GYMGO

A gym class management app

Neal Boylan (20104310)

Higher Diploma in Computer Science

Supervisor: Anita Kealy

DECLARATION

I declare that the work which follows is my own, and that any quotations from any sources (e.g. books, journals, the internet) are clearly identified as such by the use of 'single quotation marks', for shorter excerpt and identified italics for longer quotations. All quotations and paraphrases are accompanied by (date, author) in the text and a fuller citation is the bibliography. I have not submitted the work represented in this report in any other course of study leading to an academic award.

Student..... Date 13/4/25

Work Place Mentor..... Date .....

ACKNOWLEDGEMENTS

As someone who has dropped out of as many degrees and courses as I have finished, I am proud to have completed this one - there were many occasions where I wanted to throw in the towel! My friends, family, and colleagues all offered nothing but encouragement despite having to listen to me constantly complaining about assignments and projects.

A special thank you goes to my girlfriend Sarah who did the lion's share of travelling to see me despite having her own studies and work to stay on top of.

I would also like to thank my supervisor Anita Kealy for keeping me focused over the last 3 months, and for the expert advice and guidance.

Thank you to all the lecturers on the course for their fantastic content. Everything was made to a very high standard and tough but fair as it should be.

Special thanks to Dave and all the coaches and members in SDSC. You inspired me to make this project and it is something I hope to someday make available for you all to use.

Finally, this project is dedicated to my cat Theo who sadly passed halfway through the course. He kept me company during the late-night coding sessions and was a calming influence when I needed it most.

1 RESEARCH & ANALYSIS

1.1 BACKGROUND

The reason for developing this app is to make life easier for owners and trainers in small class-based gyms. I work in a gym that has around 100 members and offers classes for various training modalities; some classes focus on improving strength, while others focus on cardio. There is a limit on the number of members that can attend each class, so there needs to be a way for them to sign-in to ensure the sessions aren't over capacity. This also gives the gym owner feedback on which times are busy and lets them know if there are any members who they should follow up with due to low attendance. We are currently using an app called Glofox to manage class sign-ins, but it doesn't provide all the functionality I would like. I want to create a similar app that also allows the members to upload data on the exercises they did and possibly add photos/videos. This data would then ideally be used to create charts for both the trainers and members to visualize someone's progress.

1.2 OBJECTIVES

GymGo could be a mobile, web, desktop, and embedded app, but the scope of this project is to develop a mobile app that runs on iOS and Android phones. The app will include the following features:

- user account creation for different user types
    - gym owner
    - gym coach/trainer
    - gym member/trainee
- user authentication
- the owner can

    - create and edit class lists
    - view and edit trainer and member information
- trainers can
    - mark members attendance for classes
    - view member workouts
- members can
    - view and sign-in to classes
    - add information about their workout
    - view stats regarding their progress
- CRUD for
    - classes, coaches, members by the gym owner
    - member workout data by the member, trainer, and gym owner

The user interface is a list of cards containing basic information – class time, class focus (strength, cardio, open), class size, number of spaces available

1.3 MODEL

The gym owner creates many gym classes.

Each member of the gym can join many classes.

Each member can create many workouts.

Each workout can have many exercises

Classes are coached by one coach, and the coach can instruct many classes

For each member we record memberId, name (first and last), email, phone number, and date of birth

For each gym class we record classId, description, date, time, duration, and size

For each workout we record workoutId, date, time, and a list of exercises

For each exercise we record exerciseId, exerciseName, sets, reps, and weight

For each coach we record coachId, name (first and last), email, phone numbers, and date of birth

Entities:

- Owner
- Member
- Class
- Workout
- Exercise
- Coach

Relationships:

- Owner creates many gym classes, gym classes can only be created by the owner
- Owner 1..1 creates 0..* Gym Classes

- Member joins many gym classes, the gym classes can have many members
- Member 1..* joins 0..* Gym Classes

- Each workout logged by a member is recorded, and each workout belongs to one member
- Member 1..1 logs 0..* Workouts

- Each exercise is part of one workout, and each workout can have many exercises
- Workout 1..1 contains 1..* Exercises

- Gym classes are coached by one coach, and a coach will instruct many gym classes
- Coach 1..1 coaches 1..* Gym Classes

Entity Type attributes:

Member: memberId, name (firstName, lastName), email, phone, dateOfBirth

GymClass: classId, description, date, time, duration, size

Workout: workoutId, title, date, time

Exercise: exerciseId, name, sets, reps, weight

Coach: coachId, name (firstName, lastName), email, phone, dateOfBirth

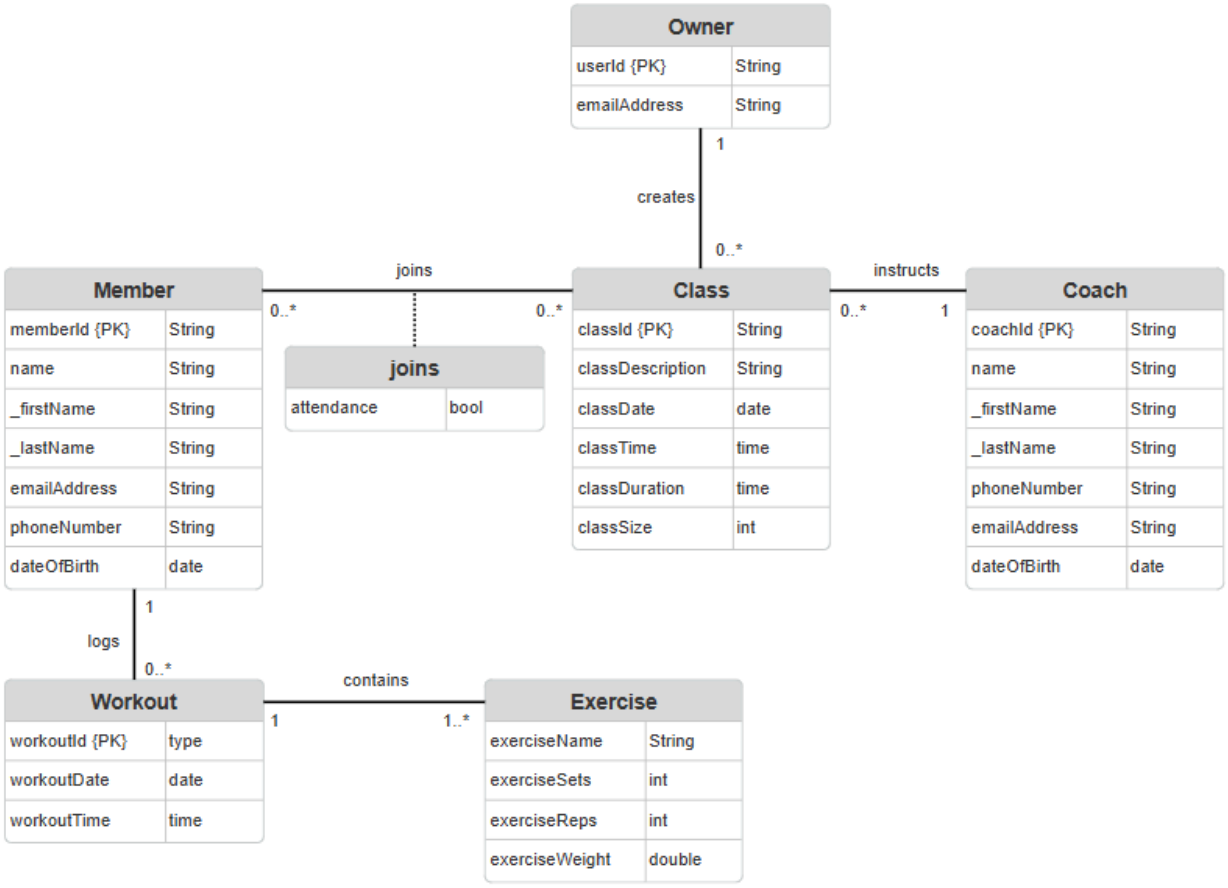


Figure 1-1: ER Diagram

1.4 TECHNOLOGY

1.4.1 FRONTEND

1.4.1.1 NATIVE vs PROGRESSIVE

I learned how to develop Android apps in the last semester of the hdip course, so I had a good idea of what it would take to make a useful mobile application. The downside of native apps being that they will only work on a single set of devices, and I would need to build the app again using a different technology if I wanted it to be available for use on iOS devices. From my own research and observation, I knew that I would need a solution that enabled me to use the same code to develop apps that would work across all phones. I began searching for solutions and found that there are several options to help make this happen [1] [2] [3]. Figure 1-2 shows some of the options and their popularity.

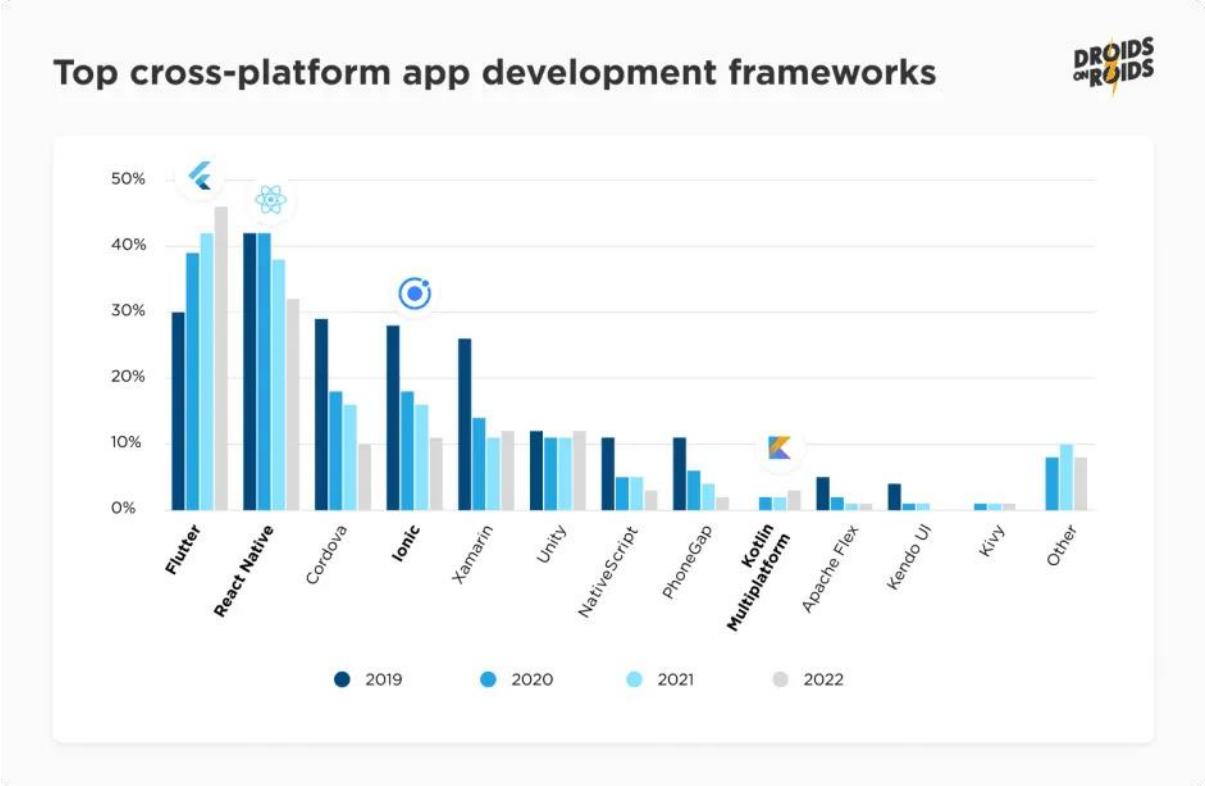


Figure 1-2: Cross platform development frameworks by popularity

1.4.1.2 FLUTTER

Flutter was released by Google in 2018. It is an open-source framework that supports app development on six platforms: Android, iOS, Windows, MacOS, Linux, and web. Flutter apps are made up of widgets, which can be easily customised and scaled by the developer. These widgets can also be styled using Cupertino (to give them an iOS feel), or Material Components (for more of an Android feel) (*Figure 1-3*).

```
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
@override
Widget build(BuildContext context) {
return MaterialApp(
title: 'is flutter better than react native',
theme: ThemeData(
primarySwatch: Colors.white,
),
home: MyHomePage(title: 'Flutter and React Native App'),
);
}
}

class MyHomePage extends StatelessWidget {
final String title;
const MyHomePage({Key? key, required this.title}) : super(key: key);

@override
Widget build(BuildContext context) {
return Scaffold(
appBar: AppBar(
title: Text(title),
),
body: Center(
child: Text(
'BrowserStack Flutter vs react native: HomePage ',
),
),
);
}
}
```

Figure 1-3: Example of the nested style Dart code

Pros

- Can easily be coded to change style depending on device OS
- Small number of widgets needed to create UI
- Dart programming language relatively easy to learn

- Great documentation and learning resources
- Hot reload function to speed up development

Cons

- Widget trees can become unwieldy
- Learning a new programming language
- Large app size

1.4.1.3 REACT NATIVE

React Native is an open-source mobile application framework created by Meta, first released in 2015. React Native apps are typically built using JavaScript or TypeScript. React Native is component based and uses JSX syntax to build the UI, while relying on native iOS and Android components instead of its own UI components (*Figure 1-4*).

```
import * as React from 'react';
import { Text, View, Button, StyleSheet } from 'react-native';
import 'react-native-gesture-handler';
import { NavigationContainer } from '@react-navigation/native';
import { createStackNavigator } from '@react-navigation/stack';

const AppBar = createStackNavigator();

export default function App() {
return (
<>
<NavigationContainer>
<AppBar.Navigator initialRouteName="App Title">
<AppBar.Screen name="Flutter and React Native app" component={HomeScreen} />
</AppBar.Navigator>
</NavigationContainer>
</>
);
}

function HomeScreen() {
return (
<View style={style.container}>
<Text>BrowserStack Flutter vs React Native: HomePage</Text>
</View>
);
}

const style = StyleSheet.create({
container: {
flex: 1,
alignItems: 'center',
justifyContent: 'center',
},
});
```

Figure 1-4: Example of React code written in TSX

Pros

- Uses a well-established language that I'm familiar with
- Code is more readable
- Large community
- Fast refresh function to speed up development
- Extensive Third-Party libraries
- Smaller app size

Cons

- Uses a bridge to connect to native components, resulting in slower development and running times
- Documentation slightly more confusing to navigate

1.4.1.4 KOTLIN MULTIPLATFORM

KMP is a SDK by JetBrains that enables developers to create cross-platforms apps. It was released as part of Kotlin 1.2 in 2017. It requires the developer to create separate frontends that share a common business logic. It is a relatively new technology, but something I considered due to previous experience with Kotlin and Java.

Pros

- Uses Kotlin language
- Consistent architecture across all platforms
- Access to native libraries

Cons

- Requires building separate UIs
- New, less developed technology

- No hot reloads

1.4.1.5 IONIC

Ionic is an open-source UI toolkit for building cross-platform applications. It makes use of technologies such as HTML, CSS, and JS/TS, and was originally released in 2013.

Pros

- Builds on popular pre-existing languages, making it easy to learn for web developers

Cons

- Less control over the UI
- Relies on plugins for native integration
- No Hot Reload feature
- Primarily used for web apps with mobile app secondary

1.4.2 BACKEND/DATABASE

Having decided on my frontend technology, the next step was to decide what to use for the backend. There were two options that I thought would be feasible for this project.

1.4.2.1 NODE.JS

I had worked with node while building web applications as part of this course. Node.js is a server-side runtime environment that allows JavaScript to be executed. With its non-blocking I/O and event-driven architecture, Node.js is designed for building scalable network applications. [4]

Pros

- Flexible. The backend architecture can be structured any way the developer desires
- Performance. Node uses a V8 engine and can handle many simultaneous connections
- Ecosystem. Developers can use npm to access a large library of modules and packages

Cons

- Difficult to learn
- Server management must be handled manually
- Time consuming setup

1.4.2.2 FIREBASE

I used Firebase as part of the Mobile App Development module so I had more recent experience working with it. It offers multiple services including storage and authentication alongside acting as a database

Pros

- No server management
- RTDB that allows for syncing across all clients
- Multiple authentication methods
- Easily scales to handle growth

Cons

- Is only free up to a point. Costs can spiral as app grows
- Basic backend logic

2 DESIGN

2.1 SCREEN FLOW

There will be 3 user types for this app (owner/admin, coach, and member), therefore there will be 3 different navigation models.

