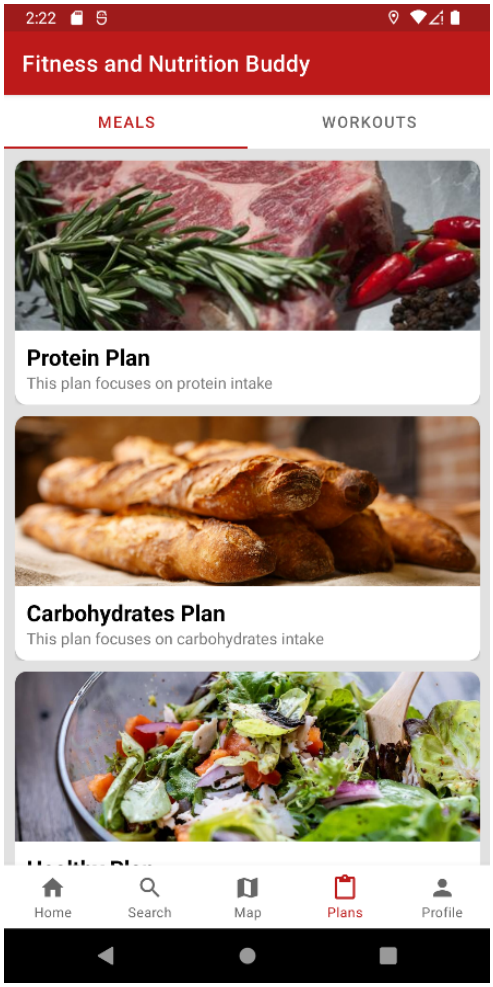
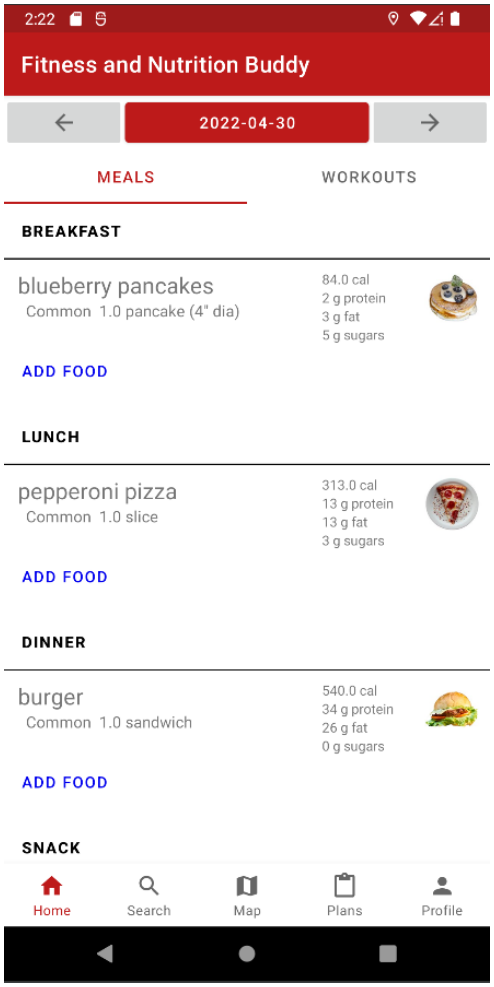
Fitness and Nutrition Buddy Final Report





Prepared by
Cristian Trandafir, John Mistica, Jayanth Podapati, Thomas Kubik
for use in CS 442
at the
University of Illinois Chicago
April 30, 2022

Table of Contents

List of Figures

List of Tables

I Project Description

1 Project Overview

Fitness and Nutrition Buddy is a mobile application that allows users to track their meals and workouts throughout the day. The user can also view macronutrient information of an added meal such as carbs, fats, proteins, sugars, etc… Users are also able to track their progress throughout a week and select a fitness/meal plan that they are working towards. Additionally, users will be able to look for their next meal by searching for nearby restaurants menus and add meals according to the preferences and filters. Users can also track their net calorie intake which shows them the amount of calories they have consumed and also burned from workouts.

2 Project Domain

This project lies within the health and fitness industry. This application will serve as a health and fitness aid to any user that uses it. It should act like a fitness trainer or dietician aiding the user in making healthy decisions. Users must be able to access and use this application via their mobile device (eg. Android). The user must be able to search for and add meals. The meals must adhere to the users preferences and filters. The user must also be able to search for and add workouts. Additionally, the user must also be able to view nearby restaurants relative to their current location via GPS. The user should be able to track their progress and past meals/workouts that they have added to their profile. Ultimately, the user must be able to view their progress over a week relative to their selected meal/workout plan and make fitness and health decisions based on the information they receive from the application. This application should satisfy any users health and fitness goals and advise them to live a healthier lifestyle.

3 Relationship to Other Documents

This project refers its requirements and design from Fitness and Nutrition Buddy Final Report by Andy Hansana, Kevin Elliott, Vincent Weaver, and Ryan Crowley. This document also relates to the project scenario documents: Meal Search and Selection Scenario, Profile Customization and Logging Meal Scenario, and Meal Planning and Touch Ups Scenario.

4 Naming Conventions and Definitions

4a Definitions of Key Terms

Preferences: User sets their preferences by setting calorie limits and various nutritional limits via the settings page.

Plans: A premade workout/meal plan that contains limits and goals that the user can select. The plan is confined to a weekly goal and only collects progress from a weekly interval.

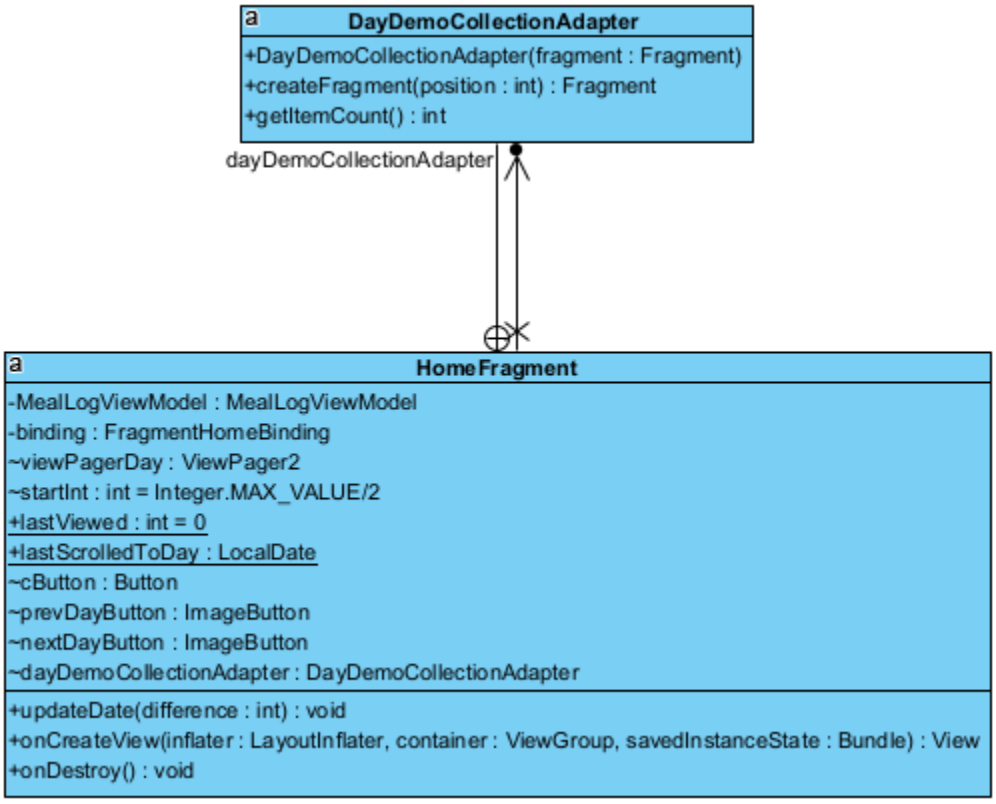
Progress: A physical bar that allows the user to view their progress towards achieving a certain goal. Eg. a calorie bar that displays the total calorie intake relative to the calorie limit.

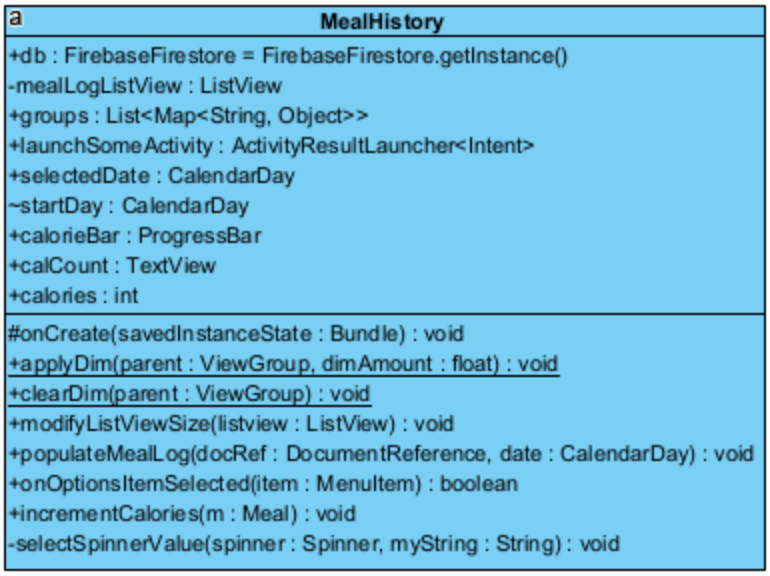
Touch Up: Refactoring code to improve UI elements such as adjusting button styling to adhere to applications theme.

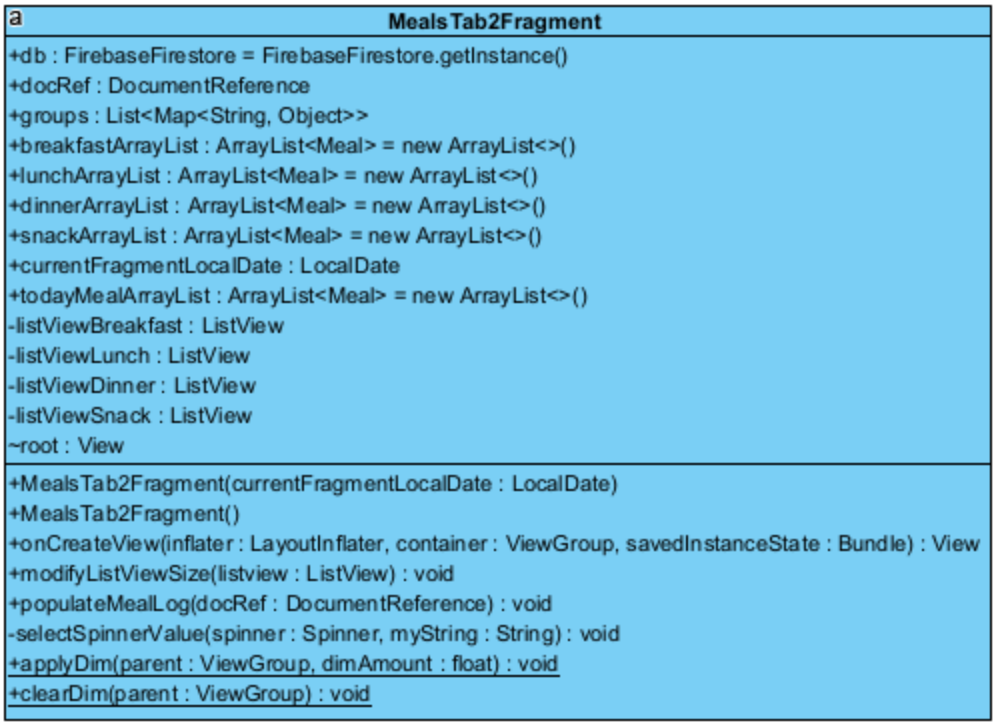
Customization: Ability for user to edit their profile data like age, weight, height, etc…

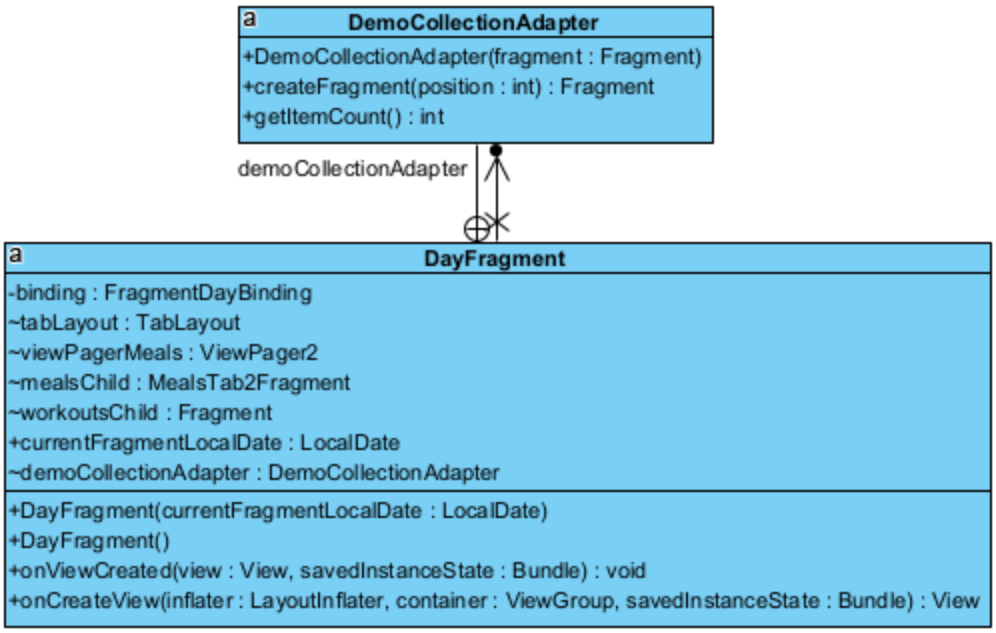
4b UML and Other Notation Used in This Document

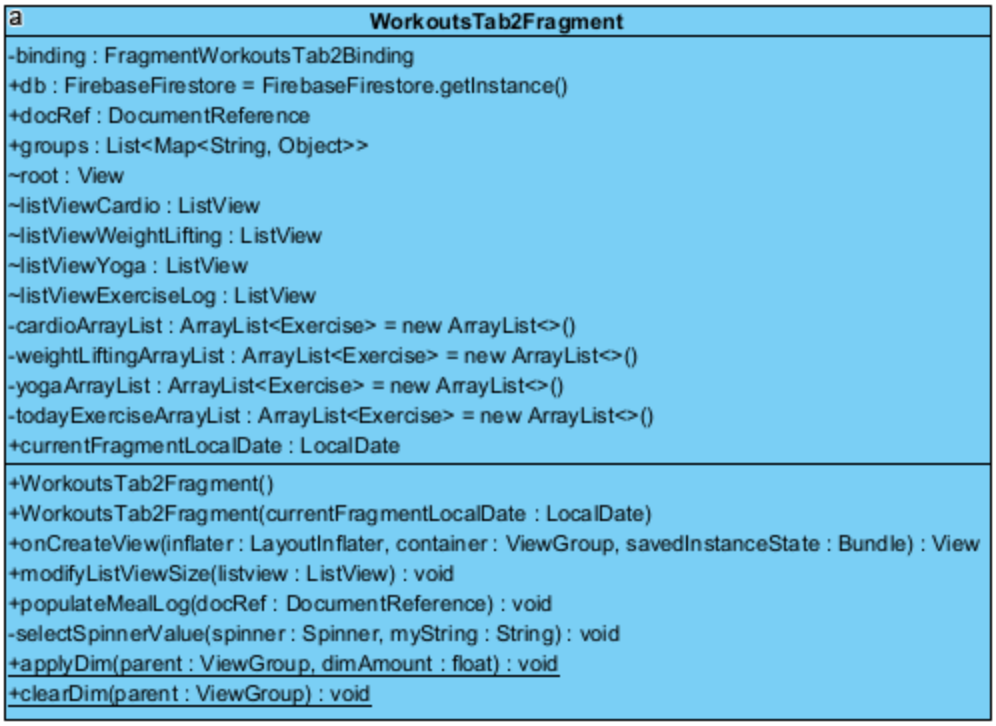
Home Screen UML



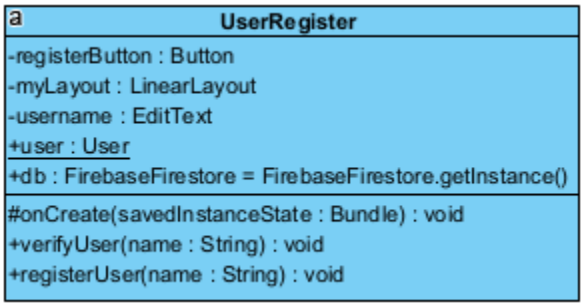

a
MealHistory
+db : FirebaseFirestore = FirebaseFirestore.getInstance()
-mealLogListView : ListView
+groups : List<Map<String, Object>>
+launchSomeActivity : ActivityResultLauncher<Intent>
+selectedDate : CalendarDay
~startDay : CalendarDay
+calorieBar : ProgressBar
+calCount : TextView
+calories : int
#onCreate(savedInstanceState : Bundle) : void
+applyDim(parent : ViewGroup, dimAmount : float) : void
+clearDim(parent : ViewGroup) : void
+modifyListViewSize(listview : ListView) : void
+populateMealLog(docRef : DocumentReference, date : CalendarDay) : void
+onOptionsItemSelected(item : MenuItem) : boolean
+incrementCalories(m : Meal) : void
-selectSpinnerValue(spinner : Spinner, myString : String) : void

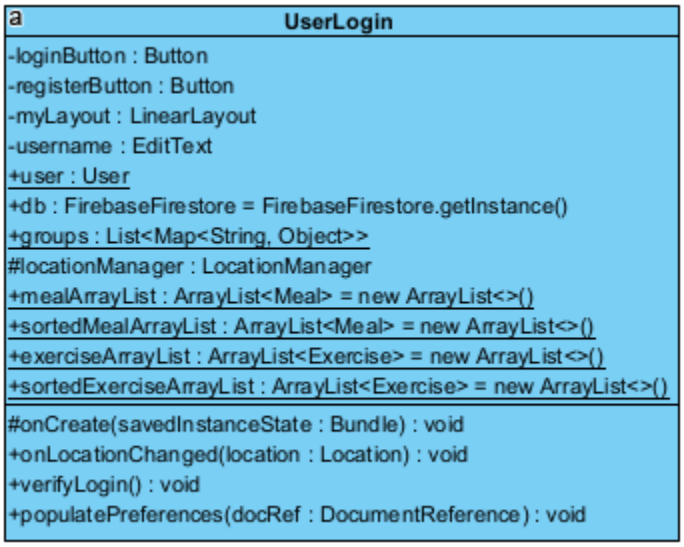

a
MealsTab2Fragment
+db : FirebaseFirestore = FirebaseFirestore.getInstance()
+docRef : DocumentReference
+groups : List<Map<String, Object>>
+breakfastArrayList : ArrayList<Meal> = new ArrayList<>()
+lunchArrayList : ArrayList<Meal> = new ArrayList<>()
+dinnerArrayList : ArrayList<Meal> = new ArrayList<>()
+snackArrayList : ArrayList<Meal> = new ArrayList<>()
+currentFragmentLocalDate : LocalDate
+todayMealArrayList : ArrayList<Meal> = new ArrayList<>()
-listViewBreakfast : ListView
-listViewLunch : ListView
-listViewDinner : ListView
-listViewSnack : ListView
~root : View
+MealsTab2Fragment(currentFragmentLocalDate : LocalDate)
+MealsTab2Fragment()
+onCreateView(inflater : LayoutInflater, container : ViewGroup, savedInstanceState : Bundle) : View
+modifyListViewSize(listview : ListView) : void
+populateMealLog(docRef : DocumentReference) : void
-selectSpinnerValue(spinner : Spinner, myString : String) : void
+applyDim(parent : ViewGroup, dimAmount : float) : void
+clearDim(parent : ViewGroup) : void


DemoCollectionAdapter
+DemoCollectionAdapter(fragment : Fragment)
+createFragment(position : int) : Fragment
+getItemCount() : int
demoCollectionAdapter
DayFragment
-binding : FragmentDayBinding
~tabLayout : TabLayout
~viewPagerMeals : ViewPager2
~mealsChild : MealsTab2Fragment
~workoutsChild : Fragment
+currentFragmentLocalDate : LocalDate
~demoCollectionAdapter : DemoCollectionAdapter
+DayFragment(currentFragmentLocalDate : LocalDate)
+DayFragment()
+onViewCreated(view : View, savedInstanceState : Bundle) : void
+onCreateView(inflater : LayoutInflater, container : ViewGroup, savedInstanceState : Bundle) : View

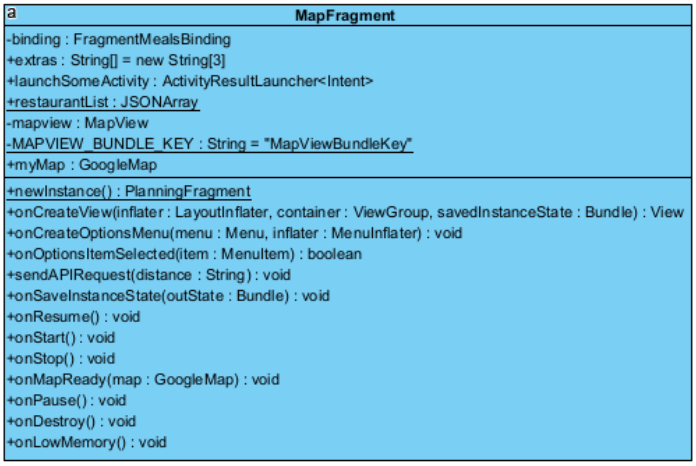

WorkoutsTab2Fragment
-binding : FragmentWorkoutsTab2Binding
+db : FirebaseFirestore = FirebaseFirestore.getInstance()
+docRef : DocumentReference
+groups : List<Map<String, Object>>
~root : View
~listViewCardio : ListView
~listViewWeightLifting : ListView
~listViewYoga : ListView
~listViewExerciseLog : ListView
-cardioArrayList : ArrayList<Exercise> = new ArrayList<>()
-weightLiftingArrayList : ArrayList<Exercise> = new ArrayList<>()
-yogaArrayList : ArrayList<Exercise> = new ArrayList<>()
-todayExerciseArrayList : ArrayList<Exercise> = new ArrayList<>()
+currentFragmentLocalDate : LocalDate
+WorkoutsTab2Fragment()
+WorkoutsTab2Fragment(currentFragmentLocalDate : LocalDate)
+onCreateView(inflater : LayoutInflater, container : ViewGroup, savedInstanceState : Bundle) : View
+modifyListViewSize(listview : ListView) : void
+populateMealLog(docRef : DocumentReference) : void
-selectSpinnerValue(spinner : Spinner, myString : String) : void
+applyDim(parent : ViewGroup, dimAmount : float) : void
+clearDim(parent : ViewGroup) : void

