError.setVisible(true);
        editWeight();
        return false;
    }
```

```
/*
 *
 * Description: This method saves and verifies Controller.user inputted height
 * Param:
 * Returns:
 *
 */
public boolean saveEditHeight(){
    String heightIn =user_enter_height.getText();
    boolean valid = false;
    try {
        double ageConversion = Double.parseDouble(heightIn);
        valid = true;
    }
    catch (NumberFormatException e) {
        valid = false;
    }
    if(valid &&  Double.parseDouble(heightIn) >0 &&  Double.parseDouble(heightIn) < user.getHEIGHT_MAX_CM() && user.getHeightUnitPreference().equals("cm")){//250
        user.setHeight( Double.parseDouble(heightIn));
        heightError.setVisible(false);
        return true;

    }
    else if(valid &&  Double.parseDouble(heightIn) >0 &&  Double.parseDouble(heightIn) < user.getHEIGHT_MAX_IN() && user.getHeightUnitPreference().equals("in")){//96
        user.setHeight( Double.parseDouble(heightIn)*2.54);
        heightError.setVisible(false);
        return true;

    }
    else{
        heightError.setVisible(true);
        editHeight();
        return false;
    }

}
```

A.5 Unit test source code.

File - C:\Users\James\Desktop\MyFitnessApp1.96\MyFitnessApp\src\unitTest\UnitTest.java