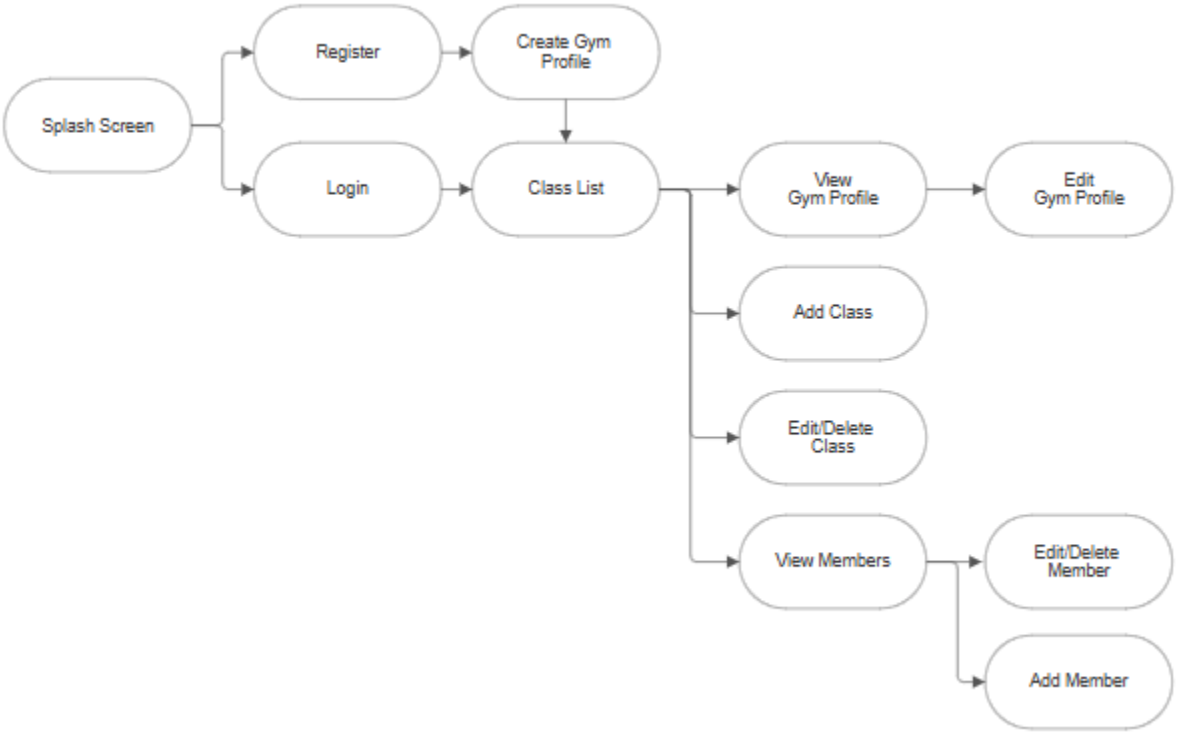


Figure 2-1: Page navigation for Owner/Admin user

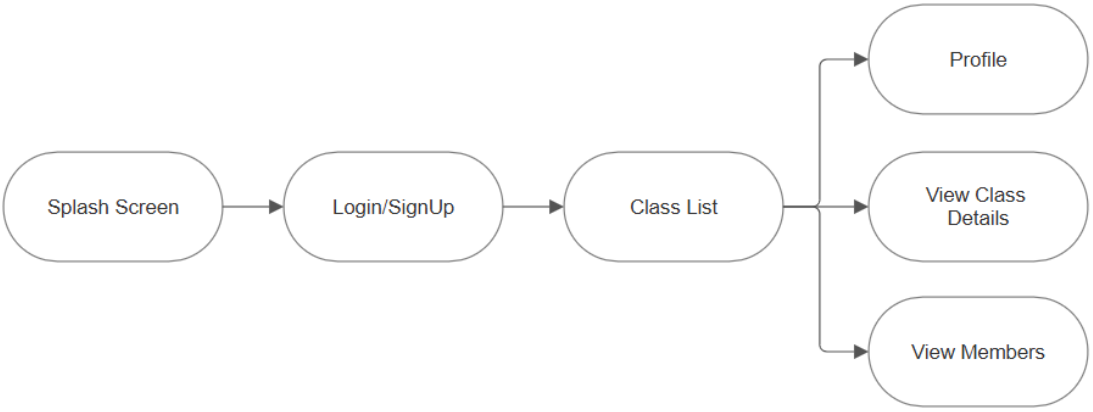


Figure 2-2: Page Navigation for Coach user

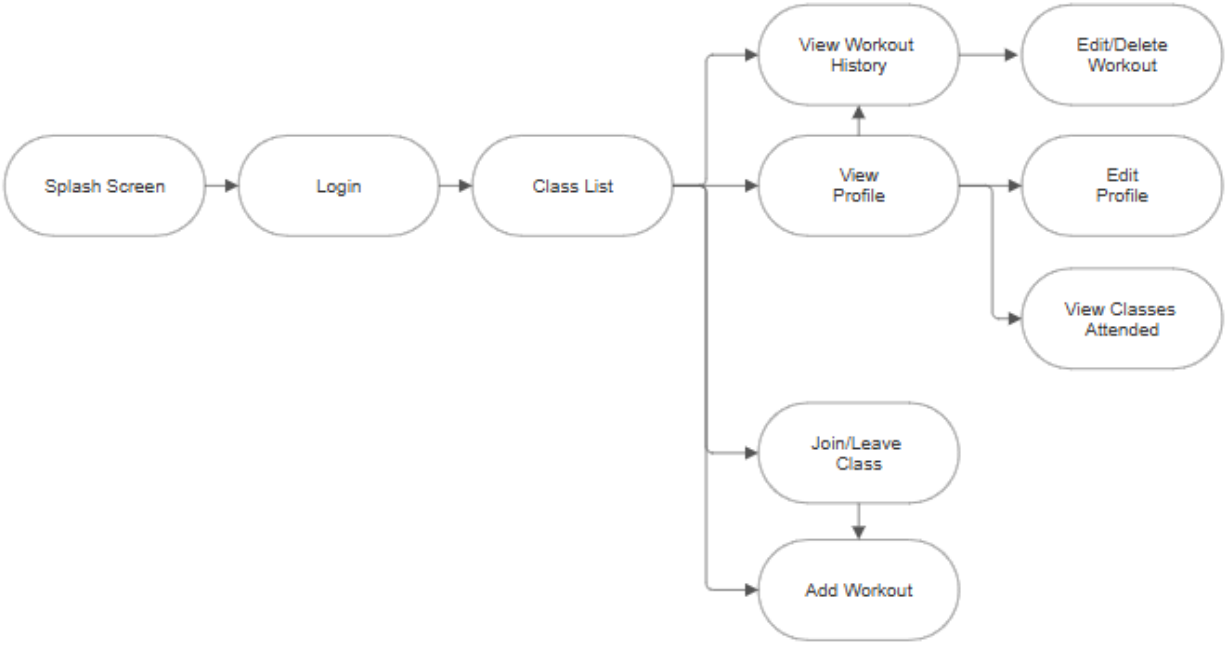


Figure 2-3: Page Navigation for Member User

2.2 USER INTERFACE

Logo created using Looka.com. All screen designs built using Figma

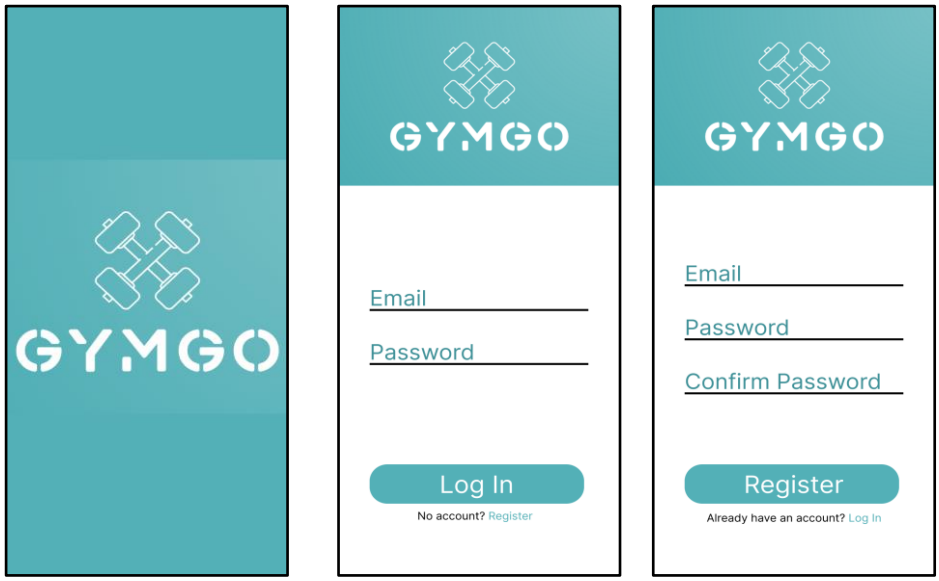


Figure 2-4: Splashscreen and login/singup screen designs

The following screens (*Figure 2-5*) are for the admin/gym owner. They can use a floating action button (FAB) to add a class or use the navigation bar to view the member list. In the member list they can view each member's details.

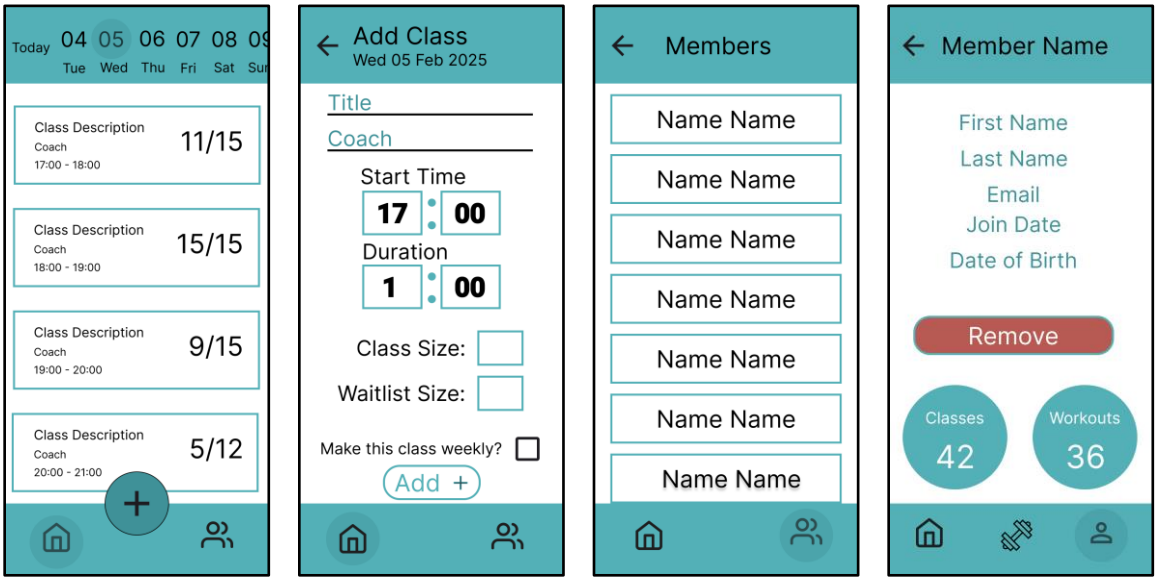


Figure 2-5: Screen designs for admin/gym owner

The following screens (*Figure 2-6*) are for the member user. Their home screen displays a list of available classes. The member can click on a class card to sign in or join the waitlist. After a class has started, clicking on the card will allow them to add exercises and submit a workout

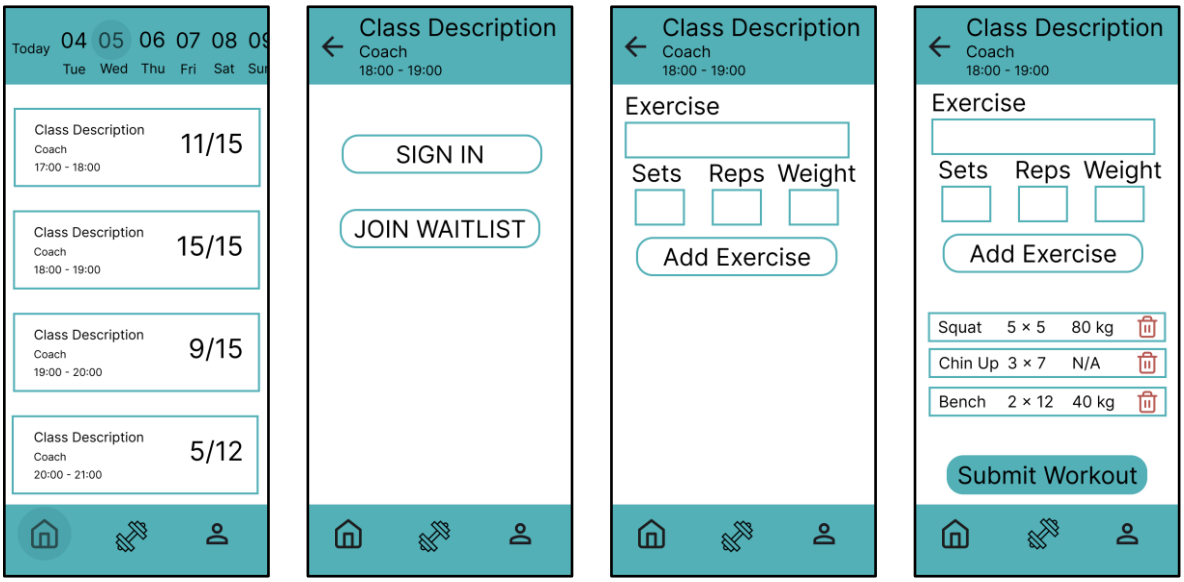


Figure 2-6: Screen designed for gym member to sign in to class and add workouts

Using the nav bar, the members can view all their logged workouts. Clicking on a workout card will let the member view/update/delete (*Figure 2-7*).

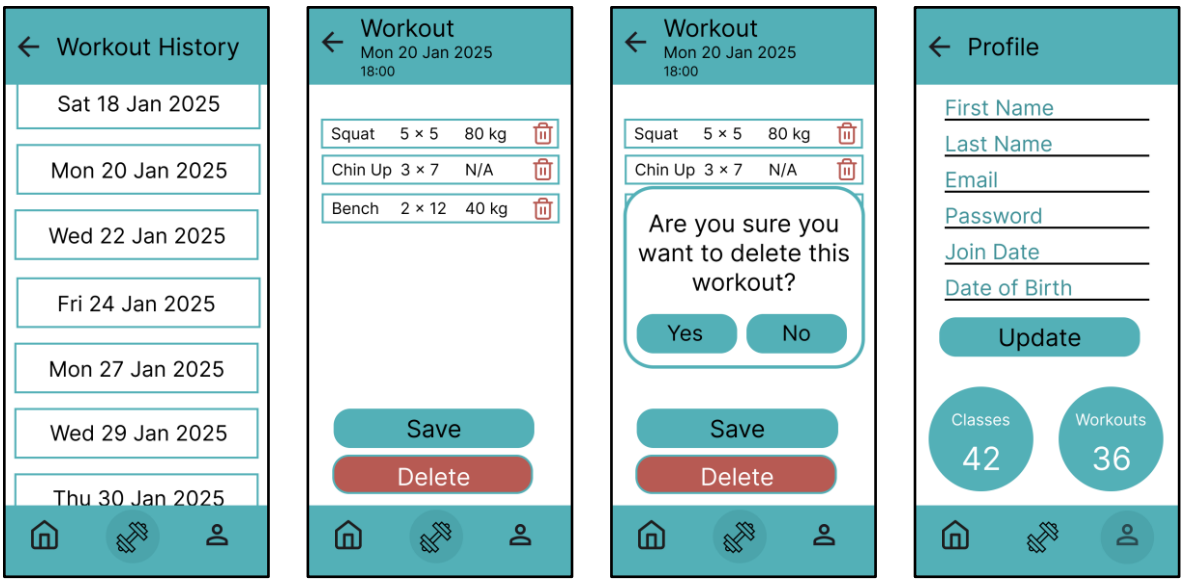


Figure 2-7: Screen designs to view and edit workouts

3 ITERATIONS

My plan is to develop the project using the Agile method. I will build a series of versions of the app that get progressively more complex

3.1 VERSION 1

Allow the gym owner (admin) to sign in and create, update, and delete gym classes. The gym classes will be displayed as a list of cards displaying the class time, duration, coach, and purpose. There will be a date selector at the top of the screen, and a FAB to redirect to the gym class creation screen. This information will all be stored in a firebase DB. This work is planned to be carried out in sprint 3.

3.2 VERSION 2

Add an additional user type (member). Members will not be able to create or edit classes. Member accounts are created by the admin. A member can log-in, and the sign-in/sign-out of classes that the gym owner has posted. The class cards will now indicate the size of the class, and how many spots have been filled. This work is planned to be carried out in sprint 4.

