

BurnBoss – giving freedom to users in a workout app.

By Charles Fellows [017162], Queen Elizabeth's Grammar School [23205].

Submitted 2024.

Contents

0. Introduction (pg. 3)

1. Analysis (pg. 3 - 11)

1.1 Computational Amenability (pg. 3 – 4)

- *Creator*
- *Selector*
- *Calendar*
- *Stopwatch*
- *Player*

1.2 Identification of stakeholders (pg.4 – 5)

- *Those who want to stay in shape.*
- *Personal Trainers*

1.3 Research into existing solutions (pg. 5 – 8)

1.3.1 Pre-existing Solutions

- o *Workout Maker*
- o *Workout Builder*

1.4 The proposed solution (pg. 8 - 9)

1.4.1 Requirements for User's system

1.4.2 Implementation Requirements

1.4.3 Limitations of the proposed solution

1.5 Success Criteria (pg. 9 - 11)

2. Design (pg. 12 - 25)

2.1 Decomposition of methods (pg. 12-18)

2.2 Algorithms and validation (pg. 19-27)

2.2.1 Flowcharts

2.2.2 Pseudocode

2.3 Usability (pg. 27-28)

2.3.1 Learnability

2.3.2 Efficiency

2.3.3 Memorability

2.3.4 Errors

2.3.5 Satisfaction

2.4 Testing (pg. 29-30)

2.5 Interface Designs (pg. 31-33)

3. Development (pg. 34 - 64)

3.1 Prototypes (pg. 34)

- *test_app*
- *workout_app*

3.2 BurnBoss (pg. 36)

3.3 Theme management (pg. 39)

3.4 Routing (pg. 41)

3.5 Improvements (pg. 41)

3.6 Icon (pg. 41)

3.7 Functional app (pg. 41 – 63)

- *Authorisation*
- *Creating new workouts*
- *Groups*
- *Activities*
- *More Authentication*
- *Password Obscurity*
- *Stopwatch*
- *Selecting workouts*
- *Editing workouts*
- *User Tests*
- *Activity Types*
- *Playing workouts*
- *Activity Pages*

3.8 Bug fixes (pg. 63-64)

4. Testing (pg. 65 - 73)

4.1 Pre-Development Testing (pg. 65)

4.2 Post-Development Testing (pg. 65 – 73)

4.2.1 Testing for Usability

- o *Research Testing*
- o *Post-Use Testing*

4.2.2 Testing for Functionality

4.2.3 Testing for Robustness

5. Evaluation (pg. 73 - 82)

5.1 Success Criteria (pg. 73 – 78)

5.2 Justification of Usability Features (pg. 79 – 80)

5.3 Maintenance (pg. 80 - 81)

5.4 Limitations and Remedial actions (pg. 81 - 82)

5.5 Conclusion (pg. 82)

6. Appendix (pg. 83)

0. Introduction

Even in the modern day, the world with ‘an app for everything,’ we create a societally wide issue. The belief that one cannot live their own lives, without being backed by the hive-mind that is the online presence of society.

As much as each individual tries to pursue the perfect body, they are pushed by ‘influencers’ online to follow cult-like workout programs which just aren’t perfected to each user – my app aims to rectify this.

Once upon a time, people were able to create workouts that were personalized, and relieved them from the crushing chains of commercialized workouts. However, even if a person were to create their own workout, they would have to write it down, or follow other crude methods of saving it. This workout app solves this issue, and allows the user to save their workouts and take them anywhere, reducing the hassle. Research has shown that it takes 66 days to solidify a habit, and the hardest part is making it to this point – the ease of use of my app allows no hinderance to this process.

Later in this project I will discuss in detail the market for this application, which will allow me to tailor it to the needs of those who will actually be using it. This will include research into the current availability of exercise apps, but it is important to note that my app is set apart from the current market, due to its availability of freedom from the burden of commerciality, and its workouts that can do more harm than good.

When my app is developed, there will no longer be the worry that the workout you are following isn’t built for you – only the security of mind that you are free.

1. Analysis

1.1 Computational Amenability

As was mentioned previously in my introduction, there are few harder things within everyday life than setting a habit, and it can be stressful to keep track of every new self-improvement mission. This can lead to difficulty keeping a habit, and the overwhelming feeling of self-disappointment. My app has the intention of relieving each user of this stress, and creating a beautifully simple expressway to completing goals and achieving satisfaction.

Each person has their own way of keeping track of their progress through a routine such as a workout, and along with that comes the difficulty of sorting out reminders and organization. These issues can be easily eradicated with the introduction of a simple application, dedicated to keeping track of workouts, which provides the ability to provide regulation to the workout/self-improvement scene within many people’s lives.

According to statistics (taken from Statistica, 2022), the main reason (at 40%) for leaving/quitting the gym was that the gym was just too expensive to justify consistently. Soon after that, at ~30%, the reason was lack of motivation to continue. Given that my solution has the ability to overcome these issues (due to the free

installation and reassurance and automation), my application will widely wipe-out the unstable relationship between the user and their workouts.

Throughout this section, I break down the intended functions of the application and include checking their computational amenability:

- **Creator:** One of the key features of my application is the ability to create each workout individually, and for them to be saved for access. This is the main focus of my application, and would have a devoted section within. This would be much more beneficial to the user than simply writing the workouts down, as it gives the ability to store all of them in the same place, leading to an easier experience.
- **Selector:** The selector would allow the user to select any of their workouts and start them. This solution gives freedom within the app, and could negate the issue of locating and using each workout. Within design this may be combined with the player.
- **Calendar:** The calendar has the intent of storing each workout done on each day, this gives an overview of the progression made by the user. In the future I would like to improve this function by adding the ability to create notes or add pictures to each day, giving the ability to view personal progression, not only workout progression. This feature is heavily weighted towards a computerized base, as it provides a stable platform (outside of the risks of losing paper) which can help with

keeping up the habit (due to motivation).

- **Stopwatch:** The stopwatch feature will contain a stopwatch and various timers which can be personalized, allowing quick use of a common timer.
- **Player:** This is the section of the app which should be most prevalent for the user, and most of the time will be spent here. It will automatically set up the next workout within the pre-defined sequence set up by the user when creating the workout. If the user chooses to have more than one workout, then they will have to define the sequence. This reduces hinderance to the process.

1.2 Stakeholders

- **Those who want to stay in shape:** The primary stakeholder for my application is those who choose to opt in for freedom within their workouts. They would use my app to create their own workouts, as their primary aim is to curate an individually specific set of workouts, which can accelerate progression. Looking at the characteristics of this stakeholder I can determine the needs of my application.

Requirements:

-Lack of time: Many people who work out have busy lives, so finding time for a workout can be incredibly difficult. When designing my app it is important to regard the fact that I must slim-line the process of creating a workout, and accessing each workout. The function within my app to automatically play each workout on their specific due dates.

This reduces hinderance and increases the ease-of-access.

-Typically high IT literacy: Assuming that the average user has the intention of downloading a workout app, specifically to use it to its full extent, and also has the ability to use gym equipment, I can gather that the user would have quite a high IT literacy. This gives me leeway for prioritizing functionality over simplicity, giving me more opportunity and use out of my app.

-Lack of motivation: It can be difficult to keep the motivation to pursue working out, especially if it is not due to the love of working out, but instead for aesthetics or mental wellness. I intend to pursue ease of use, meaning that it causes the least amount of bother or stress to carry on and complete the habit.

- **Personal trainers**

One other application of this could be via professional help. Workout trainers and exercise enthusiasts commonly assist other people in creating workouts suited to them – that's their job. I believe that this app would help each personal trainer to pursue their career, which as much help and support for their clients, easing the process.

Requirements:

-IT literacy variation: If the small business owner that is a personal trainer has got to the point at which they must research their market and conclude that an application such as this is a necessity, then I can assume that they have quite a high IT literacy. However, I cannot assume that every customer/client

that the trainer works with has a high TI literacy, which means that I should aim towards an easier to use 'front end' which clients would use (areas such as the player/selector), and a more functional 'backend' which the trainer may use to set up their client's profiles.

-Profiles: Leading on from the previous point, it would create a wide range of new abilities with the introduction of profiles, such as for easy set up for the personal trainer (the process of setting up a workout and asking the client to follow it would be aided greatly), portability for both the clientele and the trainer, and remote access to the database (the trainer does not have to meet the client in person, but can instead remotely set and change workouts as they please.

-Offline use: Sometimes the trainer may not have access to Wi-Fi, or it may not be in the budget to purchase the data needed to run an application which is always online. This means that this stakeholder would appreciate the availability of a working offline app, which only uploads data when online. This means that they would be able to take it out to clients efficiently and cost effectively.

1.3 Research into existing solutions:

1.3.1 Pre-existing solutions:

Prior to development, I have endeavored to find multiple types of research which will both support my approach to designing my application – through improvement on

other's previous work, and showing what I must aim towards to achieve success.

-Workout Maker:

Workout Maker is an app designed to achieve the same solutions as my application. It is one of the best solutions out on the market due to its functionality.

Positives:

- The app is designed around functionality, and yet still maintains a level of aesthetic, which shows that it can be achieved.
- The designers have included many features such as: Workouts, Exercises, Logs, Calendar, Profile. (As shown in the image below)
 - o The feature of Exercises is something that I could introduce later on if I choose to improve my application past its current boundaries.
- When organizing the workouts, it is possible to create folders to store them in. This is a feature I would like to implement.
- When creating a workout, it has options for weights, e.g. None/Free

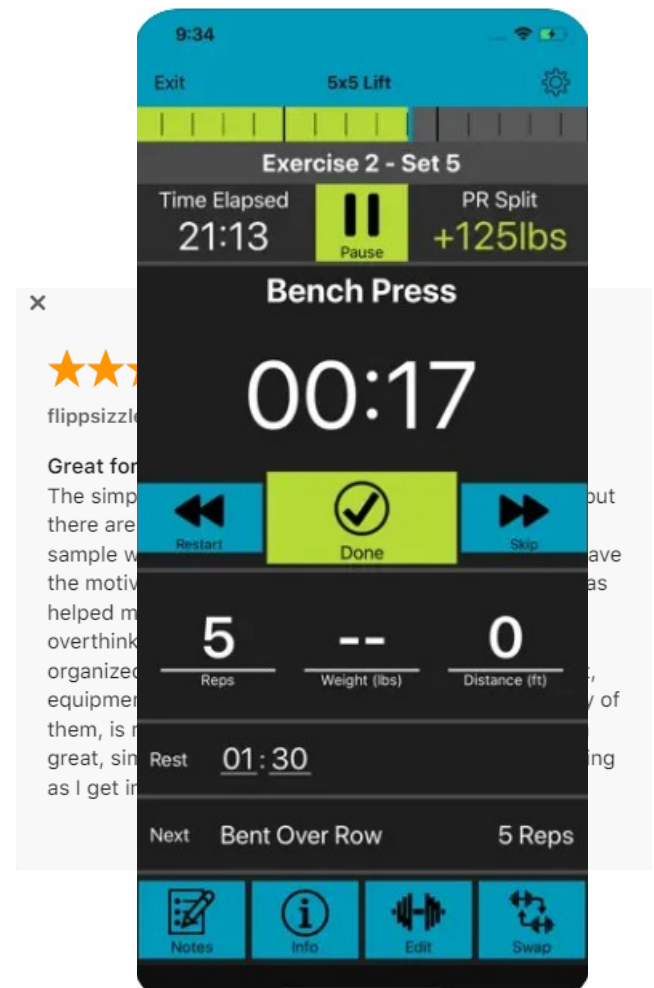


weights/Cables/Strength machine/Multi Use machine/Pull up bar. In addition to this, I would like to add a custom option.

Negatives:

- When creating an exercise, it implies that you must have 'in-detail' knowledge about the muscles that are targeted in a workout. This is a feature I'd like to avoid, as it is demotivating to the user and adds unnecessary difficulty.
- The design of the app is not particularly focused around a

modern, simplistic look, which can



be damaging to the experience for the user. Within my app I intend to follow a more elegant structure, which should hopefully relieve the user from too much of an issue with the flow of the application.

- Overall, this application – while well designed with functionality in mind – feels like it may become too overwhelming to the user, specifically if they are an older user, such as may be the case for a client of the personal trainer.
- The app is only available on the Apple Store, which reduces accessibility

Extra notes:

- The comment here shows that the aim of my application can be achieved, that I can create a fully functional app while maintaining simplicity.
- Also, I can see that due to the high rating, motivation is a crucial factor when working out, and that if I can nurture that motivation then it will elevate user satisfaction and the longevity of my app.

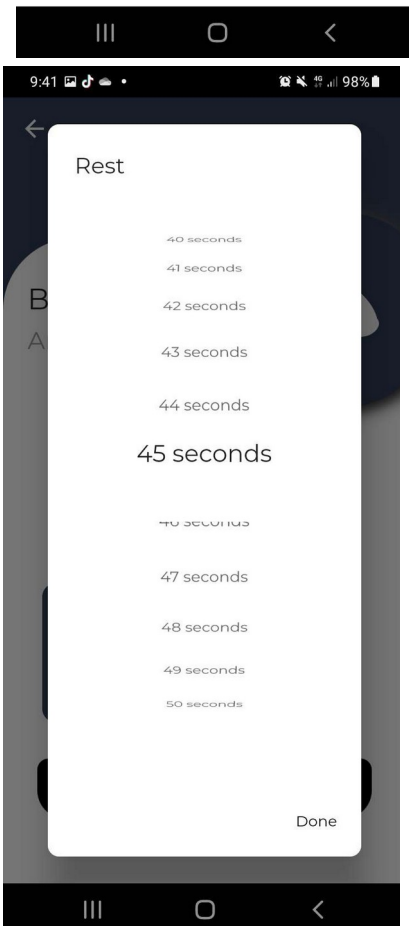
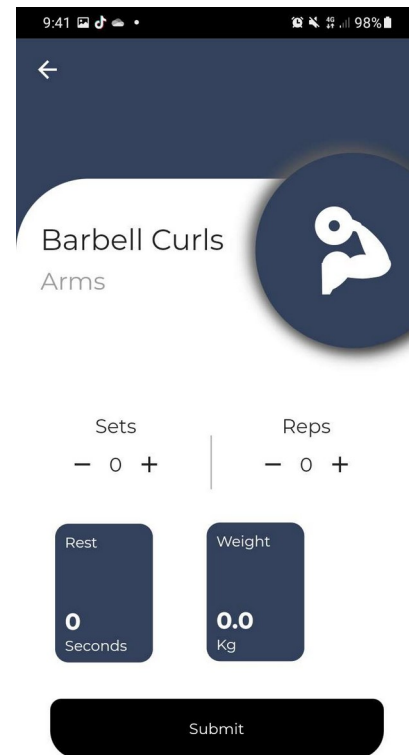
-Workout Builder:

Workout Builder is an app on the Google Play Store. It does not feel as professional as *Workout Maker* but does achieve most of the same aims and features. There are both pros and cons.

Positives:

- The way that *Workout Builder* attempts to create workouts is exactly what I was looking for to model that section off my app from. It includes all of the characteristics which I would also like to include, e.g. Sets, Reps, Rest (which is also done in an interesting way which I intend to add to my app).
- Overall, this app has a simplistic design, which leads to quite easy access and use.

Negatives:



- When create a new workout, the method for choosing the number of seconds of rest is, in my opinion, inefficient (see picture). I now know that I must avoid this method when creating my app.
- *Workout Builder* does not have a feature for profiles, which means that the app must be local to the device, which can lead to issues if the device is out of charge or lost. This can ruin the progress made for a habit and can stop motivation.

1.4 Proposed solution:

1.4.1 Personal software requirements:

I plan on developing my application within Android Studio (2021.2.1 and forward). This allows me to use the Flutter and Dart plugins which allow me to develop my app in the language I need. I use my own laptop for this, with the use of Git and GitHub for version control.

I decided to use my own laptop and computer due to the restrictions of the school computers, as I would not be able to use android studio to its full potential.

1.4.2 Requirements for the user's system:

These are the criteria that the end-user/stakeholder's system will have to meet in order to run my application. My app is designed to be used on a mobile device, perhaps a tablet. These requirements are based on the requirements of other applications of this sort, and the features of this application.

- *Internet access:* I would like to design my app so that it can run both with, or without internet access, however internet access cannot be

forgone due to the necessity of uploading profile details to a database.

- *Android version:* My app has been tested on version 11 android and has been tested virtually on Android 11 too. I intend to run tests on multiple other versions of android. *Workout Builder* (the second my research pieces) can be run on Android 4.1 and up, I intend to mimic this.
- *iOS possibility:* I would like to include support for iOS systems, via the use of swift. This is one of the benefits of using flutter, as it can be configured to be used on either android or iOS quite easily. This feature will only be available if I complete the rest of my success criteria, and with sufficient equipment.

1.4.3 Limitations of the proposed solution:

There are several limitations to my proposed solution, some of which can be overcome with enough time input, whereas others cannot be done due to technological hinderances.

Here are some of the things my app cannot do but could be overcome with more time:

- *Link to smartwatch:* My app would not be able to connect to other devices such as a smartwatch, in order to track progress or update the app automatically. There are other applications such as Garmin Connect, which has this feature. I think that with enough time and devotion I could adapt my app to work in this situation, but it is not one of my aims for my app currently.
- *Show any pre-loaded exercises:* My application is not intended to be a dictionary of exercises, like many

other apps of this kind are. This is because I do not feel knowledgeable about this specific area and feel like it could act against the main purpose of my application – giving the user an easy way out, and reducing the potential efficiency of each workout, and therefore the efficiency of the user's process.

- *Create a personalized workout for the user:* Leading on from showing pre-loaded exercises, my application currently has no intention of creating personalized workouts due to the algorithms behind it. However, as a close follow-up to my completion of my success criteria, I could add in the option for favoriting workouts after they have been created, and then group those workouts together.

Furthermore, these are some of the limitations that I cannot overcome, even with time and money:

- *Show how many calories burnt:* My application cannot connect to other body monitors such as heart-rate or blood oxygen levels. This means that it cannot logically work out, nor show how many calories burnt. I do not intend to include this feature, as it is both out of my scope, and would need support for each of these devices, some of which are protected due to proprietary software (such as some apple devices).

1.5 Success criteria:

Below are the functions that I would like my application to achieve, upon completion. I intend to “tick off” and judge my progress using these criteria and before development, this is how I

intend to consider my application a success once finished. I would like to question my stakeholders and testers if they feel as if these criteria are complete, and how to efficiently work towards them while in development – this means that I can adapt my solution as it is being built.

- 1) *The ability to create workouts:* One of the main functions of this app is to allow the user to create their own workouts, and give freedom to the user for the intensity or type. This includes the ability to customize how the application is used, expanding its real-world uses. For example, a user may enjoy rock climbing, and may have exercises dedicated to that sport, and they would have the freedom of using this feature to add these to a workout – this process could be replicated for someone who loves cardio, or perhaps even swimming! I will consider this feature to be a success if I can make creating workouts to be easy, simple, and applicable to many styles to fit to the user. To quantify this success, I could use a range of test-users to see if the app is useful to them and their style of exercising.
- 2) *The ability to play each created workout:* Once each workout has been created, the main function of the application will be to play/run each workout. This should involve a progression through each activity in the workout, showing the details of each activity, set out by a simple interface. It is important that each workout – when played – is easy to follow, so it is important that this process is simplistic and free of any

clutter – this sets me apart from lots of other applications trying to achieve the same thing, as they tend to show too much data which can confuse the user. Ideally, when each workout is created it should be possible to assign days of the week on which the workout should be played. This functionality will allow the user to press a single button once the app is entered, and enter straight into a workout. This reduces unnecessary hassle when working out and should avoid any ‘road-block’ to motivation. Since this relies on access of the date and time, and the use of a calendar, and since workouts can be played via a select screen, this feature is non-essential to a functioning app. I can consider this feature to be a success if these features work well, and checks all of the boxes for what it is supposed to do. When testing my product with stakeholders, I intend to test whether it does indeed simplify the solution, and how it could be improved in development.

- 3) *To have a simplistic interface:* I intend to have a simplistic and yet functional interface, as to not hinder the motivation of the user. If the user interface is too distracting, inefficient or – on the other hand – boring, then it could dissuade the user from progression. One way that I could meet this criterion is by using large, simple fonts, bright colours, and a colour palette used throughout. Not only the interface, but the in-app process for finding features and using the app is important to me. So, to meet this criterion I will continually

test my application with test-users from my stake holder categories. If they find the process and UI useful and easy, then I will consider this criterion met.

- 4) *Save data to profiles:* One of the features of my application is the choice of using profiles. This is one of the requirements of my secondary stakeholder, as it would allow personal trainers to access their clients accounts and set up workouts for them. Also, this feature would be useful to my primary stakeholder, as it means that they do not rely on local data, but can instead log in on any device anywhere, and so are not tied to one device, which could break a streak in developing a habit. I intend to make profile-use a choice for the user, as some may not want to sign in and give their data away. They should be able to use a guest account. I would consider this feature a success if I can save workouts to a profile, and can successfully log in on a different device to access them.
- 5) *A calendar to view workouts:* When creating a workout (as mentioned previously) it will have the option of being a repeated workout for set days of the week. Not only this, but also just one off, on the day workouts will be stored within the calendar of my application. Any workout done will be stored within the calendar, so the user can see their progress (leading to an increase in motivation for continuity), and what works for them. It will also show “cheat days” or days that have been skipped within a workout

routine. This gives a full overview of the user's calendar, so that they can see what works for them, and what doesn't. When testing my solution, I will check whether this feature is a useful addition, or just clutter. I will be able to quantify this as a success if it is as useful as I aim for it to be, based on user response.

