A Web Application for Recording a Personal Diary with Spaced Repetition Training

Nyasha Sibanda

MSc Computing

Supervisor: Dr. Alia Abdelmoty

School of Computer Science and Informatics

Cardiff University

Abstract

The proposed project is a web application for recording a personal or study diary, implementing spaced-repetition training to allow users to better remember important information and the events of their lives. There are numerous diary apps and websites currently available, with both online and offline functionality. These programs typically serve a number of productivity functions, including personal organisation and time management. These apps are ostensibly designed - at least in part - to help users remember information. Learners need to access such information during exams, assignments or complex tasks. Diary-keepers use this information to remind themselves of the events of their lives, and to keep track of upcoming events.

Current apps allow users to organise their information, but make no effort to help users recall information they deem important. This project attempts to fulfil this need through the design and implementation of a web application that allows users to keep an ongoing diary, processing this information into relatively atomised elements, and generating digital "flashcards" that users can use to revise the information they enter. These flashcards will be shown at intervals defined by a spaced-repetition algorithm, to maximise retention of information while minimising the number of flashcards a user is required to review each day to retain all the information therein.

Table of Contents

Figures

Tables

Acknowledgements

This project would not have been possible without the help of my friends and colleagues on the MSc Computing course here at Cardiff University; the patient and wise counsel of my supervisor Dr Alia Abdelmoty; and the heroic efforts of the whole COMSC staff during these most trying of times.

This project is dedicated, as ever, to my family, for whom everything and anything is worth doing.

1. Introduction

1.1 Problem Statement

1.1.1 Current tools

The current landscape of diary applications and web services is relatively rich and varied. Numerous tools exist that allow users to take notes, make diary entries and keep track of the various events in their lives. These applications broadly fall into four categories, although most straddle two or more categories in their overall uses.

Personal organisation

These applications focus on the personal organisation of notes, documents and multimedia including images, audio and video. This is the category most similar to traditional diaries, which aim to organise and collate the information most important to its users over time, through frequent periodic input. These applications can also feature tools aimed towards students – including those studying material outside of formal education institutions – by structuring their interfaces and categories to easily organise information by subject or their constituent topics.

Time management

These applications focus on time management, and schedule management. Typically integrated with in-built calendar functionality – as well as allowing integration with popular calendar services such as Google Calendar, Microsoft Outlook and Apple iCal – these services often target professional users. Rather than focusing on past events, as many diary practices do, these applications are designed to organise the unfolding of future events, meetings, appointments and other obligations.

Synchronisation

Closely related with the previous category, these applications highlight the ability to synchronise related data across multiple contexts, and with multiple users. Beyond time management, these applications can also be used to marshal the input of multiple otherwise-independent contexts, such as the collaborative work of different users, or the automated output of disparate online and offline services, such as notifications from web services.

Long-term tracking

These applications are designed to be used over extended periods of time, in order to track distinct data-points. They can track quantitative measures, such as a user's weight or daily step count; or qualitative measures such as mood. These applications often highlight automated input methods, such as using health tracking devices like Fitbit or Apple Health to track the biometric and activity data of users.

1.1.2 Unserved need: memorisation

All of these categories of applications are designed to help their users remember information that is important to them, either to retain useful information about past events or to provide structure and reminders about their future plans and obligations. For students, this includes helping them to organise study materials in such a way as to help them recall information during assignments or examinations. Outside of study, such organised information can aid in the performance of complex tasks, particularly where processes and procedures require adherence to a sequence of complicated actions, or reference to myriad pieces of information. For traditional diarists, this information allows them to look back on the events of their lives and serve as a perpetual memento of their experiences and feelings.

While these applications allow users to organise this information in various ways, they do not make an effort to help users independently recall important information in their day-to-day lives. Diaries are useful reminders of information, but serve to effectively 'offload' information from the user's own memory to the application itself. The diaries thus created are a useful reminder, but not an effective way to help users strengthen and maintain their own memories long term.[1]

1.2 Aims and Objectives

1.2.1 Aim of this project

I propose to fulfil this unserved need through the design and implementation of a diary application that allows users to reinforce their own memories regarding the information

[1] The act of writing diary entries does have a beneficial effect on the ability for the writer to remember information, but without reinforcement, this information will still be hard to retain in long-term memory.

they write about. This will be a web application that provides for the maintenance of an ongoing diary, the entries of which will be processed automatically into digital flashcards that users can study and revise. The study process will use a space-repetition algorithm, which will maximise the ability for users to retain the information stored within their diaries. By using spaced repetition, the number of flashcards needed to complete a daily study session for any particular diary entry will be reduced over time, as flashcards are shown less frequently. This will allow for the more frequent study of recently added flashcards, or flashcards that are more frequently misremembered or forgotten.

1.2.2 Primary objectives

The primary objectives of the project are as follows:

- To create a backend web API to allow for the creation, retrieval and deletion of diaries and diary entries.
- To create a microservice that takes diary entries and processes them into useful flashcards.
- To create a responsive frontend client with an intuitive user interface.
- To allow users to register an account and authenticate with the web application, and maintain their own diaries, entries and flashcards.
- To allow users to create and view diaries and entries through the frontend client.
- To allow users to study flashcards through the frontend client.

2. Background

As one of the fundamental processes of the human mind, memory has long served as a subject of great interest across diverse fields. The study of memory, and its enhancement, has driven research within – for example – physiology, aiming to understand the biological systems that bring forth the phenomenon of memory itself; and medical science, with the aim of restoring or preserving the memory function of sick or injured patients. Research into the functioning, enhancement or restoration of memory function is well beyond the scope of this dissertation, but this section aims to provide some context for the proposed solution to the problem statement.

2.1 Memory, Recall, Diary-keeping

2.1.1 Memory and retrieval

Physiologist Lauralee Sherwood defines memory as 'the storage of acquired knowledge for later recall.'[2] The concept of memory in computer science borrows its general idea from that of memory in human physiology – computer memory is used for the storage of information for manipulation and processing by the CPU and other components of the computer system.

Computer memory functions as a short-term memory store; it is volatile, designed to be randomly accessed but not for long-term storage. For this, typical computer systems use more durable storage media such as hard-disk drives, solid state drives or portable media such as CDs or DVDs. In much the same way, human memory is often thought to consist of two closely related but distinct systems: long-term memory (LTM) and short-term memory (STM, also commonly referred to as working memory (WM)).[3] The exact functioning of these distinct processes is still subject to much debate within the fields of neurology and physiology; the classic model first proposed by Richard Atkinson and Richard Shiffrin in 1968 of a multi-store memory has come under significant scrutiny since its first

[2] Lauralee Sherwood, *Human Physiology: From Cells to Systems* (Cengage Learning, 2015), 157.

[3] Dennis Norris, 'Short-Term Memory and Long-Term Memory Are Still Different', *Psychological Bulletin* 143, no. 9 (2017): 992–1009, https://doi.org/10.1037/bul0000108.

publication.[4] Indeed, some researchers argue that these are not distinct systems at all, but rather the different functional expressions of the same single memory store; Nathan Rose et. al. write of the confounding similarities between STM and LTM memory retrieval function, as an example of the many ways in which the multi-store model is unable to fully explain the functioning of human memory.[5]

The study of memory retrieval (and the closely related process of memory encoding) is a sub-field within memory studies that is of particular interest to this project.[6] Numerous studies aim to expand general understanding of how memory retrieval functions in the mind, and the as-yet not fully understood capacity for memory retrieval generally. Green et. al. discuss experimental evidence that 'memory retrieval can, under favourable circumstances, proceed with essentially no interference while central resources are otherwise occupied.'[7] Similarly, Einstein et. al. suggest 'it seems unlikely that most people normally rely on constant and capacity-consuming processes when faced with [prospective memory] demands over anything other than brief delays.'[8] Both studies suggest that the innate memory retrieval abilities of neurotypical people are non-demanding mental processes. Miner et. al. go further, writing that their research suggests 'there is no distinction between the fidelity of visual working memory and visual long-term memory, but

---

[4] See, for example, Eugen Tarnow, 'Why The Atkinson-Shiffrin Model Was Wrong From The Beginning', *WebmedCentral Neurology* 1, no. 10 (2010): 13.

[5] Nathan S. Rose et al., 'Similarities and Differences Between Working Memory and Long-Term Memory: Evidence From the Levels-of-Processing Span Task', *Journal of Experimental Psychology. Learning, Memory, and Cognition* 36, no. 2 (2010): 471–83, https://doi.org/10.1037/a0018405.

[6] For a useful survey of studies on this subject, see Amanda Parker, Edward L. Wilding, and Timothy J. Bussey, eds., *The Cognitive Neuroscience of Memory: Encoding and Retrieval*, 1. publ, Studies in Cognition Series (Hove: Psychology Press, 2002).

[7] Collin Green, James C. Johnston, and Eric Ruthruff, 'Attentional Limits in Memory Retrieval—Revisited.', *Journal of Experimental Psychology: Human Perception and Performance* 37, no. 4 (2011): 1096, https://doi.org/10.1037/a0023095.

[8] Gilles O. Einstein et al., 'Multiple Processes in Prospective Memory Retrieval: Factors Determining Monitoring Versus Spontaneous Retrieval.', *Journal of Experimental Psychology: General* 134, no. 3 (2005): 341, https://doi.org/10.1037/0096-3445.134.3.327. Here, prospective memory refers to memories related to actions to be performed in the future, such as remembering appointments or obligations.

instead both memory systems are capable of storing similar incredibly high fidelity memories under the right circumstances.'[9]

2.1.2 Long-term memory enhancement and diary-keeping

If short-term memory and long-term memory can be considered, under optimal circumstances, to be of equivalent – or at least similar – fidelity and 'accuracy', and that the process of memory retrieval can be shown to be a relatively trivial process for the brain to undertake, it is no surprise that much research has focused on the diagnosis and treatment of disorders relating to long-term memory, and the general enhancement of these functions.[10] This is an area rich with experimental research. Simons et. al. show how the ability of dementia patients – with severely degraded long-term memory – to retrieve memories based on pictorial stimuli provides 'compelling evidence in favour of the multiple input model of long-term memory.'[11] Frith et. al. provide experimental evidence for high-intensity exercise being 'effective in enhancing long-term memory (for both [20 minute] and [24 hour] follow-up assessments.'[12]

Of primary interest to this project is the experimental evidence relating to writing and diary-keeping as memory-enhancement tools. Szőllősi et. al. studied the effect of sleep and

---

[9] Annalise E. Miner, Mark W. Schurgin, and Timothy F. Brady, 'Is Working Memory Inherently More "Precise" Than Long-Term Memory? Extremely High Fidelity Visual Long-Term Memories for Frequently Encountered Objects', *Journal of Experimental Psychology. Human Perception and Performance* 46, no. 8 (2020): 813, https://doi.org/10.1037/xhp0000748.

[10] For a general overview of research in this sub-field, see Arseni K. Alexandrov and Lazar M. Fedoseev, *Long-Term Memory Mechanisms, Types and Disorders*, Neuroscience Research Progress Series (Hauppauge, N.Y.: Nova Science Publishers, 2012).

[11] J. S. Simons, 'Recollection-Based Memory in Frontotemporal Dementia: Implications for Theories of Long-Term Memory', *Brain* 125, no. 11 (1 November 2002): 2523, https://doi.org/10.1093/brain/awf247.

[12] Emily Frith, Eveleen Sng, and Paul D. Loprinzi, 'Randomized Controlled Trial Evaluating the Temporal Effects of High-Intensity Exercise on Learning, Short-Term and Long-Term Memory, and Prospective Memory', *European Journal of Neuroscience* 46, no. 10 (2017): 2557, https://doi.org/10.1111/ejn.13719.

circadian rhythm on the ability of participants to retrieve memories recorded in diaries.[13] They found that, while the time of day did not affect the ability for participants to retrieve memories, 'participants were significantly better at remembering events that had been recorded in the evening [rather than the following morning] about a month earlier.'[14] Ho et. al. show that a rehabilitation program consisting in part of diary training saw 'a significant increase in children's abilities to perform daily routines that demanded recall of information and events.'[15] Linderholm and Abrams show evidence of the effectiveness of expressive writing to improve memory and cognition, writing 'the present experiment showed that expressive writing about a negative life event gives a boost to long-term memory performance.'[16]

The above serves to highlight the growing consensus that writing, and diary-keeping, provide measurable benefits to the functioning of long-term memory, both to general populations as well as those with specific clinical needs. This suggests the utility of diary-keeping tools as a method of enhancing and preserving long-term memory function. The following section continues this exploration, focusing on spaced repetition training as another method for improving memory function, and a complement to diary-keeping.

2.2 Spaced Repetition

Spaced repetition training (SRT) has been studied by researchers since at least the late 1970s.[17] The concept is relatively straightforward. Firstly, a subject encodes a memory,

---

[13] Ágnes Szőllősi et al., 'A Diary after Dinner: How the Time of Event Recording Influences Later Accessibility of Diary Events', *Quarterly Journal of Experimental Psychology* 68, no. 11 (November 2015): 2119–24, https://doi.org/10.1080/17470218.2015.1058403.

[14] Szőllősi et al., 2123.

[15] Joanna Ho et al., 'Rehabilitation of Everyday Memory Deficits in Paediatric Brain Injury: Self-Instruction and Diary Training', *Neuropsychological Rehabilitation* 21, no. 2 (1 April 2011): 183, https://doi.org/10.1080/09602011.2010.547345.

[16] Tracy Linderholm and Lise Abrams, 'The Benefits of Expressive Writing on Long-Term Memory Performance', in *Long-Term Memory: Mechanisms, Types and Disorders* (Nova Science Publishers, 2012), 144.

[17] A useful brief bibliography of studies on spaced repetition training can be found in Shiri Oren, Charlene Willerton, and Jeff Small, 'Effects of Spaced Retrieval Training on Semantic Memory in

either through recording or memorising some piece of information. After an interval, they attempt to recall that memory. If successful, they wait for a longer interval and recall the memory again. With each successful retrieval, the interval before the next attempt grows larger. A failed attempt at retrieval shortens the interval until the next attempt. In this way, information is measurably stored within long-term memory in a consistent and repeatable way. Miner et. al. write, 'It is well known that long-term memory improves with repetition [...], with a large literature demonstrating this for a variety of materials [...], and many influential studies asking about how best to space these repetitions to maximize the improvement in memory.'[18]

Initially conceived as a therapeutic tool to alleviate the problems of chronic clinical memory deficits,[19] SRT has also gained widespread traction as a tool for memory improvement within the general population. Pham et. al. show how a card-based mobile application designed to help teach users English through SRT resulted in 'better retention and [...] also enabled the learning items to be more appealing to the users.'[20] In their student project, Karch et. al. found similar success with a web application utilising SRT principles to aid students with the study of histology.[21]

Alzheimer's Disease: A Systematic Review', *Journal of Speech, Language & Hearing Research* 57, no. 1 (February 2014): 247–48, https://doi.org/10.1044/1092-4388(2013/12-0352).

[18] Miner, Schurgin, and Brady, 'Is Working Memory Inherently More "Precise" Than Long-Term Memory?', 814.

[19] See Oren, Willerton, and Small, 'Effects of Spaced Retrieval Training on Semantic Memory in Alzheimer's Disease'; Karri S. Hawley et al., 'A Comparison of Adjusted Spaced Retrieval versus a Uniform Expanded Retrieval Schedule for Learning a Name–Face Association in Older Adults with Probable Alzheimer's Disease', *Journal of Clinical and Experimental Neuropsychology* 30, no. 6 (18 July 2008): 639–49, https://doi.org/10.1080/13803390701595495.

[20] Xuan-Lam Pham et al., 'Card-Based Design Combined with Spaced Repetition: A New Interface for Displaying Learning Elements and Improving Active Recall', *Computers & Education* 98 (July 2016): 142, https://doi.org/10.1016/j.compedu.2016.03.014.

[21] Dominik Karch et al., 'Efficiency of Web Application and Spaced Repetition Algorithms as an Aid in Preparing to Practical Examination of Histology: PS195', *Porto Biomedical Journal* 2, no. 5 (September 2017): 187–88, https://doi.org/10.1016/j.pbj.2017.07.030.

The above studies highlight the popularity of SRT as a study method. Numerous other empirical studies demonstrate the effectiveness of the method for the retention of educational information by students in various subjects, particularly highly fact-based disciplines like STEM subjects.[22] It has also proven especially useful in second-language acquisition, where large quantities of vocabulary words and their definitions must be learned by students.[23] The use of computer software in particular has attracted notable academic attention, particularly as an automated and relatively straightforward way for users to study and manipulate their bank of information for study purposes.[24]

The preceding two sections have provided some of the academic context for this project, as it relates to the effectiveness of diary-keeping and SRT for the improvement of long-term memory function. The next section will provide an overview of the current landscape for consumer software products that are most closely related to this dissertation project.

2.3 Current Applications

2.3.1 – Diary applications

Numerous diary applications already exist on the market, targeting a variety of needs and use-cases. Three such applications will be discussed here, but this is in no way an

---

[22] Diane Persellin, *A Concise Guide to Improving Student Learning: Six Evidence-Based Principles and How to Apply Them*, First edition. (Sterling, Virginia: Stylus, 2014); Anton Lambers and Adrian J. Talia, 'Spaced Repetition Learning as a Tool for Orthopedic Surgical Education: A Prospective Cohort Study on a Training Examination', *Journal of Surgical Education* 78, no. 1 (2021): 134–39, https://doi.org/10.1016/j.jsurg.2020.07.002.

[23] Aroline E. Seibert Hanson and Christina M. Brown, 'Enhancing L2 Learning through a Mobile Assisted Spaced-Repetition Tool: An Effective but Bitter Pill?', *Computer Assisted Language Learning* 33, no. 1–2 (2 January 2020): 133–55, https://doi.org/10.1080/09588221.2018.1552975.

[24] Siddharth Reddy et al., 'Unbounded Human Learning: Optimal Scheduling for Spaced Repetition', vol. 13-17-, KDD '16 (ACM, 2016), 1815–24, https://doi.org/10.1145/2939672.2939850; Wisam Al-Rawi, Lauren Easterling, and Paul C. Edwards, 'Development of a Mobile Device Optimized Cross Platform-Compatible Oral Pathology and Radiology Spaced Repetition System for Dental Education', *Journal of Dental Education* 79, no. 4 (2015): 439–47, https://doi.org/10.1002/j.0022-0337.2015.79.4.tb05902.x.

exhaustive list of current options – rather, they should serve as examples of existing functionality.

