THERAPEASE – AN ALL-IN-ONE ADMINISTRATIVE PLATFORM FOR THERAPISTS

by

Aoife Lynch

N00222385

Supervisor: John Dempsey

Second Reader: Stefan Paz Berrios

Year 4 2025/2026

DL836 – BSc(Hons) Creative Computing

Abstract

Therapists in private practice frequently manage administrative tasks across multiple disconnected tools, resulting in fragmented workflows, increased cognitive load and reduced efficiency. This project presents TherapEase, a web-based administrative platform designed to unify scheduling, client record management, payment processing and automated communications into a single, integrated system. The motivation for this project came from conversations with a practising therapist, along with a survey of 19 participants that identified fragmentation, time consumption and lack of automation as the primary pain points in existing workflows. TherapEase was developed using the MERN stack, with security implemented through JSON Web Tokens and two-factor authentication to ensure GDPR compliance. External integrations include Stripe for payment processing, Zoom for online appointment links, Twilio for SMS reminders and Redis with BullMQ for asynchronous background processing. Development followed an Agile methodology across six sprints, while using user-centred design principles. Technical testing was conducted using Playwright and Jest, with all 24 unit tests and end-to-end workflows passing successfully across multiple browsers. Usability testing with three participants showed a 70% reduction in task completion time compared to existing workflows, with an average System Usability Scale score of 97.5. The results confirm that TherapEase successfully addresses the administrative challenges faced by therapists through an integrated, user-friendly and GDPR-compliant platform.

Keywords: *frontend development, backend development, full-stack application, practice management, usability, user-centred design, GDPR, MERN stack, asynchronous processing, therapist administration, web application.*

Acknowledgements

I want to give my greatest thanks to my supervisor, John Dempsey, whose guidance and feedback throughout the development of this project was invaluable. I would also like to thank my second reader, Stefan Paz Berrios, for his input and expert insight on user interface design.

To my mother, Anita, thank you for your professional insight as a psychotherapist that led to the initial idea of TherapEase. Beyond that, I am incredibly grateful for your constant encouragement and the many evenings spent discussing and expanding the project.

I owe a special thanks to Chloe, Julia and all my classmates, for your friendship over the course of this degree. It has made every challenge more manageable and I am so grateful to have shared this journey with you all.

Finally, I'd like to express my deepest appreciation to my wonderful family, friends and boyfriend, both in Ireland and in Australia. Thank you all for your support and patience throughout this project, as well as the four years leading up to it. I'm so grateful for every one of you.

"Giorraíonn beirt bóthar"

Declaration of Ownership

The incorporation of material without formal and proper acknowledgement (even with no deliberate intent to cheat) can constitute plagiarism.

If you have received significant help with a solution from one or more colleagues, you should document this in your submitted work and if you have any doubt as to what level of discussion/collaboration is acceptable, you should consult your lecturer or the Course Director.

WARNING: Take care when discarding program listings lest they be copied by someone else, which may well bring you under suspicion. Do not to leave copies of your own files on a hard disk where they can be accessed by other. Be aware that removable media, used to transfer work, may also be removed and/or copied by others if left unattended.

Plagiarism is considered to be an act of fraudulence and an offence against Institute discipline. Alleged plagiarism will be investigated and dealt with appropriately by the Institute. Please refer to the Institute Handbook for further details of penalties.

The following is an extract from the B.Sc. in Creative Computing (Hons) course handbook. Please read carefully and sign the declaration below

Collusion may be defined as more than one person working on an individual assessment. This would include jointly developed solutions as well as one individual giving a solution to another who then makes some changes and hands it up as their own work.

***DECLARATION**:*

I am aware of the Institute's policy on plagiarism and certify that this thesis is my own work.

***Student**: Aoife Lynch*

Signed 

Failure to complete and submit this form may lead to an investigation into your work.