3.3 VERSION 3

Add yet another user type (coach). Coaches can click on the gym class cards to see which members have signed in to the class (as another list of cards). The coach can then mark them present. Coaches can see all classes, but can only click on classes that have been assigned to them by the admin. This work is planned to be carried out in sprint 5.

3.4 VERSION 4

Allow members to add a workout. This can be done by clicking on a gym class card that they are signed in to. This will bring up a screen with inputs for ‘exercise’, ‘sets’, ‘reps’ and ‘weight’, along with a button to add additional exercises and a button to submit the workout. These workouts will be saved to the DB with the user id. Each member will be able to access their workouts using the navigation bar at the bottom of the screen. The workouts will be displayed as a list of cards arranged by date. This work is planned to be carried out in sprint 6.

3.5 VERSION 5

Allow the admin (and possibly coaches) to view the list of members and click on their name to view all their workouts. This work is planned to be carried out in sprint 7.

4 DEVELOPMENT

4.1 SPRINT 1

6th January – 19th January

The first sprint of this project was spent researching the advantages and disadvantages of different app building technologies. Once I made my decision to use the Flutter framework with a Firestore backend, I started learning how the basics by working through a book called “Flutter for Dummies’ and reviewing my notes from the Mobile App Development module to brush up on my firebase skills.

4.2 SPRINT 2

20th January – 2nd February

I discussed potential features of the app with the owner of the gym I work in, as well as with the members. At this stage I built up wireframes to get an idea of what the app pages might look like and designed the data model.

4.3 SPRINT 3

3rd February – 16th February

To develop the first iteration of GymGo, I started by creating a basic app that was connected to Firebase for authentication - I preferred to start working with the remote database from the start instead of storing data locally before migrating. This required me to create a firebase project and connect it to my app [7]. I created the project to work with Android, iPhone, Windows, and web applications as I would ultimately like to be able to use this app across all platforms.

The initial code for this stage was based on a tutorial by Rivaan Ranawat [5] [6]. I reformatted this code from a 'To Do List' to represent a 'Gym Class List' list instead.

4.3.1 USER AUTHENTICATION

Initially the User Authentication is only for the admin or the 'Gym Owner', gym members will be added later. The gym owner registers their account on the signup_page and can then log in via the login_page. Registration is done using email and password (no Google sign-in etc.). To handle the user interaction on the sign-up and login pages, I made a file named auth_service.dart (*Figure 4-1*). This auth-service class contained functions that communicated with the firebase project, and had some error checking and validation features to ensure the credentials are accurate and secure. Sign-in and Login features are normally trivial aspects of an application, but for this project it was important the functionality. *Figure 4-2* below shows where a gym owner would create an account for their gym. Members and coaches do not create their accounts using this page – their accounts are created by the gym owner, after which the member/coach can select their gym from a dropdown menu and log in.

At this point in development the dropdown menu for selecting your gym is just a placeholder and isn't functional.

Figure 4-1: auth_service.dart code linking to firestore for user authentication

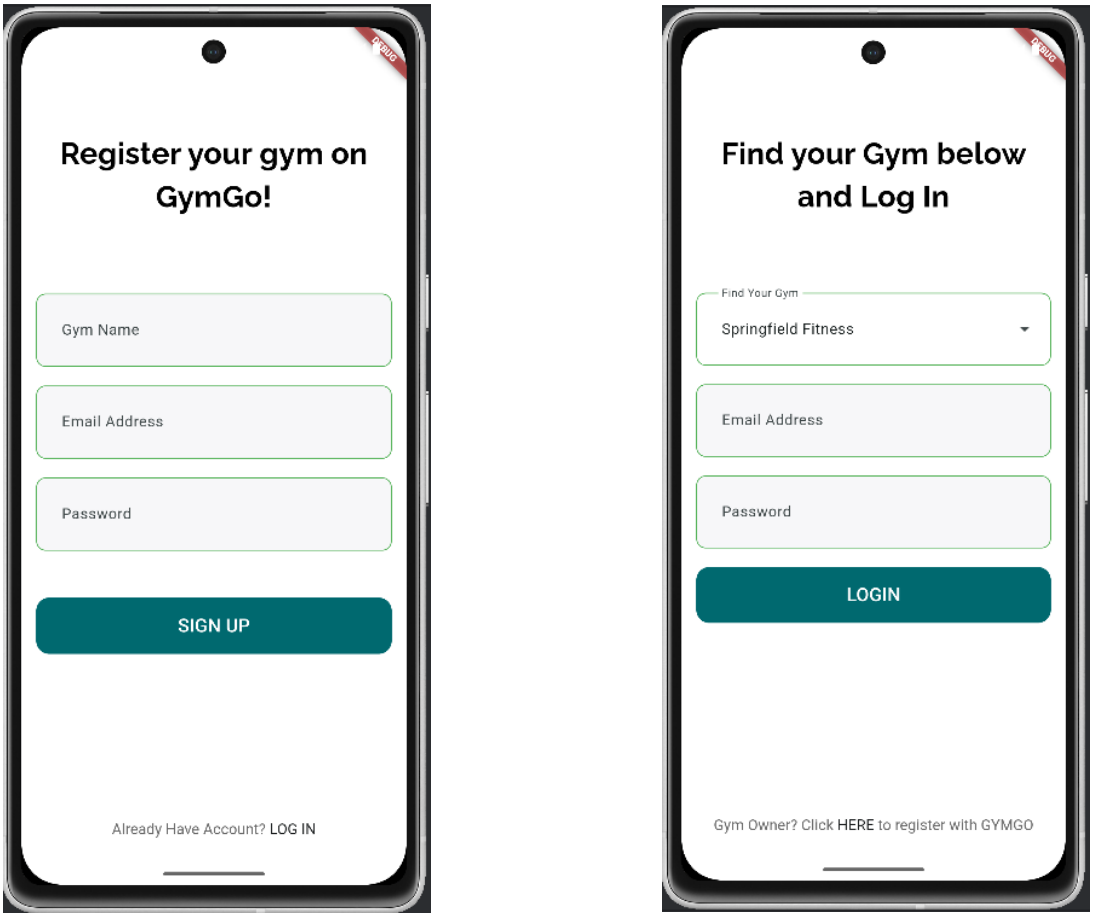


Figure 4-2: Screen to register new gym (left) and screen to log in to a gym (right)

4.3.2 HOME PAGE

Once logged in, the user is directed to the home_page. The home_page (*Figure 4-3*) consists of

- app bar
- DateSelector()
- ClassList()
- navigation bar
- Floating Action Button to add new class

The app bar simply has a title displaying the active page, and a small icon in the top right to make signing out of the app easy during development.

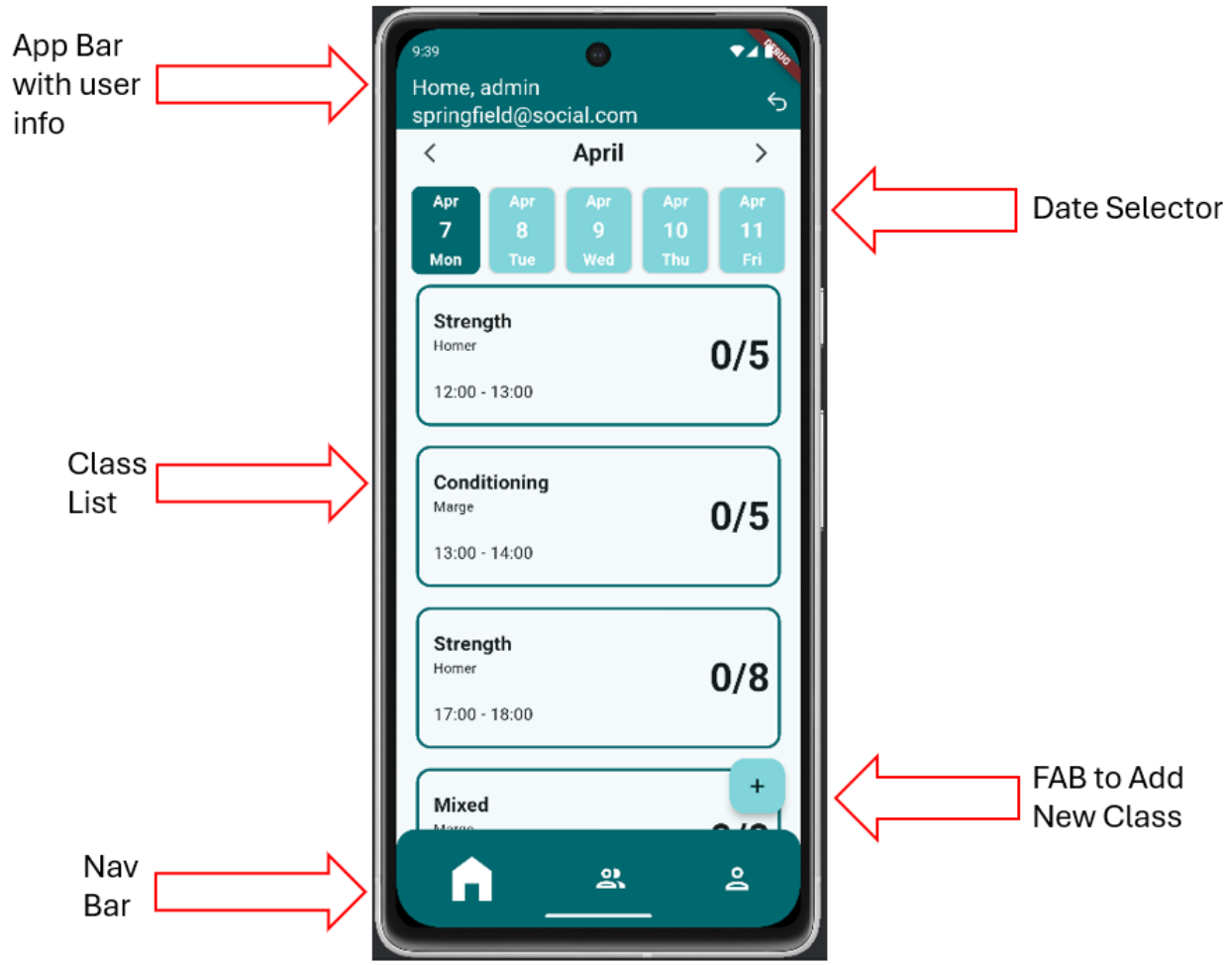


Figure 4-3: GymGo home page

DateSelector() is a widget that shows a scrollable row of dates and highlights whatever date is selected by the user. This widget was taken from the code by Rivaan Ranawat [5] [6], but slightly modified to show the month in 3 letters above the day on the tabs (*Figure 4-4*).

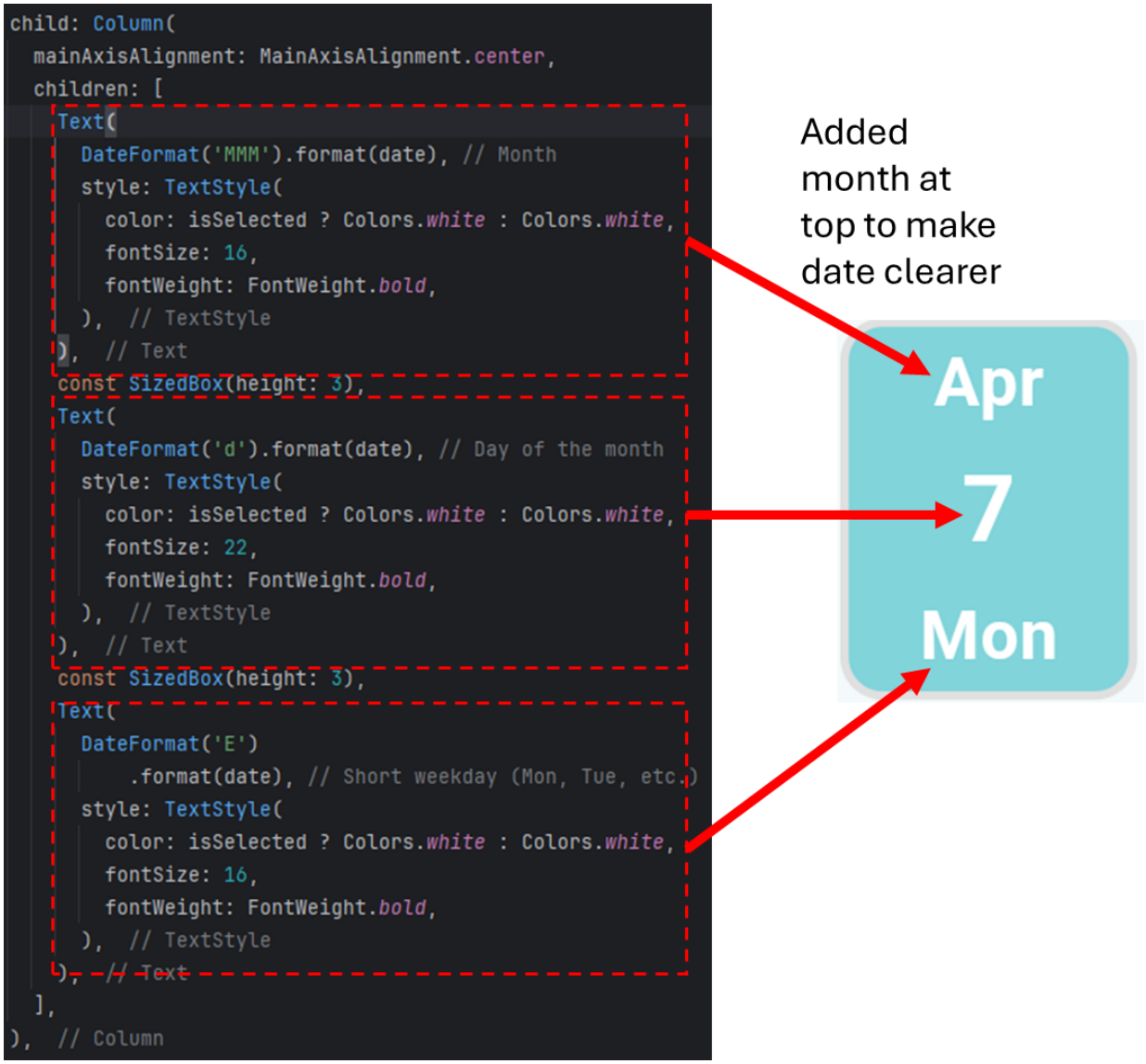


Figure 4-4: Code added to date selector widget

4.3.3 GYM CLASS LIST

The ClassList() class displays all gym classes as a scrollable column of cards. I made a class.card widget that could be added to a list and display information about individual classes (*Figure 4*-5). Using the selectedDate state that has been set by the DateSelector() widget, it will interact with the database to display classes that match the specified date - if there are no classes, a message will be displayed to indicate it (*Figure 4-6*).