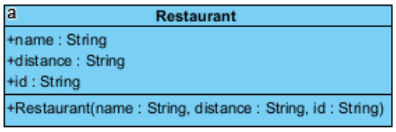
Login Screen UML

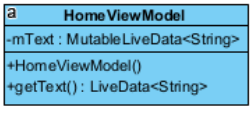


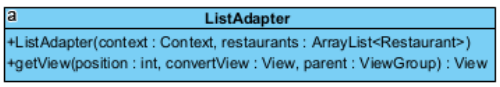


Map Screen UML

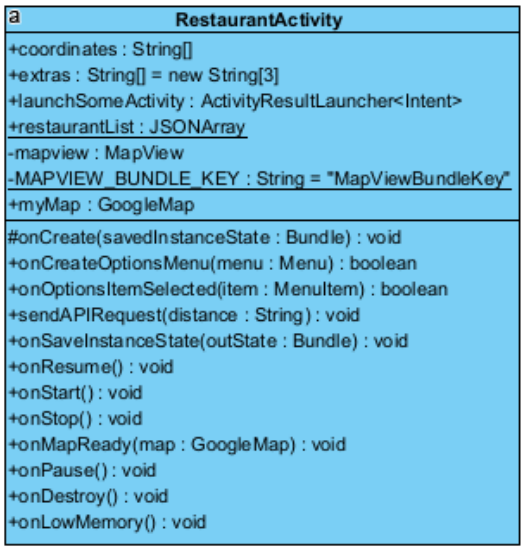


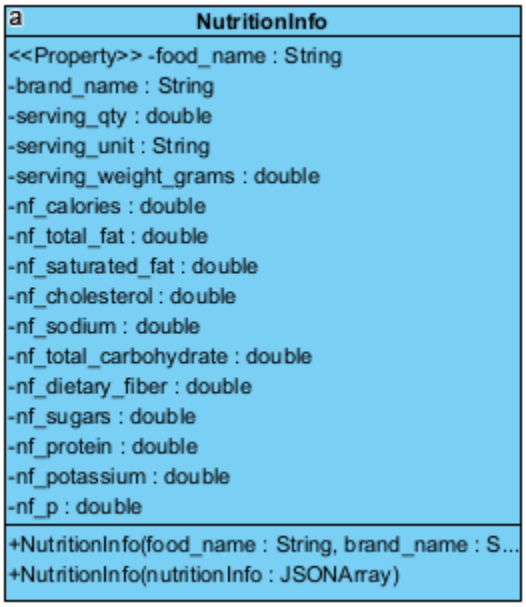




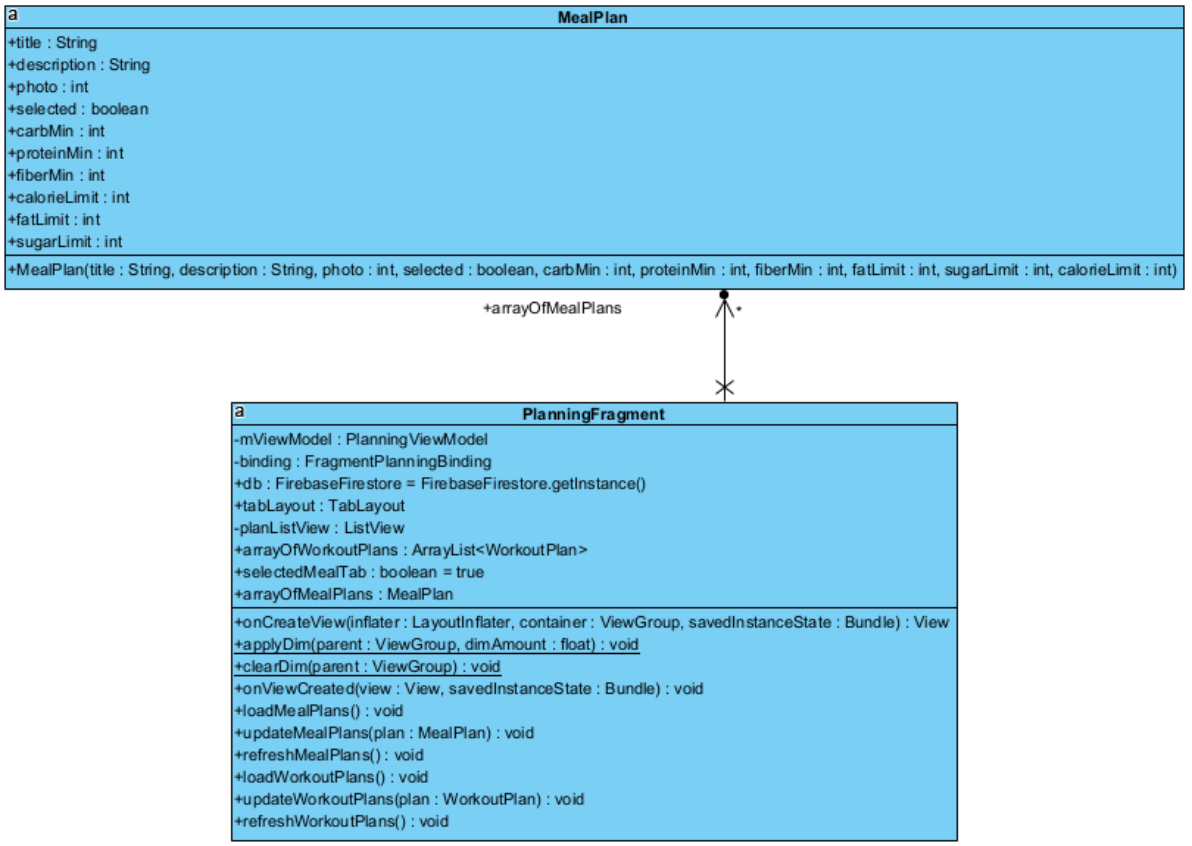


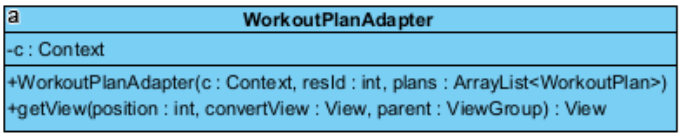

RestaurantMenu
-lv : ListView
~outerDC : DataContainer
~arrayAdapter : MealAdapter
~concatList : ArrayList<Meal> = new ArrayList<Meal>()
~nutritionFilteredList : ArrayList<Meal> = new ArrayList<Meal>()
~tempMeal : Meal
~thePhoto : Photo
-SearchViewModel : SearchViewModel
-binding : FragmentSearchBinding
-searchView : SearchView
-values : String[]
~launchSomeActivity : ActivityResultLauncher<Intent>
+db : FirebaseFirestore = FirebaseFirestore.getInstance()
+nutritionInfoBranded : JSONArray
+nutritionInfoCommon : JSONArray
+groups : List<Map<String, Object>>
+calories_lte : int = -1
+calories_gte : int = -1
+protein_lte : int = -1
+protein_gte : int = -1
+fat_lte : int = -1
+fat_gte : int = -1
+sugars_lte : int = -1
+sugars_gte : int = -1
+ref : DocumentReference
#onCreate(savedInstanceState : Bundle) : void
+onOptionsItemSelected(item : MenuItem) : boolean
+sendAPIRequest() : ArrayList<Meal>
+populatePreferences(docRef : DocumentReference) : void

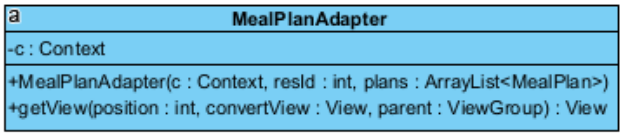

RestaurantActivity
+coordinates : String[]
+extras : String[] = new String[3]
+launchSomeActivity : ActivityResultLauncher<Intent>
+restaurantList : JSONArray
-mapview : MapView
-MAPVIEW_BUNDLE_KEY : String = "MapViewBundleKey"
+myMap : GoogleMap
#onCreate(savedInstanceState : Bundle) : void
+onCreateOptionsMenu(menu : Menu) : boolean
+onOptionsItemSelected(item : MenuItem) : boolean
+sendAPIRequest(distance : String) : void
+onSaveInstanceState(outState : Bundle) : void
+onResume() : void
+onStart() : void
+onStop() : void
+onMapReady(map : GoogleMap) : void
+onPause() : void
+onDestroy() : void
+onLowMemory() : void

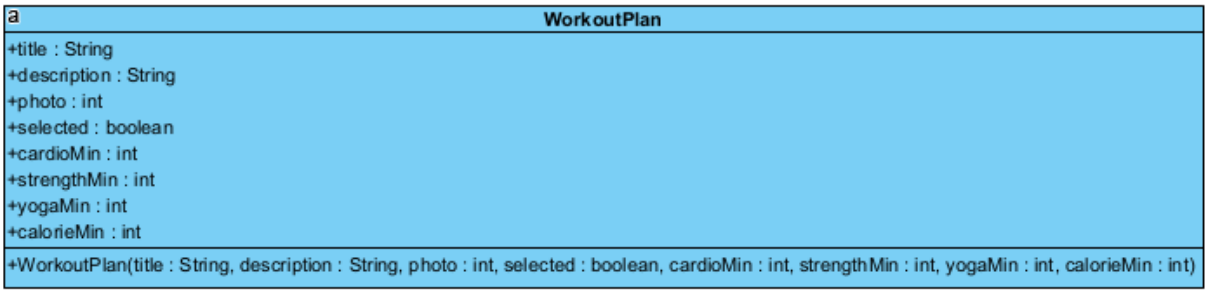

NutritionInfo
<<Property>> -food_name : String
-brand_name : String
-serving_qty : double
-serving_unit : String
-serving_weight_grams : double
-nf_calories : double
-nf_total_fat : double
-nf_saturated_fat : double
-nf_cholesterol : double
-nf_sodium : double
-nf_total_carbohydrate : double
-nf_dietary_fiber : double
-nf_sugars : double
-nf_protein : double
-nf_potassium : double
-nf_p : double
+NutritionInfo(food_name : String, brand_name : S...
+NutritionInfo(nutritionInfo : JSONArray)

Planning Screen UML

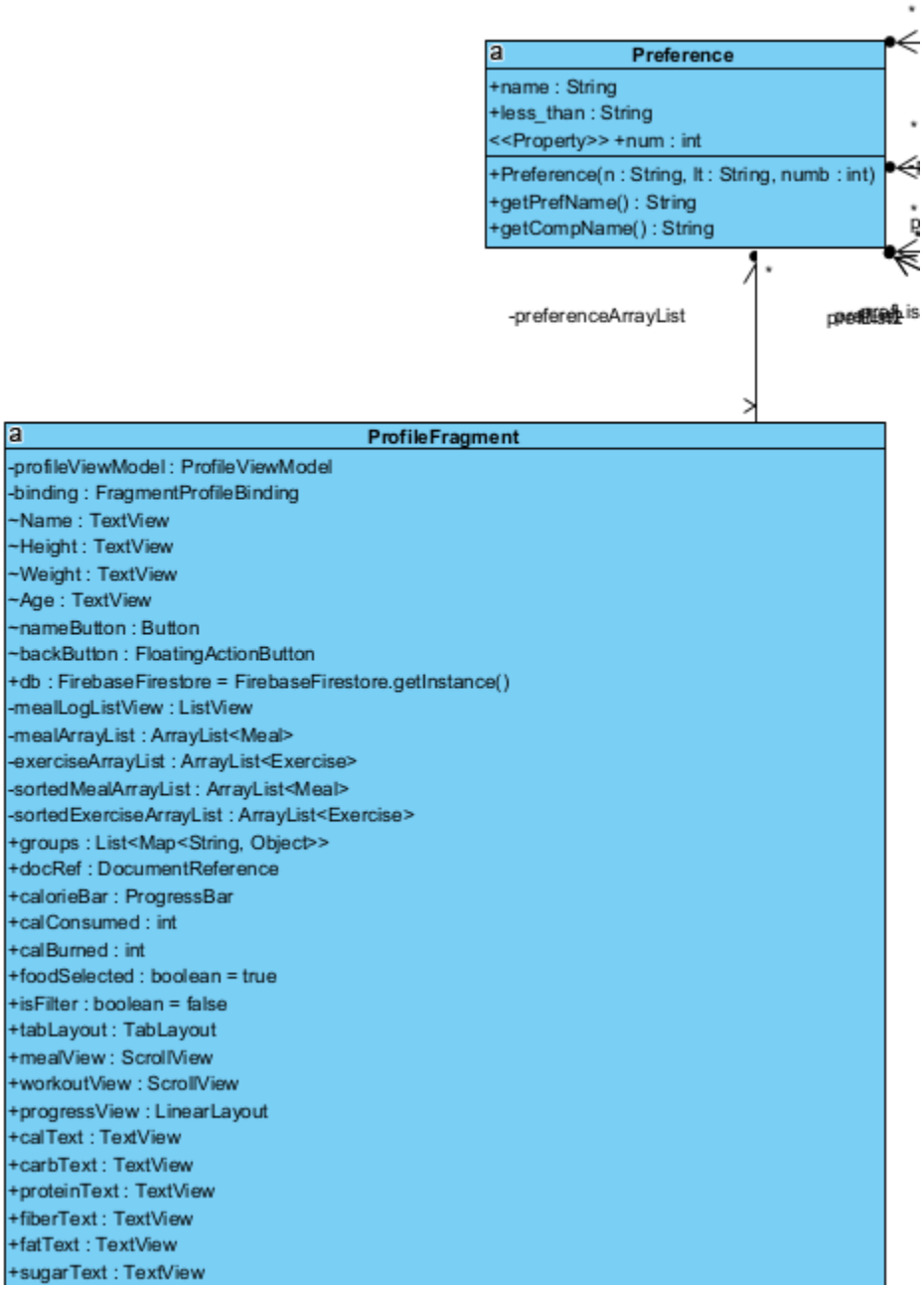




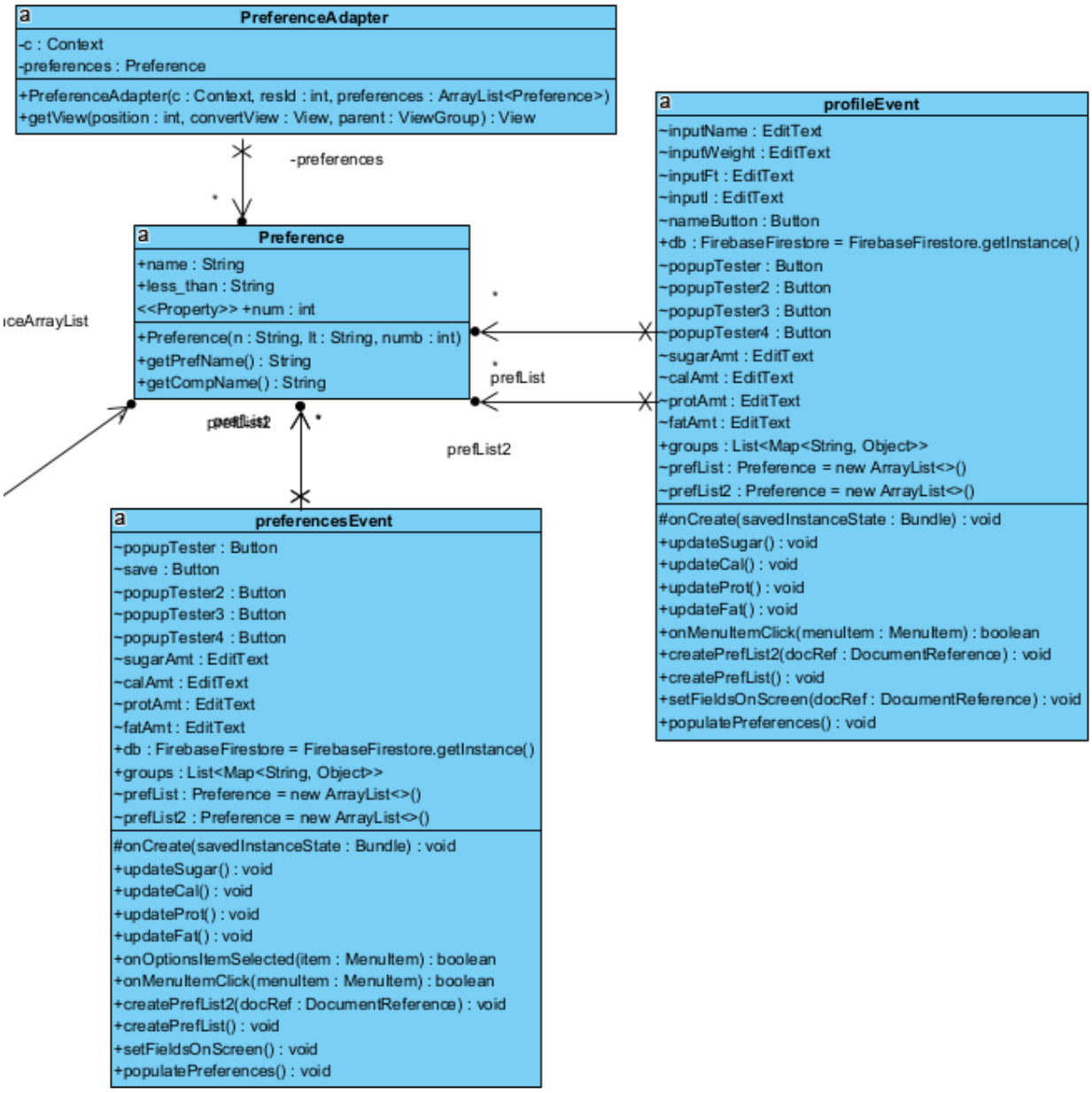


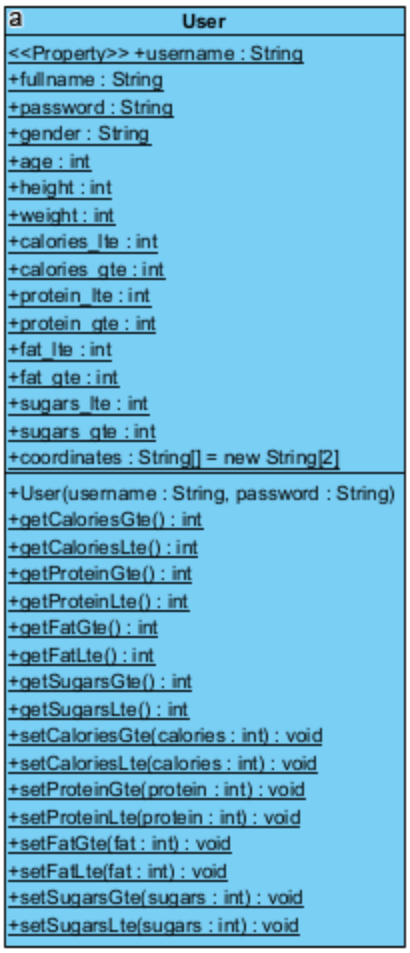


Profile Screen UML

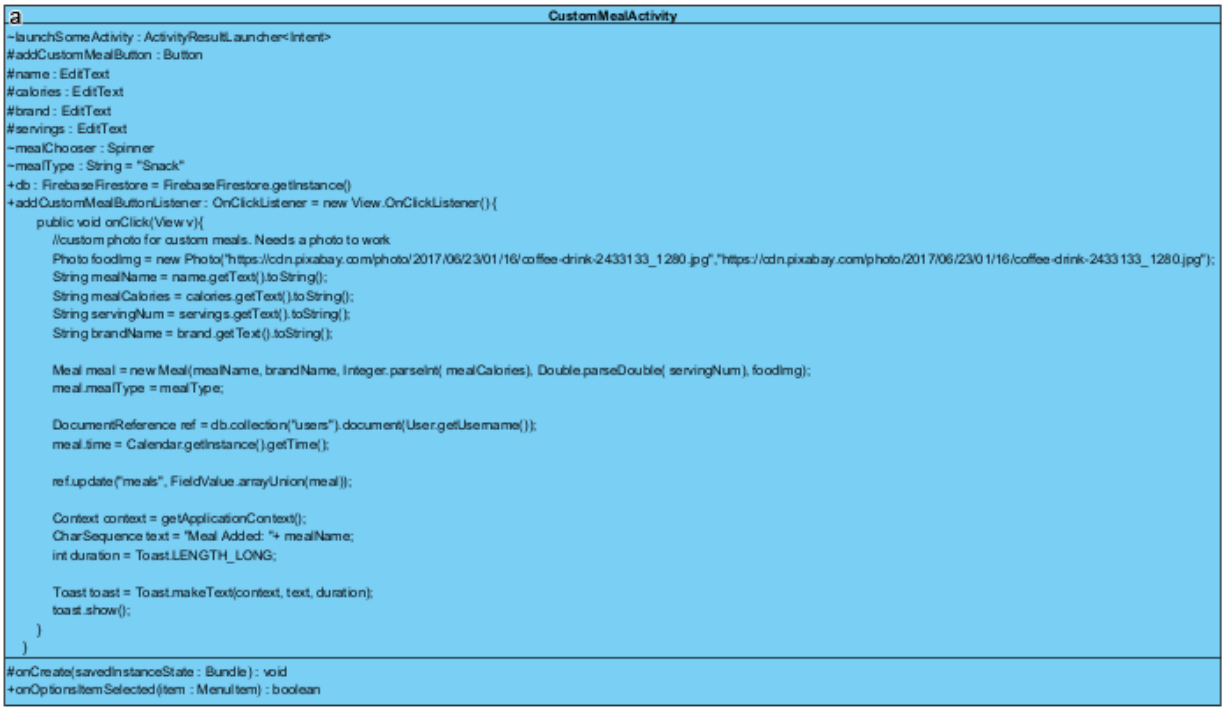


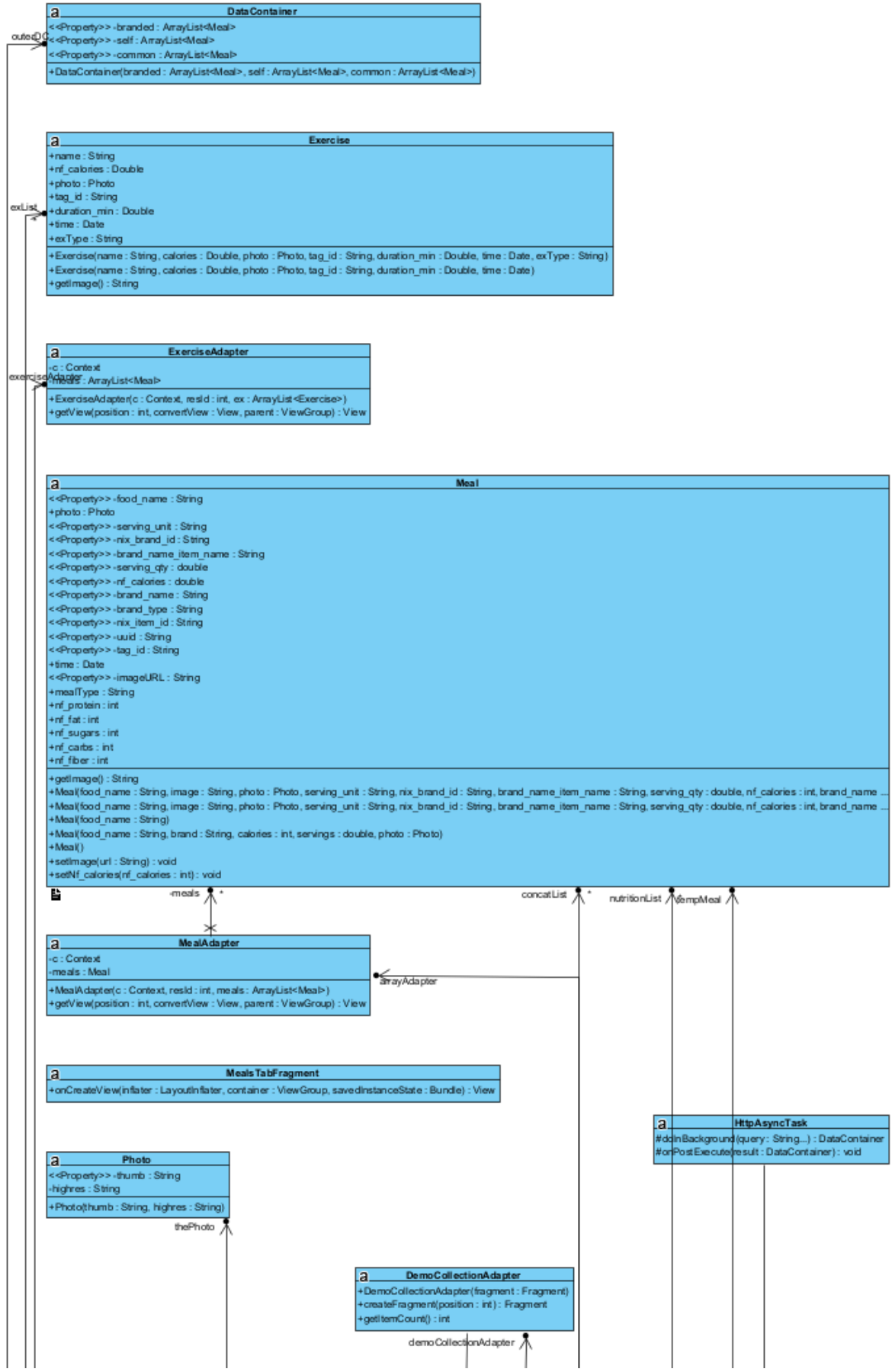

+calBar : CircularProgressIndicator
+carbBar : CircularProgressIndicator
+proteinBar : CircularProgressIndicator
+fiberBar : CircularProgressIndicator
+fatBar : CircularProgressIndicator
+sugarBar : CircularProgressIndicator
+calProgress : double
+carbProgress : double
+proteinProgress : double
+fiberProgress : double
+fatProgress : double
+sugarProgress : double
+calCount : int
+carbCount : int
+proteinCount : int
+fiberCount : int
+fatCount : int
+sugarCount : int
+caloricText : TextView
+strengthText : TextView
+yogaText : TextView
+cardioText : TextView
+caloricBar : CircularProgressIndicator
+strengthBar : CircularProgressIndicator
+yogaBar : CircularProgressIndicator
+cardioBar : CircularProgressIndicator
+caloricProgress : double
+strengthProgress : double
+yogaProgress : double
+cardioProgress : double
+caloricCount : int
+strengthCount : int
+yogaCount : int
+cardioCount : int
+netText : TextView
+netConsumedTitle : TextView
+netBurnedTitle : TextView
+textProgress : TextProgressBar
+curCalCount : int
+curBurnCount : int
-preferenceArrayList : Preference
+mealplan : MealPlan
+workoutplan : WorkoutPlan
+onCreateView(inflater : LayoutInflater, container : ViewGroup, savedInstanceState : Bundle) : View
+calculateWeeklyRange() : Pair<Calendar, Calendar>
+matchesDate(w : Calendar, t : Calendar) : boolean
+populateNetCalorieBar() : void
+populateWorkoutBars() : void
+populateMealBars() : void
+getFireStoreData() : void
+onDestroyView() : void
+populateProfileInfo(docRef : DocumentReference) : void
+onCreateOptionsMenu(menu : Menu, inflater : MenuInflater) : void
+onOptionsItemSelected(item : MenuItem) : boolean
+onViewCreated(view : View, savedInstanceState : Bundle) : void