Abbreviations & Acronyms

MERN	MongoDB, Express, React.js & Node.js
JWT	JSON Web Tokens
2FA	Two-factor Authentication
UCD	User-centred Design
GDPR	General Data Protection Regulation
API	Application Programming Interface
HTTP	Hypertext Transfer Protocol
REST/RESTful	Representational State Transfer
CORS	Cross-Origin Resource Sharing
MVP	Minimum Viable Product
CRUD	Create, Read, Update, Delete
ERD	Entity Relationship Diagram
UX	User Experience
UI	User Interface
SUS	System Usability Scale
TOTP	Time-based One Time Password
SMS	Short Message Service
EHR	Electronic Health Record
CLI	Command Line Interface
CSS	Cascading Style Sheets
HTML	Hypertext Markup Language
XSS	Cross-Site Scripting

IDE	Integrated Development Environment
SDK	Software Development Kit
SOAP	Subjective, Objective, Assessment, Plan (note template)
DAP	Data, Assessment Plan (note template)
BIRP	Behaviour, Intervention, Response, Plan (note template)
PIP	Problem, Intervention, Plan (note template)
QR	Quick Response (code)
URL	Uniform Resource Locator
HIPAA	Health Insurance Portability and Accountability Act

Table of Contents

1. Introduction

Many therapists face ongoing challenges in managing the administrative side of their practice, often relying on a range of separate, disconnected applications rather than a single integrated application. Tasks such as scheduling, client record management and payment processing are typically handled across separate tools, which requires manual coordination and repeated data entry.

Motivation for this project comes from a real-world problem provided by a practising therapist, who explained the difficulties of managing administrative tasks across several platforms. Research into this shows that the use of fragmented workflows has been shown to increase cognitive load and contribute to inefficiencies in task completion (Melnick et al., 2020; Olakotan et al., 2025).

This project aims to design and develop TherapEase, a web application that aims to streamline the administrative workflow for therapists. Common administrative tasks such as appointment booking, appointment and payment reminders, client management and payment processing will all be accessible through this singular application, reducing the administrative burden and improving workflow efficiency. A key objective of the application is to enhance usability and overall user experience, which will be achieved by applying user-centred design principles.

TherapEase is developed using a MERN stack, with Node.js (OpenJS Foundation, 2025) and Express (OpenJS Foundation, 2017) for the backend, MongoDB (MongoDB, Inc., 2024) with Mongoose (Mongoose, 2011) for data management and React (Meta Open Source, 2013) for the frontend. Security is handled with JSON Web Tokens (Auth0, 2025) and two-factor authentication. External integrations including Stripe (Stripe, Inc., n.d.), Twilio (Twilio, 2024), Nodemailer (Nodemailer, 2010) and Zoom (Zoom Video Communications, n.d.) are used for payments and communication through SMS and email, along with Redis (Redis, Inc., n.d.) and BullMQ (BullForce Labs AB, 2018) for the asynchronous processing of background tasks.

Functional testing is conducted to ensure the reliability and functionality of the application. The Jest testing framework (Jest, 2017) is used to test individual components, along with the Playwright testing framework (Microsoft, 2025a) to perform end-to-end testing. User testing is done to evaluate the system's usability and effectiveness, including time-on-task testing to compare old workflows and TherapEase workflows, errors made and the System Usability Scale (SUS). The results of these tests are used to identify usability issues, improve user interface design and ensure the application meets the objectives.

The development process for TherapEase follows principles of iterative and user-centred design that involves multiple cycles of design, development and feedback with real users to guide and refine the project. Project management tools are used throughout the project phases, including Trello, Miro, Notion and GitHub.

Chapter 2 presents a review of existing research related to workflow efficiency, usability and user-centred design, with a focus on administrative tools. Chapter 3 discusses the requirements analysis and system planning for the TherapEase platform. Chapter 4 outlines the proposed design of the system architecture and user interface. Chapter 5 describes the implementation of the application. Chapter 6 evaluates the system through technical and usability testing. Chapter 7 discusses the project management tools and methodologies, followed by Chapter 8, to conclude and discusses potential areas for future development.

2. Research & Background

This chapter shows the research undertaken prior to the development of TherapEase. The chapter is structured in two sections. The first is a literature review examining existing evidence on usability challenges, the benefits of good design and the limitations of current tools available to therapy practices. The second is a technical research section that explores the feasibility of building TherapEase, focusing on data protection compliance and the implementation of key technical features. These two research sections guide the requirements, design and development decisions described in the chapters that follow.