2. Design

2.1 Decomposition of methods

Class	Type	Name	Functionality
Wrapper	Variable	user	Depending on if the user is signed in (has a non-null value), calls either Authentication() or Home()
Authenticate	Variable	showSignIn	Decides whether to show sign in or register
SignIn	Instance	_auth	Instance of AuthService
		_formKey	Global key for each form (email and password)
	Variable	loading	Bool value
		_passwordVisible	Private Boolean value for if to show the password
		email	Holds value of the inputted email
		password	Holds value of the inputted password
		error	Holds value of any error
Register	Same variables as SignIn		
AuthService	Instance	_auth	Instance of FirebaseAuth
	Method	_userFromFirebaseUser	Returns a customUser object with the email and password of the user.
		signInAnon	Uses the firebase function to sign in as a guest with no email or password.
		signInWithEmailAndPassword	Uses email and password parameters to sign in.
		registerWithEmailAndPassword	Creates a new account for a user, with the email and password parameters
		signOut	Uses the firebase function to sign out of an account
CustomUser	Variable	uid	Holds the value of the uid of the user
		Email	Attribute to hold the value of the email of the user.
DatabaseServices	Variable	uid	Parameter to be filled with the uid of the current user in order to carry out functions specific to the

			user.
		WorkoutsCollection	Holds the collectionReference path of the collection which holds the workouts
		usersCollection	Holds the path to where the users unique IDs are held.
	Method	updateUserData	Creates or updates the details document of each user
		updateTheme	Updates the value of the theme for each user.
		getTheme	Gets the theme from the database.
		createWorkout	Creates a new document for each new workout, then creates a collection of activity documents.
		getAllWorkouts	Fetches a list of workouts, and for each workout maps it to an object of the Workout class. For each activity within each workout, maps each to an Activity Object. Returns the list of workout objects.
		deleteWorkout	For the workout being deleted, get all activity documents within the collection and deletes each activity. Then, delete the workout.
		editWorkoutName	Using the workout ID, change the document field of the workout name
		updateWorkoutProgress	Using the workout ID, change the document field of the page progress.
		editActivities	For each activity in ActivityIDsDeleted list parameter, delete the activity document. If the workout contents is not empty, make a list of the activity documents. If the activity being edited has already been created,

			use the activityID to give a path to update the details. If it has not been created, generate an ID and a path, and set the details.
main	Method	runApp	Runs the application
BurnBoss	Instance	_themeManager	
	Variable	themeIsDark	Holds bool value for if theme is light or dark
	Method	dispose	removes listener for theme
		initState	assigns initial values for themeIsDark and adds theme listener
		themeChangeListener	If mounted, it will set the themeMode as being dark
Home	Widget	buildHomeCard	Builds a new card for each call, with parameters for the icon, the label, and the action
NavDrawer	Variable	theme	Checks the theme of the context
		isLightTheme	Boolean value of the theme
		email	Takes an instance of the FirebaseAuth and converts the email attribute to string
	Method	_getUsername	Private function to create the username of the user
	Widget	buildNavBarItem	Builds each item dependent on the inputs for label, icon and action
Workout	Variable	workoutName	Attribute for the name of the workout
		workoutID	Attribute of the ID of the workout
		activities	A list of activity objects of type Activity.
		pageProgress	Attribute to hold the progress through the workout.
	Method	toMap	Maps the data of the workout to fields of a json document stored in the Firebase database.
		updatePage	Updates the pageProgress attribute.

Activity	Variable	activityID	
		activityName	
		reps	
		weightsUsed	
		weights	
		time	
		activityType	
		stopwatchUsed	
	Method	toMap	Maps the data of the workout to fields of a json document stored in the Firebase database.
		fromMap	Maps the data from the json document to be attributes of the activity object.
		updateReps	Updates the value of the reps.
		updateWeight	Updates the value of the weight used.
		updateTime	Updates the value of the time.
		updateActivityType	Updates the type of activity.
ActivityCard	Instance	activity	Takes a parameter for each activity object.
		onEdit	Parameter function to perform an action.
		onDelete	Parameter function to perform an action.
CreatePage	Widget	buildCreateCard	Builds a card for each item to be shown on the create page, with an icon and a title.
NewWorkoutPage	Instance	workoutNameAdd	Adds a TextEditingController to track the text for the workout name.
	Variable	_formKey	Creates a key for each new workout page
		workoutName	An empty string shown in the text field for the workout name
		activities	A list of activity objects
		error	An empty string for errors.
ActivityList	Instance	activityNameController	Text controller for the activity name to be added.
	Method	addActivityItem	Creates a new activity

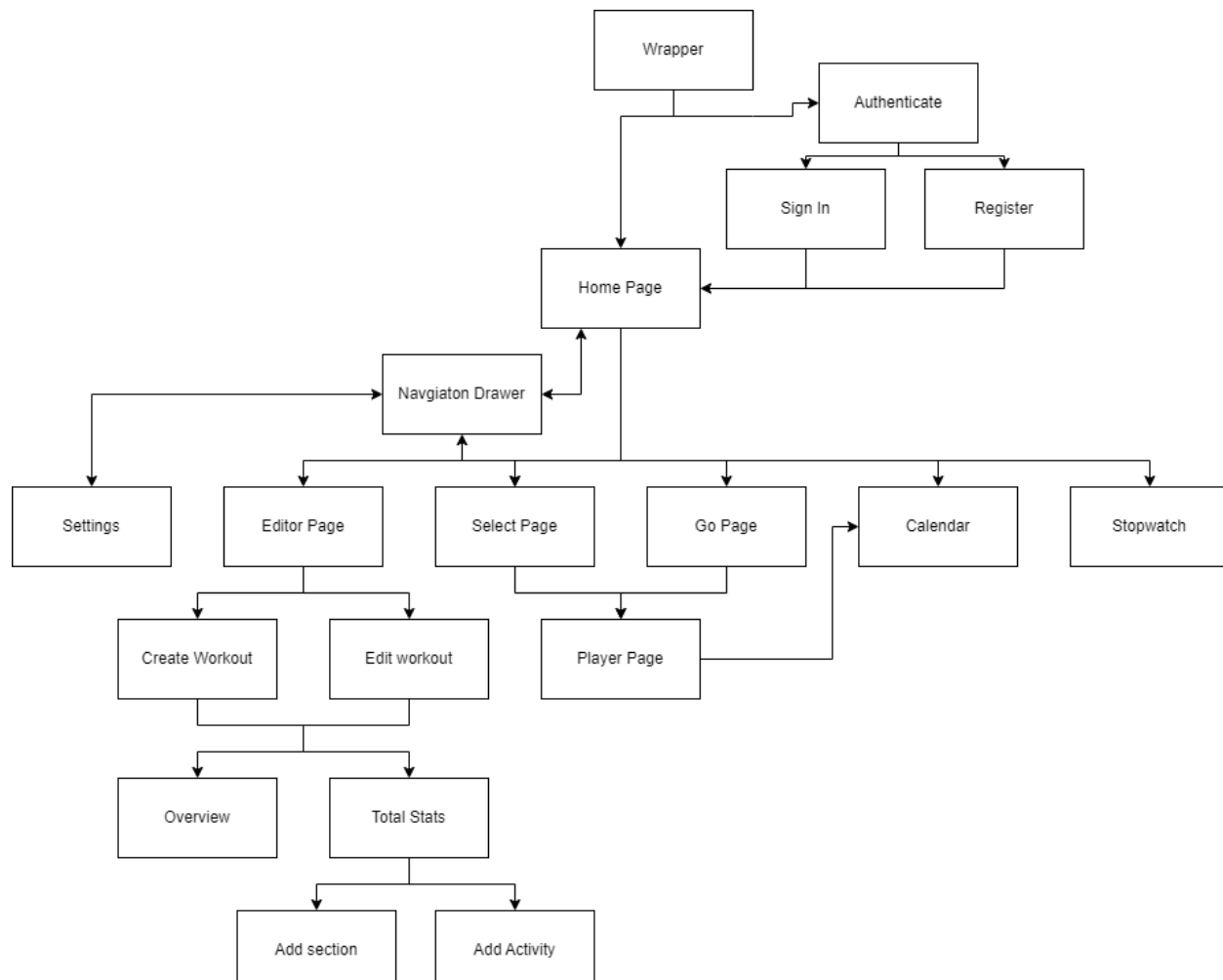
			using the activity class.
		editActivityItem	Function which opens the editActivity page.
EditActivity	Instance	Activity	Instance of activity class
	Required function	onUpdateReps	
		onUpdateWeight	
		onUpdateTime	
		onUpdateActivityName	
		onUpdateStopwatchUsed	
	Variable	isSelected	List of activity types selected
		activityOptions	List of activity types for drop down menu
	Method	initState	Sets the selected activity type.
EditWorkoutPage	Method	initState	Initializes the page and gets all workouts to be displayed.
		_refreshWorkoutList	Rebuilds the list of displayed workouts by using the function getWorkouts from the database class
	Instance	futureWorkouts	Creates a late list of workout objects
WorkoutEditor	Instance	Workout	Requires a workout object
	Variable	changesMade	Bool value of if changes within the workout have been made.
		addingActivities	Bool value for if an activity is being added.
		editingTitle	Bool value for if the title is being edited
		activityIDsDeleted	List of activities which are to be deleted from the database.
	Method	initState	Initializes the number of activities to the length of activities within the workout.
WorkoutPlayer	Instance	Workout	Requires a workout object to be passed in.
	Required Function	onUpdatePage	
	Instance	_pageController	Assigns the page controller for the page builder
	Variable	_currentPage	Holds the index of the

			current page.
	Method	initState	If the page progress is the same as the length, then the workout has previously been completed, so the current page should be set to 0 to restart the workout. Else, set the current page to the saved page progress.
	Widget	buildActivityPage	Builds each activity as its own page, depends on the activity type.
		buildFinishPage	Builds the last page showing that the workout is finished.
StopwatchPage	Instance	_activityStopwatch	Creates an instance of the stopwatch
ActivityStopwatch	Instance	_instance	Creates a singleton instance of the stopwatch to be passed through when called.
		_stopwatch	Creates an instance of the built in flutter stopwatch class
		_stopwatchTimer	A late timer instance to use the built in functionality to keep track of the stopwatch time.
	Variable	_stopwatchResult	Holds the result of the stopwatch to be displayed once the stopwatch updates.
		_stopwatchIsRunning	Bool value keeping track of if the stopwatch is running.
	Method	initState	Initializes the class
		stopwatchDispose	Cancels/disposes of the stopwatch.
		_toggleStopwatchStartStop	Toggles if the stopwatch is to be started or stopped.
		_resetStopwatch	Cancels/disposes of the stopwatch, resets it, stops it, resets the display to '00:00:00'
ActivityTimer	Instance	initialTime	Parameter to be passed in – a duration.
		_timer	An instance of the built in

			timer function.
	Variable	_currentTime	Holds the value of the current time dictated by the timer.
		_timerIsRunning	Bool value to hold if the timer is running
	Method	initState	Initializes the value of the current time for the start of the timer to count down from. Depends on the time value set within the activity object
		_timerCallback	If the timer still holds a value above 0, decrease the timer by 1 second. Else, cancel it.
		dispose	Cancel the timer, dispose of the class instantiation
		_startTimer	If the timer is not active, call the _timerCallback function every second
		_pauseResumeTimer	If the timer is running, pause the timer, if not, resume the timer.
		_resetTimer	Reset the current time to the initial time dictated by the activity object.

2.2 Algorithms and Validation

2.2.1 Flowcharts

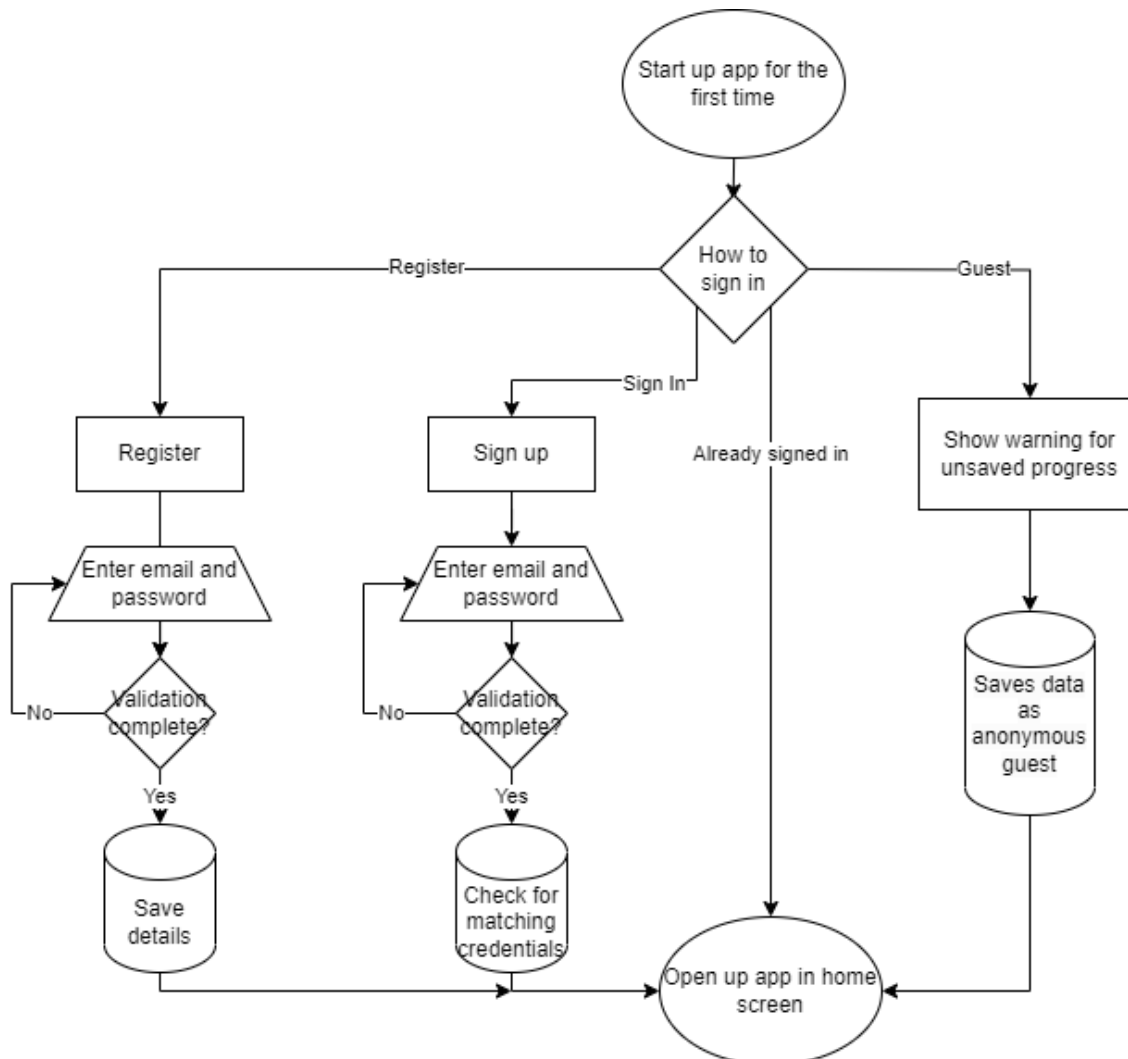


Description:

I intend on having my application focused heavily on easy navigation. I intend on doing this with a navigation bar which is present on each page, so that the user does not feel locked into a single page and can always change with pages. I will describe within *development* about how I will use named routes to generate each page as soon as the app starts.

Hopefully my app should keep to a simplistic approach to navigation in order to not dissuade the user. It should keep a fast paced and logical path throughout, for the user to stay on top of and therefore they should not need to be reminded on how to move through the app.

My home and authentication widgets are wrapped in the Wrapper widget, this is to listen out for any authentication changes which would change the screen shown to the user.



Login:

I have designed the algorithm to authenticate user's login, when given the choice. It is important to me to have the choice of using a guest account, where data is stored locally, as it gives freedom to the user. It is important for users to have an account (either guest or signed in), to save workouts to profiles. Within this process: the user is provided with the sign in page. They are given the option to register for an account, or to sign in as a guest. When using the guest option, it gives a warning that progress and workouts will not be saved to an account in the cloud. I intend on using a database to store workouts and account data.

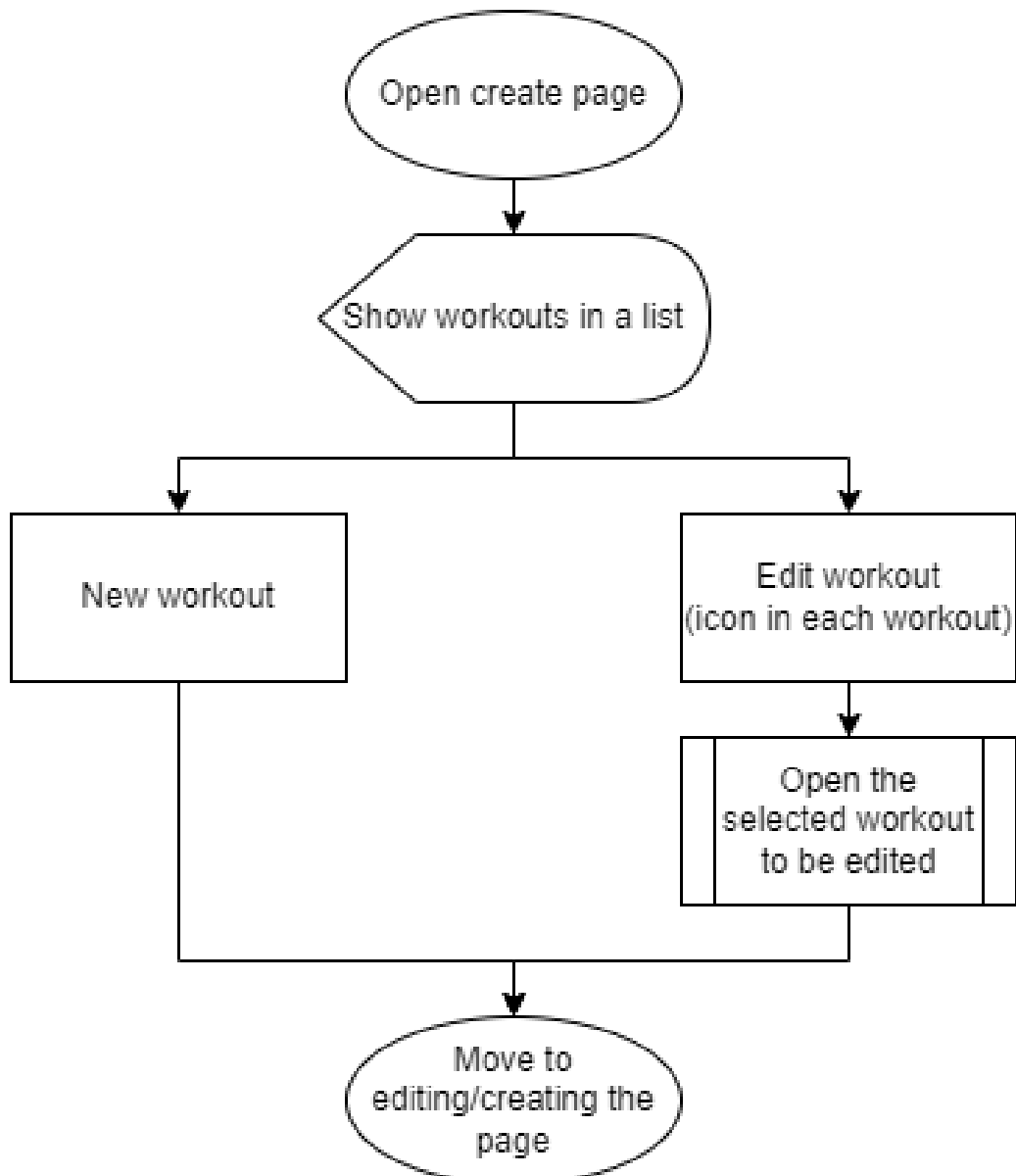
Validation:

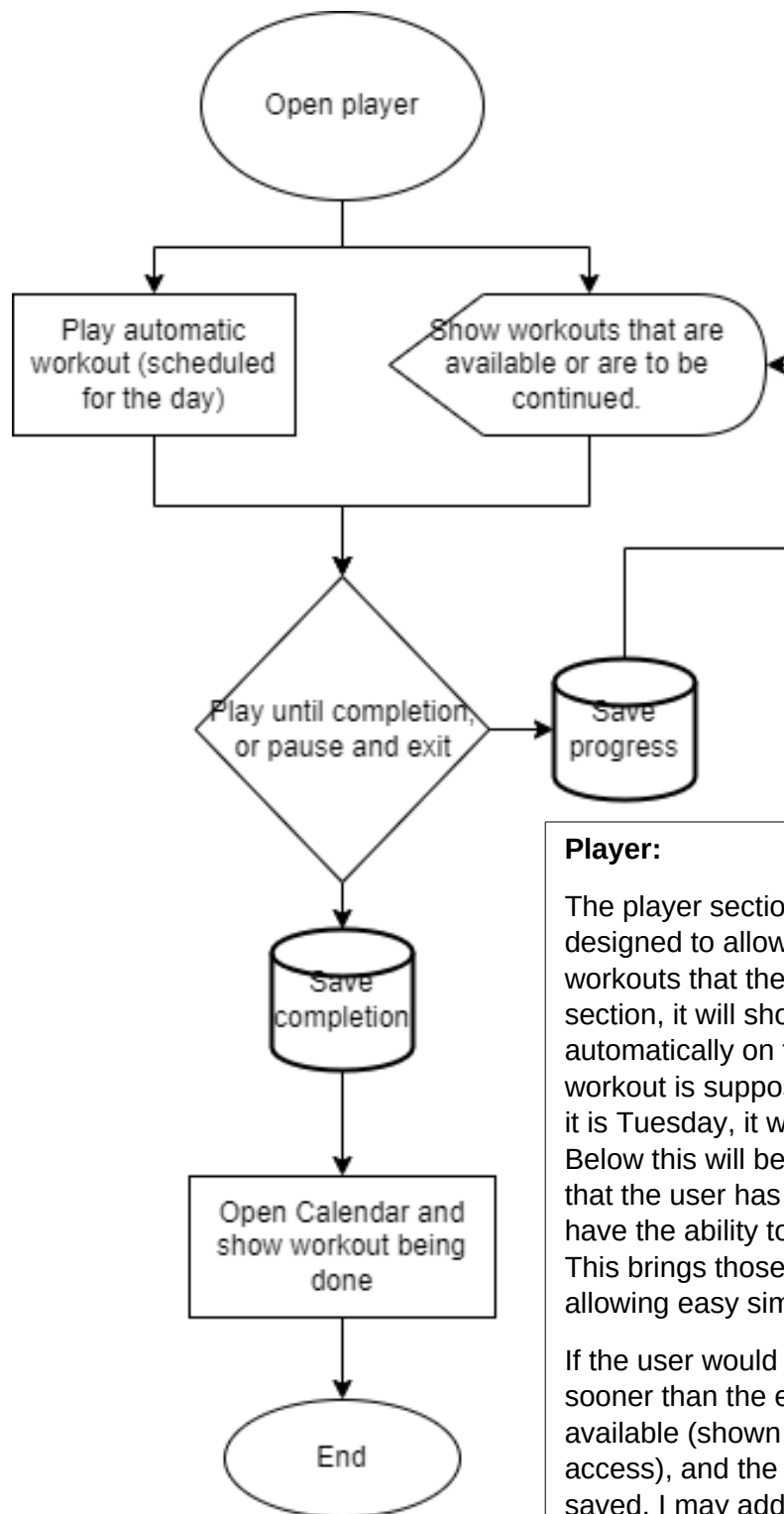
The security and main validation is handled mainly by Google's Firebase. This handles errors in username and password registration or mismatching.

I intend on keeping this process as simple as possible, in order to reduce confusion, and reduce a loss of security through lack of the user's information. Maximum user efficiency should be achieved through large boxes and a simple system.

Create page:

I plan to develop my create page in order to give the user access to creating their own workouts. This is a very important feature of my app, as it gives freedom to the user. I allow for multiple choices to be made, to give as much optimization for the user as possible. It presents 2 tabs: one which shows the overall statistics of the workout being created – this will be updated as the workout is made; and the other, Overview, which allows the user to create a series of activities and class them under groups.





