1. Introduction

1.1 Statement of the problem

Building routines, having personal rituals, and living to a schedule are well-established foundations for healthy living. Routines ensure that fundamental needs, such as hygiene, diet, and sleep, are maintained. Routines are also pivotal to dealing with stressful situations and maintaining good mental health - keeping to routines ensures positive functioning and helps a return to normalcy after a stressful encounter.

This project aims to discover how to highlight an individual's perception of the routines that make up their life, encouraging their maintenance and the growth of new, healthy ones.

1.2 Project Goal

The project pertains to the complete design and development of a modern Android application that assists its users with self-management – in particular, the application will assist with the management and maintenance of a user's routines and habits.

The application will allow a user to create routine tasks and will encourage the user to engage with them via established effective engagement techniques, driven by logged interaction data. This will make the user generally more aware of the routines that make up their day-to-day life, facilitating routine maintenance and preventing them from being neglected and falling out of habit.

While the application will be helpful to anyone, the target demographic is students new to university. The change in circumstances and independence that comes from early student life results in a need to build up new routines to succeed in university, a naturally stressful environment.

1.3 Scope

The scope of the project includes the research, analysis, and review of existing literature, the design and development of the self-management application itself, developmental testing of the application, an evaluation of the application and the project overall, and planning for user-based testing.

2. Literature Review

2.1 Self Management and Goal Consolidation

Self-management is defined as "*The taking of responsibility for one's own behaviour and well-being"* [1]. It is a vital skill for independence, but it is not one that is generally taught.

Taking abstract ideas and consolidating them into set goals is an effective first step to self-management. A goal allows a vague notion, such as “my room needs cleaning”, to become something more tangible, giving it attainability it did not have before. Goals “structure experience by channelling attention” [2]; a problem is more achievable because it has been given attention and the first steps to overcoming it have been identified.

“Goals” are broad and can vary in quality. “SMART” is a recognized framework for effective goal creation [2], [3]. “SMART” is an acronym, defining five attributes that should be considered for goal creation: Goals should be Specific, progress towards a goal should be Measurable, goals should be reasonably Achievable, the objective of a goal should be Relevant, and goals should be Time-based.

Overall, setting and completing goals is well established as leading to increased feelings of independence and autonomy [3]. This is known as self-determination theory - by achieving goals, an individual feels more independent, and feels encouraged to set themselves more goals, striving for goal completion.

2.2 Routines

Routines can be considered a ‘next-step’ from simple goal setting. A routine is defined as a *“repeated behaviour involving a momentary time commitment task that requires little conscious thought.”* [4]. The important part of this definition is that routines are a *repeated behaviour* – routines aim to make healthy behaviour repeated, extending an individual goal into a long-term self-management solution.

There is a variety of research highlighting the importance of routines for healthy day-to-day living. Drive To Thrive Theory [5] states that an individual’s routines is what determines their stress resilience. It proposes that those who have longstanding, “well-built” routines, are more resilient to both post-traumatic and chronic stress.

Drive to thrive emphasizes the importance of routine maintenance. According to Drive To Thrive, routines fall into two categories: Primary routines link to health and fundamental needs, such as cleaning regularly, eating healthily, and sleeping consistently. Secondary routines are much more personal to an individual—activities such as hobbies or social events. When under stress, an individual’s focus is taken from their secondary routines and refocused on the stressor. As they lose the daily structure their secondary routines provided them with, they approach a “break point” - where primary routines and personal health and wellness start to fall apart.

This observation is not novel to Drive to Thrive – the link between the breakdown of daily routine leading to a drop in daily, self-maintenance and self-care is well established [6].

2.2.1 Developing New Routines

While routine maintenance is important, having well established, maintained routines by itself is not enough. Breaks in routine are inevitable – for example, due to a change in environment. The SARS-Cov-2 (COVID-19) pandemic resulted in a severe, global impact on mental health [7]. This has been attributed in part to the breakdown of routines caused by the quarantine – When the pandemic hit, many did not know how to consolidate and replace the secondary routines that the quarantine made impossible [8]. It is important to be always ready to create new routines.

2.3 Habits

Habits can be defined as a *"psychological disposition to repeat past behaviour" [9]*. A habit differs from a routine in that it is more about an individual's attitude towards the routine and their compulsion to perform it than about a desire to perform the routine itself – a routine becomes a habit once it becomes "natural", no longer being driven by a desire to meet a goal [10].

Building habits in an incredibly effective way to maintain a routine. Because they are more of a compulsion, Habits act as a subconscious guide for behaviour, allowing an individual to engage in a routine in a "thoughtless" manner.

A habit can be broken down into three distinct parts; a cue, a behaviour, and a reward [11]. A cue acts as a prompt to begin a habit. The behaviour is the activity itself, and the reward is what is gained from engaging the habit.

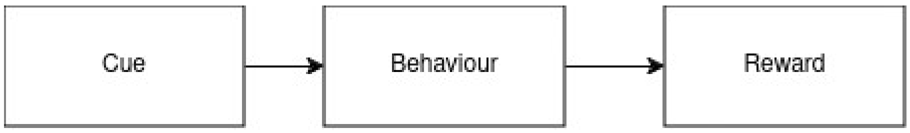


Figure 1: The relationship between habit cues, behaviours, and rewards

Habits are, by definition, built through repetition. Repetition builds an expectation that when a cue occurs, a behaviour and reward will follow through. Once a habit is ingrained, it becomes hard to break – when the cue related to the habit occurs, the expectation for the behaviour will come to mind. Cues are the most effectual part of a habit; other aspects, such as informational campaigns - telling an individual that a behaviour is healthy, or that a behaviour is important - is not enough to effectively alter a behaviour [12].

The long-term effectiveness of building habits is well documented – once a habit has been established, a long term tendency to behave a certain way develops, called a default behaviour. A default behaviour is a behaviour that an individual feels compelled to automatically carry out, leading to a permanent behaviour change [13].

Controlling habits, intentionally nurturing cues to develop long term default behaviours, is a powerful way to effectively self-manage.

2.4 Nudges and Cues

A “nudge” is the process of using a prompt to guide an individual to an action. Using nudges is an effective method to guide behaviour, and makes an individual more likely to carry out an action [14]. Bringing attention to an action through a nudge plays on an individual’s cognitive predisposition to automatically respond and carry it out [15].

Nudges can be applied to building routines and guiding habits. By periodically sending a notification, a nudge encourages an individual to engage in a behaviour routinely. Eventually, this routine transitions into a habit, the nudge becoming the habit's cue [12].

2.4.1 Digital Interrupts - Mobile Notifications

A mobile notification can be an effective way to deliver a nudge. However, its success as a cue depends on a few factors.

For one, it is important that the user sees a notification when it is sent; The more time that passes after a notification is sent, the less effective the notification is [16]. If a notification is not actively taking the user’s attention, it is not as effective. A user’s focus being on another activity, however, is inevitable. If a user is engaged in another task, they are less likely to engage with the notification, with more complex tasks decreasing the likelihood that a notification is checked [17].

Another factor affecting the success of a notification is that the user needs to be willing to act on it. This is dependent on the notification’s usefulness - If a notification ends up annoying the user, they will ignore it [16].

2.5 Gamification

Engaging in a routine before it has been established as a habit is not as easy as it would seem. The fundamental property that makes habits so effectual – their automatic, subconscious nature – works against the growth of new habits; habits resist change [12].

Gamification can be simply defined as “The use of game design elements in non-game contexts”. [18]. It is a technique that has been applied to a wide spectrum of topics, from education and marketing, to engagement and motivation, with significant success [19]. Within the context of routines, gamification can act as the “reward” for engaging with a habit.

2.6 Existing Application Review

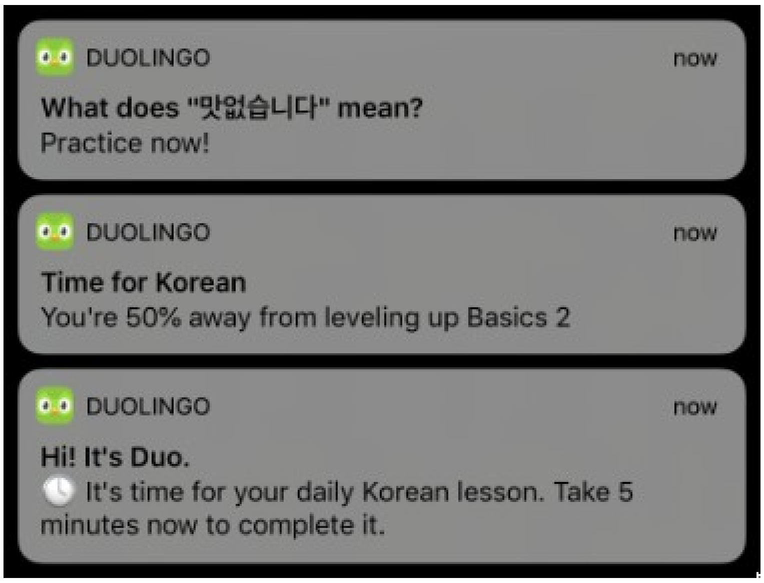
Gamification is a popular, well-used engagement technique, that has received focus for its potential applications in marketing, customer engagement, or education [19]. As a result, there are many applications using gamification in some way. To help identify effective gamification techniques, two applications – *Duolingo*[1] and *Habitica*[2] – have been explored in detail.

1 https://www.duolingo.com/
2 https://habitica.com/

2.6.1 Duolingo

Duolingo is a gamified language-learning educational application that teaches language skills through short, bite-sized lessons. It is the world’s most downloaded educational mobile application, making it an excellent place to look into and learn from for its engagement techniques.

In addition to being hugely popular, Duolingo is well researched —Duolingo themselves have written a paper justifying and evidencing the methods they have used to encourage user learning [20]. Such research is encouraging and bolsters Duolingo as a good case study because it shows that the approaches taken by Duolingo are thought out and backed up by science.



DUOLINGO now

What does "맛없습니다" mean?
Practice now!

DUOLINGO now

Time for Korean
You're 50% away from leveling up Basics 2

DUOLINGO now

Hi! It's Duo.
It's time for your daily Korean lesson. Take 5 minutes now to complete it.

Figure 2: Notifications within Duolingo. Sourced from (Yancey and Settles 2020)

Duolingo employs notifications to keep its users learning. Duolingo has extensively researched [21] the effectiveness of their notifications as an engagement technique; it is a core functionality of Duolingo.

Outside of Notifications, Duolingo combines various gamification elements to keep users engaged. Three of note are Experience, Streaks, and Lingots, Duolingo’s in-app currency.

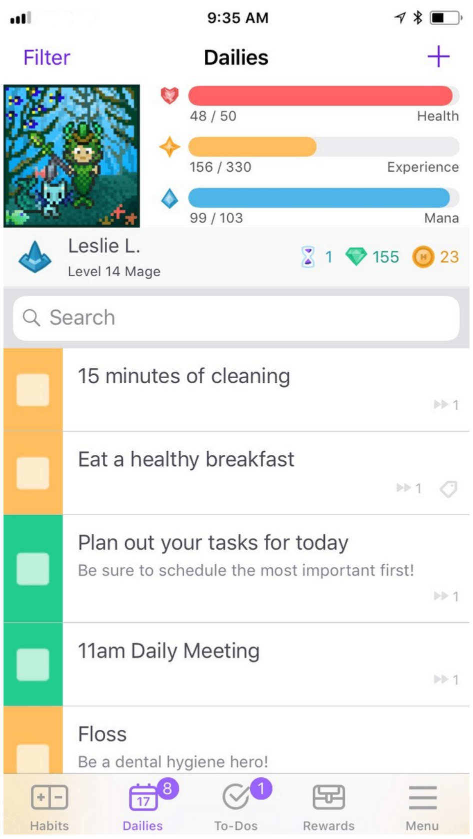
“Experience” in Duolingo is used to quantify user engagement. Experience gets rewarded for successfully interacting with a lesson, with more experience rewarded for doing well. Experience can then be compared against other users competitively, on a scoreboard.

Duolingo’s other gamification aspect is Streaks. Users are encouraged to maintain “streaks”; daily interactions with Duolingo, rewarded with multipliers to the amount of experience they earn. Lingots, Duolingo’s coins, can be earned as an additional reward for completing lessons, and can be used to allow for Streak “skip days”, either, or retroactively, “repairing” a broken streak.

Streaks are important to Duolingo as a company, and thus they have researched it extensively. They describe engagement as one of the hardest parts of learning, and use gamification as their way to combat it, forming one of their “Five pillars of learning” – their “stay motivated” pillar [20].

2.6.2 Habitica

Habitica is a task management application that, like Duolingo, makes use of gamification, but to a much greater degree than Duolingo- Its gamified nature is its selling point. Users set up tasks they want to complete, be it deadlines, or re-occurring, daily tasks. Completing tasks rewards your avatar – a virtual representation of you - with experience, which develops them with more hit-points and accessories. Failing tasks makes it take “damage”, losing hit-points.



15 minutes of cleaning

Eat a healthy breakfast

Plan out your tasks for today
Be sure to schedule the most important first!

11am Daily Meeting

Floss
Be a dental hygiene hero!

Figure 3: The representation of habits and “stats” in Habitica. Sourced from habitica.com/static/press-kit

Habitica is a relevant case study for the project, thanks to a report into the counter-productive effects of gamification that used Habitica as a case study [22]. The report detailed two studies: one more qualitative, on an individual, interviewed user, and another more quantitative. Their study identified a variety of counter-productive effects Habitica’s gamification was unintentionally causing.

Habitica was found to “punish productivity”. In times of high productivity, a user might not break from what they are doing to check off tasks in Habitica, resulting in tasks not being marked completed in time. This encouraged users to break during times of high productivity, breaking them out of productive behaviour; The case study user found themselves too busy to check off their to-do tasks in Habitica, resulting in them receiving demotivating negative feedback.

Another significant counter-productive effect was Habitica rewarding procrastination. Habitica's reward structure encourages users to defer tasks that could be completed spontaneously to be completed the next day, encouraging procrastination over getting things done. The case study user found themselves putting off tasks such as organizing their emails for tomorrow, as it would a higher reward, when there was no reason to not complete the task on the spot. The realization of which demotivated them away from completing the task at all.

A third counter-productive effect has confusing rewards. In Habitica, rewards are either qualitative or quantitative. They take the form of "stat increases" for your avatar, or gamified "items" your avatar can equip. Both can be confusing – it is unclear how valuable the quantitative "stat increases" are, as are the items.

Table 1: A table comparing application features

Feature	**Duolingo**	**Habitica**
Streaks	Yes	Yes
Notifications	Yes	Yes
Competitive Scoreboards	Yes	Yes
In-App "shop"	to-an-extent	Yes
Skip Days	Yes	No

2.7 Streaks

From the reviewed applications, Streaks were chosen as a gamification technique to explore in further detail - it complements the periodic nature of routines.

A streak is simply *"a series of three repeated, consecutive events"* [23]. It is considered a gamification technique because of how streaks can compel behaviour; there are many well documented cases of individuals going out of their way to maintain streaks [24].

A more specific definition of streaks that links them to periodic routines is "*a repetitively completed task attributed to the resolve of the actor, defined by absolute performance and temporal parameters, and construed and quantified by the actor as an uninterrupted series.*" [24]. Routine engagement streaks, which have a temporal parameter, are quantified into streaks by repeated, concurrent completions.

A streak's quantification is where it becomes a gamification technique. The symbolic representation of a streak – how a streak is represented and recorded – brings awareness to an individual's streaks and successful engagement, having an effect on how they are "processed, encoded, and remembered" [25].

The effectiveness of streaks is well recognized; As well as Duolingo and Habitica, over 101 distinct applications [25] were found to have incorporated streaks in some way to encourage user engagement.

2.8 Beneficiaries - Students

The first year of University is a massive adjustment in personal autonomy for university first-years. On top of living independently, which, for many students, is a first[3], students have to adapt to the general self-motivated nature of university. Unlike lower levels of education, where learning is heavily structured, a student’s studies are expected to be largely independent, with help only being provided if a student chooses to reach out. There’s also the environmental, social and cultural changes to consider; if a student is living away from home, they have to also adapt to an entirely different culture, and have to build themselves an entirely new social group. These have all be attributed to causing adjustment difficulties for students [26].

It is unsurprising that many students struggle with adjusting and managing their time. Failing to adjust to the independency required by university can be a very real reason for why a student academically fails, or drops-out of university altogether [27]. This does not have to be the case; Research [28] have shown that simply providing students with support in dealing with their new independence is, for many students, enough to get them back on track with their studies.

Students who have just began university have gone through a huge environmental change; they need to be willing to compose new routines and habits if they want to achieve healthy self-management.