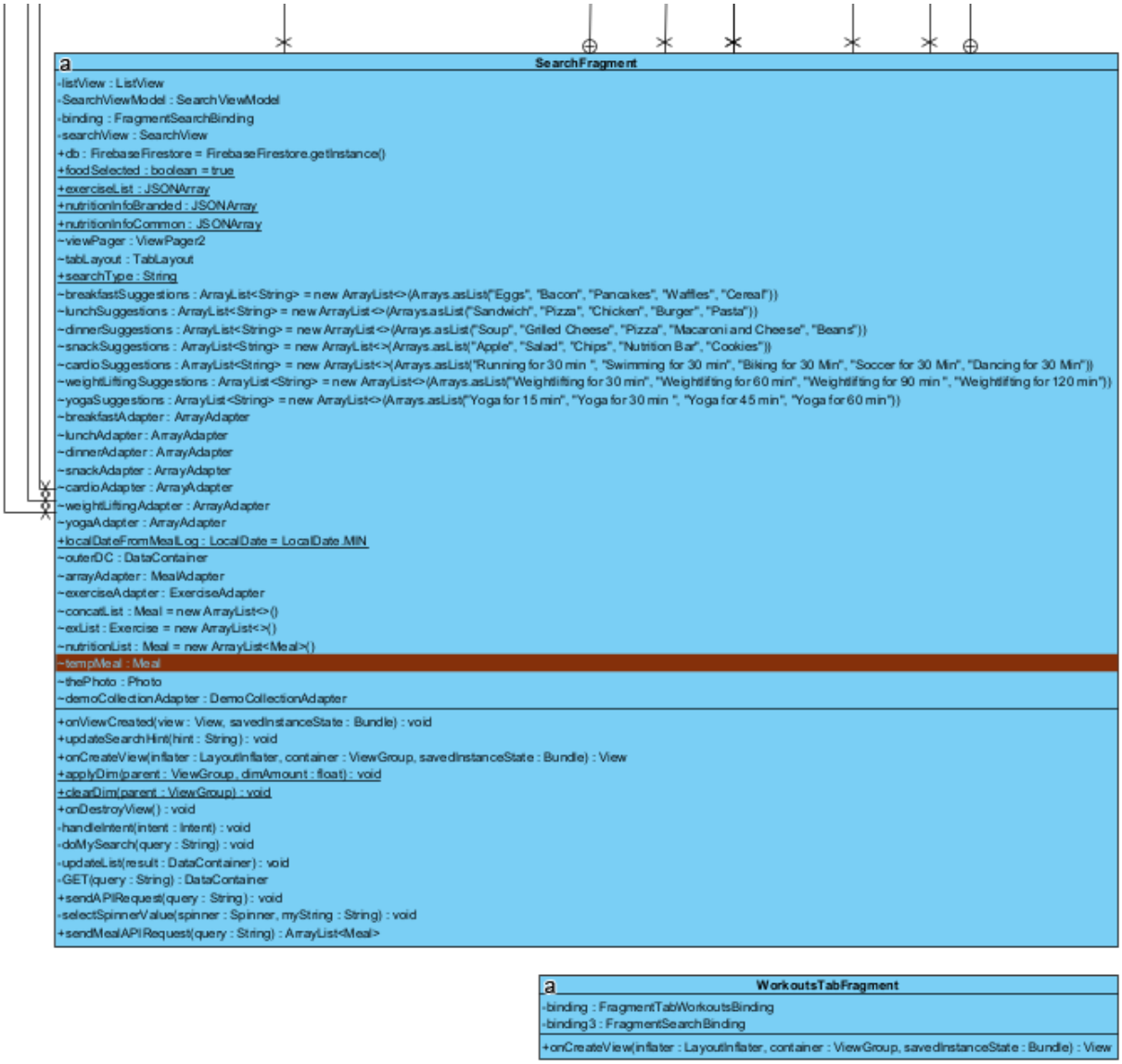

PreferenceAdapter
-c : Context
-preferences : Preference
+PreferenceAdapter(c : Context, resId : int, preferences : ArrayList<Preference>)
+getView(position : int, convertView : View, parent : ViewGroup) : View
-preferences
Preference
+name : String
+less_than : String
<<Property>> +num : int
+Preference(n : String, lt : String, numb : int)
+getPrefName() : String
+getCompName() : String
profileEvent
~inputName : EditText
~inputWeight : EditText
~inputFt : EditText
~inputI : EditText
~nameButton : Button
+db : FirebaseFirestore = FirebaseFirestore.getInstance()
~popupTester : Button
~popupTester2 : Button
~popupTester3 : Button
~popupTester4 : Button
~sugarAmt : EditText
~calAmt : EditText
~protAmt : EditText
~fatAmt : EditText
+groups : List<Map<String, Object>>
~prefList : Preference = new ArrayList<>()
~prefList2 : Preference = new ArrayList<>()
#onCreate(savedInstanceState : Bundle) : void
+updateSugar() : void
+updateCal() : void
+updateProt() : void
+updateFat() : void
+onMenuItemClick(menuItem : MenuItem) : boolean
+createPrefList2(docRef : DocumentReference) : void
+createPrefList() : void
+setFieldsOnScreen(docRef : DocumentReference) : void
+populatePreferences() : void
prefList
prefList2
preferencesEvent
~popupTester : Button
~save : Button
~popupTester2 : Button
~popupTester3 : Button
~popupTester4 : Button
~sugarAmt : EditText
~calAmt : EditText
~protAmt : EditText
~fatAmt : EditText
+db : FirebaseFirestore = FirebaseFirestore.getInstance()
+groups : List<Map<String, Object>>
~prefList : Preference = new ArrayList<>()
~prefList2 : Preference = new ArrayList<>()
#onCreate(savedInstanceState : Bundle) : void
+updateSugar() : void
+updateCal() : void
+updateProt() : void
+updateFat() : void
+onOptionsItemSelected(item : MenuItem) : boolean
+onMenuItemClick(menuItem : MenuItem) : boolean
+createPrefList2(docRef : DocumentReference) : void
+createPrefList() : void
+setFieldsOnScreen() : void
+populatePreferences() : void


a
User
<<Property>> +username : String
+fullname : String
+password : String
+gender : String
+age : int
+height : int
+weight : int
+calories_lte : int
+calories_gte : int
+protein_lte : int
+protein_gte : int
+fat_lte : int
+fat_gte : int
+sugars_lte : int
+sugars_gte : int
+coordinates : String[] = new String[2]
+User(username : String, password : String)
+getCaloriesGte() : int
+getCaloriesLte() : int
+getProteinGte() : int
+getProteinLte() : int
+getFatGte() : int
+getFatLte() : int
+getSugarsGte() : int
+getSugarsLte() : int
+setCaloriesGte(calories : int) : void
+setCaloriesLte(calories : int) : void
+setProteinGte(protein : int) : void
+setProteinLte(protein : int) : void
+setFatGte(fat : int) : void
+setFatLte(fat : int) : void
+setSugarsGte(sugars : int) : void
+setSugarsLte(sugars : int) : void

Search Screen UML




DataContainer
<<Property>> -branded : ArrayList<Meal>
<<Property>> -self : ArrayList<Meal>
<<Property>> -common : ArrayList<Meal>
+DataContainer(branded : ArrayList<Meal>, self : ArrayList<Meal>, common : ArrayList<Meal>)
outerDC
Exercise
+name : String
+nf_calories : Double
+photo : Photo
+tag_id : String
+duration_min : Double
+time : Date
+exType : String
+Exercise(name : String, calories : Double, photo : Photo, tag_id : String, duration_min : Double, time : Date, exType : String)
+Exercise(name : String, calories : Double, photo : Photo, tag_id : String, duration_min : Double, time : Date)
+getImage() : String
exList
ExerciseAdapter
-c : Context
-meals : ArrayList<Meal>
+ExerciseAdapter(c : Context, resId : int, ex : ArrayList<Exercise>)
+getView(position : int, convertView : View, parent : ViewGroup) : View
exerciseAdapter
Meal
<<Property>> -food_name : String
+photo : Photo
<<Property>> -serving_unit : String
<<Property>> -nix_brand_id : String
<<Property>> -brand_name_item_name : String
<<Property>> -serving_qty : double
<<Property>> -nf_calories : double
<<Property>> -brand_name : String
<<Property>> -brand_type : String
<<Property>> -nix_item_id : String
<<Property>> -uuid : String
<<Property>> -tag_id : String
+time : Date
<<Property>> -imageURL : String
+mealType : String
+nf_protein : int
+nf_fat : int
+nf_sugars : int
+nf_carbs : int
+nf_fiber : int
+getImage() : String
+Meal(food_name : String, image : String, photo : Photo, serving_unit : String, nix_brand_id : String, brand_name_item_name : String, serving_qty : double, nf_calories : int, brand_name ...
+Meal(food_name : String, image : String, photo : Photo, serving_unit : String, nix_brand_id : String, brand_name_item_name : String, serving_qty : double, nf_calories : int, brand_name ...
+Meal(food_name : String)
+Meal(food_name : String, brand : String, calories : int, servings : double, photo : Photo)
+Meal()
+setImage(url : String) : void
+setNf_calories(nf_calories : int) : void
meals
*
concatList
*
nutritionList
tempMeal
MealAdapter
-c : Context
-meals : Meal
+MealAdapter(c : Context, resId : int, meals : ArrayList<Meal>)
+getView(position : int, convertView : View, parent : ViewGroup) : View
arrayAdapter
MealsTabFragment
+onCreateView(inflater : LayoutInflater, container : ViewGroup, savedInstanceState : Bundle) : View
HttpAsyncTask
#doInBackground(query : String...) : DataContainer
#onPostExecute(result : DataContainer) : void
Photo
<<Property>> -thumb : String
-highres : String
+Photo(thumb : String, highres : String)
thePhoto
DemoCollectionAdapter
+DemoCollectionAdapter(fragment : Fragment)
+createFragment(position : int) : Fragment
+getItemCount() : int
demoCollectionAdapter



4c Data Dictionary for Any Included Models

Net Calories = Calories Consumed - Calories Burned

Workout Plan = Workout Goals + Calorie Goals

Workout = Type + Calories + Timestamp

Meal Plan = Nutritional Goals + Caloric Limits + Nutritional Limits

Meal = Nutrients + Calories + Timestamp

User = Biodata + Preferences + Location

Restaurant = Name + Distance + ID

Meal Log List = Entire meal history

Workout Log List = Entire workout history

Meal List View = List of meals with nutritional and caloric information

Workout List View = List of meals with duration and caloric information

Daily Meal List View = List of meals consumed on specified date

Daily Workout List View = List of workouts accomplished on specified date

Meals Search Result = Meal List View + Preferences

Workout Search Result = Workout List View + Biodata

Location = Latitude + Longitude

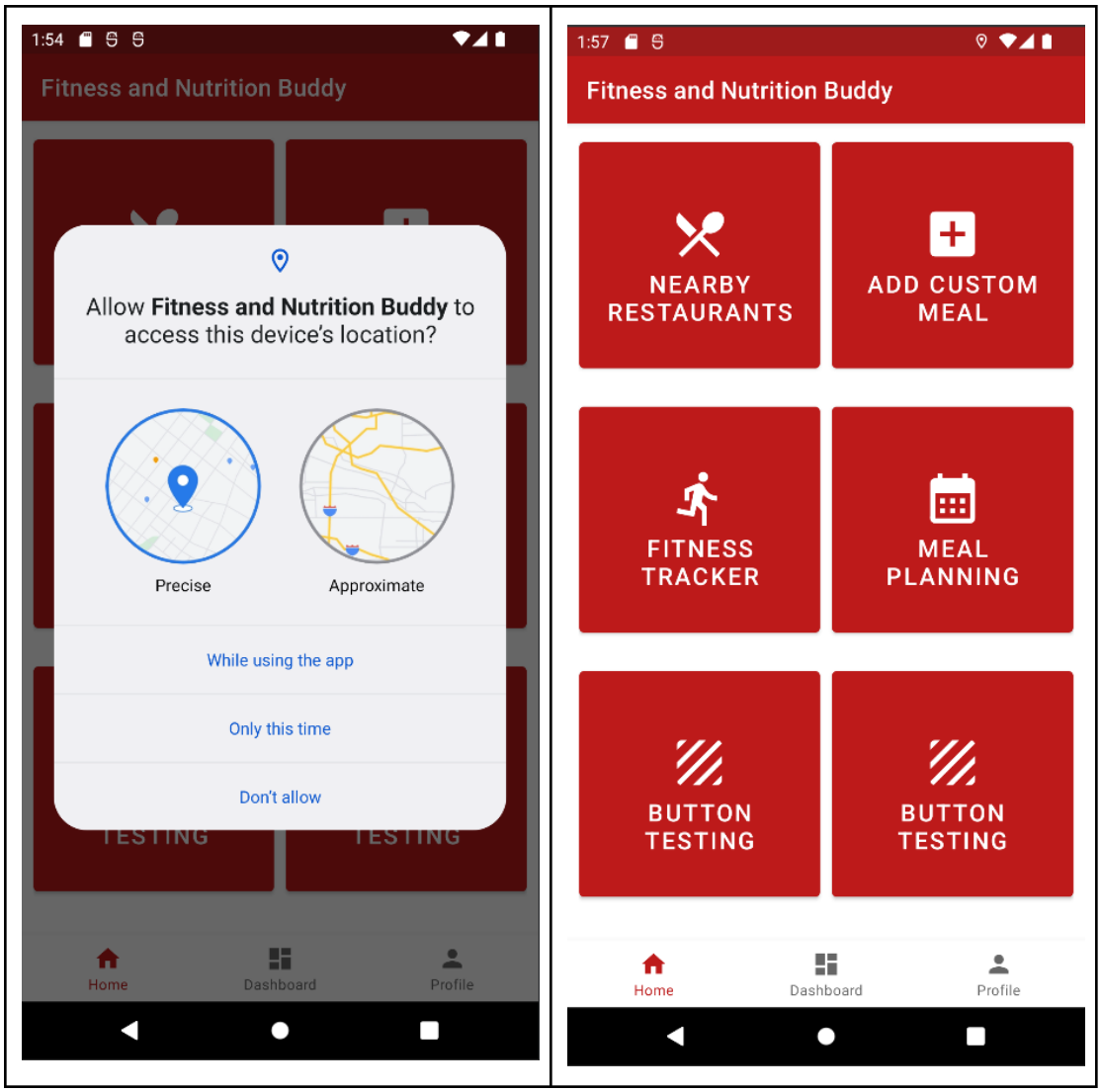
II Project Deliverables

The project deliverables were three releases throughout the semester. The first release focused on getting the NutritionIX API working for tracking meals. The second release was focused on setting up the FireStore database for the application and saving user meals and displaying them to the user and filtering restaurant menu resulta based on user preferences. The third release was focused on adding meal plans, workout plans, track more nutrients from the users meals, and make the application function better and make it easier to use.

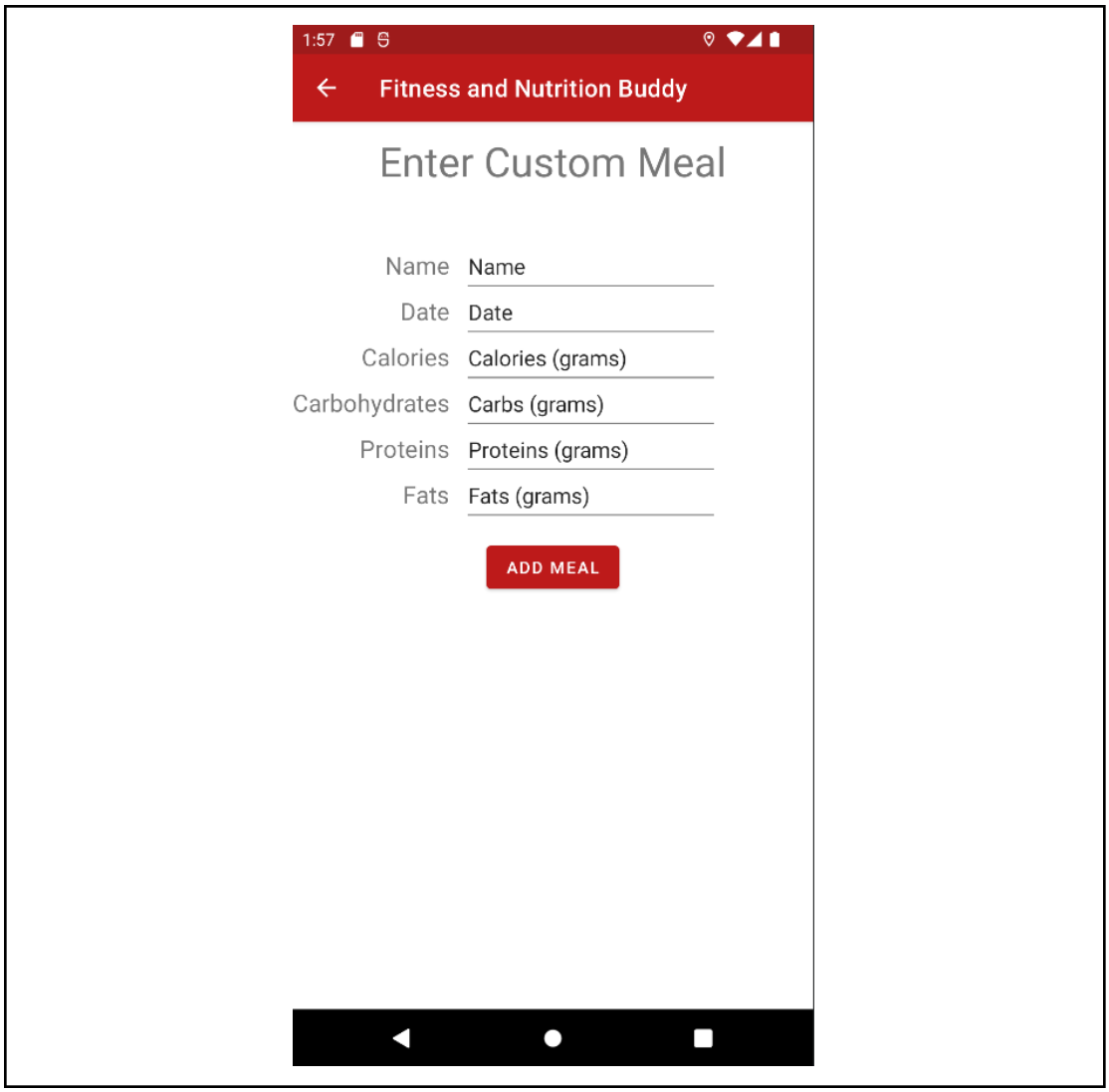
1 First Release

The first release occurred on friday February 25th.

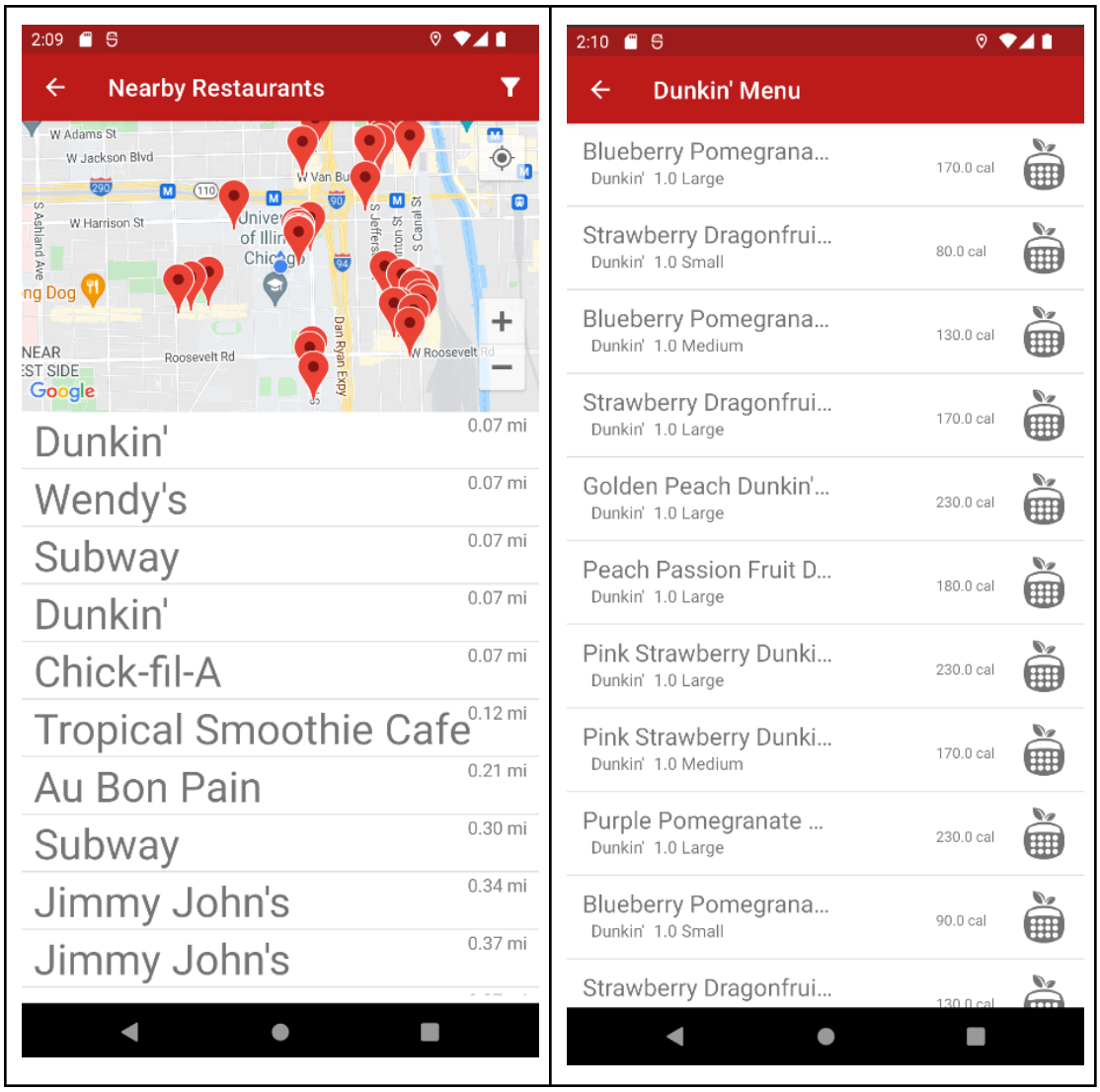
The first release was largely centered around finding restaurants near the user and showing them the food items available at that location as well as the calorie count for those food items. Below are images of various screens from the android application and further discussion about them.



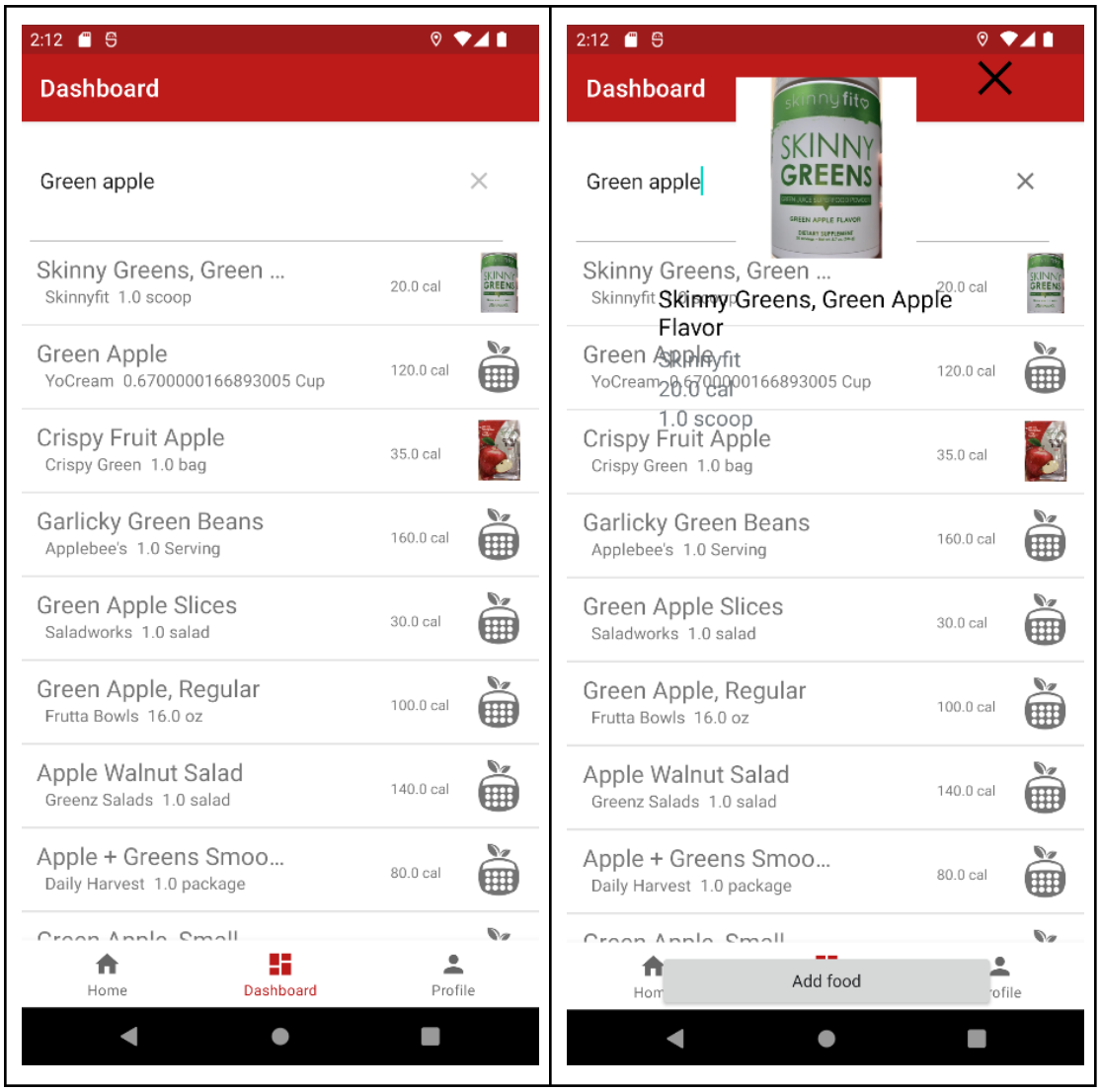
Above we can see the screen the user is welcomed to when launching the app. The application asks for permission to access the user's location. The application has three tabs which are home, dashboard and profile. Above we see home where the only buttons which have features tied to them as of the first release are “Nearby Restaurants” and “Add Custom Meal”.