2.1 Usability in Administrative Systems

A key issue identified across the administration platforms used by therapists is usability. Many rely on a fragmented mix of applications, such as Outlook for email and calendars, SharePoint for document storage and SumUp for payment processing. While each tool performs its specific function, none are designed with therapists' workflows in mind and their lack of integration can cause significant problems, including repetitive data entry and limited automation.

Research into Electronic Health Record (EHR) systems provides relevant insight into how poorly designed administrative software affects users. Although EHR systems differ in scope from tools like Outlook or SharePoint, the usability challenges are comparable as both involve managing schedules, client profiles and sensitive information. Olakotan et al. (2025) identify prolonged navigation paths, inefficient task flows and poorly structured interfaces as key contributors to documentation burden. Their scoping review identifies four challenges that link together: fragmented interfaces force task-switching, which increases cognitive load, slows task completion and raises the likelihood of error. These challenges are defined as fragmented data, making retrieval slow and error-prone, redundant data entry due to a lack of integration, frequent task-switching between disconnected applications and

misalignment between interface design and real-world workflows. Melnick et al. (2020) report a strong association between low system usability and increased cognitive task load, which then contributes to professional burnout, which is significant for therapists managing administrative work alongside emotionally demanding client work.

In contrast, well-designed systems have been shown to measurably reduce these burdens. Mazur et al. (2019) found that physicians using enhanced EHR systems experienced significantly lower cognitive load, resulting in improved performance levels. Nielsen (1994) establishes that systems designed around error prevention and visibility of system status, such as keeping users informed through clear feedback and supporting recovery from mistakes, reduce frustration and minimise the need for workarounds. Together, these findings establish that usability is a functional requirement for TherapEase, not a cosmetic consideration. The design goal is an integrated interface that reduces task-switching, eliminates redundant data entry and aligns with how therapists actually work.

2.2 User-Centred Design and Iterative Testing

The second part of the research question asks how user-centred design principles can be applied to address the usability challenges identified above. Gulliksen et al. (2003) define User-Centred Design (UCD) as a process focusing on usability throughout the entire development process. They argue that without enough user involvement, systems are more likely to fail to reflect real operational practices.

Saparamadu et al. (2021) demonstrate how UCD can be applied in practice through a process involving focus groups, wireframes, user testing and iterative refinement. Their approach ensures that user needs are defined and understood before any key design decisions are made. Analysing users' daily tasks and pain points before committing to a system architecture leads to improved usability outcomes.

Gould and Lewis (1985) identify three principles for designing usable systems: an early focus on users and their tasks, empirical measurement of how users interact with the system and iterative

refinement based on evaluation findings. TherapEase is developed through iterative prototyping, with usability testing informing revisions to the interface before a final version is built. The testing chapter describes this process in detail and assesses whether or not the application successfully addresses the challenges identified.

Applying the usability principles established by Nielsen (1994), TherapEase incorporates inline form validation that flags errors before submission, confirmation messages after key actions such as booking an appointment and consistent navigation that does not require users to remember where features are located.

2.3 Analysis of Existing Applications

Before beginning development of TherapEase, it is necessary to examine existing applications on the market. Existing applications for therapists vary widely in functionality, cost and usability. A competitor analysis was performed to examine what these applications provide, where they fall short and help establish what gap TherapEase is designed to fill. While some European practice management tools exist, such as Writeupp and Cliniko, these platforms are primarily designed for physiotherapy and general allied health rather than psychotherapy specifically and do not reflect the administrative workflows of Irish therapists. TheraNest and SimplePractice were selected as the most widely adopted platforms by therapists with the most similar features to TherapEase.

TheraNest is a practice management platform aimed at therapists and psychologists in private or small practices. It integrates appointment scheduling, client records, billing, invoicing and optional telehealth (Ensora Health, 2025). From a design perspective, TheraNest has a functional, neutral aesthetic, with a persistent dark navy sidebar and contrasting white icons and text. The core sections include Schedule, Tasks, Clients, Staff, Billing, Dynamic Forms, Reports and Organisation. Several of these items contain expandable dropdowns, indicating a deeper navigation structure that may increase