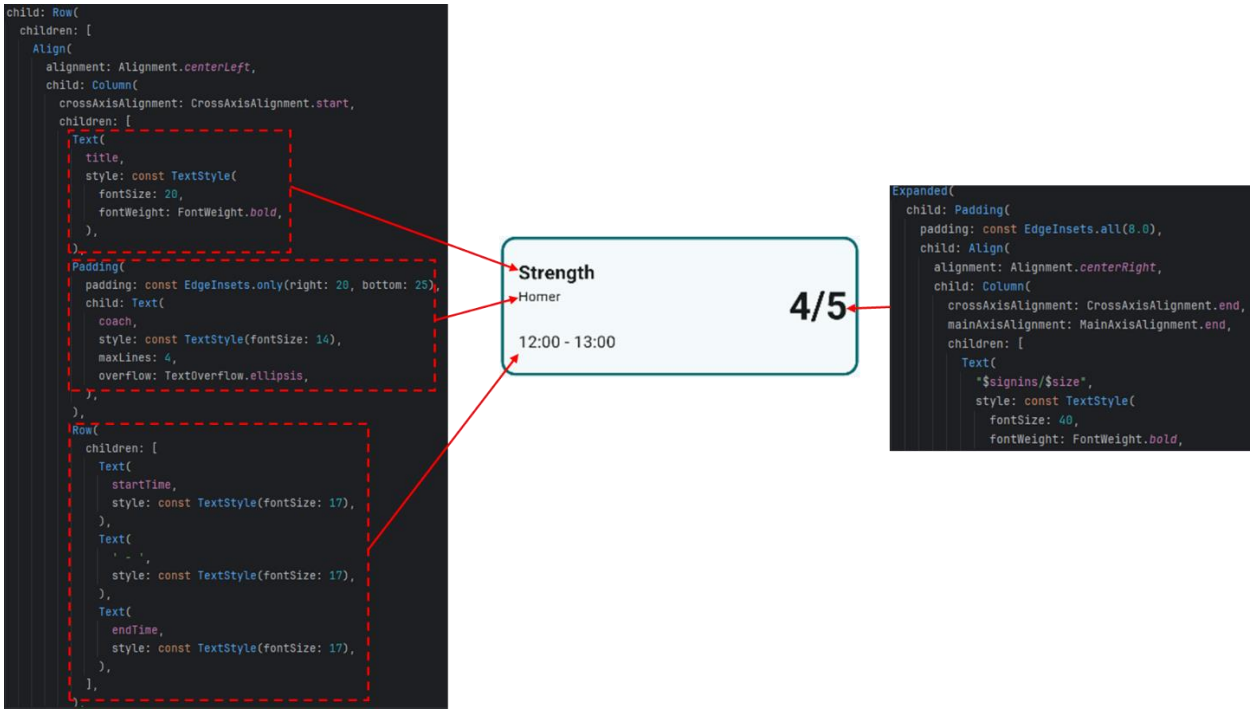


Figure 4-5: class_card widget and associated code

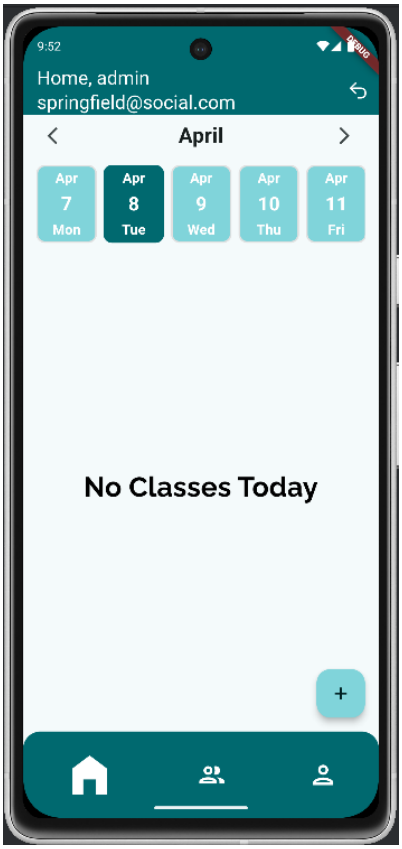


Figure 4-6: Home page on date with no scheduled classes

4.3.4 ADD GYM CLASS

To add a gym class to the database I employed a floating action button (FAB). Upon clicking the FAB, the user is redirected to the add_new_class page. This page contains text input fields to specify the title, coach, and size of the gym class (*Figure 4-7*). There are also options to select the start time and end time. I used the built in showDatePicker() [8] and showTimePicker() [9] functions to set the values. There is also a checkbox to indicate if the gym class will be recurring on a weekly basis - this button had no functionality at this point so is currently in as a placeholder for future iterations.

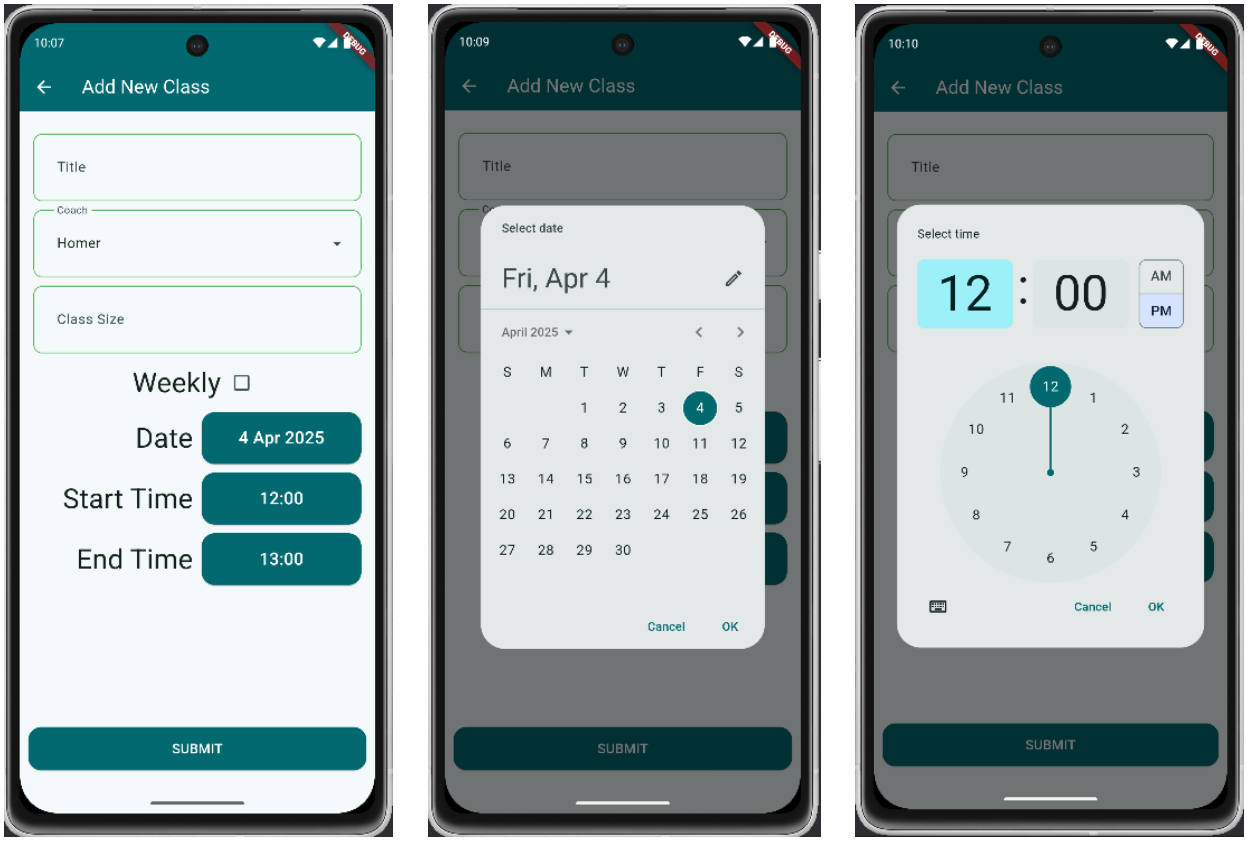


Figure 4-7: Add new class page (left), class date selector widget (centre), and class time selector widget (right)

The data is then sent to the database using the uploadClassToDb() function, which will create a document like *figure 4-8*.

```dart
Future<void> uploadClassToDb() async {
  try {
    int index = coachNameList.indexOf(selectedValue.toString());
    String coachId = coachIdList[index];

    await FirebaseFirestore.instance.collection("classes").add({
      "title": titleController.text.trim(),
      "coach": selectedValue,
      "coachId": coachId.toString(),
      "size": int.parse(sizeController.text.trim()),
      "startTime": startDateTime,
      "endTime": endDateTime,
      "weekly": weekly,
      "signins": [],
      "attended": [],
      "gymId": StaticVariable.gymIdVariable,
    });

    final snackBar = SnackBar(
      content: const Text('Class added'),
      action: SnackBarAction(
        label: 'Undo',
        onPressed: () {
          // Some code to undo the change.
        },
      ),  // SnackBarAction
    );  // SnackBar
    if (context.mounted) {
      ScaffoldMessenger.of(context).showSnackBar(snackBar);
    }
  } catch (e) {
    print('e: $e');
  }
}
```

Figure 4-8: Gym Class document on Firebase

4.3.5 VIEW/EDIT GYM CLASS

At this stage of development there are no gym members to sign in to the classes, so all classes are empty when viewed. From this page the admin can navigate to the Edit Class page. I used the add_new_class page as a template but implemented the firestore update function (along with the id for the gym class being edited) instead of the add function (*Figure 4-9*).

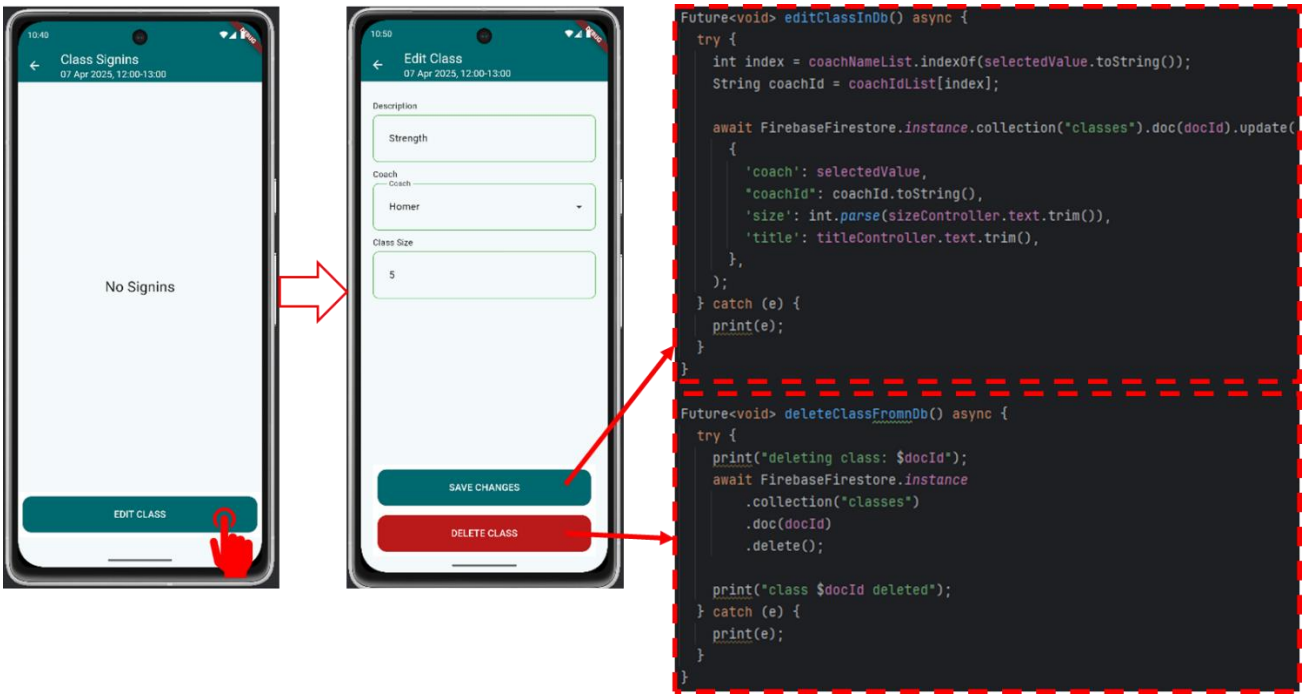


Figure 4-9: View Class and Edit Class pages

4.4 SPRINT 4

17th February – 2nd March

For the second iteration of my app, I created a new user type (member), and gave them the following features:

- View all gym classes
- Sign in to gym class
- Sign out of gym class
- Update profile

4.4.1 ADD MEMBER

I made a new page for the admin called add_new_member (*Figure 4-10*). This page lets the admin input the new member's email address and assign them a placeholder password (that the member can then change once they have logged in). I once again used the firestore auth function *createUserWithEmailAndPassword* to add the new user

and give them credentials to login to the app, but I also added their details to the firestore database in a collection called ‘members’. I had to use the firestore *set* method rather than the typical *add* method as I wanted to make sure that the member document id in the database matched the user id in the authorisation details (*Figure 4-11*). This ensured that the document didn’t get assigned a random id. Doing so made it easier to check if the logged in user is a member or an admin.

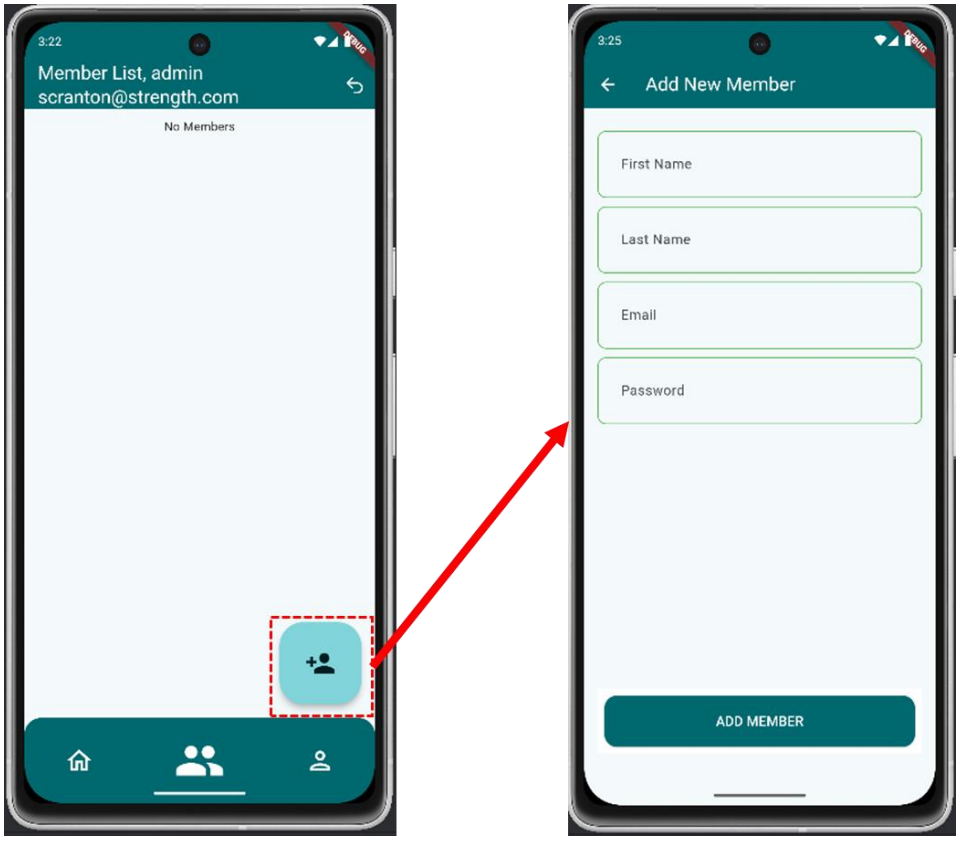


Figure 4-10: FAB directs admin to add_new_member page

Figure 4-11: Add the new member to the Firestore database and make sure it has the same id as the newly created user in the Firebase Auth

4.4.2 MEMBER PROFILE

It was vital to allow the members to easily edit their details and change their password since the initial password was assigned by the admin. To this end, I created several new pages – view_member_profile, edit_member_profile, and change_password. I based my solution for changing a user password on code written by Code Kad [10], which prompts the user to input their old password and to input their new password twice to confirm. There is some built-in error checking to make sure that the old password is correct, and a snack bar that pops up to confirm that the password has been changed successfully (*Figure 4-12*).

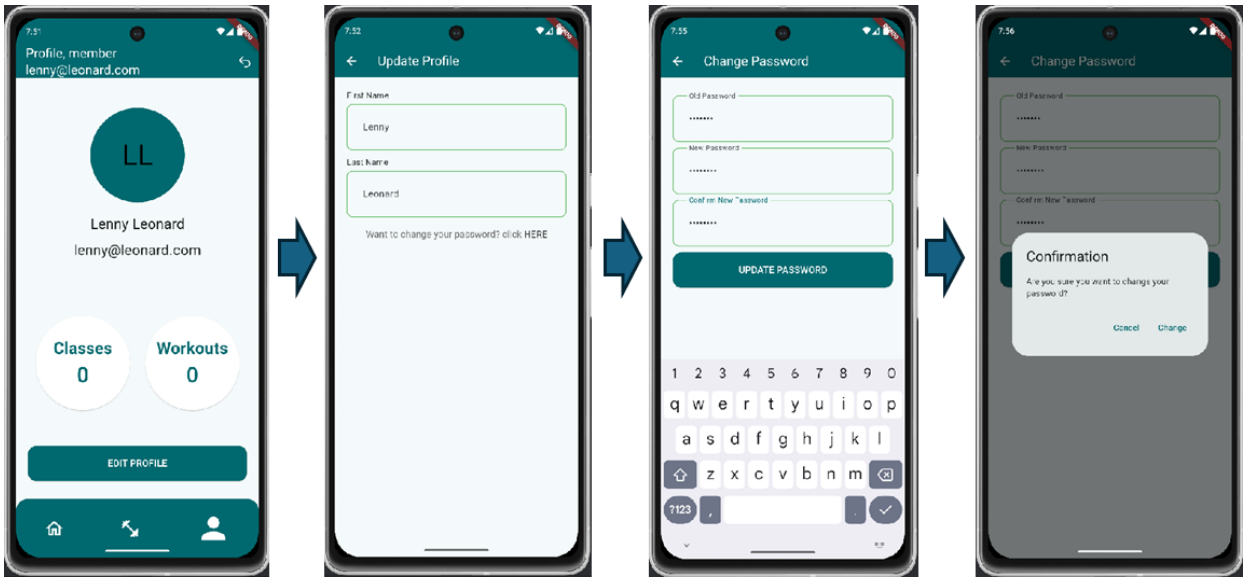


Figure 4-12: Changing password in GymGo app

4.4.3 SIGN IN TO CLASS

I made an alternate page to display when a member clicks on a gym class card, so that they do not see a list of sign-ins or have the option to edit the class as the admin would. I made a page called signin_class that either has a single button to allow the member to sign-in/sign-out of a class or a message that informs the member that class is full. When a member clicks the button to sign in, their userId is added to the “signins” array of the gym class document in firebase (*Figure 4-13*).