Player:

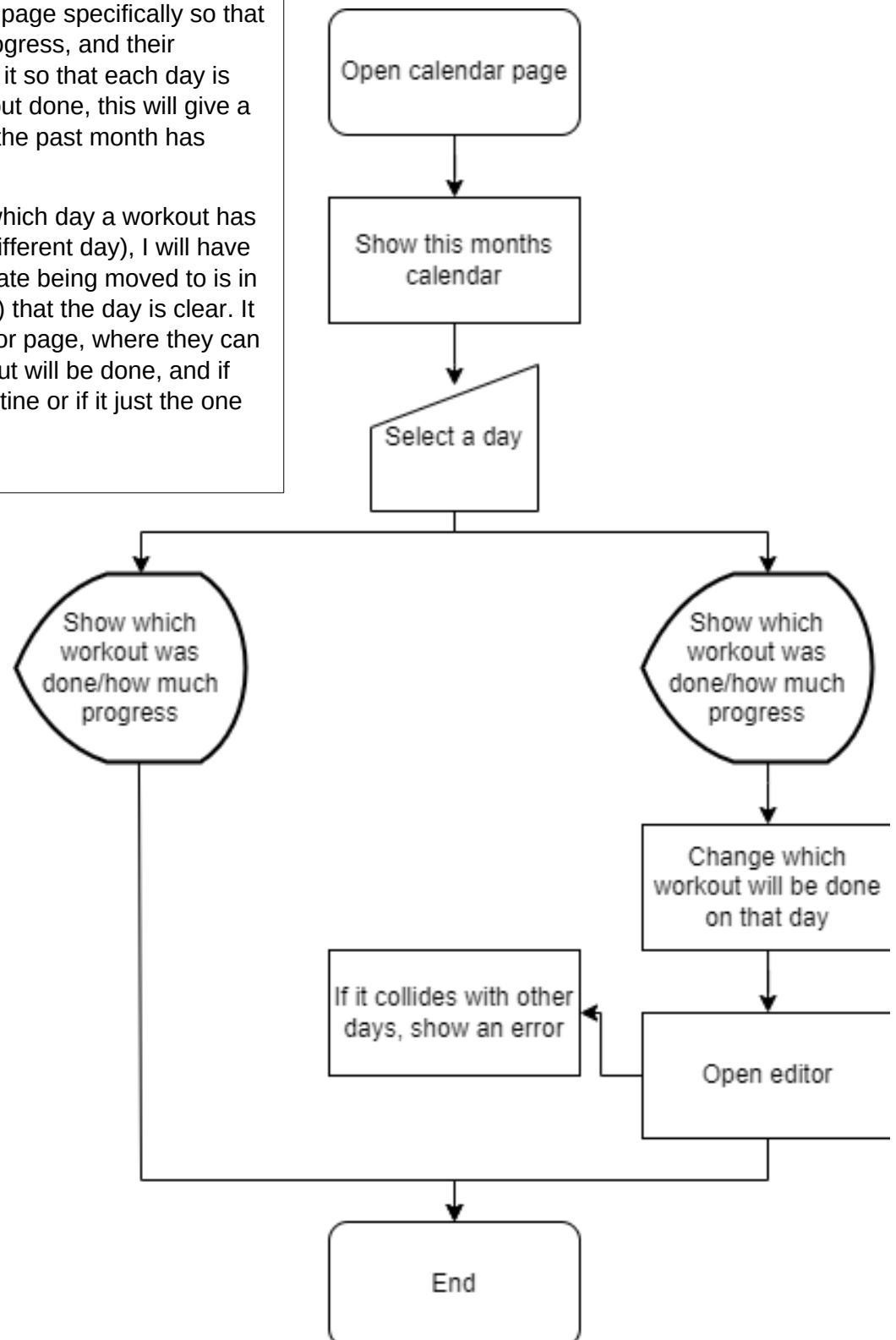
The player section of my application is designed to allow the user to access the workouts that they have created. Within this section, it will show the workout that will automatically on for that day (e.g. if a workout is supposed to be on Tuesday and it is Tuesday, it will be at the top of the list). Below this will be all of the other workouts that the user has created. Ideally, I would have the ability to 'favorite' a few workouts. This brings those workouts to the top, allowing easy simplistic access.

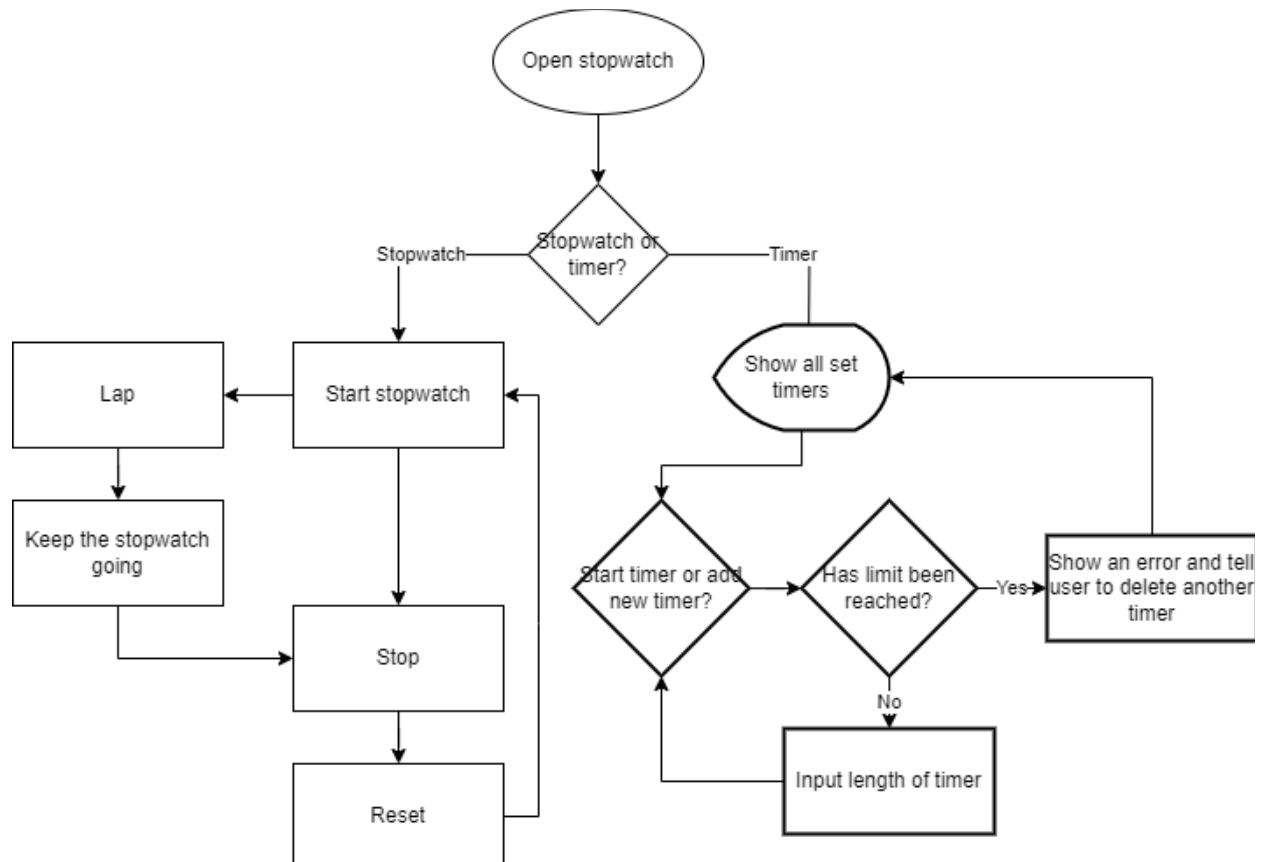
If the user would rather end a workout sooner than the end, a pause button will be available (shown with an icon for easy access), and the workout progress will be saved. I may add in a warning, to confirm that the user would definitely want to exit. This is to reduce errors on the user's part. The progress of the workout will be saved in order to show the user on the calendar later.

Calendar:

I have created the calendar page specifically so that the user could view their progress, and their journey. I plan on designing it so that each day is colour coded with the workout done, this will give a very easy oversight of how the past month has been.

If the user chooses to edit which day a workout has done (move it around to a different day), I will have to A) validate whether the date being moved to is in the past (not possible), or B) that the day is clear. It will take the user to the editor page, where they can edit the days that the workout will be done, and if they want to change the routine or if it just the one off.



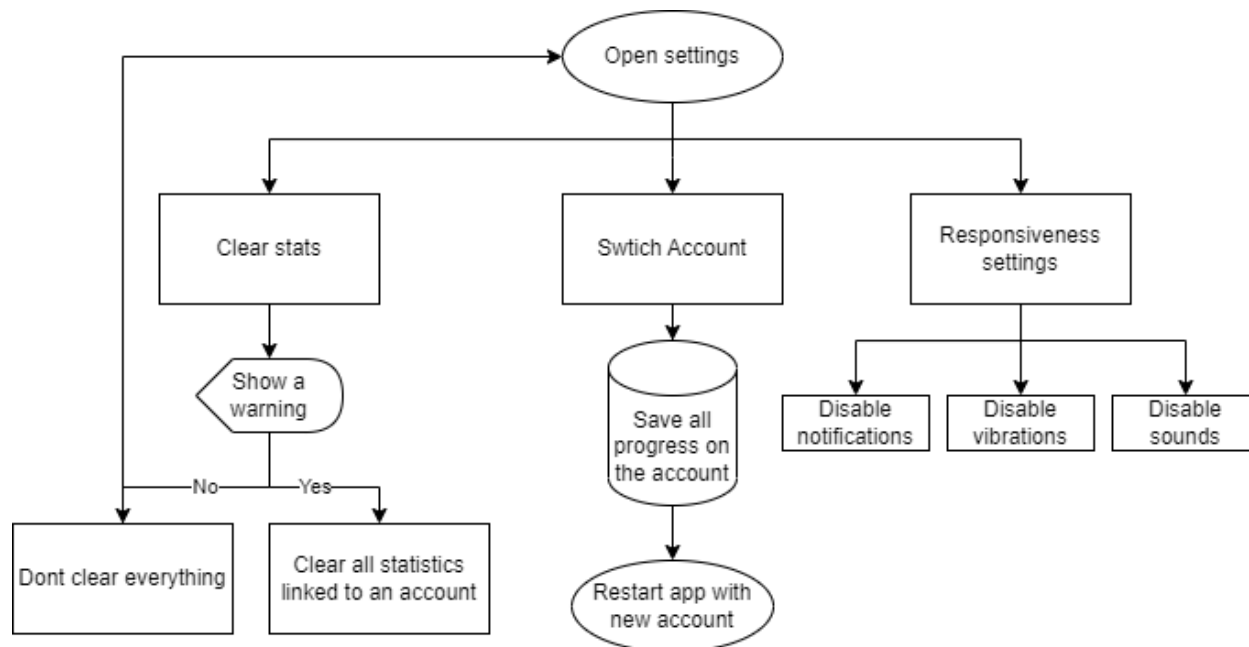


Stopwatch:

I have included a stopwatch/timer section to my app to allow users the freedom of not having to create a full workout if they just want to time one specific exercise in the moment.

Validation:

Within the timer function, I want to only allow the user to have a limited number of timers, so that they can't overload the database that I am using. I am not worried about this limiting the users ability to workout, as they will be able to change the timers very easily.



Settings:

I will need a settings page to allow the users to change their preferences within the app with things such as notification, vibrations, and sounds. Each of these can be positively reinforcing, but to some they can be annoying, so the user should have the choice. They will be set to the <on> position by default, as there is a wider userbase that will use positively react to these noises. This is backed by the use of psychoacoustics within app development. I plan on using high pitched tones in order to follow convention which has been proved to work.

Validation:

The user will have the option to clear their saved statistics. This can be useful if the user has spent a long period of time away from the app and revisit it, or if they have too many workouts that don't suit them anymore. However, it is important to make sure that the user is sure of their decision, as they will have no way of undoing the process of deleting everything. I plan on using a big warning pop up that is coloured red, in order to subconsciously draw attention to the action.

2.2.2 Pseudocode

Workout class:

```
class Workout
  string workoutName
  int pageProgress
  list activities

  public function toMap()
    'workoutName': workoutName
    'pageProgress': pageProgress
    'activities': activities.map(activity.toMap()).toList

  public function updatePage(newPageProgress)
    pageProgress = newPageProgress
```

Activity class:

```
class Activity
  string activityName
  int reps
  int weights
  duration time
  bool stopwatchUsed

  public function toMap()
    'activityName': activityName
    'reps': reps
    'weights': weights
    'time': time
    'stopwatchUsed': stopwatchUsed

  public function fromMap()
    return Activity(
      activityName: map['activityName']
      reps: map['reps']
      weights: map['weights']
      time: duration(milliseconds: map['time'])
      stopwatchUsed: map['stopwatchUsed']
    )

  public function updateReps(newReps)
    reps = newReps

  public function updateReps(newWeights)
    weights = newWeights

  public function updateTime(newTime)
    time = newTime

  public function updateStopwatchBool(newStopwatchUsed)
    stopwatchUsed = newStopwatchUsed
```

Creating a workout (database):

```
function createWorkout(workout)
  Map workoutData
    'workoutName': workout.workoutName
    'pageProgress': workout.pageProgress

  set(workoutData)

  newCollection('activities')

  for each activity in workout.activities

    Activity activity = workout.activities[index]

    Map activityData
      'activityName': activity.activityName
      'reps': activity.reps
      'weights': activity.weights
      'time': activity.time.inMilliseconds

    set(activityData)
```

2.3 Usability

2.3.1 Learnability

When creating and designing my project, it was important to me to make sure that it was easy to use, and easy to learn to adapt to. One of my main themes for the app is that it should not dissuade users from wanting to work out by being too complex. I aim to achieve this simplicity and availability by accompanying buttons with widely used standardized icons. Hopefully, this will allow the user to learn how to use the app, and how to create workouts. When using the workout for the first time, the user should be able to move through it unhindered.

It is important to test my products learnability with first-time, representative users. This will be covered in 2.4 *Testing*.

I intend to design my application using the inspiration from competitors'

products, as if their product is hard to use/easy to be lost in/fails to do its job, then their users leave and the app would not be successful. Some functions of these apps are a skip/rewind button set when carrying out a workout (from *Workout Maker*), as these are widely used terms and icons on things such as music apps. This makes it easy to understand how to use the workout player function. Considering the aspects of UI that their apps use, I can work towards a functional, easy to



use/learn application which serves its purpose.

Another important feature is confirmation of actions such as deleting a workout. Also, when trying to override an automatic feature, it should either

give a warning that it will cause a difference between workout and calendar (for example).

While in the app, to increase usability and learnability, I would like functions to link to other sections or features of the app, such as comments when saving a workout that it will be inputted to a calendar. Ideally, this will link the app to itself, showing users how to navigate around, without the need for a tutorial.

2.3.2 Efficiency

Considering efficiency is very important for my application, as I can afford to sacrifice some ease of use. However I still need to create a fine balance of ease of use and efficiency, in order not to dissuade users or put them off.

It is crucial that I consistently relay back to the overall ethos of my application: that it should give the user freedom, without impediment. Ideally, my stakeholder would move through the workout process quickly and easily. According to *Psychreg*: the 3rd most common reason for quitting the gym is lack of progress, and the 4th most common reason is boredom or lack of interest. Maximizing the efficiency of my application increases the attractiveness of it, and the long-term use by stakeholders, as it keeps them interested and does not destroy the habit created.

2.3.3 Memorability

Throughout my stakeholder selection, it is important that my app is easy to use, memorable, and works as intended. It is vital that my application is set apart from the competition to reduce any external

stress from deciding that the program is not built very well and having to find a new application. This ends up meaning that the user could lose interest in the hobby cultivated, and so would sway them away from working out and using my app in particular. My application's main focus is that the user can create their own workouts. This, combined with the ability to log a workout, means that my application should be set apart from other apps of this kind, and so will hopefully be a memorable piece of software.

2.3.4 Errors

My target audience is typically deemed to have quite a high IT literacy but this does not mean that I should intend to make my app difficult to navigate or use. I would like to make my app useable for everyone, so I need to make sure errors are handled correctly with helpful messages and re-directs, so that users are not dissuaded from using my app. I will test my application with destructive testing, and from this I can adapt using feedback from sample groups to create aesthetically pleasing outputs which help users.

2.3.5 Satisfaction

The main function of my product is to give the user freedom with creating their own workouts. This means that user satisfaction is important to me when developing the application, as to give users freedom, I should cater to their needs. To do this, I have created an interface (*mentioned in 2.5 Interface designs*) which will be subject to change and testing, in order to help the user move freely between sections. This

testing will also help me to see if a prospective user could follow the logic/design, and can be altered if there is other paths should be taken. I will do this testing early on, to avoid any rebuilding.

2.4 Testing

I intend of using multiple types of testing, including unit tests, integration tests, functional tests and user tests done by representatives.

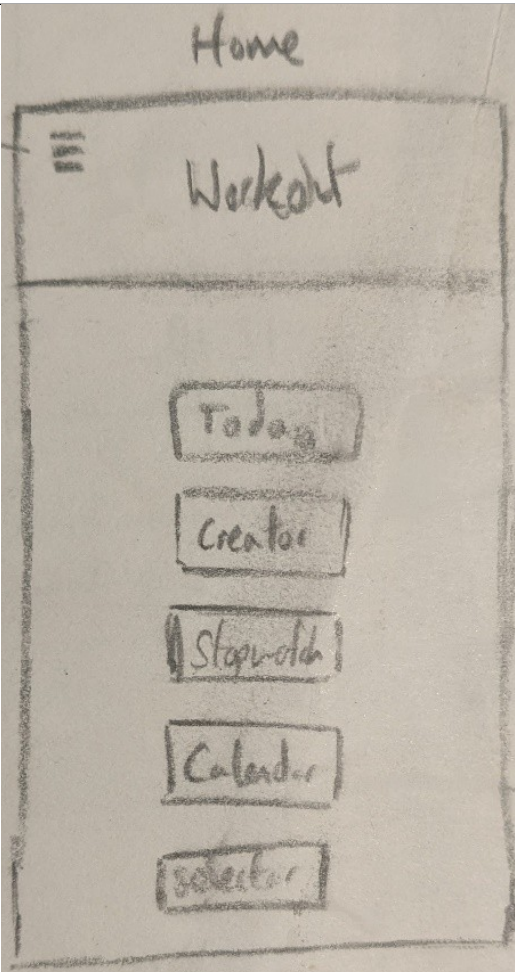
For user testing, I plan on using a small group of around 5 people within my school and friends who would typically use an app like mine, so that I can reduce cost and/or time taken. I plan on testing my app iteratively, checking and changing features of functionality regularly, to avoid complete architecture of my app, which would waste time. Instead of performing tests such as questionnaires, I plan on watching users actually use my app, in order to see the steps they take through the process.

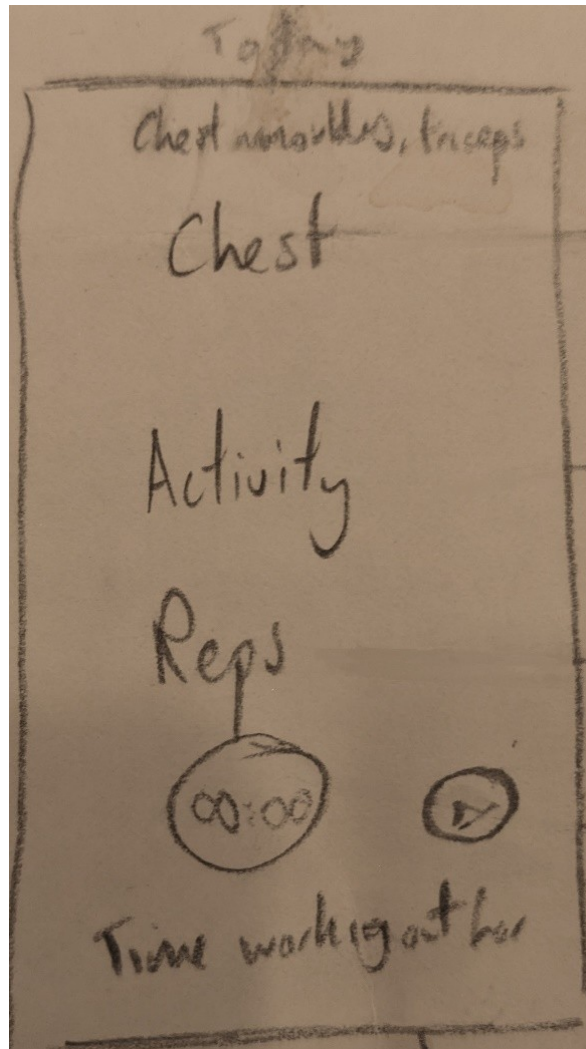
Type of testing	Reasoning	Examples
Feature testing	In order to test my app iteratively within the development process in order to reduce cost/time taken. Ideally, I would reduce the need to totally re-design my application to suit my stakeholders.	Instead of giving my users a lot of questionnaires throughout my development and design process, I will watch how a sample of users actually use my application in order to see their logic and steps throughout the process. An example of this is: How does this create workout page work for you? Are there any steps which are too difficult to complete? How would you approach this task?
Field Study/Adaptive Interviews	I intend on users' feedback to develop my application, as it is important to me that when using abstraction, I focus on the main uses of the app. I must be careful when conducting such interviews, as the human memory is fallible, so when, for example, I ask a user about their gym experience, I will be relying on their memory. I must be wary of the Query Effect (the effect where interviewees create an opinion on the spot to satisfy the question). I intend on reducing this error by asking	I plan to give my users a selection of questions both through development and after they have tried my app in order to achieve maximum efficiency in changing my application. These questions will be things such as: How would you choose to create a workout? What are the types of things you would like to know when you have finished the workout creation? What do you regard to be important when receiving notifications?

	<p>a selection of questions <i>before</i> I explain my design and reasoning, in order to reduce the risk of swaying their opinion.</p>	<p>How often would you use an app of this kind? Where would you use such an app? What would help to inspire you to work out more regularly? This style of question will ideally avoid the query effect. I plan to include an 'I don't know' option to reduce fake answers.</p>
Adaptive architecture testing	<p>When I am developing my app, I plan to use user feedback on small areas on my application to see which is best. This will ideally help me to not have to completely restructure my application at the end, when I complete the rest of my testing. These tests should eliminate any usability errors. This is different to feature testing as it is more directed to areas such as colour schemes</p>	<p>I will use multiple small groups of users to assess small areas of the application. I will do this by giving them a few options and seeing which is best. I may do this by using a few individual small groups and giving them each a different option, and seeing which is used best/most easily. I will take this feedback into account. At the end of development, I will run through this process again to see user thoughts, and to see if I need to rebuild the app with a better structure.</p>
Unit testing.	<p>At checkpoints throughout development, I will run unit tests on small chunks of code. This will help me to eliminate errors that could be overlooked if I were to only look at the end of development. This is important for areas which require a range (e.g. input for amount of rest time), as it will take in use of destructive testing with erroneous data. This is important to create a system that can handle errors within the system, and to improve how the user is redirected. This is important for teaching the user how to navigate the application.</p>	<p>I will run standard unit tests each time I restart the application to catch bugs in compiling, but I will also use thorough unit tests using erroneous data to test inputs. This is how I will 'stress test' my application. From these, I will improve my app to handle such data, or to provide barriers to stop users from inputting such data.</p>

2.5 Interface Designs

Throughout my design process, I have found a few different ways of creating the application I would like to develop. These designs have been changed throughout their process to cater to the user's logic and expectations. I try to keep my designs simple so as to not over-clutter the app, but with enough features to create a well fleshed app.