the learning curve for new users. As seen in Figure 1, the schedule page has secondary horizontal tabs provide more navigation without crowding the main interface, such as Availability and Bulk Edit Appointments. Action buttons, such as “New Appointment”, “Add Progress Note” and “Add Invoice”, are clearly labelled and are in line with the schedule, reducing the need to navigate elsewhere to complete post-session tasks. The constant presence of a floating "Get Started" button, a live chat bar, and a "Contact Us" prompt in the bottom-right corner of the page may feel intrusive in a clinical workflow context. Despite these usability considerations, TheraNest's primary limitations are its cost structure and US-centric design. Its base subscription of $29 per month limits users to 30 active clients, with costs rising significantly as the practice grows. SMS reminders, telehealth, and insurance billing features all incur additional fees, making the cost considerably higher than it may appear. The platform is priced in US dollars and shows no evidence of GDPR compliance, creating a significant compliance risk for Irish practitioners.

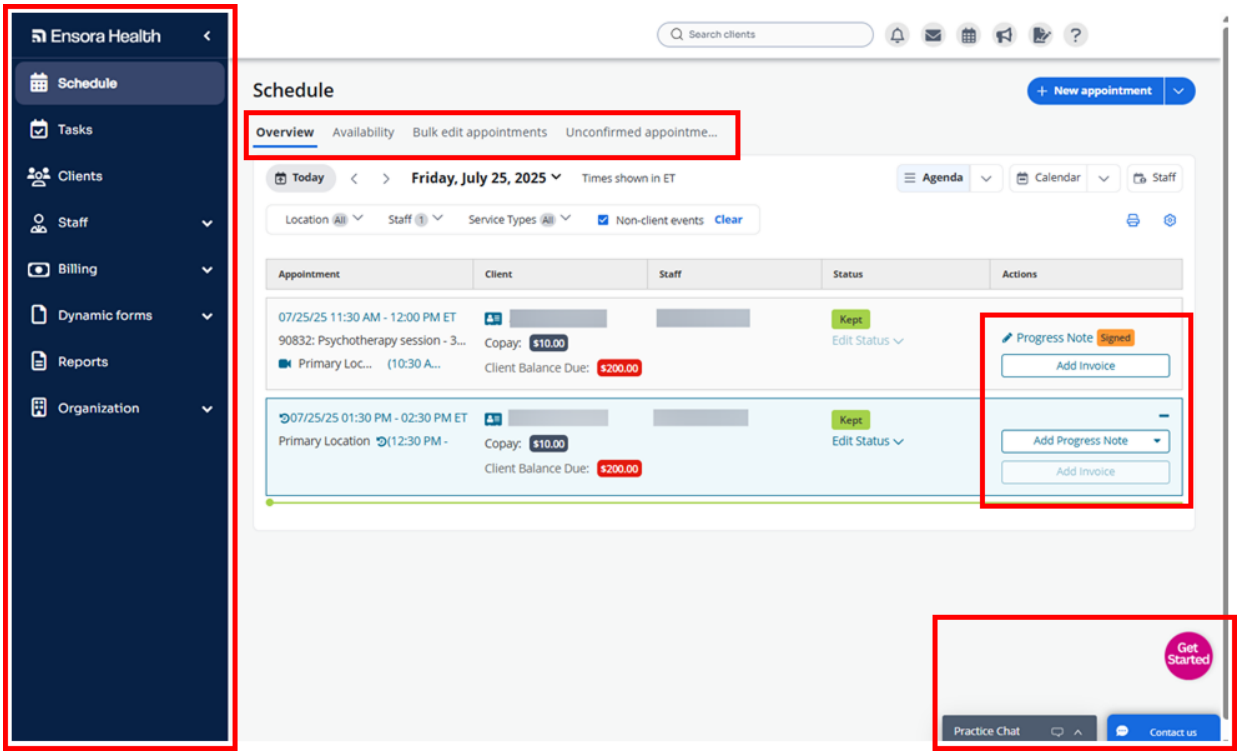

Figure 1 - TheraNest Schedule Page

SimplePractice is one of the most widely used platforms in the market, serving over 250,000 users (SimplePractice, n.d.). It has scheduling, telehealth, automated invoicing, customisable note templates, a client portal, and online booking all in a single system. Visually, it presents a polished, minimalistic interface with clean typography (Figure 2). A persistent left-hand sidebar in a light, minimal style provides immediate access to key areas including Calendar, Clients, Billing and Settings. The client profile interface organises information across clearly labelled tabs, with a persistent right-hand panel surfacing financial summaries and upcoming appointments without requiring navigation away from the current screen. Client notes are embedded directly within the client overview, reducing friction around documentation. The calendar view uses colour-coding by appointment status, giving practitioners a broad overview of their schedule (Figure 3). The monthly view being so information-heavy means individual appointments display only abbreviated details, requiring additional clicks to access full context. However, the extent of features available, including website creation, marketing tools, and prescription management, introduces interface complexity that is largely irrelevant to Irish practice and risks overwhelming less-experienced users. Pricing starts at $49 per month, rising to $99 for the Plus plan and has increased several times since 2022. Like TheraNest, the platform is built around HIPAA rather than GDPR, making it a compliance risk for Irish therapists.