Figure 4-13: Member Ids are added to array in gym class firestore document

I made a function called fetchData() to retrieve the list of ‘signIns’ for the selected gym class (*Figure 4-14*).

Figure 4-14: Code to get list of “signins” for gym class

I then implemented some logic in the code to check if the currently signed-in user is already attending the class. I programmed some logic (*figure 4-15*) to handle 3 possible scenarios for the members:

1. The member is signed in to the class (show “Sign Out” button)
2. The class is not full, and the member is not signed in (show “Sign In button)
3. The class is full (Display “Class is full” message)

Figure 4-16 shows these screens as would be seen by a member.

```
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(...),  // AppBar
    body: signedIn.contains(uid)
        ? Padding(...)  // Padding
        : signedIn.length < size
            ? Padding(...)  // Padding
            : Padding(
                padding: const EdgeInsets.all(20.0),
                child: Center(
                  child: const Text('Class is full'),
                ),  // Center
              ),  // Padding
  );  // Scaffold
}
```

Check if current userId is in "signin" list

Check if gym class has spaces available

Else display 'Class is full' message

Figure 4-15: Logic to create widget on class_signin page for members

Member IS signed in to class

Member IS NOT sign in to class
&&
Class IS NOT full

Member IS NOT signed in to class
&&
Class IS full

Figure 4-16: Screens showing possible scenarios for member clicking on class card

4.4.4 NAVIGATION BAR

Upon signing in to the app, I had to run some code in the initState() function (*Figure 4-17*) of the home_page to check if the signed in user is an admin or a member.

```
@override
void initState() {
  super.initState();
  checkIfMember().then((updatedVal) {
    setState(() {
      _member = updatedVal;
    });
  });
  queryValues();
}
```

Figure 4-17: This code gets run when the home_page is loaded.

```
Future<bool> checkIfMember() async {
  QuerySnapshot snap = await FirebaseFirestore.instance
      .collection('members')
      .where("userId", isEqualTo: userId)
      .get();

  if (snap.docs.isNotEmpty) {
    return true;
  } else {
    return false;
  }
}
```

Figure 4-18: This function queries firebase to check if current user is a member

I used the ternary operator to check if the _member variable is true or false (*Figure 4-18*). This allowed me to give different features for the app depending on the status of the signed in user.

For example, a member will have different options on the navigation bar (*Figure 4-19*). In the home_page file different icons will display, resulting in different pages being linked to them.

```
bottomNavigationBar: _member
    ? Container(...)  // Container
    : _coach
        ? Container(...)  // Container
        : Container(...),  // Container
```

Figure 4-19: home_page bottomNavigationBar uses ternary operator to set icons

I created a different array of pages for the admin and members to link to the navigation bar icons (*Figure 4-20*).

```
final List<Widget> pages = [
  const ClassList(
    member: false,
    coach: false,
  ),
  const MemberCoachList(
    coach: false,
  ),
  ViewAdminProfile(
    docId: FirebaseAuth.instance.currentUser!.uid,
  ),
];
```

```
final List<Widget> memberPages = [
  ClassList(
    member: true,
    coach: false,
  ),
  const WorkoutList(),
  ViewMemberProfile(
    docId: FirebaseAuth.instance.currentUser!.uid,
    coach: false,
    member: true,
  ),
];
```

Figure 4-20: List of pages linked to admin profile and member profile along with their nav bar icons below

4.5 SPRINT 5

3rd March – 16th March

The 3rd iteration of my app implemented another new user type (coach), and gave them the following abilities:

- View only gym classes they are coaching
- View list of members who are attending
- View list of gym members

4.5.1 ADD NEW COACH

I made a new collection in the database called ‘coaches’. The admin could then add a coach to the gym by creating a new member as before, with the additional step of checking a box to assign coaching privileges to the user (*Figure 4-21*). The code to implement this logic is outlined in *Figure 4-22*.

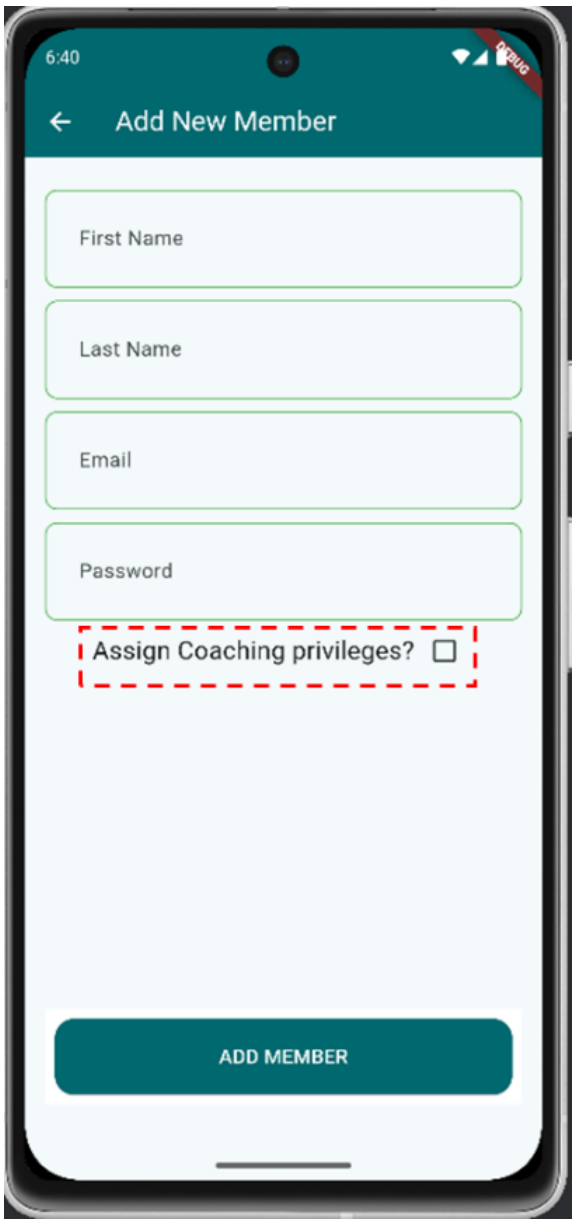


Figure 4-21: Admin adds coach to firestore by checking the box when adding a new member

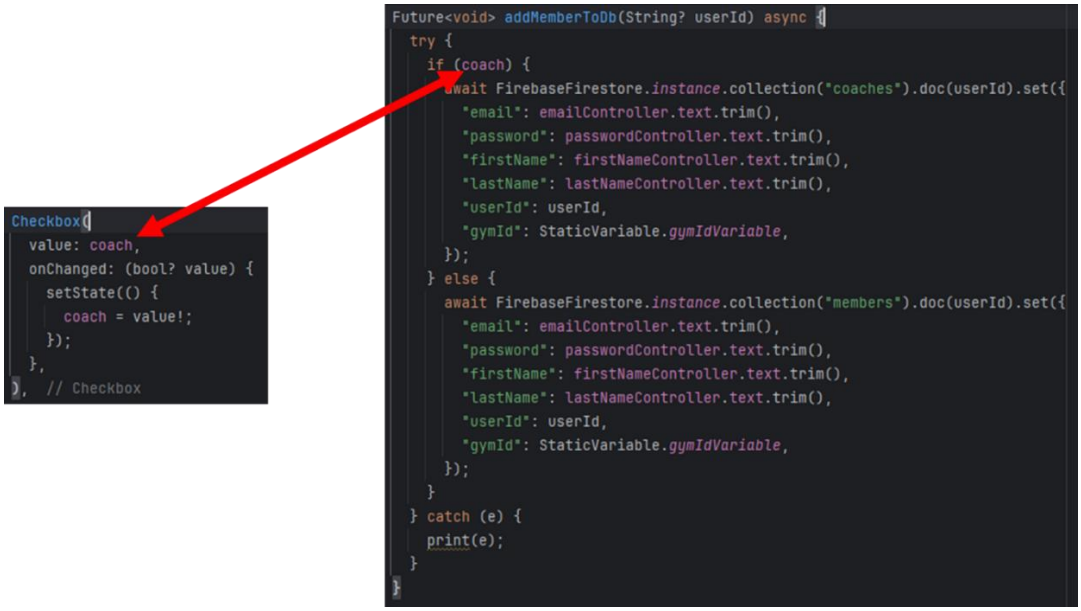


Figure 4-22: The checkbox in the add_new_member page sets the 'coach' variable to true or false. This is then used to determine which collection the new user belongs to

4.5.2 UPDATE HOME PAGE TO HANDLE NEW USER TYPE

I extended the homepage logic to not only check if the logged in user is a member (i.e. in the members collection on firestore), but to also check if they are a coach (i.e. in the 'coaches' collection on firestore).

I added a new Boolean parameter to the ClassList constructor called 'coach' - if this value is true (*Figure 4-23*), only the gym classes assigned to the logged in coach will be displayed (*Figure 4-24*).

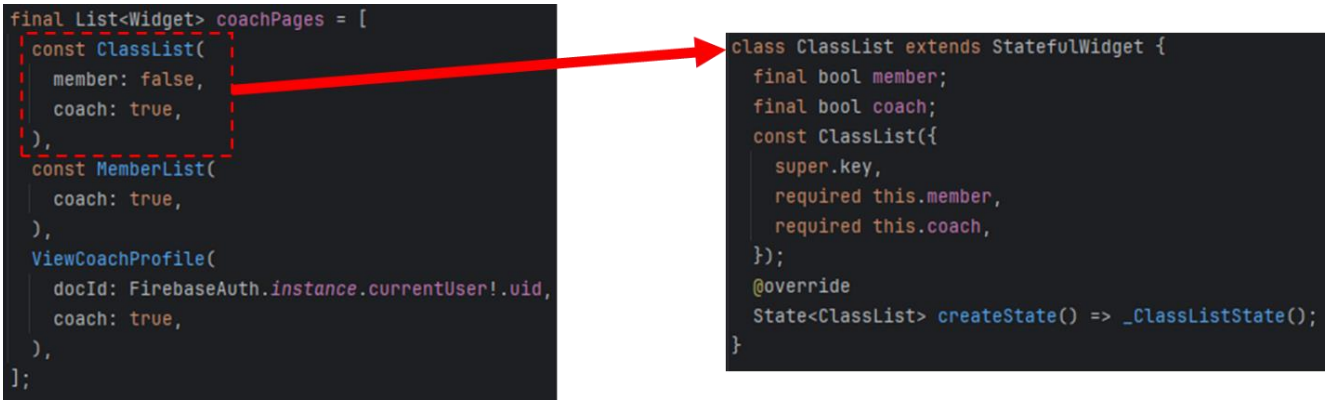


Figure 4-23: I passed data through the constructor of the ClassList widget to change its appearance.

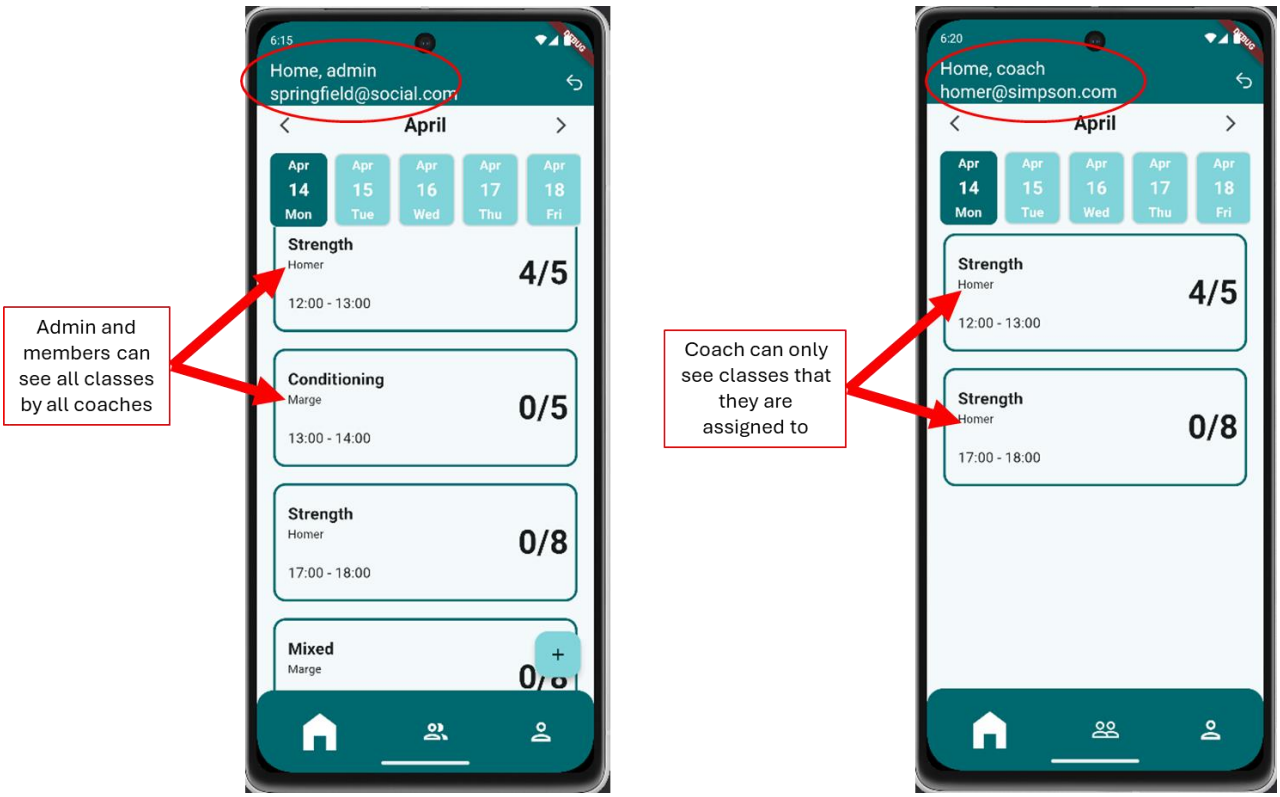
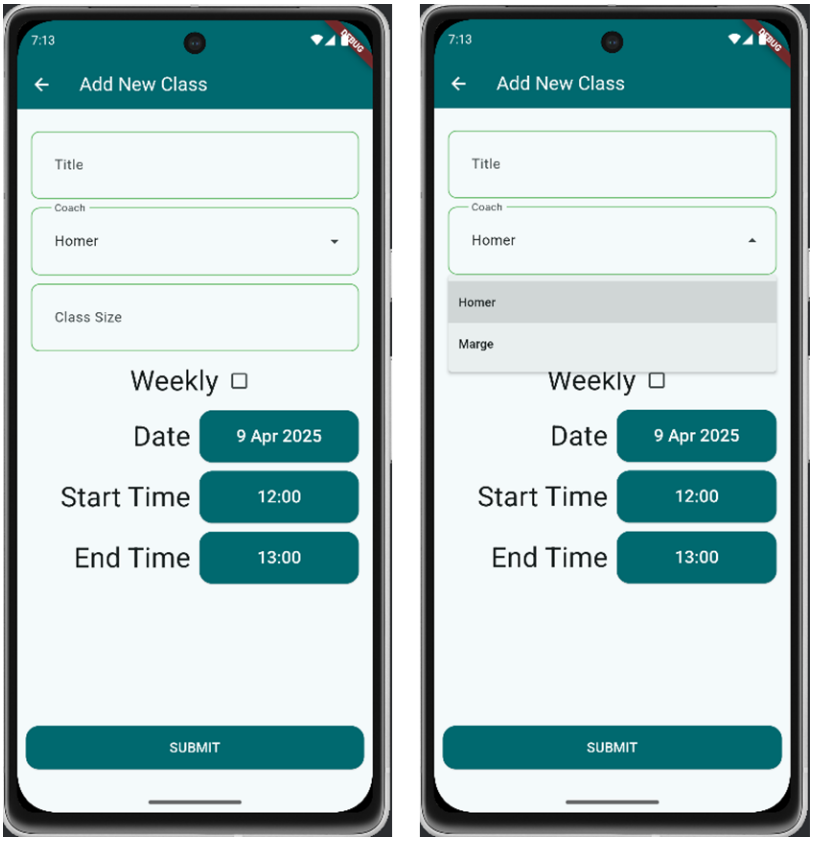