One popular diary-keeping application is Day One.[25] Available for the Apple ecosystem, across macOS and iOS, the app has a focus on easy journaling that can be as compact or prosaic as the user intends. It features markdown as a text formatting approach, as well as integration for rich metadata and tags. It can also incorporate images, audio and other multimedia, as well as integration with social media services such as Instagram. For retrospective entry viewing, the app provides an 'on this day' function, showing entries from the current calendar date in previous years.

Another popular application is Diarium.[26] It shares many features with Day One, including integrations with other popular web services via public APIs, the use of rich metadata and multimedia and structures views for reliving previous diary entries. Unlike Day One, Diarium is available across a broader spectrum of operating systems, including macOS, iOS, Windows and Android. Both apps also include synchronisation over the cloud, to keep and retrieve diary entries across multiple devices.

Another application is Grid Diary, available for macOS, iOS and Android.[27] This application aims to solve the issue of writer's block, by providing prescribed templates and structures to lower the barrier to creating journal entries. It is named for its grid format, which provides a clear structure for the creation of entries and prospective plans. It also provides a library of writing prompts, which ostensibly allow for users to engage more easily with their memories as they write.

All three applications provide a strong emphasis on the act of creating diaries and diary entries. Their focus is primarily on the writing and journaling process, providing numerous ways to embed rich information and metadata into entries, often automatically. Fundamentally, they act as data stores, with diary entries – whilst being organised and filtered according to the metadata included – acting as static entities to be browsed in a relatively freeform manner after the act of writing them. None of these apps make claims or

[25] 'Day One: Your Journal for Life', accessed 27 September 2021, https://dayoneapp.com/.

[26] 'Diarium, by Timo Partl', accessed 27 September 2021, https://timopartl.com/#diarium.

[27] 'Grid Diary: The Simplest Way to Get Started with Keeping a Diary | Grid Diary', accessed 27 September 2021, https://griddiaryapp.com/.

efforts to enable users to remember their entries independently of the application, beyond the ability to look them up within the system.

2.3.2 – Spaced repetition systems

SRT is often presented instead as a 'spaced repetition system' (SRS). Generally speaking, SRS is an implementation of an SRT regime, typically prescribed by the application in question, although some provide elements of customisation with regards to the specific intervals used.

By far the most prominent SRS application is Anki.[28] Anki uses digital flashcards that are created directly within the application; each card has a 'front' and 'back' side, where the front contains a prompt for the user to respond to, and the back contains the desired answer. Users then self-report their answer as correct or incorrect, and the interval until the next time that card is shown is increased/decreased depending on that answer and the spacing regime that the user has selected for the current deck of flashcards. Collections of user-created flashcards for a wide variety of subjects are made available online for download.[29] User progress can be made and synced across a variety of devices, including macOS, iOS, Android and Windows, as well as an online web application.

Another web application is Memrise.[30] Here, material is organised into 'courses', with much less focus on self-generated unstructured content than Anki's flashcard creation tools. Memrise has a clear focus on language learning in particular, although courses exist for a variety of other subjects as well. Courses can be user created, although many of the most popular courses are curated by Memrise itself, providing a much more prescribed and seemingly authoritative approach than Anki, whose shared flashcards are wholly user made and only ranked according to user feedback. Content can be answered by typing directly into the application, with the written response of the user graded against expected answers.

Both applications – and indeed virtually all other SRS implementations – are targeted towards learners, either those engaged in self-study or students within educational

---

[28] 'Anki - Powerful, Intelligent Flashcards', accessed 4 October 2021, https://apps.ankiweb.net/.

[29] 'Shared Decks - AnkiWeb', accessed 4 October 2021, https://ankiweb.net/shared/decks/.

[30] Memrise, 'Learn a Language. Meet the World. | Memrise', accessed 4 October 2021, https://www.memrise.com.

institutions who require structured study and revision material. Neither provide any direct interface between freeform study notes and the flashcards; instead, flashcards must either be manually created directly within the service, or users must find appropriate pre-existing collections to use.

Two applications do exist that allow users to create flashcards using Markdown-like text editors, rather than through specific flashcard-creation interfaces. Mochi is an SRS application that uses Markdown to format the front and reverse of cards.[31] It is compatible with Anki flashcards, allowing users to import cards created in that system. As can thus be expected, beyond the use of Markdown as a formatting tool, it largely emulates the functionality of already-existing SRS tools. Remnote is a study application that allows users to create freeform study notes, incorporating metadata such as cross referencing and in-depth organisation.[32] Using a highly structured formatting style, flashcards can be created from excerpts of this text, according to the needs of the student.

Both of these apps still require flashcards to be specifically created – either through a dedicated flashcard creation interface like Mochi, or by selecting specific sections and sentences in Remnote. Like all SRS applications previously mentioned, both also are squarely aimed at students and learners.

[31] 'Mochi — Spaced Repetition Made Easy', accessed 4 October 2021, https://mochi.cards/.

[32] 'RemNote | The Best Way to Remember and Organize What You Learn', accessed 4 October 2021, https://www.remnote.io/.

3. Product

The design of this application will primarily involve three major components:

- A diary keeping component, where users enter the primary information for their diary and interact with it as a normal diary.
- A flashcard revision component, where users review flashcards based on their diary entries, according to the schedule maintained by the spaced repetition system.
- A flashcard generation component, where diary entries are automatically processed into flashcards, based on user-customisable criteria.

3.1 Overview of Major Components

3.1.1 – Diary keeping

This component will largely function like a conventional diary-keeping application. As the focus of this application is remembering information, rather than productivity or personal management, the focus will be on the note-taking experience.

While users will be able to create notes in their own preferred style and format, information and templates will be provided to assist the flashcard generation process by structuring data in a more easily processed way. The text editor will feature Markdown support, allowing users to format their notes while providing further context clues for the processing of this data.

3.1.2 – Flashcard revision

This component will focus on the management and revision of the flashcards generated by the application. Upon addition to the schedule, users can review the generated cards and edit them to be more useful for their needs. Ideally, the system will produce cards that are usable without further editing.

New cards will be reviewed in the order they are added, and then scheduled for revision later after being reviewed the first time. After the specified interval (each card has an interval attached to it), the card will be shown again. The user will recall the information the card is prompting, and specify whether they remember it or not. If they do, the card is rescheduled with a longer interval; if not, the card is rescheduled with a shorter interval. More easily remembered information is not prompted for increasing periods of time, while difficult to remember information is more frequently reinforced.

3.1.3 – Flashcard generation

This component will process the diary entries and notes entered by the user and generate flashcards. The primary flashcard format will be cloze sentences/passages, where users will see a portion of their diary entry with one or more missing words and will be prompted to fill the blanks.[33]

To create these cards, diary entries will be split into their component sentences and paragraphs, and key words will be identified, such as by:

- Privileging words in emphasising punctuation (e.g. words in "double-quotes", bold or *italics*) or grammar (e.g. Capitalised Words, ALL-CAPS WORDS, etc.)
- Filtering out high frequency words, and certain parts of speech such as interjections, conjunctions and pronouns.
- Parsing markdown-formatted text, such as lists, code snippets, headings, blockquotes.

Generated cards are then scheduled for revision, or to be further edited/refined by the user.

3.2 Design and Requirements

3.2.1 Requirements

Functional requirements

- Authentication
    - Users need to be able to register and log in
    - *Acceptance criteria:*
      The registration and log-in functions are clearly visible and usable from the home screen.
- Single Sign-On
    - Users should be able to log into the system once to access all parts of the program.

[33] Cloze testing – also colloquially known as gap-fill or fill-the-blanks questions – are often used across different disciplines to measure text comprehension and aid in memorisation. Suzanne Kleijn, Henk Pander Maat, and Ted Sanders, 'Cloze Testing for Comprehension Assessment: The HyTeC-Cloze', *Language Testing* 36, no. 4 (October 2019): 553–72, https://doi.org/10.1177/0265532219840382.

    - *Acceptance criteria:*
      After logging in, the user can access the diary and study pages.

- User Accounts
    - Users can have user accounts, to organise all their data.
    - *Acceptance criteria:*
      Users have exclusive access to their own diaries, entries and flashcards, and cannot see those made by other users.
- Diary and Entry Creation
    - Users need to be able to create diaries and write diary entries
    - *Acceptance criteria:*
      Users can create a diary and a diary entry for that diary.
- Markdown Text Editor
    - Users can create freeform diary entries using a markdown text editor
    - *Acceptance criteria:*
      The interface for creating a diary entry allows for rich text, and the ability for users to preview their entries before posting them.
- Diary Entry Metadata
    - Metadata like dates, locations and mood should be added with entries.
    - *Acceptance criteria:*
      Users can add a date, location and mood to each diary entry during creation.
- Diary Viewing
    - Users must be able to read through their diary like a traditional chronological diary
    - *Acceptance criteria:*
      Users can select diary entries and see them displayed with their formatting.
- Card Generation
    - Diary entries need to be converted to cards
    - *Acceptance criteria:*
        - Flashcards are automatically generated for a new diary entry if a user choose to create flashcards.
        - Flashcards consist of a front-side, which has a prompt, and a back-side, which has the solution that the user should remember.
- Automatic Card Generation
    - Using markdown syntax, metadata and other methods, cards are generated from the diary entries.
    - *Acceptance criteria:*
      Generated flashcards use information from the diary entry in their body.
- Study

    - Users can study their cards
    - *Acceptance criteria:*
        - Users can choose a diary or diary entry and study only the associated flashcards.
        - Flashcards should be shown one after another.
        - Users can select if they remembered or forgot the information on the back-side of the card.
- Scheduling of cards
    - Cards are scheduled using an SRT algorithm
    - *Acceptance criteria:*
      Flashcards are scheduled to be due for study at increasing intervals when users remember them, or to be shown sooner if they forget.

Non-functional requirements

- Usable interface
    - The application should be simple to use with no prior instruction
    - *Acceptance criteria:*
        - Users can register, create a diary and a diary entry, and study the generated flashcards without external direction.
        - Any required instructions should be present in the application.
- Responsiveness
    - The application should be responsive and work on a range of devices.
    - *Acceptance criteria:*
      Users can use the application on a desktop computer, laptop or smartphone.

3.2.2 Use-case Diagram

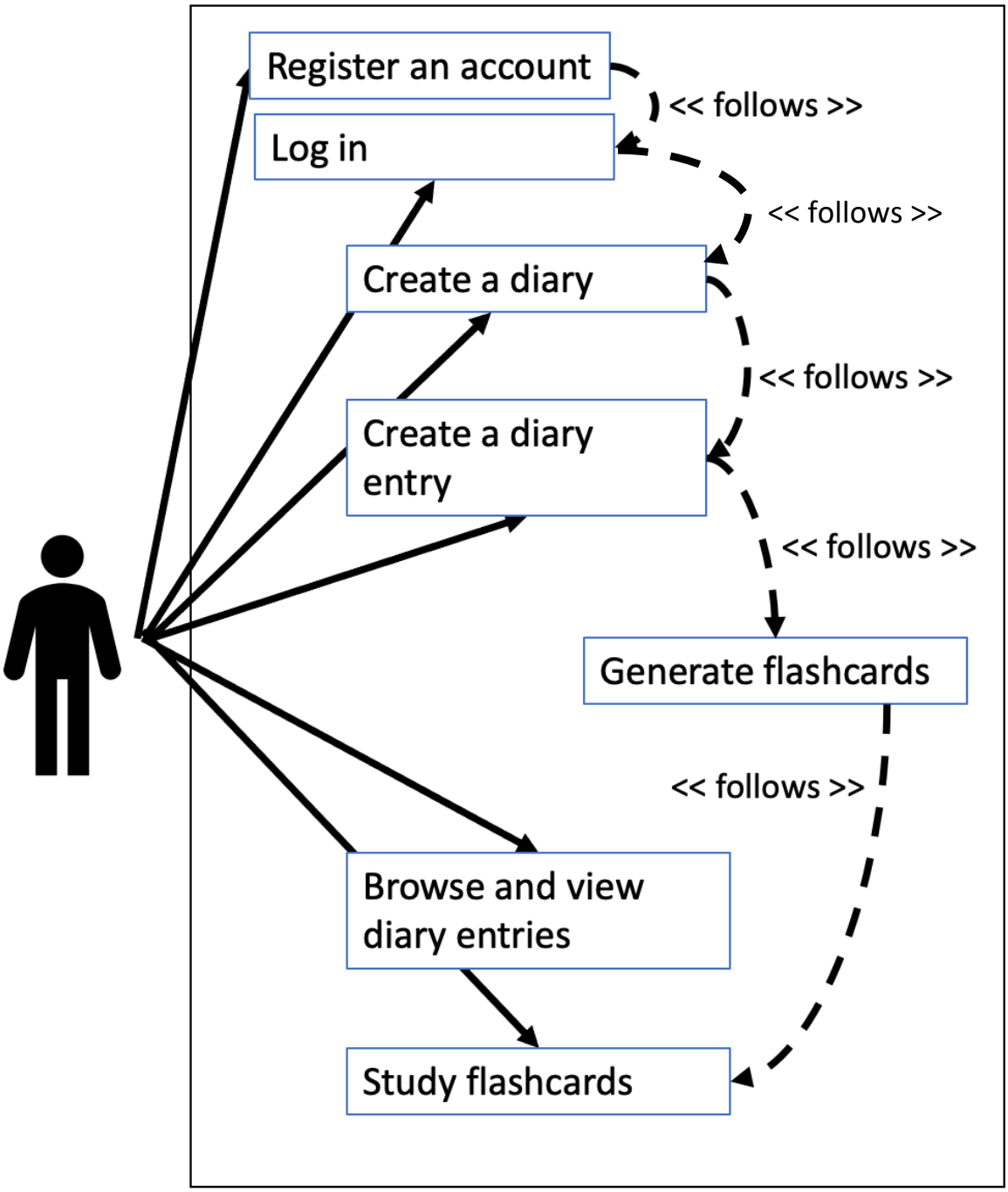

Figure 1 - Typical use-case diagram

The above use-case diagram shows the typical usage of the application by a user. Included in the diagram are follower arrows, indicating steps/activities that must be performed in order. In other words, flashcards can only be studied after they have been generated, which in turn can only happen once a diary entry has been created, and so on.

3.2.3 API endpoints

The backend server of the application will use a RESTful API to manage the flow of data from the database through to the user, and to standardise the communication between the frontend React application and the server's functionality. Below is a list of API endpoints made available by the server, as well as a brief description of each endpoint's purpose and the specific controller methods they call.

- Users
  - `GET /api/users/current`
    - Returns information for currently logged-in user.
    - Method: `UserController.getCurrentlyLoggedInUser`
  - `GET /api/users/:id`
    - Returns user information.
    - Method: `UserController.getUserById(id)`
  - `POST /api/users/register`
    - Signs up a new user and logs them in.
    - Method: `UserController.postUser`
  - `POST /api/users/login`
    - Logs in a user
    - Method: `UserController.loginUser`
  - `PUT /api/users/:id`
    - Updates user information.
    - Method: `UserController.updateUserInfo`
- Diaries
  - `GET /api/diaries/`
    - Returns all diaries for currently logged-in user.
    - Method: `DiaryController.getUserDiaries`
  - `GET /api/diaries/:id`
    - Returns a diary.
    - Method: `DiaryController.getDiary(id)`
  - `POST /api/diaries/`
    - Creates a new diary.
    - Method: `DiaryController.createDiary`
  - `PUT /api/diaries/:id`
    - Updates diary information (not needed for MVP).
    - Method: `DiaryController.updateDiary(id)`
  - `DELETE /api/diaries/:id`
    - Deletes a diary.
    - Method: `DiaryController.deleteDiary(id)`
- DiaryEntries
  - `GET /api/diaries/:id/entries`
    - Returns all diaries for a specific diary by diary ID.
    - Method: `DiaryEntryController.getDiaryEntries(id)`
  - `GET /api/entries/`
    - Returns all diary entries for a user.
    - Method: `DiaryEntryController.getUserEntries`

    - `GET /api/entries/:id`
        - Returns a diary entry by ID.
        - Method: `DiaryEntryController.getEntry(id)`
    - `POST /api/diaries/:id/newentry`
        - Creates a new entry for a specific diary by diary ID.
        - Method: `DiaryEntryController.createEntry(id)`
    - `PUT /api/entries/:id`
        - Updates diary entry information (not needed for MVP).
        - Method: `DiaryEntryController.updateEntry(id)`
    - `DELETE /api/entries/:id`
        - Deletes a diary entry.
        - Method: `DiaryEntryController.deleteEntry(id)`
- Flashcards
    - `GET /api/diaries/:id/flashcards`
        - Returns all flashcards for a specific diary by diary ID.
        - Method: `FlashcardController.getDiaryFlashcards(id)`
    - `GET /api/diaries/:id/flashcards/due`
        - Returns all flashcards due for study, for a specific diary by diary ID.
        - Method: `FlashcardController.getDiaryDueFlashcards(id)`
    - `GET /api/entries/:id/flashcards`
        - Returns all flashcards for a specific diary entry by entry ID.
        - Method: `FlashcardController.getEntryFlashcards(id)`
    - `GET /api/entries/:id/flashcards/due`
        - Returns all flashcards due for study, for a specific diary entry by entry ID.
        - Method: `FlashcardController.getEntryDueFlashcards(id)`
    - `GET /api/flashcards/`
        - Returns all flashcards for a user.
        - Method: `FlashcardController.getUserFlashcards`
    - `GET /api/flashcards/due`
        - Returns all flashcards due for study, for a user.
        - Method: `FlashcardController.getDueUserFlashcards`
    - `GET /api/flashcards/:id`
        - Returns a flashcard by ID.
        - Method: `FlashcardController.getFlashcard(id)`
    - `POST /api/entries/:id/newflashcard`
        - Creates a new flashcard for a specific diary entry by entry ID (not directly used in MVP).
        - Method: `FlashcardController.createFlashcard(id)`
    - `PUT /api/flashcards/:id`
        - Updates flashcard information.
        - Method: `FlashcardController.updateFlashcard(id)`
    - `DELETE /api/flashcards/:id`
        - Deletes a flashcard.
        - Method: `FlashcardController.deleteFlashcard(id)`

3.2.4 Database Schema

This section outlines the database schema for the application. These schema model the different entity components of the application, including their internal references to one another.