```
 1 package unitTest;
 2
 3 import Model.email.Email;
 4 import Model.group.Group;
 5 import org.junit.Test;
 6 import Model.reminder.Reminder;
 7 import Model.user.User;
 8
 9 import static org.junit.jupiter.api.Assertions.assertEquals
   ;
10
11 /**
12  * Class UnitTest
13  *The UnitTest provides several tests the unit
   functionaries of the application
14  *@version 1.0
15  */
16 public class UnitTest {
17
18
19     /**
20      *Unit test for calculateBMI method in User class
21      */
22     @Test
23     public void testCalculateBMI()
24     {
25
26         User tester =new User();
27         tester.setHeight(170);
28         tester.setWeight(60);
29
30         assertEquals(20.76,tester.calculateBMI(),0.02);
31     }
32
33     /**
34      * Method testAddUser
35      * Test method addUser in class Group for user to join
   a group
36      */
37     @Test
38     public void testJoinGroup()
39     {
40         //Create a new user
41         User tester1=new User();
42         tester1.setName("Tester1");
43         User tester2=new User();
```

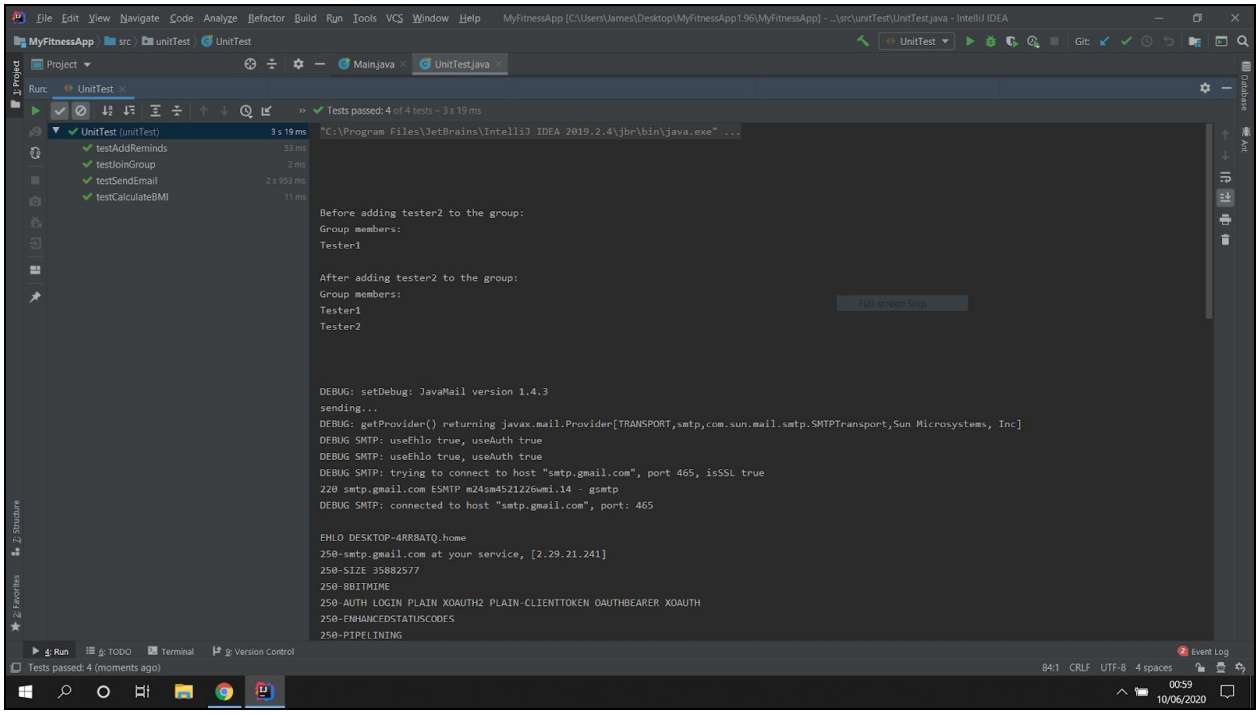
File - C:\Users\James\Desktop\MyFitnessApp1.96\MyFitnessApp\src\unitTest\UnitTest.java

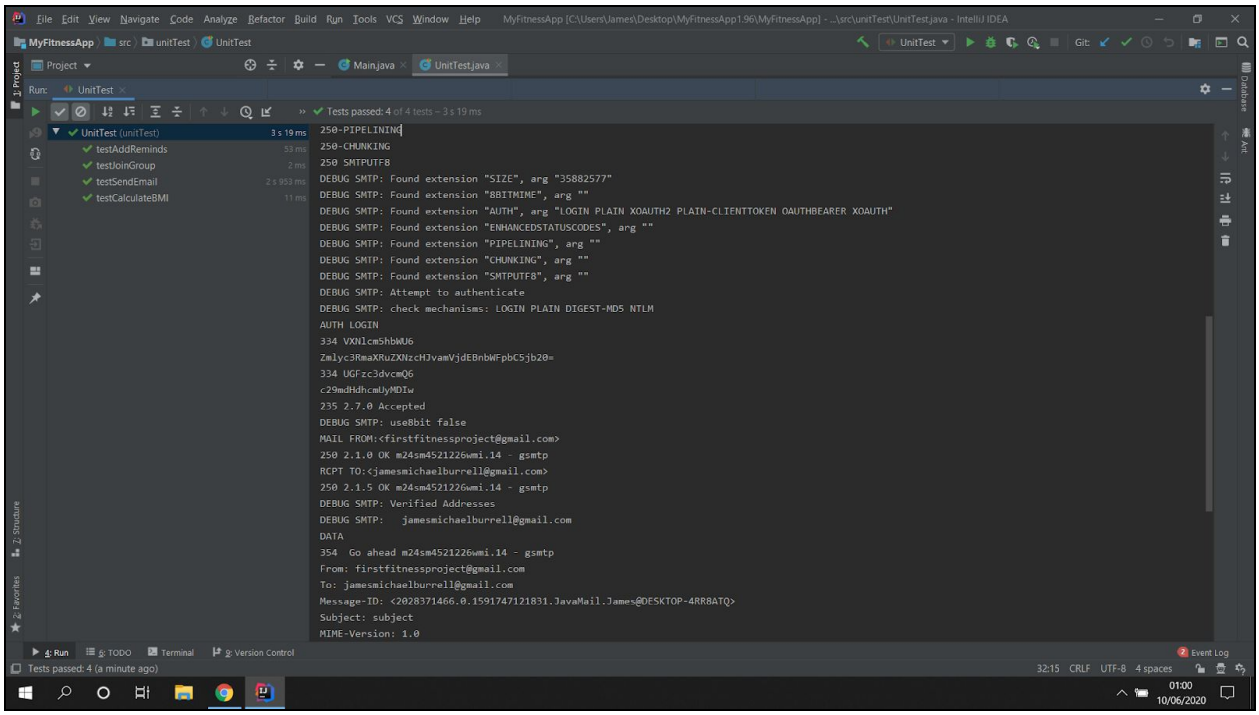
```
44          //Create second new user
45          tester2.setName("Tester2");
46          //create a new group
47          Group testGroup=new Group(tester1,"test Group");
48
49          //before adding tester2 to the group
50          System.out.println("Before adding tester2 to the
    group:");
51          System.out.println("Group members:");
52          for(User u:testGroup.getUsers())
53          {
54              System.out.println(u.getName());
55          }
56          System.out.println();
57          //after adding tester2 to the group
58          System.out.println("After adding tester2 to the
    group:");
59          System.out.println("Group members:");
60          testGroup.addUser(tester2);
61          for(User u:testGroup.getUsers())
62          {
63              System.out.println(u.getName());
64          }
65      }
66
67      @Test
68      public void testAddReminds()
69      {
70          //Create a new user
71          User tester1=new User();
72          tester1.setName("Tester1");
73          Reminder remind=new Reminder();