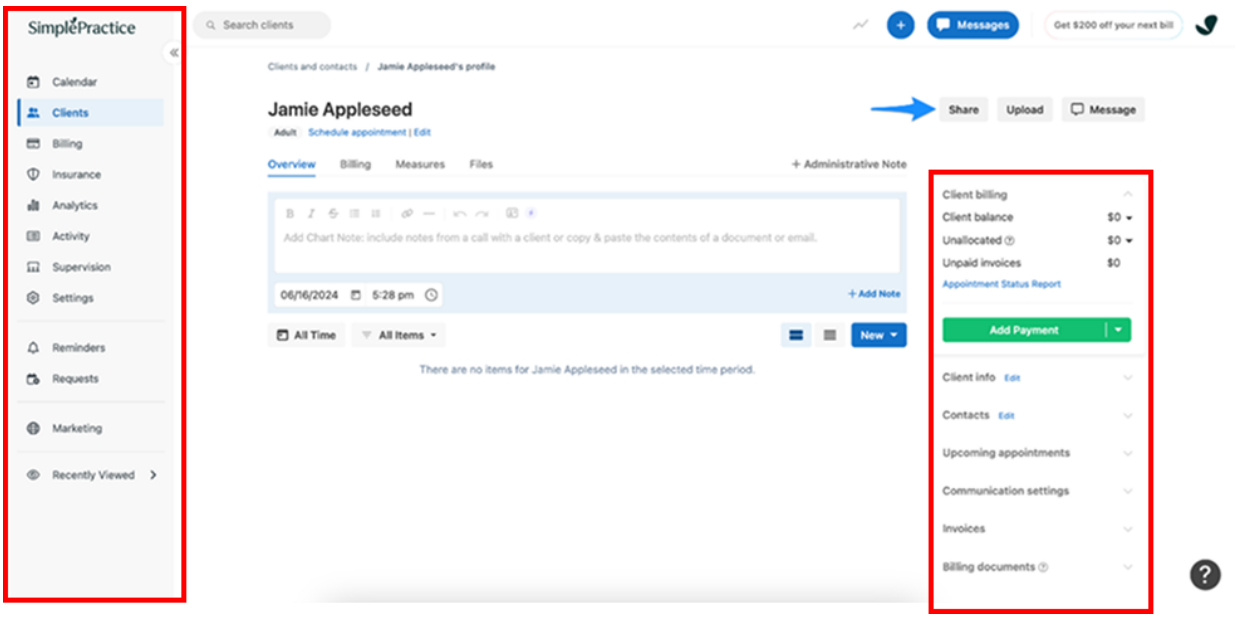

Figure 2 - SimplePractice Client Page

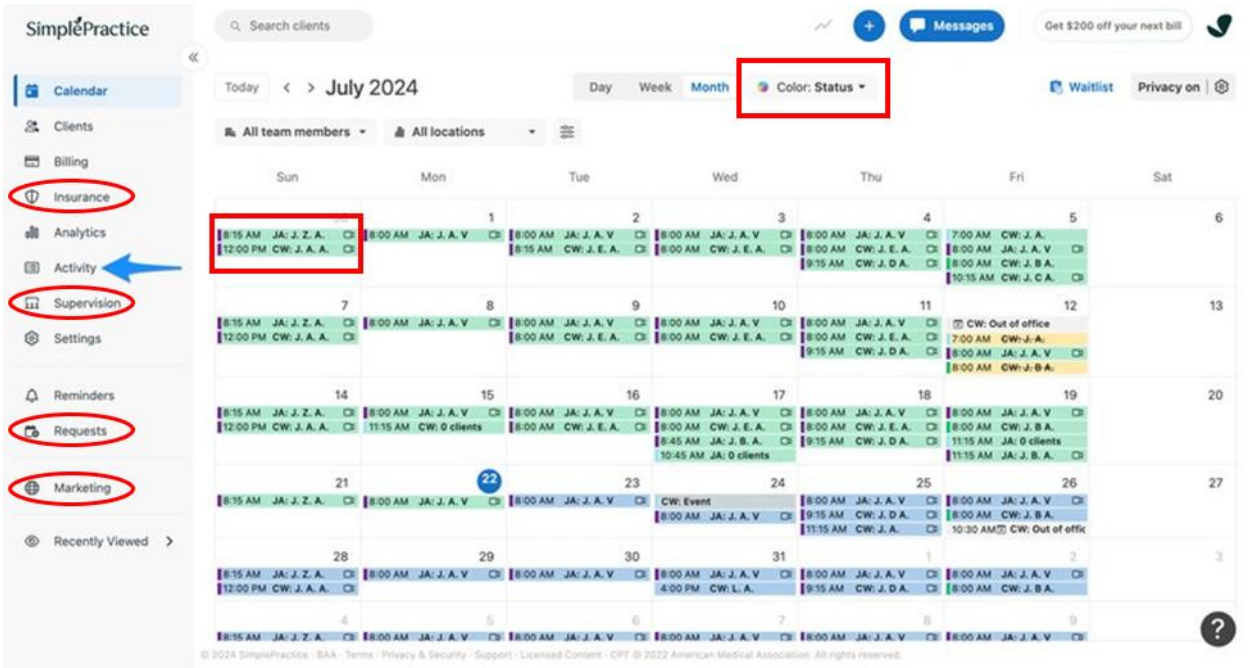

Figure 3 - SimplePractice Calendar Page

Both platforms demonstrate that integrated practice management software is technically achievable and valuable when implemented well. TheraNest is built around operational efficiency, while

SimplePractice prioritises visual simplicity and ease of navigation. However, neither has been designed with the Irish therapist as its primary user, with both built around a different regulatory environment, billing model and scale of practice.

2.4 GDPR

The General Data Protection Regulation (GDPR) is a European Union regulation introduced in 2018 to protect individuals' personal data and privacy (European Data Protection Supervisor, 2018). Therapists are legally and ethically responsible for protecting client confidentiality. Client data held by therapists, including session notes, intake assessments and clinical history, is classified as health data under GDPR. Article 9(1) of the GDPR places health data in the highest category of protected personal data, meaning its processing is strictly regulated (European Parliament & Council of the EU, 2018). Therapists must therefore ensure that client data is accurate, only collected when necessary, kept securely and deleted when no longer needed.

To meet these GDPR requirements, two-factor authentication (2FA) can be implemented as a secondary layer of security. To achieve this, the primary authentication layer uses JSON Web Tokens (JWT), where a token is generated at login and sent with each request the user makes. The 2FA layer uses Speakeasy (Bao, 2016) to generate and verify Time-Based One-Time-Passwords (TOTP) and a QR code generator allows users to add their account to their chosen authenticator app. Article 9(3) of the GDPR states that health data may only be processed by a professional bound by an obligation of secrecy (European Parliament & Council of the EU, 2018). Two-factor authentication ensures that client data cannot be accessed by a compromised password alone.

2.5 Asynchronous Email Confirmation

To reduce the workload of the users, automation of routine administrative tasks such as appointment email confirmations and reminder messages is essential. However, synchronous operations mean that the system must wait for a response before continuing. This means that attempting to execute tasks such as sending emails and SMS messages synchronously within an application can negatively impact the system's performance through increased latency and lower responsiveness.

Research into asynchronous system design emphasises the importance of separating time-consuming operations from the main application workflow. By splitting these processes, systems can handle user interactions more efficiently while delegating background tasks to be processed independently. Ramesh Maharjan et al. (2023) highlight the effectiveness of message queue architectures in supporting asynchronous communication between system components. Their findings demonstrate that message queues allow tasks to be executed without requiring an immediate response, therefore reducing latency and improving overall system performance. Additionally, their benchmarking results show that Redis outperforms alternative message queue technologies in terms of latency, making it suitable for time-sensitive applications.