Prototype 1: Workout App	
Interface tab	Description and Reasoning
Home Page	<p>This is the first prototype for my homepage of the application. It is an important part of the app to have a central homepage which allows the user to navigate through the “sectioned” layout. Each card will have a label and an icon. The icons are quite universal, and their intention is to visually guide the user to navigate through the app. This is important as it keeps the app clean and gives simplicity to the users view. This is to stop the user from getting confused when changing sections.</p> <p>The colour scheme of the home page in this prototype was intended to be quite dark with pops of colour (as is typical of other active/gym style apps), as at this point it was expected that the user's routine will be to use the application either early in the morning or quite late at night – to fit in with a busy lifestyle.</p> <p>The menu hamburger in the top right contains four sections at a time (this is because 1 is active at all times). This is for easy navigation, in order to skip the long path back to the home screen.</p>
	



Within this first prototype is a rough idea for the player section of the app. This gives almost all of the features that I would like, but this has been updated in other prototypes. The prominent features that are here to stay are:

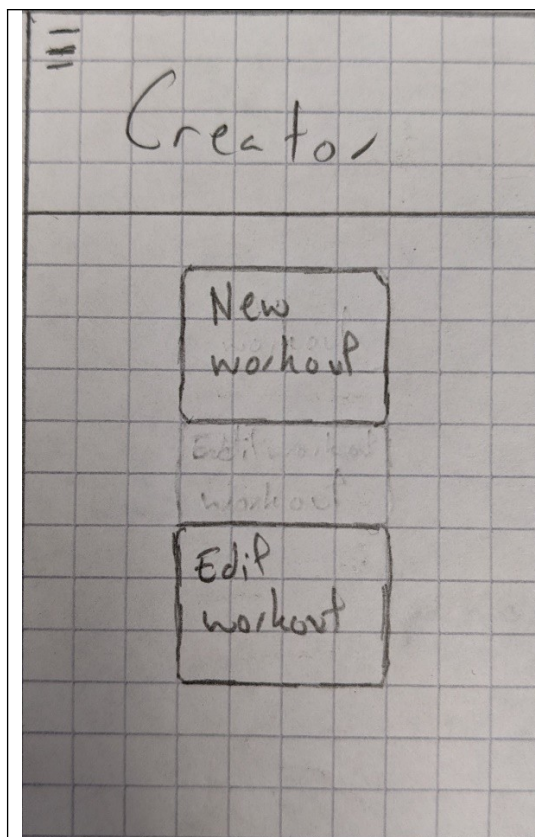
Workout header: this shows which workout the user is on and shows the group of muscles. This is an important focus when doing a workout as it enforces the mind to muscle connection.

Activity name: this shows the activity name, which is important for giving a simple follow through of the process. This keeps the user on track and can help to keep focus.

This attitude is followed through with the "Reps left", which gives the user some peace of mind, reducing the overwhelming feeling of a never-ending workout. Very dissuading!

A timer to show breaks between activities. This should be paired with an alarm to auditorily warn the user. This is a standard type of feature used in workout routine apps.

The video showcase of each activity is an idea only within prototype 1. I have decided that this could be something that could be implemented in the future, but it is not my idea to add this when it is not my original aim to create these videos and upload them.



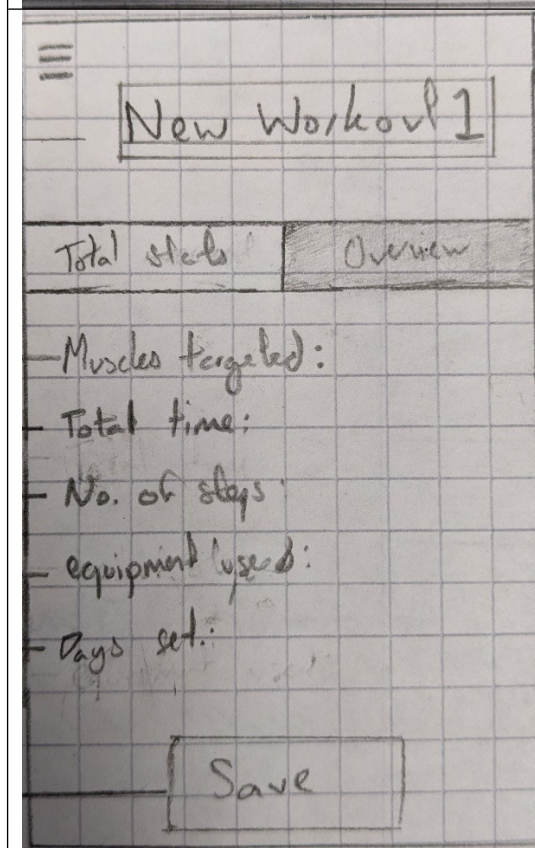
The create section of my app is designed to allow easy navigation and development of workouts. From this point forward (within the app), the user will deal with only the editing or making of workouts.

This modular design of my application is key, as it intended to help the user navigate, and keep a clear map of the application.

From within the creator section there is two sections:

The "New workout" button takes you to a page to create an entirely new workout.

The "Edit workout" will take you to a different page, showing all of the other workouts that are previously made and saved to the users profile.



This page/tab will show the overview of this individual workout.

Ideally this will help the user to proof read their created workouts, and see if it suits them personally. These stats will be linked to other areas of the app such as the calendar.

For example, the 'Days set' section will automatically update the calendar and the player, so that when the user either opens up or refreshes the page for either player or calendar, the workout will show up.

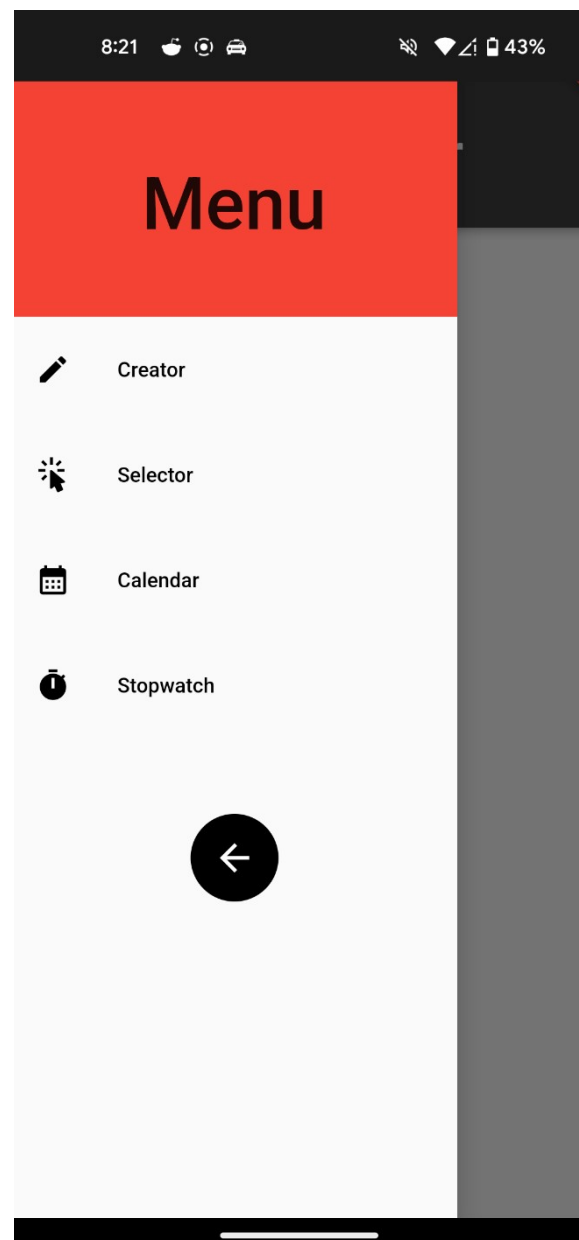
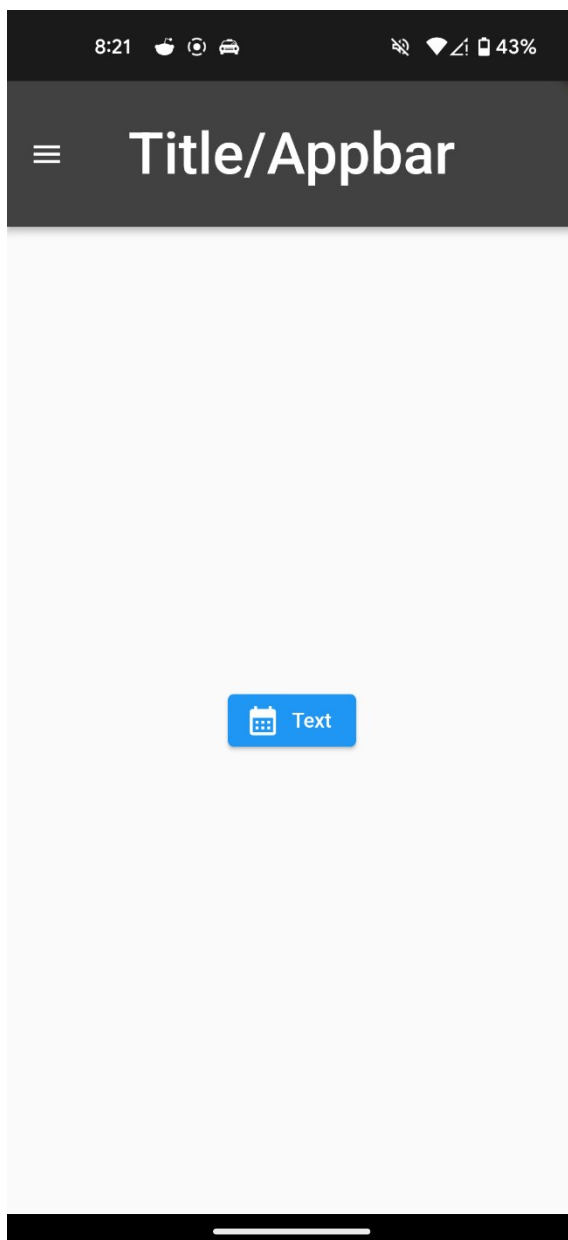
3. Development

3.1 Prototypes:

Before starting the project, I needed to get a footing with flutter to understand the language Dart and the type of application I will be working towards. I have included the videos mentioned throughout this section in a folder called Videos

test_app:

The first prototype that I created was very basic, and included only the navigation drawer and a basic plan on how I wanted my app to fit together. It was a good way to visualize how I wanted my app to work, and to get to grips with creating a clean user interface, and the colours I had designed to use. Shown in video: *test_app*

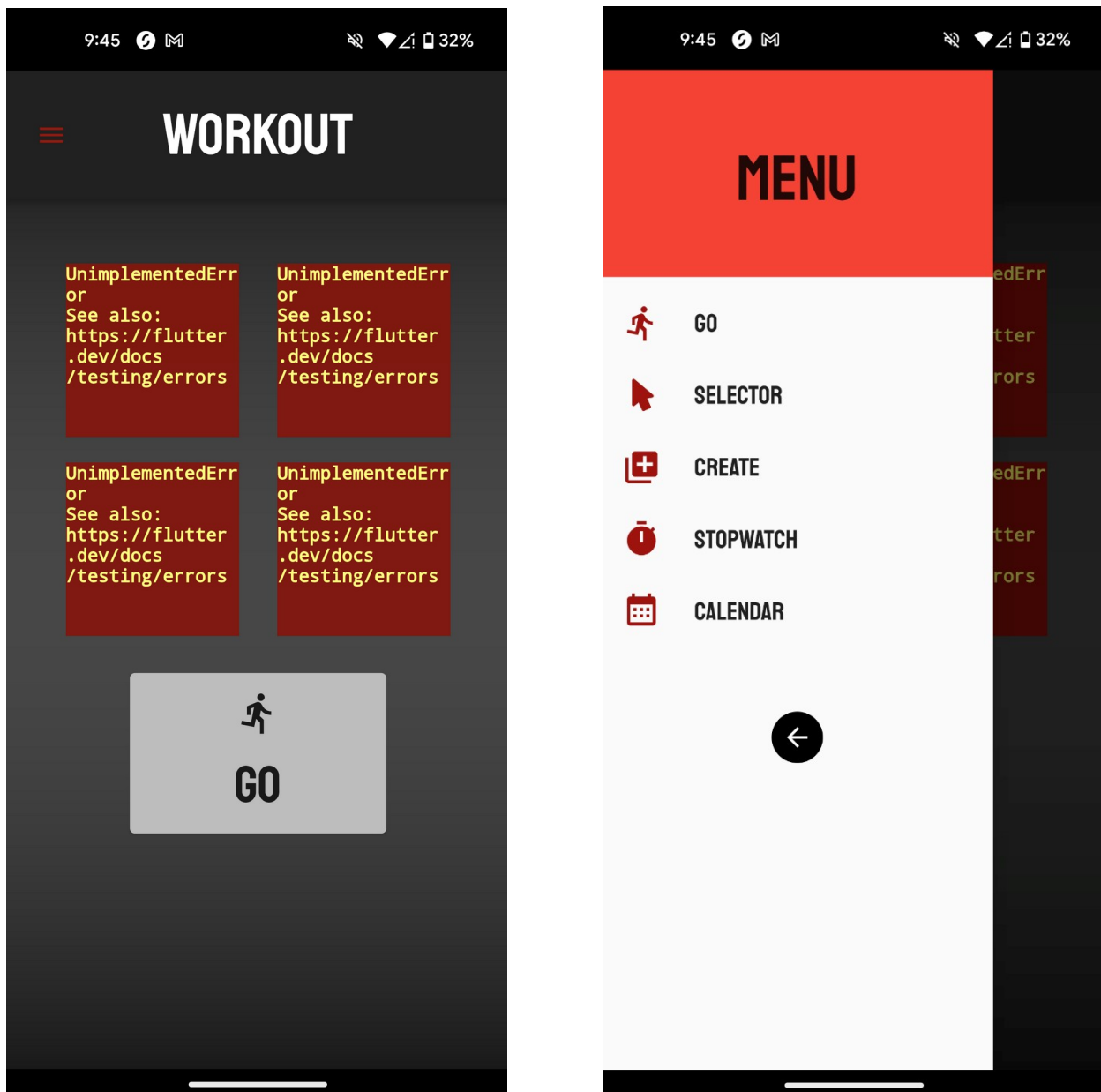


workout_app:

The second prototype of my application didn't have a name yet, I had not designed a name which suited it well enough to commit to naming the project.

Within this prototype, I got further than I did with the last prototype *test_app*, trying to start using modularized code using cards within the home page. However, at this point I was not well enough versed with Dart to implement this and had to research more. This version was only a play towards using different types of code within Dart, and getting used to the more intricate areas.

As is shown in the screenshots, I did not get the cards to work in this prototype. At this point I moved on to the final version of BurnBoss as I felt I had learned enough to make a well-informed decision on design choices and styles.



3.2 BurnBoss:

Throughout my development I have referred back to the documentation for flutter, either api.flutter.dev, or docs.flutter.dev.

I am developing my application and using Git and GitHub for version control. From this, I can walk through the changes easily.

Home Page and Navigation Bar:

In my initial commit, I have created the base for my application. This is the structure which all my “screens” contain as they are the basis for showing information. Without these sections, it is only a class. The different sections are:

`AppBar()` contains the toolbar, which is a banner of sorts which goes along the top of the app. It contains a structure of: leading, title, actions. I have utilized all of these later in the development of the home page. The `appBar` will most likely be present on all of the pages.

`Drawer()` is the navigation drawer which slides out from the left. It contains the routes to other pages.

The next step that I took was to modularize my code by splitting off the navigation drawer into its own class within its own file. Within that, I created a new build widget which meant that I could pass in text and an icon, and it can rebuild each item or this case. This made my code a lot easier to read.

```
buildMenuItem(  
  text: 'Player',  
  icon: Icons.play_arrow_outlined,  
  // selectedIcon: Icons.play_arrow,  
),
```

```
@override  
Widget build(BuildContext context) {  
  return Scaffold(  
    backgroundColor: Color(0xff121212),  
    appBar: AppBar(  
      centerTitle: true,  
      backgroundColor: Color(0xff292929),  
      toolbarHeight: 100,  
      title: const Text(  
        'BurnBoss',  
        style: TextStyle(  
          fontSize: 50,  
          fontWeight: FontWeight.bold,  
          letterSpacing: 2.0,  
          fontFamily: 'Babas',  
        )),  
    ),  
    leading: Builder(  
      builder: (context) => IconButton(  
        color: Color(0xfffff2c14),  
        icon: Icon(Icons.menu_rounded),  
        onPressed: () => Scaffold.of(context).openDrawer(),  
      )),  
    ),  
    drawer: Drawer(),  
    body: const Center(  
      child: Text('Workout now'),  
    ),  
    floatingActionButton: FloatingActionButton(  
      onPressed: () {},  
      child: const Text('Start workout'),  
    ),  
  );  
}
```

```
return Drawer(  
  backgroundColor: Color(0xff121212),  
  child: SingleChildScrollView(  
    child: Column(  
      crossAxisAlignment: CrossAxisAlignment.stretch,  
      children: [  
        buildHeader(context),  
        buildMenuItems(context),  
      ],  
    ),  
  ),  
);
```

At this point, I created a blank page for the Editor Page. This will be referenced to throughout my code (e.g. within the navigation bar) but I will not show it now, as it does not show my progress through the Home Page and navigation bar.

I further modularized by code by giving the Drawer widget only 2 children to display. Inside these widgets I built the header for the NavBar, and the items. These items were wrapped in a SingleChildScrollView which is 'A box in which a single widget can be scrolled'. This meant that if the number of Menu items becomes longer than the screen of the phone, then they can be scrolled through. In my current design, I only have 5 items in my navigation bar. This follows the material3 rules and guidelines which will hopefully make it easier for the user to understand.

```
Widget buildMenuItem({
  required String text,
  required IconData icon,
}) {
  final color = Colors.white70;

  return ListTile(
    leading: Icon(icon, color: Colors.white),
    title: Text(text, style: TextStyle(color: Colors.white)),
    onTap: () {},
  );
}
```

This shows the widget to build the header for the NavBar. The important things in this are the CircleAvatar and the Text. In the future I plan on updating these with respect to the set profile picture and username/email of the users' profile. Currently these are static.

```
Widget buildHeader(BuildContext context) => Container(
  padding: EdgeInsets.only(
    top: 24 + MediaQuery.of(context).padding.top,
    bottom: 24,
  ),
  decoration: const BoxDecoration(
    color: Color(0xff292929),
    border:
      Border(bottom: BorderSide(width: 2, color: Color(0xff7f160a))),
  child: const Column(
    children: [
      CircleAvatar(
        radius: 40,
        backgroundImage: NetworkImage(
          'https://sm.askmen.com/t/askmen_in/article/f/facebook-p/facebook-profile-picture-affects-chances-of-gettin_fr3n.1200.jpg'),
      ),
      SizedBox(height: 12),
      Text('Username'),
    ],
  ),
);
```

Within this code I have written the buildMenuItems function, and included one of the ListTiles to show the home item.

Within the container, I have used padding to position each tile, and then created a Wrap to contain each tile further, so that they can be evenly spaced.

Each ListTile contains a leading icon, the title text, and then what happens when the tile is pressed. Currently this is done with pushReplacement, but this is inefficient as it has to recreate each page when it is loaded up, and the back arrow on users' phones will cause them to exit out of the app. This is not ideal.

```
Widget buildMenuItems(BuildContext context) => Container(
  padding: const EdgeInsets.all(15.0),
  child: Wrap(runSpacing: 14, children: [
    ListTile(
      leading: const Icon(
        Icons.home_outlined,
        color: Colors.white,
      ),
      title: const Text(
        'Home',
        style: TextStyle(color: Colors.white),
      ),
      onTap: () =>
        Navigator.of(context).pushReplacement(MaterialPageRoute(
          builder: (context) => Home(),
        )),
    ),
  ]),
);
```

I had then decided to improve this navigation bar, as it was too built up and not very efficient. My new method involved creating a single widget called buildMenuItem, which contained the structure for each menu item. This meant that I could call the widget, and input each required feature which would be built when the class was called. This meant that I didn't have to rewrite each menu item.

After this, I changed the onTap required element to be action, which meant I could give each menu item their own action.

It also meant that the navigation bar header was set within the Drawer, as it did not need to be taken out.

```
Widget buildCard({
  required IconData pageIcon,
  required String label,
  required GestureTapCallback action,
}) {
  return Card(
```

I took the same approach with building the cards separately for the home page, creating 5 cards which acted as buttons, to move to other pages. This was very similar to the navigation bar structure.

3.3 Theme management:

Theme switch test	Expected result	Result	Show with fix test?
	change the theme	change the theme	few pages

My next step was to work on theme management. I wanted to change the theme between light and dark with a switch.

I had to set up what I wanted each theme to look like:

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

const COLOR_PRIMARY = Colors.deepOrangeAccent;

ThemeData lightTheme = ThemeData(
  brightness: Brightness.light,
  primaryColor: COLOR_PRIMARY,
  useMaterial3: true,
);

ThemeData darkTheme = ThemeData(
  brightness: Brightness.dark,
  useMaterial3: true,
);

themeListener() {
  if (mounted) {
    setState(() {});
  }
}
```

Then I had to set up the class that manages the switch, but it didn't work, I completed thorough testing (shown above).

```
class ThemeManager with ChangeNotifier{

  ThemeMode _themeMode = ThemeMode.light;

  get themeMode => _themeMode;

  toggleTheme(bool isDark){
    _themeMode = isDark?ThemeMode.dark:ThemeMode.light;
    notifyListeners();
  }
}
```

Along with this, I needed functions to start listening, listen for the change of theme, and stop listening again (these were part of the problem as to why it didn't work):

Then, I had to create the switch that the user interacts with to change the theme:

```
actions: [
  Switch(
    value: _themeManager.themeMode == ThemeMode.dark,
    onChanged: (bool newValue) {
      setState(() {
        _themeManager.toggleTheme(newValue);
        print('Switch changed');
      });
    })
],
```

then fixed the problem by moving the listener for the theme change further up the path, meaning that it could listen to what it needed to. Also, I added in some print statements throughout, to show where the process was getting stuck.

```
| void initState() {
  super.initState();
  themeIsDark = _themeManager.themeModeIsDark;
  _themeManager.addListener(themeChangeListener);
}

themeChangeListener() {
  if (mounted) {
    setState(() {
      print("themeListener called");
      themeIsDark = _themeManager.themeModeIsDark;
    });
  }
}
```


Before, I was not calling the function to relay the listener, or changing anything to show that the theme had changed.

The switch now actually set the theme to be dark, and showed it.

```
Switch(  
  value: widget.themeManager.themeModeIsDark,  
  onChanged: (bool switchIsOn) {  
    setState(() {  
      print('Switch changed to $switchIsOn');  
      widget.themeManager.setThemeToDark(switchIsOn);  
    });  
  })
```

Within the theme manager, I created a kind of if else statement, where if the switch

changed the theme to dark, then the thememode would be set to dark, otherwise, it is set to light.