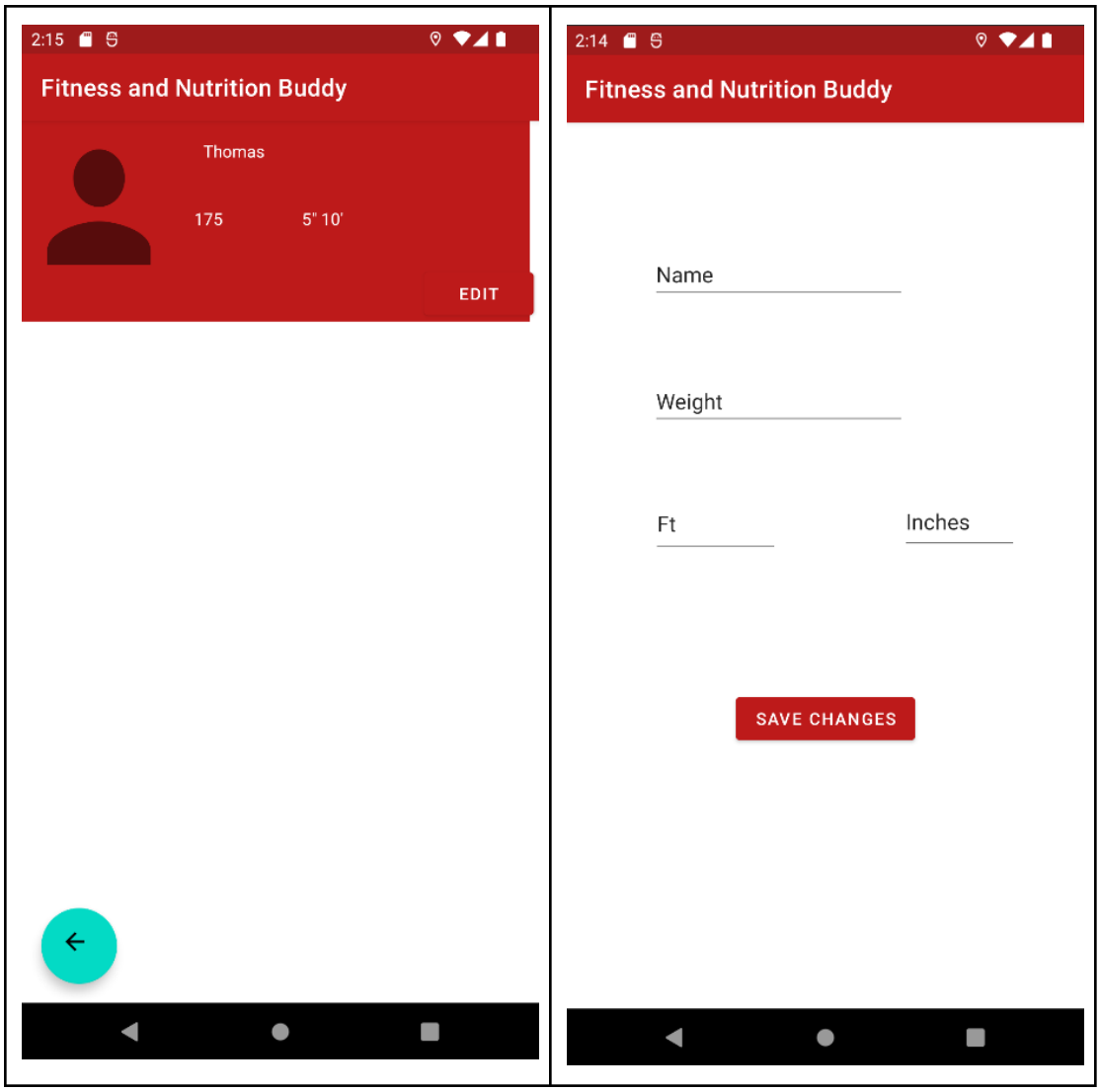
Above is the "Add Custom Meal" feature from the home page and as of the first release the fields can be updated but are not saved anywhere.



Next, as can be seen above is the "Nearby Restaurants" feature. This feature uses the user's location and displays a list of restaurants nearby. The list of restaurants is obtained by calling the NutritionIX V2 API. This API was used thoroughly in this application. After clicking on a restaurant from the list the user then sees a list of food items from that restaurant. Once again these food items are obtained by querying the NutritionIX API with the brand anime of the restaurant. The food menu list items show the name of the item, the brand, the serving size, calories in the item, and an image of the item if one exists in the system. If an image does not exist the API provides the filler item image as seen above on the right.



In the dashboard tab of the application is a search bar. This search bar can be used to query the NutritionIX API for any kind of food item. This can be done by typing in the search bar and hitting enter. The user is then shown a list of food items from NutritionIX. If a list item is clicked a box pops up with the information about that item and an add food button. The add food button does not add the food to anything as of the first release.

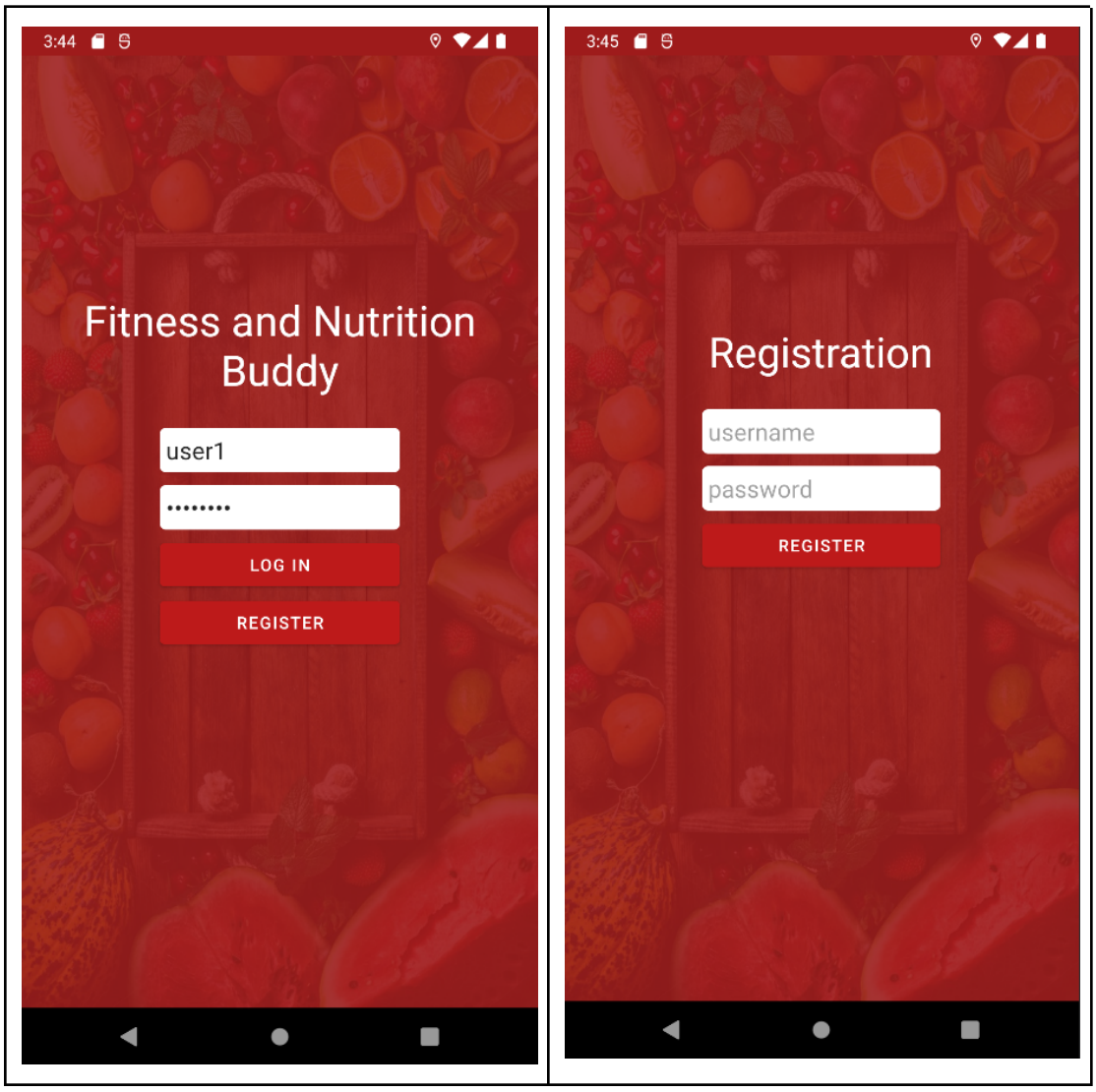


The final feature that was added was a profile screen that is accessible through a button in the profile tab. The profile screen has a name, weight, and height and these are editable through the edit button. These changes can be saved but will be lost on application shutdown as of the first release.

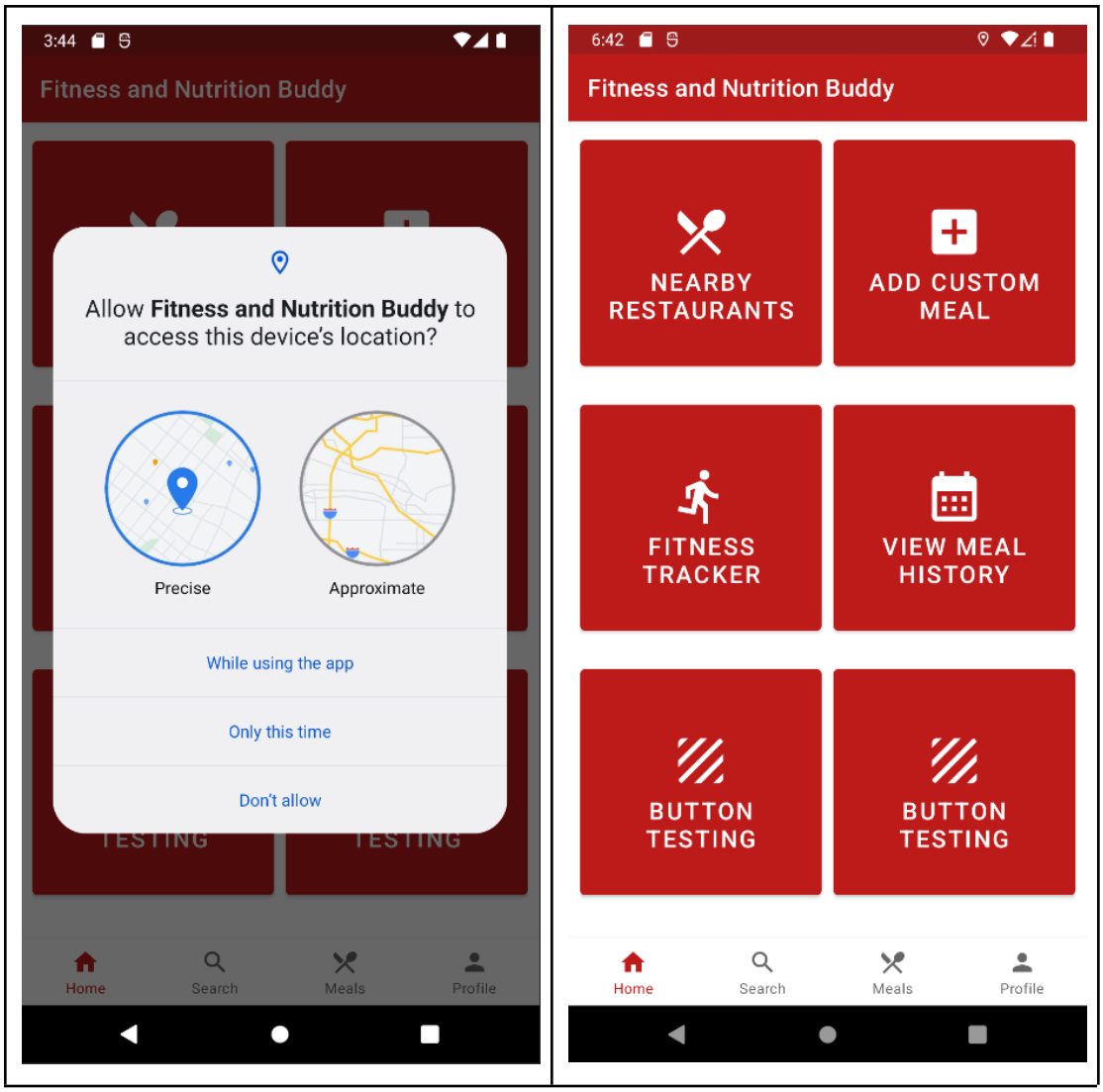
2 Second Release

The second release happened on Friday April 1st but the demo occurred on Monday April 4th due to some scheduling conflicts and the TAs flexibility.

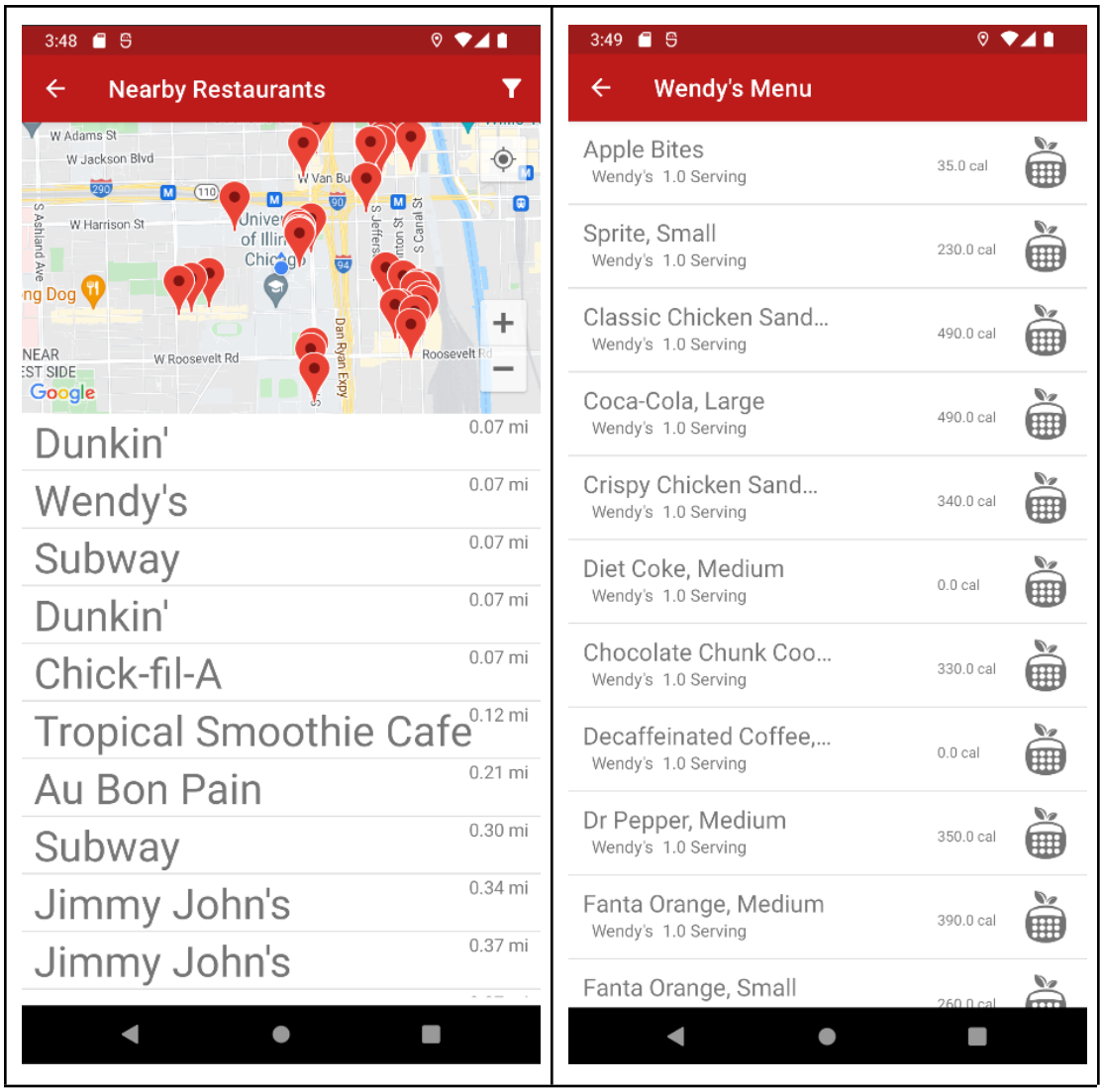
The second release focused on adding workout functionality to the application. Exercise could be added to the user's profile by using natural language search in the workouts section of the search tab. The profile would then keep track of the user's calories burned. The workouts are obtained through the same NutritionIX API. This release also contained a connection to a FireStore database so all users' meals, workouts, meal preferences, and information is saved for retrieval on application usage. This release also contains a login and registration page to allow for information to be saved for various users.



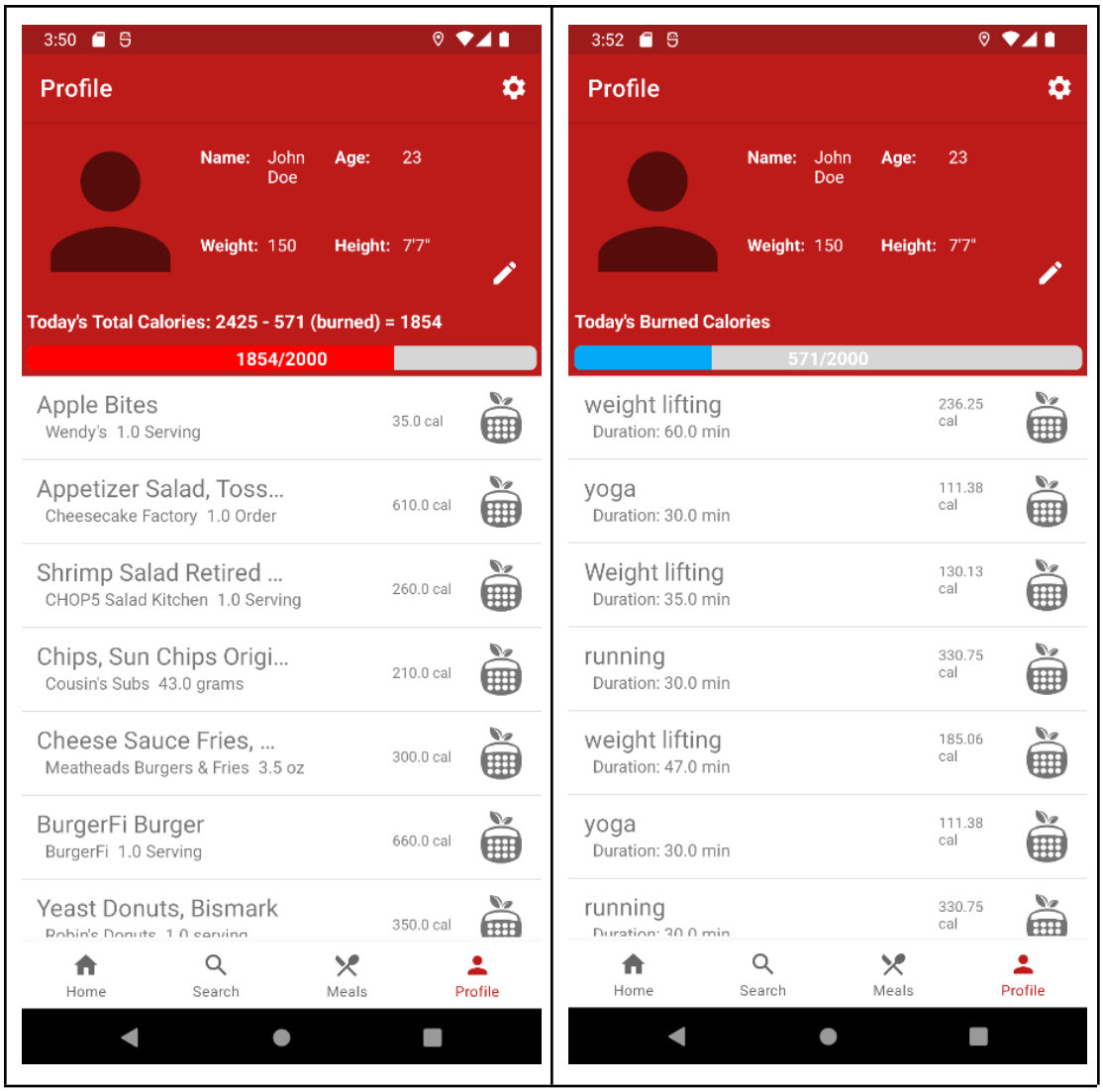
Above we can see what a user first sees when launching the android application. users have the option to login or register as a new user.



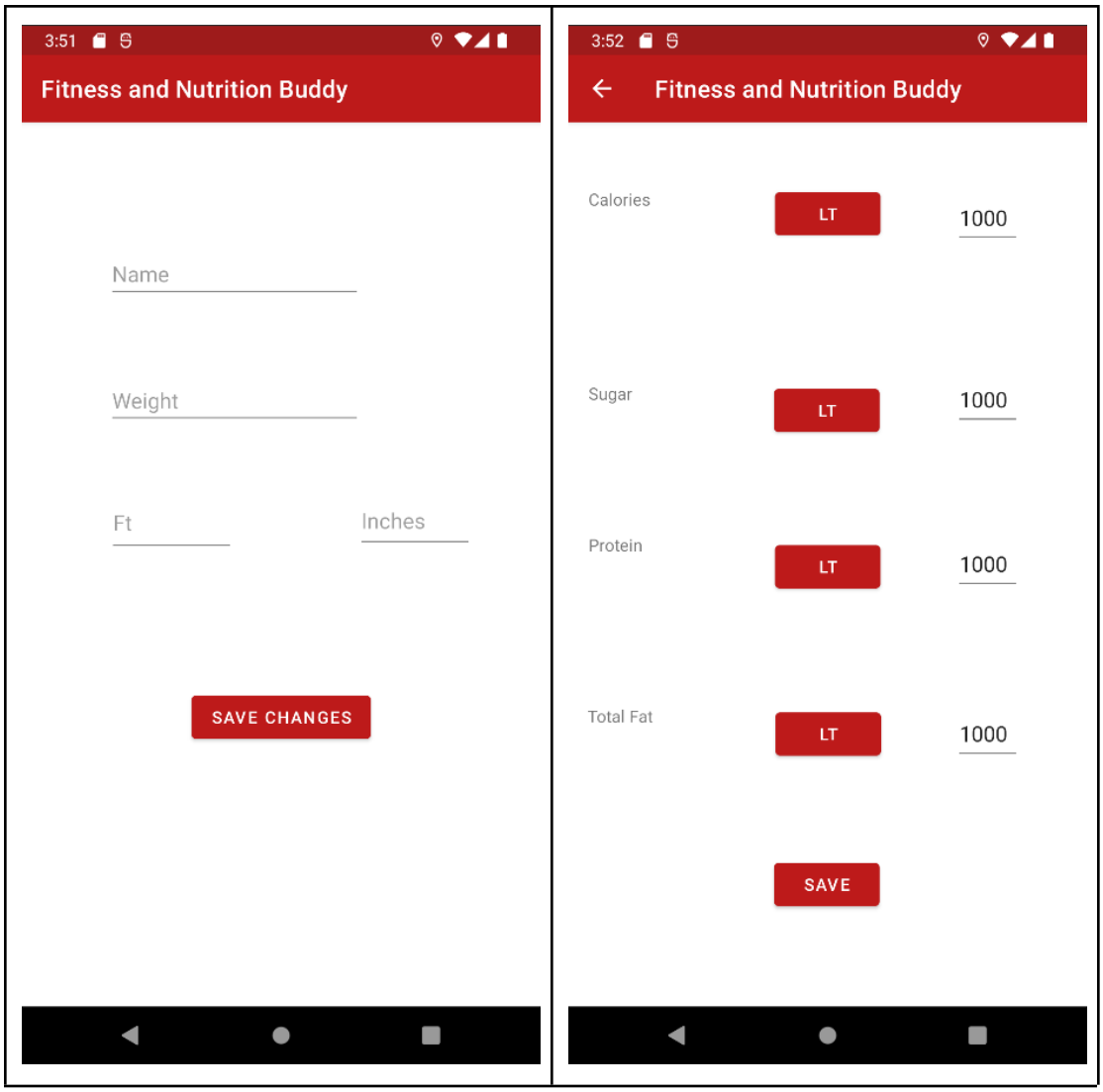
After the user logs in the application asks for location privileges if not previously granted. After that they see the home screen which contains the working functionality of "Nearby Restaurants", "Add Custom Meal", "Fitness Tracker", and "View Meal History". There is also a new tab on the bottom called Meals.



The restaurant list screen and the restaurant menu screen look the same however in this release the restaurant menu items that are shown are filtered based on the user's preferences that they set in their profile under preferences. The restaurant menu items can be filtered on calories, protein, fat, and sugars. This is done by sending a POST query to the NutritionIX API with the proper body with nutritional filtering information.

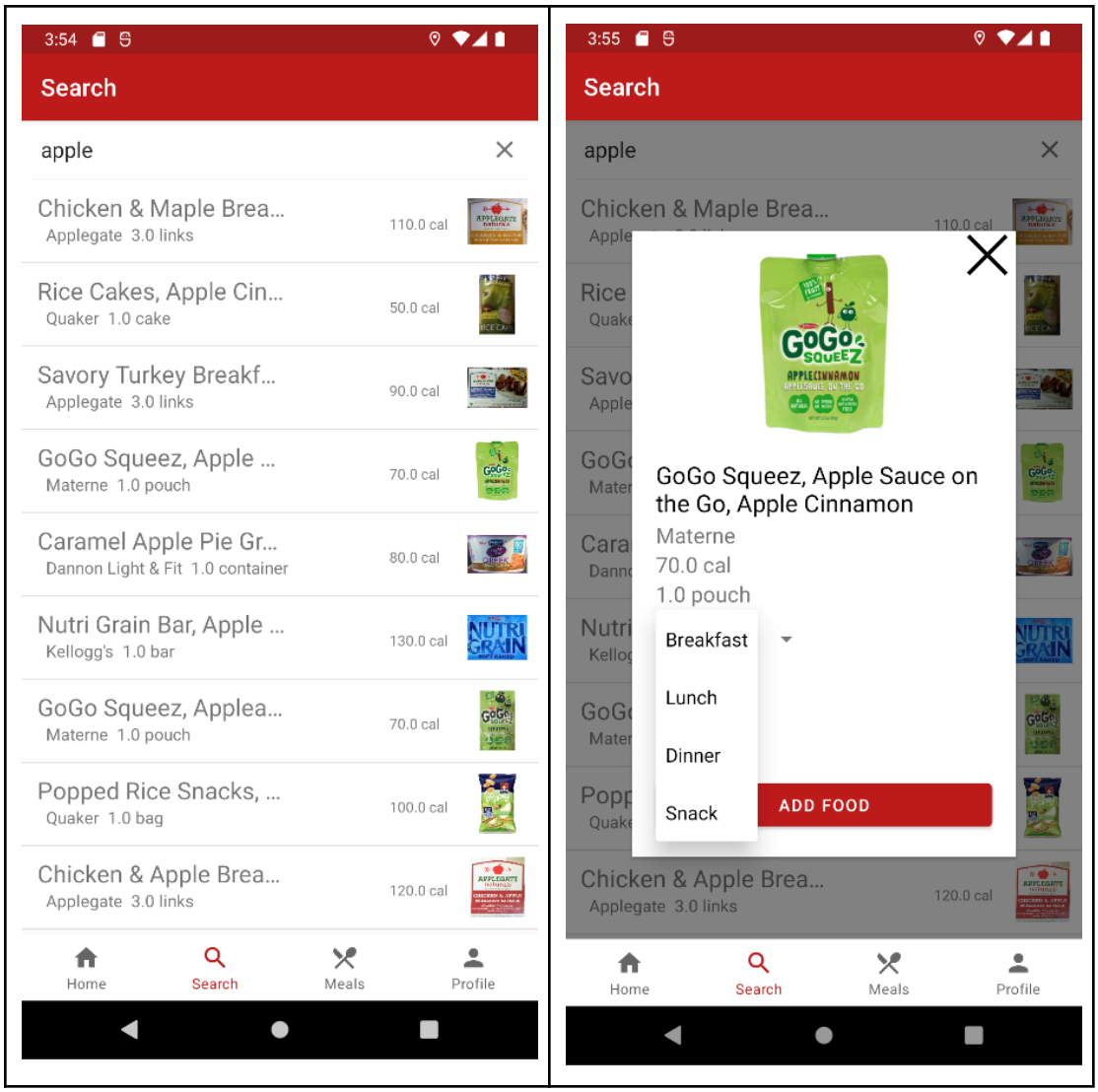


Now that the application is saving information such as meals and workout to the FireStore the meals and workouts are now shown in the meals and workouts section of the profile. These two screens can be switched between by clicking on the gear icon in the upper right and choosing meal log or Workout log. Most recent workouts and meals are shown at the top of their respective lists. In these screens both the calories taken in and burned are shown in a progress bar. As in release one the profile information is mostly the same with an addition of age.