3 https://cls.ucl.ac.uk/class-of-2023-more-likely-to-be-stay-at-home-students/

3. Project Management

3.1 Skills Audit

To help identify weak points that need to considered when scheduling the project, a skills audit has been created. It identifies areas that need to have more or less consideration put into. The skills audit has been represented in two skill matrices; a “technical skills” matrix, and a “project skills” matrix.

Table 2: The Technical Skills matrix

Skills	**Ability (1-5)**
Programming	4
Interface Design	3
SQL	5
Database Design	4
Database Querying, Operations	5

Table 3: The Project Skills matrix

Skills	**Ability (1-5)**
Project Planning	3
Research	3
Code Planning / Modelling	3
Testing	2

3.2 Risk assessment

To identify potential risks that could come up while developing the project, a risk assessment has been carried out. Risks, their probability, and the severity of them occurring has been assessed, and mitigations to prevent them from happening have been defined.

Table 4: The project risk assessment

Risk	**Severity (1-10)**	**Probability (1-10)**	**Mitigation**
Sickness, other commitments taking project time	2	9	Flexibility in project planning, over-estimation of task duration length to prevent issues affecting the project.
Loss of Work due to technical failure	9	3	All project files have are backed up across multiple devices. The code for the project itself will also be backed up to a Gitlab[4] repository.
Lack of Skills	5	5	Consulting learning resources, reaching out if struggling.
Under-estimation of time	8	5	Project planning for time using a Gantt chart, careful consideration of scope.
Scheduling Mismanagement	7	7	Frequently meetings with the Project Supervisor, and by scheduling the project with a Gantt chart.

3.3 Development Model

To encourage effective development, an established model was chosen to base development off of.

Waterfall Model

The waterfall model was decided against. An iterative development model, that allows for the project to be researched, planned, and developed in stages, was preferable, as it allows for the project to be split into smaller, manageable chunks.

AGILE

Agile, one of the most well-known iterative models, was considered for the project, but was decided against- AGILE has a strong focus on collaborative work, and on working closely with clients to produce high quality, deployable code at each iteration. This does not suit the nature of the project.

Spiral Model

The final model decided on was the Spiral model. The spiral model sits between AGILE and the Waterfall model – it is iterative, but without collaboration and client focus of AGILE. As there is no need to produce a fully deployable product until the project is finished, the spiral model is ideal.

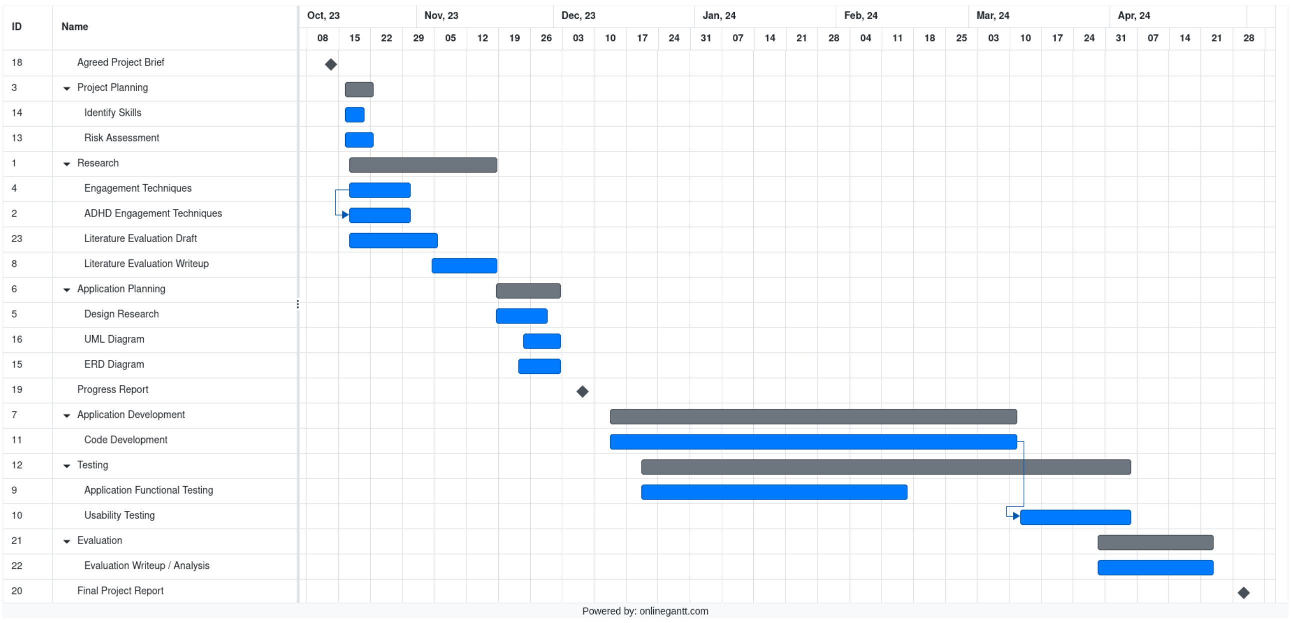
4 https://about.gitlab.com/

3.4 Project Planning

To manage the development time of the project, an initial Gantt chart was created, using *onlinegantt.com*[5]. This acted as a project top level overview, guiding the entire project.

Every week, as the project progressed, a weekly review meeting was held with the project supervisor. This meeting identified any blocks or hold-ups in development, areas that needed more focus than what they were being given, areas that were being over-focused on, and generally discussed changes in the project plan due to other commitments.

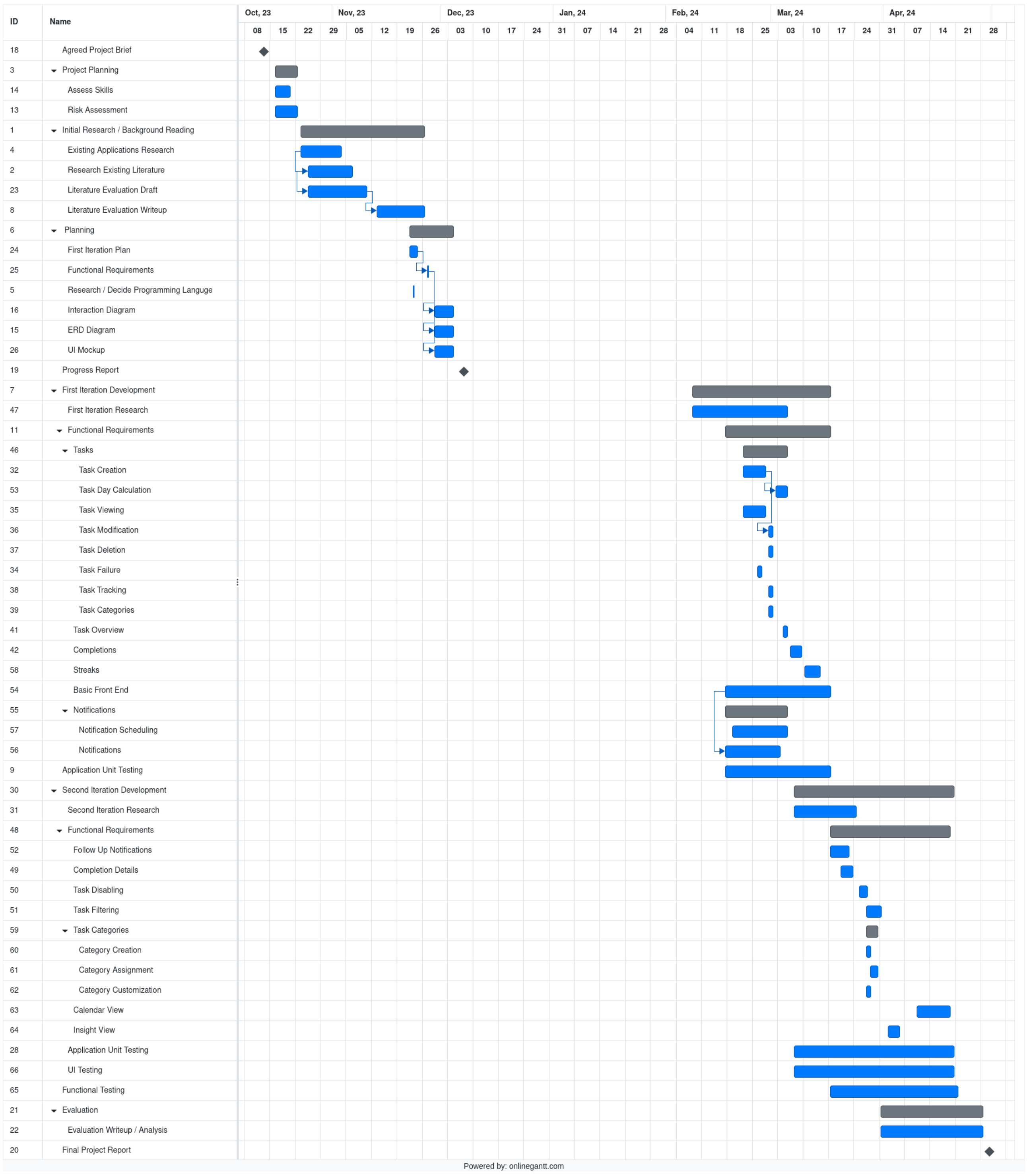
To keep the project Gantt chart useful, it was adjusted to the new time requirements and expanded to new requirements as the project progressed. The Gantt chart grew significantly as the project went on and requirements were explored in more detail.



ID	Name
18	Agreed Project Brief
3	Project Planning
14	Identify Skills
13	Risk Assessment
1	Research
4	Engagement Techniques
2	ADHD Engagement Techniques
23	Literature Evaluation Draft
8	Literature Evaluation Writeup
6	Application Planning
5	Design Research
16	UML Diagram
15	ERD Diagram
19	Progress Report
7	Application Development
11	Code Development
12	Testing
9	Application Functional Testing
10	Usability Testing
21	Evaluation
22	Evaluation Writeup / Analysis
20	Final Project Report

Figure 4: The initial project Gantt chart

5 https://www.onlinegantt.com/#/gantt



ID	Name
18	Agreed Project Brief
3	Project Planning
14	Assess Skills
13	Risk Assessment
1	Initial Research / Background Reading
4	Existing Applications Research
2	Research Existing Literature
23	Literature Evaluation Draft
8	Literature Evaluation Writeup
6	Planning
24	First Iteration Plan
25	Functional Requirements
5	Research / Decide Programming Languge
16	Interaction Diagram
15	ERD Diagram
26	UI Mockup
19	Progress Report
7	First Iteration Development
47	First Iteration Research
11	Functional Requirements
46	Tasks
32	Task Creation
53	Task Day Calculation
35	Task Viewing
36	Task Modification
37	Task Deletion
34	Task Failure
38	Task Tracking
39	Task Categories
41	Task Overview
42	Completions
58	Streaks
54	Basic Front End
55	Notifications
57	Notification Scheduling
56	Notifications
9	Application Unit Testing
30	Second Iteration Development
31	Second Iteration Research
48	Functional Requirements
52	Follow Up Notifications
49	Completion Details
50	Task Disabling
51	Task Filtering
59	Task Categories
60	Category Creation
61	Category Assignment
62	Category Customization
63	Calendar View
64	Insight View
28	Application Unit Testing
66	UI Testing
65	Functional Testing
21	Evaluation
22	Evaluation Writeup / Analysis
20	Final Project Report

Figure 5: The final project Gantt chart

3.5 Ethics

Before developing the project, it is important to consider any ethical problems that it could face.

3.5.1 Notifications

The mobile notifications the application uses function by interrupting a user's focus. This can be detrimental to a user; a notification could distract them from an important task or activity. To mitigate this, the application will send notifications at a regular level of urgency. This means that the Android System will hide the application's notifications while the phone is in "Do not Disturb" mode, allowing for the user to prevent notifications from distracting them. In addition to this, if a user does not want an interrupt from a particular task, they are provided with an option to disable it.

3.5.2 User Testing

To effectively user test the application, it needs to be used by individuals who are looking to build new routines – ideally, the model demographic for the application, university students. Those who are looking to establish new routines are vulnerable; routines play an important role in healthy living. As a result, testing the application is ethically difficult; if the application were to fail to be useful, or worse, detrimental, it could cause a significant decrease in an individual's well-being.

Another important thing to consider for testing is that a user's routines and routine behaviour is personal, and that they may not be willing to discuss their behaviour or the routines that they are trying to engage with. This makes surveys with guided questions a better user testing technique than more in-depth interviews.

4. System Specification and Design

4.1 Literature Conclusions

The literature reviewed in section 2 acts as the scaffold for selecting the features to develop into the application to ensure that it is effective.

Habits and Routines

The main focus of the application is to encourage a user to engage in routines, until they become habitual. The application will accomplish this by prompting its user to create “periodic routines”; completable routines created with the aim that the user will eventually fall into a time-cued habit of completing. Routines will be able to be completed at any point during the day, and the reward will be a simple incremental streak. This is to prevent the issues of “punishing productivity” and “confusing rewards” that were found in Habitica.

Habit Cues

Notifications, more specifically, Android System Notifications, will be a core part of the application. These will act as the application’s nudge, and the eventual cue of the habits that the user is trying to build.

Notifications will function by sending a notification a configured amount of time before the user’s task completion time. The user will then have time to complete the task.

Follow-up Cues

If a user fails to complete a task, they will be prompted with a follow-up notification, and they will still have the opportunity to complete the task late. By re-sending the notification, the user is given two cues, increasing the chances of their engagement.

SMART Goal Creation

The application will allow its user to easily create custom tasks that they wish to build into an eventual habit. As a result of being created with becoming a routine in mind, these tasks act as goals with natural SMART characteristics: tasks will have a set “repeat period” that acts as a natural time-bound, the completion of a task acts as a quantifier, making any set goal inherently measurable, and tasks will naturally have to be achievable, to function as routines.

Gamification and Streaks

Streaks, similar to Duolingo’s engagement streaks, will be an important feature within the application, acting as its core engagement technique - A user’s streak, and how they have maintained it, works as the reward of the routine that the user is trying to maintain. Streaks will form on a per-task basis as the user uses the application.

4.2 Stakeholder Analysis

Because of the nature of the project, the only project stakeholder to be considered is the end user.

A model end user for the project is a University student, new to living independently. They want to use the application to build, establish and maintain healthy self-management habits as they settle into the new environment. The application will not, however, be exclusive to fresh university students – the application is usable by any individual looking to develop routines.

4.3 User Stories

User stories were created to understand the different wants a model end user might have from the system.

Table 5: User Stories, with goals and justifications

Story Name	Goal	Justification
Task Creation	I want to be able to create custom task goals, with a customizable due time, due date, and completion period.	So that the application is personal to me and the routines I want to build.
Task Summary	I want to be able to easily view a summary of the tasks I have to do.	So that I can view an overview of my upcoming tasks.
Task Completion	I want to be able to quickly mark a task as complete or incomplete.	So that the application is easy to use – task completion will be my main interaction.
Task Modification	I want to be able to view, modify, and delete my existing tasks.	So I can remind myself about them and update any out-of-date information.
Task Disabling	I want to be able to disable tasks.	So that I do not get prompted to complete tasks I am unable to complete.
Completion Alerts	I want to be alerted periodically when it is time to complete a task.	So that I have a digital interrupt to bring my attention to the task.
Completion Follow-ups	I want to receive a "follow-up" alert if I miss a task.	So that I am prompted if I missed my initial prompt.
Streaks	I want to be able to view how long I have maintained a task completion streak.	So that I am motivated to continue maintaining consistent task completion.
Streak Summary	I want to be able to easily view the streaks that I have completed, and the streaks that I will have to complete.	So that I can view a longer-range overview of my upcoming streaks.
Completion Insights	I want to view insights into what time I have been completing my tasks.	So that I better understand how well I am keeping to my routine goals.
Task Filtering	I want to be able to filter my tasks.	So that I can find appropriate tasks easier.
Notification Interaction	I want to be able to interact with the application through notifications.	So that I can quickly and conveniently interact with the application.
Insight View	I want to be able to view an informational overview of all my streaks and tasks.	So I can view data about my tasks as a whole.
Task Categorization	I want to be able to create categories for my tasks.	So that I can organize my tasks, and gain a per-task understanding about how well I am completing them.

4.4 Application Use Cases

4.4.1 Use Case Diagrams

To explore the interactions that will be had with the application, some Use Case diagrams were created. Initially, a table of use cases and a single, large use-case diagram, split into interactions within the application and interactions with the android system outside the application, were created (Appendix 1). This was then refined into categories, representing the different areas of the application.

Because of the data-driven nature of the application, interactions with persistent system storage were not shown; almost every use case would interact with persistent storage. Instead, modelling pertaining to persistent storage can be found in Section 5.1.