Figure 4-24: Admin and members can see all classes, coaches can only see their assigned classes

4.5.3 ADD DROPDOWN MENU TO ASSIGN COACH TO WORKOUT

To make sure a coach can only see their own classes, I had to ensure that there were no typos when the admin created the class. For example, if the coach is named Claire, but the admin assigns the class to ‘Clare’, my firestore query to find all of Claire’s classes would not return this class. Instead of having a text field for assigning the coach to the class, I used a dropdown menu [11] widget (*Figure 4-25*).




```dart
DropdownMenu<String>(
  label: Text('Coach'),
  expandedInsets: EdgeInsets.zero,
  initialSelection: coachNameList.first,
  onSelected: (String? value) {
    setState(
      () {
        selectedValue = value!;
      },
    );
  },
  dropdownMenuEntries: coachNameList.map(
    (String value) {
      return DropdownMenuEntry<String>(
        value: value,
        label: value,
      );  // DropdownMenuEntry
    },
  ).toList(),
),  // DropdownMenu
```

Figure 4-25: Dropdown widget for assigning coach to class and code used to create it

When the add_new_class page is launched, a query is sent to the database to get all documents from the ‘coaches’ collection. The dropdown menu widget is then populated with the ‘name’ field from these documents (*Figure 4-26*).

```dart
// Function to fetch Firestore values for dropdown menu
Future<void> fetchDropdownValues() async {
  List<String> values = await getFieldValues("coaches", "firstName");
  setState(() {
    coachList = values;
    if (coachList.isNotEmpty) {
      selectedValue = coachList.first; // Set default selected value
    }
  });
}
```

```dart
// Function to get field values from Firestore
Future<List<String>> getFieldValues(
    String collectionName, String fieldName) async {
  List<String> fieldValues = [];
  try {
    QuerySnapshot querySnapshot = await FirebaseFirestore.instance
        .collection(collectionName)
        .where('gymId', isEqualTo: StaticVariable.gymIdVariable)
        .get();
    for (var doc in querySnapshot.docs) {
      if (doc.data() is Map<String, dynamic> &&
          (doc.data() as Map<String, dynamic>).containsKey(fieldName)) {
        fieldValues.add(doc[fieldName].toString());
      }
    }
  } catch (e) {
    print("Error fetching field values: $e");
  }
  return fieldValues;
}
```

Figure 4-26: Code to fetch data to populate the coach selector drop down menu

4.6 SPRINT 6

17th March – 30th March

For the 4th iteration of GymGo, I incorporated a feature for members to log workouts.

4.6.1 WORKOUT LIST

A new page was added to display the list of workouts. I made this new page to be accessible via the navigation bar for logged in members (*Figure 4-27*)

Figure 4-27: Navigation bar for members with dumbbell icon to view workout log

4.6.2 ADD NEW WORKOUT

I made a new collection in the firestore database called ‘workouts’ (*Figure 4-28*)

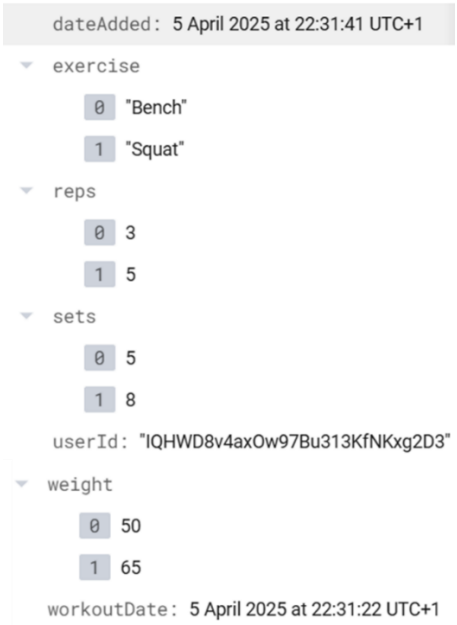


Figure 4-28: Example of a ‘workout’ document in firestore database

As can be seen in the previous figure, arrays were used to store the exercise, sets, reps, and weight values. In a new page call add_new_workout (*Figure 4-29*)

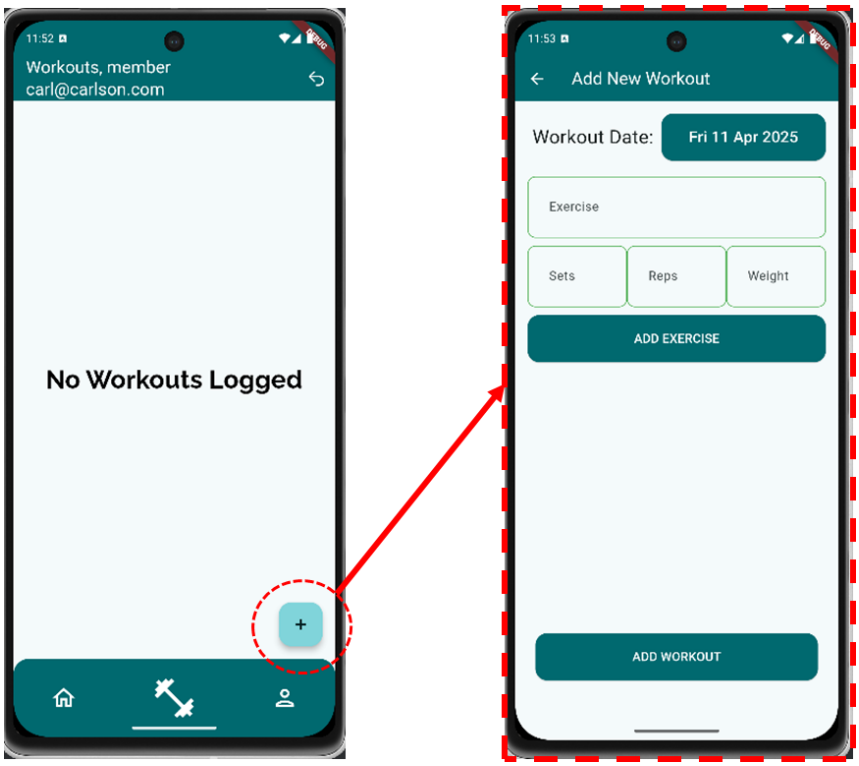


Figure 4-29: Member can navigate to add_new_workout page using FAB on workout log

The add_new_workout page consists of a date picker, 4 text fields, and 2 buttons. The first button is labelled ADD EXERCISE. When the user enters data into the text fields and clicks the ADD EXERCISE button the addExercise() function is called, which stores these values list variables (*Figure 4-30*)

```
List<String> exercises = [];
List<int> sets = [];
List<int> reps = [];
List<double> weight = [];
```

```
Future<void> addExercise(String ex, int s, int r, double w) async {
  try {
    setState(() {
      exercises = [...exercises, ex];
      sets = [...sets, s];
      reps = [...reps, r];
      weight = [...weight, w];
    });
  } on FirebaseAuthException catch (e) {
    print(e.message);
  }
}
```

Figure 4-30: List variable declared and addExercise() function to update their state

As the user adds exercises, they will appear in a list below the button. The individual exercises can be swiped left to be deleted (with confirmation) if a mistake is made (*Figure 4-31*).

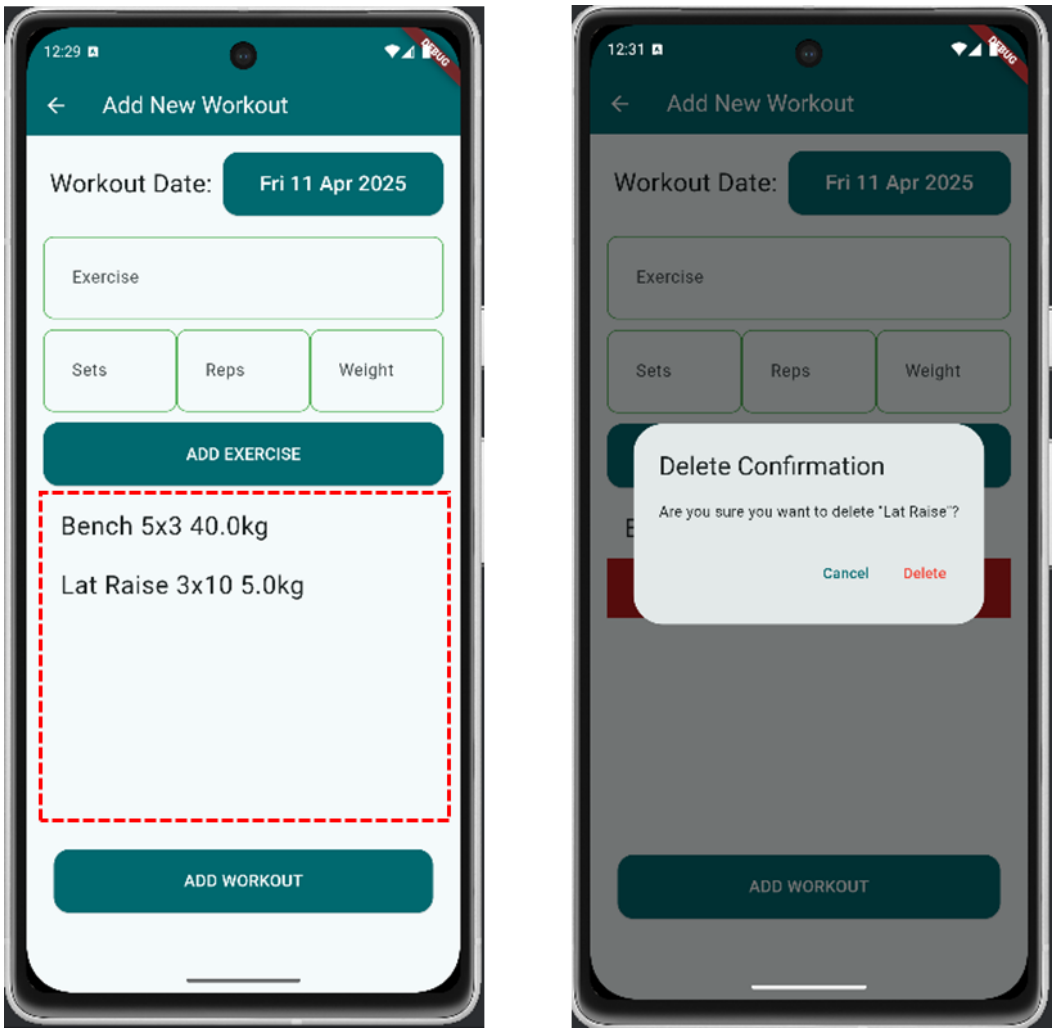


Figure 4-31: Exercises being added to list and exercise being deleted from list

To implement the swipe-to-delete feature I used Flutter's *Dismissible* widget. Within that widget I added an *AlertDialog* widget to prompt the user to confirm the deletion (*Figure 4-32*).

Figure 4-32: Code used to swipe items from a list

123

4.6.3 VIEW AND EDIT WORKOUT

Once a workout had been added it appeared in the workout log of the user. The workouts are listed by date, starting with the most recent addition. The workout cards can be swiped to be deleted (with confirmation) or opened by clicking. Once opened the individual exercises cards are displayed and can be edited or deleted by swiping. The user can also

add extra exercises to the workout (*Figure 4-33*).

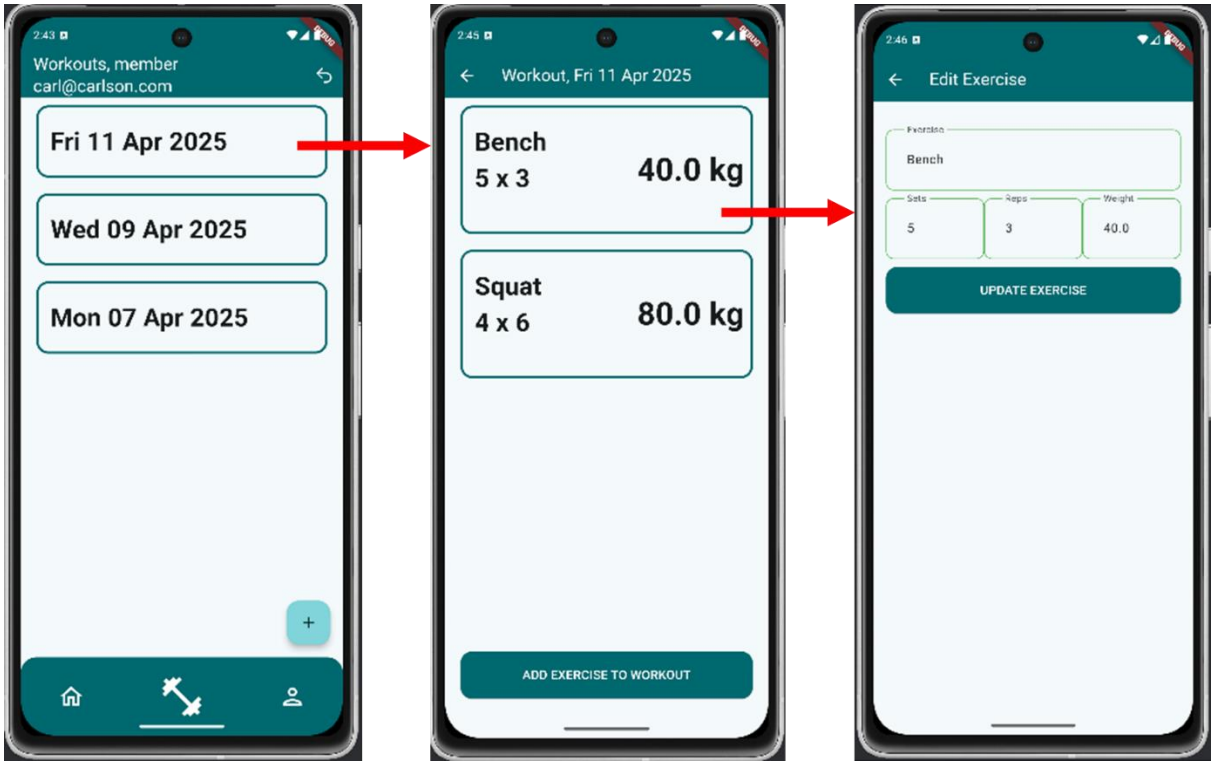


Figure 4-33: Workout list of workout_card widgets, exercise list of exercise_card widgets, and exercise edit screens

4.6.4 VIEW MEMBER WORKOUTS

The admin and coaches were given the option of viewing a list of the members and viewing details about an individual member. They could open a member profile from the list and click on the circular ‘Workouts’ button the view all workouts entered by that member (*Figure 4-34*), but cannot delete or edit the data.

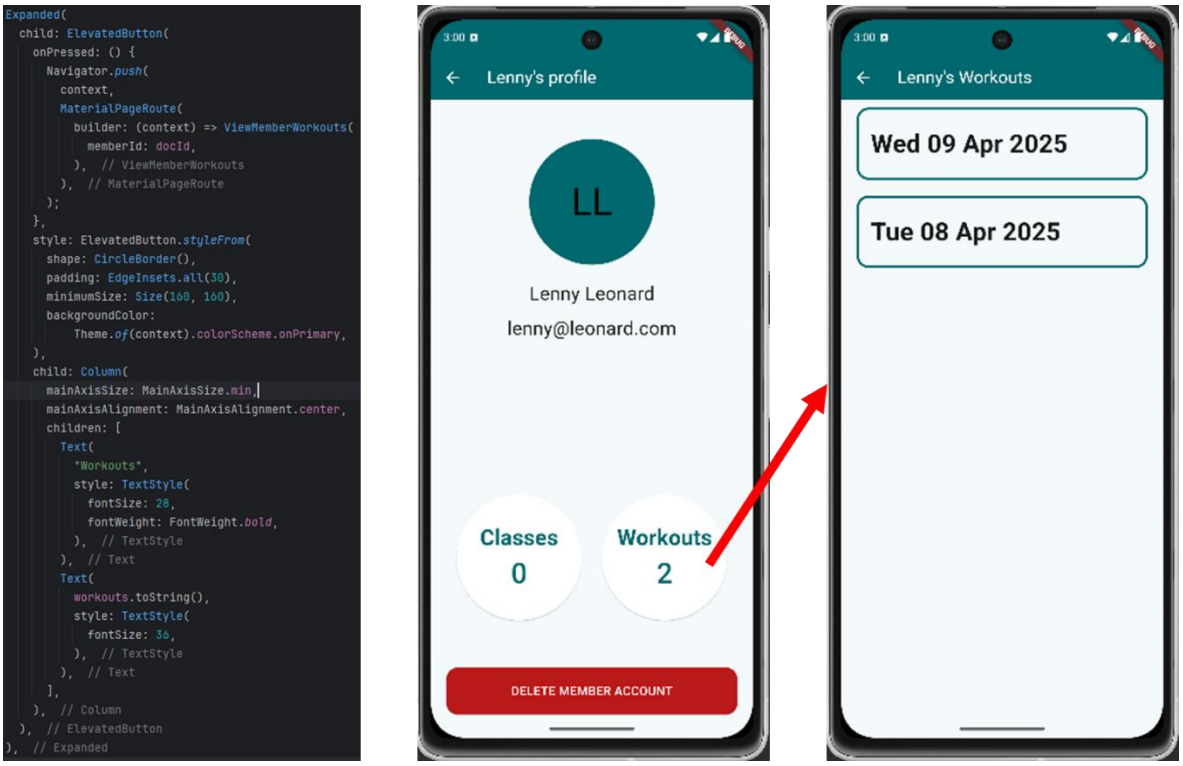


Figure 4-34: Code for circular buttons on member profile and screenshots of functionality

4.7 SPRINT 7

31st March – 13th April

The 5th version of the app involved finding a way to add multiple different gyms to the database, and for the gym admin to edit the gym profile and repeat weekly classes. I also built functionality to take attendance in gym classes.