User

Property	Data type	Unique?	Required?	Details
username	string	yes	yes	3-30 characters long
firstName	string	no	yes	
lastName	string	no	yes	
email	string	yes	yes	
password	string	no	yes	Automatically generated hash[34]
birthday	date	no	no	

Diary

Property	Data type	Unique?	Required?	Details
title	string	yes	yes	
userId	objectId	no	yes	Reference to User

[34] During user registration, the password provided by the user will be automatically passed through a hashing algorithm, and the resulting hash will be stored in the database. At no point will plain-text passwords be stored in the database.

DiaryEntry

Property	Data type	Unique?	Required?	Details
date	date	no	yes	Automatically defaults to current date, can be edited.
title	string	no	no	
content	string	no	yes	Markdown-formatted text string
location	string	no	no	
mood	string	no	no	Enumerated list
diaryId	objectId	no	yes	Reference to Diary
userId	objectId	no	yes	Reference to User

Flashcard

Property	Data type	Unique?	Required?	Details
entryDate	date	no	yes	Date of the original DiaryEntry
type	string	no	yes	Automatically assigned: ‘fill-the-blanks’, ‘mood’, ‘location’ or ‘date’
frontSide	string	no	yes	Markdown-formatted text
backSide	string	yes	yes	Markdown-formatted text
diaryId	objectId	no	yes	Reference to Diary
userId	objectId	no	yes	Reference to User
diaryEntryId	objectId	no	yes	Reference to DiaryEntry

contextUp	array	no	no	Array of nearby sentences from the original DiaryEntry, in order from closest to furthest, going higher up the card. (Only for 'fill-the-blanks' questions)
contextDown	array	no	no	Array of nearby sentences from the original DiaryEntry, in order from closest to furthest, going lower down the card. (Only for 'fill-the-blanks' questions)
correctStreak	integer	no	yes	Increases by one each time card is answered correctly; resets to zero if answered incorrectly. Defaults to zero.
nextDue	date	no	yes	Date that the card is due to be reviewed again. Defaults to card's creation time.

3.3 Technologies

For the technical implementation of this project, a number of modern web technologies were chosen, including:

- NodeJS and ExpressJS for the primary backend services, and the RESTful API.
- ReactJS for the frontend user interface, along with a WYSIWYG Markdown text editor.
- MongoDB as a non-relational database to store loosely-structured data for diary entries and flashcards.
- Visual Studio Code for development, and Git for source and version control.

Rudimentary natural language processing was considered, to further enhance the application's ability to produce useful flashcards, but due to the high likelihood of requiring considerably more time and computing resources for relatively slight gain, this approach was dismissed. Furthermore, the viability and usefulness of the system will be made suitably clear with a simpler approach to processing text. Instead, a generalised algorithmic approach was chosen, detailed further in Section 4.
The application was split into two major components: a backend web server, and a frontend web client that communicates with the server via a RESTful API. In order to harmonise the development process as much as possible, JavaScript was chosen as the primary development language throughout.

3.3.1 Backend Server

The design of the web server followed the principles of RESTful API design. Representational state transfer (REST) is a paradigm that prescribes how resources distributed on the Internet can be accessed and shared through consistent and standardised implementations. An API – or application programming interface – is the means by which computers and software can communicate with one another; each component part of a software application, its classes and libraries, expose methods that can be called to utilise the capability of that software. RESTful APIs are web APIs that use HTTP methods – such as GET, POST, DELETE and PUT – as the interface by which to call the methods of the software on a web server. Web servers implement API endpoints at different URLs, that can be called with these HTTP methods. A GET request to a particular URL may have a different effect to a POST request to that same URL, as they are implemented as different endpoints. This approach creates a unified style that most public web APIs implement, in order to facilitate smooth communications between systems and reduce false assumptions about the purpose of an endpoint. Rather than returning HTML documents for display directly in the browser, RESTful APIs will typically respond with data that can be used and processed further by client programs, usually as JavaScript Object Notation (JSON) documents (these are textual representations of JavaScript objects that can be directly parsed into actual JavaScript objects – or similar structures for other programming languages – for use by other programs).

The web server was written in Express.js, described as a 'fast, unopinionated, minimalist web framework.'[35] It is a JavaScript library designed for delivering web applications running on Node.js, a JavaScript runtime.[36] As stated, it takes a minimalist approach to delivering resources to web clients, with no built-in model-view-controller framework found in more full-featured frameworks such as Flask or Ruby on Rails. This presents a challenge, as it is not sufficient in and of itself to develop a web application, but it provides a transparent and easily configured API interface for the creation of applications using a variety of other libraries and frameworks in tandem. Third-party libraries (also known as packages) were managed in the backend using Node Package Manager (NPM), which is included with standard Node.js installations and links to the NPM software registry, through which packages can be installed and maintained.[37]

Express.js and Node.js are two elements of the so-called MERN software stack, with which this web application was built. The other two elements are React.js, discussed further in section 3.2.2, and MongoDB. MongoDB is a NoSQL database program, closely tied into an online platform (named MongoDB Atlas) for serving database records via API.[38] A NoSQL database is one that does not use structured query language, typically also referred to as a 'non-relational' database. Rather than storing records in tables, with relationships defined by primary and foreign keys columns in those tables, MongoDB uses atomic documents that closely resemble JSON. A document-based database was chosen for this project as it most closely resembles the structure of diaries and flashcards – while there are relationships between diaries, their entries, and the flashcards derived from those entries, each of these elements is interacted with by the user in relative isolation. The extensibility of document-based databases – which do not require a fixed schema for their documents – also means that different entries and flashcards can have more or fewer fields of data, without affecting the overall integrity of the system.

---

[35] 'Express - Node.Js Web Application Framework', accessed 5 October 2021, https://expressjs.com/.

[36] Node.js, 'Node.Js', Node.js, accessed 5 October 2021, https://nodejs.org/en/.

[37] 'Npm', accessed 7 October 2021, https://www.npmjs.com/.

[38] 'The Most Popular Database for Modern Apps', MongoDB, accessed 5 October 2021, https://www.mongodb.com.

In order to interact most efficiently with Express.js, the interface with MongoDB was made through another library, Mongoose. Mongoose allows for object-data modelling within JavaScript and Express, allowing MongoDB documents to be represented as JavaScript objects with defined characteristics.[39] These objects – defined by Mongoose 'models', an implementation of the modelling aspect of the MVC paradigm – were constructed using schemas, defining the attributes that each object would contain, and thus the fields within the MongoDB documents created. By interacting and processing documents in the server through Mongoose, the process of validating and type-casting data was made robust and systematic.

For user authentication, a variety of libraries were chosen. Bcrypt.js is a JavaScript implementation of Niels Provos and David Mazières's bcrypt hashing function.[40] The detasils of the function are beyond the scope of this report, but its general appeal as a secure password hashing function derive from its inherent ability to scale its complexity as general computing power increases. Traditional hashing functions are generally complex enough to provide security against contemporary attackers, but as computational power grows (which it does exponentially over time, per Moore's Law), they become easier to decrypt through brute-force attacks or other methods. Bcrypt.js is trivial to incorporate into wider applications such as this project. Once a user is logged in, they are provided with a web token through the libraries Passport-jwt, which is an authentication strategy for the wider middleware library Passport.[41] This web token contains an encrypted representation of the basic user information required by the frontend client, as well as serving as an authorisation token for any requests to the server that can only be made by authenticated users. These web tokens expire after a certain amount of time.

3.3.2 Frontend Client

The frontend web client was also built in JavaScript, using the React framework.[42] Frontend frameworks such as React – as well as others including Vue and Angular – provide libraries

[39] 'Mongoose ODM v6.0.9', accessed 5 October 2021, https://mongoosejs.com/.

[40] Niels Provos and David Mazières, 'A Future-Adaptable Password Scheme', n.d., 13.

[41] 'Passport', npm, accessed 6 October 2021, https://www.npmjs.com/package/passport.

[42] 'React – A JavaScript Library for Building User Interfaces', accessed 6 October 2021, https://reactjs.org/.

that allow for the programmatic manipulation of the HTML and CSS displayed in the browser. It does this by leveraging the document-object model (DOM), which treats individual elements within HTML documents as software objects, forming a tree structure. JavaScript code can then add, edit or delete elements from the DOM programmatically, resulting in the dynamic alteration of the displayed web page.

React takes this concept further by being explicitly component-based. Where HTML uses standardised, atomic elements (such as `<li>` or `<input>`), React allows for the freeform creation of components comprised of programmatically generated HTML content. For example, a `<LoginForm>` component could be created from multiple `<input>` elements nested within a `<input>` element. This `<LoginForm>` component could then be added to other React components, with different attributes passed to it in the form of 'props' (short for properties). This component-based approach allows for a frontend website to be developed using strong programming principles such as the separation of concerns, non-repeating of similar code, and a single-source of truth for widely-used functions and data.

React Router is a library that gives React access to the browser URL (through the Location interface of the main Window object in the DOM).[43] By wrapping the entire application in a `<Router>` component (provided by React Router), constituent components can access information in the current URL, and display DOM elements related to the current path. It also provides the ability to redirect based on the presence or absence of authentication information; as such, attempting to access protected routes can result in a redirect to a login page, for example.

React communicates with the server by sending HTTP requests to the API endpoints described above. This is accomplished through a library called Axios, which provides an abstraction above the XMLHttpRequest interface built into JavaScript.[44] Through Axios, a React app can make an HTTP request to a web server and receive JSON data. This data can then be used to generate further DOM changes, and change the state of the React application.

---

[43] 'React Router: Declarative Routing for React', ReactRouterWebsite, accessed 6 October 2021, https://reacttraining.com/react-router; 'Location - Web APIs | MDN', accessed 6 October 2021, https://developer.mozilla.org/en-US/docs/Web/API/Location.

[44] 'Axios', accessed 6 October 2021, https://axios-http.com/.

Having individual React components send HTTP requests through Axios and receive data directly presents a number of challenges. Chief among them is the management of the global state of the application. Components are frequently created, mutated and destroyed – as such, they are unable to maintain statefulness reliably. Data that should be consistent between components can become out-of-sync, as different versions of ostensibly the same object are contained within different components. Furthermore, some components are unable to properly render until they have the required data from the backend, or are so frequently instantiated that they would send an overwhelming number of HTTP requests with each new instance. Instead, a state-management library can be used, such as React-Redux, an implementation of the more general-purpose Redux library.[45] Redux comprises a global store of data retrieved from remote server APIs. Rather than creating HTTP requests directly, React components call Redux actions, which in turn make the required HTTP requests, receive the response and update the global data store, which all React components look to for their own data.

For the diary entry interface, the React-MDE (Markdown editor) library was chosen.[46] It provides a lightweight, user-friendly editor for creating markdown text, while providing functionality for limiting the formatting options available to users for the MVP version of this program, where only bold, italics, unordered lists and headers are supported. React-Markdown was chosen to provide the conversion from Markdown text to properly formatted HTML in diary entry display pages.[47]

[45] 'React Redux | React Redux', accessed 6 October 2021, https://react-redux.js.org/. React-Redux and Redux are conflated within this report, and referred to as Redux throughout.

[46] André Pena, *React-Mde*, TypeScript, 2021, https://github.com/andrerpena/react-mde.

[47] *React-Markdown*, JavaScript (2015; repr., remark, 2021), https://github.com/remarkjs/react-markdown.

4. Implementation

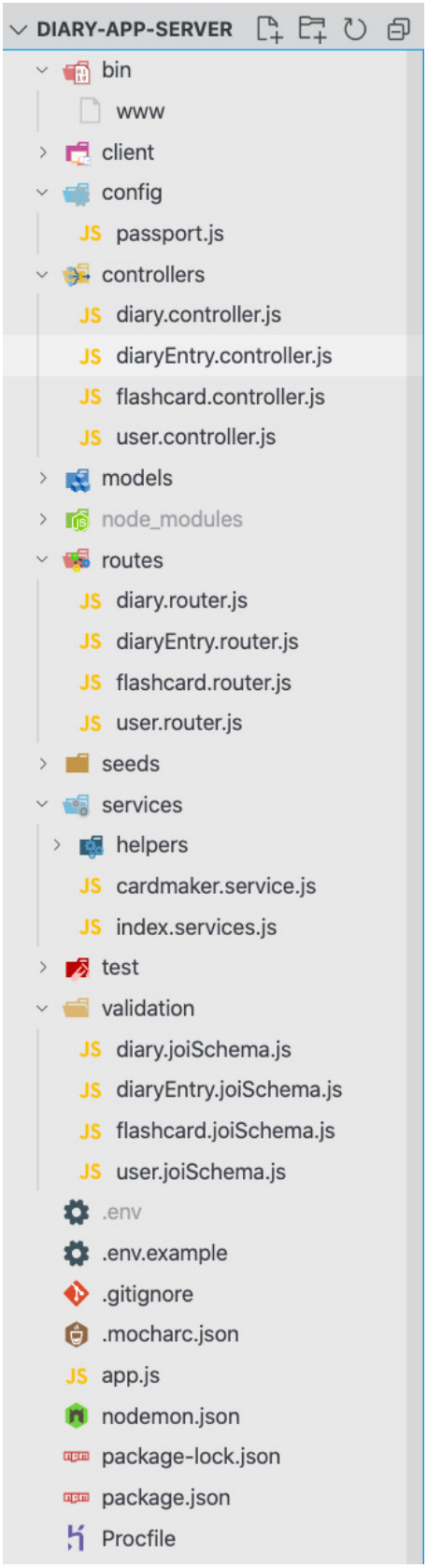

Figure 2 - Backend file structure

The project was built in a Git mono-repository, with the React client contained in a sub-directory at the project root named 'client' (see Figure 2). At the project root is a `package.json` file, which lists the various dependencies of the application, various scripts for purposes of development and deployment, as well as other project metadata. These scripts, shown in Figure 3, include `server`, which starts the application using nodemon, a version of the node runtime that watches the project folder for any code changes and restarts the server to update the version in memory. The `build` script first changes the working directory to the client

diary-app-server - package.json

```
"scripts": {
  "test": "mocha",
  "server": "nodemon bin/www",
  "build": "cd client && npm install && npm run build",
  "install-client": "cd client && npm install",
  "heroku-postbuild": "npm run build",
  "start": "node bin/www"
},
```

Figure 3 - package.json npm scripts

directory, runs `npm install` to install the various dependencies for the React client in its own internal node_modules directory, before running the `build` script of the React client's own `package.json` file.

This script consolidates the current state of the React client into a single directory of static files, which are served directly to the browser when accessing the URL of the website. During development, it is useful to run the React code directly, and so the `start` script of the React client's `package.json` is run instead. This

creates a live updating version of the React client that updates with code changes, and does not require the time-consuming process of building a static file. Static files are preferred for serving to end-users because they are minified during the build process to make them smaller and less expensive to transmit over the Internet; they are immutable, and can be versioned; and they hide much of the working of the development code from end users (although this can be reverse engineered without much difficulty).

This section will discuss the implementation of the web application, first focusing on the backend web server, before discussing the frontend React client. Please refer to section 3.3 for further context.

4.1 Backend Server

4.1.1 Express

Figure 4 shows the core setup for the Express application. A number of dependencies are imported to add functionality. Aside from those discussed in section 3.3.1, these include dotenv, which allows the application to access environment variables and thus keep sensitive data out of the codebase; path, which provides an interface for accessing the file structure from within the code; morgan, which provides detailed logs for HTTP requests and errors in the terminal for development; and bodyParser, which allows incoming data in HTTP request bodies to be parsed. An instance of Express is instantiated into the app variable, and the connection string to the MongoDB database is read from the process environment into the db variable.

diary-app-server - app.js

```
1  require("dotenv").config();
2  const path = require("path");
3
4  const express = require("express");
5  const mongoose = require("mongoose");
6  const morgan = require("morgan");
7  const bodyParser = require("body-parser");
8  const passport = require("passport");
9
10 const app = express();
11 const db = process.env.MONGODB_URI;
```

Figure 4 - Express application setup

Mongoose is used to setup a database connection, which persists throughout the life of the application. As this is itself an HTTP request, it is by definition asynchronous (also known as 'async') – all HTTP requests involve some latency between the client (which is the diary application web server in this case) and the server

(the MongoDB database server) – and so it is implemented as a JavaScript Promise. Lines 8 and 9 of Figure 5 show the then-catch format of this async call; the `connect` function is called, and once it resolves either the `then` function is called (if the async call was successful) or the `catch` function is called if there is an error.

diary-app-server - app.js

```
1  if (process.env.NODE_ENV !== "test")
2    mongoose
3      .connect(db, {
4        useNewUrlParser: true,
5        useUnifiedTopology: true,
6        useCreateIndex: true,
7      })
8      .then(() => console.log("Connected to MongoDB successfully!"))
9      .catch(err => console.log(err));
10
11 // Middleware
12 app.use(morgan("dev"));
13 app.use(bodyParser.urlencoded({ extended: true }));
14 app.use(bodyParser.json());
15 app.use(passport.initialize());
16 require("./config/passport")(passport);
17 app.use(express.static(path.join(__dirname, "client", "build")));
```

Figure 5 - Database connection and middleware setup

diary-app-server - passport.js

```
1  const JwtStrategy = require("passport-jwt").Strategy;
2  const ExtractJwt = require("passport-jwt").ExtractJwt;
3  const mongoose = require('mongoose');
4
5  const options = {
6    jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
7    secretOrKey: process.env.SECRET
8  }
9
10 module.exports = passport => {
11   passport.use(
12     new JwtStrategy(options, (payload, done) => {
13       User.findById(payload.id).then(user => {
14         if (user) {
15           const outputUser = { ...user.toJSON()};
16           delete outputUser.password;
17           return done(null, outputUser);
18         } else {
19           return done(null, false);
20         }
21       }).catch(err => console.error(err));
22     })
23   )
24 }
```

Figure 6 - Passport setup

Middleware is also setup at this early stage. These are functions that extend the behaviour of the Express application. Logging with morgan is initialised, as well as adding parsing for urlencoded and JSON data with bodyParser. Passport is also initialised, using settings from elsewhere in the codebase. These settings – shown in Figure 6 – use the JwtStrategy (Jwt being 'JSON web token') to decode the authorisation token that will be sent by authenticated users and find the mongoose User model from the database using the id from the payload.

diary-app-server - app.js

```
// ... Router setup here

app.get("*", (req, res) => {
  res.sendFile(path.resolve(__dirname, "./client/build", "index.html"));
});

module.exports = app;
```

Figure 7 - Route for serving static files

diary-app-server - www

```
const app = require('../app');
const http = require('http');

const port = process.env.PORT || 5000;
app.set("port", port);

const server = http.createServer(app);

server.listen(port);
```

Figure 8 - Creating the web server

Back in the main `app.js` file, the file ends by setting an HTTP GET route at '*' to serve the index.html file from the production build of the React client (see Figure 7). By using the wildcard * character, it ensures that any requests are served with this HTML file (and the accompanying React JavaScript file), at which point the React Router will handle information display. The app is then exported into /bin/www, a file that solely exists to create a web server using this application and listen on a port defined by the process environment (see Figure 8). This is then run by the `start` script in the server's `package.json` file.