The Overview Menu

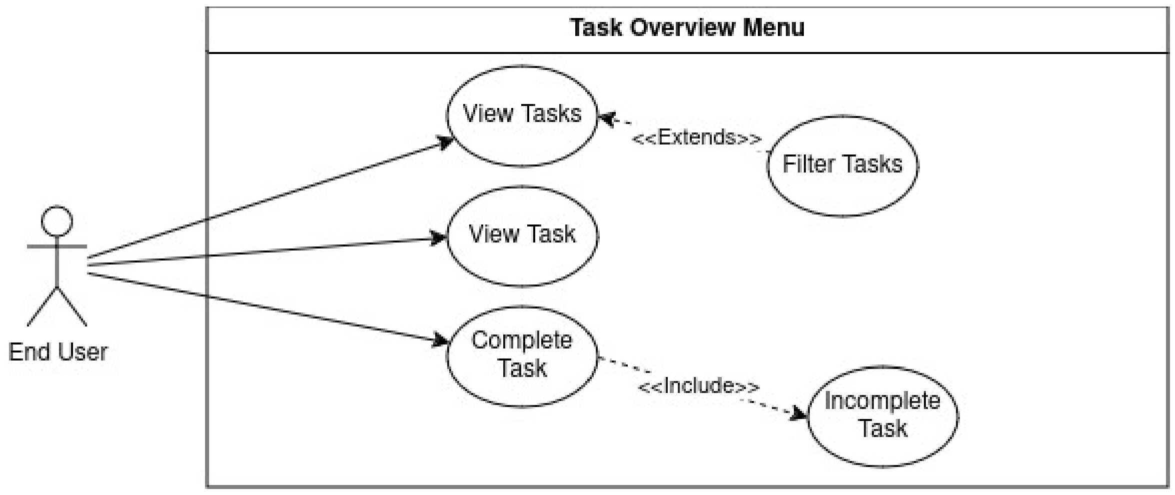


Figure 6: A use case diagram of the overview menu

The overview menu represents one of the main screens of the application. It allows for easy task summary viewing and completion.

The Task View Menu

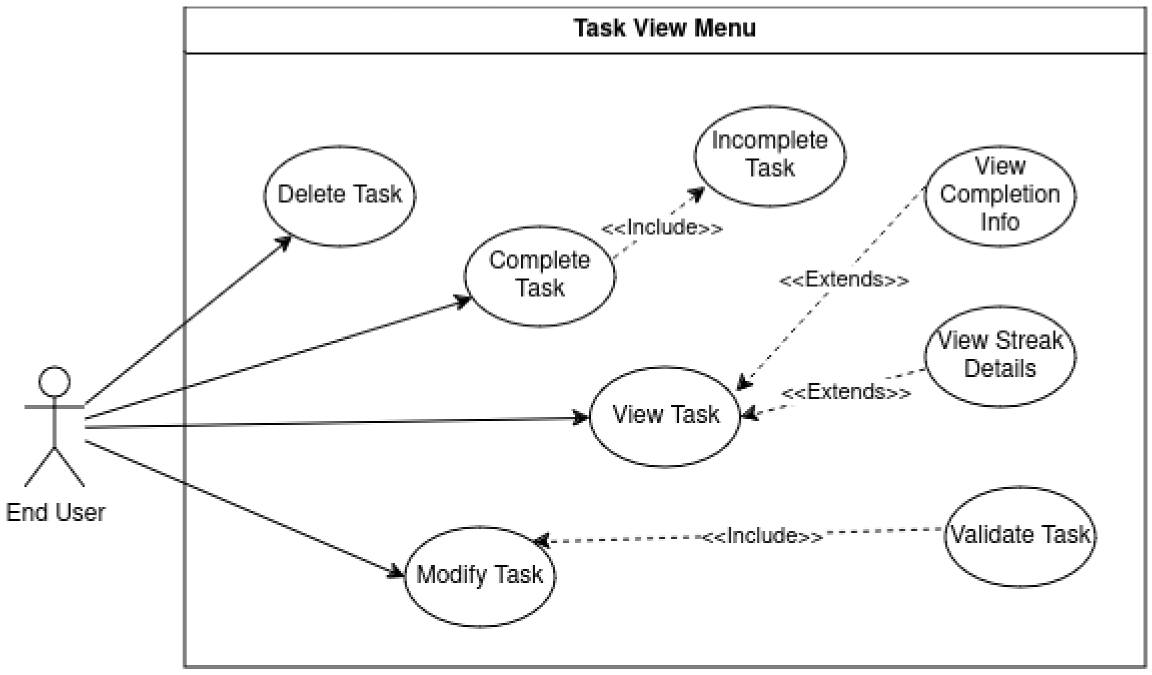


Figure 7: A use case diagram of the task view menu

The task view menu allows for a user to view detailed information about a specific task, and to do task related actions.

The Add Task Menu

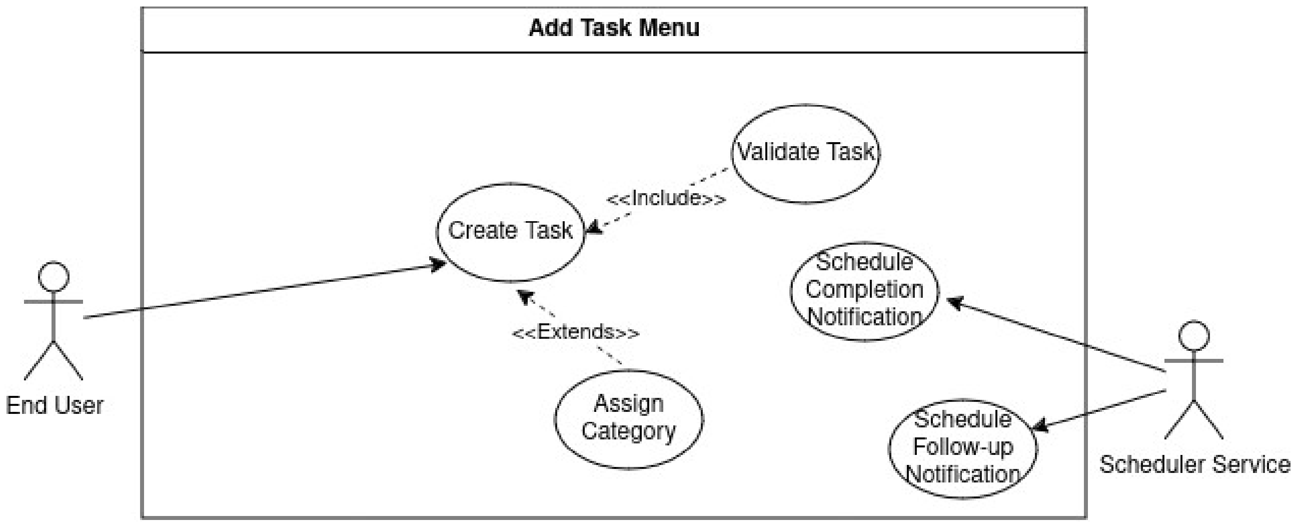


Figure 8: A use case diagram of the add task menu

The add task menu allows for valid tasks to be created, and handles the initial scheduling with the android scheduler.

The Calendar Menu

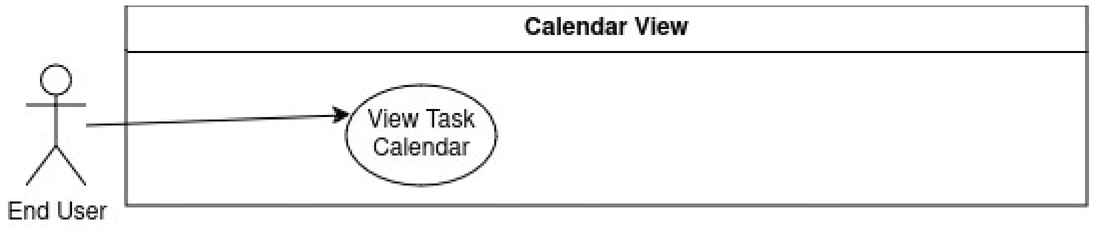


Figure 9: A use case diagram of the calendar menu

The calendar menu allows for the completion days of routines, and past routine accuracy, to be seen.

The Settings Menu

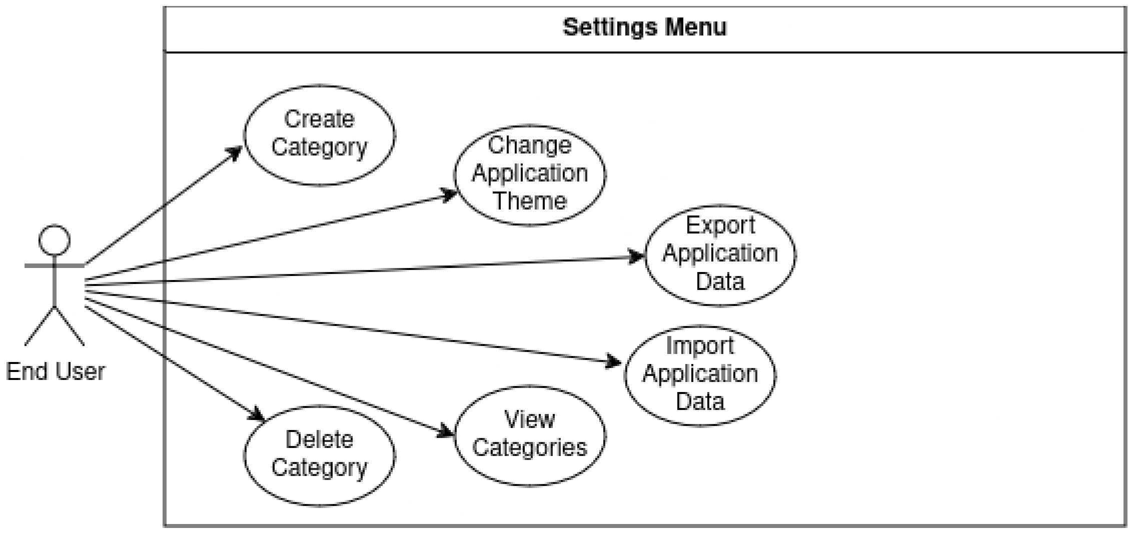


Figure 10: A use case diagram of the settings menu

The settings menu allows for changes to be made to application settings.

The Insights Menu

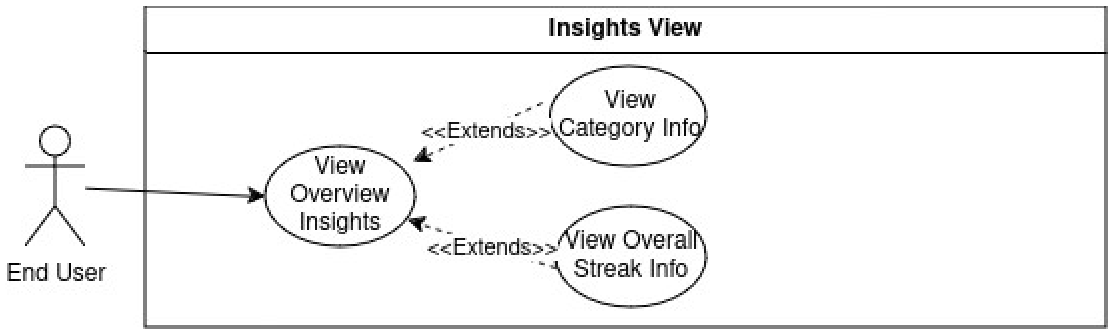


Figure 11: A use case diagram of the insights menu

The insights menu allows for insights about tasks and streaks to be viewed.

The Android System

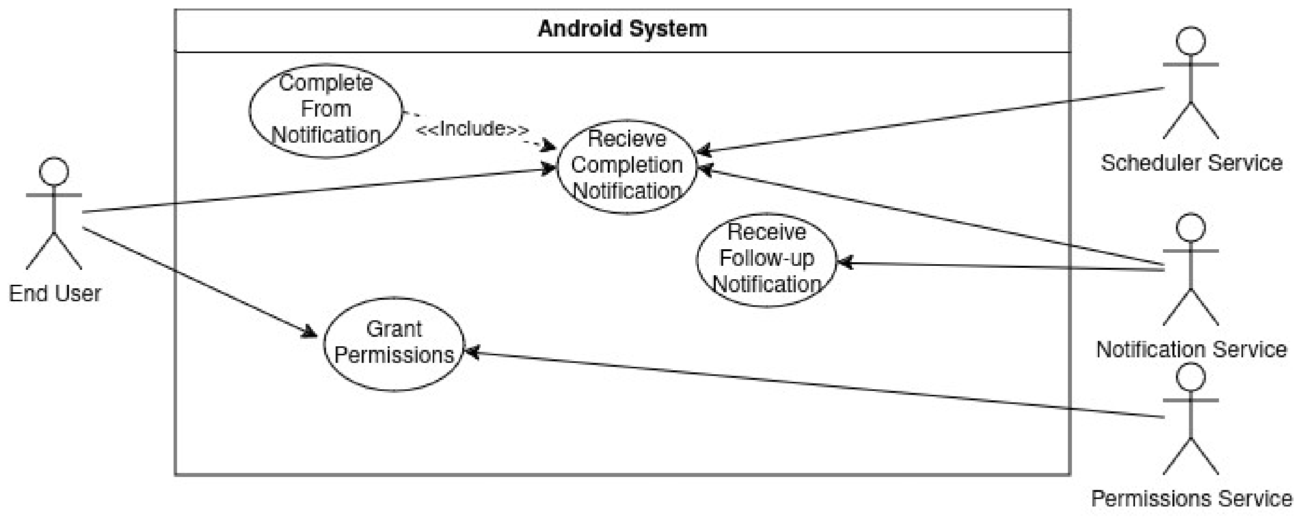


Figure 12: A use case diagram of the interactions made outside the boundary of the application, with the android system

Because of how the application will utilize android system notifications as digital interrupts, the application will have to interact with android system components outside the main application.

4.4.2 Use Case Flows

To gain a better understanding of the steps of the more involved use-cases, some Use Case Flows were created.

Table 6: A table exploring the more complex use cases of the application in more detail.

Use Case Name:	**Flows**	**Error Handling**
Add Task Use Case	1. The User opens the Application. 2. The User presses the “Add Task” button, opening the Add Task menu. 3. The User types in details, such as the name, description, period and start date for the task they wish to add, inserting correct values. 4. The User can assign the Task to any relevant categories. 5. The User confirms that they want to add the task. 6. The System validates the entered fields. 7. The System closes the Add Task menu.	• If the user types in incorrect details, they are visually shown this through “error state” fields. • If the user has not granted the application permissions, a permission prompt will open once the Add Task menu is opened. • If a task is added without sufficient permissions, the user is prompted with a warning that the digital interrupt of the task will not function correctly. • If a task is added and a field has an incorrect value, an error message describing the problem will be shown.
Task Overview Use Case	1. The User opens the Application. 2. The User is greeted with an overview of the tasks they have to complete today. 3. If the user wishes to view all of their tasks, all of today’s tasks, or tasks of a specific category, they can select filters from the task view. 4. The System shows the user completion streaks for the tasks they have completed.	• If a user does not have any tasks on the day view, a brief “You have no tasks for today” message is shown. • If a user does not have any tasks at all, a “You have no tasks” message is shown.
View Task Use Case	1. The User opens the Application. 2. The User selects a specific task from the Task Overview. 3. The User can see all the information they entered about their task, such as its name and next completion date. 4. The User can see information about their best streak and their current streak. 5. The System shows graphs about a user’s task completion history to the user.	• If a user has not completed the task before, a brief “you haven’t made any completions” message is shown.

Task Notification Use Case	1. The System sends an Android System notification at the "notification time" of the Task. 2. The User views the notification, and can see the name of the task they have to complete, and the description they entered for it. 3. The User clicks on the Notification, and the relevant task is opened within the Application. **Alternate Flow**: The user doesn't complete the task before the task is due 1. The System sends an Android System notification at the "goal time" of the Task. 2. The User clicks on the Notification, and the relevant task is opened within the Application.	• If a task has been deleted, the notification is not sent.

4.5 Interface Design

To help design the Application graphically, Figma[6], an interface design tool, has been used to create a mock-up for the components and menus of the application. Figma was chosen because it is a well-supported, featureful interface design tool, used by a variety of professional companies[7].

4.5.1 Material 3

Material Design[8] is a design system created by Google. It is a collection of guidelines, tools, and components for developing minimal, accessible, and cohesive UIs. Material Design 3, the latest revision of Material Design, has been chosen as an application style to ensure that the User Interface has a cohesive design and feel. Material Design provides a design kit for Figma, allowing for mock-ups for components and screens to be easily created.