```
get themeMode => _themeMode;
```

```
setThemeToDark(bool isDark){  
  print("Theme mode toggled to ${isDark? "dark":"light"}");  
  _themeMode = isDark?ThemeMode.dark:ThemeMode.light;  
  notifyListeners();  
}
```

```
get themeModeIsDark => _themeMode == ThemeMode.dark;
```

When designing the colour palette for my application I took the original colours from the prototypes, and used the material color model, along with research into theories into which colours cause different feelings – it was my aim to create a colour scheme which was bold enough to make the user motivated and in the mood for working out, without coming across as gaudy or offputting. The light-mode theme was put forward to external testers who confirmed my colour scheme (options shown in CHANGING_THEME_COLOURS). In regards to the dark-mode, I considered the uses. I came to the conclusion that users would need the dark-colour scheme at night (after dusk or early in the morning), and would need a ‘less-stress to the eye’ approach. This caused me to land on a dark grey, with a contrasting light/turquoise blue to hold the motivating theme.

3.4 Routing:

As mentioned above, using pushReplacement for my navigation was inefficient, as it meant that each class had to be re-rendered when it was called, and there was no path through the routes. This was an issue, as it is not how other apps work, or how I intended my app to behave. I used the routing ability of flutter to render each

```
initialRoute: '/',  
routes: {  
  '/': (context) => Home(_themeManager),  
  '/calendar': (context) => CalendarPage(),  
  '/editor': (context) => EditorPage(),  
  '/select': (context) => SelectPage(),  
},
```


'page' when the app was loaded, which meant that it could then create a path to navigate. My initial route at this stage is my home page, however this will have to be updated to work with authentication.

3.5 Improvements:

At this point, as I was finding the need for more build widgets (especially cards), I had to improve my code and give stronger variable names, e.g. from buildCard to buildHomeCard, to specify that this card was for the home page.

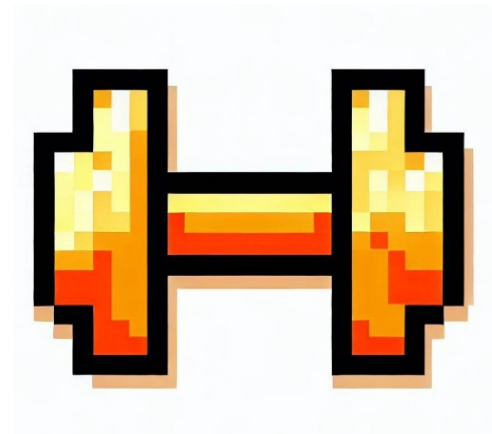
Also, in an attempt to improve my code, I separated my home page from my main class. This meant that I had a strong distinction between screens, and helped with organisation. This also helped later on, when I wanted to move my theme switch to my settings page, as it was much more obvious which code needed to be kept within my main class.

As mentioned at the start of this section, I have been using Git and Github as version control. Another feature of this is that I can create issues, and then once fixed, I can add this to my commit, and it will link my commit to the fix of the issue. This is useful to me as it shows me how I fixed issues throughout, and how I can fix such issues if they arise in the future.

3.6 Icon:

I changed my app icon to a logo that I have designed:

I then implemented it using a flutter package called flutter_launcher_icons. This works in the home page of my Google Pixel 5, but not in the tab menu. It is important to me that users can locate my application within their home screen. This dependency was added to the pubspec.yaml file.



3.7 Functional app:

My plan for this point forward was to work on making my app to solve the issue I started with, as I was satisfied with my set up (e.g. creating the home page + navigation). It is important to me to meet the deadlines set, and unrealistic of me to expect development of every feature within my success criteria to be completed. As long as the application performs the basic functions, it can be ruled as functional.

I started by creating the Creator page, which will allow users to create new workouts, and edit previous workouts.

Within the Create page, I added two cards (each as their own modular widget) which show the "New workout" option, and the "Edit workout" option. Currently, to create a functional app that fills the base requirement I intend for it, I only need work workout on the "New workout", as it is the central feature.

Within this “New workout” page, I included a tab system, as is shown in my initial designs, containing the “Total stats” tab, and the “Overview” tab. This was handled by the widget DefaultTabController, but I had to work with the theming for this, as I have both light and dark theme.

The next step for me in creating the functional app was to involve a backend. After some research, I decided on Google’s Firebase, as it will be easy for me to integrate, has control for authorisation, will allow me to utilise a nosql backend, and will fill my success criteria for my app to work both offline, and online through Cloud Firestore’s offline support.

Authorisation:

Google Firebase provides an easy ride for setting up authorisation within mobile apps.

The videos linked to this section are called: SIGN_IN_AND_REGISTER, AUTHENTICATION.

My first step was to define a clear ‘Widget tree’ which would allow me to properly set up authorisation. For this, I need a ‘Wrapper’ surrounding my Home, and my Authorisation classes, so that it can listen for any auth changes.

For authorisation, I wanted for users to be able to sign in ‘anonymously’/ as a ‘guest’.

First, I have to create the function to sign in anonymously. The function signInAnonymously is a function of Firebase. This may take a while so it is awaited, and

```
bottom: TabBar(
  indicatorSize: TabBarIndicatorSize.tab,
  indicator: BoxDecoration(
    borderRadius: BorderRadius.only(
      topLeft: Radius.circular(10),
      topRight: Radius.circular(10)),
    color: Colors.white,
  ),
  tabs: [
    Tab(text: 'Total Stats'),
    Tab(text: 'Overview'),
  ],
),
body: TabBarView(
  children: [
    Column(crossAxisAlignment: CrossAxisAlignment.start, children: [
      Text('Muscles Targeted: '),
      Text('Number of steps: '),
      Text('Equipment used: '),
      Text('Days set: '),
    ]),
    Column(crossAxisAlignment: CrossAxisAlignment.start, children: [
      Text('Group 1'),
      Text('Activity 1'),
      Text('Group 2'),
    ]),
  ],
),
```

```
if(user == null) {
  return Authenticate();
} else {
  return Home();
}
```

```
Future signInAnon() async {
  try{
    UserCredential result = await _auth.signInAnonymously();
    User? user = result.user;
    return _userFromFirebaseUser(user);
  }catch(e){
    print(e.toString());
    return null;
  }
}
```

```
UserModel.customUser? _userFromFirebaseUser(User? user) {
  // ignore: unnecessary_null_comparison
  if (user !=null) {
    return UserModel.customUser(uid: user.uid);
  }else{
    return null;
  }
}
```

```
class customUser {
  final String uid;

  customUser({ required this.uid });
}
```

wrapped in a try catch, so that it doesn't break/crash my app if the response doesn't reach my app.

Then, I have created my own customUser object, as I don't need all of the data that the request splurges, and it only responds with the uid. This is used when the request is sent to sign in, which is sent to the stream. This confirmation and data (stored as user) is sent to the Wrapper, which then shows the Home screen and allows the user access.

```
Stream<customUser?> get user {  
  return _auth.authStateChanges()  
    .map(_userFromFirebaseUser);  
}
```

After this, the user can sign back out again using the built in Firebase functions.

```
onPressed: () async {  
  dynamic result = await _auth.signInAnon();  
  if (result == null){  
    print('error signing in');  
  } else {  
    print('signed in');  
    print(result.uid);  
  }  
}
```

```
TextButton.icon(  
  icon: Icon(Icons.person),  
  label: Text('Sign Out'),  
  onPressed: () async {  
    await _auth.signOut();  
    Navigator.pushNamed(context, '/');  
  },  
)
```

```
bool showSignIn = true;  
void toggleView() {  
  setState(() => showSignIn = !showSignIn);  
}
```

As mentioned above, I need the two screens, sign in and register. I have done this within my authenticate page. This is very simple, as all it does is reverse whatever the result of showSignIn.

```
@override  
Widget build(BuildContext context) {  
  if(showSignIn == true){  
    return SignIn(toggleView: toggleView);  
  }else{  
    return Register(toggleView: toggleView);  
  }  
}
```

For signing in with an email and password I have used a Form with TextFormFields. Within the form I have a form key. This means that whenever anything inside the form is changed, it is tracked, and the form key helps to validate each field.

```
child: Form(  
  key: _formKey, //tracks state of form and helps validate it  
  child: Column(  
    children: [  
      TextFormField(...), // TextFormField  
      SizedBox(height: 10),  
      TextFormField(...), // TextFormField  
      SizedBox(height: 20),  
      ElevatedButton(...), // ElevatedButton  
      SizedBox(height: 10),  
      Text(...), // Text  
      Row(...), // Row  
    ],  
  ), // Column  
) // Form
```

```
TextFormField(  
  decoration: InputDecoration(  
    hintText: 'Email',  
  ), // InputDecoration  
  validator: (val) => val!.isEmpty ? 'Enter an email' : null,  
  onChanged: (val) {  
    setState(() => email = val);  
  },  
)
```

This is an example of validation within my login. The validator checks to see if the email text field is empty (which it cannot be), and sets the state. If the email field is empty, then it will be sent as an alert to the user. Email validation is taken further by Firebase, as it checks to see if the email is valid, as shown in the picture below.

If everything passes through correctly, then the email and password will be registered, and no error will be returned.

```
Future signInWithEmailAndPassword(String email, String password) async {
  try{
    UserCredential result = await _auth.signInWithEmailAndPassword(email, password);
    User? user = result.user;
    return _userFromFirebaseUser(user);
  } catch(e) {
    print(e.toString());
    return null;
  }
}
```

The function to sign in with an email and password takes the entered email and password, and returns the verification that the user is in the database. Otherwise it will show the error that has arisen due to the user's details being unidentifiable.

This function is called when the "Sign in" button is pressed.

Now that both of these functions were created and both work, I wanted to present both as options to the user. This completes some of my success criteria point to store data to a profile and to be able to sign in as a guest.

This code shows the warning alert (designed to increase usability), which only runs the code to sign in if the user chooses to carry on.

From this I could add the email of the user (and other implemented details) to a new collection called Details. This will contain a collection of documents including the profile picture and perhaps a username that the user enters later.

```
child: Text('Register'),
onPressed: () async {
  if (_formKey.currentState!.validate()) {
    dynamic result = await _auth.registerWithEmailAndPassword(email, password);
    if (result == null) {
      setState(() {
        error = 'Please supply valid email';
      });
    } else {
      return null;
    }
  }
},
```

```
title: const Text('Warning!'),
content: const Text(
  'Signing in as a guest means that your workouts will not be transferred!'),
actions: <Widget>[
  TextButton(
    onPressed: () =>
      Navigator.pop(context, 'Cancel'),
    child: const Text('Cancel'),
  ),
  TextButton(
    child: const Text('Proceed'),
    onPressed: () async {
      Navigator.pop(context, 'Proceed');
      dynamic result = await _auth.signInAnon();
      if (result == null) {
        error = ('error signing in');
      } else {
        print('signed in as guest');
        print(result.uid);
      }
    },
  ),
],
```

Creating new workouts:

My new workout page can be split into areas: The new workout page, the Total Stats tab, the Overview page, and the New activity page.

The new workout page:

I decided that I wanted the new workout page to be interactive, so I made the workout name as an interactive text field as the title. For this to be added to the database, it needed to be validated. As long as the workout name field is not empty, the variable that is written to the database is filled.

```
title: Form(  
  key: _formKey,  
  child: TextFormField(  
    decoration: InputDecoration(  
      hintText: 'Workout name',  
    ),  
    controller: controller,  
    validator: (val) => val!.isEmpty ? 'Enter a workout name' : null,  
    onChanged: (val) {  
      setState(() {  
        workoutName = val;  
      });  
    },  
  ),  
)
```

This method creates the page for workouts. There is no data that is currently set to the workout, so the only data that is mapped is the workout name.

This method is not very modular, so had to be moved to the database page.

This method doesn't add each workout to the user, because at this point I did not know how to take the uid from the registering function.

The next commit meant that I took the uid from the authentication database and wrote the workout to that. This meant that each users has their own selection of workouts.

```
await DatabaseService(uid:  
  FirebaseAuth.instance.currentUser!.uid).createWorkout(workoutName);  
//calls the create document from database to create the workout with the  
workout name
```

Groups:

I had originally planned on having a workout split up into groups of activities, so this section is showing the process of adding groups.

The first step was to add groups to the workout model. From this, I wanted to add in text fields which could be added with a button. I needed to be able to create the new group text field, enter the name of the workout, then add the result to the page. I did this by first defining the empty list, then taking the result and adding it to the list. Then, I mapped each group item out to a list.

From this, I could then remove each group with this code:

```
ElevatedButton(onPressed: () {setState(() {  
  groups.removeLast();  
});}, child: Text('Remove group'))
```

```
.map((group) => Column(  
  crossAxisAlignment: CrossAxisAlignment.start,  
  children: [  
    Container(  
      color: Colors.black12,  
      height: 2,  
    ),  
    Text(  
      group,  
      style: TextStyle(  
        fontWeight: FontWeight.bold,  
        fontSize: 25,  
      ),  
    ),  
  ]),  
).toList(),
```

This was an overall bad method of creating groups, even though they were only text additions. I had to re-evaluate if I needed groups as a function. They were not in my success criteria and are not an essential feature. Doing some more research – including the app Garmin, I came to design a new structure for creating workouts, where each activity was an object within the workout.

Shown in video: ADD_GROUPS.

Activities:

The first step towards adding activities to the database was to take each workout from the newWorkout page, and map the data. Then, for each activity in the activities list I could add them as their own object within the database. At this point, the only data within the workout was hard-coded sample data which I could use to test to see if the mapping worked. This used a method within the Activity class called toMap which converted each attribute to a String.

```

Map<String, dynamic> workoutData = workout.toMap();

// Add the workout to Firestore
DocumentReference workoutDocument = await WorkoutsCollection.add(workoutData);

// For each activity in the workout, create a subcollection within the workout document
for (int i = 0; i < workout.activities.length; i++) {
  CollectionReference activitiesCollection = workoutDocument.collection('activities');
  Activity activity = workout.activities[i];
  Map<String, dynamic> activityData = activity.toMap();
  await activitiesCollection.add(activityData);
}

```

The next step after this was to be able to input data into the activities. I did this by creating each new activity object as an instance of a card. I took in 2 arguments, the activityName and the action which would be completed. The reason why I had this action outside of the card was because if it was inside the card, it would struggle with creating the new page when navigating to it.

Also, I added a new page in which each object in the list of activities would be opened, however there was no data about each object.

At this point, my code was messy, and it needed to be cleaned up with some more modular classes and widgets. My structure went as follows:

NewWorkout page holds the method to set the attributes for each workout.

activities list holds the list of activity objects

ActivityList holds the functions to add, delete, and edit each activity.

Add function instantiates the activity, and adds it to the list.

Edit function opens the edit activity page for each activity object.

Delete function removes the activity object in the index.

ActivityCard is the blueprint for each activity item in the workout page. It assigns both the edit icon and the delete icon to the correct functions.

Each activity is added by entering the name of the activity within a text box, and then the activity is created.

Obviously there is no need for both the New Activity button and the add activity button, so one of them has to be removed.

The video *ACTIVITY_ADDING* shows how activities are added.

Validation:

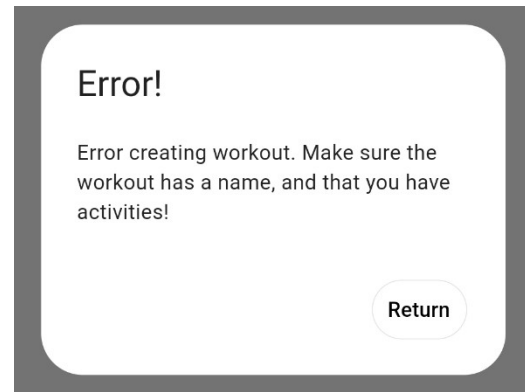
In order to make sure that I maintain referential integrity within my database, I needed to validate the state of a workout before it is saved. Some of the things I needed to watch out for were:

If the workout had a name.

If the workout contained activities.

If I can make sure that these criteria are met, I know that when I come to read the activities, I won't run the risk of reading empty workouts. I can also make sure that my database won't be filled up with empty workouts.

Only when the conditions are met can the workout be created and pushed. If the conditions aren't met, then an alert dialog is shown:



```
if (workoutName == '' || activities.isEmpty) {
  showDialog<String>(
    context: context,
    builder: (BuildContext context) => AlertDialog(
      title: const Text('Error!'),
      content: const Text('Error creating workout. Make sure the
        actions: [
          TextButton(
            onPressed: () => Navigator.pop(context, 'Return'),
            child: const Text(
              'Return'),
          ),
        ],
      ),
    );
}

Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text(activity.activityName),
    ),
  );
}
```

Moving on from this, each activity object will be opened when the edit activity button is pressed. I have taken the activity name (set when adding the activity) from this object.

To recap: the activity class is a blueprint for an activity object; each activity object is added to the list of type Activity; when each activity card is pressed, it will open the index position of the list, therefore the activity object (which holds attributes).

This is shown in:

```
MaterialPageRoute(builder: (context) => editActivity(activity: widget.activities[index]),),
));
```

To improve code efficiency and to follow with “good code”, I have made sure to be accurate when naming functions and variables, as shown:

```
void deleteActivity(int index) {
void deleteActivityItem(int index) {
```

Each activity is designed to hold attributes. First of all I wanted to add a number of reps to each activity. This formed the starting point for how activities can be pushed to the database, so that they had meaningful fields. Once the text field was added to the editActivity page, the value of the reps attribute could be changed. This worked, but only within the editActivity page, and not outside. This was not an ideal solution, as the subtitle of each activity card should show the main attribute.

This is shown within the video: REPS_UPDATING_V1

```
// Access the current theme
ThemeData theme = Theme.of(context);

// Determine if the theme is light
bool isLightTheme = theme.brightness == Brightness.light;

// Set the color based on the theme
Color cardColor = isLightTheme ? Colors.white : Colors.grey[800];
```

Each activity card is given a set theme by the theme_constants values for each theme (light/dark). However, for these activity cards, I wanted a different colour scheme. To achieve this, I used a simple if statement to set the cardColor variable. If the theme is light, then the card colour will be set to white. This is a design choice which I feel gives a nicer, cleaner feel to the app. Depending on how many of this type of card I get, I may decide to make this style the default within the theme_constants file.

```
setState(() {
  widget.activities[index].updateReps(newReps);
});
```

The current problem was that the reps weren't being updated once they were changed within the *Edit activity* page (shown within the video: REPS_UPDATING_V1). This meant that each activity object within the activities list would not be properly updated. This was a state management issue, and was fixed by using the setState() function built into dart, along with keys. Keys are essentially just references that items within lists can hold, and it means that flutter knows how to change the details of each individual item.

The updated, partially fixed version of the reps updating is shown in the video:
REPS_UPDATING_V2

This does not yet update the reps within the edit activity page for some reason, but this issue has been added to the backlist of items which do not fulfill the 'functional app' plan.

Validation:

Each attribute within each activity item has its own type. For example, the number of reps will be an integer, the time on a timer will be dateTime, and the stopwatch will be a Boolean value (whether the activity should be a stopwatch event or not). The keyboard type for entering values into the attribute of the reps will be of type number. This means that only digits can be entered, and that decimal points, negatives or commas cannot be. This 'fool-proofs' the way that the user can enter reps.

```

keyboardType: TextInputType.number, //VALIDATION - shows only numpad
inputFormatters: [FilteringTextInputFormatter.digitsOnly], //VALIDATION - allows only integers to be entered
onSubmitted: (reps) {
  try {
    int parsedReps = int.parse(reps);
    setState(() {
      widget.activity.reps = parsedReps;
    });
  } catch (e) {
    // Handle the case where the input is not a valid integer
    print('Invalid input: $reps');
  }
}

```

Testing:

Test type	Method	Result	Response
Feature test – Creating real workout with expected data	1– Create page opened 2– New workout opened 3– Workout created 'Morning workout' 4– Activities added: Push-ups, Plank (2:00), Rest (30s), Plank (1:45), Stretches. 5- Workout saved	When more than one activity of the same name had been created, only the last activity was saved. This is because my database function set the activity data to a new document with the name of the activity	Activity IDs were implemented – initially an empty string – and were generated when the document was created. This means that activities never override other activities with the same name.
Feature test – Creating and deleting activities when editing workouts	1- Workout created containing activities 2- Edit workout opened 3- Created workout opened 4- Activity added 5- Different activity deleted 6- Workout saved	<p>When an activity was deleted, and the workout was saved, the database function would take the instance of the function, create new activities for each activity in the workout, and delete activities in activityNamesDeleted. This meant that workouts could not actually be deleted, since they'd just been added and had new IDs.</p> <p>When a new activity was added, an object was instantiated with an empty ID. The database function to</p>	<p>Fixed by moving the for-loop to delete activities to above the update activities statement. This meant that any activities that should've been deleted are deleted and therefore can't be created new.</p> <p>To fix this, I introduced an if statement to make the decision: if the path was null, create a new document for each activity; if the path contains the activityID because the activity is only being updated, it will</p>

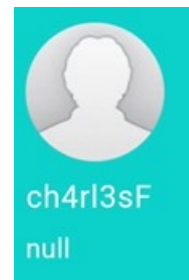
		update the workout would try and use the activity ID to map the activity data. This resulted an error because the ID was an empty string.	write to that path.
--	--	---	---------------------

More Authentication:

One feature that is common in most applications these days is detail handling – giving the ability to users to change their personal details such as email/username/password and then displaying these details.

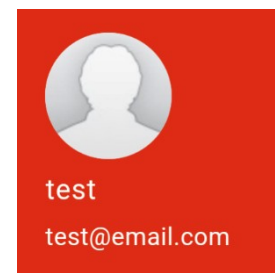
At this point, BurnBoss only takes in the users email and password when they register. This means that instead of showing a username, I can only show their email address.

This was an issue that was on my mind for a while, as the first way that I attempted to implement this was by taking an instance of the AuthService() class, and then taking the user object. However, this shows a horrible backend jumble, and does not show any useful information to the user. This is also shown when the user is a guest, at which point they do not have specific details saved to their account, therefore the value is null.



This issue was fixed by using the logic that if the user is signed in (which they must be to access the rest of the app), then I can access the currentUser class, and the email attribute within it. This mean I could take only this attribute and display it to the user. As is shown in the image above, the profile picture and username are static. These are things that aren't necessary to have a functional app, but I would like to implement, so they shall be implemented later.

```
var email = AuthService().user;
var email = FirebaseAuth.instance.currentUser!.email.toString();
```



Another issue within the Sign In and Register pages that I was notified of through feature testing was the Menu Drawer icon button. This should not be here, and was there as a result of me copying the code to style the appbar of these pages. This icon also would not work even when pressed, as the function is only called once the user is signed in.

Password Obscurity:

Because I am using my app a lot as I develop it, I find the need to sign in and sign out quite a lot, and so do those who test my app. I have received feedback that although when a password is entered, It shows a brief preview of each letter, this is not enough, as errors can be made in the password and can't be checked properly. This became an annoyance to users, so it was necessary for me to add in a password visibility switch.