For therapy practice management systems, asynchronous processing is suitable for handling automated communications such as booking confirmations and appointment reminders. These tasks are not critical to the immediate user interaction and can therefore be processed in the background. By implementing asynchronous processes, the system can minimise administrative burdens while still maintaining a responsive and efficient user experience.

The research covered in this chapter highlights the issues in administrative systems used by therapists, with studies showing a clear relationship between poor system usability and increased cognitive load. The competitor analysis further reinforces the need for a unified application due to the limitations in

cost, complexity and workflow alignment in the currently available applications. Using these findings, the project is guided by the following research question: “What are the key usability challenges in administration tools commonly used by therapists and how can user-centred design principles be applied in an integrated web application to streamline administrative workflows?” This question influences the requirements, design decisions, implementation and testing approach taken in the development of TherapEase.

3. Requirements Analysis

A clear understanding of system requirements is important to developing an effective and user-centred application. This chapter outlines the requirements gathered through a user survey, persona creation and use case diagrams. It defines the user, functional and non-functional requirements that will guide development. This will ensure the final product is well-built and meets the needs of its target users.

3.1 User Requirements

To find and support the system requirements for TherapEase, a survey was conducted with therapists (Appendix A). The survey aimed to gain insight into current workflows, tools used, pain points and desired improvements in administrative processes. The questions in the survey were based on early discussions with therapists about their current workflows, pain points and what they would like to see in an administration platform. The survey was distributed online to 19 individuals, which included both qualified practising therapists and student therapists in different age groups and levels of experience. The participants' responses were anonymous to ensure honest feedback. The questions were primarily closed-ended for quantitative data, along with one open-ended question at the end to gather any qualitative feedback or additional feature suggestions.

The questions included:

- Types of administrative tasks performed
- Tools currently used to perform those tasks
- Workflow efficiency and time consumption
- Frequency of switching between systems
- Barriers to adopting new tools
- Impact of current systems on client experience

- Desired features in an improved system.

The findings were used to decide on the functional and non-functional requirements for TherapEase.

3.1.1 Survey Findings

The findings of the survey examine the participants' current administrative workflow, tool usage, pain points and feature preferences. The responses show a consistent picture of fragmented tools, administrative burden and a want for an integrated solution.

The participants were asked which administrative tasks they perform on a weekly basis(Figure 4).

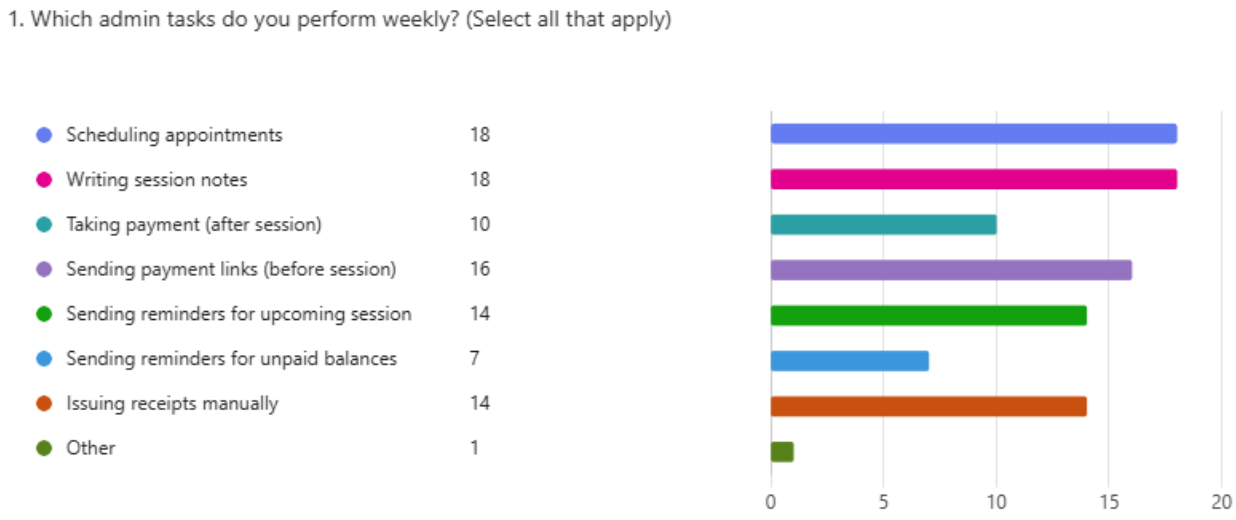

Figure 4 - Survey Question 1 (Weekly Administrative Tasks)