6 https://www.figma.com/
7 https://www.figma.com/customers/
8 https://m3.material.io/

4.5.2 Component and Menu Mock-up

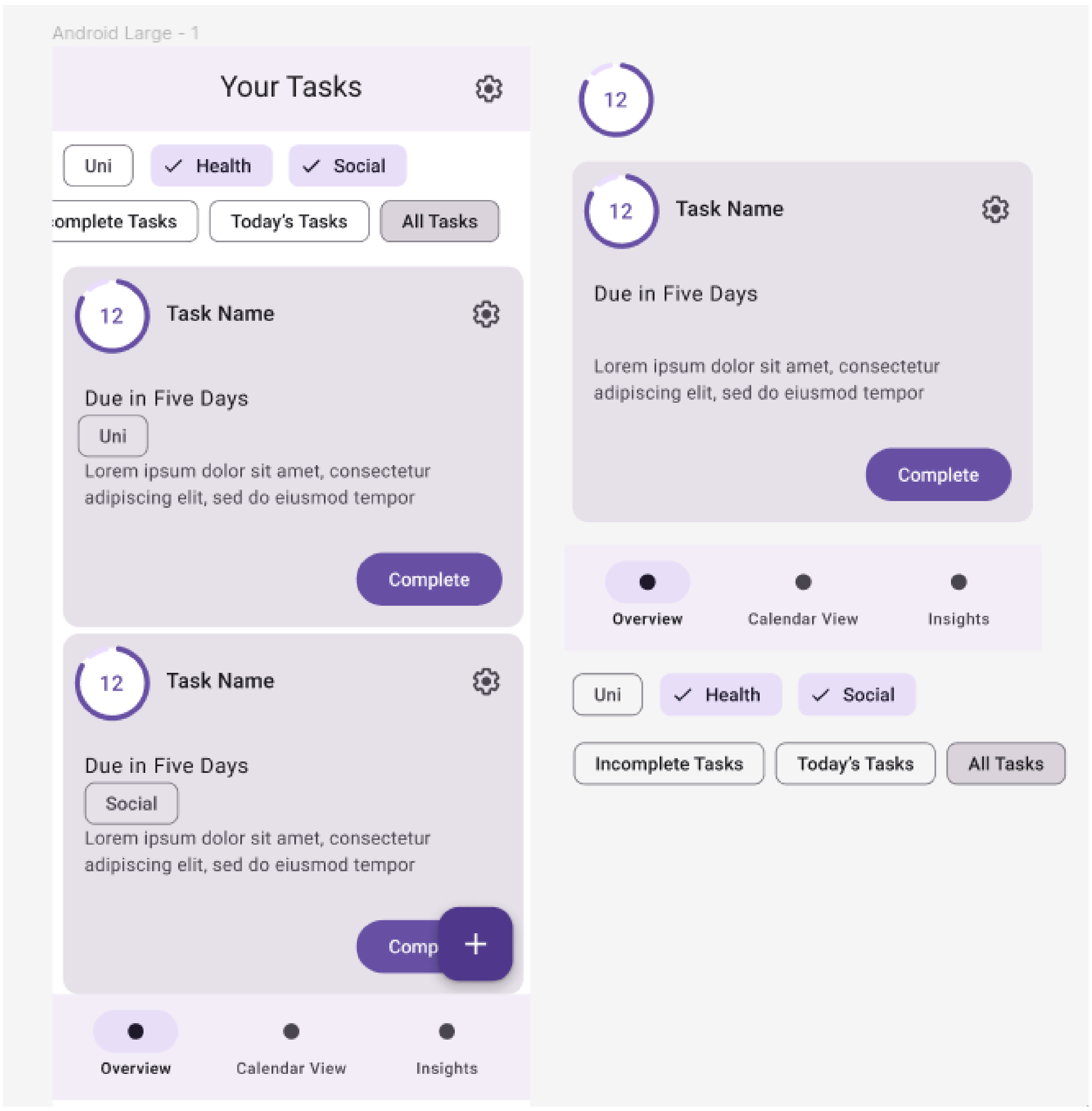


Figure 13: A Figma mock-up design of the "Overview" menu, and the components making it up

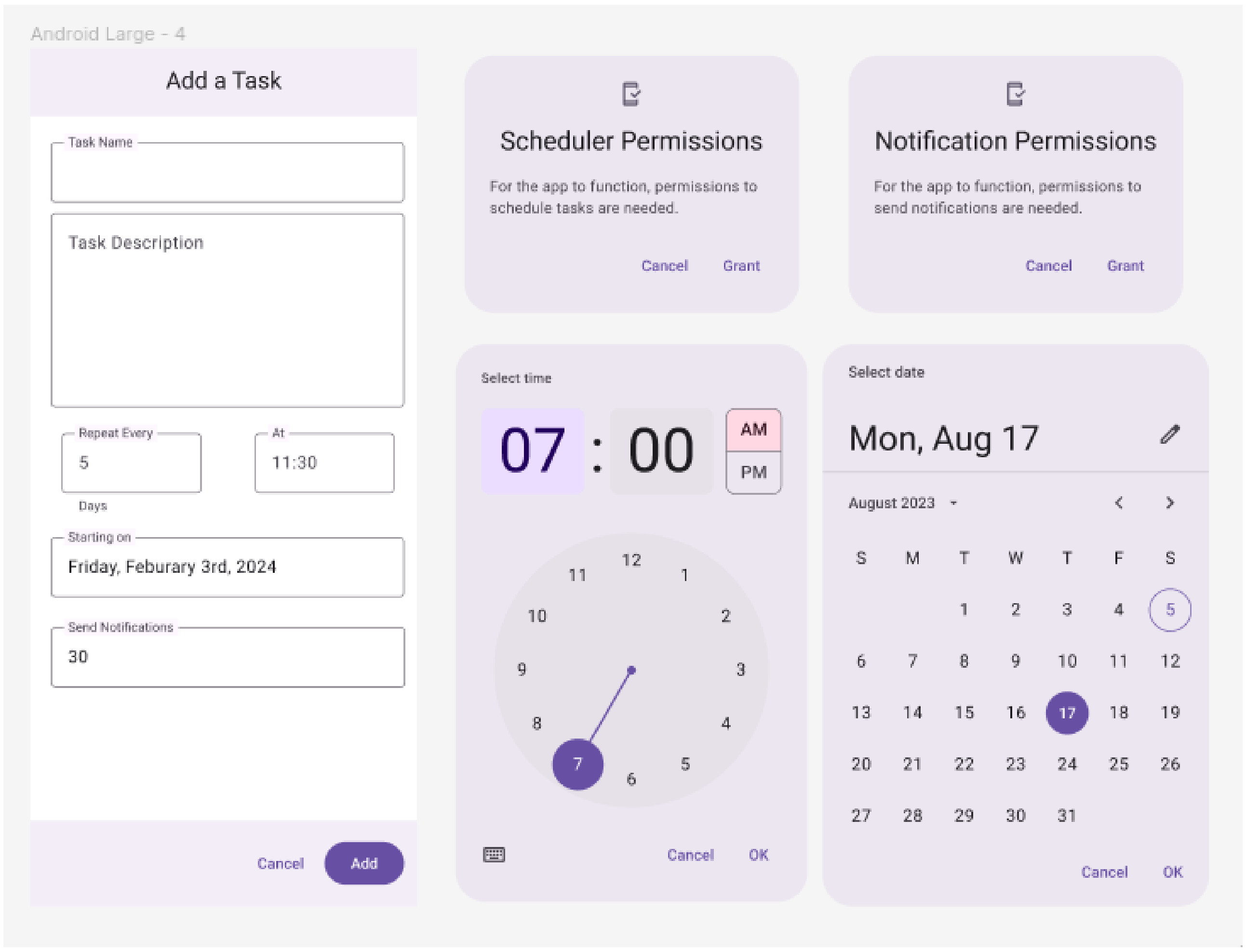
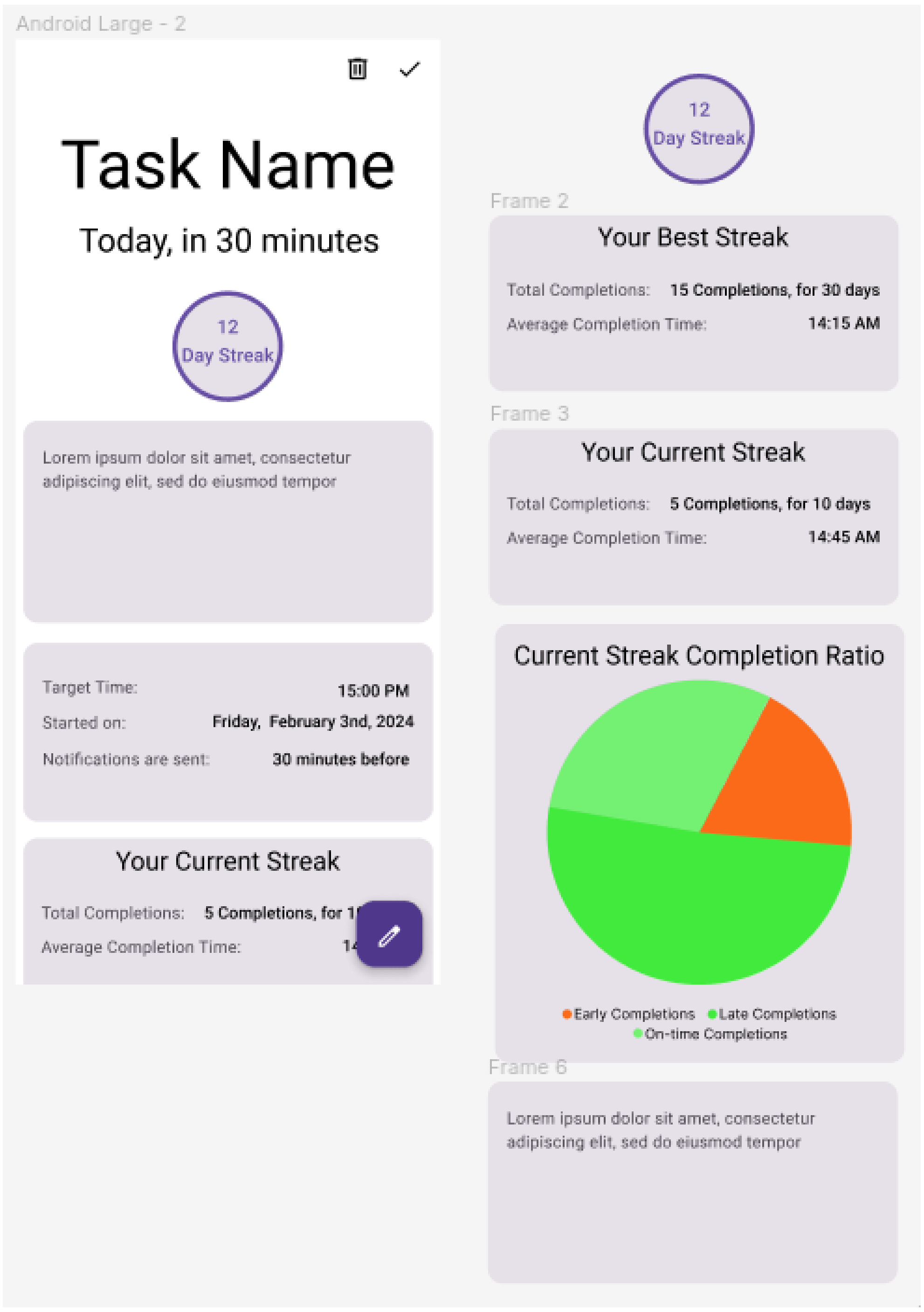


Figure 14: A Figma mock-up design of the "Add Task" menu, and the components making it up


Android Large - 2

Task Name

Today, in 30 minutes

12
Day Streak

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor

Target Time:	15:00 PM
Started on:	Friday, February 3nd, 2024
Notifications are sent:	30 minutes before

Your Current Streak

Total Completions:	5 Completions, for 1
Average Completion Time:	14

12
Day Streak

Frame 2

Your Best Streak

Total Completions:	15 Completions, for 30 days
Average Completion Time:	14:15 AM

Frame 3

Your Current Streak

Total Completions:	5 Completions, for 10 days
Average Completion Time:	14:45 AM

Current Streak Completion Ratio

● Early Completions ● Late Completions
● On-time Completions

Frame 6

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor

Figure 15: A Figma mock-up design of the "Task" menu, and the components making it up

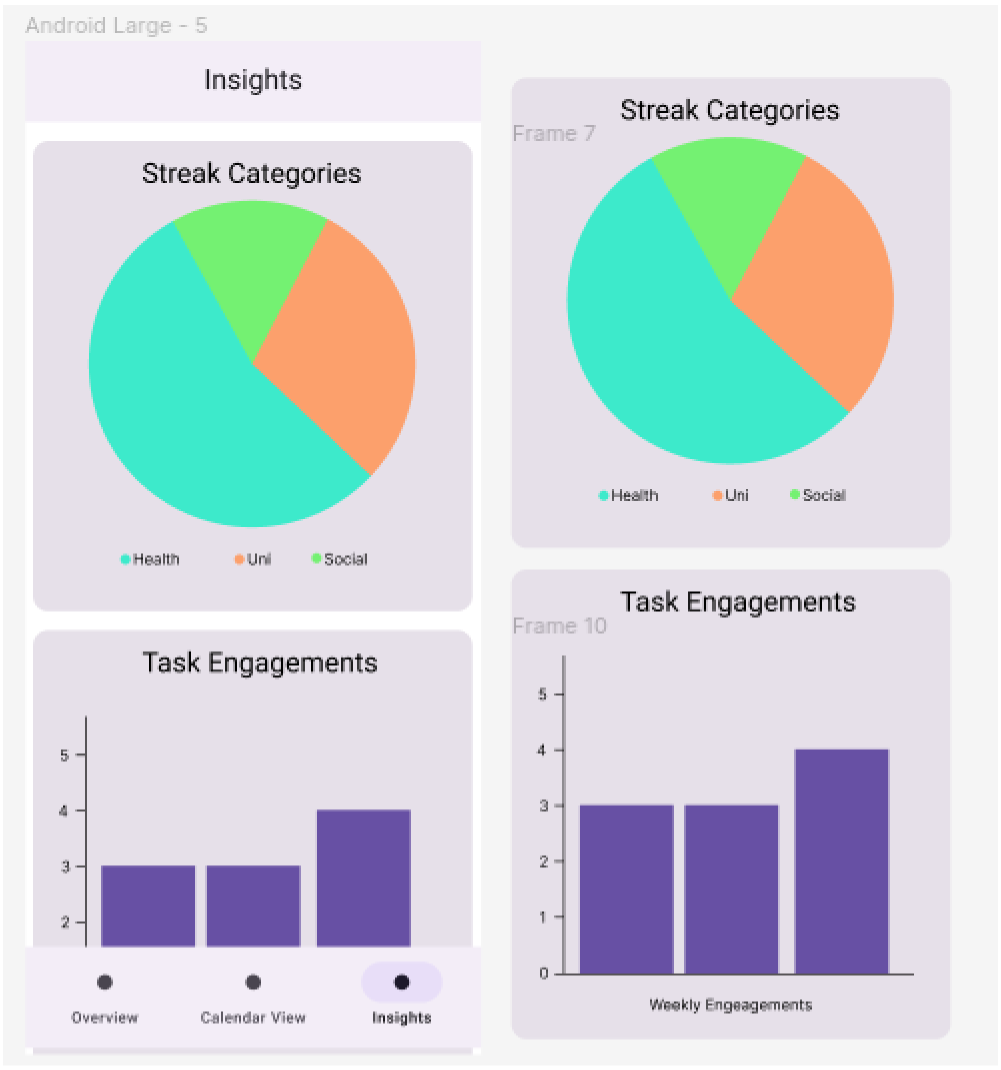


Figure 16: A Figma mock-up design of the "Insights" menu, and the components making it up

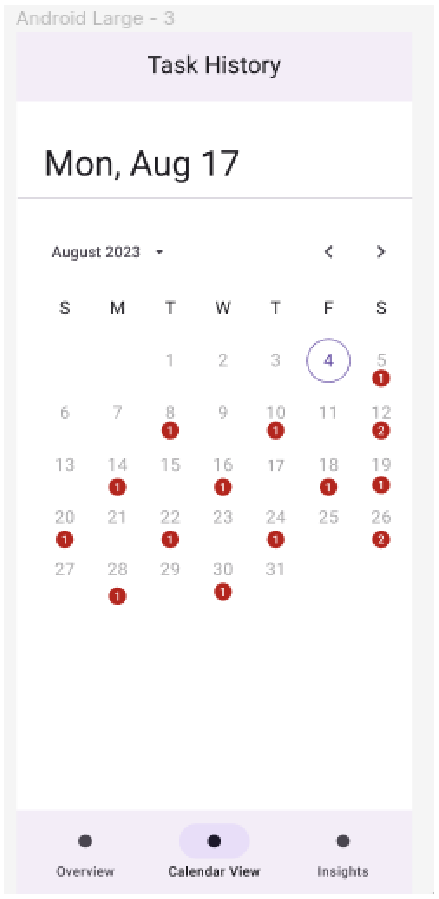


Figure 17: A Figma mock-up design of the "Calendar" menu, a single "Calendar" component.