4.1.2 Routing of API endpoints

The routes that make up the API endpoints are defined in separate files as individual Express routers. Within `app.js` these are defined just before the wildcard static file route. Express routers function as another middleware, adding the functionality of their routes to the overall application.

diary-app-server - app.js

```
1  // Routers
2  const userRouter = require("./routes/user.router");
3  app.use("/api/users", userRouter);
4  const diaryRouter = require("./routes/diary.router");
5  app.use("/api/diaries", diaryRouter);
6  const diaryEntryRouter = require("./routes/diaryEntry.router");
7  app.use("/api/entries", diaryEntryRouter);
8  const flashcardRouter = require("./routes/flashcard.router");
9  app.use("/api/flashcards", flashcardRouter);
10 const seedRouter = require("./seeds/index.seeds");
11 app.use("/api/seed", seedRouter);
```

Figure 9 - Express routers

Four main router files are defined, alongside a `seedRouter` that is used for development to seed the database with test data (see Figure 9). In each call to `app.use`, which instructs the Express application to use the routes within these routers, there are two parameters – a path, and the router itself. The path (e.g. "`/api/users`" on line 3) is from the root of the URL – an HTTP GET request to `domain-name.com/api/users/index` would be handled by a GET route defined in `userRouter` with the path "`/index`".

Part of the `userRouter` file is shown in Figure 10. Three routes are shown: GET "`/current`" (which would become "`/api/users/current`" in the main app), GET "`/:id`" and POST "`/register`". These three routes highlight some core functionality. GET "`/current`" includes the passport authentication middleware – a request to this endpoint must contain an auth web token in its header, or else it will fail and respond with a `403: Unauthorized` HTTP response. GET "`/:id`" includes a parameter `id`, indicated by the colon in front of it. Any GET request to this route, with `:id` replaced by a string, will have that string interpreted as the `id` parameter within the controller method (see section 4.1.3).This route must be defined after GET "`/current`", or else the URL will be compared to this route first, and the string "current" would be interpreted as an `id` parameter. The POST "`/register`" route contains another middleware, Joi, which is a

validation library.[48] Schemas can be defined – such as `registerUserValidationSchema` here, to ensure that data in the request body is valid and can be processed in the controller method.

diary-app-server - user.router.js

```
1  const userRouter = express.Router();
2
3  const passport = require("passport");
4  const UserController = require("../controllers/user.controller");
5  const {
6    registerUserValidationSchema,
7  } = require("../validation/user.joiSchema");
8  const joiValidator = require("express-joi-validation").createValidator({
9    passError: true,
10 });
11
12 // GET currently logged in user
13 userRouter.get(
14   "/current",
15   passport.authenticate("jwt", { session: false }),
16   UserController.getCurrentlyLoggedInUser
17 );
18 // GET one user
19 userRouter.get("/:id", UserController.getUserById);
20 // POST one new user
21 userRouter.post(
22   "/register",
23   joiValidator.body(registerUserValidationSchema),
24   UserController.postUser
25 );
```

Figure 10 - User router

Each route has a path parameter, zero or more middleware parameters, and finally a controller method parameter. All routes in the `userRouter` are controlled by methods in the `UserController`, which serves the same function as controllers in MVC web frameworks.

[48] 'Joi', npm, accessed 7 October 2021, https://www.npmjs.com/package/joi; 'Express-Joi-Validation', npm, accessed 7 October 2021, https://www.npmjs.com/package/express-joi-validation.

diary-app-server - user.controller.js

```
1  const mongoose = require("mongoose");
2  const User = require("../models/User.model");
3
4  module.exports.getUserById = async (req, res, next) => {
5    try {
6      const { id } = req.params;
7      if (!mongoose.Types.ObjectId.isValid(id)) {
8        return res.status(400).send("Invalid object ID");
9      } else {
10       const user = await User.findById(id);
11       if (user) {
12         return res.send(user);
13       } else {
14         return res.status(404).send("User not found");
15       }
16     }
17   } catch (error) {
18     next(error);
19   }
20 };
```

Figure 11 - User controller, getUserById

4.1.3 Controllers

The functions in the controllers are all async, and take three parameters: the request `req`, the response `res`, and `next`, which forwards information along to other Express middleware. Figure 11 shows the `getUserById` function, which takes the id param from `:id` in the route, checks to see if it is a valid MongoDB ObjectId (and returns a `400: Bad Request` error if not), looks for a User with that `id` in the database, and returns it in the response (automatically serialised to JSON). Controller functions are async as the connection with Mongoose is over HTTP – line 10 uses the `await` keyword when calling the Promise `User.findById`, an alternative to the `.then()` structure shown earlier, which instructs the program to wait until the Promise has resolved.

The process for registering new users is shown in Figure 12; multiple steps are involved. First, the database is checked to see if any User already has the submitted email address or username, responding with a `409: Conflict` error if so. The request body – which contains the fields submitted in the registration form – is then deserialised into a User object. However, this User still has its password as the plaintext version submitted by the form. To encrypt the password, bcryptjs is used. First a `hashSalt` is generated, with a work factor of 10 (which translates to 2^{10}, or 1024 iterations); then this salt is used to encrypt the password; lastly, the now encrypted `passwordHash` is saved into the User object in place of the plaintext password. When logging in, bcryptjs compares the hash of the submitted login password with that of the stored `passwordHash` and authenticates if they match. Finally, the newly created User is saved to the database, and logged in. The `makeToken`

diary-app-server - user.controller.js

```
const bcryptjs = require("bcryptjs");
const jsonwebtoken = require("jsonwebtoken");

async function makeToken(user) {
  const payload = {
    id: user.id,
    username: user.username,
    firstName: user.firstName,
  };

  return jsonwebtoken.sign(payload, process.env.SECRET, {
    expiresIn: 21600,
  });
}

module.exports.postUser = async (req, res, next) => {
  try {
    // Check for duplicate email or username
    const existingEmailUser = await User.findOne({ email: req.body.email });
    if (existingEmailUser)
      return res
        .status(409)
        .send("A user already exists with this email address");
    const existingUsernameUser = await User.findOne({
      username: req.body.username,
    });
    if (existingUsernameUser)
      return res.status(409).send("A user already exists with this username");

    const newUser = new User({ ...req.body });

    // Create password hash
    const hashSalt = await bcryptjs.genSalt(10);
    const passwordHash = await bcryptjs.hash(newUser.password, hashSalt);
    newUser.password = passwordHash;

    // Save new user
    const savedUser = await newUser.save();

    // Sign in user after registration
    res.send({
      success: true,
      token: "Bearer " + (await makeToken(savedUser)),
    });
  } catch (error) {
    next(error);
  }
};
```

Figure 12 - User controller, postUser

method is called with the newly saved User, which extracts the `id`, `username` and `firstName` of the User and encodes it in a JSON web token, with an expiry time of 21600 seconds, or six hours. This is returned in the HTTP response.

diary-app-server - User.model.js

```
const mongoose = require('mongoose');
const Schema = mongoose.Schema;

const UserSchema = new Schema(
  {
    username: {
      type: String,
      required: true,
      unique: true
    },
    firstName: {
      type: String,
      required: true
    },
    lastName: {
      type: String,
      required: true
    },
    email: {
      type: String,
      required: true,
      index: {
        unique: true,
        partialFilterExpression: {
          email: { $type: "string"}
        }
      }
    },
    password: {
      type: String,
      required: true
    },
    birthday: {
      type: Date
    }
  },
  {
    timestamps: true
  }
);

module.exports = User = mongoose.model("User", UserSchema);
```

Figure 13 - User model and schema

4.1.4 Models

The User referred to above is a Mongoose model, which is defined by a fixed Schema (see Figure 13). The models created for this application were `User`, `Diary`, `DiaryEntry` and `Flashcard`. The User schema illustrates some key aspects of these Schemas. Fields can be deemed required, or if they must have a unique value. This resembles a table description found in a relational database, but here serves as a self-imposed constraint to ensure consistency, rather than an inescapable trait of the database system itself. The model created from this Schema is understood by Mongoose to have a connection with database records, exposing class methods such as `.findById(objectId)`, and instance methods such as `.save()`, as seen before.

4.1.5 Flashcard generation

diary-app-server - diaryEntry.controller.js

```
createEntry: async (req, res, next) => {
  try {
    const newEntry = new DiaryEntry({
      ...req.body,
      userId: req.user._id,
      diaryId: req.params.id,
    });
    const savedEntry = await newEntry.save();

    // If user has opted to create flashcards, run cardmaker service and save resultant cards
    if (req.body.makeCards) {
      // Send sanitised user data along to cardmaker
      const user = await User.findById(req.user._id);
      const sanitisedUser = Object.assign({}, user._doc);
      delete sanitisedUser.username;
      delete sanitisedUser.firstName;
      delete sanitisedUser.lastName;
      delete sanitisedUser.email;
      delete sanitisedUser.password;
      console.log(`sanitisedUser`, sanitisedUser);
      // Async code runs while sending saved diary entry to user
      runCardmakerService({
        user: sanitisedUser,
        entries: [savedEntry],
      }).then(response => {
        Flashcard.create(response.cardData)
          .then(result =>
            console.log(
              `${result.length} new flashcards saved at ${new Date()}!`
            )
          )
          .catch(err => console.error(err));
      });
    }

    return res.send(savedEntry);
  } catch (error) {
    next(error);
  }
},
```

Figure 14 - Diary entry controller, createEntry

Flashcards are generated automatically upon the submission of a new diary entry, when the `makeCards` field in the request body is set to `true`. This process is shown in Figure 14. The body of the request is deserialised into a `DiaryEntry` object, with the `userId` derived from the decoded `Authorization` header, and the `diaryId` taken from the `id` parameter of the route (“`/api/diaries/:id/newentry`”). If `makeCards` is true, a sanitised version of the authenticated User (with personally identifying information stripped out) is

created. This `sanitisedUser`, along with the saved diary entry (in an array, since the cardmaker expects an array of diary entries), is passed to `runCardmakerService`, which creates the flashcard data. This is then used to create one-or-more flashcards (see line 26). `runCardmakerService` is an async process, but in this case the controller method does not wait for it to finish before sending the saved diary entry back in the HTTP response. This allows flashcard processing to happen in the background while the user can continue interacting with the client and the API in other ways; if the cardmaker service were to be changed to a more compute-heavy, slower process, this would not slow down the process of saving new diary entries.

diary-app-server - index.services.js

```
1  const mongoose = require("mongoose");
2  const { Worker } = require("worker_threads");
3  const DiaryEntryModel = require("../models/DiaryEntry.model");
4
5  function runCardmakerService(diaryEntries) {
6    const sanitisedEntries = diaryEntries.entries.map(x => {
7      return Object.assign({}, x._doc, {
8        userId: x.userId.toString(),
9        _id: x._id.toString(),
10       diaryId: x.diaryId.toString(),
11     });
12   });
13
14   return new Promise((resolve, reject) => {
15     const worker = new Worker("./services/cardmaker.service.js", {
16       workerData: { diaryEntries: sanitisedEntries, user: diaryEntries.user },
17     });
18     worker.on("message", resolve);
19     worker.on("error", reject);
20     worker.on("exit", code => {
21       if (code !== 0)
22         reject(
23           new Error(`Cardmaker service worker stopped with error code ${code}`)
24         );
25     });
26   });
27 }
```

Figure 15 – runCardmakerService

`runCardmakerService` first creates a Worker with the `worker_threads` node library, which provides an interface for multithreaded processing in Node. This allows for CPU-

intensive JavaScript to run in parallel to the main server thread. As shown in Figure 15, the service first sanitises the diary entry data – converting ObjectId fields into strings – before creating a Promise, within which the actual worker is created and runs. The worker uses code contained in `./services/cardmaker.service.js`, a structure which allows workers to use other processing methods in future (for instance using natural language processing) without having to refactor anything beyond this reference.

diary-app-server - cardmaker.service.js

```
1  const { workerData, parentPort } = require("worker_threads");
2
3  const newFlashcardData = [];
4
5  for (const entry of parsedDiaryEntries) {
6    newFlashcardData.push( ... makeEntryCards(entry));
7    newFlashcardData.push( ... makeDateCards(entry));
8    if (entry.location) {
9      newFlashcardData.push(makeLocationCard(entry));
10   }
11   if (entry.mood) {
12     newFlashcardData.push(makeMoodCard(entry));
13   }
14   if (user.birthday) {
15     newFlashcardData.push( ... makeAgeCards(entry));
16   }
17 }
18
19 parentPort.postMessage({ cardData: newFlashcardData });
```

Figure 16 - Cardmaker, main loop

Figure 16 shows the main loop within the implemented cardmaker. Each parsed diary entry is used to call several different ‘make’ functions, appending the resulting data to an array `newFlashcardData`, which is sent back to the parent `runCardmakerService` at the end. `parsedDiaryEntries` are created by parsing the content fields of the diary entry data into a ‘sections object’, which uses regular expressions to detect the different sections of a string of Markdown-

formatted text (see Figure 17). These sections are then passed onto a further function, `detectLists`, which looks at each section to determine whether it is an unordered list.

diary-app-server - cardmaker.service.js

```
const makeSectionsObject = content => {
  const sections = {};
  const regex = /(\n\n)|(\s)/;
  const contentWords = content.split(regex).filter(x => x && x !== " ");
  let section = [];
  let sectionCount = 0;
  for (const word of contentWords) {
    if (word == `\n\n`) {
      section.push(word);
      if (section.length > 1)
        sections[sectionCount++] = {
          type: section[0].startsWith("#") ? "heading" : "paragraph",
          sectionContent: section,
        };
      section = [];
    } else {
      section.push(word);
    }
  }
  return detectLists(sections);
};

const parsedDiaryEntries = diaryEntries.map(entry => {
  return Object.assign({}, entry, {
    content: makeSectionsObject(entry.content),
  });
});
```

Figure 17 - Cardmaker, parsedDiaryEntries and makeSectionsObject

These sections are passed through several other parsing functions until they are split into individual sentences. These sentences are then processed to create cloze sentences, with gaps as prompts for users to remember what they wrote (see Figure 18). The algorithmic approach to creating these cloze sentences is straightforward. First, sentences are checked to see if any words have been italicised or bolded; if so, these are taken as the key words within the sentence and chosen for omission in the cloze sentences. If no words have been emphasised, the function loops through the entire sentence and finds all words that are longer than three characters and not included in a set of the 1200 most common words in

diary-app-server - cardmaker.service.js

```
const makeClozeSentences = sentence => {
  const splitSentence = sentence.split(" ");
  const clozeSentences = [];
  const emphasisedWords = splitSentence
    .filter(
      word =>
        word.length > 2 &&
        ((word.startsWith("*") && word.endsWith("*")) ||
          (word.startsWith("**") && word.endsWith("**")))
    )
    .filter((word, index, self) => self.indexOf(word) === index);
  if (emphasisedWords.length > 0) {
    for (const empWord of emphasisedWords) {
      const wordIndex = splitSentence.indexOf(empWord);
      const wordLength = empWord.replace("*", "").length;
      const splitCloze = [
        ...splitSentence.slice(0, wordIndex),
        makeClozeOfLength(wordLength),
        ...splitSentence.slice(wordIndex + 1),
      ];
      clozeSentences.push(splitCloze.join(" "));
    }
  } else {
    const filteredSentence = splitSentence.filter(
      (word, index, self) => self.indexOf(word) === index
    );
    for (const word of filteredSentence) {
      if (
        !word.startsWith("#") &&
        word.length > 3 &&
        (!MostCommonWordsSet.has(strippedWord(word).toLowerCase()) ||
          strippedWord(word)[0].toLowerCase() !== strippedWord(word)[0])
      ) {
        const wordIndex = splitSentence.indexOf(word);
        const wordLength = word.length;
        const splitCloze = [
          ...splitSentence.slice(0, wordIndex),
          makeClozeOfLength(wordLength),
          ...splitSentence.slice(wordIndex + 1),
        ];
        clozeSentences.push(splitCloze.join(" "));
      }
    }
  }

  return clozeSentences;
};
```

Figure 18 - Cardmaker, makeClozeSentences

the English language; these are then chosen as candidates for cloze sentences.[49] In either case, the chosen words are removed from the original sentence and replaced with a cloze gap of underscores equal in length to the omitted word. These sentences are appended to an array, which is returned by the function.

Other cards produced by the Cardmaker include date cards, which prompt the user to remember when a particular diary entry excerpt took

[49] Derived from Josh Kaufman, *Google-10000-English/Google-10000-English.Txt*, 2021, https://github.com/first20hours/google-10000-english/blob/d0736d492489198e4f9d650c7ab4143bc14c1e9e/google-10000-english.txt.

place; mood cards, which prompt the user to remember how they felt on a certain day; and location cards, which prompt to remember where a memory took place.