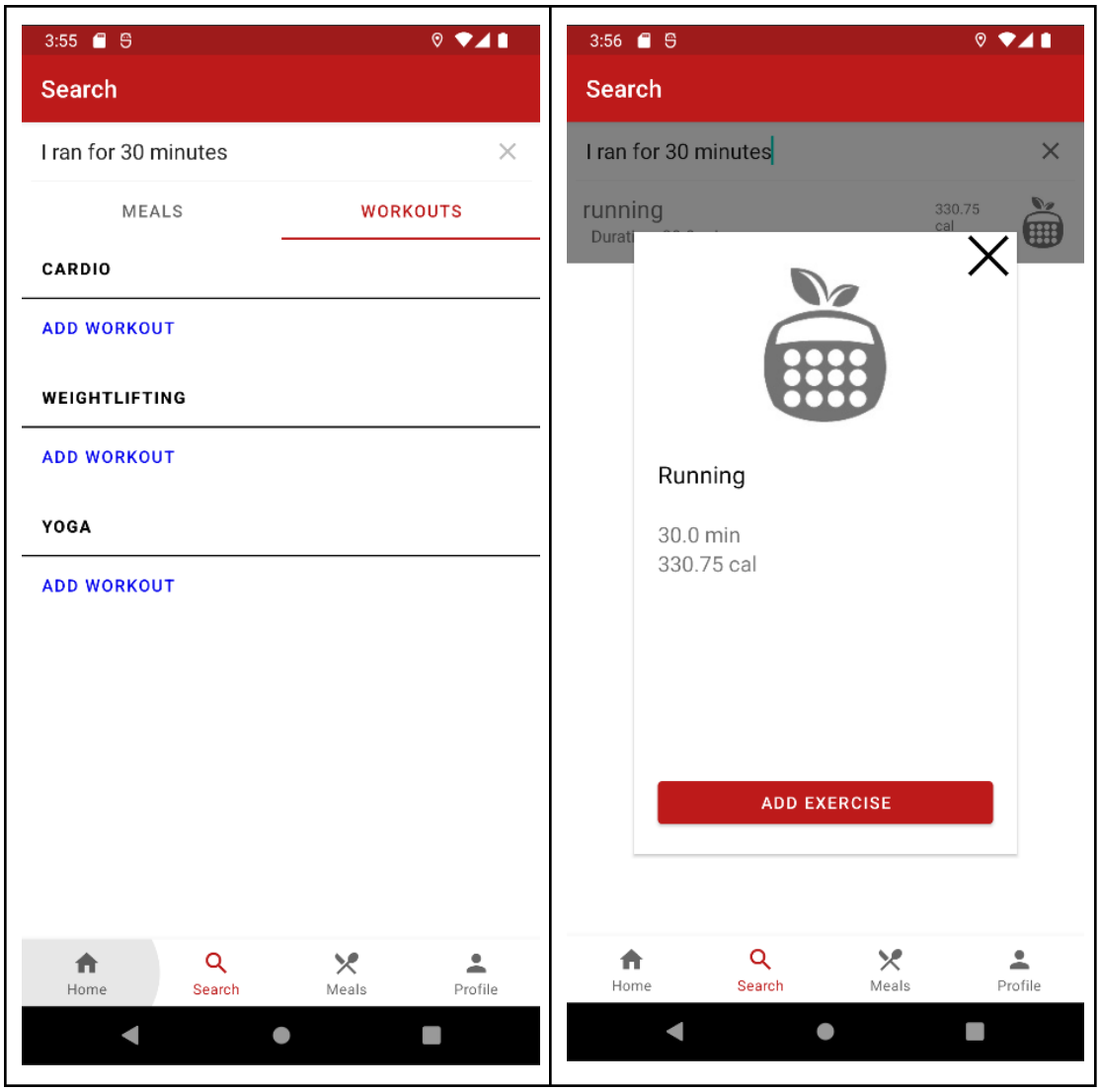


As is the same in release one the profile information is editable except for the age. There is a new activity to update preferences that is accessible through the gear icon on the profile page. In the preference activity users can choose if they want to filter restaurant menu items searched for in the nearby restaurants functionality by greater than or less than for calories, protein, total fat, and sugars. Users can enter a whole number for each piece of nutrition information and change the less than or greater than by clicking the red LT button and choosing from a drop down box. When the user presses save this information is stored in the FIreStore for that user's account.

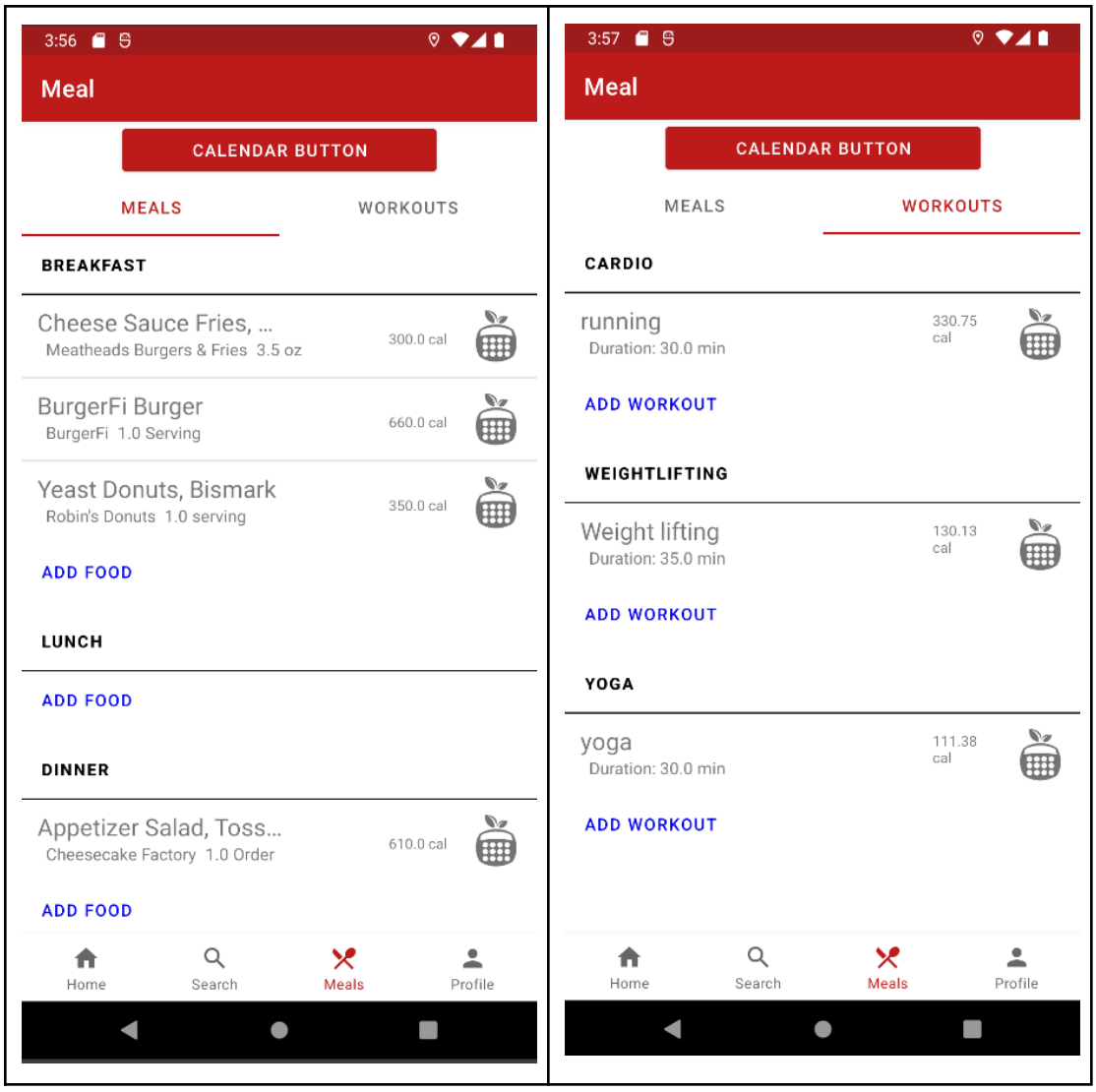
Users can search for meals in the search tab of the application. They can specify what meal they are eating the item for by clicking Add Food under each category of breakfast, lunch, dinner, or snack and then type in the search bar.



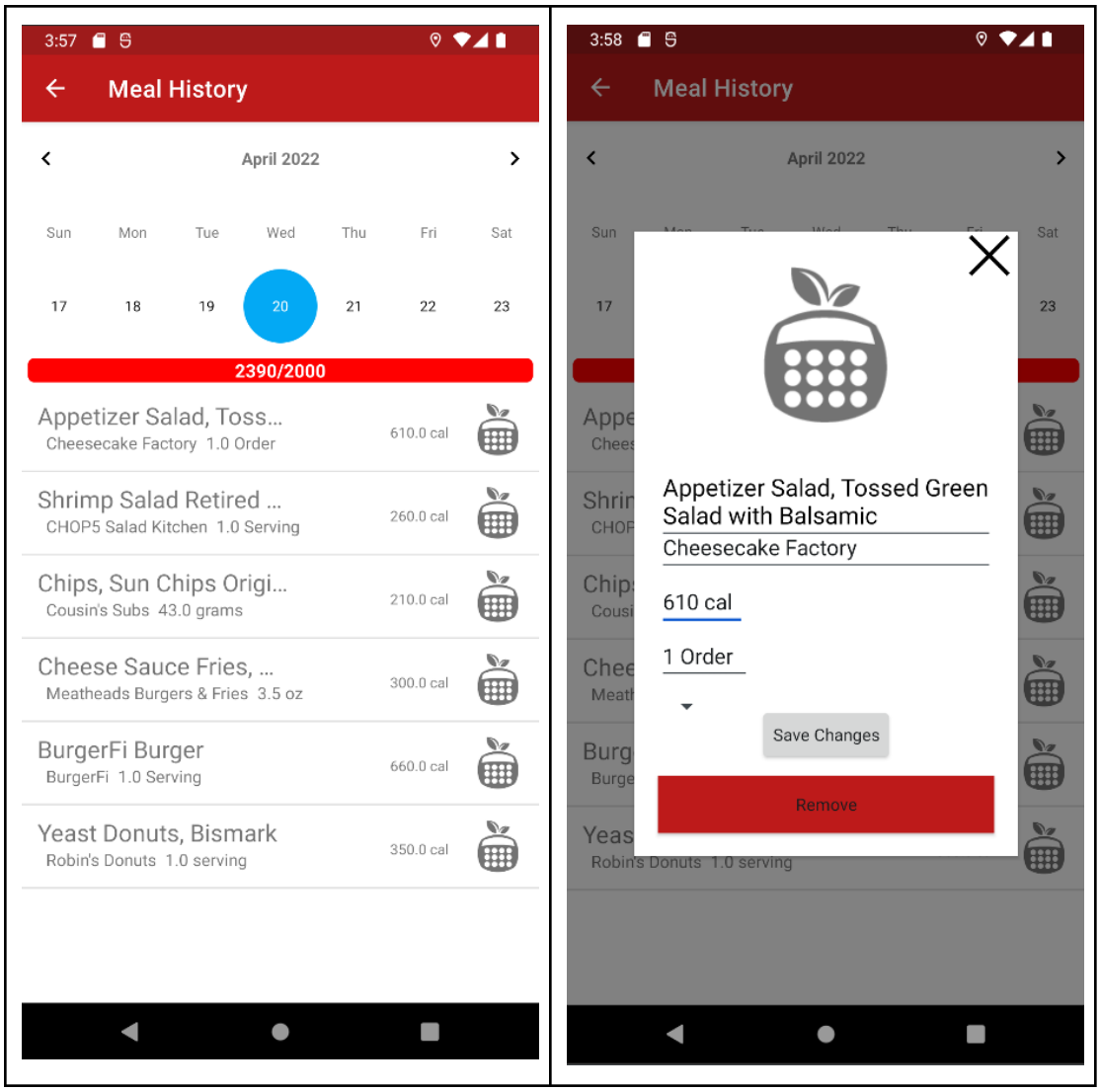
When a user searches for a food item they will see a list of items that the NutritionIX API has returned based on what the user typed in the search box. This can be seen above. If the user clicks on an item in the list they can see its information in a pop up box and change what meal they are eating it for and then add the food.



Next for the workouts portion of the search tab we can see above that the user can use natural language to search for the workout they just performed. Above in the right image we can see a list with one item in the background and in the pop up box in front that appears after clicking on the list item we can again see its details such as calories burned and then add that exercise to the user's profile. Workouts can also be categorized into cardio, weightlifting, or yoga by clicking on add workout under those categories and then using the search box.



Above we can see that under the meals tab there are two options of Meals and Workouts just like on the profile. From these two screens the user can view their meals and workouts that they added for the current day.

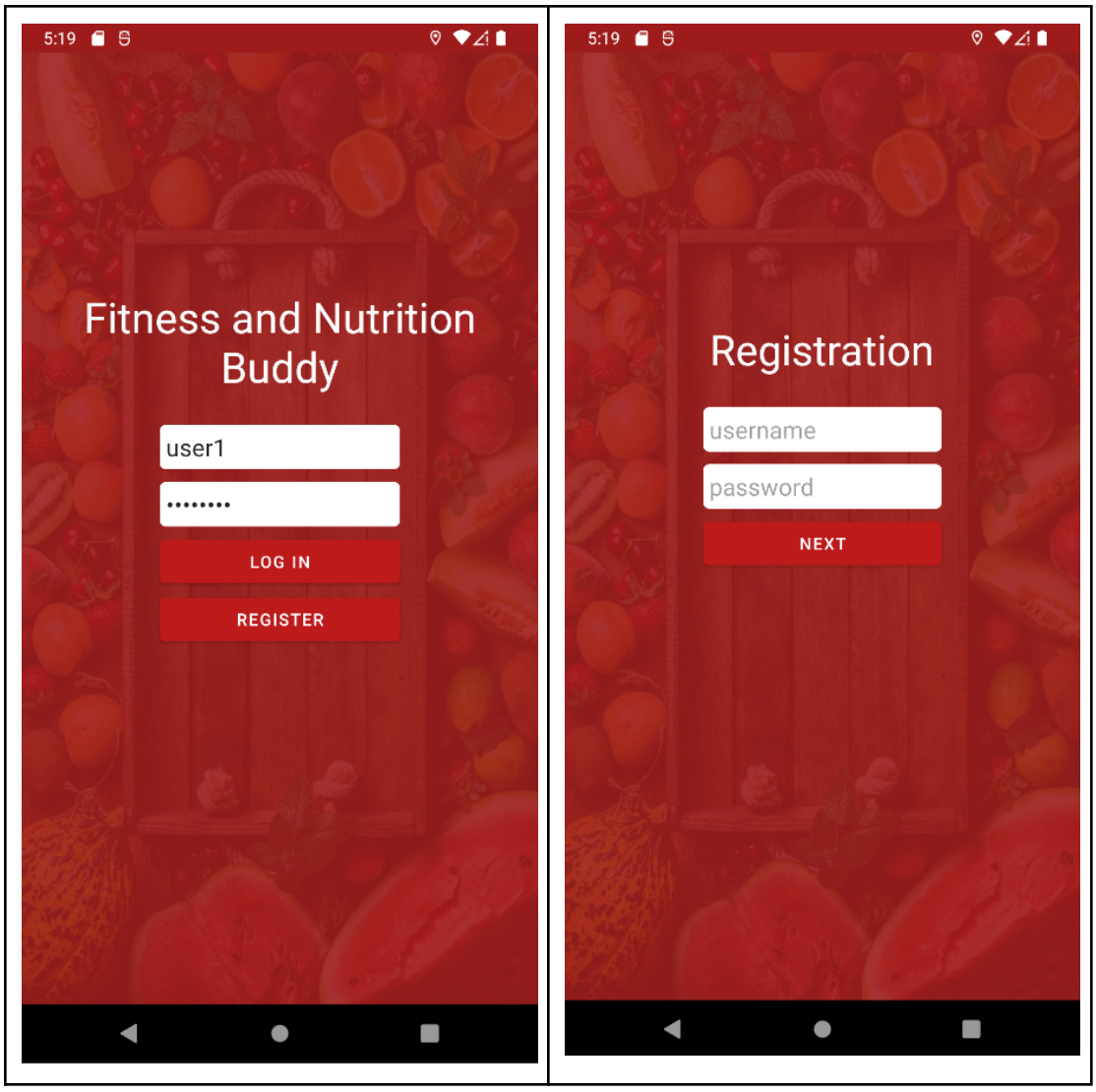


When the user clicks on the calendar button at the top of the meals tab screen the above screen on the left is shown. On this screen the user can select any day and view the meals that they have logged for that day. As is seen in the above image on the right the user can edit a meal item from the list by clicking on it and updating the various fields.

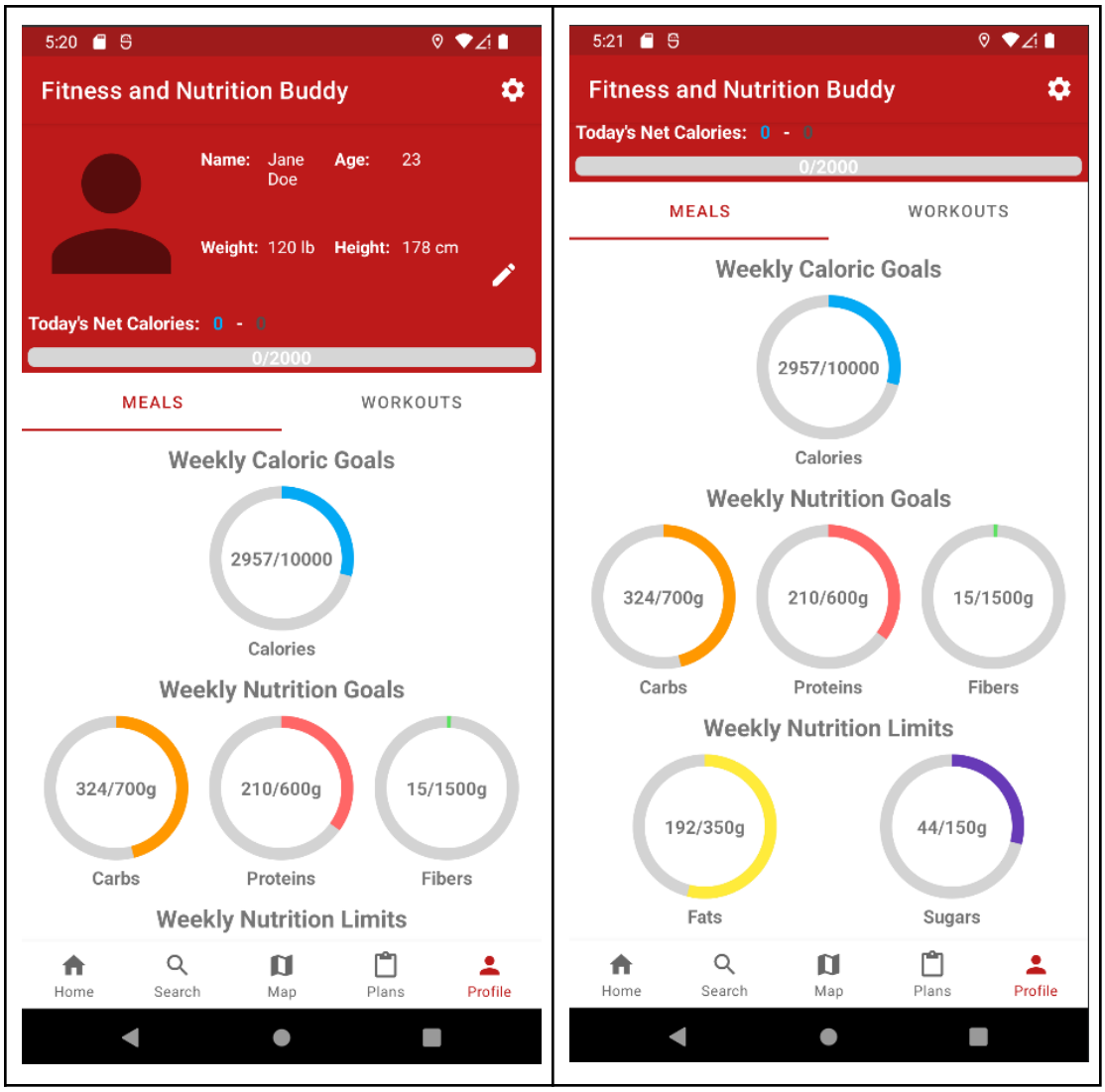
3 Third Release

The third release occurred on April 26th 2022.

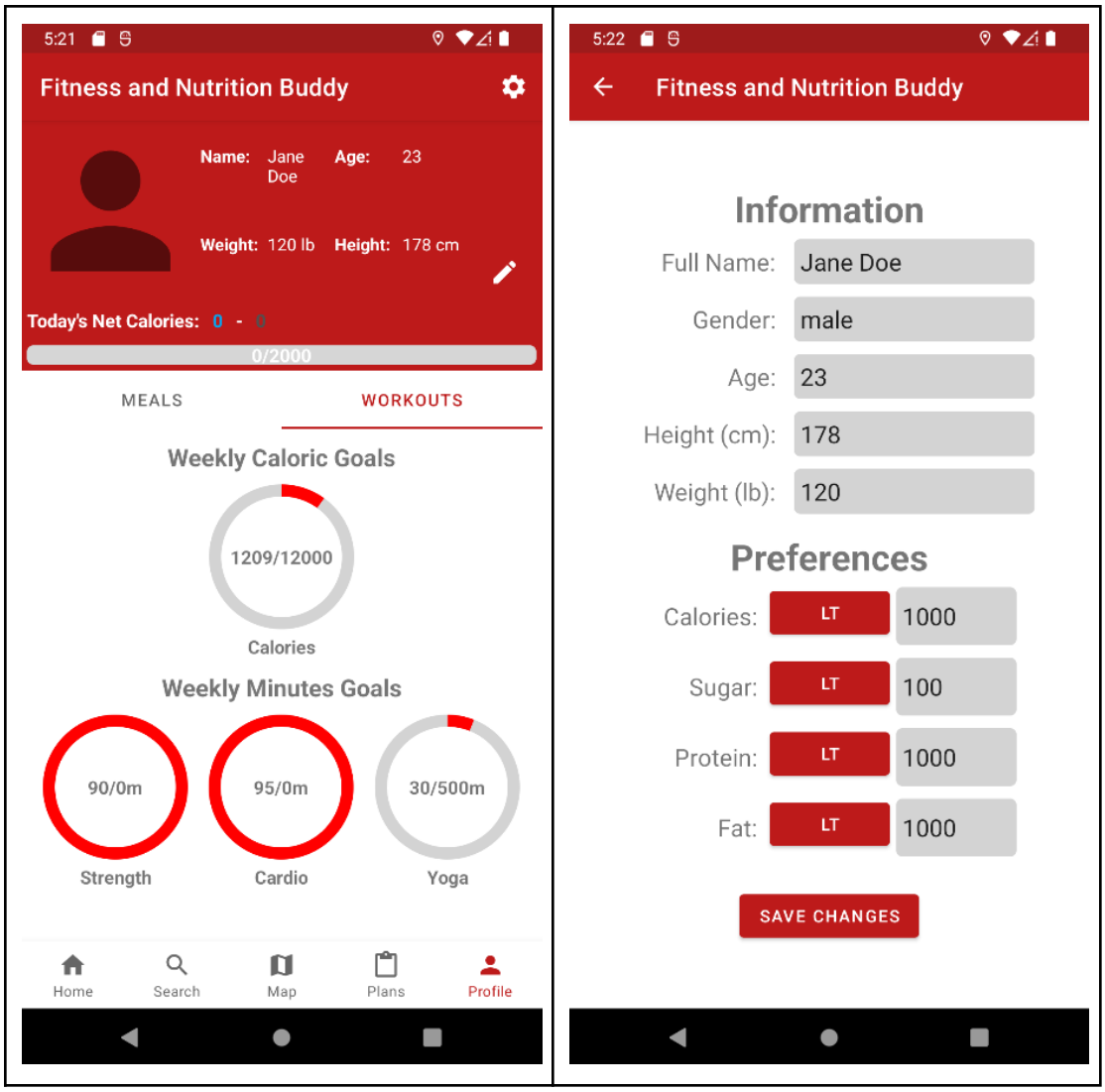
The third release was focused on adding meal plans, workout plans, more macronutrient tracking, making the application less buggy and more user friendly.