4.6 Functional Requirements

From the use cases, functional requirements for development were created. Application features were prioritized using the MoSCoW analysis method. Features are categorized into one of four categories: Must have, Should have, Could have, and Won't have.

For clarity, these categories have been highlighted, to the key of: Must have (M), Should have (S), Could have (C), and Won't have (W).

Table 7: The functional requirements of the Application

ID	Description	Priority
1	A user can add tasks with a custom name, description, start date, period, and notification time.	M
2	Tasks can calculate and show their next completion day.	M
3	Notifications are scheduled for a task’s next completion day when the task is created.	M
4	Notifications are scheduled for the next task completion day when a task is completed.	M
5	All tasks have their notifications scheduled for the next completion day when the device is started.	M
6	Created Tasks can be viewed in detail, showing all of their corresponding data.	M
7	Tasks can be viewed in an “overview” section, sorted by completion time.	M
8	Tasks can be completed at any time on the day that they are due.	M
9	Tasks can be deleted.	M
10	Notifications show appropriate information about the task they pertain to, such as completion time.	M
11	Permissions are requested if the application lacks them.	M
12	Completion Streaks can be seen.	M
13	A Task’s information can be edited.	S
14	Completions store data about their completion time, allowing for the comparison of completion time goals and actual completion times.	S
15	Tasks can be temporarily disabled, preventing notifications and the task from showing up in upcoming task menus.	S
16	Tasks in the overview can be filtered by “due today”.	S
17	Tasks can be filtered by time until completion, overdue tasks showing in the negatives.	S
18	Tasks can be assigned a category.	S
19	Tasks can be compared per category.	S
20	If a task is not completed, an appropriate follow-up notification is sent.	S
21	Graphs show graphical information about tasks, streaks, and their completion data.	S
22	A “Calendar view” shows an overview of upcoming streak days.	S
23	Task categories can be customized.	C
24	An overview page shows graphs about task categorization.	C
25	Notifications have a “complete” follow-up action, for ease of use.	C
26	An overview page shows graphs about task completion on a per-week basis.	C
27	Data can be exported and imported from the application.	C
28	The application will have a dark and light theme.	C
29	Tasks can be shown in a widget form.	W
30	Tasks can be shown within third-party applications, such as Google Calendar.	W

5. Implementation

A user manual for the use and operation of the application has been created and placed within Appendix 2. This details how the application behaves, functions, and how the user is intended to interact with it.

Development was split into two main “Spirals”; an initial spiral, focusing on the development of “Must” functional requirements of the application, and a second spiral, focusing on implementing “Should” and “Could” requirements.

5.1 Application Modelling

5.1.1 Persistent Data Storage

To model how persistent data will be stored in the application, UML has been used to create an Entity Relationship diagram.

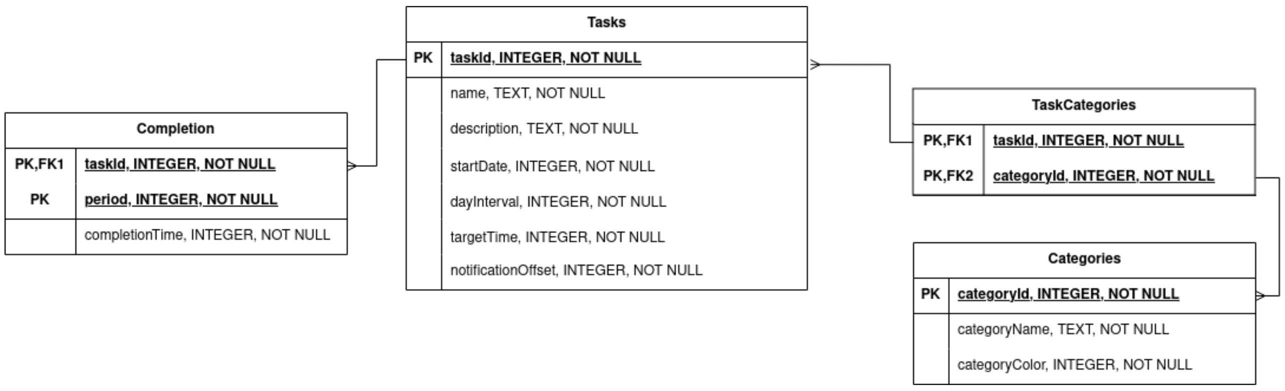
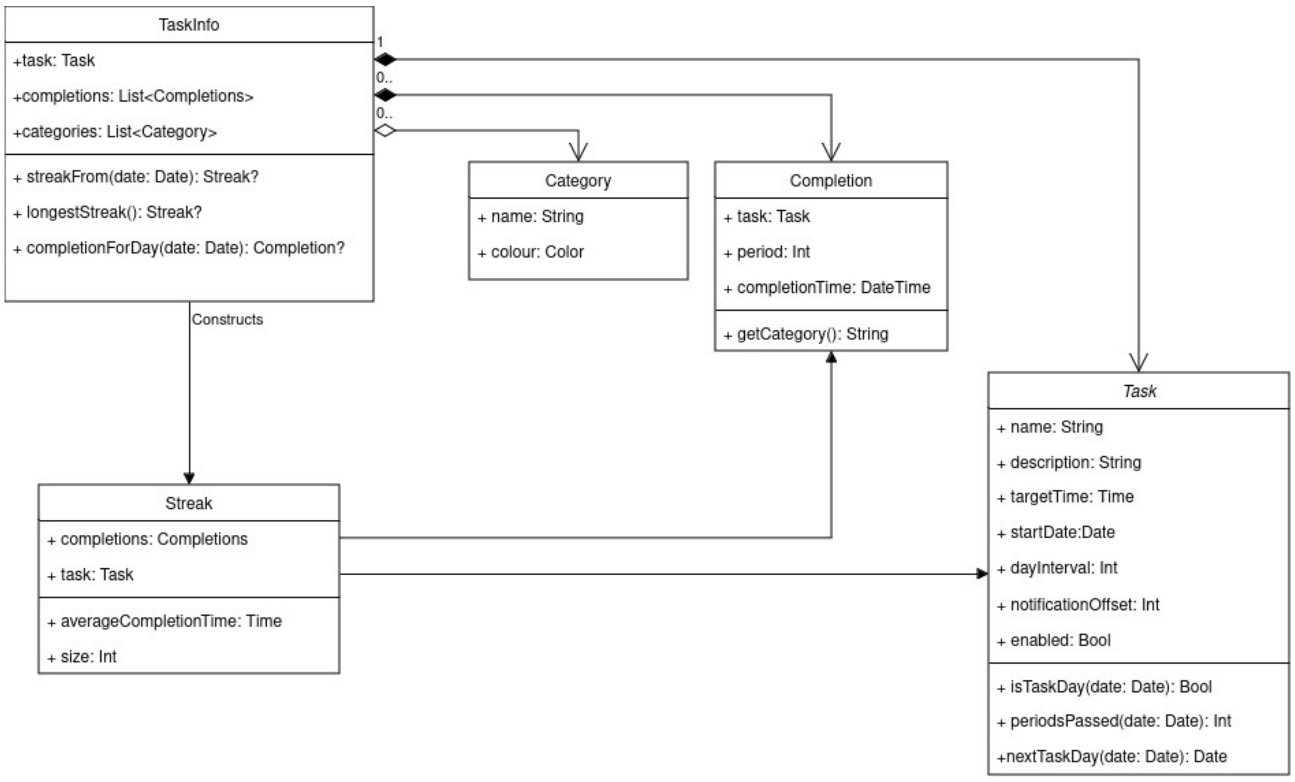

Tasks
PK
taskId, INTEGER, NOT NULL
name, TEXT, NOT NULL
description, TEXT, NOT NULL
startDate, INTEGER, NOT NULL
dayInterval, INTEGER, NOT NULL
targetTime, INTEGER, NOT NULL
notificationOffset, INTEGER, NOT NULL
Completion
PK,FK1
taskId, INTEGER, NOT NULL
PK
period, INTEGER, NOT NULL
completionTime, INTEGER, NOT NULL
TaskCategories
PK,FK1
taskId, INTEGER, NOT NULL
PK,FK2
categoryId, INTEGER, NOT NULL
Categories
PK
categoryId, INTEGER, NOT NULL
categoryName, TEXT, NOT NULL
categoryColor, INTEGER, NOT NULL

Figure 18: A UML ERD diagram modelling the application's database

5.1.2 Modelling Objects

A UML class diagram has been created to define the data entities within the application, their methods, and their relationships with other objects.



TaskInfo
+task: Task
+completions: List<Completions>
+categories: List<Category>
+ streakFrom(date: Date): Streak?
+ longestStreak(): Streak?
+ completionForDay(date: Date): Completion?
1
0..
0..
Category
+ name: String
+ colour: Color
Completion
+ task: Task
+ period: Int
+ completionTime: DateTime
+ getCategory(): String
Constructs
Task
+ name: String
+ description: String
+ targetTime: Time
+ startDate:Date
+ dayInterval: Int
+ notificationOffset: Int
+ enabled: Bool
+ isTaskDay(date: Date): Bool
+ periodsPassed(date: Date): Int
+nextTaskDay(date: Date): Date
Streak
+ completions: Completions
+ task: Task
+ averageCompletionTime: Time
+ size: Int

Figure 19: A UML Class Diagram modelling the data entities within the application

5.2 Development Decisions

Android development is a field that has evolved rapidly over the last fourteen years. As a result, there is a wide plethora of existing in-use toolkits and libraries for Android Development. The project aims to make a modern Android application, following modern android development practices.

5.2.1 Language Choice

Two categories of languages can be used to write Android applications. Cross-platform languages are used to write applications for both Android and IOS, while native languages are languages that are used to develop a platform-specific Android or IOS application.

A cross-platform language will not be considered for the project. The main appeal of cross-platform languages is that development does not need to be split between Android and IOS. An IOS application is out of scope of the project, and cross-platform languages have functionality drawbacks.

Java is the traditional tool for native Android development. Its main appeal as a language for development is that applications run on a virtual machine – the JVM. This allows for easy interoperability between different Android phones, quick build times when developing, and sandboxing, which keeps applications secure.

Modern Android Applications, however, are written in Kotlin, not Java. Kotlin is not an entirely new language from Java; instead, it is more of a refinement — Java-defined classes, libraries, and functions are interoperable with Kotlin code, and Kotlin code compiles to Java bytecode and runs inside a JVM.

Kotlin has a wide variety of significant language features that Java lacks, such as type inference and null-safety. These modern improvements result in code that is more concise and clear; 67% of professional developers[9] claim that switching to Kotlin has increased their productivity.

As of 2019, Google announced that Android development going forward will be “Kotlin first”[10]; modern Android toolkits and libraries will be specifically made to be used with Kotlin. Kotlin will be used for the project.

5.2.2 Development Environment

Android Studio is the official IDE for developing Android Applications[11]. It has a variety of features that make Android development easy, such as integration with emulators, syntax highlighting and extensive testing tools. It will be used for developing the project.

9 https://developer.android.com/kotlin/first#why
10 https://developer.android.com/kotlin/first
11 https://developer.android.com/studio/intro

5.2.3 UI development

A UI toolkit is needed to develop the application's front end. For Android applications, this has traditionally been Android Views. Views is a UI toolkit that uses an XML file to define a tree of widgets that represents the UI’s structure, similar to how HTML defines a webpage. The widgets of the UI tree are then updated with the application’s state via imperative mutations to elements of the tree.

While Views can be used for UI development, it has a variety of flaws. Because layouts are written in XML, project development in Views is divided between working on the XML layout sheets and the code used to define them. In addition, Views’ imperative nature is a significant developmental drawback; every element of a UI written in views has to have code written to configure its values and behaviour, which results in complex code being required to handle the conversion of application state to widget state. These problems combined result in boilerplate code, confusing code, and significant complexity challenges in development. As a result of Views’ shortcomings, Google created Jetpack Compose.

Compose[12] is the recommended, modern toolkit for Android UI development. The official Android documentation and the general Android Development community encourage its use - various major companies, including Reddit, Twitter, and Threads[13], have migrated to it.

Compose has a variety of advantages over Views. It is declarative, allowing UIs to be quickly built and given function with no boilerplate. It natively, intuitively supports state management. Finally, it is written in Kotlin, allowing development to be tightly integrated with the rest of the project. These advantages make Compose the best choice for the project. Composes functionality is covered in more detail in section 6.2.3.

12 https://developer.android.com/develop/ui/compose
13 https://developer.android.com/develop/ui/compose/adopt

5.2.4 Other Toolkits and Libraries

Persistent Storage

To manage the tasks, and task data, the project must use some kind of persistent storage. SQLlite is a well known, well established, file-based SQL implementation. It is the most used[14] database engine in the world – in large part, because of its deployment in Android applications.

Rooms[15] is an abstraction layer over SQLite, and is considered the modern way to handle persistent storage in Android. Rooms provides the capabilities of SQLite, but without the boilerplate that comes with converting between entries in tables and objects in code. Rooms will be used to manage task data within the project.

Dependency Injection

Hilt[16], built on top of Dagger, is the standard dependency injection manager used for android development. It is used to manage dependencies between different sections of the application. This is discussed in more detail in section 6.6.

Graphs

To implement the graphs within the application, a third party graphing library was used. Y-Charts[17], an open-source charting library, was chosen because of its native integration with Compose and its wide variety of available graph plots.

Testing

Android-studio is well integrated with Junit4[18], and so it was used for testing in section 7.

Version Control

Git and Gitlab were used to version changes and to back up the project.

14 https://www.sqlite.org/mostdeployed.html
15 https://developer.android.com/jetpack/androidx/releases/room
16 https://dagger.dev/hilt/
17 https://github.com/codeandtheory/YCharts
18 https://junit.org/junit4/

6. Development

6.1 Application Structure

The Project was structured following Android's most recent application architecture style, which is simply called "Modern App Architecture[19]". Modern App Architecture, as the name implies, is a guide to modern practices for Android development. It adapts Clean architecture[20] principles and SOLID design principles[21] for Android-specific development.

Modern App Architecture structures projects into three distinct layers, each acting as a high-level abstraction of a section of the Application. These three layers are the Presentation Layer, the Domain Layer, and the Data Layer.

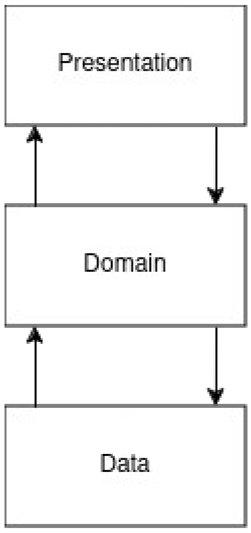


Figure 20: A diagram representing Modern Android Architecture

This layer abstraction is the backbone on which other Modern Application Architecture principles are enforced. Decomposing the application into layers ensures that components stay to a specific domain, and don't become god objects – a structural separation of concerns is enforced, keeping objects in each layer decoupled from one another.

In addition, layers encourage objects to have a single source of truth; objects "belong" to a layer, and changes cannot be made to an object outside its owner, ensuring data consistency is met.

19 https://developer.android.com/topic/architecture#modern-app-architecture
20 https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture
21 https://staff.cs.utu.fi/~jounsmed/doos_06/material/DesignPrinciplesAndPatterns.pdf

6.2 The Presentation Layer

The presentation layer is the Application's front, user-facing layer, and its role encapsulates everything related to the User Interface of the Application.

At the root of the presentation layer is the MainActivity. Projects in Compose have a single-activity architecture; the entire User Interface is contained within an Activity, and all typical User Interactions occur through the Activity.

6.2.1 Screens

A "screen" is a category within the model layer that represents a single screen of the application. To understand the makeup of a screen, it is first important to understand MVVM Architecture.

6.2.2 Model-View-ViewModel Architecture

Model-View-ViewModel, or MVVM, is an architecture pattern used within each Screen that splits a screen into three levels.

The View Level

The View level of MVVM is the level responsible for defining the UI hierarchy seen by the user. It is made up of a Composable object representing the screen, which is in turn made up of other Composable objects.

The View layer provides the view model with event calls as the user interacts with it, which updates the screen's state.

The updated state data is then "collected" from the view, and the view is recomposed with the new state data.