```
icon: Icon(_passwordVisible ? Icons.visibility_off_rounded : Icons.visibility_rounded),
```

I did this by creating a boolean value for if the password visibility should be set to visible, or obscured. I then made the IconButton dependent on the value of the boolean `_passwordVisible`, and the obscurity of the text within the text field also dependant on the state of the boolean.

Once the IconButton was pressed, the state updated to switch the value (by using `!` as notation for NOT). Shown in video

`PASSWORD_VISIBILITY`

```
setState(() {  
  _passwordVisible = !_passwordVisible;  
});
```

was the

```
obscureText: _passwordVisible,
```

Stopwatch:

One of the features which I mentioned in section *1.1 Computational Amenability* was the stopwatch page. My intention for this page was to have a stopwatch, and a few timers. This was to reduce the effort that the users had to put in to complete a simple task, e.g. keeping track of how long they could plank. This is not something that my stakeholders would intend on creating an entire workout for, and would instead just be useful to have a small section within the app that would be for this purpose.

I started this section with the stopwatch. There is a timer function built into the flutter package, and I utilized this.

I needed three functions to complete the basics of a stopwatch:

Start:

```
//function will be called when the user presses the Start button  
void _start() {  
  //Timer.periodic() will call the callback function every 100 milliseconds  
  _timer = Timer.periodic(Duration(milliseconds: 30), (Timer t) {  
    //Update the UI  
    setState(() {  
      _result = '${_stopwatch.elapsed.inMinutes.toString().padLeft(2, '0')}:${(_stopwatch.elapsed.inSeconds % 60).toString().padLeft(2, '0')}:${(_stopwatch.elapsed.inMilliseconds % 100).toString().padLeft(2, '0')}';  
    });  
  });  
  _stopwatch.start();  
}
```

Stop:

```
//function will be called when the user presses the Pause button
void _pause() {
    _timer.cancel();
    _stopwatch.stop();
}
```

Reset:

```
//function will be called when the user presses the Reset button
void _reset() {
    _pause();
    _stopwatch.reset();

    //Update the UI
    setState(() {
        _result = '00:00:00';
    });
}
```

Testing:

Doing a group survey within my testing group, I have been given feedback on both the layout and the design of the buttons used to stop/pause/reset the stopwatch:

- Issue 1: The pause button should be renamed “Stop”. This is convention across stopwatched, and it makes no sense to switch it within my application.
- Issue 2: The Start and Stop buttons should be combined into a single Start/Stop button. This is also convention and although it is easier for me to have individual buttons for individual functions, the user experience within the app is a long-term effective way of keeping my app above the competition.

I went about this by defining a private boolean value `_isRunning`, and once the Start button is pressed, it will update this value and then execute the `toggleStartStop` function. This function will depend on the value of `_isRunning`, and will toggle what the stopwatch is doing.

```
//function to toggle start and stop of the stopwatch
void _toggleStartStop() {
    if (_stopwatch.isRunning) {
        _timer.cancel();
        _stopwatch.stop();
    } else {
        _timer = Timer.periodic(Duration(milliseconds: 30), (Timer t) {

            //Update the UI
            setState(() {
                _result = '${_stopwatch.elapsed.inMinutes.toString().padLeft(2,
0')}:${_stopwatch.elapsed.inSeconds % 60).toString().padLeft(2,
0')}:${_stopwatch.elapsed.inMilliseconds % 100).toString().padLeft(2, '0')}';
            });
        });
        _stopwatch.start();
    }
}
```

I had noticed that when I started and stopped the stopwatch quickly, the milliseconds weren't in order. For example, it was possible to go from 00:01:68 to 00:01:04, by pressing start, stop, start, stop. This defeats the point in the stopwatch, which is to increment linearly and count up in time. To mitigate this error (which was localised to the milliseconds), I reassessed the need to include milliseconds and decided that there was no need within real-life applications of BurnBoss to record down to the millisecond, so I could remove them from the stopwatch. This left me with hours:minutes:seconds, and the problem no longer persisted.

Testing:

I had found that due to the ability within mobile devices to 'go back', it was possible to go back to the home screen, and then create a new instance of the stopwatch. This meant it was possible to create a memory leak, and not dispose of the stopwatch until the application was closed.

To fix this, I needed to use a singleton – a software design pattern that restricts the instantiation of a class to a singular instance – to allow only one instance of the stopwatch to be created for the stopwatch page.

Selecting workouts:

At this point, my project was starting to come together. I could register, sign in, navigate and create workouts. Once writing data to the database was sorted, I needed to read data.

I started this within the Select page, as the features of this page would be:

- Display a list of cards for each workout
- Allow the user to click the play icon on each image, and start the selected workout.

These are simple objectives that allow me to work the functions that will be used throughout the rest of the app, such as getting a snapshot of the collection of workouts. The function used to get the workouts went through a few different versions:

Using StreamBuilder:

- On my first attempt, I used a StreamBuilder to provide a stream of data, so that if anything in the collection changed, it would send the update down the stream to whatever was listening. Within the Streambuilder, it would build the list of workouts using each name.
- This option did not work. This gave me a lot of trouble because even when viewing the Firebase documentation, I did not realise that the package function `.snapshots()` needed a field within each document. The original database design I had created did not need each workout to have a name, so it was not added.
- The method of creating the list of workouts within this function was inefficient, as the function should not get a collection of workouts, but instead a snapshot of workouts which can be manipulated further down the line

Using `.get()`:

- In a second attempt, using what I had learnt about needing a field in each workout, I mapped the field for a workout name to each workout.

- From this, I could use the `.get()` function to retrieve a snapshot of all of the workouts within the collection
- Then, for each workout, I could use a for loop to create each workout document, and then take the id for that document and use it within the path to retrieve the snapshot of activities within each workout, and then map each activity to a list of attributes.

```

Fetching workouts for UID: jmYmyLGLIYV2Y9kvQ4A7fo7x0q02
Successfully completed
documents recieved: 3
Fetching activities for workout: Monday
pirshupd => {reps: 508, activityName: pirshupd}
Fetching activities for workout: activity read check
activity read check activity 1 => {reps: 0, activityName: activity read check activity 1}
activity read check activity 2 => {reps: 0, activityName: activity read check activity 2}

```

```

// Extract workout names
List workoutNames = snapshot.data!.map((workout) =>
workout[workout.id]).toList();

// Use a ListView.builder to create cards for each workout
name

return ListView.builder(
  itemCount: workoutNames.length,
  itemBuilder: (context, index) {
    String workoutName = workoutNames[index];

    // Create a card for each workout name
    return Card(
      child: ListTile(
        title: Text(workoutName),

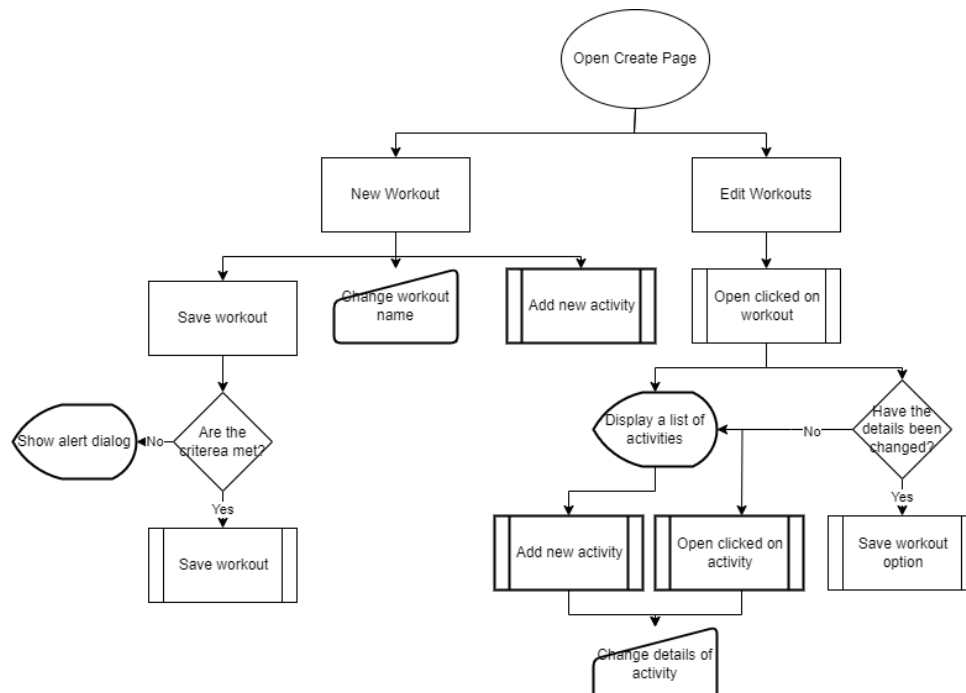
```

The 'Play' button is added to each workout card. At this point, I have not implemented the Play functionality, so this button does not do anything. This is shown in the video [SELECT_PAGE_V1](#).

Editing Workouts:

Now that the function `getAllWorkouts()` has been created, and outputs a snapshot of the workouts and all activities within, I can create the Edit workouts page.

When looking back at my design, within the *Creator* flowchat, I have decided to change some functionality. Within *2.5 Interface Designs*, I decided that the creator page should be set out in a way which shows users who may have low IT literacy the steps that they must take to both edit and create workouts. I have provided a new flowchart showcasing how other stakeholders and I who have tested this section would prefer this interface to look:



While testing this feature of reading the workouts and mapping their data to activities and then displaying this data, I found an issue. The base of these activities was that when mapped to the document for each activity, they were sorted alphabetically. This was the opposite of what my aim was, as when people create workouts they must be able to create them and retrieve them in the order that they made them to be in.

The way that I fixed this issue was for each activity, to add a position field which was their position in the list that they are stored in:

```

for (int i = 0; i < workout.activities.length; i++) {
    CollectionReference activitiesCollection =
        workoutDocument.collection('activities');
    Activity activity = workout.activities[i];
    // Use the position in the list as the ordering criteria
    Map<String, dynamic> activityData = {...activity.toMap(), 'position': i};
    await activitiesCollection.doc(activity.activityName).set(activityData);
}
  
```

Once written, I can retrieve each activity and order them by their position. Shown in EDIT_PAGE_V1.

The next feature within the Editing workouts page is to be able to delete workouts. This is done by adding an IconButton to each workouts, and calling the deleteWorkouts() function. This function originally took in the workout name as a parameter, and used it within the path to delete the workout.

However, after creating this method and observing the data in my database, I noticed that it wasn't deleting the workout or the data within the workout. This is because the documentation says that within flutter and using firebase it is bad practice to use functions to delete subcollections as it takes a lot of reads, and a lot of memory within their servers. This means that this functionality is completely removed.

To get around this, I took a snapshot of the activities within the workout, and for each activity, delete it. This meant that once this was done, I could delete the workout.

Now that workouts could be both accessed and deleted within the Edit Workouts page, it was time to work on each workout and its details. As shown above, the activities can be retrieved in the order that they were created within. The next step was to edit these activities themselves. Just as within the Create Workout page, I needed to be able to edit each activity, and so when using the ListBuilder, I could take each activity object and edit them from their index within the list of objects. From this, I can open the same editActivity class which is used before (which meant I did not have to rewrite the same code). This is shown in EDIT_PAGE_V2.

However, due to the design of the New Workout page (which used a single save button independent of each activity), I had no way to save any details which would be changed for each activity. This means that nothing was pushed to the database. To solve this issue, I needed a "Save changes" button which appears only when a detail has been edited. This save button pivoted on a boolean value which could be toggled whenever details needed to be changed, and toggled back once the saveChanges function was performed. Initially, to save the changes made to edit the workout, all I did was recreate the workout which would overwrite the activities. This was not good practice, and will be redone later due to the issues which it caused.

```
void saveChanges() {  
  print('Changes made');  
  setState(() {  
    changesMade = false;  
    DatabaseService(uid: FirebaseAuth.instance.currentUser!.uid)  
      .createWorkout(widget.workout);  
  });  
}
```

Along with this, I needed an IconButton to delete each activity. When pressed, it deleted the activity object at whichever index it is in within the activities list.

For activities, it would be possible to delete multiple activities for a single workout. To do this, I decided that I needed a list of the activities which were to be deleted, and a function to take each activity name within the list of activities, and delete it and its fields.

```
DatabaseService(uid: FirebaseAuth.instance.currentUser!.uid)  
  .deleteActivity(widget.workout.workoutName, activityNamesDeleted);  
  
setState(() {  
  widget.workout.activities.removeAt(index);  
  changesMade = true;  
  activityNamesDeleted.add(activity.activityName);  
});
```

```

Future deleteActivity(String workoutName, List activityNames) async {
  for(var activity in activityNames) {
    WorkoutsCollection.doc(workoutName).collection('activities').doc(activity).delete();
  }
}

```

User test:

Test Purpose	User Type	User Response	My Response
To test the architecture of the app, and the ease of use. Feature test of the New Workout page.	High IT literacy	<p>“When a new workout is being created, the main function is to add activities and edit their features, so the Overview Tab should be the first tab, and the one which should be open originally.</p> <p>Also, once the workout is saved, it should exit back out of that workout as I am finished working on that workout”</p>	<p>I have swapped around which tab within the New Workout is first and therefore opened first.</p> <p>Within the Save Workout function in the new workout, a navigation to the creator page is pushed.</p>

Test Purpose	User Type	User Response	My Response
Feature Test – to test color theme. Options shown in <i>CHANGING_THEME_COLORS</i>	Range of IT literacies	<p>“The bolder, more striking orange makes me feel more motivated, fresh, and in the mood to do exercise. The duller orange is more relaxing”</p>	<p>I kept the colour scheme as originally intended.</p>

Back to editing workouts:

Now that the functionality for both editing and deleting workouts is integrated, the next step was to add the ability to add activities to workouts. I did this with a text field which only appeared when the add button was pressed. This needed to be a smooth, seamless experience which didn't hinder or bore the user by opening new pages or doing anything unexpected. One of these such issues was that when an activity was created, the name for the activity was not cleared – the fix for this was to clear the controller once the submit IconButton was pressed. The main functionality was to add activities. This meant I needed to add an instance of the Activity class to the list of activities held by the Workout class. Once this was done, it was wrapped in a setState to make sure that the ListBuilder which handled building the list of activities was awoken, and could append the new activity.

The next feature and final feature of editing workouts is to change the workout name. There was two parts to the development of this feature: design and database updating. For the design, I wanted the same features as adding activities, with an iconButton which changed the stage of the page to show the workout editor. This was simple enough to implement, with a boolean value which changed if the user was editing the title, and the code shown dependent on the value. The design of this function may require some tweaking later if user feedback results in issues with changing names.

To perform the functionality to change the workout name ended up in some issues. The previous way of updating the workout with new activities was to re-create the workout with the same name (since it would not change up to this point) which would overwrite the data within the database. This would no longer work, since the workout name would change so when the workout was created, it would create a new workout, and I would have to delete the old workout. This was very inefficient, and I decided to restructure this area of my database, in order to remove the dependency on workout names. To do this I added a new field to the workout, workoutID which would be automatically generated uniquely to create a primary key for the workout. This meant that changing the workout name only needed to update a single field within the database, which

resulted in less writes which made the app more efficient (no longer waiting to re-create the entire workout). When the workout name had been changed, and the user exited the workout, the

```
//function to edit workout name
Future editWorkoutName(String workoutID, String workoutName) async {
  return WorkoutsCollection.doc(workoutID)
    .update({'workoutName': workoutName}).then(
      (value) => print("DocumentSnapshot successfully updated!"),
      onError: (e) => print("Error updating document $e"));
}
```

new name would not have changed (shown in EDIT_WORKOUT_NAME_NO_REFRESH). This is because the navigation route is set to pop the last page off a stack of pages opened, this means that it returns to the last opened page. The problem with this is that it did not rebuild this page, which left the name unchanged. One method of fixing this would be to add a stream to rebuild this widget once a change has been made – this was not the method I went with, since I was unfamiliar with streams. The other way to do this was to add a refresh pull on the page, which would rebuild the list and therefore take another snapshot of the data, updating the names. This is shown in EDIT_WORKOUT_NAME_WITH_REFRESH.

User tests:

I was at the point at which the editing page was completed (excluding editing activities), and it was important that I tested the feature with stakeholders. My testing group consisted of 2 low IT-literacy users who both use apps like fitbit to stay in shape. I presented the app to them with no tutorials or help within the app.

Test Purpose	User Type	User Respose	My response
Feature test - Creating workouts. Asked to "create a workout with	Low IT-Literacy, within the older age group	Swap the select and create card in the home page (ease of first use). Slight struggle when naming the workout. Suggested a tutorial would be	Would only be useful for the first use, once workouts are present it will be quicker to go straight to select I may need to change

the name 'Saturday workout'.		useful	the design to set the name. Also, an auto-generated name may be useful
Feature test – Editing workouts. Asked to edit the name of the workout they previously made. Given the app on the home screen.	Same group	Instinctively went to the select page in order to select a workout to edit it. It was obvious how to change the name of the workout, but the name not changing when going back to the 'Edit workouts' page was confusing – the pull-to-refresh was not obvious to the Low IT-Literacy group (shown within CHANGING_WORKOUT_NAME)	Add in an edit button to each workout within the select page, next to the play page. Maybe show a hint of the pull to refresh.

Activity Types:

When designing the layout for workouts, I knew that I needed different types of activities. The majority of activities I researched could be sorted into three types:

Reps – A set number of actions, e.g. 5 reps of bench press at a weight of 20kg.

Timer – A count-down for a set amount of time to complete an action in, e.g. 2 minutes of plank.

Stopwatch – A count-up stopwatch to record how long an action is taken for, e.g. how quickly a 50m swim can be done.

Each activity will have a type (added as a parameter of the activity class, and as a field to the activity database document), which is set default to Reps, as it was the most common type of activity I have found. To change the activity type would be a function performed for each activity, and so a drop-down menu (shown within ACTIVITY_TYPES) was added to the editActivity page. When the activity type was changed, the argument was passed into the updateActivityType function. Originally, I had planned on resetting all other variables to 0 when not in use, however the activityType attribute of the activity class meant I did not have to go through with this, and I can leave any other variables in whatever state they are in, as they will not be shown when the workout is performed. For each activity type, a different screen should be shown:

For *Reps*:

Needs input for the amount of reps

Needs a toggle switch for weights

This will update the boolean attribute weightUsed

If switched to 'No', then the integer weights attribute will be set to 0

If switched to 'Yes', then the text field for inputting weights will be shown. In order to reduce error, and as a form of validation, I have limited the keyboard to be only positive numbers with only 2 decimal places – this is to allow weights such as 2.5kg or 1.25kg, but avoid errors with inputted words or too many decimal places.

For *Timer*:

Needs a scrollable time input with Hours, Minutes and Seconds. I have used the Cupertino ScrollableTimePicker, which limits a maximum of 23Hr, 59Min, 59Sec. Shown in TIME_SELECT.

This input needs to be converted to milliseconds to be stored in the database, and then converted back to a Duration when the snapshot is read from the database.

For *Stopwatch*:

Needs a toggle switch, with the default option for the user to use a stopwatch

For validation, if the option take is to NOT use the stopwatch, then the activity type will be set to Reps

```
if (activity.activityType == 'Reps')
  Padding(
    padding: const EdgeInsets.symmetric(vertical: 30, horizontal: 30),
    child: FittedBox(
      fit: BoxFit.fitWidth,
      child: Text(
        'Reps: ${activity.reps}',
        style: TextStyle(fontFamily: 'Bebas', fontSize: 50),
      )),
  ),
if (activity.activityType == 'Timer')
  Padding(
    padding: const EdgeInsets.symmetric(vertical: 30, horizontal: 30),
    child: Text('timer')
  ),
if (activity.activityType == 'Stopwatch')
  Padding(
    padding: const EdgeInsets.symmetric(vertical: 30, horizontal: 30),
    child: Text('stopwatch')
  ),
```

Playing workouts:

When designing the workout player, and after research into different types of 'progression through pages', I chose to use PageView, which allows me to build a new page for each object in the list of activity objects, taken from the instance of the workout. The features of the workout player are:

- Progression through pages – switching between pages

- Each activity shown dependent on their type, along with all details of the activity

PageView builds a new page for each object within the list, in my case I needed a new page for each activity object within the workout. Each button (to switch between pages) has a range: the Previous button shouldn't be available when on the first page, the Next button should go up to the penultimate page, the Finish button should be shown on the last activity, and the Exit

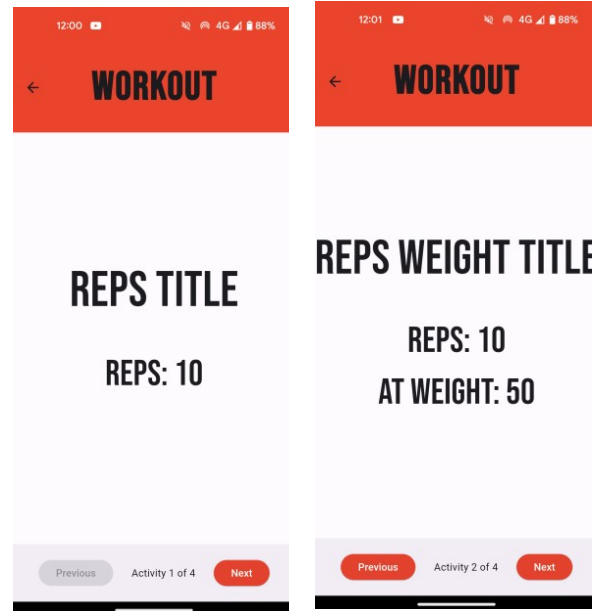
button should be shown on the last page. In order to keep track of this, whenever the page changed, the value of the current page was updated:

```
void initState() {  
  _currentPage = widget.workout.pageProgress;  
  _pageController = PageController(initialPage: _currentPage);  
  super.initState();  
}  
  
onPageChanged: (int page) {  
  setState(() {  
    _currentPage = page;  
  });  
},
```

Each page depended on the activity within, and its attribute value for activityType. In order to maintain modular code, a widget called buildActivityPage was created, and shows different things depending on the activity type:

One of the user-oriented design choices made was to maintain the progression through workouts, if they were exited mid-way through. This meant that the workout class needed a new attribute, initially zero, which held the value of the current page. A new method was created to update the page progress in both the class and the database file. This also meant that when the workout was played, it should open to the page saved to the database. I had added this attribute to the workout class and had initially set it to 0 when the class was instantiated, but when reading from the database I had not remembered to map the workout object with this pageProgress attribute, this meant that it did not open up on the correct page. This was fixed by mapping the attribute to the retrieved object, which meant I could set the initialPage value within the initState function to the pageProgress attribute of the object:

It is important to both use asynchronous functions and to provide catch statements when working with databases in order to remove any risk of the app either not waiting long enough to perform a function, or waiting too long to never receive an answer.