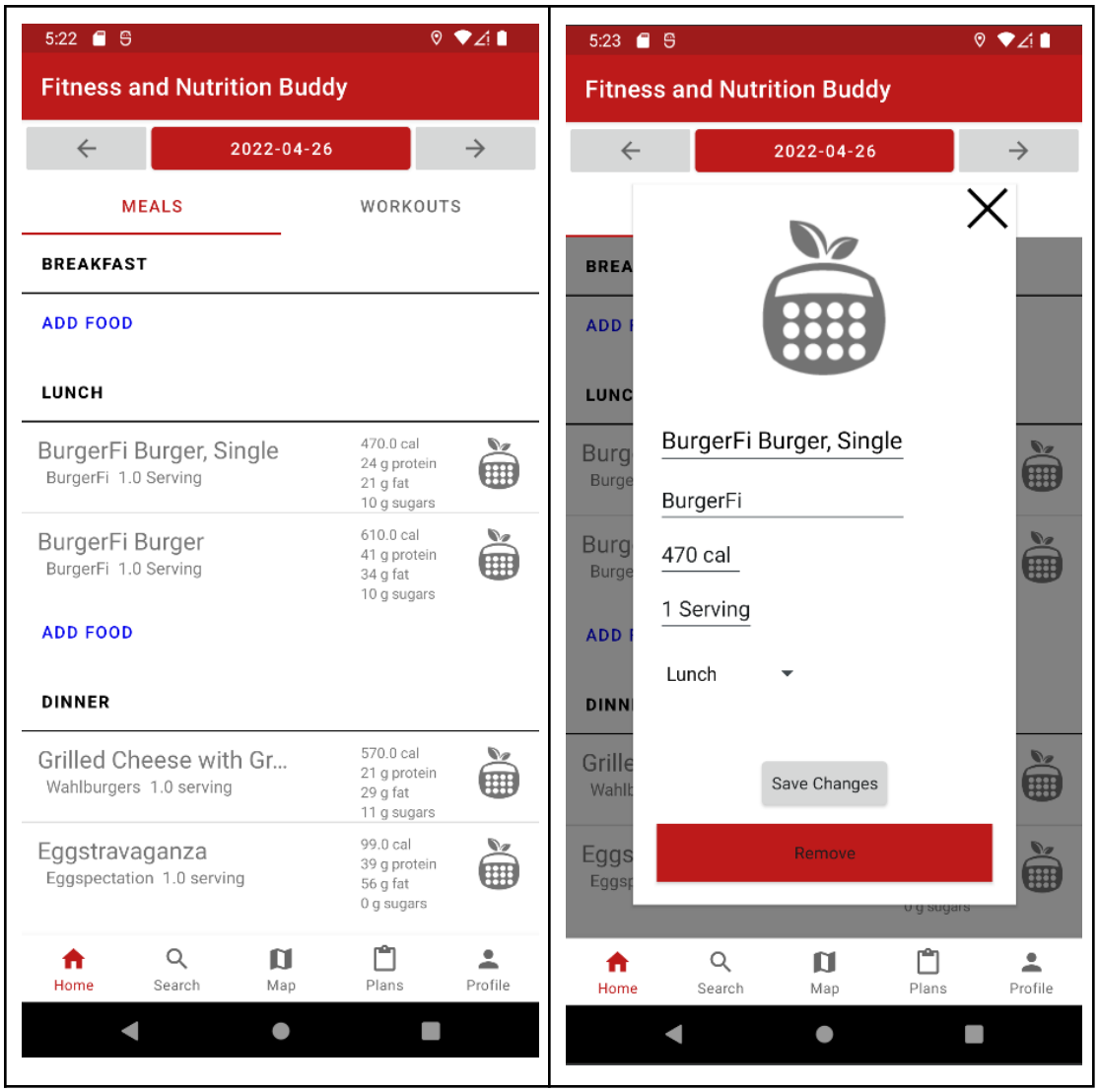
Above we can see the login and register page that is the same as the second release.



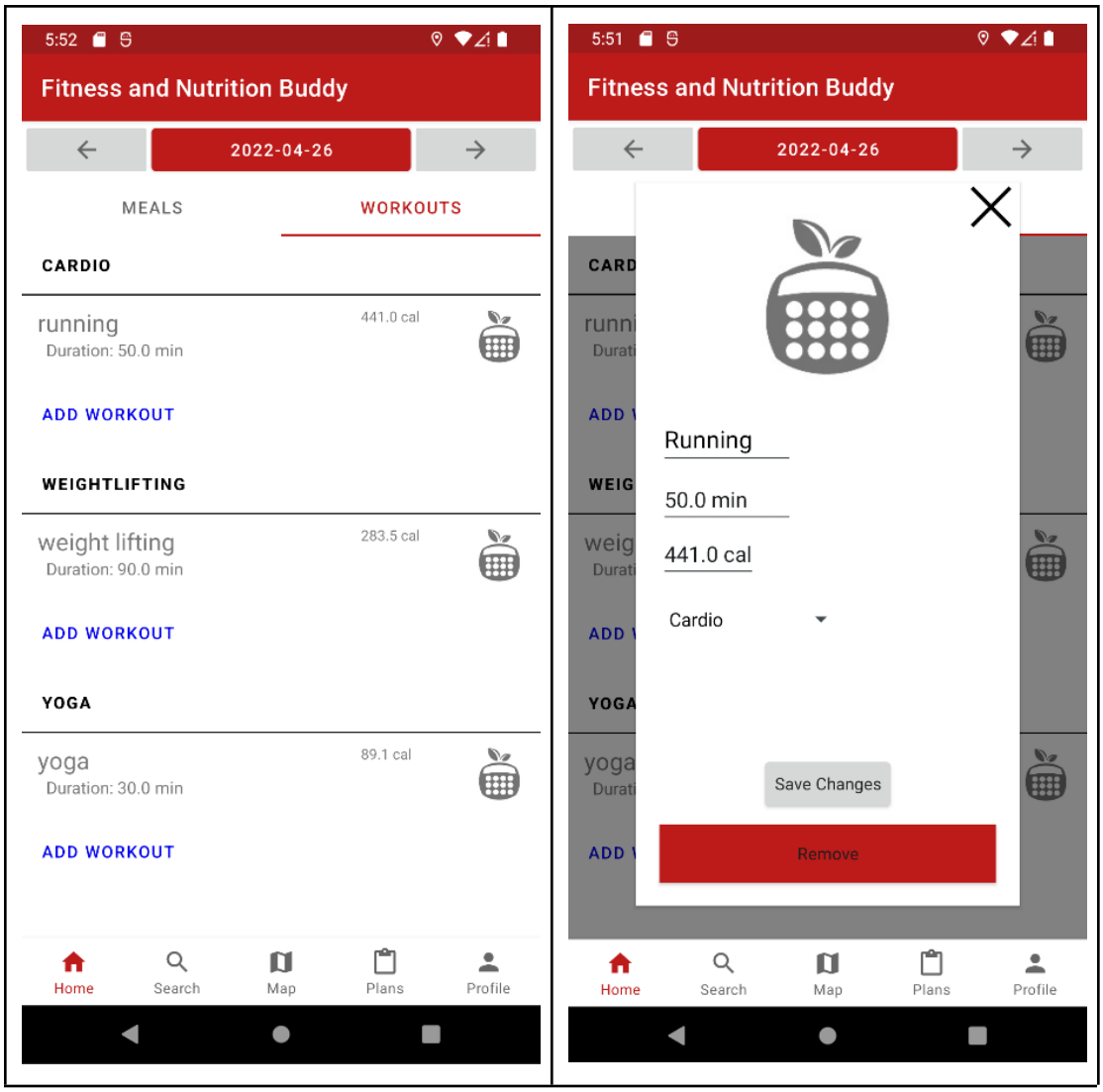
The profile screen as seen above got more tracking bars for nutrition goals and limits as set by the chosen meal plan. This was a new edition for release 3. Release three saw the addition of more macronutrients with the addition of meal plans.



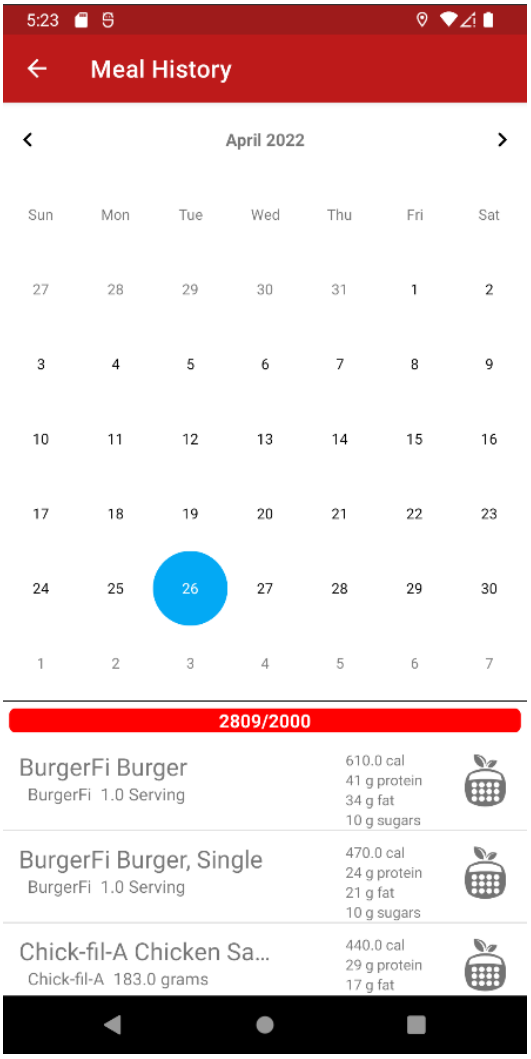
As of release three the profile also has a tab for workouts where the calories burned in the different types of workouts is tracked. Also as seen above the preferences control was added to the profile settings which is located under the pencil icon on the profile page. The preferences are used to filter the menu items that are shown when a restaurant is selected from the map screen.



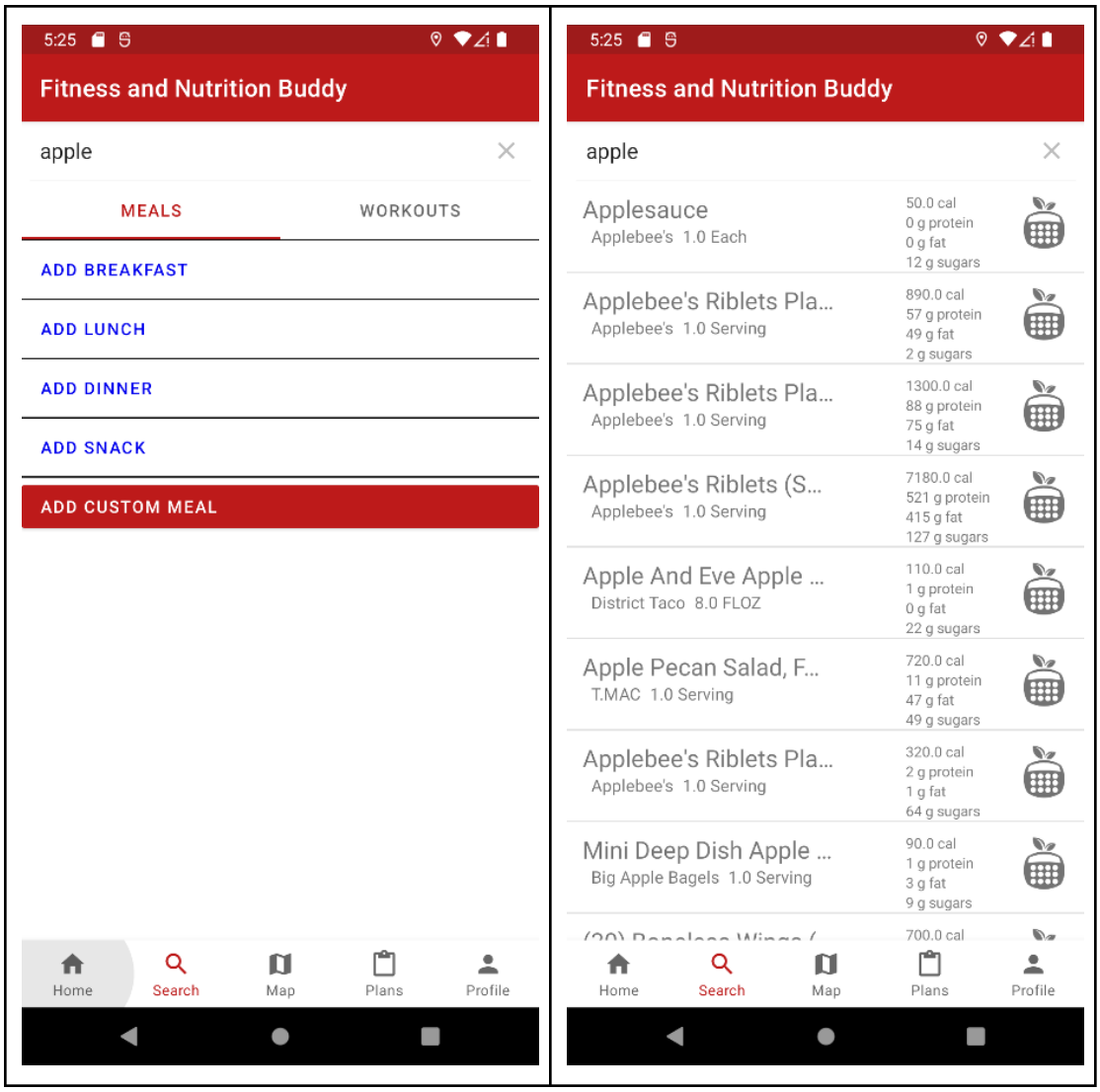
Above we can see the home page which now contains the meals and workouts tabs which will show the meals and workouts from the current day chosen which is adjustable at the top by the left and right arrows. The meals are also editable if they are clicked on and this is shown in the upper right image.



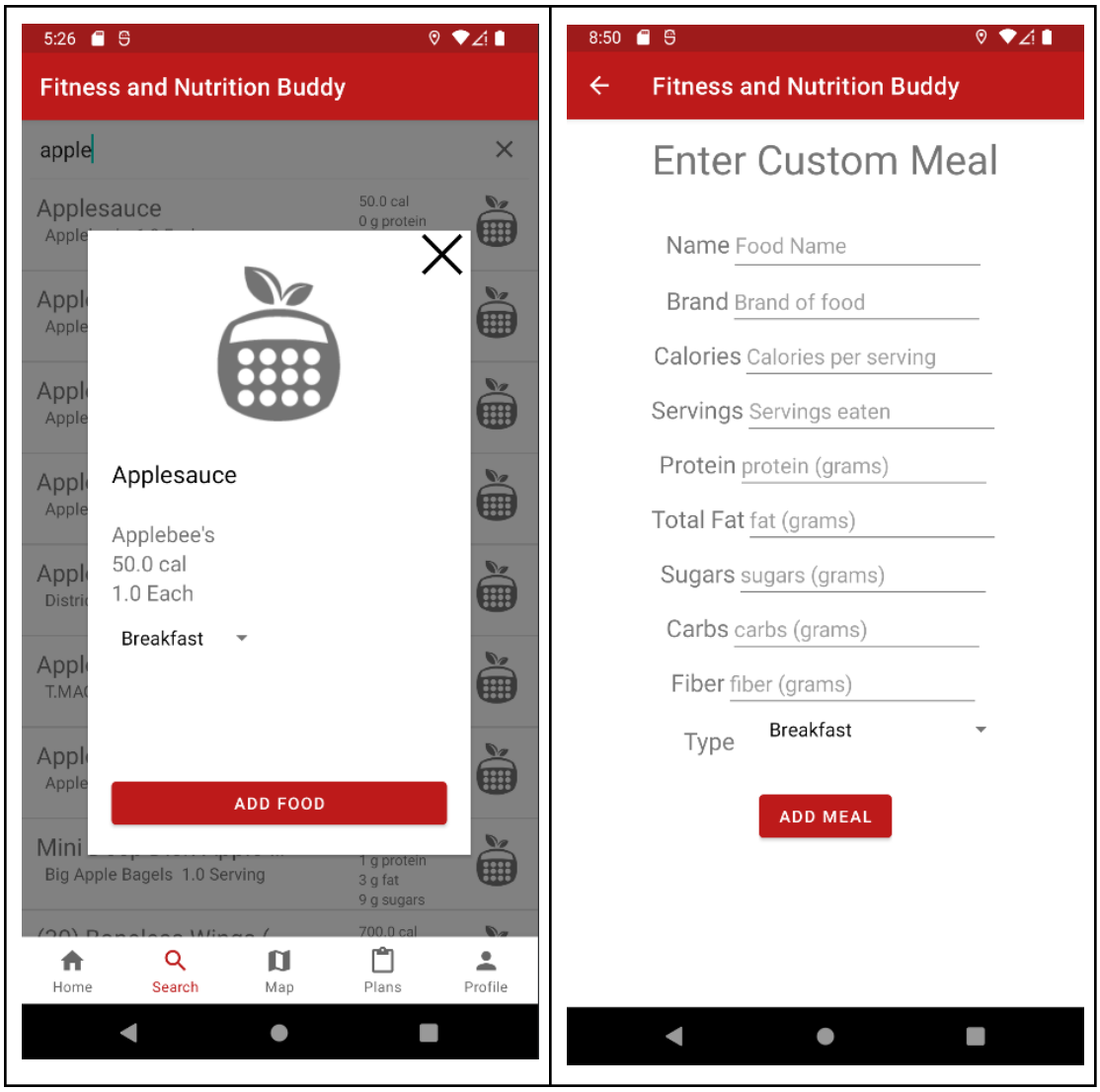
The home page also has a workouts tab where you can see the workouts that have been logged on what days as well as the ability to edit those workouts. This can be seen above.



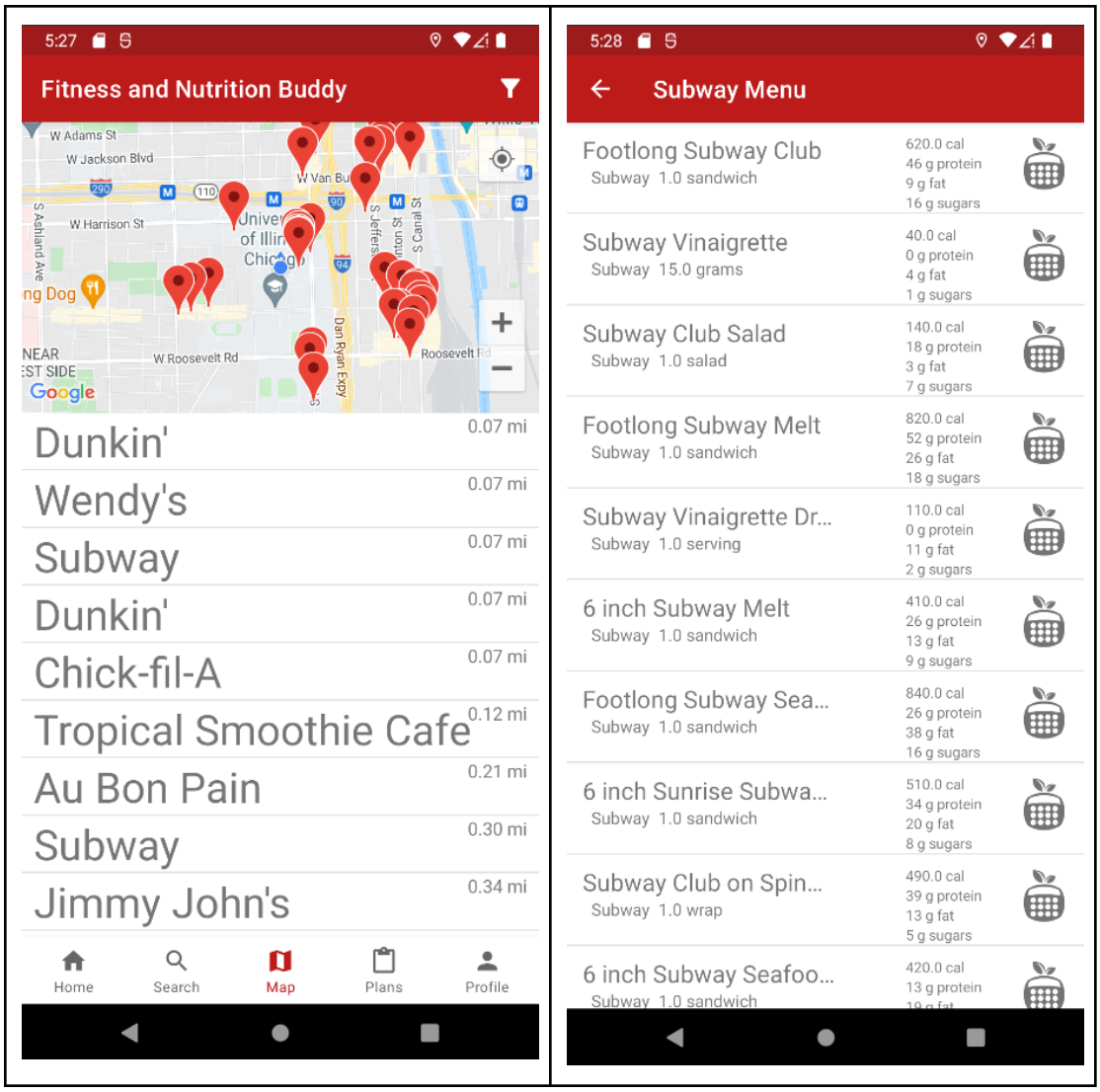
If the user clicks the red button with the date at the top of the profile page they can view a meal history calendar as seen above. The user can choose a day and view the meals logged on those days.



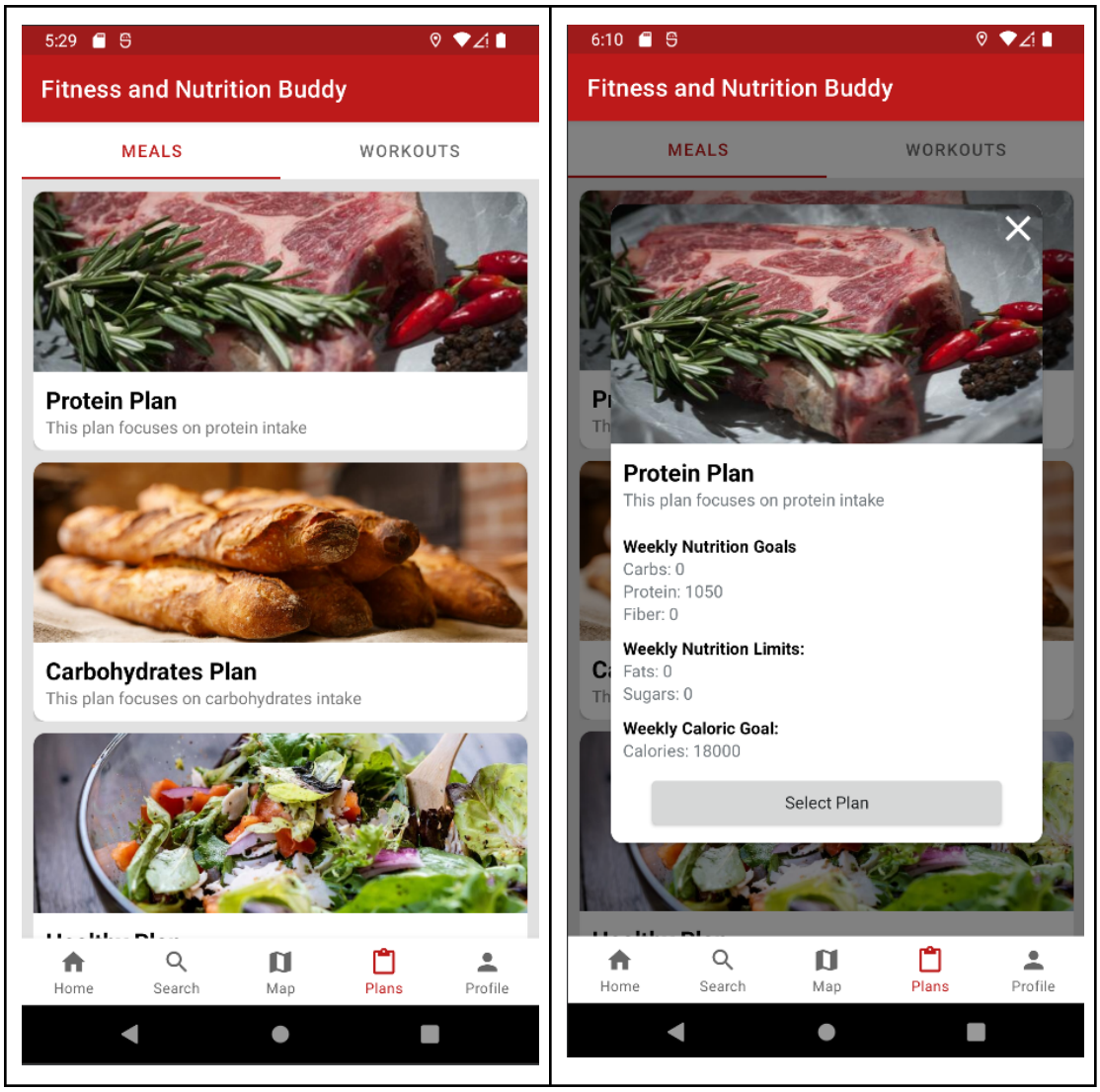
From the search page users can search for foods and even preselect what category for it to go into. When a user searches for a food they are shown a list of meals and four macronutrients for that meal item. The meal items are still queried from the NutritionIX API. The main change to this function is that the meal list items show the calories, protein, fat and sugars for each meal item.