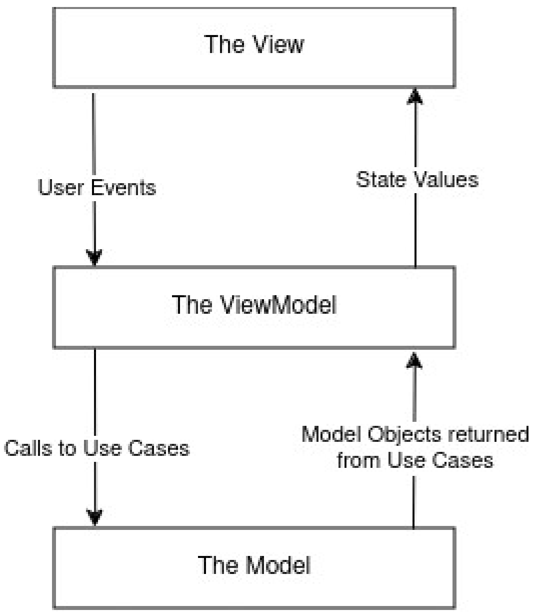


Figure 21: A diagram representing MVVM

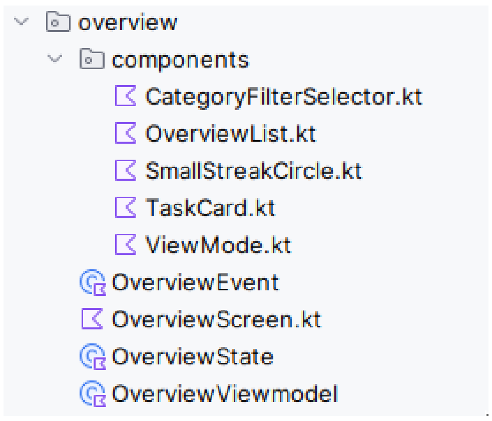
The ViewModel Level

The ViewModel receives events from the view, then logically handles and updates the screen's state. "Advanced" application logic, such as logic that requires interfacing with application data, is represented by calls to the Model.

The Model Level

The Model is the lowest level of MVVM, representing the abstraction of the application's business logic.

6.2.3 The Overview Screen

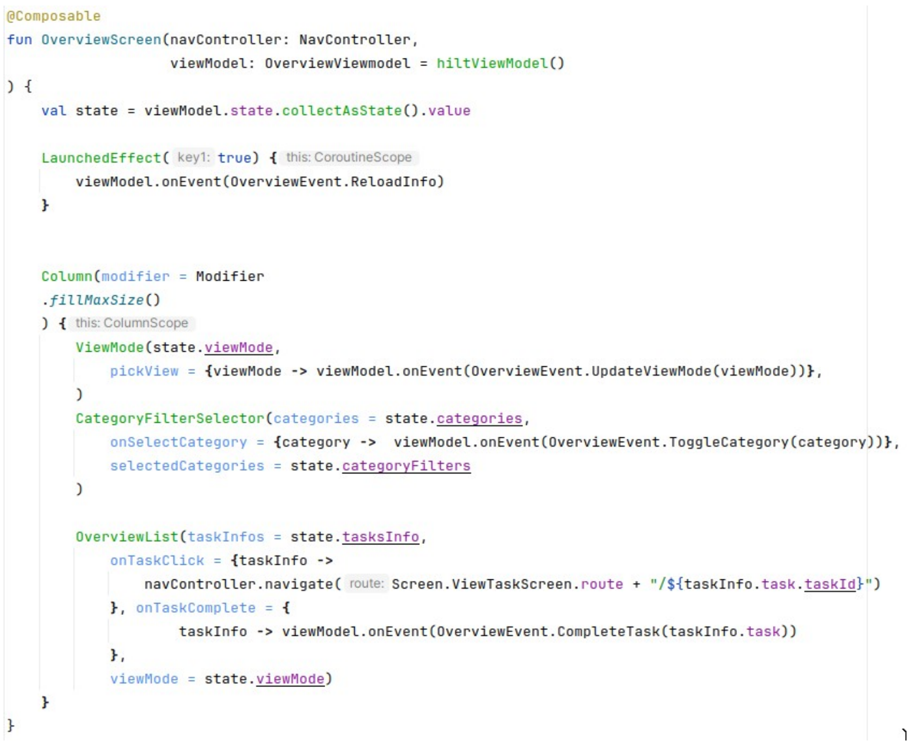


```
overview
    components
        CategoryFilterSelector.kt
        OverviewList.kt
        SmallStreakCircle.kt
        TaskCard.kt
        ViewMode.kt
    OverviewEvent
    OverviewScreen.kt
    OverviewState
    OverviewViewmodel
```

Figure 22: The Structure of the "Overview" screen.

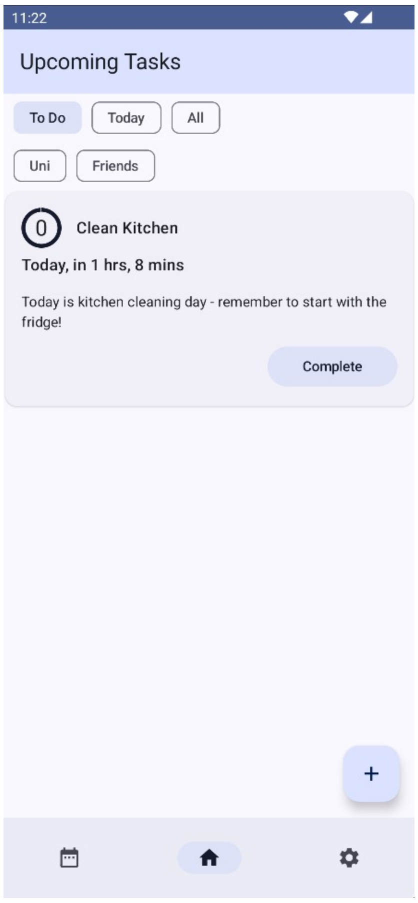
For the sake of example, the structure and different levels of MVVM within one of the application's screen of the application - the overview screen - have been explored in detail.

The View Layer



```
@Composable
fun OverviewScreen(navController: NavController,
                   viewModel: OverviewViewmodel = hiltViewModel()
) {
    val state = viewModel.state.collectAsState().value

    LaunchedEffect( key1: true) { this: CoroutineScope
        viewModel.onEvent(OverviewEvent.ReloadInfo)
    }


    Column(modifier = Modifier
    .fillMaxSize()
    ) { this: ColumnScope
        ViewMode(state.viewMode,
            pickView = {viewMode -> viewModel.onEvent(OverviewEvent.UpdateViewMode(viewMode))},
        )
        CategoryFilterSelector(categories = state.categories,
            onSelectCategory = {category ->  viewModel.onEvent(OverviewEvent.ToggleCategory(category))},
            selectedCategories = state.categoryFilters
        )

        OverviewList(taskInfos = state.tasksInfo,
            onTaskClick = {taskInfo ->
                navController.navigate( route: Screen.ViewTaskScreen.route + "/${taskInfo.task.taskId}")
            }, onTaskComplete = {
                    taskInfo -> viewModel.onEvent(OverviewEvent.CompleteTask(taskInfo.task))
            },
            viewMode = state.viewMode)
    }
}
```

Figure 24: A simplified version of OverviewScreen.kt



Clean Kitchen

Today, in 1 hrs, 8 mins

Today is kitchen cleaning day - remember to start with the fridge!

Figure 23: The "Overview" screen, as seen by the user

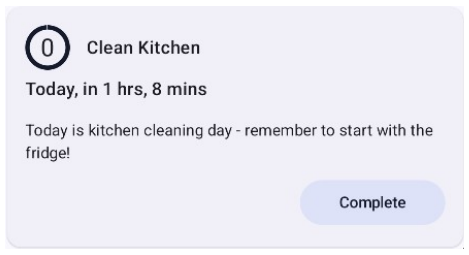
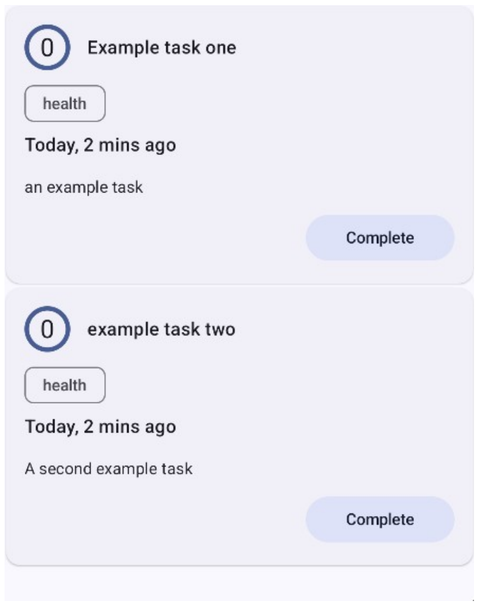
OverviewScreen.kt is a file which contains the Composable function that represents the "View" Layer of the screen. The Composable function describes the UI declaratively, taking collected state values from the viewModel and using them to define Components.

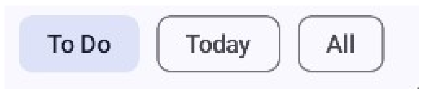
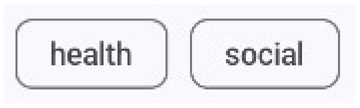
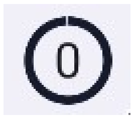
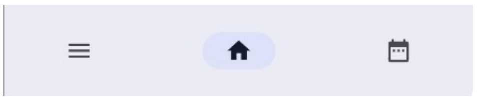
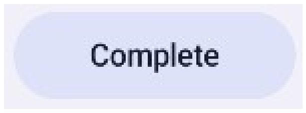
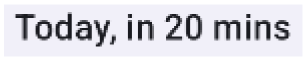
Compose and Recomposition
While the composition of a compose function looks linear, it is not – any Composable function can be recomposed independently of another, at any time, in any order.

This is an intentional feature of Compose – Composables only need to be recomposed when there is a change in their parameters, and thus a change in application state. Understanding this is important for understanding general development with Compose; no changes to application state should happen within a Composable, as recomposition is unpredictable. Instead, state changes are handled by events.

Components
Components are used to design custom UI elements in a modular way that maintains a separation of concerns between each element. The application makes use of a variety of custom components - some shared throughout the application, while others are screen-specific. The components used in the Overview are listed below:

Table 8: Components present within the overview screen, including in-application screenshots

Component	**Purpose**	**In-App appearance**
TaskCard	The card used to represent a task in the overview.	 0 Clean Kitchen Today, in 1 hrs, 8 mins Today is kitchen cleaning day - remember to start with the fridge!
OverviewList	The scroll-able container of TaskCards.	 0 Example task one health Today, 2 mins ago an example task 0 example task two health Today, 2 mins ago A second example task

ViewModeSelector	The selector for the “time” filter.	
CategoryFilterSelector	The selector for what categories should be used to filter the shown cards.	
SmallStreakCircle	A circle showing how long is left until the next streak, with the user’s current streak count within.	
AppNavigation	A component used to manage navigation between the application’s main three screens.	
TaskCompletionButton	Used to complete tasks, and to represent completed tasks.	
TaskTime	Used to represent the time until the next task completion.	

The ViewModel

The ViewModel is represented by OverviewViewmodel.kt, a file containing the ViewModel class for the screen.

Overview Events

The view communicates with the viewModel via Events. These events are defined on a per-screen basis – the events for the Overview screen are defined within overviewEvents.kt.

Table 9: Events present within the Overview Screen

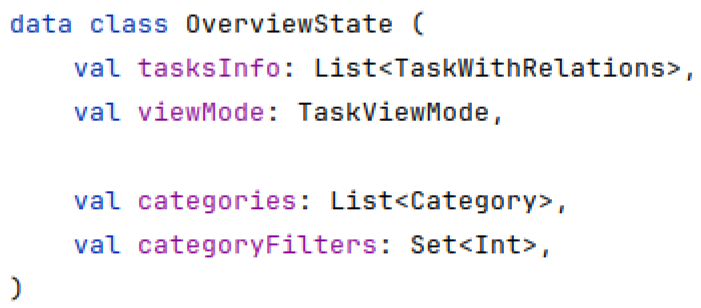
Event Name	**Event Arguments**	**Event Description**
Complete Task	The task that has been completed.	Fired when a task is completed.
Reload Information	None.	Fired when the screen is opened or navigated back to.
Toggle Category	The category that is being toggled.	Fired when a category is selected by the category filter selector.
Update View mode	The newly selected view mode.	Fired when a view mode is selected by the view mode selector.

Overview State
The ViewModel stores state within a custom data class, OverviewState.kt. Every time the state is updated, its new values are communicated back to the view, and the appropriate Composables are recomposed.

State is communicated with the View by a “Mutable State Flow” – an asynchronous object that is re-collected in the view each time its value is updated by the ViewModel. This updates the parameters of Composables, triggering recomposition when appropriate.

Table 10: State values within the overview screen

Value	Description
Tasks	The tasks being shown in the Overview list.
View Mode	The View Mode being used to filter the tasks in the task list.
Categories	The categories that can be used to filter the tasks in the overview list.
Category Filters	The currently active category filters on the tasks in the overview list.



```
data class OverviewState (
    val tasksInfo: List<TaskWithRelations>,
    val viewMode: TaskViewMode,

    val categories: List<Category>,
    val categoryFilters: Set<Int>,
)
```

Figure 25: The Overview State data object

The Model
The view makes calls to use cases within the model layer. This exists in the next layer of Modern App Architecture – the domain layer.

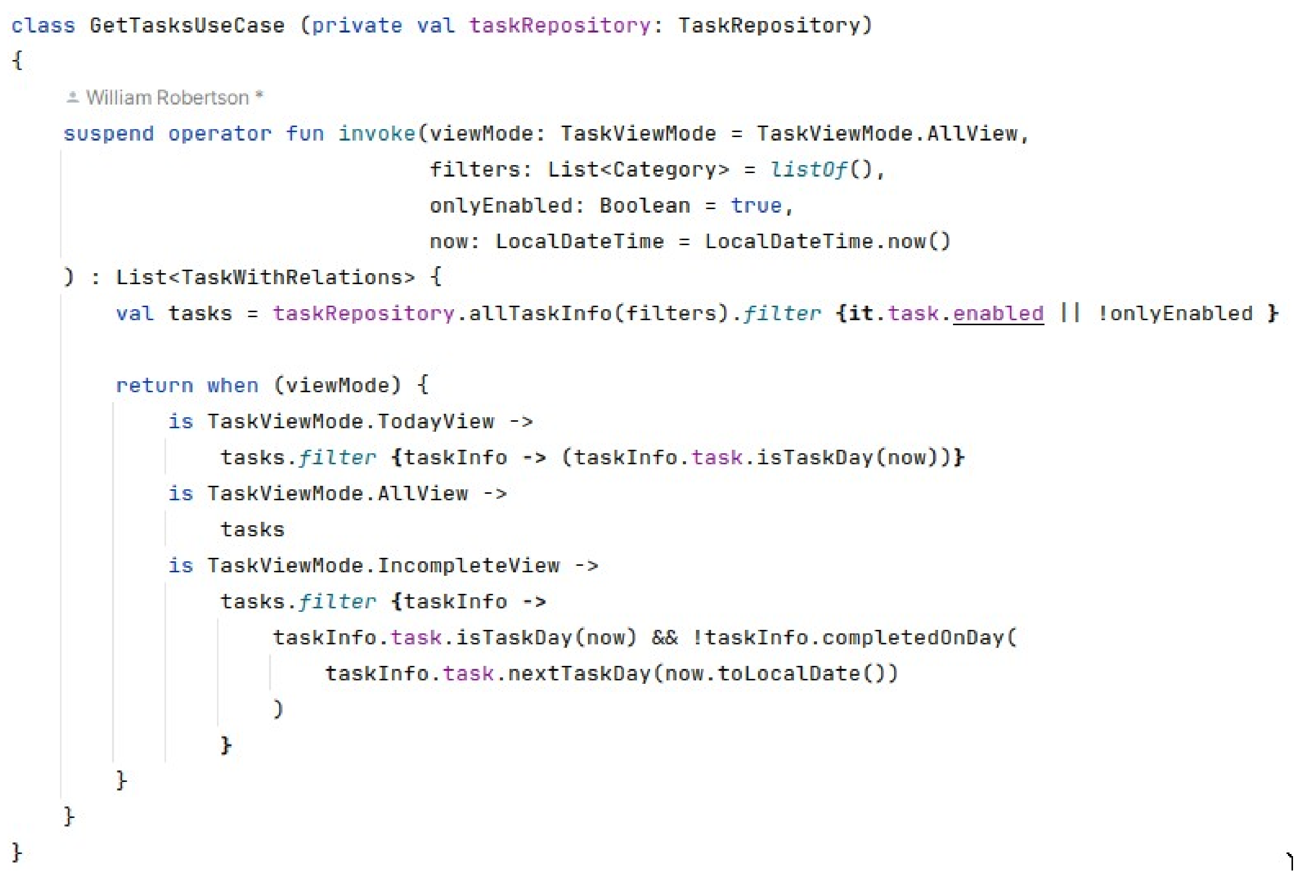
6.3 The Domain Layer

The domain layer represents the business logic of the application – functions used throughout the application, modelling objects, and repository interfaces.