4.7.1 ADD MULTIPLE GYMS TO DATABASE

I added a new collection to the firestore database called ‘gyms’ (*Figure 4-35*).

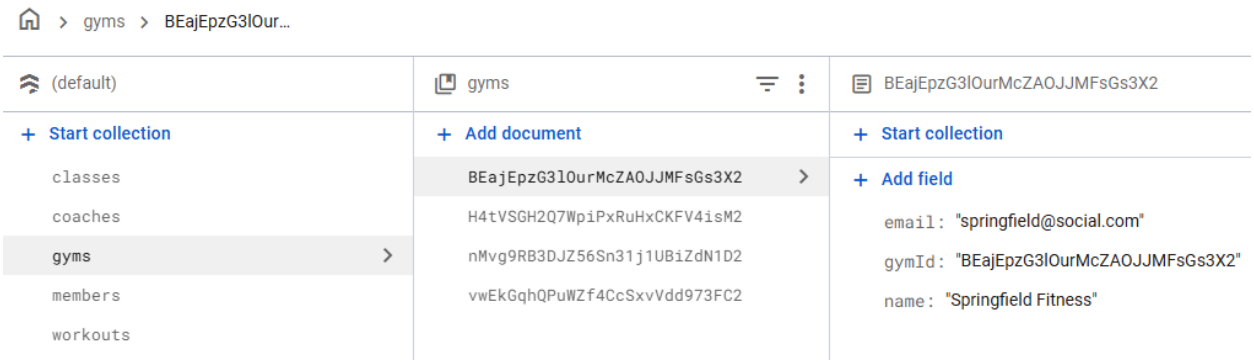


Figure 4-35: Gyms collection in firestore database

When a user registers on the sign_up page, they give their gym a name, and it is added to the database. I connected some code to the dropdown menu to the login_page so that a user can log in and only see data related to the gym they are a part of (*Figure 4-36*).

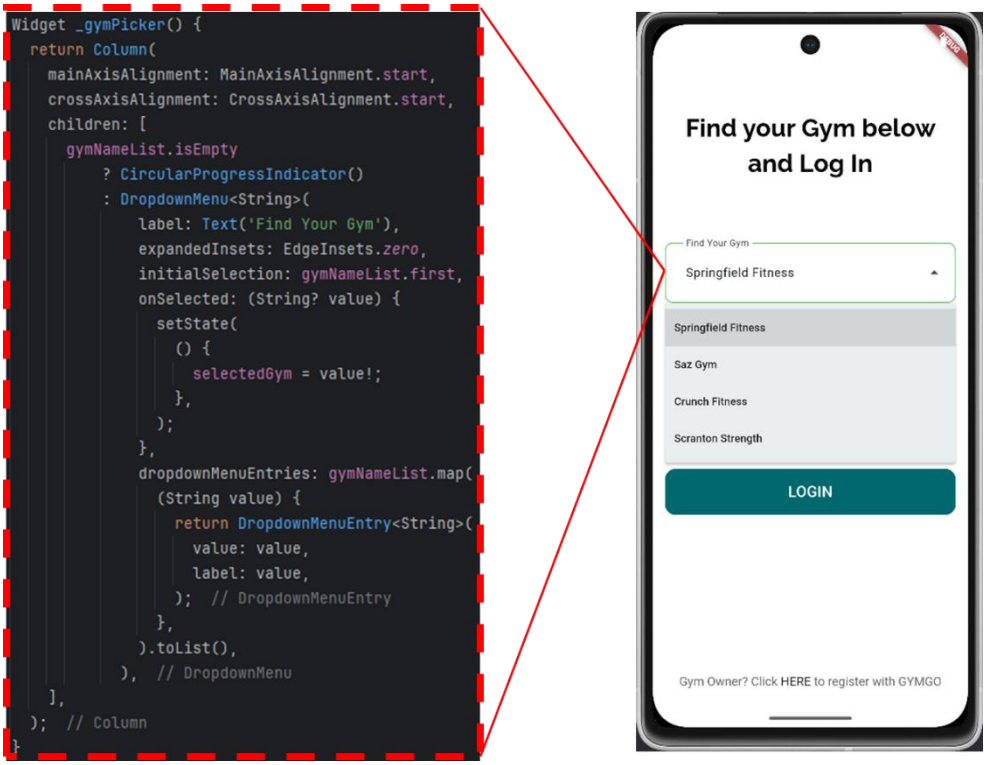


Figure 4-36: Gym Picker dropdown widget on Login Page

I added a new field to all user collections in the database (admin, coach, member) called gymId. When a gym owner adds a new gym to the app, their gym is assigned an id that is the same as their user uid. When this gym owner adds members or coaches, they will be linked by this id. I coded some logic to check if the user trying to sign in

belongs to the selected gym (*Figure 4-37*).

```dart
Future<void> signin(
    {required String email,
    required String password,
    required BuildContext context,
    required String gymName}) async {
  try {
    var userInGym = await checkIfUserIsGymMember(email, gymName);

    if (userInGym) {
      StaticVariable.gymIdVariable = await getGymId(gymName);

      await FirebaseAuth.instance
          .signInWithEmailAndPassword(email: email, password: password);
```

```dart
Future<String?> getGymId(String gymName) async {
  try {
    QuerySnapshot querySnapshot = await FirebaseFirestore.instance
        .collection('gyms')
        .where('name', isEqualTo: gymName)
        .limit(1)
        .get();

    if (querySnapshot.docs.isNotEmpty) {
      var doc = querySnapshot.docs.first;
      return doc['gymId'];
    } else {
      return null;
    }
  } catch (e) {
    print("Error: $e");
    return null;
  }
}
```

```dart
Future<bool> checkIfUserIsGymMember(String email, String gym) async {
  try {
    var gymId = await getGymId(gym);
    QuerySnapshot membersQuery = await FirebaseFirestore.instance
        .collection('members')
        .where('email', isEqualTo: email)
        .where('gymId', isEqualTo: gymId)
        .limit(1)
        .get();

    QuerySnapshot coachesQuery = await FirebaseFirestore.instance
        .collection('coaches')
        .where('email', isEqualTo: email)
        .where('gymId', isEqualTo: gymId)
        .limit(1)
        .get();

    QuerySnapshot gymsQuery = await FirebaseFirestore.instance
        .collection('gyms')
        .where('email', isEqualTo: email)
        .where('gymId', isEqualTo: gymId)
        .limit(1)
        .get();

    return membersQuery.docs.isNotEmpty ||
        coachesQuery.docs.isNotEmpty ||
        gymsQuery.docs.isNotEmpty;
  } catch (e) {
    Fluttertoast.showToast(
      msg: "Selected gym does not have user with this email",
      toastLength: Toast.LENGTH_LONG,
      gravity: ToastGravity.SNACKBAR,
      backgroundColor: Colors.black54,
      textColor: Colors.white,
      fontSize: 14.0,
    );
    return false;
  }
}
```

Figure 4-37: Code to check if user is in the selected gym

Instead of passing the gymId through every single page of the app, I decided to implement a static variable that is accessible by all pages. In the utils folder a made a new file called static_variable with a single class (*Figure 4-38*).

static_variable.dart

```dart
class StaticVariable {
  static String? gymIdVariable;
}
```

Figure 4-38: Variable accessible by all pages

When a user successfully logs in, the variable ‘gymIdVariable’ gets updated to the value of ‘gymId’ in the user’s database document. To make use of this new variable, I added a new part of all queries to the firestore database (*Figure 4-39*).

```
FirebaseFirestore.instance
  .collection("classes")
  .where('gymId', isEqualTo: StaticVariable.gymIdVariable)
  .where('startTime', isGreaterThanOrEqualTo: selectedDate)
  .where(
    'startTime',
    isLessThan: DateTime(selectedDate.year,
        selectedDate.month, selectedDate.day + 1),  // DateTime
  )
  .where('coachId',
      isEqualTo:
          FirebaseAuth.instance.currentUser!.uid.toString())
  .snapshots()
```

Figure 4-39: Code snipped highlighting additional query to gymId

4.7.3 REPEAT CLASSES MARKED AS WEEKLY

When the admin added a new gym class, they had the option of marking them as ‘weekly’. I added a button to the admin_profile page to trigger a function called refreshClasses() (*Figure 4-40*).

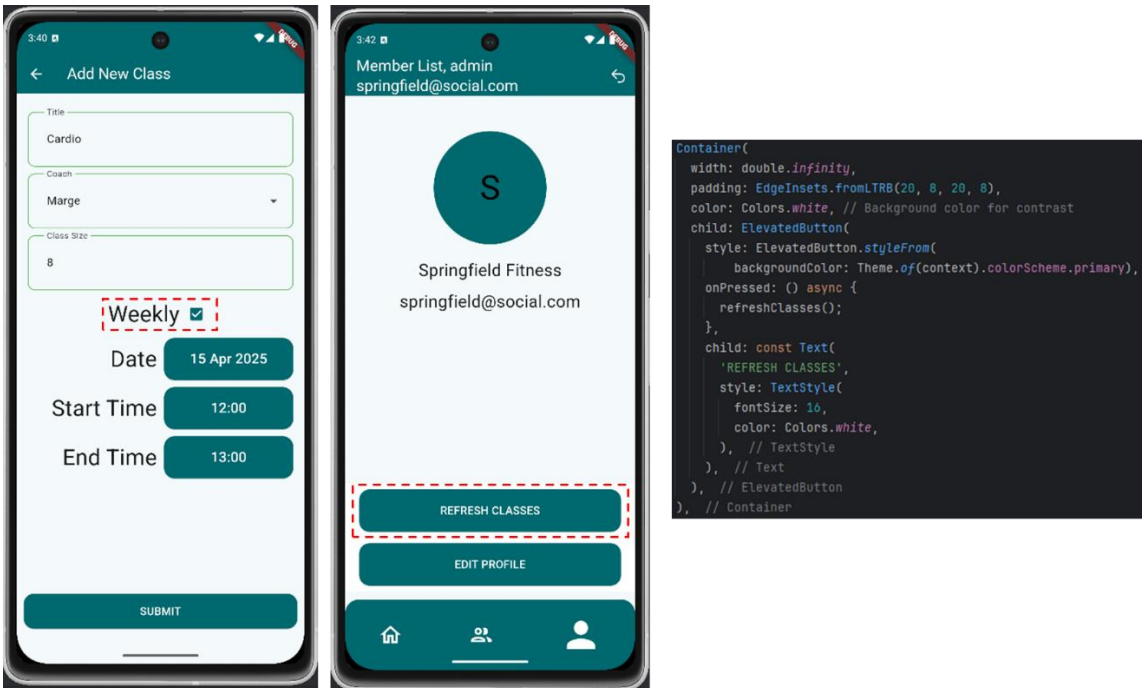


Figure 4-40: Marking classes as ‘weekly’ and the REFRESH CLASSES button on the admin_profile page to trigger the refreshClasses() function

This function first searched for any classes in the past week that are marked as weekly (*Figure 4-41*)

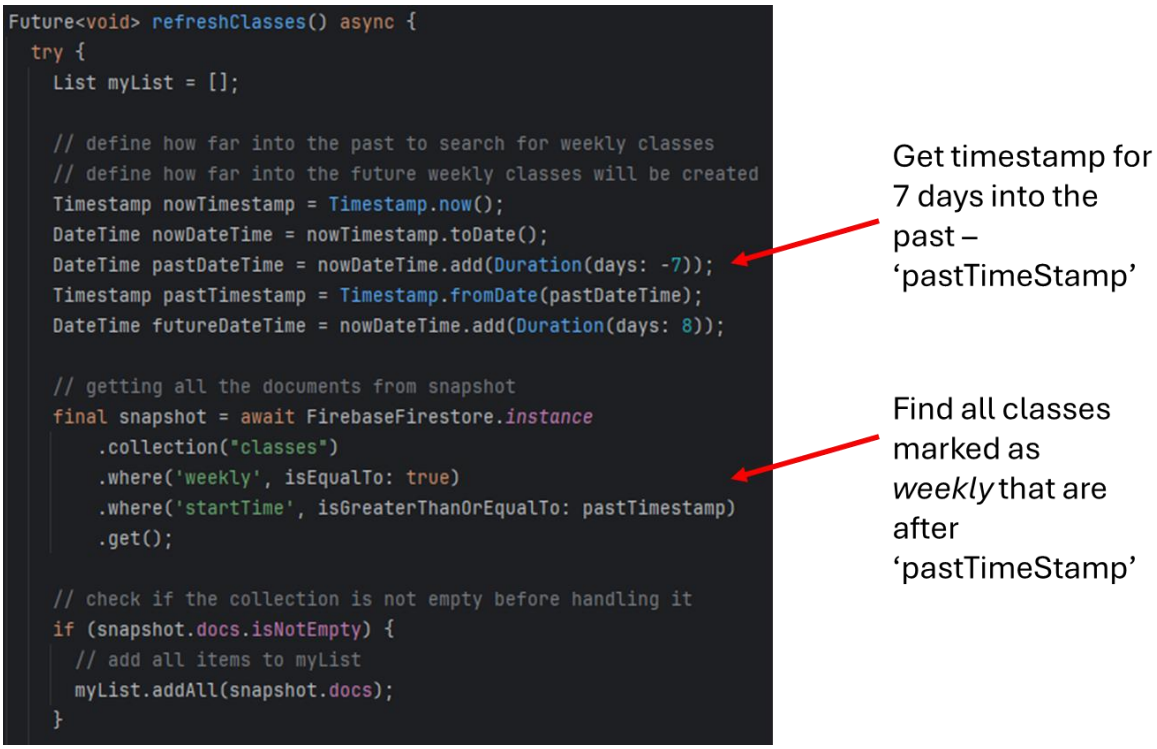


Figure 4-41: First part of refreshClasses() function to find previous gym classes marked as weekly

The rest of the function looped through these weekly classes and added created new gym classes for the week ahead (*Figure 4-42*).