```
//function to update the current workout progress
Future updateWorkoutProgress(String workoutID, int newPageProgress) async {
  return WorkoutsCollection.doc(workoutID).update({'pageProgress': newPageProgress}).then(
    (value) => print('DocumentSnapshot successfully updated with new page progress'),
    onError: (e) => print('Error updating document $e')
  );
}
```

In my original plan, I had not intended to create a finish page, but when testing the page progression, I saw that even when the workout was exited on the last activity, the percentage value was not 100%. This is because there is no way to know if the last activity has been completed before the workout was exited. This is why a finish page was introduced, so that all activities could be progressed through.

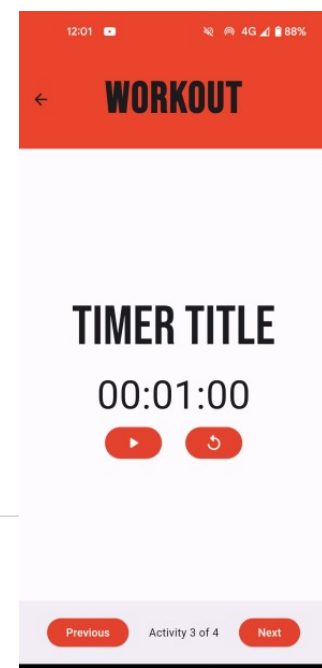
This is shown in the video [WORKOUT_PAGE_VIEW](#).

Activity Pages:

The aim of the workout player is to display the details of each activity to the user. It is important that this area is well designed, since it is the main focus of the app, and the area in which the user will spend the most of their time.

Reps:

The details to be shown within the Reps page are: the title of the activity, the number of reps, and if used, the weight. The main focus is the activity title, followed by the number of reps. The font is sized



accordingly. I think that this page needs to be designed differently, as at the moment it doesn't catch the user's attention.

Timer:

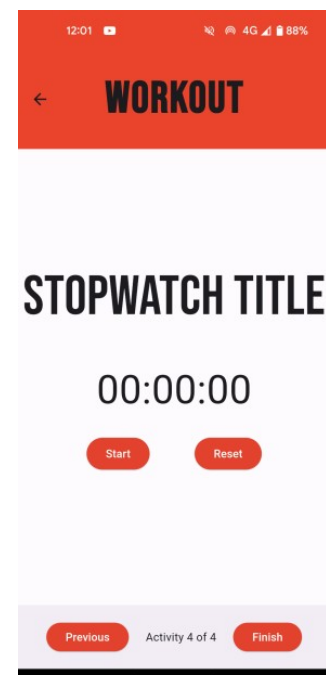
Initially, I had intended on using a flutter package to give me a well-designed, animated timer which catches the users attention and retains interest in the workout and application, however the initial package `circular_countdown_timer` was depreciated and could not be used with the version of flutter I am using in my app. The second package I tried to use was `flutter_timer_countdown`, which performed the functions of a basic timer – counting down from a set duration – but did not include any ability to pause or reset the timer. These are functions which allow ease of use for the users of my app, and are necessary components within the design of this activity type.

This meant that I needed to code my own solution for the Timer. In order to maintain modular code, I needed a class for the timer. The functions and methods for the timer are to start, to pause, to resume, to reset, and to display the timer. Once this class had been developed, I could call it each time an activity uses the timer. Originally, I had used this function within the widget build, passing in the index of each object of an activity from the activities list. This had worked, it would build the timer whenever the timer was needed, but it meant that it was a single instance of the timer for all activities where the timer was required. This meant that when, for example, a start button was pressed in an activity, it would affect all other timers. The solution to this was to use keys to separate each widget. Keys are used for modifying a collection of widgets of the same type which hold a state – they are a way of keeping a reference to state, and state at different times or maintain it while modifying the widget tree. I could use flutter's `GlobalKey`, assigned to each instance of the timer. This kept each timer separate.

Another error which I had encountered was that when timer was called, the duration had not been set. This meant that it would try on functions which had not yet been started. To fix this, I created a new method of the timer class `_timerCallBack`, which set the state of the duration. A basic version of the timer is shown in `BASIC_COUNTDOWN_TIMER`

Stopwatch:

Since I had created a class for the timer, I could replicate this for the stopwatch. Since there was no need for a callback, that method could be removed.



3.8 Bug fixes:

1. Navigation drawer

When pages were open, it was possible to re-open them (build a new page on top of the existing page) which allowed for issues such as creating multiple instances of the stopwatch.

To fix this, I added two new variables into the `NavBarWidget` class:

currentPage – this is a required parameter of the class which is passed in from any page which opened the navigation drawer. This value is checked against the current path, and determines the value of isSelected.

```
buildNavBarItem(  
  label: 'Home',  
  featureIcon: Icons.home_filled,  
  isSelected: currentRoute == '/',  
  action: () {  
    Navigator.pushNamed(context, '/');  
  }  
)  
  
leading: Icon(  
  featureIcon,  
  size: 30,  
  color: isSelected? Colors.grey : null,  
) // Icon
```

isSelected – holds the value of if the page that the navigation drawer is the same as the item. The colour and function of the item are both changed if isSelected is true.

2. Activity deleting

It was possible to delete all activities within a workout, and then when the activity was played, it would break both the progress percentage, and the workout would have nothing in it. This should not be possible, so I included a variable numberOfActivities which was set in the initState() to be the length of the workout. When an activity was removed, numberOfActivities was reduced in increments of 1. It would then only show the option to delete an activity if the number of activities was greater than one.

3. Persistent theme

Since the user can change the theme of the application, it is their decision on which theme they prefer. If they choose to change the theme, then their decision should be permanent. To achieve this, I needed database functions to both write to, and read from the database. When the app is initially run, the theme should be dependent on the value from the database. To do this, the theme value should be initialised when the app is run.