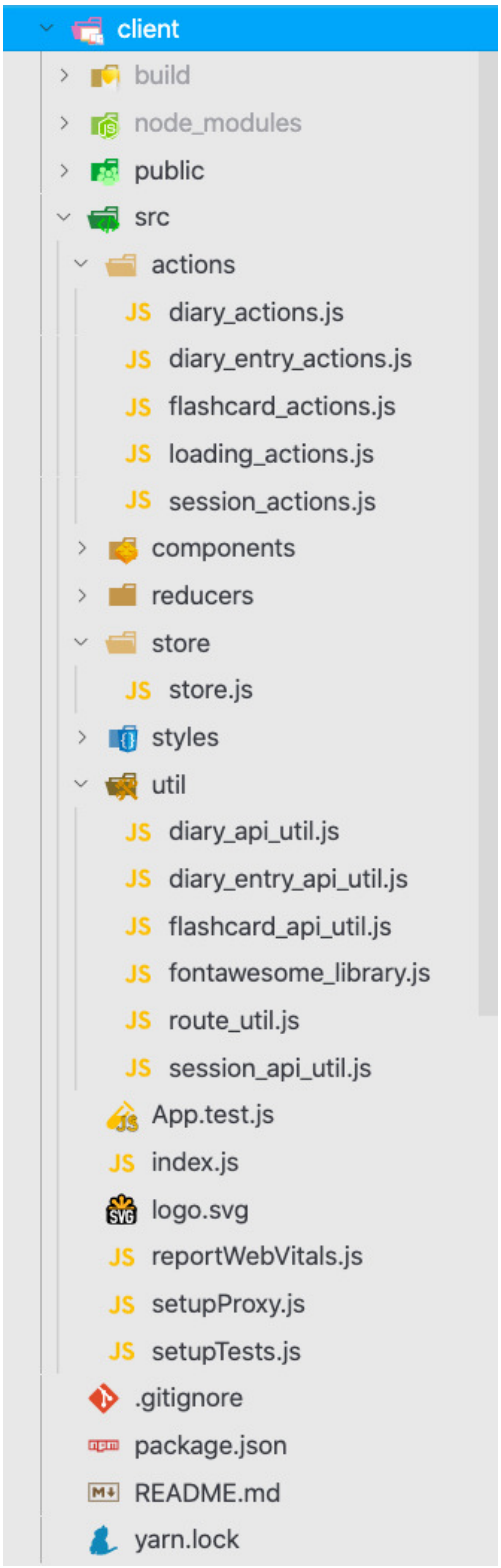

Figure 19 - React client, file structure

4.2 Frontend Client

As mentioned above, the React client was contained within a /client sub-directory within the overall project root. The file structure of the frontend client is shown in Figure 19. This structure was chosen as the client and server are quite tightly coupled; however, as a standalone application, the frontend could exist separately within a different Git repository and deployment pipeline. Since the frontend and backend only communicate through a RESTful API, the client could theoretically be refactored to call endpoints of a different server with related endpoints; in the same way, other frontend clients (such as other websites, mobile applications, or plugins to other larger web services) could call the API endpoints of the web server to retrieve the same data and manipulate/display it in different ways.

As a standalone application, the frontend client also contains a separate `package.json` file, which contains the dependencies for the React project, as well as scripts to build the production application or run it in development mode. Rather than using NPM, many modern React projects (including this one) use Yarn as a package and script manager.[50] It is functionally similar to NPM, and uses the same `package.json` syntax and package repositories; indeed, its scripts are largely interoperable.

[50] 'Yarn', Yarn, accessed 7 October 2021, https://classic.yarnpkg.com/en/.

4.2.1 React setup and routing

Figure 20 shows the basic initial setup for the React application. This index.js file serves as the entry point for the overall app. An event listener – a type of JavaScript function that allows for a callback (another function) to be run when a particular event occurs – is added to the document object in the browser, which is the root node of the DOM. The browser's `localStorage`, which stores files and cookies for individual web pages on a local machine, is checked for an existing JSON web token, and the authenticated User is decoded if one is found and added to a `preloadedState` object, which represents some data that will be sent to Redux state management. A method `configureStore()` is called (either with this preloaded state if a web token is found, or not), which sets up the Redux data store.

diary-app-server - index.js

```
1  import React from 'react';
2  import ReactDOM from 'react-dom';
3  import jwt_decode from 'jwt-decode';
4  import { setAuthToken } from "./util/session_api_util";
5  import configureStore from "./store/store";
6  import Root from "./components/root";
7  import { logout } from "./actions/session_actions";
8  import "./util/fontawesome_library";
9  import "./styles/main.scss";
10
11
12 document.addEventListener("DOMContentLoaded", () => {
13   let store;
14
15   if (localStorage.jwtToken) {
16     setAuthToken(localStorage.jwtToken);
17     const decodedUser = jwt_decode(localStorage.jwtToken);
18     const preloadedState = {
19       session: { isAuthenticated: true, user: decodedUser },
20     };
21     store = configureStore(preloadedState);
22     const currentTime = Date.now() / 1000;
23
24     if (decodedUser.exp < currentTime) {
25       store.dispatch(logout());
26     }
27   } else {
28     store = configureStore();
29   }
30
31   const root = document.getElementById("root");
32
33   ReactDOM.render(<Root store={store} />, root);
34 });
```

Figure 20 - React application setup

diary-app-server - root.js

```
1  import React from "react";
2  import { Provider } from "react-redux";
3  import { HashRouter } from "react-router-dom";
4  import App from "./app";
5
6  const Root = ({ store }) => (
7    <Provider store={store}>
8      <HashRouter>
9        <App />
10     </HashRouter>
11   </Provider>
12 );
13
14 export default Root;
```

Figure 21 - React, Root component

Finally the root of the HTML document is found – a `<div>` element with the `id` of 'root', which is created during the build process – and ReactDOM's render class method is called. This method takes two parameters – a React component, and an HTML element in the DOM – and renders the React component in the place of the HTML element. In this way, on line 33 the DOM is replaced by the `<Root />` component, which itself contains the rest of

the application nested within it (see Figure 21). The Redux store is passed to this component as a prop. The Root component is a utility component that wraps <App /\>, which contains the actual business logic of the application, in a React Redux <Provider/\>, which allows nested components to access the Redux store and to dispatch actions; and a React Router <HashRouter /\>, which provides access to the URL for path routing.

diary-app-server - app.js

```
<Switch>
  <AuthRoute
    loggedIn={props.loggedIn}
    exact
    path="/login"
    comp={LoginFormContainer}
  />
  <AuthRoute
    loggedIn={props.loggedIn}
    exact
    path="/register"
    comp={RegisterFormContainer}
  />
  <ProtectedRoute
    loggedIn={props.loggedIn}
    path="/diaries"
    comp={DiariesIndex}
  />
  <ProtectedRoute
    loggedIn={props.loggedIn}
    path="/study-menu"
    comp={StudyDiariesIndex}
  />
  <Route
    path="/"
    render={props => (
      <HomePage
        openSessionForm={openSessionForm}
        logoutUser={logoutUser}
        loggedIn={props.loggedIn}
        user={props.user}
      />
    )}
  />
</Switch>
```

Figure 22 - React, main application routing

It is called a 'hash' router as it can access everything after the hash (#) in a URL. For instance, a route defined within the application for '/login' will be accessible at the URL domain-name.com/#/login. Routing is achieved using a <Switch /\> component from React-Router. As shown in Figure 22, routes are defined for several paths. '/login' and '/register' are <AuthRoute /\> components, defined within the application to only be accessible to Users who are not logged in, redirecting them to the homepage if they are. Conversely, a <ProtectedRoute /\> can only be accessed if authenticated, and redirects to the '/login' route. The comp prop for each of these Routes is the React component that will be rendered here if the route is accessed. The last route has no authentication rules, and renders the <HomePage /\> if none of the other routes are matched.

4.2.2 Redux: actions, reducers and store

This section will discuss the implementation of state management and the interactions between the React app and the backend API. As discussed in section 3.3.2, Redux allows API interactions to occur outside of individual components using actions.

Figure 23 shows an excerpt of the actions used to interact with diary routes in the API. HTTP requests formed in Axios are imported here as `DiaryAPIUtil` (see Figure 24). In the `fetchAllDiaries` function, this request is made in a `.then().catch()` block (as an HTTP request it is async). The diaries received in the HTTP response are then used to call the `receiveDiaries` method, which makes an action object with a type (defined by constants at the top of the file) and any other arbitrary keys, in this case the diaries data. This action object is itself used in the dispatch call, which is provided in the `<Provider />` parent component and accessed by React. This dispatch function is how data from HTTP responses is passed to the Redux store. Dispatches can occur for other reasons, however; here in this method, a `diariesLoadingOn` dispatch is made before the HTTP request is called, which can be used to inform the wider system that a request has been made but the resulting data is not yet available. A corresponding `diariesLoadingOff` dispatch is made after the request is

diary-app-server - diary_actions.js

```
import { diariesLoadingOff, diariesLoadingOn } from "./loading_actions";
import * as DiaryAPIUtil from "../util/diary_api_util";

export const RECEIVE_DIARIES = "RECEIVE_DIARIES";
export const RECEIVE_DIARY_ERRORS = "RECEIVE_DIARY_ERRORS";

const receiveDiaries = diaries => ({
  type: RECEIVE_DIARIES,
  diaries,
});

const receiveDiaryErrors = errors => ({
  type: RECEIVE_DIARY_ERRORS,
  errors,
});

export const fetchAllDiaries = () => dispatch => {
  dispatch(diariesLoadingOn());
  DiaryAPIUtil.getAllDiaries()
    .then(diaries => {
      dispatch(receiveDiaries(diaries.data));
      dispatch(diariesLoadingOff());
    })
    .catch(err => {
      dispatch(receiveDiaryErrors(err.response.data));
      dispatch(diariesLoadingOff());
    });
};
```

Figure 23 - React, diary actions

diary-app-server - diary_api_util.js

```
import axios from "axios";

export const getAllDiaries = () => axios.get("/api/diaries/");
```

Figure 24 - React, diary API utils

diary-app-server - diaries_reducer.js

```
1  import {
2    RECEIVE_DIARIES,
3  } from "../../actions/diary_actions";
4
5  const DiariesReducer = (state = {}, action) => {
6    Object.freeze(state);
7    switch (action.type) {
8      case RECEIVE_DIARIES:
9        const diariesOutput = {};
10       action.diaries.forEach(diary => {
11         diariesOutput[diary._id] = diary;
12       });
13       return Object.assign({}, diariesOutput);
14     default:
15       return state;
16   }
17 };
18
19 export default DiariesReducer;
20
```

Figure 25 - React, diaries reducer

fulfilled. This prevents components from trying to access data that does not yet exist within the Redux store.

Before dispatched data can be useful to other components, it must first be processed and incorporated into the current data store of the application. For this, the Redux store is configured with a 'reducer' that manages the state of the data in the application in a JavaScript object; this reducer is in turn is composed of several other reducers, each of which manage a 'slice' of the state. The reducer (and its component reducers) receives two parameters when a dispatch is received; the current `state` (or the default value if no state has been created yet), and the `action` object. Each reducer contains a switch statement, which runs some code if the `action.type` is one of the defined cases. This code creates a new object that is returned to Redux to replace the old slice of state. For example, on lines 9-13, an empty `diariesOutput` object is created; it iterates over the array of diaries from the API response and assigns each diary to a key in `diariesOutput`, the key being the diary's ObjectID. This object is then returned to represent the 'diaries' slice of the application state. Components that wish to access a diary with a particular ID can look it up within the state object.

The root reducer, which collates all the slices of state created by other reducers, is passed to the Redux `createStore` function, along with any preloaded state from existing web tokens. This is the store passed to the `<Root />` component of the overall application.

diary-app-server - store.js

```
1  import { createStore } from "redux";
2  import rootReducer from "../reducers/root_reducer";
3
4  const configureStore = (preloadedState = {}) =>
5    createStore(
6      rootReducer,
7      preloadedState,
8    );
```

Figure 26 - React, configureStore

4.2.3 Component design and modularity

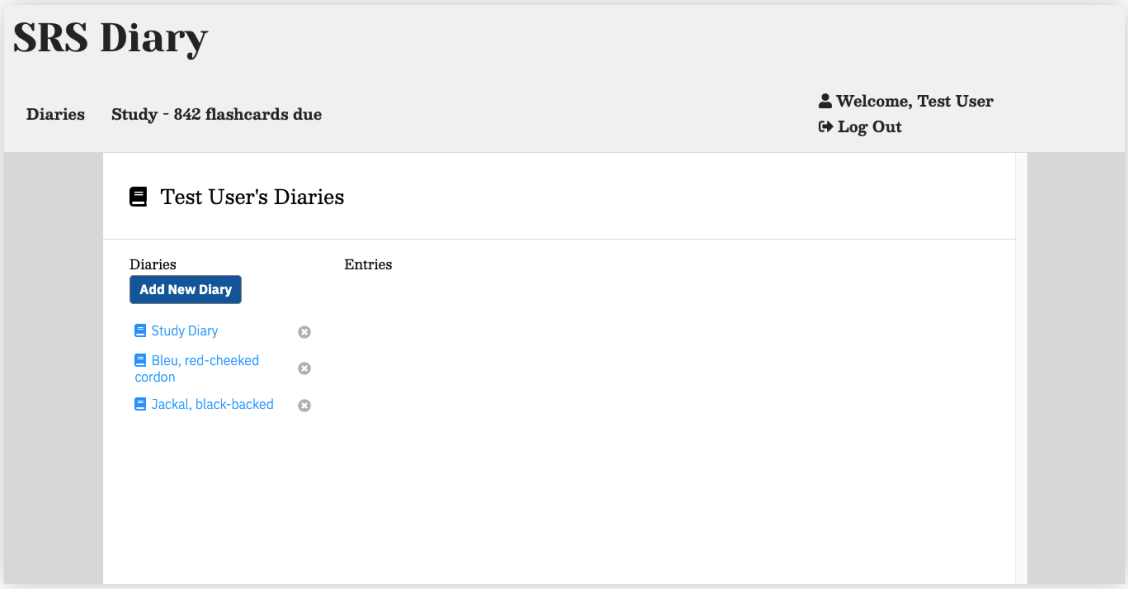

Figure 27 - React, diaries index (/#/diaries)

Once logged in, the main diaries index is displayed. The view is split into three columns, with diaries listed on the left, diary entries in the centre and individual entries displayed on the right. Figure 27 shows the initial index, and Figure 28 shows a diary entry displayed.

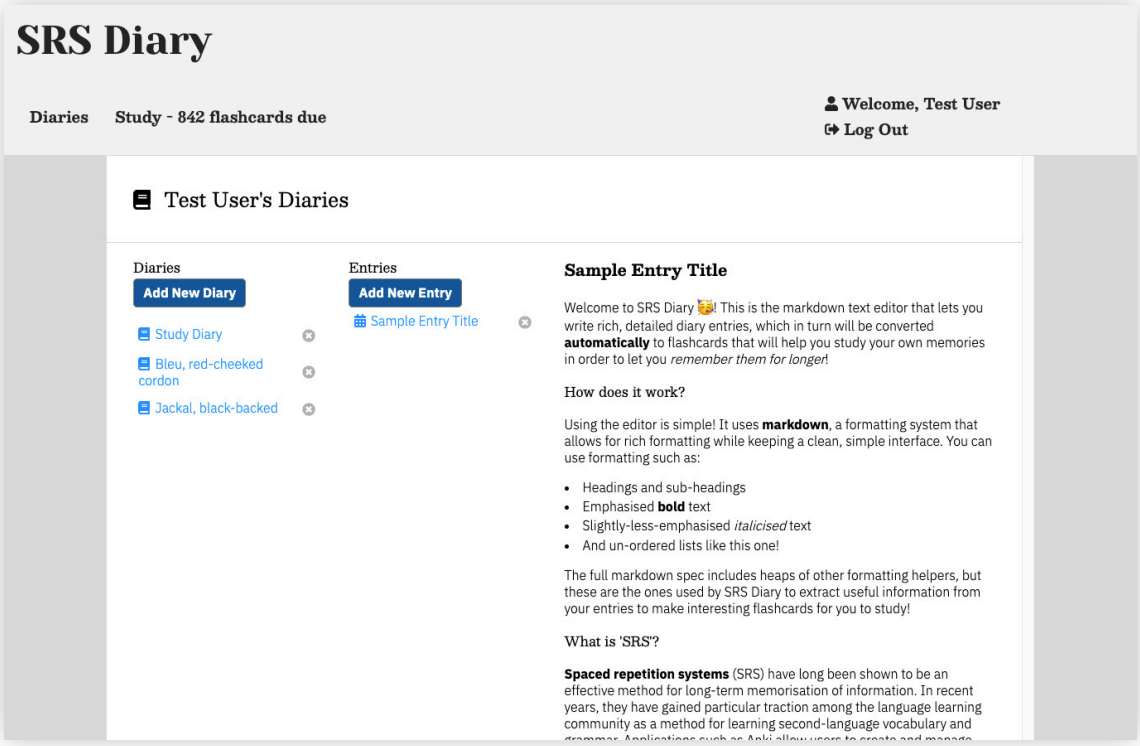

Figure 28 - React, diary entry show (/#/diaries/:diaryId/entry/:entryId)

diary-app-server - diaries_index.js

```
<ul>
  {Object.values(stateDiaries)
    .reverse()
    .map((diary, index) => (
      <DiaryIndexItem
        diary={diary}
        key={`${diary._id}-diary-${index}`}
        deleteDiary={this.handleDeleteDiary(diary._id)}
      />
    ))}
</ul>
```

Figure 30 - React, diaries index generation

diary-app-server - diary_index_item.js

```
export default function DiaryIndexItem({
  diary,
  study,
  deleteDiary,
  dueFlashcardCount,
}) {
  const [showDelete, setShowDelete] = useState(false);
  let link = `/diaries/${diary._id}`;
  if (study) link = `/study-menu/diary/${diary._id}`;
  return (
    <li className={study && "study-index-item"}>
      <div>
        <Link to={link}>
          <FontAwesomeIcon icon="book" />
          <span>{diary.title}</span>
        </Link>
        {!study && (
          <button
            className={"delete-button" + (showDelete ? " show" : "")}
            onClick={e => setShowDelete(!showDelete)}
          >
            <FontAwesomeIcon icon="times-circle" />
          </button>
        )}
      </div>
      {showDelete && (
        <div className="delete-div">
          <span>Delete this diary?</span>
          <button className="confirm-button" onClick={deleteDiary}>
            Yes
          </button>
          <button
            className="confirm-button"
            onClick={e => setShowDelete(!showDelete)}
          >
            No
          </button>
        </div>
      )}
    </li>
  );
}
```

Figure 29 - React, diary index item

This view is constructed from numerous React components that programmatically display data from the Redux state. As an example, within the parent `<DiariesIndex/>` component, there is an unordered list that maps over the values in the diaries slice of state, and creates a `<DiaryListItem />` component for each (see Figure 30). This component creates an `<li>` element that contains data about the individual diary and provides interactions to change the current URL location in the browser. This component also contains some internal state, which toggles whether the delete menu is shown, depending on whether the user clicks the button defined on lines 18-23. If this state value is true, the elements on lines 27-38 are added to the DOM.