6.3.1 Use Cases

Use cases within the domain layer should not be confused with the Application use cases defined within section four – they are unrelated concepts that just happen to share a naming convention.

A “Modern Android Architecture” use case exists within the Domain Layer, and represents a single function of the application. They handle an application’s business logic and interface with the data layer. Use cases are stateless, self-contained, and self-describing. They keep application functions simple and understandable, modular and usable from any point within the application. This modularity also makes testing easy - a Unit Test can be written for each Use Case.



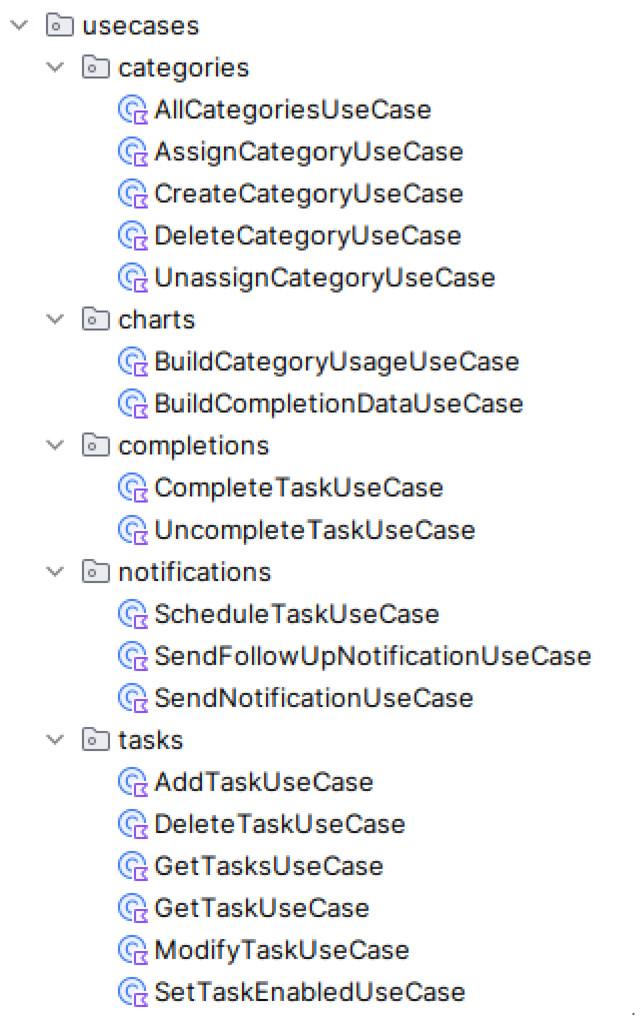
William Robertson *

```
class GetTasksUseCase (private val taskRepository: TaskRepository)
{
    suspend operator fun invoke(viewMode: TaskViewMode = TaskViewMode.AllView,
                                filters: List<Category> = listOf(),
                                onlyEnabled: Boolean = true,
                                now: LocalDateTime = LocalDateTime.now()
    ) : List<TaskWithRelations> {
        val tasks = taskRepository.allTaskInfo(filters).filter {it.task.enabled || !onlyEnabled }

        return when (viewMode) {
            is TaskViewMode.TodayView ->
                tasks.filter {taskInfo -> (taskInfo.task.isTaskDay(now))}
            is TaskViewMode.AllView ->
                tasks
            is TaskViewMode.IncompleteView ->
                tasks.filter {taskInfo ->
                    taskInfo.task.isTaskDay(now) && !taskInfo.completedOnDay(
                        taskInfo.task.nextTaskDay(now.toLocalDate())
                    )
                }
        }
    }
}
```

Figure 26: An example of a use case within the application

Within the project, the Use Cases handle relatively benign logic, such as error checking before adding a task, and more advanced logic, such as handling notifications and task scheduling.

Use Cases within the application



```
usecases
  categories
    AllCategoriesUseCase
    AssignCategoryUseCase
    CreateCategoryUseCase
    DeleteCategoryUseCase
    UnassignCategoryUseCase
  charts
    BuildCategoryUsageUseCase
    BuildCompletionDataUseCase
  completions
    CompleteTaskUseCase
    UncompleteTaskUseCase
  notifications
    ScheduleTaskUseCase
    SendFollowUpNotificationUseCase
    SendNotificationUseCase
  tasks
    AddTaskUseCase
    DeleteTaskUseCase
    GetTasksUseCase
    GetTaskUseCase
    ModifyTaskUseCase
    SetTaskEnabledUseCase
```

Figure 27: All of the use cases present within the application

6.3.2 Modelling Objects

Modelling objects are used to represent entities within the project. They are used to represent the entities defined in section 5.1.2. Modelling objects are either created programically, by another object or a use case, or are created to represent data from a repository.

6.3.3 Repositories

Repositories act as an intermediate between the data layer and the domain layer. They manage data providers—in the case of the project, the Rooms / SQLite database that the application uses for storage. The domain layer provides interfaces, defining the methods used to access data, while the Data layer provides an actual implementation, connecting a data source to the repository.

This intermediary implementation allows for the separation of data and its accessor object; a repository can be easily modified for testing or re-implemented to work using a different provider for a back-end restructuring.

6.4 The Data Layer

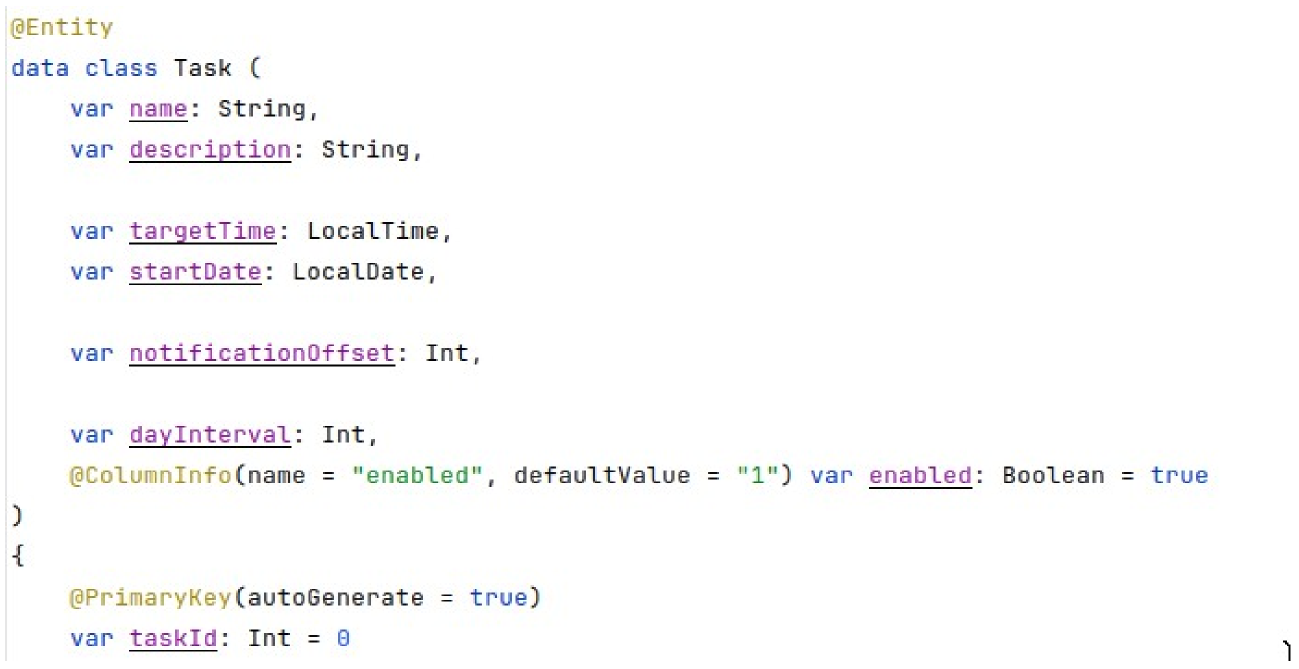
The data layer is responsible for the persistent data within the application, through the implementation of repositories. Repository implementations are simple, serving as a link between the interface defined in the domain layer and Rooms.

6.4.1 Rooms

Rooms is a toolkit used to implement and interact with the database defined in section 5.1.1. It does this by defining a database object, which in turn references Entities, a Database Access Object (DAO) used to make queries, and type converters needed to serialize data into an SQL-compatible form.

Entities

Rooms allows for any class within Kotlin to be declared as a database entity. This allows for data to be handled in an intuitive way; the columns of a table are directly associated with a Kotlin object, which is returned by the DAO.

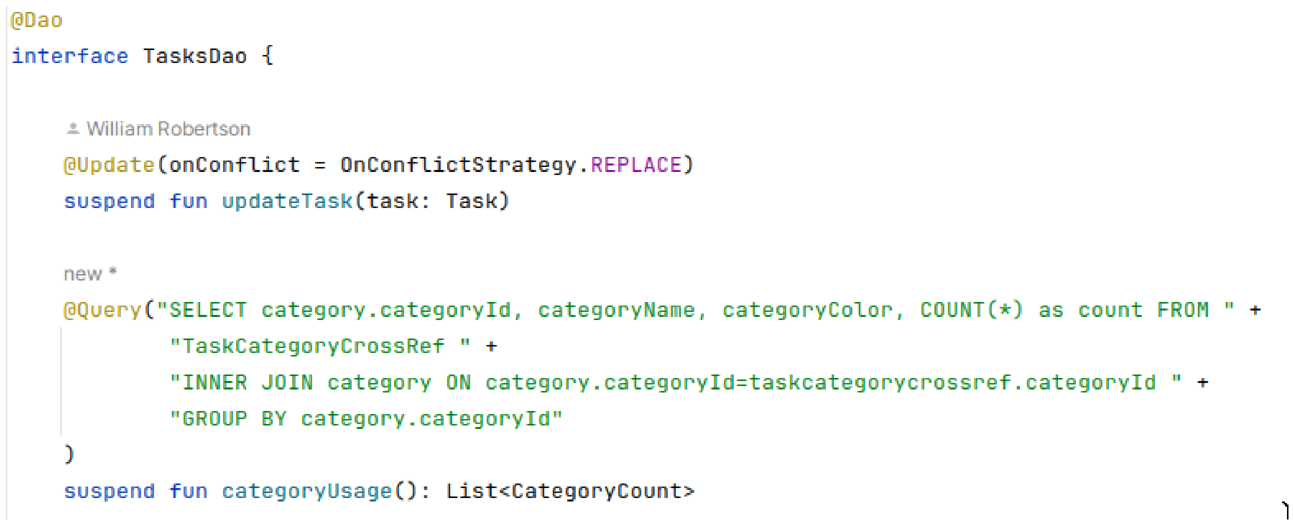


```
@Entity
data class Task (
    var name: String,
    var description: String,

    var targetTime: LocalTime,
    var startDate: LocalDate,

    var notificationOffset: Int,

    var dayInterval: Int,
    @ColumnInfo(name = "enabled", defaultValue = "1") var enabled: Boolean = true
)
{
    @PrimaryKey(autoGenerate = true)
    var taskId: Int = 0
```

Figure 28: An example of a Rooms entity defined within the application

The Database Access Object

The DAO allows for SQL queries to be easily defined with no boilerplate – predefined “database entity” objects are populated with the results of queries.



```
@Dao
interface TasksDao {

    William Robertson
    @Update(onConflict = OnConflictStrategy.REPLACE)
    suspend fun updateTask(task: Task)

    new *
    @Query("SELECT category.categoryId, categoryName, categoryColor, COUNT(*) as count FROM " +
            "TaskCategoryCrossRef " +
            "INNER JOIN category ON category.categoryId=taskcategorycrossref.categoryId " +
            "GROUP BY category.categoryId"
    )
    suspend fun categoryUsage(): List<CategoryCount>
```

Figure 29: Two Query functions within the DAO, one simple, one more involved

Type Converters

Rooms can automatically handle converting standard data types into table parameters and back. It cannot, however, automatically handle other objects, such as the LocalDate or LocalTime used to represent the startDate and targetTime of a task.

Rooms cleanly manages this by giving databases a “Converter” object, which allows for methods to be defined to convert them to and from an SQL-serialisable format.

6.5 Application Navigation

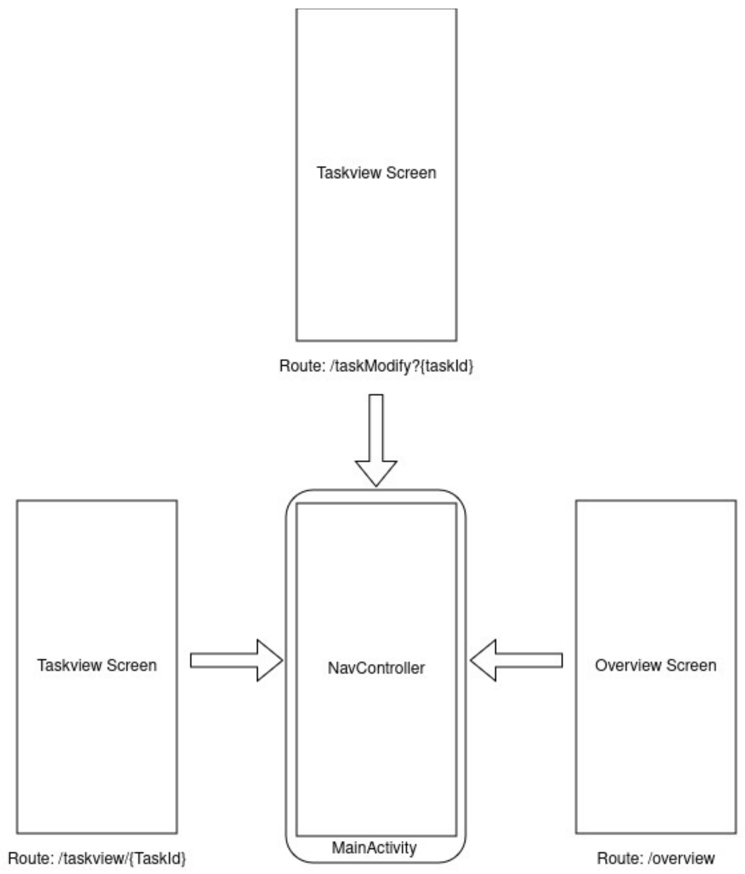


Figure 30: The Relationship between the NavController and the screens it presents

Within the Project, navigation is handled by a NavController — an object used to manage navigation through screens, in a manner akin to URLs. Composable Screens are stored with a string path called a route. This route can also take optional and required arguments, which allow data transfer between screens.

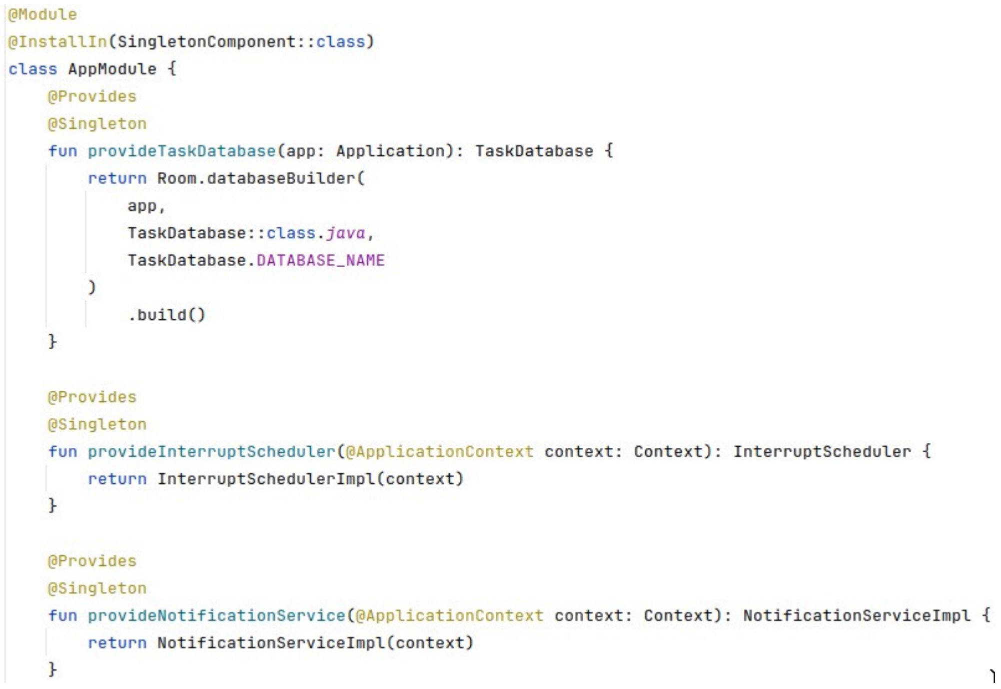
Using a NavController to manage navigation allows for the Android's "back stack" (swiping from the screen edge to go to the previous menu) to be seamlessly integrated with the application, simplifying navigation and making it feel native.