```
ThemeManager() {  
  initializeTheme();  
}  
  
Future<void> initializeTheme() async {  
  bool? isLightTheme = await DatabaseService(uid: FirebaseAuth.instance.currentUser!.uid).getTheme();  
  if (isLightTheme != null) {  
    _themeMode = isLightTheme ? ThemeMode.light : ThemeMode.dark;  
  }  
}
```

4. Testing

4.1 Pre-Development Testing

When initially developing my app through basic designs, I created various prototypes. Each prototype was analyzed, with the areas which functioned as intended to be kept,

and the areas which needed improvement were abandoned or worked on. I maintained this attitude throughout the development of my application, and iteratively tested, fixing errors as they were found. It was important to refine sections of my app as much as possible, to keep a clean codebase, and a sleek design. As shown in 3.2 *BurnBoss*, I have performed different types of tests throughout, such as feature tests using small-test groups. It was important to keep these test groups small, to give personal, unique responses, with a quick return time.

4.2 Post-Development Testing

To complete section 4. *Testing*, I have tested functionality, robustness, and usability. I have initially completed these tests myself, and then brought in a sample test group of stakeholders, outlined within 1. *Analysis*. To maintain quality within my test data, it is important to attempt to reach the right demographic, with unbiased and thought-out responses.

4.2.1 Testing for Usability

Due to developing my application, I have an understanding of how my application and all of the features work. It was important for me to see how the application is viewed by others. Small sample groups were used to test *BurnBoss* in real situations and daily life.

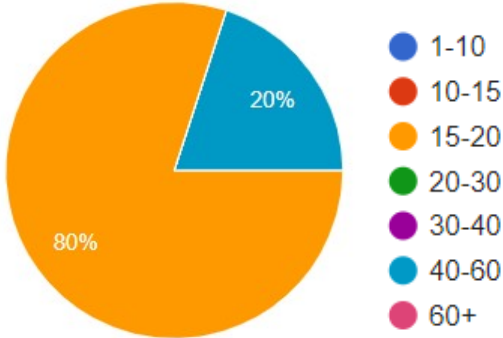
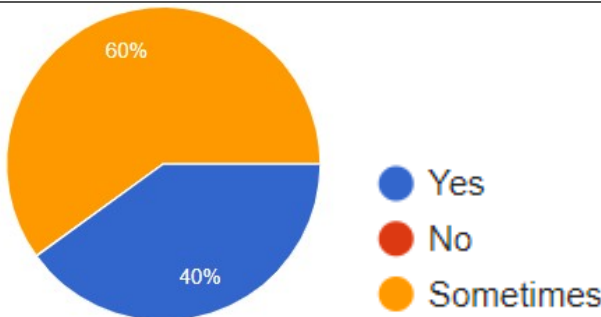
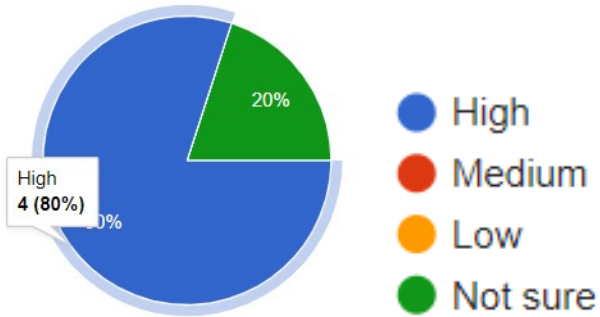
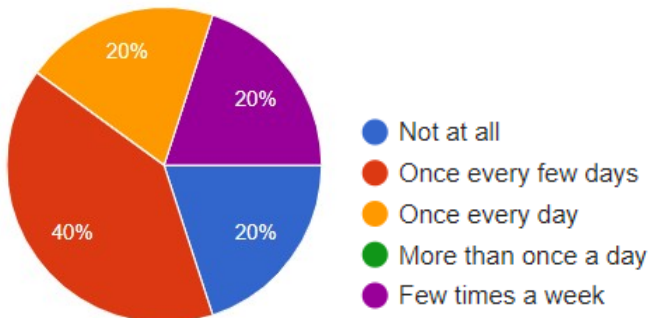
Each participant was chosen based on how they represent the app's intended user audience. This is to give accurate responses to how it will be used, and how the functions of the app work within the situations they are intended for.

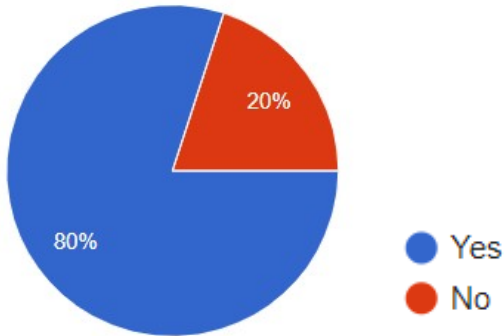
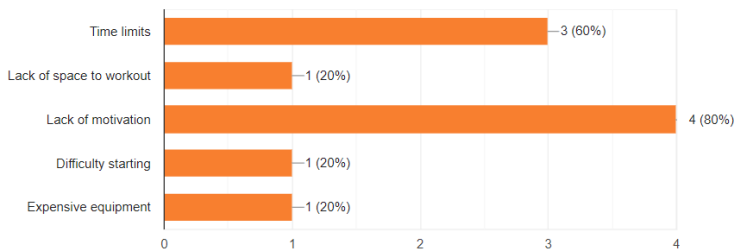
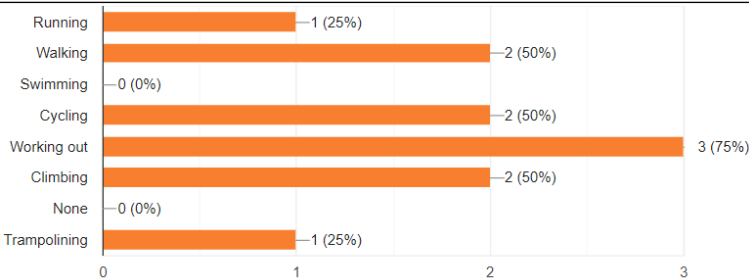
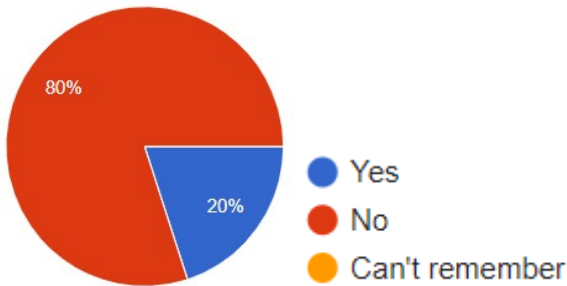
Research Testing:

In order to test how the user interacts with my application, I need an initial set of control data. This helps me to evaluate how my application will be used if it were to be released to the public by looking at how useful it would be to a sample group.

These are the responses to the form:

Test Question	Responses	What does this tell me?
---------------	-----------	-------------------------

How old are you?	 <p>80% 20%</p> <ul style="list-style-type: none"> 1-10 10-15 15-20 20-30 30-40 40-60 60+ 	The userbase of my application is primarily young, which tells me that although I should maintain a simple application interface, I can afford to include more features which require a knowledge of mobile use and skills.
Do you eat healthily?	 <p>60% 40%</p> <ul style="list-style-type: none"> Yes No Sometimes 	A lot of the users try to maintain a healthy lifestyle, as shown by a healthy diet. This tells me that the users should not have difficulty maintaining an attitude focused towards regular exercise (and therefore regular use of my app).
How would you describe your IT literacy?	 <p>High 4 (80%) 20%</p> <ul style="list-style-type: none"> High Medium Low Not sure 	Most of the userbase of my app has a high IT literacy, which means I can afford to include features which may not be as intuitive, but will provide more functionality. I think my application meets this criteria, as creating workouts could be daunting to new users, but allows full functionality
How often do you work out/exercise per week?	 <p>20% 20% 20% 40%</p> <ul style="list-style-type: none"> Not at all Once every few days Once every day More than once a day Few times a week 	This shows a wide range of different responses – some users tend to work out more often than others. It is important for me to focus on both ends of this spectrum, catering for both those who workout intensely, and those who need more assistance in motivation to workout.

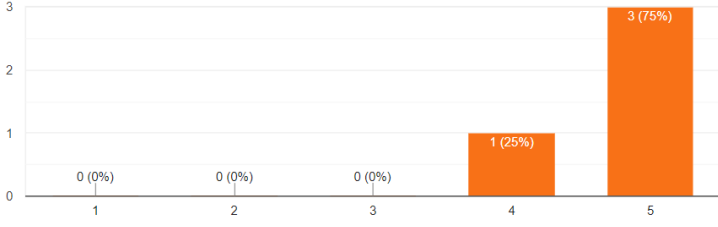
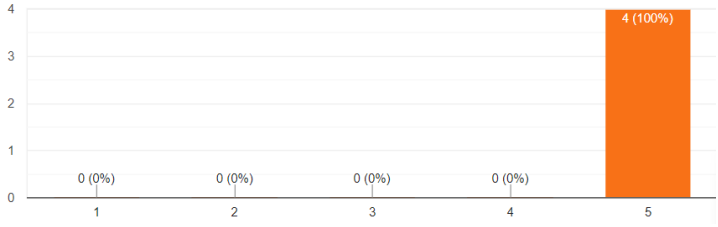
Would you like to exercise more?	 <p>A pie chart with a blue section representing 'Yes' at 80% and a red section representing 'No' at 20%. A legend to the right shows a blue circle for 'Yes' and a red circle for 'No'.</p>	This shows evidence for the point made in the above question response.																											
What challenges do you normally face when going to workout?	 <p>A horizontal bar chart with orange bars. The x-axis is labeled from 0 to 4. The y-axis lists challenges. The data is as follows:</p> <table border="1"> <thead> <tr> <th>Challenge</th> <th>Count</th> <th>Percentage</th> </tr> </thead> <tbody> <tr> <td>Time limits</td> <td>3</td> <td>60%</td> </tr> <tr> <td>Lack of space to workout</td> <td>1</td> <td>20%</td> </tr> <tr> <td>Lack of motivation</td> <td>4</td> <td>80%</td> </tr> <tr> <td>Difficulty starting</td> <td>1</td> <td>20%</td> </tr> <tr> <td>Expensive equipment</td> <td>1</td> <td>20%</td> </tr> </tbody> </table>	Challenge	Count	Percentage	Time limits	3	60%	Lack of space to workout	1	20%	Lack of motivation	4	80%	Difficulty starting	1	20%	Expensive equipment	1	20%	<p><i>Time limits:</i> My application must be easy to use, and quick to navigate and get started. Nobody works out just to get to use the app, so BurnBoss should be efficient.</p> <p><i>Lack of motivation:</i> I should avoid any barriers to users working out, for example I should try to remove areas which would cause hassle (in features such as creating workouts).</p>									
Challenge	Count	Percentage																											
Time limits	3	60%																											
Lack of space to workout	1	20%																											
Lack of motivation	4	80%																											
Difficulty starting	1	20%																											
Expensive equipment	1	20%																											
What types of exercise do you do?	 <p>A horizontal bar chart with orange bars. The x-axis is labeled from 0 to 3. The y-axis lists exercise types. The data is as follows:</p> <table border="1"> <thead> <tr> <th>Exercise Type</th> <th>Count</th> <th>Percentage</th> </tr> </thead> <tbody> <tr> <td>Running</td> <td>1</td> <td>25%</td> </tr> <tr> <td>Walking</td> <td>2</td> <td>50%</td> </tr> <tr> <td>Swimming</td> <td>0</td> <td>0%</td> </tr> <tr> <td>Cycling</td> <td>2</td> <td>50%</td> </tr> <tr> <td>Working out</td> <td>3</td> <td>75%</td> </tr> <tr> <td>Climbing</td> <td>2</td> <td>50%</td> </tr> <tr> <td>None</td> <td>0</td> <td>0%</td> </tr> <tr> <td>Trampolining</td> <td>1</td> <td>25%</td> </tr> </tbody> </table>	Exercise Type	Count	Percentage	Running	1	25%	Walking	2	50%	Swimming	0	0%	Cycling	2	50%	Working out	3	75%	Climbing	2	50%	None	0	0%	Trampolining	1	25%	I should focus on developing my app to be primarily used when working out (e.g. weights training, gym workouts), but I should also allow users to customize the app to work for other types, such as climbing or walking.
Exercise Type	Count	Percentage																											
Running	1	25%																											
Walking	2	50%																											
Swimming	0	0%																											
Cycling	2	50%																											
Working out	3	75%																											
Climbing	2	50%																											
None	0	0%																											
Trampolining	1	25%																											
Have you ever used any workout apps before?	 <p>A pie chart with a blue section for 'Yes' at 20%, a red section for 'No' at 80%, and a yellow section for 'Can't remember' at 0%. A legend to the right shows a blue circle for 'Yes', a red circle for 'No', and a yellow circle for 'Can't remember'.</p>	I should aim to set a high standard for workout apps, and I should provide as much functionality as I can in order to lose any users interest in my app.																											
Which apps have you used?	Garmin, komoot, strava																												

Which features of previous apps did you find useful?	Activity logging, rep counts	These are features which I had planned to implement in future versions. I had intended to create a calendar which would allow workout logging, to track how the users workouts have progressed or changed.
Are there any features you felt were missing from other workout apps?	Ease of use - they all try to do too much and lose the simplicity	I should focus BurnBoss on ease of use, in order give the user exactly what they expect from my app.

Post-use testing:

Once the user has experienced my application, I sent out a form to see the opinions and suggestions given to be by the test group I used. This allows me to see which improvements should be made to BurnBoss, if I carry on development.

These are the responses:

Test Question	Response	What went well/what can I improve?
What do you think about the design of BurnBoss?	 <p>Bad Good</p>	The design is satisfactory for the users.
What do you think about the navigation of BurnBoss?	 <p>Bad Good</p>	The navigation works well in BurnBoss. This should stop any users from losing motivation due to a difficult navigation/app.
Are there any features that you would change about the interface?	<p><i>Response 1:</i></p> <p>Make the go button do what select does, or at least play the most recent workout. Makes more sense to me if I want to do a workout to hit "Go" than "Select".</p> <p>Make the words in the side menu and main menu</p>	<p><i>Response 1:</i></p> <p>If I carried on developing BurnBoss, I could add the following changes quickly:</p>

	<p>consistent (I.e. "Create" vs "Creator") Fix issue with guest usernames in side menu Automatically display the edit menu when adding an activity to a workout, or have it be part of the adding an activity (Make it all one menu) to make it more obvious and cohesive.</p> <p><i>Response 2:</i> More options than reps as the app develops</p>	<p>Play the most recent workout from the "Go" button.</p> <p>Change 'Creator' to 'Create'</p> <p>Open the edit activity page when an activity is initially created.</p> <p><i>Response 2:</i></p>
--	---	---

		I should add better visibility for the other activity types, as they were missed when this user tested BurnBoss
Are there any features which you found particularly useful?	<p><i>Response 1:</i> The buttons were the perfect size for selection. Dark mode was also an attractive feature as well as the ability to view my password when signing in.</p> <p><i>Response 2:</i> Clear activity lost</p> <p><i>Response 3:</i> Really straightforward and easy to use, no clutter</p>	This shows that the interface and design steps taken to improve the user experience were satisfactory for each user.
Are there any things you would like to include?	<p><i>Response 1:</i> Go button as stated above. Ability to set reps achieved and display previous best, for attempting improvement/personal best. Same for weights. Rather than having to edit the reps/weight for the workout each time. Could be a different mode.</p> <p><i>Response 2:</i> The stopwatch feature is simple to use but could have included a lap counter of some kind to record interval distance times. The calendar feature would have been good to record different workouts.</p> <p><i>Response 3:</i> Distance tracking</p> <p><i>Response 4:</i> Graph over time? Maybe calendar dots or symbols showing which days I've done what workouts</p>	<p><i>Response 1:</i> I could include a new feature which would allow the user to input the number of reps/weight achieved in an activity, which could be shown in the next play through of the workout. This would allow the user to improve and progress with their training.</p> <p><i>Response 2:</i> I think it would be worth it to add in the lap feature, in order to expand the app to be used in other areas of exercise such as running or swimming.</p> <p><i>Response 3:</i> Distance tracking is a feature that I don't see myself adding to BurnBoss. It would require location use and calculations. However, I could add in an input field for distance.</p>

After using BurnBoss, how much do you think it has increased your motivation to workout?	<p>Bad Good</p>	The focus of BurnBoss is to be a simple, customizable application, which doesn't hinder users while they are working out. The main aim isn't to increase motivation to workout, but allow and provide for an easy workout. As is shown, this is an attractive feature to some users.
Would you use BurnBoss consistently?	<p>● Yes ● No ● Maybe</p>	It is shown that it may be useful to increase some user feedback/satisfaction to draw users back to the app – features such as a reward system, or a gamified algorithm.
Did you find any bugs while using the application?	<p><i>Response 1:</i></p> <p>It appears to have fixed itself but I could record the weight in KG. It has since started recording changes to weights. I an hit the select button.</p>	When this user was given access to the user, they were given the wrong version of the application. When they were given the right version, this problem was fixed.

4.2.2 Testing for Functionality

The main testing technique I will use is functional acceptance testing, in which I give my application to a sample group of users and ask them to perform a function (e.g. Create a workout with the name “Monday strength training”), without the use of any tutorial given by me previously. This should test both the architecture of my application, and the ease of use of each function.

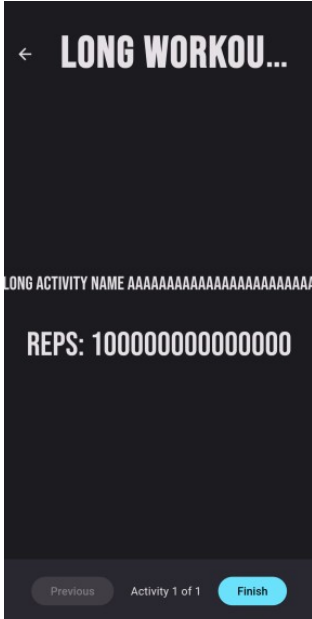
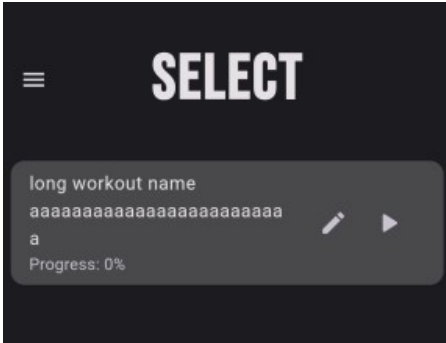
Test case and expected outcomes	Result of tests
Authentication: The user should be able to: Register Sign In Sign Out Sign in as Guest	<p><i>Registering:</i></p> <p>Suggestions –</p> <ul style="list-style-type: none"> - Sign in with google account - Email clears when incorrect, it should be editable to fix mistakes. <p><i>Signing in:</i></p>

	<p>Everything worked as expected, apart from the same issue as shown on the register page.</p> <p><i>Signing out:</i> Everything worked as intended.</p> <p><i>Signing in anonymously:</i> Suggestions –</p> <ul style="list-style-type: none"> - Have either a username or say “guest account” instead of null, in the navigation drawer.
<p>Creating a new workout: The user should be able to: Name a workout Add activities Edit activities Save the workout</p>	<p><i>Naming the workout:</i> It worked as expected.</p> <p><i>Adding activities:</i> Bugs –</p> <ul style="list-style-type: none"> - The weight didn't persist. It was added and then didn't save, or it didn't show when being read from database. <p><i>Editing activities:</i> Suggestions –</p> <ul style="list-style-type: none"> - Change the name of the activity <p><i>Saving the workout:</i> Worked as intended.</p>
<p>Editing a workout: The user should be able to: Open workout Editing workout name Editing activities Saving the workout</p>	<p>Everything worked as expected, apart from the weights which had the same issue as before.</p>
<p>Playing a workout: Opening workouts Playing workouts Use activities: Reps Timer Stopwatch Exiting the workout</p>	<p>Each feature worked as intended, and no bugs or errors could be found. I personally think that the design of the workout player could do with some work, as it feels incomplete.</p>
<p>Using the stopwatch: Using the stopwatch</p>	<p>The stopwatch only keeps track of full seconds. If a user presses start/stop in less than a second, it will keep it at 0 seconds, even if for example 3 0.2s intervals have been set. This is not an issue</p>

	currently, as users working out in the intended format should not need less than a second of accuracy.
--	--

4.2.3 Testing for Robustness

In order to test my application for robustness and error handling, I conducted various destructive and handling tests. The aim of these tests was to catch any errors which would become obstructions to the end user.

Test	Result	Follow-up/Improvements
Entering a long name for a workout, and checking throughout the application for how it handles the long name	 	<p>The workout name is cut off when in the player. It does not change font size depending on the length of the name. Currently, I do not plan on changing this, but if user feedback shows a different opinion then I will take it into consideration.</p> <p>Within the select page, the card extends to fit the entire length of the name.</p> <p>As a design choice, I may impose a character limit on the name of a workout.</p>
Entering a long activity name.		<p>The activity name fills the width of the page, without starting any new lines. This means that the entire name is shown (which is the important data), but it is badly implemented.</p>
Entering a large amount of reps. Editing the digit length until the fetch function no longer returns an object		<p>I had noticed that when too many digits of the reps was added to the activity, the workout would no longer show in the list of workouts. I expect that when the digit length hits the limit of Dart's integer range (-2^{53} to 2^{53}), the fetch function cannot return a complete workout object.</p> <p>This is not necessarily an issue, as it is unlikely a user would want to, for example, perform 9 quadrillion push ups, however I think a limit should be added to the number of reps to be inputted.</p>
Doing the same test as		<p>The same thing happens with weight, as to reps.</p>

with reps.		
Authentication validation		<p>The emails are currently non-validated, and it is possible to enter and use fake emails, as long as they fit the input criteria.</p> <p>There is no limit to the amount of guest accounts a user can create. This could database flooding issues. This could be solved by assigning each MAC address a</p>

5. Evaluation

In this section, I will evaluate how my application has met the initial idea I had worked off, and how it solves the problem which I intended to solve. I would like this to be a useful application which helps my stakeholders in their daily lives. I will now evaluate if it meets these criteria.

5.1 Success Criteria

I intend to show how I have met each of the success criteria, or if there is room for improvement.

1. The ability to create workouts:

Expectation: The user should be able to design and create their own workouts.

Criteria:

Name the workout
 Name and create an activity
 Change an activity type
 Edit details of an activity
 Save the workout

"I will consider this feature to be a success if I can make creating workouts to be easy, simple and applicable to many styles to fit to the user."

Outcome:

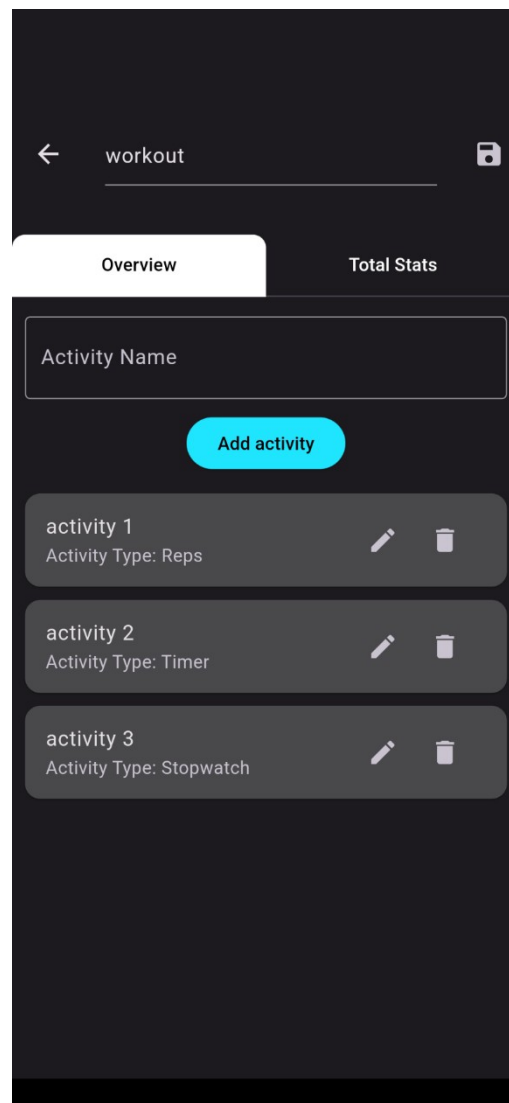
The user can:
 Name a workout
 Add activities
 Edit activities

- Change activity type
- Change details of each activity type
- Delete activities

Save workout

Some of the activities have some errors/bugs, but as a functional feature, I consider it to have met the criteria. I believe users can customize each workout enough to create a strong plan to suite a lot of different workout types, however I haven't included a section to add notes to each activity. This may hinder some users (for example, if a user was climbing and they wanted to add how they want to do each route, they will be limited to the title of each activity).

Evidence:



2. The ability to play each created workout

Expectation: The user should be able to play each workout they have created, with all activities shown.

Criteria:

- Play each workout
- Show each activity and their details
- Have a simple interface
- Assign days of the week to do each workout
- Enter straight into a workout

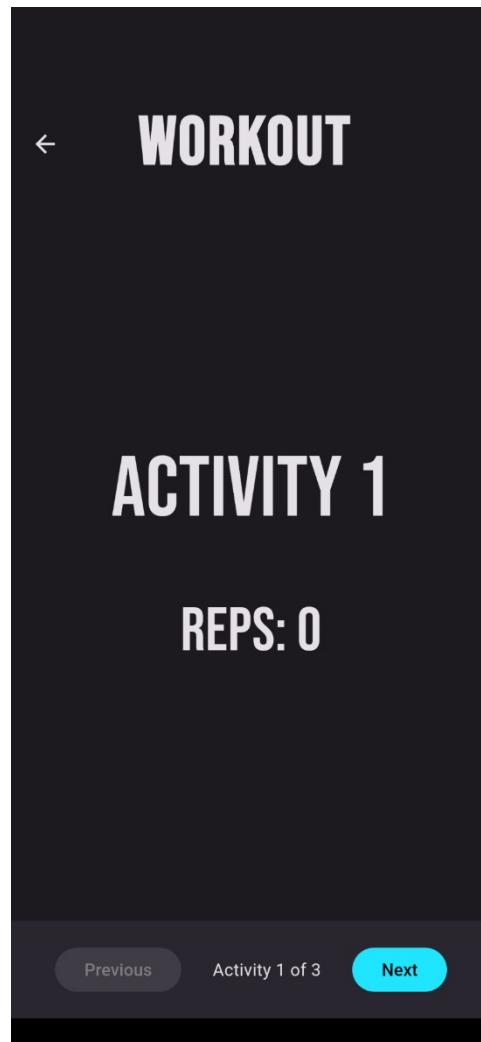
Outcome:

The user can:

- Play each workout
- Follow each activity and view their details
- Save their progress of the workout
- Replay the workout
- Select each workout from the select page

It is not possible at the moment to organize a routine for each workout to be played on each day of the week, which means the user has to select the workout they want to play from a list of created workouts. This covers the basic functionality of the application, but it creates an extra step which the user must complete. This puts a burden on opening workouts, which should be avoided. If I continue to develop this application, I plan to implement this routine feature.

Evidence:



3. To have a simplistic interface

Expectation: The application should have a simplistic interface which is easy to use, and functional.

Criteria:

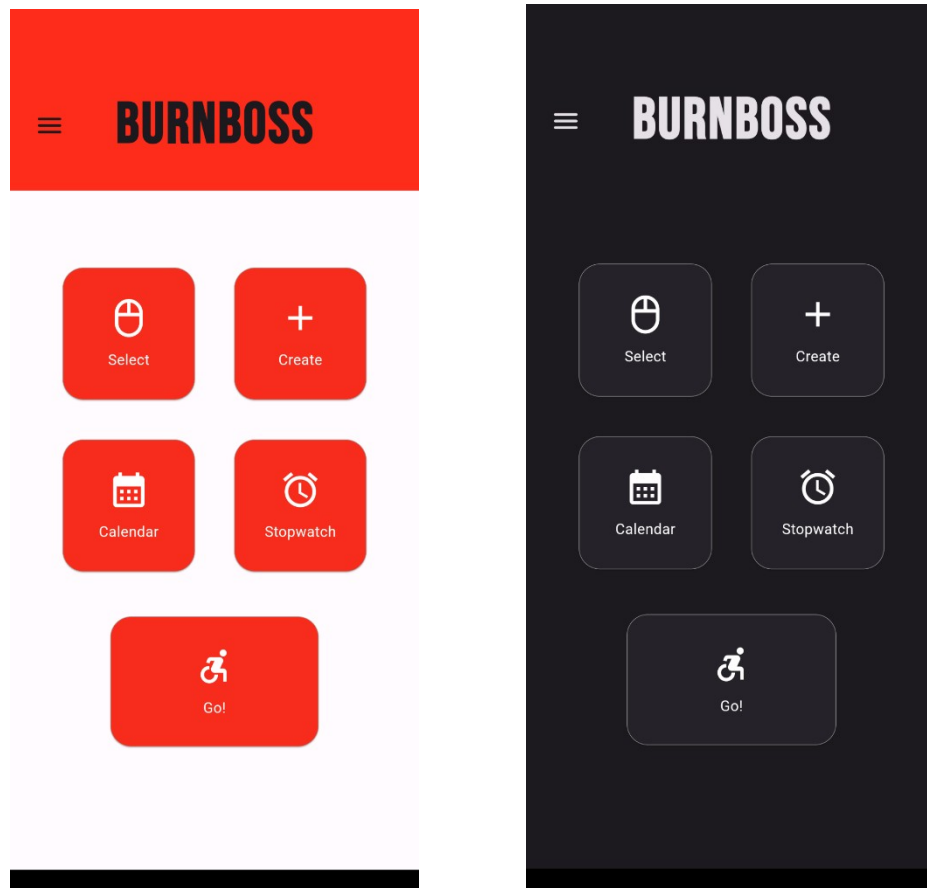
The interface should be:

Simplistic
Efficient
Consistent
Bright
Easy to navigate

Outcome:

I believe that the interface meets these criteria, and allows the user to navigate the application easily. It does not bore the user, and holds a consistent theme which can be changed easily.

Evidence:



4. Save data to profiles

Expectation: The user should have the choice to register for an account, so that they can save their data (e.g. workouts).

Criteria:

User can register for a guest account, or register with their email and password.

The user can save each workout to their profile

The user can save their details to each profile

The user can log back in to their account from any mobile device

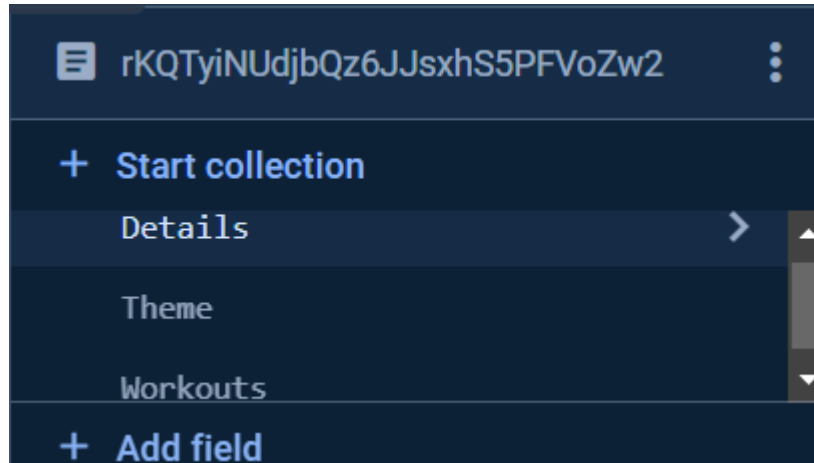
Outcome:

I think that these criteria have been met fully. However, the user does not have that many details which can be saved to their account. They have:

Workouts
Email (this cannot be changed)
Theme

I would like to include a username for each user, which can be used throughout the app. Also, I would like to give each user the ability to change their email after their account has been created.

Evidence:



5. A calendar to view workouts

Expectation: The user should be able to view their progress with their workouts in a calendar, and view the workouts coming up.

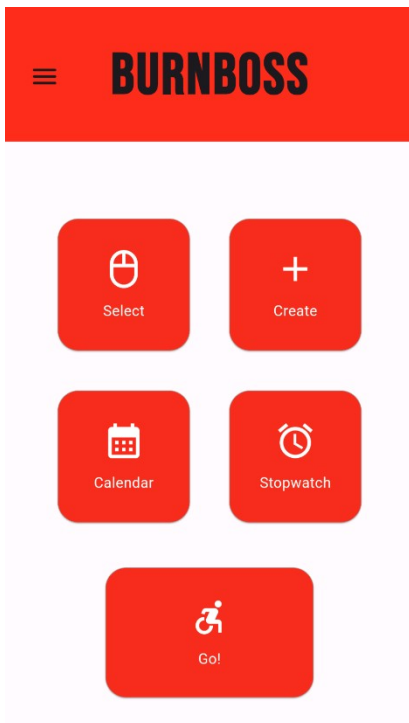
Criteria:

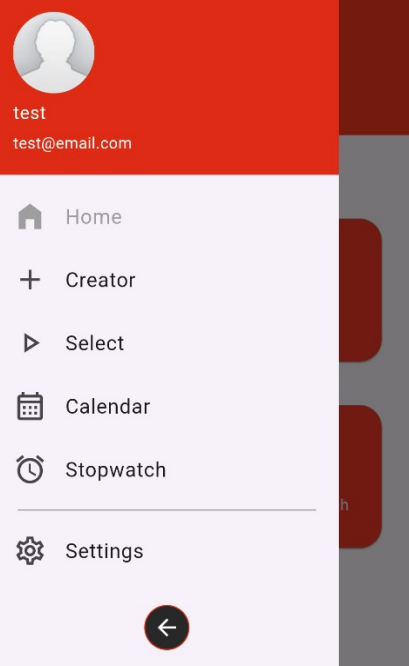
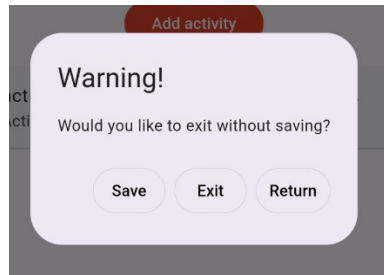
Set workouts within a routine
Store workouts completed in a workout, which can be viewed
Add notes to each day

Outcome:

I have not achieved this criterion. I would like to add this functionality to my application after the app has been released, but at the moment it has not been added. It is not a functional feature which is necessary to run the app, and complete its basic needs.

5.2 Justification of Usability Features

Learnability		
<p>Simplistic interface:</p> <p>It was important to me to make sure that the interface was simple, and easy to view. It was important for my interface to be easy to navigate, as this meant I shouldn't need a tutorial and that my app is easy to pick up. This is important for my low IT literacy userbase.</p>		<p>I think that I have met the criteria to quantify this as a success. The fonts and colours give an exciting and fresh look to my application. The simplistic theme allows easy navigation, promoted by icons throughout the app. A dark mode is also available within the settings.</p>
<p>Recognizable icons:</p> <p>An important area in an easy-to-understand app is recognizable icons. Icons which are used within multiple other apps, and show what each feature does with just a small picture.</p>		<p>This criterion has been met throughout the application. I have used the standard bank of icons provided with the built-in material-3 support. It is a lot easier to look at an icon, rather than to read words, when navigating my app.</p>
Efficiency		

<p>Easy navigation: It was my aim to give each user a quick and easy workout app, which doesn't get in the way of their workout routine.</p>		<p>I think that I have achieved this, and that my app is easy to navigate. It is clear which page is which, and which features are available for each section. I think this should allow users to easily move between pages, and use my application to the full extent.</p>
<p>Memorability</p>		
<p>App intention: It is important that my application is set apart from others within the same market.</p>		<p>I think that the experience achieved within BurnBoss is different from those which aim to achieve the same idea. The simplicity of creating the workouts and playing them allows the user less hinderance when using the app.</p>
<p>Errors</p>		
<p>Error handling and redirection: I intend to make my app available to the general public, which means I must create a useable and simple environment, in which low IT literacy users can use each function just as well as those with a higher IT literacy. Helpful redirects will allow users to learn from any mistakes made, and should not dissuade them from using</p>		<p>Throughout my app, I have tried to limit any possibilities of creating errors through the app. This has been done by using destructive testing from specific testers who were given instruction to try and create errors. When errors or vulnerable areas were found, they were fixed and sealed with either character limits, or alert dialogs. These stop any users from breaking/ crashing the app.</p>

BurnBoss		
----------	--	--

5.3 Maintenance

In the future, I will need to maintain BurnBoss, in order to keep up with system capabilities and relevance. It is unlikely that I will need to change the base features of my app, as it is unlikely that methods of working out will change radically. In order to maintain the code and integrations for BurnBoss, I have written code that should be easily comprehensible and will be easy to pick up again. My code and integrations with Firebase should be explained and understandable enough for me to introduce it to a new developer, and for them to understand and pick it up on the functionality.

Here are some of the features I have included:

Comments – These describe code and its functionality, without affecting the code itself. It conveys readable descriptions intended to help developers understand what is going on. I have used them to describe what different methods and functions.

Sensible variable/function names – It is important to name each variable and function sensibly, as it makes it easier to cross reference and navigate code. Each name should describe the function it performs, or what the value relates to or represents. It also makes it more comprehensible to third parties.

Indentation – Dart follows strict indentation conventions so I had to stick to these and format my code correctly. Indentation helps to break up code and make it easy to distinguish between sections.

Modular functions – It was important to separate my code into modular sections. Each function should ideally be independent from the main code, and should be called when appropriate. This allows me to re-use different functions throughout my code, which saves me time and effort. It helps me when de-bugging or error correcting, as it means I only have to correct a single function which is rolled out to different areas.

Database conventions – While using Firebase, I have tried to follow good practices, such as creating automatically generated primary keys for different entities such as workouts and activities. This creates a more robust database, and removes risk of errors due to incorrect referencing.

5.4 Limitations and Remedial actions

BurnBoss successfully achieves the initial, base functionality as intended. It allows users to create their own personalized workouts with various activity types, and play these workouts. The interface and navigation allow each user a clean, simplistic user experience which shouldn't hinder them in their endeavor to work out and improve

themselves. However, there are some areas which are still incomplete, underdeveloped, or missing:

- **Calendar:** The calendar functionality of the workout has not been achieved, and it means that users cannot track their workouts and their progress. There are a few ways in which I could work on this:
 - o I could allow users to view which days they have completed a workout on (and delete those which they don't want to keep), and show statistics or graphs to visualize progress.
 - o I could create a month-by-month calendar in which users can see each day and the workouts completed, along with notes. This could also include workout planning which would show future workouts. At the moment, I am not sure how to plan my database to accommodate for this, but it would be a useful feature and would improve the user experience and could motivate the user to return to BurnBoss
- **Activity depth:** I feel that the activities only perform the basic functions for what the user would expect. I would like to broaden the functionality of activities by including other details such as:
 - o I could include sets for a 'Reps' activity, allowing the user to do multiple repetitions of reps (e.g. 5 sets of 10 push ups). This was requested by a user while working out using BurnBoss. This would also include rest between sets.
 - o Feedback in testing showed that one user would like another activity type: Personal best. This would include a count-up approach to reps or the stopwatch, and allow users to see their previous personal best. This could increase the motivation to work out, and to beat past performance. Currently the user has very little support for progression within BurnBoss.
- **Workout player:** Although the workout player performs the functions necessary to do a workout, I feel like the interface design needs improving, and other features could be added to allow ease of use. For example:
 - o The stopwatch or timer could be shown within a notification when the user leaves the app. It could show the value of the stopwatch, or how long the timer has left. This would require permissions for notifications, and I would have to research how to implement this feature.

There are some limitations which are imposed by integration with other technologies, and it would be beneficial to some users. Technologies such as smart watches and heart rate monitors would expand the capabilities, but they would require strong integration and new features. This is not something that I'm planning on adding, but it would be an interesting capability of BurnBoss.

5.5 Conclusion

Assessing my application as a whole, I can conclude that BurnBoss's basic functionality has been met. Workouts can be created, saved, edited and played – it meets most of the assigned success criteria, but it lacks in areas such as progress statistics or visualization. There are definitely a few features which the application would benefit from such as a gamified algorithm or a reward system. I believe that my app has achieved a simple interface which suits a userbase with a low IT literacy, which sets BurnBoss apart from other competition on the market. I believe that with development with care taken, BurnBoss could rival competition. It is important to keep a close relationship with my userbase, so that I can design and maintain features which users appreciate. To finish, I am pleased with the product I have designed and built, as BurnBoss is a strong foundation for a well-designed application.

6. Appendix

- All source code along with each commit (therefore all versions) are stored in the Github repository: <https://github.com/ch4rl3sF/BurnBoss>
- All form results: <https://drive.google.com/drive/folders/1zDXimzn4rQZK7TfOAquru6gJ7zSOliE0?usp=sharing>
- All videos referenced within 3. *Development* are in the attached folder Videos
- If you would like to download BurnBoss for yourself, all releases will be downloadable from: https://drive.google.com/drive/folders/1jfNhiQZjAsaDXkaF1KlkP0sggfSijcj3?usp=drive_link