The diary entry show component, on the right-hand column of Figure 28, shows the content of the diary entry. It uses a `<ReactMarkdown />` component from the react-markdown library, that converts Markdown formatted text into HTML elements.

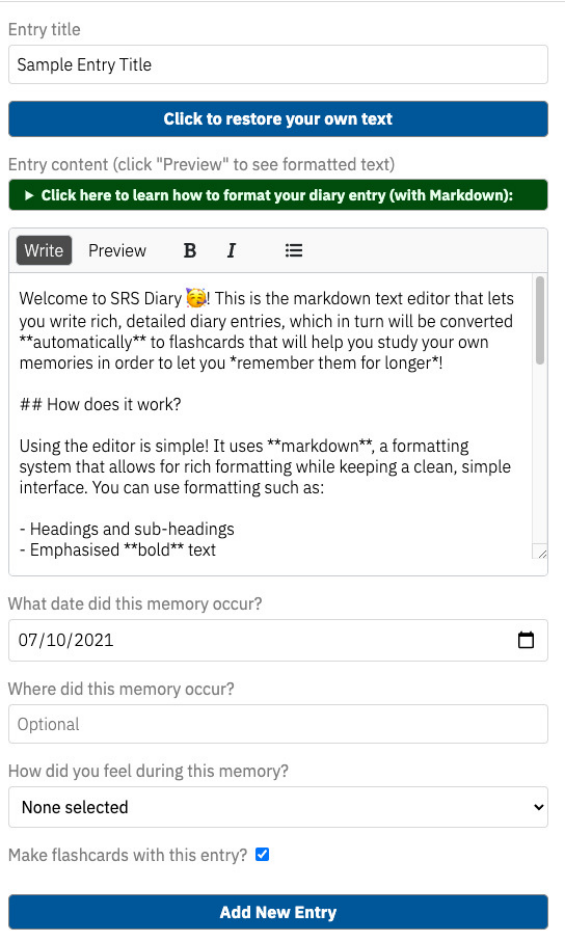

Figure 32 - React, markdown text editor, write view

important information from these entries.

The entry form also includes fields for an entry title, date, location and mood (enumerated from a hardcoded list). This metadata is used to generate other types of cards, as discussed in section 4.1.5.

Diary entries are created using a Markdown text editor imported from the React-MDE library. Figure 32 and Figure 31 respectively show the write and preview views of the editor. This allows for users to write their freeform text content in the write panel, and see the current formatted state of their entry in the preview pane, as it would be represented in the diary show view of the main diaries index view. As mentioned earlier, formatting is used as context clues for the Cardmaker service to derive the most

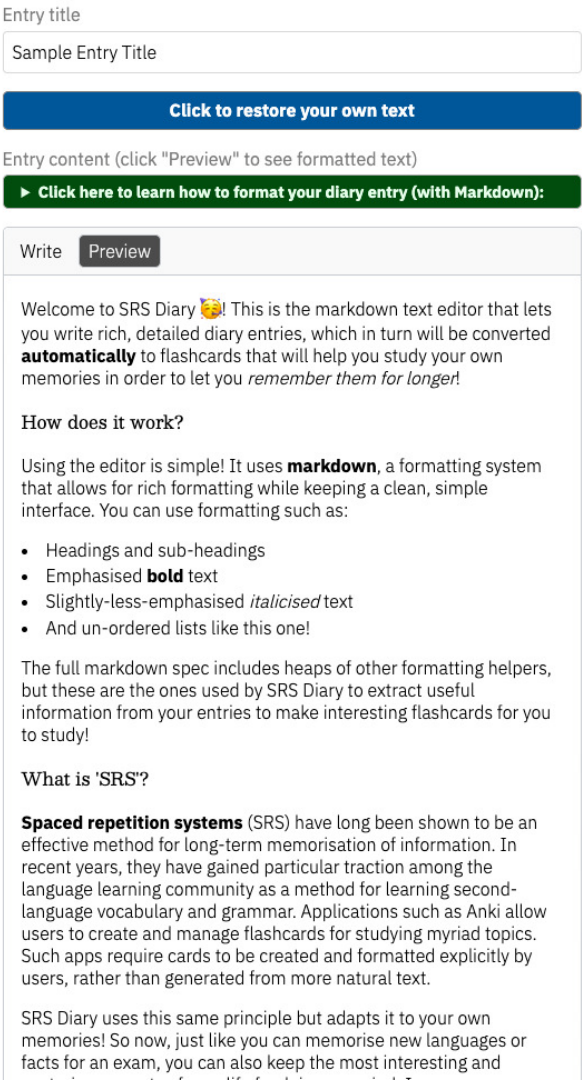

Figure 31 - React, markdown text editor, preview view

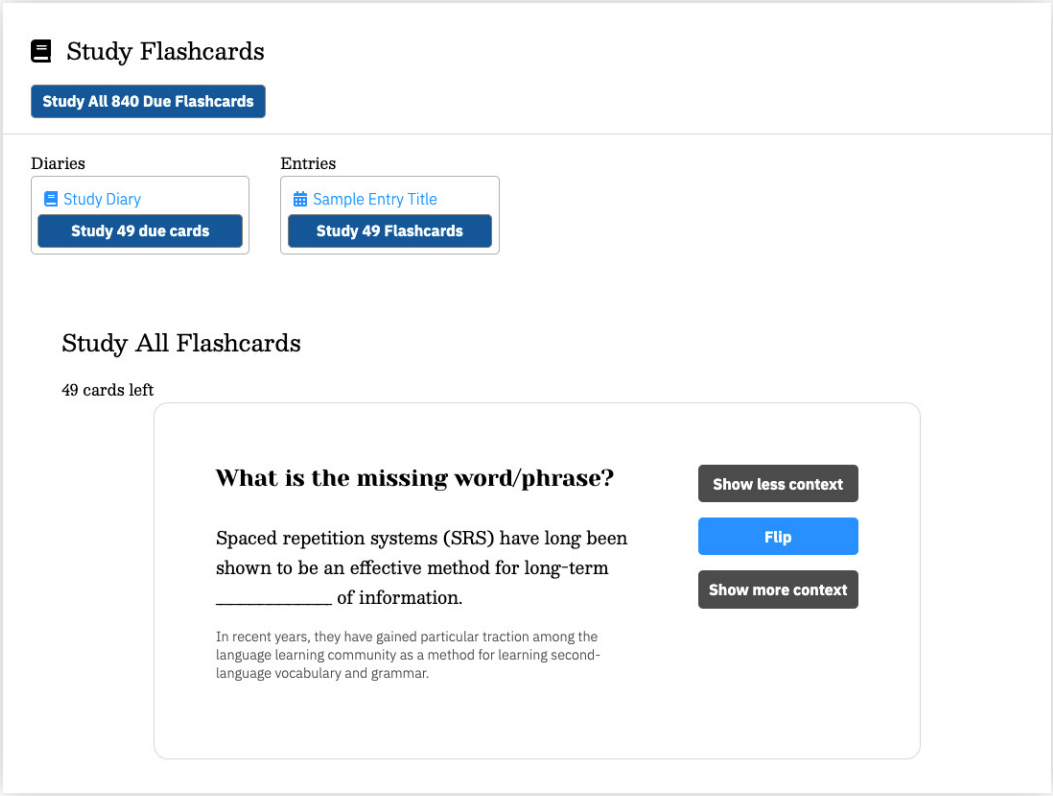

Figure 33 - React, flashcard front

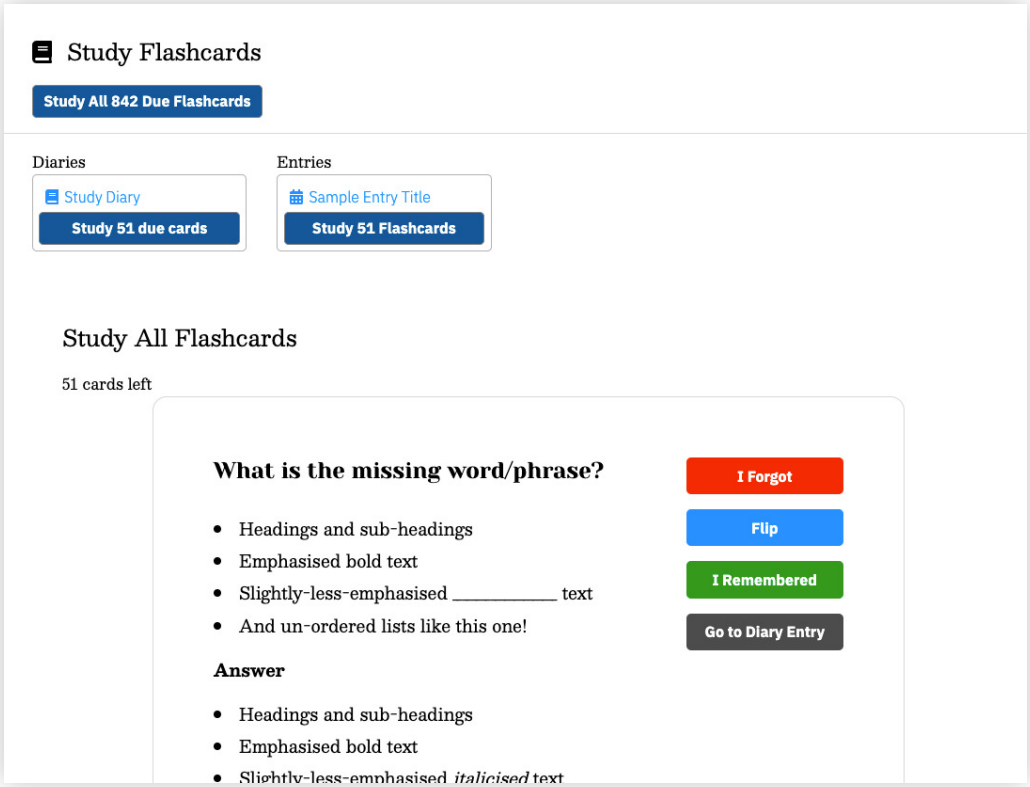

Figure 34 - React, flashcard back

Figure 33 and Figure 34 show the flashcard studying interface. At the top of the view (which is still nested under the site header shown in Figure 27, cropped out of this screenshot) is a similar three-column navigation menu to the diaries index. Beneath each diary or entry is a button that displays a count of all the flashcards that are currently scheduled to be studied for that element. If a diary has an entry with 10 cards due, and another with 15 cards due, the diary will show 25 cards due. Each flashcard – contained in a `<StudyCard />` component – has a front and back, toggled using the Flip button. If the user has remembered the information on the back of the card, they can click 'I Remembered'; otherwise they can click 'I Forgot', which will reset the `correctStreak` of the card and its scheduling.

diary-app-server - study_card.js

```
rememberedCard(e) {
    e.preventDefault();
    const { card, studyFlashcard, nextCard } = this.props;
    const daysSinceDue = moment().diff(moment(card.nextDue), "days");
    const bonusStreak = Math.floor(Math.sqrt(daysSinceDue)) + 1;
    const newStreak = Math.max(bonusStreak, card.correctStreak + 1);
    const nextDue = moment().add(Math.pow(2, newStreak), "days");
    const updatedData = {
      correctStreak: newStreak,
      nextDue: nextDue.toISOString(),
    };
    console.log('updatedData :', updatedData);
    studyFlashcard(card._id, updatedData);
    nextCard();
  }

  forgotCard(e) {
    e.preventDefault();
    const { card, studyFlashcard, nextCard } = this.props;
    const newStreak = 0;
    const nextDue = moment().add(Math.pow(2, newStreak), "days");
    const updatedData = {
      correctStreak: newStreak,
      nextDue: nextDue.toISOString(),
    };
    studyFlashcard(card._id, updatedData);
    nextCard();
  }
```

Figure 35 - React, card answer functions

Figure 35 shows the functions attached to these buttons. A remembered card increases its `correctStreak` by the formula $s = \max(\sqrt{d} + 1, c + 1)$ where n is the new `correctStreak`; d is the number of days between the current date and when this card became due; and c is the current value for the `correctStreak`. This is used to calculate the next `dueDate` for the card, defined as 2^s days from the current date. This creates an exponential effect by which the increase of the interval between each revision of a card grows larger as cards become easier to remember. This is the implementation of spaced-repetition within the application.

4.2.4 Sass stylesheets

The app was styled with Sass, which is a pre-processor for cascading style sheets (CSS), a fundamental building block of all modern websites.[51] Sass allows for convenient

[51] 'Sass: Syntactically Awesome Style Sheets', accessed 7 October 2021, https://sass-lang.com/.

diary-app-server -

```sass
.study-card {
  max-width: 50rem;
  margin: 0 auto;
  padding: 4rem;
  border: 1px solid $graylighter;
  border-radius: 15px;
  display: flex;

  .card-heading {
    margin-bottom: 2rem;
    font-size: x-large;
    font-family: $display;
  }

  .front-side,
  .back-side {
    margin: 1rem 0;
    font-size: large;
    font-family: $serif;

    h2 {
      font-weight: 700;
    }
  }
}
```

Figure 36 - Sass, stylesheet for flashcards

functionality such as variables and mixins, providing a more programmatic approach to the creation of stylesheets. Unlike pure CSS files, which can be written and read directly by the browser, Sass stylesheets must first be compiled into regular CSS, which is what the browser interprets. Figure 36 shows an excerpt of the Sass sheet for the `<StudyCard />` component. Sass allows nesting within stylesheets – which is not currently available in pure CSS, although it is likely to be possible in the near future[52] – which provides for a clean inheritance structure for child elements within the DOM. Here, the `.card-heading` selector only applies to an `<element class="card-heading">` that is the child of `<element class="study-card">`. Also, note the use of variables such as `$graylighter` and `$display`; both are defined in other Sass sheets and allow for the site-wide alteration of attribute values from a single source of truth.[53]

[52] 'CSS Nesting, Specificity and You | Kilian Valkhof', 4 August 2021, https://kilianvalkhof.com/2021/css-html/css-nesting-specificity-and-you/.

[53] CSS variables are currently available within pure CSS as well, known as custom-properties, though their use and implementation differs substantially to Sass.

5. Testing

5.1 Test Driven Development

Test-driven development (TDD) is an approach to software engineering that privileges the writing of automated tests throughout the development process, in order to ensure interoperability of code and the consistency of results regardless of changes and refactors. Each individual piece of functionality of the code is tested in isolation as it is developed, also known as unit tests (a unit being an atomic piece of code, such as a statement, function, class or module); then the overall interoperability between different code elements is also tested, in integration tests (which can be across classes, modules or even entire services and applications).

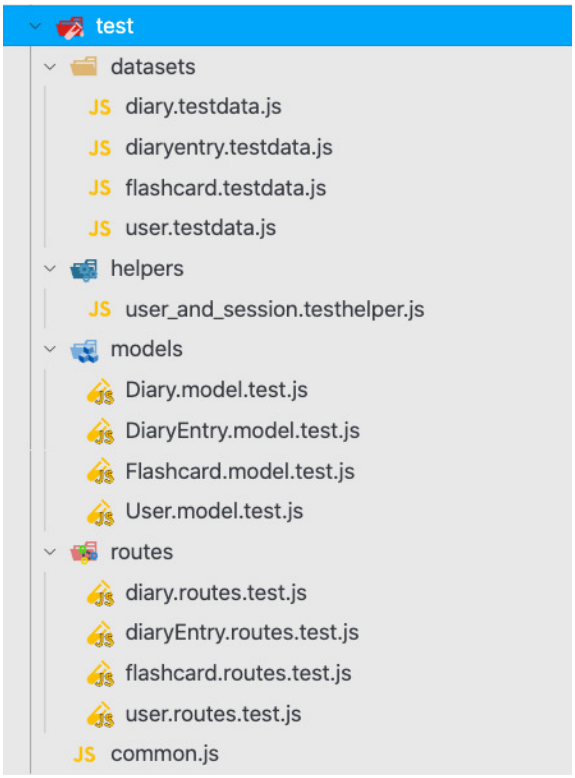

Figure 37 - Test file structure

TDD was used throughout the development of the backend server; the file structure for the tests is shown in Figure 37. Several JavaScript libraries were used specifically for testing: Mocha, a test framework that allows for the simple description of test cases; Chai, which provides a straightforward interface for otherwise complex assertions within tests; and Supertest, which allows for the testing of HTTP requests within an Express application without using a live web server.[54] MongoDB was installed on the local development environment to allow for a test database to be created and used whilst tests are running.

[54] 'Mocha - the Fun, Simple, Flexible JavaScript Test Framework', accessed 7 October 2021, https://mochajs.org/; 'Chai', accessed 7 October 2021, https://www.chaijs.com/; 'Supertest', npm, accessed 7 October 2021, https://www.npmjs.com/package/supertest.