74          tester1.getReminderTracker().addReminder(remind);
75      }
76      @Test
77      public void testSendEmail()
78      {
79          Email.sendEmail("jamesmichaelburrell@gmail.com","
    subject","message");
80      }
81  }
82
```

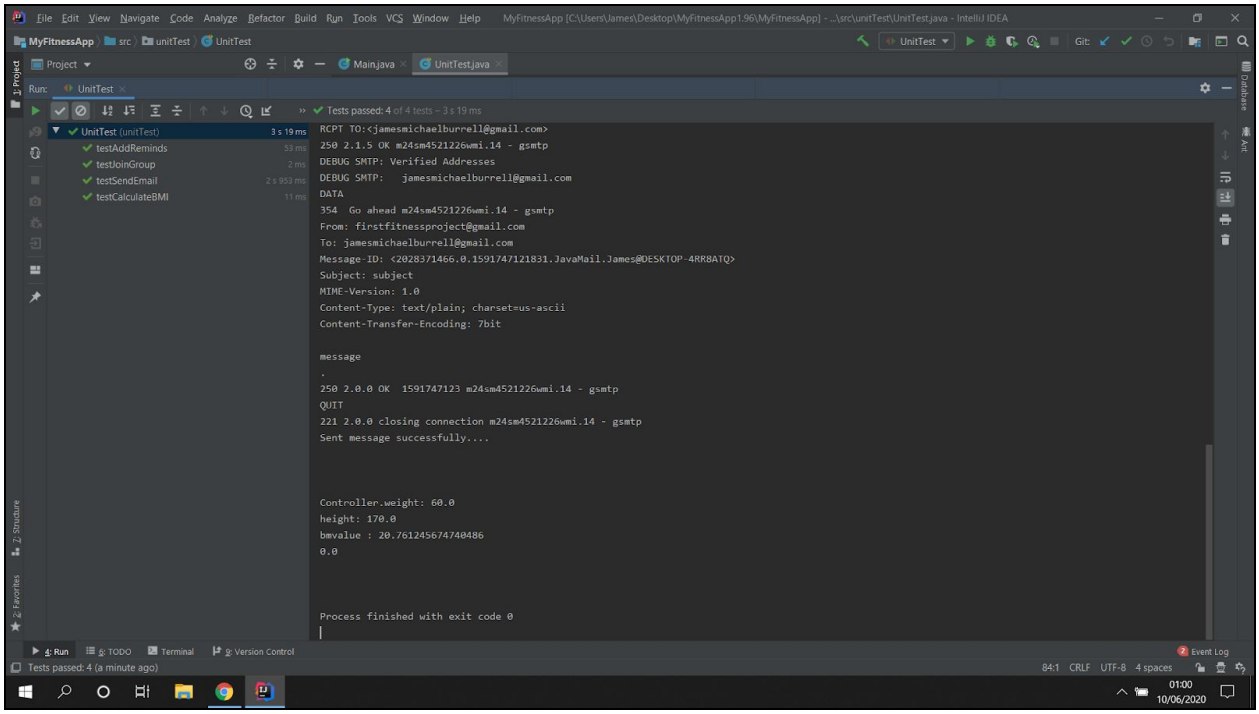
A.6 Unit test results.

[Link to unit test results page.]

A.7 Screenshots of tests.






RCPT TO:<jamesmichaelburrell@gmail.com>
250 2.1.5 OK m24sm4521226wmi.14 - gsmtp
DEBUG SMTP: Verified Addresses
DEBUG SMTP: jamesmichaelburrell@gmail.com
DATA
354 Go ahead m24sm4521226wmi.14 - gsmtp
From: firstfitnessproject@gmail.com
To: jamesmichaelburrell@gmail.com
Message-ID: <2028371466.0.1591747121831.JavaMail.James@DESKTOP-4RR8ATQ>
Subject: subject
MIME-Version: 1.0
Content-Type: text/plain; charset=us-ascii
Content-Transfer-Encoding: 7bit
message
.
250 2.0.0 OK 1591747123 m24sm4521226wmi.14 - gsmtp
QUIT
221 2.0.0 closing connection m24sm4521226wmi.14 - gsmtp
Sent message successfully....
Controller.weight: 60.0
height: 170.0
bmvalue : 20.761245674740486
0.0
Process finished with exit code 0

4. Glossary of terms.

Word	Description
User	People using software. Used to record user account information, physical information, and contact information.
Group	Created for many users, you can share personal experiences and set common exercise goals for multiple users.
Water tracker	Track and record users' daily drinking water.
Weight tracker	Track and record users' daily weight changes.
Exercise tracker	Track and record users' daily, weekly and monthly exercise.
Exercise	Select the item of exercise and the duration of the exercise.
Food tracker	Track and record users' daily diet.
Meal	Add or select food and choose the type of food to calculate total calories.
Sleep tracker	Track and record user sleep quality.
Tracker	Contains the current user and the user's goals. An abstract class for all tracker classes.
Database connector	Connect java code with database.
WeightDB	For weight page data storage operations.

FoodDB	For food page data storage operations.
FileTool	For meals table data storage operations.
Email	When a user invites another user to the group, a message is sent
AdminClass	For sleep, group, water, exercise, user, data storage operations.
Goal	An aim or purpose, the user sets the goal and goes to achieve it.