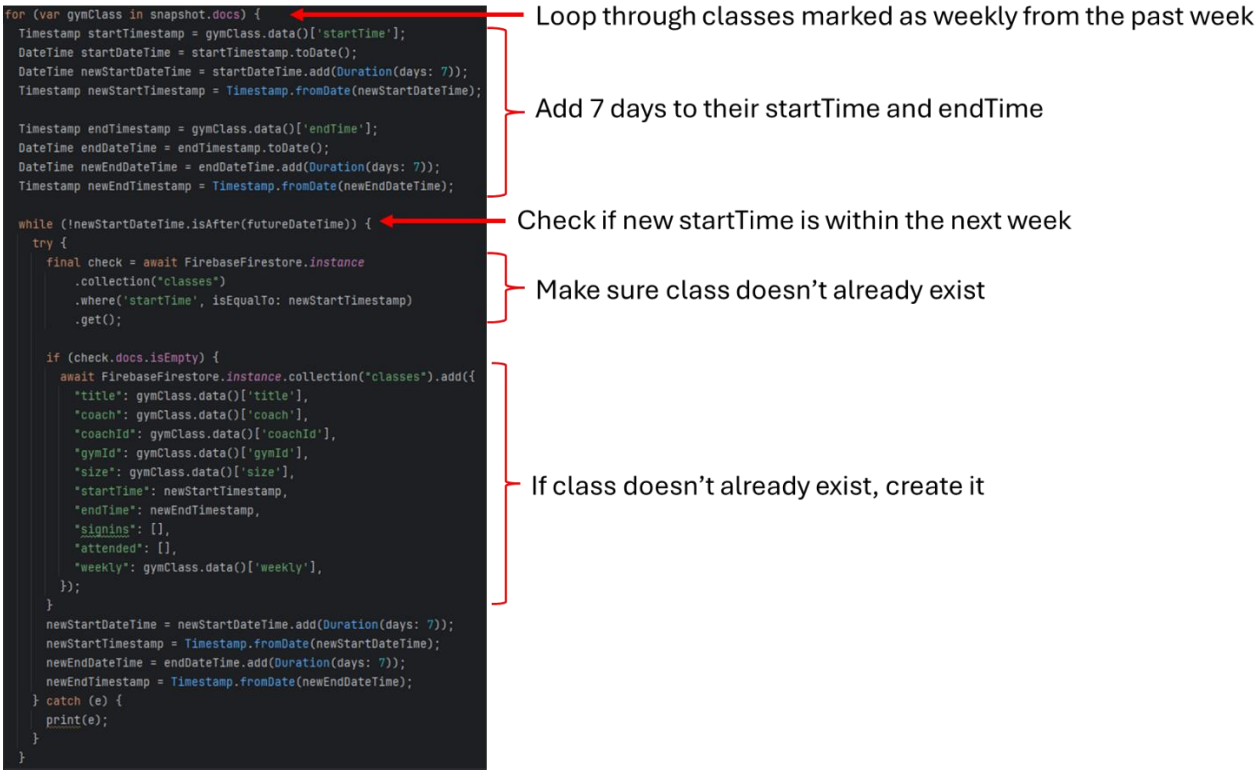


Figure 4-42: Second part of refreshClasses() function to check if future classes already exist, and create them

4.7.4 TAKE CLASS ATTENDANCE

To keep track of members who don't show up to classes I added a feature that allowed coaches to take attendance for the gym classes. I added a checkbox to the sign_in_card widget (*Figure 4-43*). When the checkbox is checked the addMemberToAttendedDb() function adds the member's id to the 'attended' array in the class document, while unchecking the checkbox calls the removeMemberFromAttendedDb() array (*Figure 4-44*).

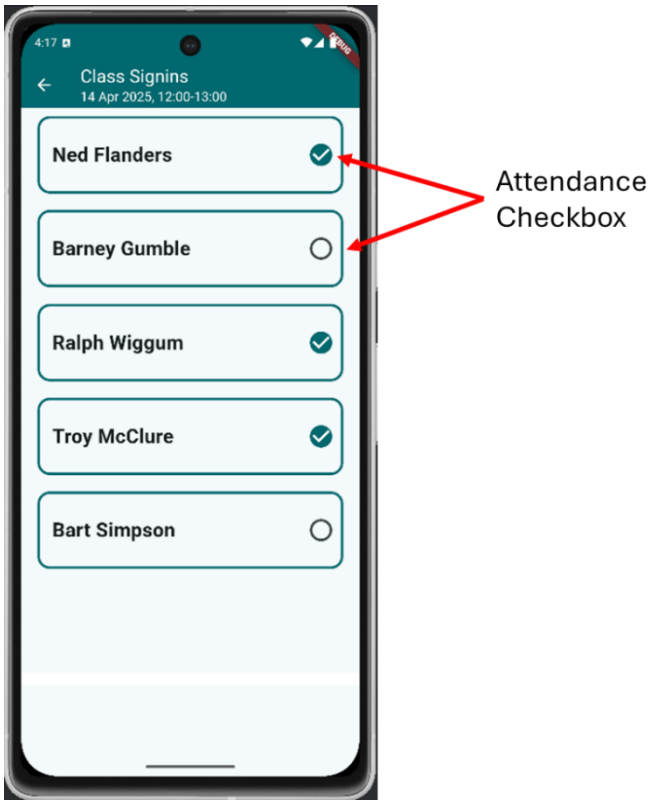


Figure 4-43: Class sign-in screen for admin and coaches showing members who have signed in to the class and checkbox to mark attendance

```dart
Transform.scale(
  scale: 1.5,
  child: Checkbox(
    value: isChecked, // widget.attended,
    onChanged: (bool? value) {
      setState(() {
        isChecked = value!;
      });
      if (value != null) {
        value
            ? addMemberToAttendedDb(widget.memberId)
            : removeMemberFromAttendedDb(widget.memberId);
      }
    },
    shape: CircleBorder(),
    checkColor: Colors.white,
    activeColor: Theme.of(context).colorScheme.primary,
  ),  // Checkbox
)  // Transform.scale
```

```dart
Future<void> addMemberToAttendedDb(String memberId) async {
  try {
    FirebaseFirestore.instance
        .collection("classes")
        .doc(widget.docId)
        .update({
      'attended': FieldValue.arrayUnion([memberId])
    });
  } catch (e) {
    print(e);
  }
}

Future<void> removeMemberFromAttendedDb(String memberId) async {
  try {
    FirebaseFirestore.instance
        .collection("classes")
        .doc(widget.docId)
        .update({
      'attended': FieldValue.arrayRemove([memberId])
    });
  } catch (e) {
    print(e);
  }
}
```

Figure 4-44: Code on sign_in_card widget for attendance checkbox and linked functions

5 CHALLENGES

5.1 LEARNING FLUTTER/DART

It took a while before I was comfortable developing using the flutter framework. I had some experience using React and Kotlin to develop apps, so some of the general principles I learned could be applied to this project. As I learned more about flutter, I realised that the structure of my app was not typical, but it worked to my satisfaction, so I didn't feel the need to try to refactor everything and risk wasting time.

5.2 CHECK USER TYPE

I found it challenging to create a way for the app to determine if the current user was a gym admin, gym coach, or gym member. I found a library called 'firebase_admin' [13] that I wanted to use, but I decided to take a different approach that was more flexible. Since the register page was only for creating gym admin accounts, and member/coach accounts could only be created by the gym admin I could tie the firebase auth account creation to the firestore database:

- When a new gym (admin) account is created, the details are stored in the 'gyms' collection in the database.

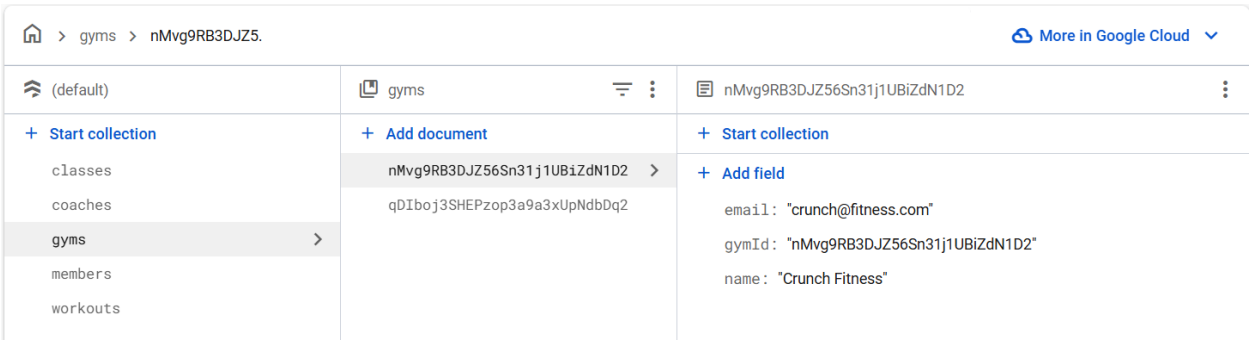


Figure X: When a new gym (admin) account is created, details are saved in database

- When a new coach account is created, the details are stored in the ‘coaches’ collection in the database.

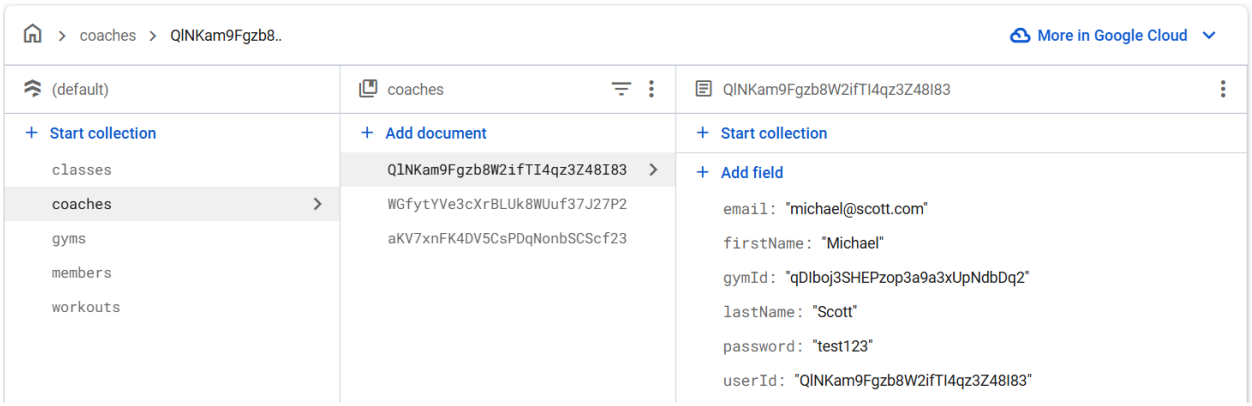
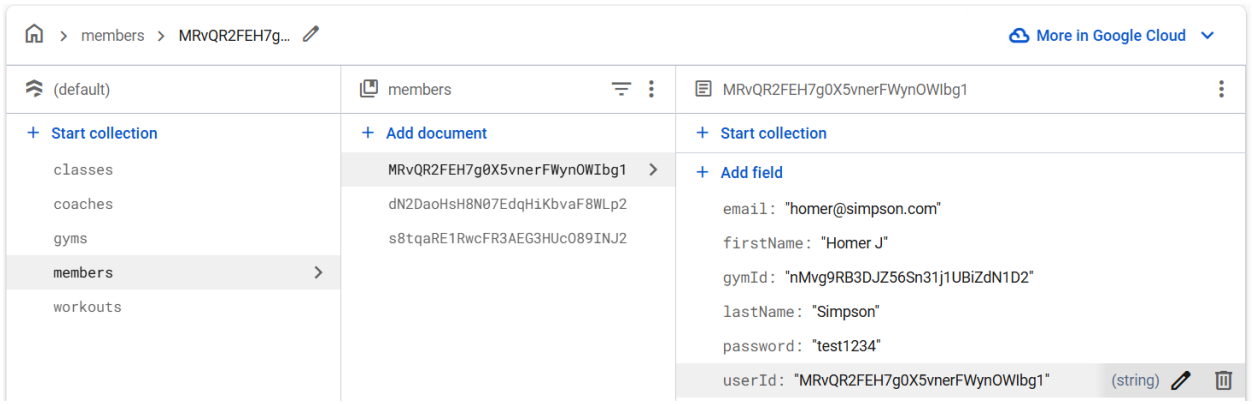


Figure X: When a new coach account is created, details are saved in database

- When a new member account is created, the details are stored in the ‘members’ collection in the database.



-
  - Figure X: When a new member account is created, details are saved in the database

In each case, the id of the newly created user was applied to the database document, so that I could get the currently signed in user (FirebaseAuth.instance.currentUser!.uid) and query the database collections to determine if the user is an admin, coach, or member.

5.3 REPEATING GYM CLASSES

I had some difficulty implementing a feature so that classes marked as ‘weekly’ are dynamically created to save space in the database. I thought this was important to figure

out to make sure that the gym admin doesn't have to waste time creating every class individually. I also didn't want to fill up the database by repeating class every week into the future.

Firebase does have a feature called 'scheduler' [14] that uses 'Cloud Scheduler' [15] to run a cron job. This would have allowed me to pick a time (e.g. every Sunday at midnight) for the database to search for all gym classes marked as 'weekly' and create new classes for the week ahead. I didn't manage to get this feature to work, so I had to find another way.

I decided to add a button to gym admin's profile page titled 'Refresh Classes'. Clicking this button searched the database 'classes' collection for any gym classes where 'weekly' is true and created new classes up to a defined point in the future (I chose to add classes 7 days into the future). To help me solve this problem I used ChatGPT -

Prompt: "add 7 days to firestore timestamp flutter"

Response:

```
import 'package:cloud_firestore/cloud_firestore.dart';

void main() {
  // Example Firestore Timestamp
  Timestamp originalTimestamp = Timestamp.now();

  // Convert Timestamp to DateTime
  DateTime dateTime = originalTimestamp.toDate();

  // Add 7 days
  DateTime newDateTime = dateTime.add(Duration(days: 7));

  // Convert back to Firestore Timestamp
  Timestamp newTimestamp = Timestamp.fromDate(newDateTime);

  print("Original Timestamp: $originalTimestamp");
  print("New Timestamp (after 7 days): $newTimestamp");
}
```

This code helped me to create the logic in the RefreshClasses() function within the view_admin_profile dart file.

6 REFLECTION

6.1 WHAT I ACHIEVED

I built a cross-platform mobile app using a framework and language that I had never used before. The app allows gym owners to keep track of the classes and members using an intuitive UI and is backed up in the cloud with Firestore. It also provides members with a basic workout log to keep track of their activities. I completed all of the goals outlined in my design, although I had to tweak the weekly classes feature to be user driven rather than automated.

6.2 WHAT I LEARNED

I learned that mobile app development is much more difficult than I would have thought before starting this course, and initially I was very overwhelmed. As I spent more time using Flutter I slowly got to a stage where I could create something as it came to mind. While my solutions might not have been the easiest or most efficient method of solving the problems at hand, they did give me starting points that I developed on as the project progressed. I now know many of the wrong ways to do things yet still have them work, which is the only way I truly learn

6.3 FUTURE DEVELOPMENT

I would like to start this project all over again to organise the code structure in a more traditional format and then continue to fine tune existing features while adding new ones. Automating the weekly class creation is first on my list. I would like to add a feature that connects to concept2 ergs (rowing machines etc.) via Bluetooth and include alerts that get sent to a user’s phone to remind them of an upcoming class

7 BIBLIOGRAPHY

[1] The Six Most Popular Cross-Platform App Development Frameworks (2025) https://www.jetbrains.com/help/kotlin-multiplatform-dev/cross-platform-frameworks.html (Accessed 22 January 2025)

[2] Bartosińska, Piekielny (2023) 4 Most Popular Cross-Platform App Development Frameworks for 2024. Available at https://www.thedroidsonroids.com/blog/top-cross-platform-app-development-frameworks (Accessed 22 January 2025)

[3] React Native vs Flutter: What to Choose in 2025 (2025) https://www.browserstack.com/guide/flutter-vs-react-native#:~:text=Flutter%2C%20which%20uses%20Dart%20programming,due%20to%20its%20familiar%20language. (Accessed 22 January 2025)

[4] Building a Backend for Your Flutter App: Firebase vs. Node.js (2024) https://www.appzoc.com/building-a-backend-for-your-flutter-app-firebase-vs-node-js/ (Accessed 24 January 2025)

[5] Flutter Firebase Tutorial For Beginners | FlutterFire Course | Firebase Auth, Firestore DB, Storage (2025) https://www.youtube.com/watch?v=LFlE8yV7lJY

[6] Rivaan Ranawat, flutter-firebase-tutorial, (2024), GitHub repository, https://github.com/RivaanRanawat/flutter-firebase-tutorial/tree/main

[7] Add Firebase to your Flutter app (2025) https://firebase.google.com/docs/flutter/setup?platform=android

[8] showDatePicker function, (accessed March 2025) https://api.flutter.dev/flutter/material/showDatePicker.html

[9] showTimePicker function, (accessed March 2025) https://api.flutter.dev/flutter/material/showTimePicker.html

[10] Code Kad, How to implement Change password with confirm password on Flutter / Firebase (Accessed Feb 27 2025) https://medium.com/@kodekadtech/how-to-

implement-change-password-with-confirm-password-on-flutter-firebase-8f6cd4df2220

[11] DropdownButton<T> class (Accessed 12 March 2025), https://api.flutter.dev/flutter/material/DropdownButton-class.html

[12] https://pub.dev/packages/dropdown_search (Accessed 15 March 2025)

[13] https://pub.dev/packages/firebase_admin (Accessed 23 February 2025)

[14] Schedule functions, (accessed 16 March 2025) https://firebase.google.com/docs/functions/schedule-functions?gen=2nd

[15] Schedule and run a cron job using the Google Cloud console, (Accessed 16 March 2025) https://cloud.google.com/scheduler/docs/schedule-run-cron-job