Hooks were used to set up the test database, clear the data before each test (to avoid polluting results with unpredictable data), and deleting the test database upon the conclusion of testing (see Figure 38). The test database exists on localhost; first the NODE_ENV is changed to 'test', which prevents the Express app from making a connection to the main production database (see line 1 of Figure 5, page 38). Then a mongoose `connection` is made to the local MongoDB server.

diary-app-server - common.js

```
1  process.env.NODE_ENV = "test";
2
3  before(done => {
4    const options = {
5      useNewUrlParser: true,
6      useUnifiedTopology: true,
7      useCreateIndex: true
8    };
9    mongoose.connect("mongodb://localhost/diary_test", options);
10   mongoose.connection
11     .once("open", () => done())
12     .on("error", error => console.warn("warning", error));
13 });
14
15 beforeEach(async () => {
16   const collections = mongoose.connection.collections;
17
18   for (const key in collections) {
19     const collection = collections[key];
20     await collection.deleteMany();
21   }
22 })
23
24 after(async () => {
25   await mongoose.connection.dropDatabase();
26   await mongoose.connection.close();
27 });
```

Figure 38 - Testing, common.js

Figure 39 shows an excerpt of the user routes tests. The first tests that a user is returned when GET "`/api/users/:id`" is called with a valid ObjectId. The test contains two expect statements: that the HTTP response is `200: OK`, and that the response body contains a `username` property, indicative of a `User` model. The other two tests check for expected behaviour on erroneous requests: that the server responds with `400: Bad Request` if the ObjectId is malformed, and with `404: Not Found` if the ObjectId is valid, but does not match any records in the database.

diary-app-server - user.routes.test.js

```
const app = require("../../app");
const request = require("supertest");
const { expect } = require("chai");
const TestUsers = require("../datasets/user.testdata");
const User = require("../../models/User.model");

describe("api/users", () => {
  describe("GET /:id", () => {
    it("should return a user if a valid ID is passed", async function () {
      const user = new User(TestUsers.testUser);
      await user.save();
      const res = await request(app).get("/api/users/" + user._id);
      expect(res.status).to.equal(200);
      expect(res.body).to.have.property("username", "testuser");
    });

    it("should return 400 if invalid objectId is passed", async function () {
      const res = await request(app).get("/api/users/9");
      expect(res.status).to.equal(400);
    });

    it("should return 404 if a valid but non-existent objectId is passed", async function () {
      const res = await request(app).get("/api/users/610d3757fc13ae5d51000514");
      expect(res.status).to.equal(404);
    });
  });
});
```

Figure 39 - User routes tests

The output of the tests include the responses from supertest (see Figure 40). It displays the exact HTTP request made, the response status, and the time it took for the response to be returned after making the request.

```
  api/users
    GET /:id
GET /api/users/615e5bf9d3d9517490406347 200 1.941 ms - 263
      ✔ should return a user if a valid ID is passed
GET /api/users/9 400 0.296 ms - 17
      ✔ should return 400 if invalid objectId is passed
GET /api/users/610d3757fc13ae5d51000514 404 1.000 ms - 14
      ✔ should return 404 if a valid but non-existent objectId is passed
```

Figure 40 - Test output, user routes

5.2 User Testing

In order to test the user experience of the application, live user testing was also used. A participant recruitment letter (see page 83) was distributed on social media asking for

participants, with the recruitment post directing them to a participant consent form.[55] Through this process, three participants were recruited for the study, who will be referred to as:

- pA (female, aged 50-64)
- pB (female, aged 18-34)
- pC (male, aged 50-64).

Participation was split into two parts. First, participants were asked to talk through their thought processes and experiences as they used the website to perform specific tasks given to them by the researcher. After this, participants were asked to answer a questionnaire (see page 84) about their experiences with productivity software in general and the application in particular. During the first part, participants were asked to:

- Register a new account
- Log out and log back in
- Create a diary
- Create a diary entry
    - Bold some text
    - Add a sub-heading
- Add a diary entry using the sample text
- Study flashcards generated from their entry

All participants were instructed not to use any personally identifiable information throughout the UX test, including in the registration and diary entry writing processes.

5.2.1 Key findings

Design and layout

All three participants found the design and layout of the website to be clear and relatively intuitive. When asked to register a new account, all were able to find the Register button, fill out the registration form, and register their account. pA was uncomfortable entering date of birth information (regardless of the anonymised fake information used in the test), and felt it was unclear why the website needed that information. All participants were likewise able

[55] 'Participation Information and Consent Form', Google Docs, accessed 7 October 2021, https://docs.google.com/forms/d/e/1FAIpQLSfwQxM2Fpif9esq0TNJSXXpiMJzxzR5mZUN_sYqZQTDhRqL9Q/viewform?usp=embed_facebook.

to log out and back into the website with no issue. When asked to navigate between diaries, diary entries and flashcards, participants were able to find the required areas of the website with little issue and no need for further prompting. Two participants understood the 'title' field in the New Diary form to be prompting for a diary title, but pC mistook this to be asking for his own title ('Mr') – furthermore, pB thought of the New Diary as being an entry rather than a collection of entries, and titled their diary as such. Clearer labelling and hints may help to alleviate these issues.

All participants commented on the grey appearance of the website, particularly its homepage, commenting that more colour would increase its appeal.

Markdown editing

Unfortunately, all participants found formatting with the Markdown text editor to be confusing to use at first. The diary entry form as a whole was simple to use, with the entry title, location, mood and date fields being clear to understand. Participants also had little issue with entering plain text within the textarea field of the Markdown editor. However, when asked to make some text bold, there was confusion – after selecting a word and clicking the 'bold' button on the taskbar, the word was surrounded by asterisks, which was surprising to participants. pB mentioned that she had seen this before in other contexts, but understood the asterisks themselves to be the desired result, rather than a representation of formatting yet to be applied/displayed. Participants did not consider clicking the Preview button until directly prompted to look for such functionality, at which point they saw their selected word formatted in bold as initially expected.[56]

When asked to insert a sub-heading, participants were similarly confused. It was not until direct prompting that they considered clicking on the 'learn how to format your diary entry' helper tab, which informed them about the Markdown formatting options available. After seeing this panel, they were able to insert a sub-heading (although needed extra help

[56] There is also a bug that means that if a word is selected along with whitespace before or after it, and then made bold, the inserted asterisks encapsulate the whitespace (e.g. resulting in **hello ** rather than **hello**). This whitespace breaks the parsing of the Preview pane, meaning 'hello' is not rendered in bold. This is another source of confusion, and one that likely requires a software solution.

adding a space after the # symbols to help the parser along, which is something that ought to be clarified in the help panel).

When asked to create an entry with the sample text, and given the opportunity to compare the Markdown version with the formatted Preview version, pA and pB reported that they were able to better understand the relationship between the Markdown symbols and the resultant formatting. pA specifically mentioned that seeing the sample text before attempting her own entry would have made the process clearer to understand. All participants considered the process of writing Markdown to be something they could get used to now that they had attempted it. In post-test questionnaires, none of the participants said they had used Markdown before. Much more information must be presented to new users in order for them to understand the process of using Markdown comfortably. pA suggested that the editor be replaced with a WYSIWYG editor, that parses and displays formatting live, without having to switch to the Preview pane.

Flashcards

All users reported enjoying using the flashcards, and found them to be a surprisingly fun way to interact with their diary entries. pA and pC, upon seeing a cloze-deletion card, initially expected to be able to type their answer into the gap in the sentence and submit it. However, after a brief explanation that the flashcards are for self-reported memory training, they felt happy with the present implementation. pA felt that there was an absence of feedback from the 'Remembered' and 'Forgot' buttons, which obscured what the impact of clicking one or the other was. As the algorithmic approach to interval expansion is built into the `<StudyCard />` component, this information would be trivial to surface and would be engaging for users.

All participants recognised the utility of the application, and reported that it would be a good way to remember the information in their diary entries. pA imagined that having to click 'Forgot' for flashcards would be demoralising, and might put her off using the study feature, but also recognised that there would be a complementary sense of accomplishment from remembering flashcards.

Questionnaire results

All participants reported using productivity software for work, particularly calendars and appointment diaries. Participants thought of diaries as being prospective data stores –

rather than using them to record past events, they use them to remind themselves of future events or obligations. None of the participants had used space-repetition programs before, but understood the principles behind it (in particular pA and pB). None had used Markdown for text formatting.

All participants considered the Markdown editor to be the most frustrating aspect of the software, despite generally feeling that they could ultimately get used to using it. As a fairly uncommon interface for writing formatted text, particularly when many apps use WYSIWYG editors, this is likely to be a significant issue that would need to be addressed in future versions of the program.

6. Evaluation and Conclusion

The aim of this project, as outlined in section 1.2.1 on page 10, was to create a website that allows users to create an ongoing diary with rich text formatting, whose entries can be converted into flashcards to be studied using spaced-repetition training. While establishing the efficacy of this application as a memory-enhancement tool is well beyond the scope of this project, it is important to reflect on the implementation of this aim in software.

6.1 Achievement of acceptance criteria

Feature	Acceptance Criteria Achieved?	Notes
Functional Requirements		
Authentication Users need to be able to register and log in *Acceptance criteria:* The registration and log-in functions are clearly visible and usable from the home screen.	✓	
Single Sign-On Users should be able to log into the system once to access all parts of the program. *Acceptance criteria:* After logging in, the user can access the diary and study pages.	✓	
User Accounts Users can have user accounts, to organise all their data. *Acceptance criteria:* Users have exclusive access to their own diaries, entries and flashcards, and cannot see those made by other users.	✓	

Feature	Acceptance Criteria Achieved?	Notes
Diary and Entry Creation Users need to be able to create diaries and write diary entries *Acceptance criteria:* Users can create a diary and a diary entry for that diary.	✓	
Markdown Text Editor Users can create freeform diary entries using a markdown text editor *Acceptance criteria:* The interface for creating a diary entry allows for rich text, and the ability for users to preview their entries before posting them.	✓	
Diary Entry Metadata Metadata like dates, locations and mood should be added with entries. *Acceptance criteria:* Users can add a date, location and mood to each diary entry during creation.	✓	
Diary Viewing Users must be able to read through their diary like a traditional chronological diary *Acceptance criteria:* Users can select diary entries and see them displayed with their formatting.	✓	

Feature	Acceptance Criteria Achieved?	Notes
Card Generation Diary entries need to be converted to cards *Acceptance criteria:* Flashcards are automatically generated for a new diary entry if a user choose to create flashcards. Flashcards consist of a front-side, which has a prompt, and a back-side, which has the solution that the user should remember	✓	
Automatic Card Generation Using markdown syntax, metadata and other methods, cards are generated from the diary entries. *Acceptance criteria:* Generated flashcards use information from the diary entry in their body.	✓	
Study Users can study their cards *Acceptance criteria:* Users can choose a diary or diary entry and study only the associated flashcards. Flashcards should be shown one after another. Users can select if they remembered or forgot the information on the back-side of the card.	✓	

Feature	Acceptance Criteria Achieved?	Notes
Scheduling of cards Cards are scheduled using an SRT algorithm *Acceptance criteria:* Flashcards are scheduled to be due for study at increasing intervals when users remember them, or to be shown sooner if they forget.	✓	
Non-functional requirements		
Usable interface The application should be simple to use with no prior instruction *Acceptance criteria:* Users can register, create a diary and a diary entry, and study the generated flashcards without external direction. Any required instructions should be present in the application	Partial	User testing revealed that, while most aspects of the interface were self-explanatory, some functionality – particularly regarding the Markdown editor – is not easy to use without guidance.
Responsiveness The application should be responsive and work on a range of devices. *Acceptance criteria:* Users can use the application on a desktop computer, laptop or smartphone	No	Current version of the application has no CSS breakpoints and as such is generally unusable on screens with a width smaller than 600px.

Broadly speaking, the technical aims of the project were achieved. The backend and frontend components have all the functionality implemented to satisfy the functional

requirements of the project. However, as illustrated above, the non-functional requirements were only partially met.

6.2 General evaluation

6.2.1 Key successes

Dynamic flashcards

The Cardmaker service, with its dynamic generation of context-rich flashcards, has proven successful. The core programming challenge within this project was to create an algorithmic approach to parsing freeform, loosely structured text; define a rules-based system for establishing important information (in this case through formatting hints and the use of uncommon words); and generating flashcard data. The current approach is still very naïve, but provides a fundamental basis upon which to expand in future.

Simple, extensible structure

The structure of the backend server uses RESTful principles throughout, meaning that interacting with the diary and flashcard data is easily done by other web services. Furthermore, the Cardmaker service is parallelised, and can be easily swapped out for other more sophisticated cardmaking programs, or even external services that can return digestible flashcard data (which itself is easily enforced using the Mongoose model schema). The frontend uses React, an intrinsically modular framework, that means that further work on the diary application can be done within a highly extensible structure.

6.2.2 Areas for improvement

Accessibility

HTML5 provides a number of semantic HTML elements and tags that greatly increase the ability for screen readers and other accessibility-enhancing devices to parse web pages for users with atypical accessibility needs. The current web layout relies heavily on non-semantic tags such as and that do not provide these benefits for users. Refactoring React components to use semantically rich tags would not be a significant additional task, and would provide great benefit to an often undervalued user base.

Responsiveness

The website is currently only usable on screens larger than 600px, which is a significant hurdle – a large number of people use mobile devices as their primary internet device, and

the website's current design drastically limits the ability to study flashcards or make ad-hoc diary entries whilst away from a desktop or laptop computer. React's modular design makes the implementation of responsive web layouts relatively straightforward, and this change would provide significant benefit for limited development time.

Usability

The application largely performs well in its usability, but UX testing revealed significant difficulties with the Markdown text editor. Furthermore, there are fields and prompts in some forms which are not intuitively understood by all users. Further guidance ought to be incorporated into the interface to alleviate these issues, such as a Markdown tutorial. Alternatively, the text editor could be replaced wholesale by a WYSIWYG editor, although this may require significant additional development and refactoring of the Cardmaker and `<EntryShow />` parsers

6.2.3 Future scope

There are several aspects of the application, beyond resolving issues raised in section 6.2.2, that could be improved to further enhance its overall functionality. These include:

- The use of natural-language processing to create more context-sensitive flashcards. Particularly by using models such as the highly sophisticated GPT-3, there is the potential for direct questions to be generated from diary content.
- As the backend server is a RESTful API, integrations could be made with other web services that could access diary or flashcard data. Creating a public API spec would allow for other services to build out such integrations.
- Conversely, the application – either through the backend or through data transformation in the React client – could send HTTP requests to other public APIs to enhance the content of user diaries. This could be used in a variety of ways; for example, publicly accessible weather data could be referenced for specific dates and locations, allowing flashcards to be created about what the weather was like on a particular day.
- The therapeutic benefits of diary keeping and spaced-repetition training were a key motivating factor for this project, and collaboration with mental health and

neurology experts to establish the efficacy of this project towards such goals would be of great benefit for the future direction of the project.

- The application currently uses the most common English words as a filter for ignoring unimportant words when generating cloze deletion cards. Unfortunately this only works for English-language diary entries (although the rest of the algorithm should be language agnostic, at least for languages that incorporate whitespace as a separator for words). A localisation layer would allow the application to be more readily useful for users from other language communities.

6.3 Personal reflection

Working on this project has afforded me the opportunity to explore a number of subjects of interest to me.

Spaced repetition

As a student, I have used numerous study tools and applications throughout the years. I have often found that tools are typically either designed in such an opinionated way that they are only useful for a small subsection of students or subjects; or in such a broad way as to require a significant level of organisation and management to stay useful. Spaced repetition system tools often fall into one of these camps. Anki is a powerful, highly customisable tool that does not in and of itself offer any study material or curricula, only being useful through the manual creation of flashcards or the use of third-party decks of cards from a community that has no enforced quality control. Conversely, Memrise is a complete study solution aimed at self-learners, with curricula and courses already designed; but which is difficult to customise, and results in courses being highly generalised, which may not suit more esoteric or advanced learners. Part of the motivation to do this project was to explore the possibility of an application that could essentially generate a curriculum of flashcard content, but based on the specific needs and notes of the user.

The process of working on this project has allowed me to research the science behind spaced repetition training, and the effects it has on memory and recall. This scientific foundation was instrumental in understanding the aims of the project, as well as enhancing my own understanding of the SRT practices I use in my day-to-day life. I have been lucky enough to have had previous research experience, albeit in other fields from Computer

Science, and have been able to draw on that experience in the assessment of sources, the use of library services and electronic research repositories, and summarise research literature. That being said, I have had limited experience with the specific areas of background research required for this project, and thew experience of working through the project has been a useful opportunity to broaden my own research abilities.

User experience testing

This was my first experience with UX testing an application in live sessions with participants. The process was highly useful, both for getting a non-technical perspective of my software, but also to see what assumptions a general public might make when using this tool. In order to prepare for the testing process, it was important to plan ahead for prompts and questions I would ask of participants, and to ensure that I was measuring what I intended to. It would have been preferable to have a wider sample of UX tests, but for the purposes of this project I was able to get useful input.

Project management

Management of the software project was relatively smooth; I was able to follow many best practices for the development process, managing version control and deployment versions through Git and Heroku.[57] Generally, however, time management was a challenge throughout the project, particularly regarding the non-technical aspects. In future, I would hope to take better care to finalise requirements and specifications ahead of time, to avoid similar issues. Time management is an ongoing area in need of improvement and this project, with its multiple ongoing obligations, has been a useful object lesson in its importance.

[57] The application is deployed live on Heroku. 'SRS Diary', accessed 7 October 2021, https://srs-diary.herokuapp.com/#/.

References

Alexandrov, Arseni K., and Lazar M. Fedoseev. *Long-Term Memory Mechanisms, Types and Disorders*. Neuroscience Research Progress Series. Hauppauge, N.Y.: Nova Science Publishers, 2012.

Al-Rawi, Wisam, Lauren Easterling, and Paul C. Edwards. 'Development of a Mobile Device Optimized Cross Platform-Compatible Oral Pathology and Radiology Spaced Repetition System for Dental Education'. *Journal of Dental Education* 79, no. 4 (2015): 439–47. https://doi.org/10.1002/j.0022-0337.2015.79.4.tb05902.x.

'Anki - Powerful, Intelligent Flashcards'. Accessed 4 October 2021. https://apps.ankiweb.net/.

'Axios'. Accessed 6 October 2021. https://axios-http.com/.

'Chai'. Accessed 7 October 2021. https://www.chaijs.com/.

'CSS Nesting, Specificity and You | Kilian Valkhof', 4 August 2021. https://kilianvalkhof.com/2021/css-html/css-nesting-specificity-and-you/.

'Day One: Your Journal for Life'. Accessed 27 September 2021. https://dayoneapp.com/.

'Diarium, by Timo Partl'. Accessed 27 September 2021. https://timopartl.com/#diarium.

Einstein, Gilles O., Mark A. McDaniel, Ruthann Thomas, Sara Mayfield, Hilary Shank, Nova Morrisette, and Jennifer Breneiser. 'Multiple Processes in Prospective Memory Retrieval: Factors Determining Monitoring Versus Spontaneous Retrieval.' *Journal of Experimental Psychology: General* 134, no. 3 (2005): 327–42. https://doi.org/10.1037/0096-3445.134.3.327.

'Express - Node.Js Web Application Framework'. Accessed 5 October 2021. https://expressjs.com/.

npm. 'Express-Joi-Validation'. Accessed 7 October 2021. https://www.npmjs.com/package/express-joi-validation.

Frith, Emily, Eveleen Sng, and Paul D. Loprinzi. 'Randomized Controlled Trial Evaluating the Temporal Effects of High-Intensity Exercise on Learning, Short-Term and Long-Term Memory, and Prospective Memory'. *European Journal of Neuroscience* 46, no. 10 (2017): 2557–64. https://doi.org/10.1111/ejn.13719.