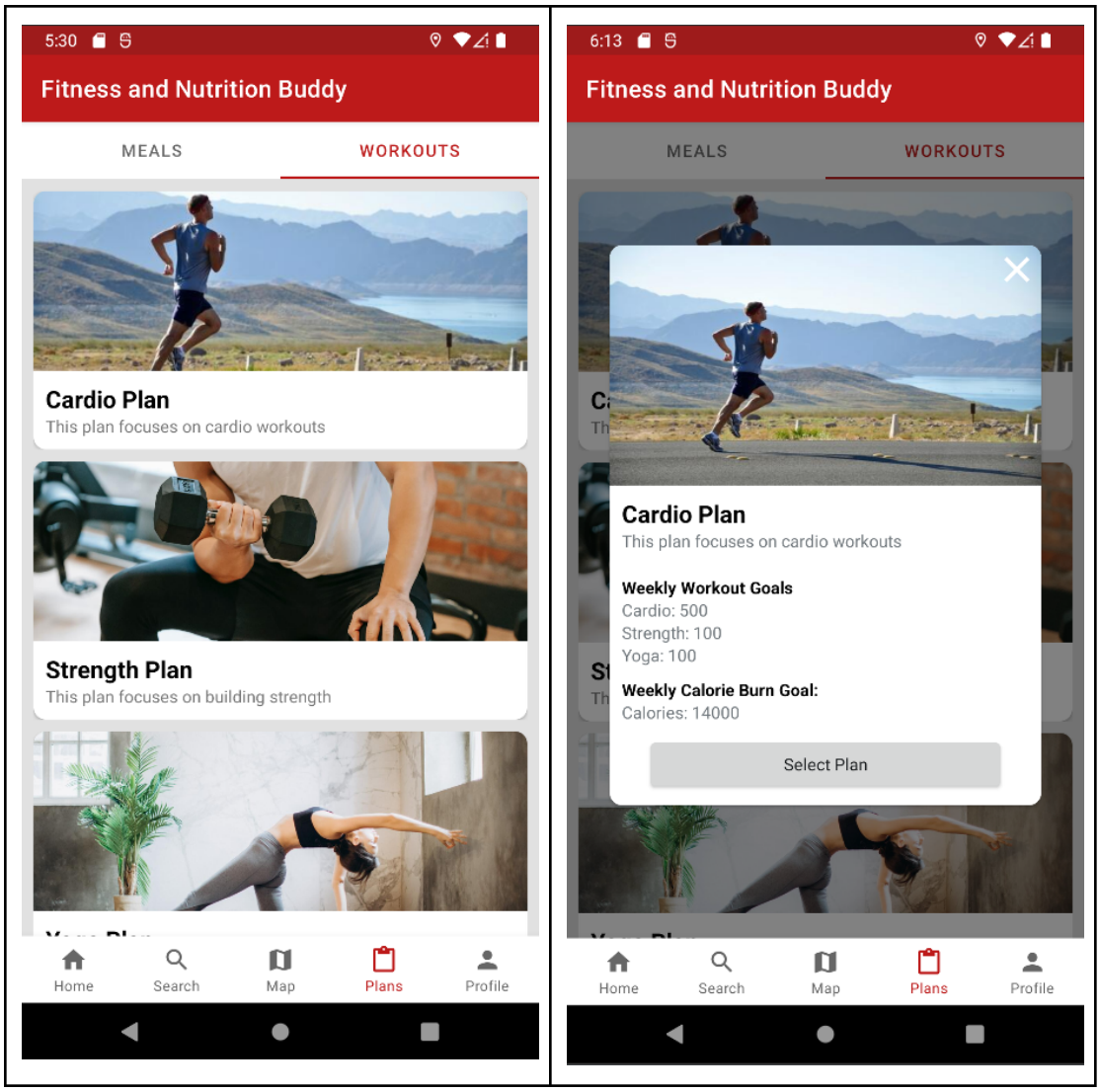
As seen above the user can select a food from the search page and edit what meal you ate it for and add the food to the meal log. In the above right image is the custom meal activity which is not completely new but can now be accessed from the search screen and has the ability to enter the macronutrients for the meal being added.



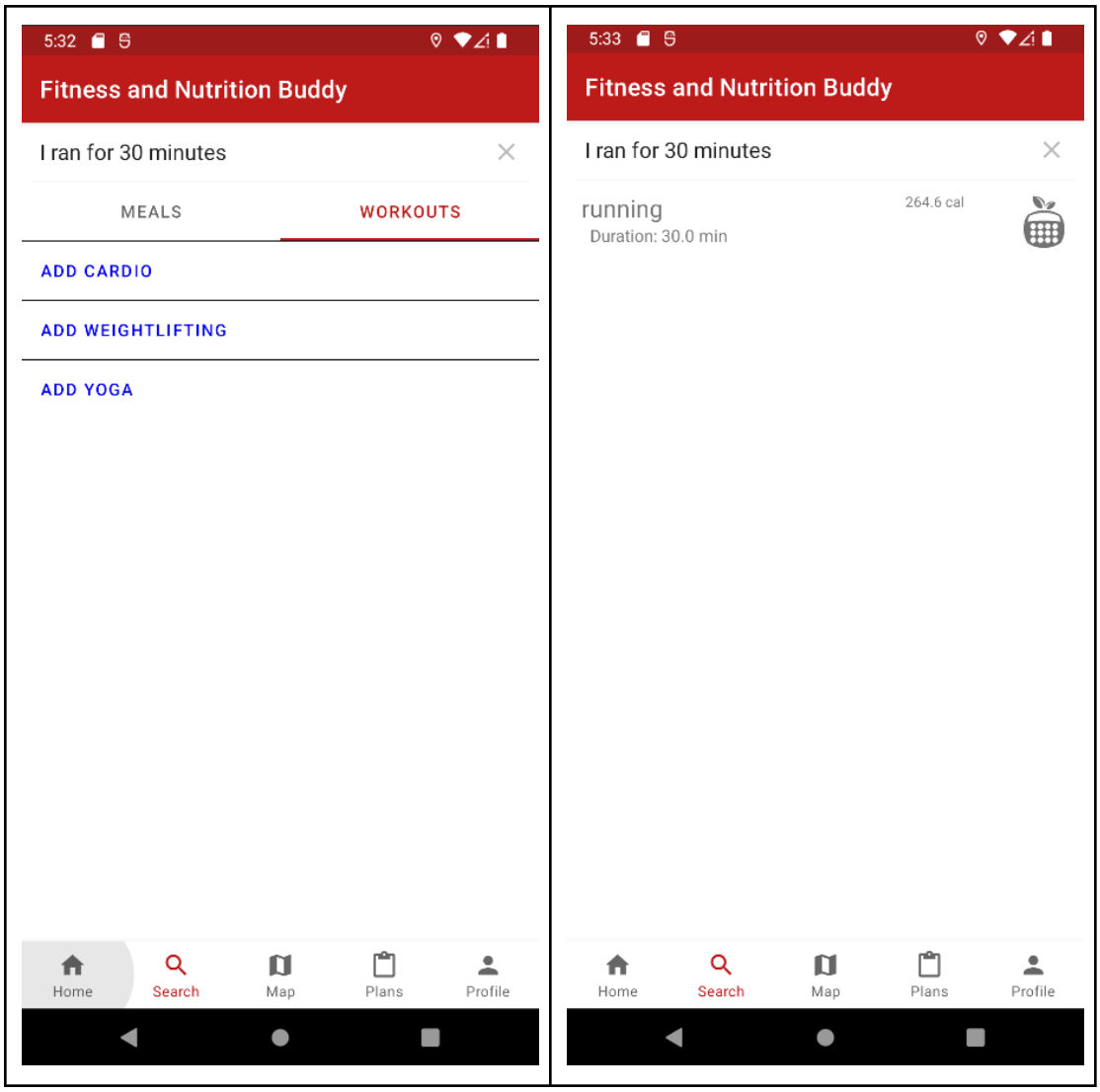
On the map page of the application as seen in the above left image the user location is used to show restaurants nearby and a list of those restaurants. If the user selects a restaurant the menu items are obtained from the branded section of the NutritionIX database and the menu items shown are filtered by the users preferences. As of release 3 the meal items here also show the amount of calories, protein, fat, and sugars.



Above we can see the meal plans in the plans section of the application where a user can select a meal plan. The effect this has is the nutrition goals on the profile page are changed to have different values for the different macronutrients. The nutrition goal number and the nutrition limit numbers will be adjusted based on the meal plan chosen.



Just like for meal plans the application also has workout plans and these adjust the workout goals shown in the bars on the workout section of the profile page.



Above we can see the ability to search for a workout the user has done to add it to their workout log. This uses the natural language search function from NutritionIX.

4 Comparison with Original Project Design Document

The original project has five scenarios which outline the functionality of the project. The five scenarios are nutrition tracking, meal plan, restaurant search, map, and menu data [2].

For the first scenario, nutrition tracking, the application needs to allow users to track their calories and nutrition data. The application that was created allows for users to track their caloric intake as well as their intake of protein, fats, sugars, carbohydrates, and fiber. The application meets this requirement of the original project design.

Next scenario is the meal plan scenario. The project design document states that the user can create meal plans based on restaurants they search for and sorting foods by nutrition groups. The application that was developed implements meal plans in a

more direct way. The user can choose a meal plan from the list of options and can track their progress in meeting that meal plan based on what they eat in a day.

The restaurant search scenario from the design document states that the user can find restaurants in their general vicinity by being connected to Google maps and the user can input food type or general location. The application that was developed has the ability to search for nearby restaurants which are shown on a Google map in a fragment on the screen with the restaurants in a list below the map.

The map scenario for the design documentation states that the app displays a Google map to users which show restaurants around their location and the user's search results would show on the map. The application that was developed has a restaurants near me function which shows restaurants in a chooseable radius around the user and if the user clicks on a restaurant they see the menu for that restaurant.

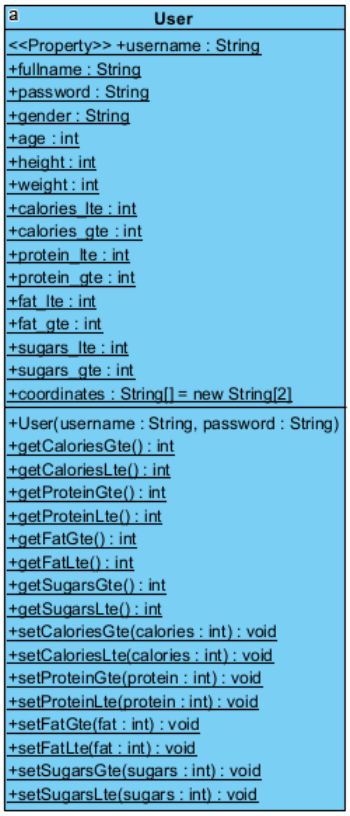
This leads to the menu data scenario. This scenario states that data is pulled from google pertaining to menu information and this information is sent to the meal planning portion of the application to do calculations to optimize meals for the user. The application that was developed allows the user to add a restaurant menu item that they ate to their profile and the calories and nutrition from that meal is kept track of. Instead of pulling restaurant menu items from google the NutritionIX API is used.

The developed application has many functions past what is stated in these five scenarios. The application has a way to search for items that the user ate to keep track of those meals. The user can add a custom meal to their profile if it does not exist in the search function. The application also contains functionality to add exercises to the profile which keeps track of the users calories that were burned. This is done through a natural language search and calculates the calories burned based on the user's profile information such as height and weight.

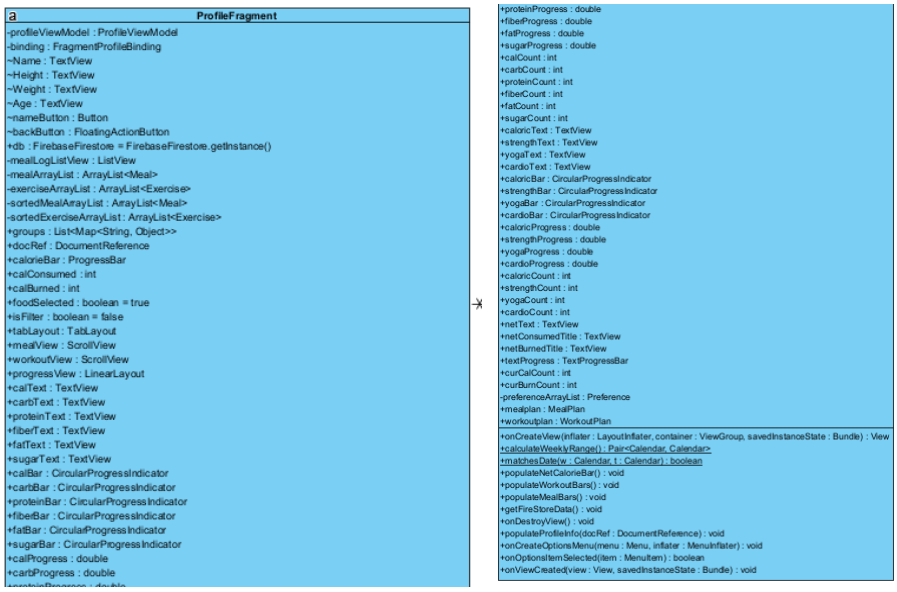
III Testing

1 Items to be Tested

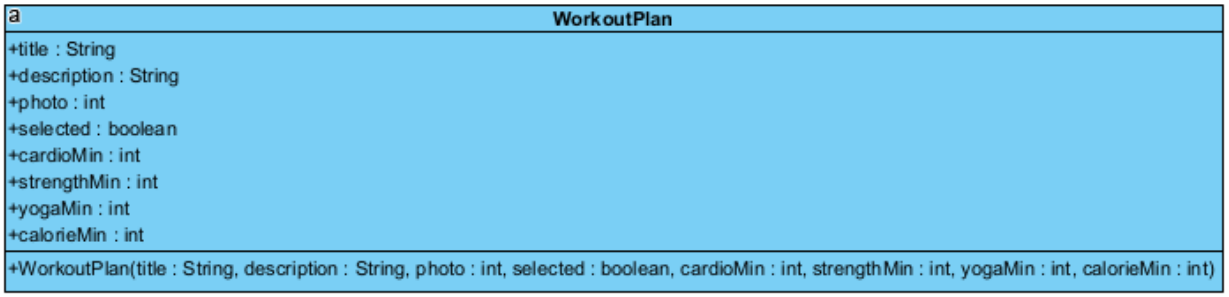
1. User.java



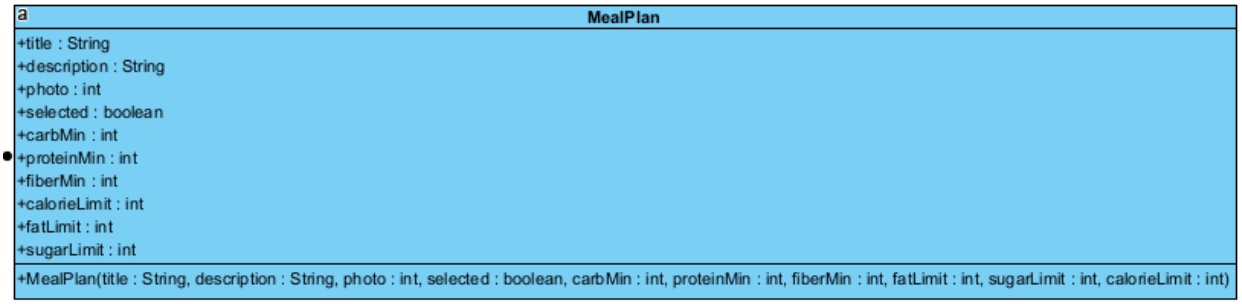
2. ProfileFragment.java



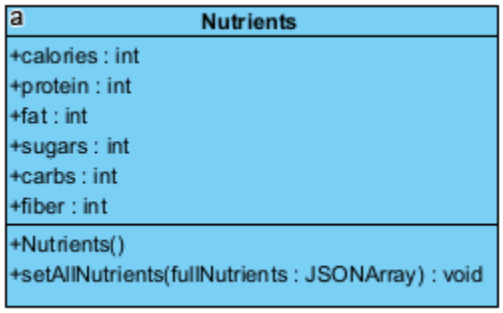
3. WorkoutPlan.java



4. MealPlan.java



5. Nutrition.java



2 Test Specifications

ID#1 - UserClassUsernameIsCorrect

Description: Check if getUsername() function returns expected value

Items covered by this test: getUsername() function in User class

Requirements addressed by this test: Retrieve username of current user

Environmental needs: JUnit

Intercase Dependencies: N/A

Test Procedures: Run Script

Input Specification: "user1" string on initialization within setup()

Output Specifications: the string that was provided on initialization ("user1")

Pass/Fail Criteria: The output equals the expected value

ID#2 - PasswordIsCorrect

Description: Check if password field is accessible and returns expected value

Items covered by this test: password field in User class

Requirements addressed by this test: Retrieve password of current user

Environmental needs: JUnit

Intercase Dependencies: N/A

Test Procedures: Run Script

Input Specification: "password" string on initialization within setup()

Output Specifications: the string that was provided on initialization ("password")

Pass/Fail Criteria: The output equals the expected value

ID#3 - AgeIsCorrect

Description: Check if age field is accessible and returns expected value

Items covered by this test: age field in User class

Requirements addressed by this test: Retrieve age of current user

Environmental needs: JUnit

Intercase Dependencies: N/A

Test Procedures: Run Script

Input Specification: integer 25 on initialization within setup()

Output Specifications: the integer that was provided on initialization (25)

Pass/Fail Criteria: The output equals the expected value

ID#4 - HeightIsCorrect

Description: Check if height field is accessible and returns expected value

Items covered by this test: height field in User class

Requirements addressed by this test: Retrieve height of current user

Environmental needs: JUnit

Intercase Dependencies: N/A

Test Procedures: Run Script

Input Specification: integer 123 on initialization within setup()

Output Specifications: the integer that was provided on initialization (123)

Pass/Fail Criteria: The output equals the expected value

ID#5 - WeightIsCorrect

Description: Check if weight field is accessible and returns expected value

Items covered by this test: weight field in User class

Requirements addressed by this test: Retrieve weight of current user

Environmental needs: JUnit

Intercase Dependencies: N/A

Test Procedures: Run Script

Input Specification: integer 130 on initialization within setup()

Output Specifications: the integer that was provided on initialization (130)

Pass/Fail Criteria: The output equals the expected value

ID#5 - CaloriesLteIsCorrect

Description: Check if calories_lte field is accessible and returns expected value

Items covered by this test: calories_lte field in User class

Requirements addressed by this test: Retrieve calories_lte of current user

Environmental needs: JUnit

Intercase Dependencies: N/A

Test Procedures: Run Script

Input Specification: integer 10 on initialization within setup()

Output Specifications: the integer that was provided on initialization (10)

Pass/Fail Criteria: The output equals the expected value

ID#6 - CaloriesGteIsCorrect

Description: Check if calories_gte field is accessible and returns expected value

Items covered by this test: calories_gte field in User class

Requirements addressed by this test: Retrieve calories_gte of current user

Environmental needs: JUnit

Intercase Dependencies: N/A

Test Procedures: Run Script

Input Specification: integer 12 on initialization within setup()

Output Specifications: the integer that was provided on initialization (12)

Pass/Fail Criteria: The output equals the expected value

ID#7 -proteinLteIsCorrect

Description: Check if protein_lte field is accessible and returns expected value

Items covered by this test: protein_lte field in User class

Requirements addressed by this test: Retrieve protein_lte of current user

Environmental needs: JUnit

Intercase Dependencies: N/A

Test Procedures: Run Script

Input Specification: integer 7 on initialization within setup()

Output Specifications: the integer that was provided on initialization (7)

Pass/Fail Criteria: The output equals the expected value

ID#8 -proteinGteIsCorrect

Description: Check if protein_gte field is accessible and returns expected value

Items covered by this test: protein_gte field in User class

Requirements addressed by this test: Retrieve protein_gte of current user

Environmental needs: JUnit

Intercase Dependencies: N/A

Test Procedures: Run Script

Input Specification: integer 8 on initialization within setup()

Output Specifications: the integer that was provided on initialization (8)

Pass/Fail Criteria: The output equals the expected value

ID#8 -matchingDateIsTrue

Description: Check if matchesDate() accurately determines if two dates are matching

Items covered by this test: matchesDate() function in ProfileFragment class

Requirements addressed by this test: Retrieve boolean to see if dates are equivalent

Environmental needs: JUnit

Intercase Dependencies: N/A

Test Procedures: Run Script

Input Specification: Current date variable on initialization within setup()

Output Specifications: Returns true when two dates are equal

Pass/Fail Criteria: The output equals the expected value

ID#9 -matchingDateIsFalse

Description: Check if matchesDate() accurately determines if two dates are matching

Items covered by this test: matchesDate() function in ProfileFragment class

Requirements addressed by this test: Retrieve boolean to see if dates are equivalent

Environmental needs: JUnit

Intercase Dependencies: N/A

Test Procedures: Run Script

Input Specification: Current date variable on initialization within setup()

Output Specifications: Returns false when two dates are equal

Pass/Fail Criteria: The output equals the expected value

ID#10 - titleWorkoutPlanIsCorrect

Description: Check if title field is accessible and returns expected value

Items covered by this test: title field in WorkoutPlan class

Requirements addressed by this test: Retrieve title of workout plan

Environmental needs: JUnit

Intercase Dependencies: N/A

Test Procedures: Run Script

Input Specification: string “yoga plan” on initialization within setup()

Output Specifications: the string that was provided on initialization (“yoga plan”)

Pass/Fail Criteria: The output equals the expected value

ID#10 - descriptionWorkoutPlanIsCorrect

Description: Check if description field is accessible and returns expected value

Items covered by this test: description field in WorkoutPlan class

Requirements addressed by this test: Retrieve description of workout plan

Environmental needs: JUnit

Intercase Dependencies: N/A

Test Procedures: Run Script

Input Specification: string “test” on initialization within setup()

Output Specifications: the string that was provided on initialization (“test”)

Pass/Fail Criteria: The output equals the expected value

ID#11 - selectedWorkoutPlanIsCorrect

Description: Check if selected field is accessible and returns expected value

Items covered by this test: selected field in WorkoutPlan class

Requirements addressed by this test: Retrieve selected flag of workout plan

Environmental needs: JUnit

Intercase Dependencies: N/A

Test Procedures: Run Script

Input Specification: boolean “true” on initialization within setup()

Output Specifications: the boolean that was provided on initialization (“true”)

Pass/Fail Criteria: The output equals the expected value

ID#12 - photoWorkoutPlanIsCorrect

Description: Check if photo field is accessible and returns expected value

Items covered by this test: photo field in WorkoutPlan class

Requirements addressed by this test: Retrieve photo of workout plan

Environmental needs: JUnit

Intercase Dependencies: N/A

Test Procedures: Run Script

Input Specification: integer 1 on initialization within setup()

Output Specifications: the boolean that was provided on initialization (1)

Pass/Fail Criteria: The output equals the expected value

ID#13 - cardioWorkoutPlanIsCorrect

Description: Check if cardio field is accessible and returns expected value

Items covered by this test: cardio field in WorkoutPlan class

Requirements addressed by this test: Retrieve cardio of workout plan

Environmental needs: JUnit

Intercase Dependencies: N/A

Test Procedures: Run Script

Input Specification: integer 2 on initialization within setup()

Output Specifications: the boolean that was provided on initialization (2)

Pass/Fail Criteria: The output equals the expected value

ID#14 - strengthWorkoutPlanIsCorrect

Description: Check if strength field is accessible and returns expected value

Items covered by this test: strength field in WorkoutPlan class

Requirements addressed by this test: Retrieve strength of workout plan

Environmental needs: JUnit

Intercase Dependencies: N/A

Test Procedures: Run Script

Input Specification: integer 3 on initialization within setup()

Output Specifications: the boolean that was provided on initialization (3)

Pass/Fail Criteria: The output equals the expected value

ID#15 - yogaWorkoutPlanIsCorrect

Description: Check if yoga field is accessible and returns expected value

Items covered by this test: yoga field in WorkoutPlan class

Requirements addressed by this test: Retrieve yoga of workout plan

Environmental needs: JUnit

Intercase Dependencies: N/A

Test Procedures: Run Script

Input Specification: integer 4 on initialization within setup()

Output Specifications: the boolean that was provided on initialization (4)

Pass/Fail Criteria: The output equals the expected value

ID#16 - calorieWorkoutPlanIsCorrect

Description: Check if calorie field is accessible and returns expected value

Items covered by this test: calorie field in WorkoutPlan class

Requirements addressed by this test: Retrieve calorie of workout plan

Environmental needs: JUnit

Intercase Dependencies: N/A

Test Procedures: Run Script

Input Specification: integer 5 on initialization within setup()

Output Specifications: the boolean that was provided on initialization (5)

Pass/Fail Criteria: The output equals the expected value

ID#17 - titleMealPlanIsCorrect

Description: Check if title field is accessible and returns expected value

Items covered by this test: title field in MealPlan class

Requirements addressed by this test: Retrieve title of meal plan

Environmental needs: JUnit

Intercase Dependencies: N/A

Test Procedures: Run Script

Input Specification: string “carb plan” on initialization within setup()

Output Specifications: the string that was provided on initialization (“carb plan”)

Pass/Fail Criteria: The output equals the expected value

ID#18 - descriptionMealPlanIsCorrect

Description: Check if description field is accessible and returns expected value

Items covered by this test: description field in MealPlan class

Requirements addressed by this test: Retrieve description of meal plan

Environmental needs: JUnit

Intercase Dependencies: N/A

Test Procedures: Run Script

Input Specification: string “test” on initialization within setup()

Output Specifications: the string that was provided on initialization (“test”)

ID#19 - selectedMealPlanIsCorrect

Description: Check if selected field is accessible and returns expected value

Items covered by this test: selected field in MealPlan class

Requirements addressed by this test: Retrieve selected flag of meal plan

Environmental needs: JUnit

Intercase Dependencies: N/A

Test Procedures: Run Script

Input Specification: boolean “false” on initialization within setup()