6.6 Lifecycle Management

Lifecycle refers to the creation and destruction of objects within the application. Use cases and repositories need to be accessed from various points within the application. Creating new objects repeatedly would be needlessly hard on resources, and passing objects between different areas of the application would add decent complexity to the application.

Dependency injection is a commonly used programming technique that is suited for modern android development – it is used to provide the connectivity between the divided use-cases, MVVMs, and repositories.

Dependency Injection provides the constructors of various components of the application with per-initialized, reusable objects. This is accomplished using Hilt, Android’s recommended[22] dependency injection library.



```
@Module
@InstallIn(SingletonComponent::class)
class AppModule {
    @Provides
    @Singleton
    fun provideTaskDatabase(app: Application): TaskDatabase {
        return Room.databaseBuilder(
            app,
            TaskDatabase::class.java,
            TaskDatabase.DATABASE_NAME
        )
            .build()
    }

    @Provides
    @Singleton
    fun provideInterruptScheduler(@ApplicationContext context: Context): InterruptScheduler {
        return InterruptSchedulerImpl(context)
    }

    @Provides
    @Singleton
    fun provideNotificationService(@ApplicationContext context: Context): NotificationServiceImpl {
        return NotificationServiceImpl(context)
    }
```

Figure 31: Dependencies being Injected into the project Using Hilt

Dependency Injection has the added benefit that it makes testing individual components of the application simple; Dependencies can be manually provided by an Application Test.

22 https://developer.android.com/training/dependency-injection/hilt-android

6.7 Y-plot Graphs

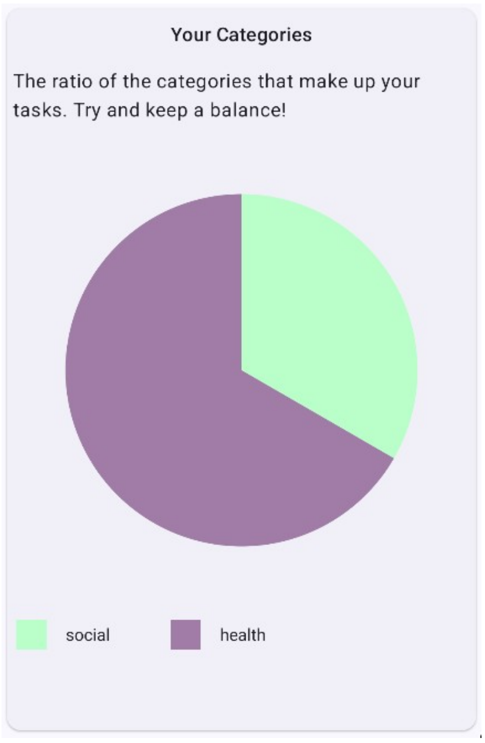

Your Categories
The ratio of the categories that make up your tasks. Try and keep a balance!
social
health

Figure 32: An example of a Y-plot graph being used within the application

Y-Plot graphs is a library that provides composables that can be used to graphically represent data within the application. It is used to present pie charts about completion times and task categorization. A use case to process the data exists within the Domain layer, which is invoked by the viewModel and presented to the composable within the presentation layer.

6.8 Interaction with the Android System

6.8.1 Android Services

The application makes use of two Android System services to function. These are the Scheduler Service, and the Notification Service. The scheduler service is used to schedule notifications for specific times, and the notification service is used to send the actual notification.

As a boundary between the system and the application, interfaces for interacting with these system components are stored within the presentation layer. The logic of scheduling and sending these notifications is handled within a use case.

6.8.2 Broadcast Receivers

Broadcast receivers are another system-application boundary, and are used to run code after a specific system event, such as the firing of a scheduled notification, or re-scheduling all scheduled tasks when the device is restarted.

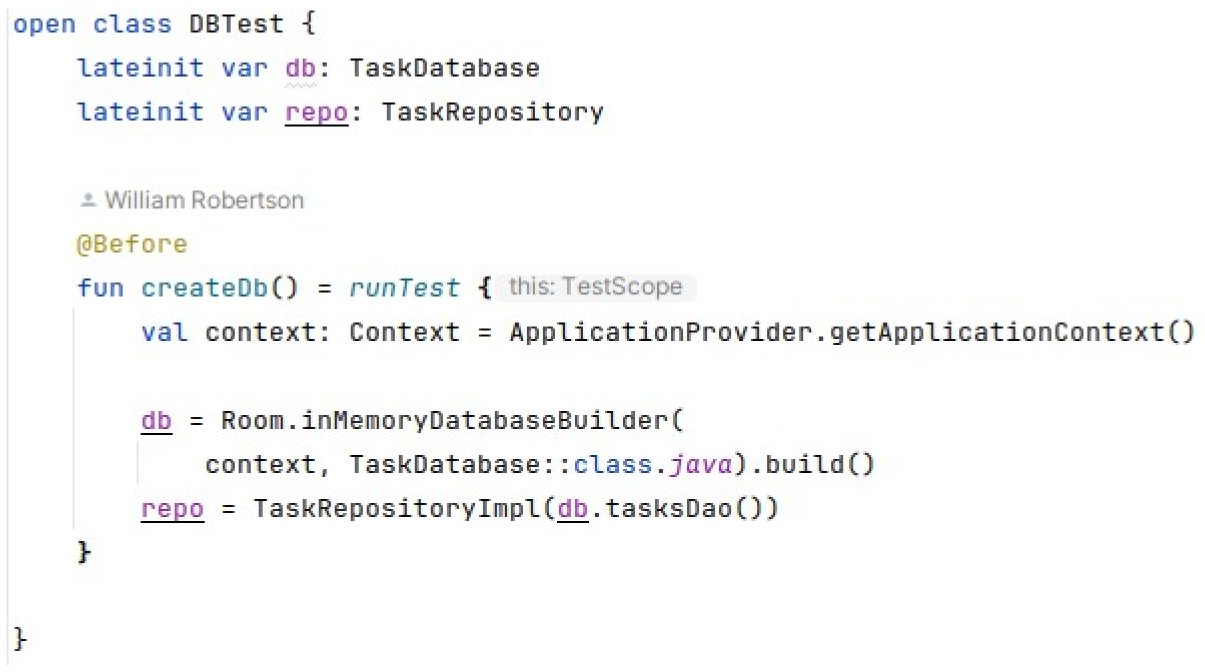
7. Testing

7.1 Developmental Testing

Application tests were used throughout the development of the entire application to ensure that the application was functioning as intended. The application was developed backend-first, so features were written to tests.

The testing table for the UI tests, the Functional Tests, and the Unit tests of the application can all be found in Appendix 3.

7.1.1 Instrumented Testing



```
open class DBTest {
    lateinit var db: TaskDatabase
    lateinit var repo: TaskRepository

    William Robertson
    @Before
    fun createDb() = runTest { this: TestScope
        val context: Context = ApplicationProvider.getApplicationContext()

        db = Room.inMemoryDatabaseBuilder(
            context, TaskDatabase::class.java).build()
        repo = TaskRepositoryImpl(db.tasksDao())
    }

}
```

Figure 33: Instrumented testing allowing for Database interactions in the test environment

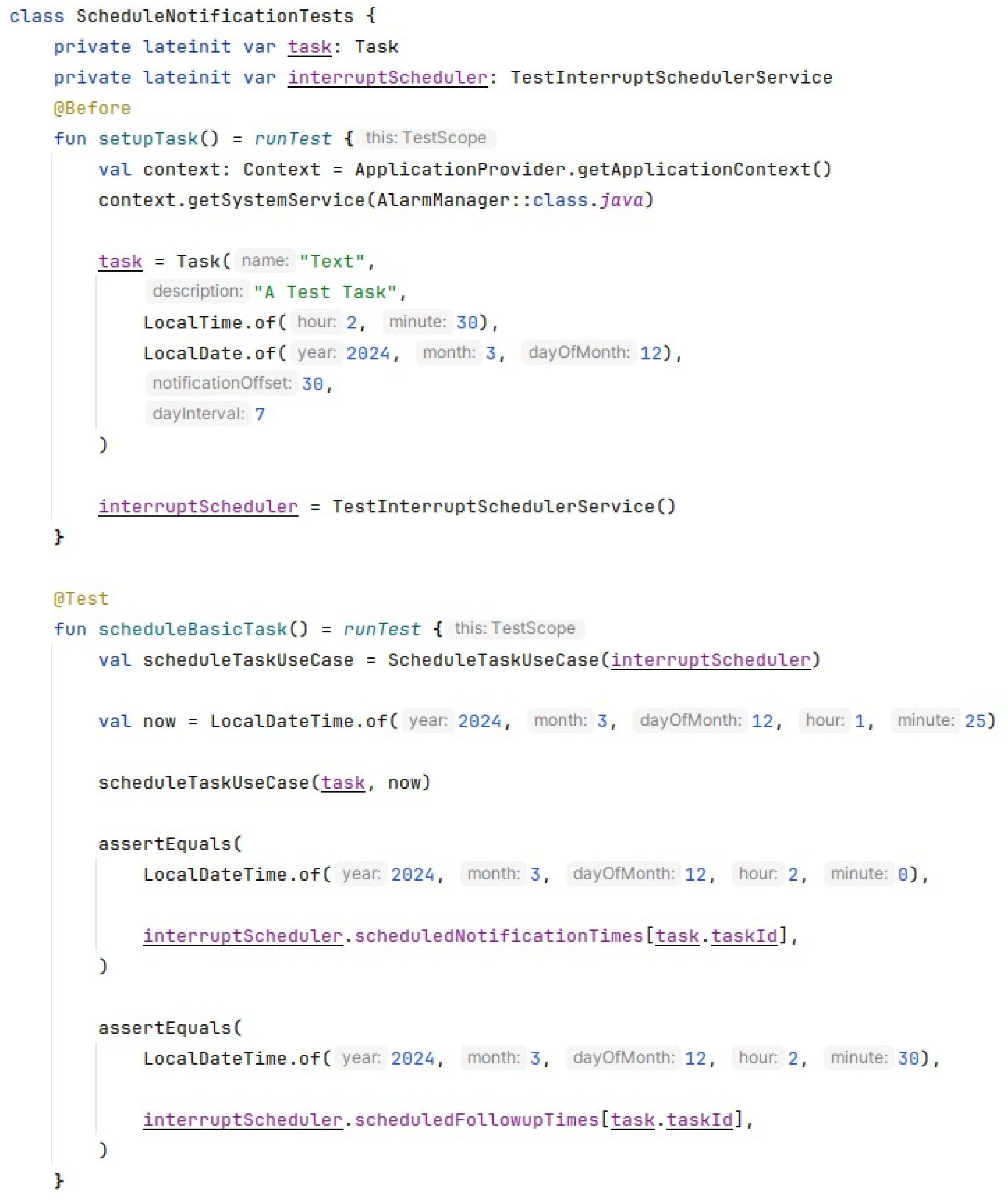
Instrumented testing is a form of testing that takes place on-device. Because Android applications are written to be run on android, this is required for most testing to take place - Rooms, and thus the application's data-layer, requires a device to function, even in a test environment on a transient database.

7.1.2 Unit Tests

Android Studio allows for integrated, instrumented unit tests to be easily performed with Junit 4.

Unit testing refers to the testing of individual components of the application. Because of the nature of Modern Android Architecture Use Cases and Dependency Injection, the application is incredibly well suited to Unit testing – each use case can have a variety of unit tests written to validate its data.

Unit tests were also written to verify the functioning of some of the application's model objects.



```
class ScheduleNotificationTests {
    private lateinit var task: Task
    private lateinit var interruptScheduler: TestInterruptSchedulerService
    @Before
    fun setupTask() = runTest { this: TestScope
        val context: Context = ApplicationProvider.getApplicationContext()
        context.getSystemService(AlarmManager::class.java)

        task = Task( name: "Text",
            description: "A Test Task",
            LocalTime.of( hour: 2,  minute: 30),
            LocalDate.of( year: 2024,  month: 3,  dayOfMonth: 12),
            notificationOffset: 30,
            dayInterval: 7
        )

        interruptScheduler = TestInterruptSchedulerService()
    }

    @Test
    fun scheduleBasicTask() = runTest { this: TestScope
        val scheduleTaskUseCase = ScheduleTaskUseCase(interruptScheduler)

        val now = LocalDateTime.of( year: 2024,  month: 3,  dayOfMonth: 12,  hour: 1,  minute: 25)

        scheduleTaskUseCase(task, now)

        assertEquals(
            LocalDateTime.of( year: 2024,  month: 3,  dayOfMonth: 12,  hour: 2,  minute: 0),

            interruptScheduler.scheduledNotificationTimes[task.taskId],
        )

        assertEquals(
            LocalDateTime.of( year: 2024,  month: 3,  dayOfMonth: 12,  hour: 2,  minute: 30),

            interruptScheduler.scheduledFollowupTimes[task.taskId],
        )
    }
```

Figure 34: A Junit Unit Test for the Application's "Schedule Notification" Use Case

7.1.3 UI Testing

Each time a feature was implemented into the application's UI, manual tests for its intended function were written and then carried out, to ensure it functioned as intended.

7.1.4 Functional Testing

Manual testing against the application's functional requirements was performed at the end of the project, to comparatively evaluate it against the initial development goals.

7.2 User Testing

User-testing was not considered to be within the scope of the project. In addition to the problems raised in Ethics (Section 3.5), to evaluate the application's habit building effectiveness would require a lengthy testing period; The development of habit is a process that takes a long time.

However, to prepare for potential future testing of the application, an initial questionnaire has been created (Appendix 4). The questionnaire details quantitative and qualitative questions that a user should be asked after trailing the tool for a sufficiently long period, such as a few months. The questions pertain to the general usability of the tool, and evaluate if the tool has been successful in forming a habit.

8. Evaluation

The application successfully implemented the researched engagement techniques defined by the project's functional requirements. All the application's "Must" and "Should" MOSCOW functional requirements were implemented, with their functionality checked by testing. The created application has all the features it needed to carry out its intended purpose.

This is largely thanks to the project being well managed; frequent, routine supervisor meetings ensured that the project stayed on course throughout development, with the project plan being adjusted when necessary.

However, not all the application's functional requirements could be implemented. Three "Could" functional requirements were missed (Appendix 3). This was because the time required to learn Kotlin, Compose, and Android Development as a whole was under-estimated in the initial project plan: The plan had to be adjusted to compromise, and it was decided that some of the application's "Could" requirements would be intentionally left unimplemented.

Because week view charts in the insight menu weren't implemented, the insight menu, while offering functionality, feels empty. The same is true for the settings menu – the only option within the settings menu is an option to create and delete task categories. Overall, the application's missing features give areas of it an "unpolished" feel. If there was more time towards the end of the project, or if this was considered earlier on in development, time could have been allocated to neatening up the application, redesigning the User Interface to be better suited to the specific features that did get implemented.

8.1 Future Development

8.1.1 User Testing

The project is theoretically proven and backed up by research, but without testing, its effectiveness cannot be measured. User testing such as the testing proposed in section 7.2 would also allow for feedback on the application's ability to form routines, and would allow for a critical assessment of the application's UI and features. This makes it an important point for future development.

8.1.2 Streak SQL Processing

Initially, the processing of Streak data was planned to occur as an SQLite query. However, due to the technical limitations of SQLite within Android, this implementation had to be scrapped; because the version of SQLite used by Android Applications is provided by the device the application is being run on, Android Studio assumes an older version of SQLite is being used. In this version, SQLite Window Functions are not supported, which were necessary to efficiently process Streak Data on the Database.

As a result, Streak data processing had to be written within Kotlin, a serviceable, but slightly less elegant solution. If Android Studio were to support a later version of SQLite, the implementation of the streak query in Appendix 5 would be another potential point of future development.

8.2 Conclusion

This project set out with the goal of creating an application that supported a user's routines, helping them become habits. Engagement techniques were researched from existing literature, their strengths and weaknesses assessed, and their usage in existing applications studied. A modern Android application implementing these techniques was then planned and developed.

The developmental testing for the application shows that these engagement techniques were effectively implemented; the project successfully achieved its goal. The next step for the application would be a user testing period, to determine the application's usability and effectiveness at building habits.