Green, Collin, James C. Johnston, and Eric Ruthruff. 'Attentional Limits in Memory Retrieval—Revisited.' *Journal of Experimental Psychology: Human Perception and Performance* 37, no. 4 (2011): 1083–98. https://doi.org/10.1037/a0023095.

'Grid Diary: The Simplest Way to Get Started with Keeping a Diary | Grid Diary'. Accessed 27 September 2021. https://griddiaryapp.com/.

Hanson, Aroline E. Seibert, and Christina M. Brown. ‘Enhancing L2 Learning through a Mobile Assisted Spaced-Repetition Tool: An Effective but Bitter Pill?’ *Computer Assisted Language Learning* 33, no. 1–2 (2 January 2020): 133–55. https://doi.org/10.1080/09588221.2018.1552975.

Hawley, Karri S., Katie E. Cherry, Emily O. Boudreaux, and Erin M. Jackson. ‘A Comparison of Adjusted Spaced Retrieval versus a Uniform Expanded Retrieval Schedule for Learning a Name–Face Association in Older Adults with Probable Alzheimer’s Disease’. *Journal of Clinical and Experimental Neuropsychology* 30, no. 6 (18 July 2008): 639–49. https://doi.org/10.1080/13803390701595495.

Ho, Joanna, Adrienne Epps, Louise Parry, Miriam Poole, and Suncica Lah. ‘Rehabilitation of Everyday Memory Deficits in Paediatric Brain Injury: Self-Instruction and Diary Training’. *Neuropsychological Rehabilitation* 21, no. 2 (1 April 2011): 183–207. https://doi.org/10.1080/09602011.2010.547345.

npm. ‘Joi’. Accessed 7 October 2021. https://www.npmjs.com/package/joi.

Karch, Dominik, Krzysztofa Kopyt, Aleksandra Gauden, and Michal Nowakowski. ‘Efficiency of Web Application and Spaced Repetition Algorithms as an Aid in Preparing to Practical Examination of Histology: PS195’. *Porto Biomedical Journal* 2, no. 5 (September 2017): 187–88. https://doi.org/10.1016/j.pbj.2017.07.030.

Kaufman, Josh. *Google-10000-English/Google-10000-English.Txt*, 2021. https://github.com/first20hours/google-10000-english/blob/d0736d492489198e4f9d650c7ab4143bc14c1e9e/google-10000-english.txt.

Kleijn, Suzanne, Henk Pander Maat, and Ted Sanders. ‘Cloze Testing for Comprehension Assessment: The HyTeC-Cloze’. *Language Testing* 36, no. 4 (October 2019): 553–72. https://doi.org/10.1177/0265532219840382.

Lambers, Anton, and Adrian J. Talia. ‘Spaced Repetition Learning as a Tool for Orthopedic Surgical Education: A Prospective Cohort Study on a Training Examination’. *Journal of Surgical Education* 78, no. 1 (2021): 134–39. https://doi.org/10.1016/j.jsurg.2020.07.002.

Linderholm, Tracy, and Lise Abrams. ‘The Benefits of Expressive Writing on Long-Term Memory Performance’. In *Long-Term Memory: Mechanisms, Types and Disorders*, 131–46. Nova Science Publishers, 2012.

‘Location - Web APIs | MDN’. Accessed 6 October 2021. https://developer.mozilla.org/en-US/docs/Web/API/Location.

Memrise. ‘Learn a Language. Meet the World. | Memrise’. Accessed 4 October 2021. https://www.memrise.com.

Miner, Annalise E., Mark W. Schurgin, and Timothy F. Brady. ‘Is Working Memory Inherently More “Precise” Than Long-Term Memory? Extremely High Fidelity Visual Long-Term Memories for Frequently Encountered Objects’. *Journal of Experimental Psychology.*

Human Perception and Performance 46, no. 8 (2020): 813–30. https://doi.org/10.1037/xhp0000748.

'Mocha - the Fun, Simple, Flexible JavaScript Test Framework'. Accessed 7 October 2021. https://mochajs.org/.

'Mochi — Spaced Repetition Made Easy'. Accessed 4 October 2021. https://mochi.cards/.

'Mongoose ODM v6.0.9'. Accessed 5 October 2021. https://mongoosejs.com/.

Node.js. 'Node.Js'. Node.js. Accessed 5 October 2021. https://nodejs.org/en/.

Norris, Dennis. 'Short-Term Memory and Long-Term Memory Are Still Different'. *Psychological Bulletin* 143, no. 9 (2017): 992–1009. https://doi.org/10.1037/bul0000108.

'Npm'. Accessed 7 October 2021. https://www.npmjs.com/.

Oren, Shiri, Charlene Willerton, and Jeff Small. 'Effects of Spaced Retrieval Training on Semantic Memory in Alzheimer's Disease: A Systematic Review'. *Journal of Speech, Language & Hearing Research* 57, no. 1 (February 2014): 247–70. https://doi.org/10.1044/1092-4388(2013/12-0352).

Parker, Amanda, Edward L. Wilding, and Timothy J. Bussey, eds. *The Cognitive Neuroscience of Memory: Encoding and Retrieval*. 1. publ. Studies in Cognition Series. Hove: Psychology Press, 2002.

Google Docs. 'Participation Information and Consent Form'. Accessed 7 October 2021. https://docs.google.com/forms/d/e/1FAIpQLSfwQxM2Fpif9esq0TNJSXXpiMJzxzR5mZUN_sYqZQTDhRqL9Q/viewform?usp=embed_facebook.

npm. 'Passport'. Accessed 6 October 2021. https://www.npmjs.com/package/passport.

Pena, André. *React-Mde*. TypeScript, 2021. https://github.com/andrerpena/react-mde.

Persellin, Diane. *A Concise Guide to Improving Student Learning: Six Evidence-Based Principles and How to Apply Them*. First edition. Sterling, Virginia: Stylus, 2014.

Pham, Xuan-Lam, Gwo-Dong Chen, Thi-Huyen Nguyen, and Wu-Yuin Hwang. 'Card-Based Design Combined with Spaced Repetition: A New Interface for Displaying Learning Elements and Improving Active Recall'. *Computers & Education* 98 (July 2016): 142–56. https://doi.org/10.1016/j.compedu.2016.03.014.

Provos, Niels, and David Mazières. 'A Future-Adaptable Password Scheme', n.d., 13.

'React – A JavaScript Library for Building User Interfaces'. Accessed 6 October 2021. https://reactjs.org/.

'React Redux | React Redux'. Accessed 6 October 2021. https://react-redux.js.org/.

ReactRouterWebsite. ‘React Router: Declarative Routing for React’. Accessed 6 October 2021. https://reacttraining.com/react-router.

React-Markdown. JavaScript. 2015. Reprint, remark, 2021. https://github.com/remarkjs/react-markdown.

Reddy, Siddharth, Igor Labutov, Siddhartha Banerjee, and Thorsten Joachims. ‘Unbounded Human Learning: Optimal Scheduling for Spaced Repetition’, 13-17-:1815–24. KDD ’16. ACM, 2016. https://doi.org/10.1145/2939672.2939850.

‘RemNote | The Best Way to Remember and Organize What You Learn’. Accessed 4 October 2021. https://www.remnote.io/.

Rose, Nathan S., Joel Myerson, Henry L. Roediger, and Sandra Hale. ‘Similarities and Differences Between Working Memory and Long-Term Memory: Evidence From the Levels-of-Processing Span Task’. *Journal of Experimental Psychology. Learning, Memory, and Cognition* 36, no. 2 (2010): 471–83. https://doi.org/10.1037/a0018405.

‘Sass: Syntactically Awesome Style Sheets’. Accessed 7 October 2021. https://sass-lang.com/.

‘Shared Decks - AnkiWeb’. Accessed 4 October 2021. https://ankiweb.net/shared/decks/.

Sherwood, Lauralee. *Human Physiology: From Cells to Systems*. Cengage Learning, 2015.

Simons, J. S. ‘Recollection-Based Memory in Frontotemporal Dementia: Implications for Theories of Long-Term Memory’. *Brain* 125, no. 11 (1 November 2002): 2523–36. https://doi.org/10.1093/brain/awf247.

‘SRS Diary’. Accessed 7 October 2021. https://srs-diary.herokuapp.com/#/.

npm. ‘Supertest’. Accessed 7 October 2021. https://www.npmjs.com/package/supertest.

Szőllősi, Ágnes, Attila Keresztes, Martin A. Conway, and Mihály Racsmány. ‘A Diary after Dinner: How the Time of Event Recording Influences Later Accessibility of Diary Events’. *Quarterly Journal of Experimental Psychology* 68, no. 11 (November 2015): 2119–24. https://doi.org/10.1080/17470218.2015.1058403.

Tarnow, Eugen. ‘Why The Atkinson-Shiffrin Model Was Wrong From The Beginning’. *WebmedCentral Neurology* 1, no. 10 (2010): 13.

MongoDB. ‘The Most Popular Database for Modern Apps’. Accessed 5 October 2021. https://www.mongodb.com.

Yarn. ‘Yarn’. Accessed 7 October 2021. https://classic.yarnpkg.com/en/.

Appendices

Wireframes

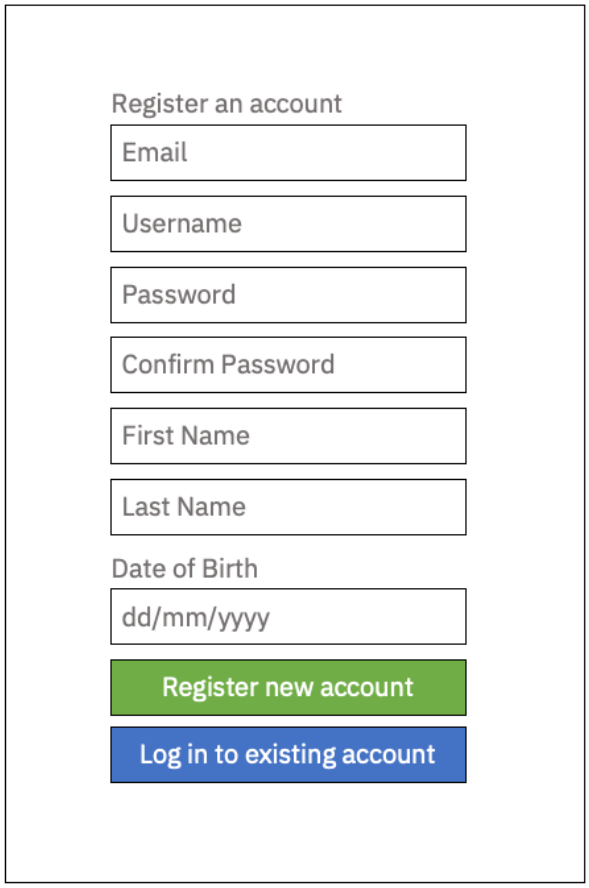

Figure 41 – Wireframe, registration form

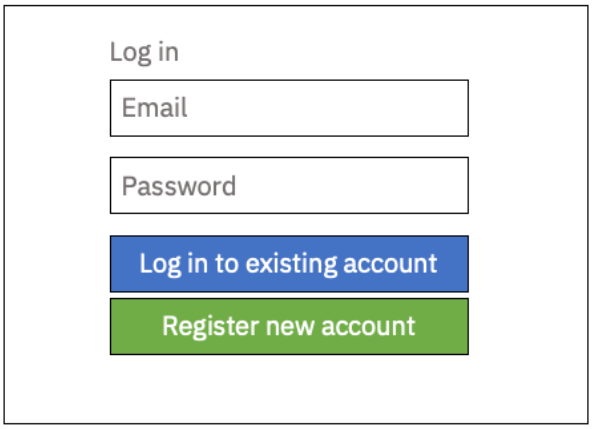

Figure 42 – Wireframe, log in form

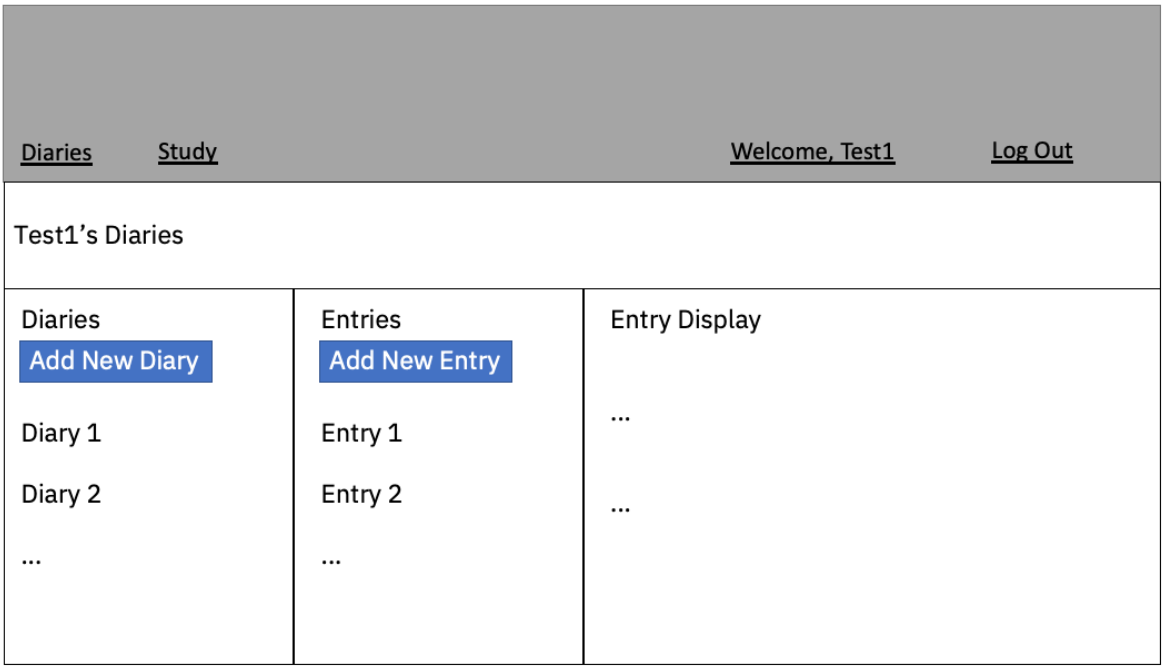

Figure 43 – Wireframe, diary index

Test case

Table 1 - Test case 1: User registration

Test Case Id: 1	**Test Purpose: Users can register on the website**		
Server Environment: Written in Express.js 4.17.1, MongoDb 3.6.10, Mongoose 5.13.4, running on Node 16.6.1 **Client Environment: Written in React.js 17.0.2, React-Redux 7.2.4, running on Chrome 94.0.4606.61**			
Preconditions: User is on the home page of the website, and not logged in.			
Test Case Steps:			
Step No	Procedure	Expected Response	Pass/Fail
1	Choose 'Register'	Displays the registration form modal (see Figure 41 – Wireframe, registration form)	P
2	Fill registration form: Email: testcase1@example.com Username: testcase1 Password: password Confirm password: password		P

	First Name: Test1 Last Name: Case1 Date of Birth: 01/01/2000		
3	Click ‘Register new account’	The application redirects to the diary index (see Figure 43 – Wireframe, diary index	P
Comments: [provide details of any failed steps]			
Related Tests:			
Author: N Sibanda, C21009631		**Checker:**	

Documents

Participant recruitment letter

A Web Application for Recording a Personal Diary with Spaced Repetition Training

Recruitment Invitation Letter

Dear _____

I would like to invite you to be a user-testing participant for a web application I am developing as part of my MSc dissertation research project.

About the Project:

The project is a web application for recording a personal or study diary, implementing spaced-repetition training to allow users to better remember important information and the events of their lives. There are numerous diary apps and websites currently available, with both online and offline functionality. These programs typically serve a number of productivity functions, including personal organisation and time management. These apps are ostensibly designed - at least in part - to help users remember information. Learners need to access such information during exams, assignments or complex tasks. Diary-keepers use this information to remind themselves of the events of their lives, and to keep track of upcoming events.

Current apps allow users to organise their information, but make no effort to help users recall information they deem important. This project attempts to fulfil this need through the design and implementation of a web application that allows users to keep an ongoing diary, processing this information into relatively atomised elements, and generating digital "flashcards" that users can use to revise the information they enter. These flashcards will be shown at intervals defined by a spaced-repetition algorithm, to maximise retention of information while minimising the number of flashcards a user is required to review each day to retain all the information therein.

What Participation Involves:

The participatory element of this project will involve user testing; participants will be asked to use the developed application with minimal guidance, to complete several tasks, such as registering a new account, logging in, creating diary entries, and studying generated flashcards. During this live testing session, the researcher will ask participants about their experience using the software in real-time, to identify strengths and weaknesses in its design and implementation. Participants will then be asked to

complete a questionnaire to reflect on their overall experience using the application. This data will be used to evaluate the success of the application in achieving the goals of the project, and to identify areas for potential future improvement or expansion.

Your participation in this research project would be immensely appreciated, and I would love to speak with you further about it. You will be provided with all the information you need in order to decide whether you would like to proceed with participation; informed consent is a cornerstone of ethical research, and all participants must feel comfortable and confident in their willingness to be involved.

Please let me know if participation in this project would be of interest to you, and I will be able to send along further information.

Yours sincerely,

Nyasha Sibanda

Principal Investigator

Student – MSc Computing

Participant post-test questionnaire

1. Do you typically use productivity software? This includes applications such as diary-taking software, digital calendars, note-taking software, study tools or time management software. (Yes/No)
2. If yes, describe your typical use of these programs, including why and how often you use them.
3. Have you ever used Spaced Repetition Training software? This includes applications such as Anki and Memrise. (Yes/No)
4. If yes, describe your typical use of these programs.
5. Have you ever used Markdown to format text? This includes applications such as Notion, or the text formatting syntax used by Reddit.
6. If yes, describe your typical use of Markdown.
7. What were your initial impressions of the web application you were asked to use?
8. What did you think of the layout of the content?
9. Was it easy to accomplish the tasks you were asked to do?
10. Is the purpose of the application clear?

11. What did you like the most about using the application?
12. What did you like the least?
13. Were there any aspects of the application that were frustrating?
14. How would you compare the application to other productivity tools you may have used in the past?
15. How would you change the application?
16. Would you use an application like this in your own life?