Output Specifications: the boolean that was provided on initialization (“false”)

Pass/Fail Criteria: The output equals the expected value

ID#20 - photoMealPlanIsCorrect

Description: Check if photo field is accessible and returns expected value

Items covered by this test: photo field in MealPlan class

Requirements addressed by this test: Retrieve photo of meal plan

Environmental needs: JUnit

Intercase Dependencies: N/A

Test Procedures: Run Script

Input Specification: integer 1 on initialization within setup()

Output Specifications: the boolean that was provided on initialization (1)

Pass/Fail Criteria: The output equals the expected value

ID#21 - carbMealPlanIsCorrect

Description: Check if carb field is accessible and returns expected value

Items covered by this test: carb field in MealPlan class

Requirements addressed by this test: Retrieve carb of meal plan

Environmental needs: JUnit

Intercase Dependencies: N/A

Test Procedures: Run Script

Input Specification: integer 2 on initialization within setup()

Output Specifications: the boolean that was provided on initialization (2)

Pass/Fail Criteria: The output equals the expected value

ID#22 - proteinMealPlanIsCorrect

Description: Check if protein field is accessible and returns expected value

Items covered by this test: protein field in MealPlan class

Requirements addressed by this test: Retrieve protein of meal plan

Environmental needs: JUnit

Intercase Dependencies: N/A

Test Procedures: Run Script

Input Specification: integer 3 on initialization within setup()

Output Specifications: the boolean that was provided on initialization (3)

Pass/Fail Criteria: The output equals the expected value

ID#23 - fiberMealPlanIsCorrect

Description: Check if fiber field is accessible and returns expected value

Items covered by this test: fiber field in MealPlan class

Requirements addressed by this test: Retrieve protein of meal plan

Environmental needs: JUnit

Intercase Dependencies: N/A

Test Procedures: Run Script

Input Specification: integer 4 on initialization within setup()

Output Specifications: the boolean that was provided on initialization (4)

Pass/Fail Criteria: The output equals the expected value

ID#24 - fatMealPlanIsCorrect

Description: Check if fat field is accessible and returns expected value

Items covered by this test: fat field in MealPlan class

Requirements addressed by this test: Retrieve fat of meal plan

Environmental needs: JUnit

Intercase Dependencies: N/A

Test Procedures: Run Script

Input Specification: integer 5 on initialization within setup()

Output Specifications: the boolean that was provided on initialization (5)

Pass/Fail Criteria: The output equals the expected value

ID#25 - sugarMealPlanIsCorrect

Description: Check if sugar field is accessible and returns expected value

Items covered by this test: sugar field in MealPlan class

Requirements addressed by this test: Retrieve sugar of meal plan

Environmental needs: JUnit

Intercase Dependencies: N/A

Test Procedures: Run Script

Input Specification: integer 6 on initialization within setup()

Output Specifications: the boolean that was provided on initialization (6)

Pass/Fail Criteria: The output equals the expected value

ID#26 - calorieMealPlanIsCorrect

Description: Check if calorie field is accessible and returns expected value

Items covered by this test: calorie field in MealPlan class

Requirements addressed by this test: Retrieve calorie of meal plan

Environmental needs: JUnit

Intercase Dependencies: N/A

Test Procedures: Run Script

Input Specification: integer 7 on initialization within setup()

Output Specifications: the boolean that was provided on initialization (7)

Pass/Fail Criteria: The output equals the expected value

ID#27 - NutrientsSetCorrectly

Description: Check to see if nutrients are extracted from JSONArray correctly

Items covered by this test: setAllNutrients(JSONArray nutrientArray)

Requirements addressed by this test: Get meal nutritional info

Environmental needs: JUnit

Intercase Dependencies: N/A

Test Procedures: Run Script

Input Specification: various values for each nutritional detail

Output Specifications: the nutrition value from the Nutrients object which was set by using the JSONArray

Pass/Fail Criteria: The output equals the expected value

3 Test Results

ID#1 - UserClassUsernameIsCorrect

Date(s) of Execution: 4/20/22

Staff conducting tests: John

Expected Results: Return “user1”

Actual Results: Returned “user1”

Test Status: Pass

ID#2 - PasswordIsCorrect

Date(s) of Execution: 4/20/22

Staff conducting tests: John

Expected Results: Return “password”

Actual Results: Returned “password”

Test Status: Pass

ID#3 - AgeIsCorrect

Date(s) of Execution: 4/20/22

Staff conducting tests: John

Expected Results: Return 25

Actual Results: Returned 25

Test Status: Pass

ID#4 - HeightIsCorrect

Date(s) of Execution: 4/20/22

Staff conducting tests: John

Expected Results: Return 123

Actual Results: Returned 123

Test Status: Pass

ID#5 - WeightIsCorrect

Date(s) of Execution: 4/20/22

Staff conducting tests: John

Expected Results: Return 130

Actual Results: Returned 130

Test Status: Pass

ID#5 - CaloriesLteIsCorrect

Date(s) of Execution: 4/20/22

Staff conducting tests: John

Expected Results: Return 10

Actual Results: Returned 10

Test Status: Pass

ID#6 - CaloriesGteIsCorrect

Date(s) of Execution: 4/20/22

Staff conducting tests: John

Expected Results: Return 12

Actual Results: Returned 12

Test Status: Pass

ID#7 -proteinLteIsCorrect

Date(s) of Execution: 4/20/22

Staff conducting tests: John

Expected Results: Return 7

Actual Results: Returned 7

Test Status: Pass

ID#8 -proteinGteIsCorrect

Date(s) of Execution: 4/20/22

Staff conducting tests: John

Expected Results: Return 8

Actual Results: Returned 8

Test Status: Pass

ID#8 -matchingDateIsTrue

Date(s) of Execution: 4/20/22

Staff conducting tests: John

Expected Results: Return true

Actual Results: Returned true

Test Status: Pass

ID#9 -matchingDateIsFalse

Date(s) of Execution: 4/20/22

Staff conducting tests: John

Expected Results: Return false

Actual Results: Returned false

Test Status: Pass

ID#10 - titleWorkoutPlanIsCorrect

Date(s) of Execution: 4/20/22

Staff conducting tests: John

Expected Results: Return “yoga plan”

Actual Results: Returned “yoga plan”

Test Status: Pass

ID#10 - descriptionWorkoutPlanIsCorrect

Date(s) of Execution: 4/20/22

Staff conducting tests: John

Expected Results: Return “test”

Actual Results: Returned “test”

Test Status: Pass

ID#11 - selectedWorkoutPlanIsCorrect

Date(s) of Execution: 4/20/22

Staff conducting tests: John

Expected Results: Return true

Actual Results: Returned true

Test Status: Pass

ID#12 - photoWorkoutPlanIsCorrect

Date(s) of Execution: 4/20/22

Staff conducting tests: John

Expected Results: Return 1

Actual Results: Returned 1

Test Status: Pass

ID#13 - cardioWorkoutPlanIsCorrect

Date(s) of Execution: 4/20/22

Staff conducting tests: John

Expected Results: Return 2

Actual Results: Returned 2

Test Status: Pass

ID#14 - strengthWorkoutPlanIsCorrect

Date(s) of Execution: 4/20/22

Staff conducting tests: John

Expected Results: Return 3

Actual Results: Returned 3

Test Status: Pass

ID#15 - yogaWorkoutPlanIsCorrect

Date(s) of Execution: 4/20/22

Staff conducting tests: John

Expected Results: Return 4

Actual Results: Returned 4

Test Status: Pass

ID#16 - calorieWorkoutPlanIsCorrect

Date(s) of Execution: 4/20/22

Staff conducting tests: John

Expected Results: Return 5

Actual Results: Returned 5

Test Status: Pass

ID#17 - titleMealPlanIsCorrect

Date(s) of Execution: 4/20/22

Staff conducting tests: John

Expected Results: Return “carb plan”

Actual Results: Returned “carb plan”

Test Status: Pass

ID#18 - descriptionMealPlanIsCorrect

Date(s) of Execution: 4/20/22

Staff conducting tests: John

Expected Results: Return “test”

Actual Results: Returned “test”

Test Status: Pass

ID#19 - selectedMealPlanIsCorrect

Date(s) of Execution: 4/20/22

Staff conducting tests: John

Expected Results: Return false

Actual Results: Returned false

Test Status: Pass

ID#20 - photoMealPlanIsCorrect

Date(s) of Execution: 4/20/22

Staff conducting tests: John

Expected Results: Return 1

Actual Results: Returned 1

Test Status: Pass

ID#21 - carbMealPlanIsCorrect

Date(s) of Execution: 4/20/22

Staff conducting tests: John

Expected Results: Return 2

Actual Results: Returned 2

Test Status: Pass

ID#22 - proteinMealPlanIsCorrect

Date(s) of Execution: 4/20/22

Staff conducting tests: John

Expected Results: Return 3

Actual Results: Returned 3

Test Status: Pass

ID#23 - fiberMealPlanIsCorrect

Date(s) of Execution: 4/20/22

Staff conducting tests: John

Expected Results: Return 4

Actual Results: Returned 4

Test Status: Pass

ID#24 - fatMealPlanIsCorrect

Date(s) of Execution: 4/20/22

Staff conducting tests: John

Expected Results: Return 5

Actual Results: Returned 5

Test Status: Pass

ID#25 - sugarMealPlanIsCorrect

Date(s) of Execution: 4/20/22

Staff conducting tests: John

Expected Results: Return 6

Actual Results: Returned 6

Test Status: Pass

ID#26 - calorieMealPlanIsCorrect

Date(s) of Execution: 4/20/22

Staff conducting tests: John

Expected Results: Return 7

Actual Results: Returned 7

Test Status: Pass

ID#27 - NutrientsSetCorrectly

Date(s) of Execution: 4/20/22

Staff conducting tests: Thomas

Expected Results: calories = 10, protein = 20, fat = 30, sugars = 40, carbs = 50, fiber = 60

Actual Results: calories = 10, protein = 20, fat = 30, sugars = 40, carbs = 50, fiber = 60

Test Status: Pass

4 Regression Testing

N/A

IV Inspection

1 Items to be Inspected

John: In MapFragment.java file inspect sendAPIRequest() and onMapReady() functions which populate nearby restaurant array and display pins on the Google Map view.

Thomas: In the RestaurantMenu.java file inspect the sendAPIRequest() function which creates an ArrayList<Meal> based on user preferences and the restaurant provided from the activity before restaurant menu.

Cristian: In the WorkoutsTab2Fragment file inspect the populateMealLog() function which displays the list of workout items.

Jay: In the ProfileEvent.java file inspect the nameButton.setOnClickListener() function which reads in user input and saves the values to the firestore database.

2 Inspection Procedures

Checklist Reference:

java_checklist.pdf

3 Inspection Results

Johns Inspection Results:

For Thomas:

Time: 6:00pm

Date: 4/20/22

Results (items not checked off):

- Are descriptive variable and constant names used in accord with naming conventions?
    - There are a few confusing variable names such as concatList, lv, and tempMeal. Need to rename variables to be more descriptive to their purpose.
- Are there attributes that should be local variables?
    - fullNutrients.getJSONObject(j) could be assigned to a local variable since it is duplicated.
- Does every method, class, and file have an appropriate header comment?
    - The method does not have any header comments and lack comments in general. Should add some comments explaining the function of some code blocks since there is a lot of different processes happening.
- For each method: Is it no more than about 60 lines long?
    - The method is very long spanning 200 lines. Could extract functions in order to compartmentalize logic.

For Jay:

Time: 1:00pm

Date: 4/29/22

Results (items not checked off):

- Is every variable and attribute correctly typed?
    - Weight, height, and age should be typed integers.
- Does every method, class, and file have an appropriate header comment?
    - There are no header comments above the methods.
- Are there enough comments in the code?
    - Code in general needs more comments explaining chunks of logic.
- Does each switch statement have a default case?
    - There should be an additional if case the input check to see if the input is null. This should notify user of invalid input.

For Cristian:

Time: 1:30pm

Date: 4/29/22

Results (items not checked off):

- For each method: Is it no more than about 60 lines long?
    - The method is very long spanning over 400 lines. Could extract functions in order to compartmentalize logic and reduce repetition.
- Are there any computations with mixed data types?
    - Could potentially be mixing integer and string in updateButton listener.
- Are descriptive variable and constant names used in accord with naming conventions?
    - Some variable names like adapter4 could be renamed to be more descriptive.
- Is there repetitive code that could be replaced by a call to a method that provides the behavior of the repetitive code?
    - There is a lot repetitive code that could be replaced by extracting a function.
- Are there enough comments in the code?
    - There are very little to no comments in the code. Should add some to explain code blocks where logic is difficult to understand.

Thomas Inspection Results:

For John:

Time: 8:30 pm

Date: 4/29/22

Results:

- Are there variables or attributes with confusingly similar names?
    - Some variables like jObject are vague.
- Are descriptive method names used in accord with naming conventions?
    - No, sendAPIRequest() is vague. What is being requested?
- Is every method parameter value checked before being used?
    - No distance for sendPIRequest is not being checked and neither is map in onMapReady
- Does every method, class, and file have an appropriate header comment?
    - No, on map ready does not
- Does every attribute, variable, and constant declaration have a comment?
    - No, but this seems like it would be comment overkill. Should focus on descriptive variable names

For Cristian:

Time: 9:00 pm

Date: 4/29/22

Results:

- Are descriptive method names used in accord with naming conventions?
    - No, the method name is populateMealLog but it populates the workout log.
- Is every method parameter value checked before being used?
    - No, docRef is not being checked before being used.
- Does every method, class, and file have an appropriate header comment?
    - No, does not have a header comment
- Does every attribute, variable, and constant declaration have a comment?
    - No, variables do not have declaration comments.
- For each method: Is it no more than about 60 lines long?
    - PopulateMealLog function is incredibly long

For Jay:

Time: 9:30 pm

Date: 4/29/22

Results:

- Are descriptive variable and constant names used in accord with naming conventions?
    - No, nameButton is vague when I think it is the saveButton
- Does every method, class, and file have an appropriate header comment?
    - No, the onClickListener doesn thave a header comment.
- Does every attribute, variable, and constant declaration have a comment?
    - No, but descriptive variable names are used.
- For each method: Is it no more than about 60 lines long?
    - It's not super long but it's filled with many function calls so the setting of name and so on could be extracted to a function as well.

Cristians Inspection Results:

For Jay:

Time: 10:45 pm

Date: 4/28/22

Results:

- Are descriptive variable and constant names used in accord with naming conventions?
    - Yes, it is obvious that name refers to the user's name and so on.
- Are there attributes that should be local variables?
    - Yes, but I think local variables should be made into instance variables.
- Does every method, class, and file have an appropriate header comment?
    - No. It would be better to create an OnClickListener method and comment what it does, then set it to be the button's listener. That way the code in onCreate is more concise and the method can have documentation comments.
- For each method: Is it no more than about 60 lines long?
    - No. Although the method could have the conditionals eliminated if the null check was replaced with a default value. That would result in an even more concise method.

For Thomas:

Time: 10:34 pm

Date: 4/28/22

Results:

- Are descriptive variable and constant names used in accord with naming conventions?
    - No. If some of the code was factored out, then the variable names could be more obvious. The method is doing too much work - what do thePhoto, fullNutrients, jsonBody, lv have in common?
- Are there attributes that should be local variables?
    - Yes, but like I said previously, I think all local variables should be instance variables to better keep track of them.
- Does every method, class, and file have an appropriate header comment?
    - No, there is no header comment for the method.
- For each method: Is it no more than about 60 lines long?
    - It is very long, although a lot of lines are used as breaks between lines of code that are cohesive together. Good start for breaking up the function into lots of smaller ones.

For John:

Time: 1:12 pm

Date: 4/2/22

Results:

- Are descriptive variable and constant names used in accord with naming conventions?
    - No. Many of the variable names are very vague and hard to understand without prior knowledge of the code and what the function is meant to do.
- Are there attributes that should be local variables?
    - Yes, but I prefer making each local variable an instance variable. This lets it get accessed in multiple callback methods like onResume, it's easily tested, and all the declarations are in one place at the top of the code file.
- Does every method, class, and file have an appropriate header comment?
    - Yes for sendAPIRequest, no for onMapReady. Ideally there would be documentation comments for each method. sendAPIRequest is obvious enough on what it does. onMapReady could definitely be a callback called at any time the user clicks on the map, so it should definitely have comments.
- For each method: Is it no more than about 60 lines long?
    - onMapReady is relatively short, sendAPIRequest is very long. The bad parts are the conditionals and Overrides - these bloat up the screen and make it hard to interpret the control flow. Both methods should definitely be broken down into smaller methods.

Jays Inspection Results:

For John:

Time: 1:27 pm

Date: 4/29/22

Results:

- Are descriptive variable and constant names used in accord with naming conventions?
    - Yes, it is obvious that name refers to the user's name and so on.
- Are there attributes that should be local variables?
    - I think that all the variables that were declared and used exactly where they should have been.
- Does every method, class, and file have an appropriate header comment?
    - No. SendAPIRequest() does have a basic header comment, but uses comments sparsely inside the method making it easy for the inspector to get lost in the code. Especially with how long the code is. onMapReady() does not have any comments explaining the purpose of the function
- For each method: Is it no more than about 60 lines long?
    - onMapReady is short and under 60 lines long. sendAPIRequest() on the other hand is much longer and at 63 lines.

For Christian:

Time: 1:27 pm

Date: 4/29/22

Results:

- Are descriptive variable and constant names used in accord with naming conventions?
    - There are many variables that followed the naming conventions, but there are also some variables that were a bit more vague and less professional like having a variable called adapter2.
- For each method: Is it no more than about 60 lines long?
    - This method is very long, going over 60 lines by a great margin. It would have been better to split the function to have subfunctions to make it more readable.
- Are descriptive method names used in accord with naming conventions?
    - Yes, the method name was very descriptive and gave a clear idea of what the method was being used for.
- Could any non-local variables be made local?
    - No, I believe that all variables were defined exactly where they should have been defined.

For Thomas:

Time: 1:27 pm

Date: 4/29/22

Results:

- Are descriptive variable and constant names used in accord with naming conventions?
    - The variables are defined and named very aptly, and even with the many variables used in the function there is no room for doubt what exactly each variable is storing.
- Are the comparison operators correct?
    - Yes, there were a wide range of comparison statements inside the function that at first glance looked repetitive. But each statement tested the right thing properly and sent new information to the API
- For each method: Is it no more than about 60 lines long?
    - No, the function was much longer than 60 lines and quite hard to read. It would have been better to split up the function and reduce its length.
- Are descriptive method names used in accord with naming conventions?

- No. The function that made the API call is named very vaguely. However there are comments that let the reader identify what the string request code is doing.

V Recommendations and Conclusions

VI Project Issues

1 Open Issues

In the Calendar view, the meal list items are clickable. The menu that pops up when the meal item is clicked functions fine, but clicking the "Remove" or "Save" button crashes the app.

Sometimes the HomeFragment displays duplicates of each meal item. Removing one of them works fine, but removing the second copy crashes the app because no such meal is found.

Pressing the back button while in the Calendar view does not always update the HomeFragment to match the date that was last selected in the Calendar view.

2 Waiting Room

All of the user's meal and workout history is computed when they log in. It is stored in 4 separate static data structures that are accessed through the app - MealArrayList, SortedMealArrayList, WorkoutArrayList, and SortedMealWorkoutList. Each fragment has its own code that loops through each sorted list and adds a meal/workout to its TodayArrayList if the meal or workout matches that fragment's Date. This can be optimized by creating 2 additional static map data structures: WorkoutMapByDate and MealMapByDate. The keys will be the toString values of each Date. The values will be a "TodayArrayList" that has all of the meals/workouts from that Date stored in it. This way each fragment can hash into the map by Date in constant time, which would avoid the lag that occurs when each new fragment has to run the same exact operation looping through the sorted lists.

The Search menu currently returns all items that match a query. It would be more efficient and less wasteful of the user's 4G data to only return say the first 10 items that match the query. Then include a button at the end of the search list that says "Load more results" that the user can click on.

The editable meal list items only show part of the total nutritional info that we store. Ideally clicking on a meal would allow you to edit every single nutritional item, from vitamins to sugars to macros to calories.

Both the editable meal list items and the profile picture in the ProfileFragment are not editable by the user. Ideally the user can upload whatever picture they want. It would probably be safer to prevent the user from pushing this change to the database, so it would need to be saved locally.

When the user logs in, their entire meal and workout history is downloaded from Firebase. It would be more efficient for only the past few weeks or months to be downloaded, and the rest to be downloaded in batches upon necessity, like when the user starts scrolling back in the Calendar view.

The Calendar view is currently only scrollable by months. Ideally you should be able to click where it displays the month and year to open up a new menu that lets you scroll by years, then decades.

The preferences only work to filter out the restaurant meals. There is no problem in adapting the preferences to work for the meals returned from the Search menu; this just was not done to save time. Ideally the user should be able to blacklist the meals retrieved from Search and sort them by their preferences.

NutritionIX can be queried upon each letter the user types in. This was not done because it would have pushed us above our 1000 searches per day limitation. The search suggestions that are mere strings like “Eggs” and “Bacon” in our app could be replaced by NutritionIX’s search suggestions.

3 Ideas for Solutions

Regarding the crashing in the Calendar view when the “Save” or “Remove” button is clicked: In the HomeFragment the current date’s meals are sorted into a “TodayArrayList.” The meals are not sorted in the Calendar view. Sorting them will at least rule out that no indexing error is occurring.

Regarding the HomeFragment displaying a copy of all the meals: This is likely due to the “TodayArrayList” not being cleared before it is used again to recompute an addition or removal of a meal. Changing the populateMealLog() method so that a .clear() is run on the ArrayList will likely fix this issue.

Regarding the back button not always updating the HomeFragment’s date: This can be fixed by overriding onBackPressed and copying the code that currently exists in onOptionsItemSelected.

4 Project Retrospective

We met weekly and planned out what each of us would do for the next week. We also updated each other via Discord when we made major commits. We also communicated when we would be working on the same Java file if there were ever any conflicts. We made constant pushes to Github so there was never much conflicting edits to files. We also had our own branches to work on to test code before

submitting something completely broken to the master branch. Using Google docs also made communication easy.

The only time there weren't steady additions to the app was during the first week when we were just getting the project set up and during midterm weeks. Every other week including spring break week had a lot of additions. We had a lot of success using the Kanban board because it made it obvious which stories our members were working on. If they were "in progress" or "done," the story was clearly taken, but the stories in the "to-do" column were contestable. No one cared if they were giving up a story they hadn't started on.

One possible improvement could have been to learn about TDD earlier in the semester so we could apply it to our project. We had a lot of weird confusing bugs that were generated and designing tests first or at least alongside our project could have saved us some time. But overall everyone was satisfied with the work we got done.

VII Glossary

Macros: Short for macronutrients and are the three categories of nutrients you eat the most and provide you with most of your energy: protein, carbohydrates and fats.

Calories: A unit that measures the amount of energy in a food, most diets are based around calorie consumption in a day with standard numbers being around 2000. Can be shortened to Cals.

Protein: A macronutrient needed by the human body for growth and maintenance.

Fats: A class of macronutrients used in metabolism called triglycerides.

Carbohydrates: A macronutrient that is used to produce sugars for the body.

Diet: A restriction of certain foods and macros in a person's diet, mostly used for losing weight or maintaining a certain weight. Common diets could be a low carb diet which heavily restricts daily carb intake while aiming for high protein and fat intake instead, and also limiting the user at 1900 cals a day.

Meal: Meal in the context of this application refers to the order of food that the app recommends the user to get from a restaurant. Meals fit in a user's macros by default and the app will not recommend a 'bad' meal which does not fit in their diet plan.

Net Daily Calories: The number displayed by the Daily Calorie Bar in the user's profile and in the calendar view. This is calculated by Daily Calories Consumed - Daily Calories Burned.

Daily Calories Consumed: Each meal item saved comes with a calorie count. The Daily Calories Consumed is the addition of all meals's calories consumed in a single day from 12:00 am to 11:59 pm.

Daily Calories Burned: Each workout item saved comes with a calories burned count. The Daily Calories Burned is the addition of all the workouts' calories burned performed in a single day from 12:00 am to 11:59 pm.

Weekly Calories Consumed: The number displayed on the user's profile that adapts according to which meal plan they choose. The weekly calories consumed is calculated by summing the Daily Calories Consumed for the past 7 days, including the current day.

Weekly Calories Burned: The number displayed on the user's profile that adapts according to which workout plan they choose. The weekly calories consumed is calculated by summing the Daily Calories Burned for the past 7 days, including the current day for the user.

Workout Plan: The plan the user chooses to modify their weekly exercise goals.

Meal Plan: The plan the user chooses to modify their weekly caloric, protein, carbohydrate, and fat goals.

Location: The user's latitude and longitude pulled by Google Maps.

Meal Log: All of the "Meals" tab shown in HomeFragment. It is composed of subfragments that populate lists of meals and contains "Breakfast," Lunch," Dinner," and "Snack" divisions.

Workout Log: All of the "Workouts" tab shown in HomeFragment. It is composed of subfragments that populate lists of workouts and contains "Weightlifting," "Cardio," and "Yoga" divisions.

Restaurant: Any item that gets returned when the user updates their location in the MapFragment. Restaurants are shown on the map, listed below the map for the user to view, and when clicked display menus.

Menu: The totality of items that get returned when the user clicks on a restaurant. The menu contains all of the available restaurant meals the user can add to their Meal Log.

VIII References / Bibliography

[1] Robertson and Robertson, Mastering the Requirements Process.

[2] Hansana, Andy, et al. Chicago, IL, 2021, pp. 1–65, *Fitness and Nutrition Buddy Project*.

[3] J. Bell, "Underwater Archaeological Survey Report Template: A Sample Document for Generating Consistent Professional Reports," Underwater Archaeological Society of Chicago, Chicago, 2012.

[4] M. Fowler, UML Distilled, Third Edition, Boston: Pearson Education, 2004.