Scheduling appointments and writing session notes were performed by 18 of the 19 participants, making them the most common tasks. Following that were sending payment links before sessions (84%), issuing receipts manually (74%) and sending reminders for upcoming sessions (74%).

This indicates that most therapists are managing a substantial administrative workload across multiple systems for scheduling, documentation and payments weekly.

Following that was a question of what tools the participants currently use for administrative work. The responses show a heavy reliance on general-purpose tools with only 3 of the 19 participants using a specialist platform for practice management. Outlook and Gmail were identified as the most dominant tools for managing a calendar and communication, along with SumUp for payments and issuing receipts. In a follow-up question on most valued improvements for an administrative platform, automation was the highest priority in terms of receipts, client reminders and payment links as selected by 79% of participants. Following that was easier note creation and storage (68%), simpler interface (63%) and fewer tools to switch between (53%). This reinforces the finding that fragmentation is a pain point for all participants and that automation would decrease the task load of the administrative work.

To understand the current workflows of therapists, participants were asked how often they switch between multiple tools to complete a single administrative task. 95% of participants reported that they switch between tools at least occasionally to complete a single task and 52% of those do so constantly or frequently. This points to a fragmented workflow where no single application can cover the full range of weekly tasks.

Participants were asked to rate their administrative process on a scale of 1 to 5 in terms of time consumption. 1 being not time-consuming at all and 5 being extremely time-consuming (Figure 5).

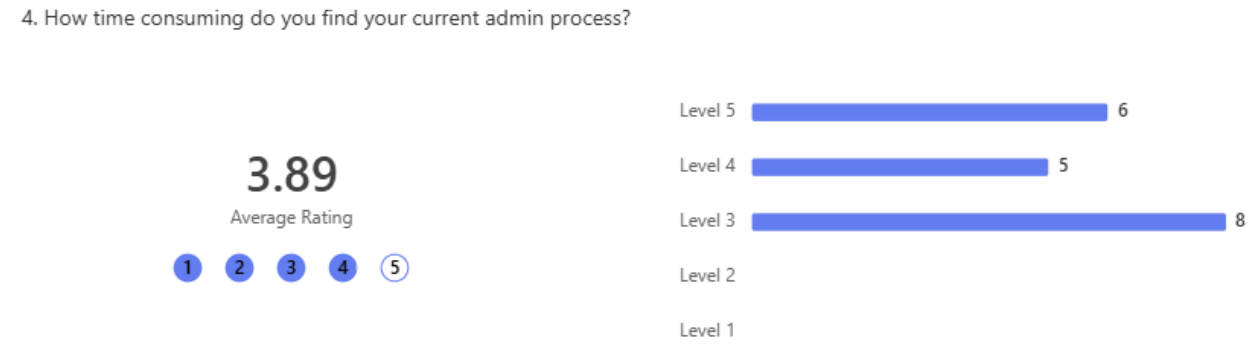

Figure 5 - Survey Question 4 (Time Consumption of Admin Tasks)

No respondent rated their administrative burden as a 1 or 2, indicating that none of the therapists finds their administrative work easy or efficient. The mean score was 3.89 out of 5. This shows that a majority of therapists believe their administrative work to be lengthy, which aligns with the multi-system workflow described previously.

Cost was identified as the primary barrier by 68% of respondents when asked to identify any barriers that prevent them from changing administrative tools. Setup time and fear of losing data were tied as the second most common barrier at 37% and only three participants reported being satisfied with their current system. This reinforces the idea of dissatisfaction with current workflows, but being constrained by cost.

Participants were asked if their current administrative process had ever caused issues for clients, such as missed receipts, scheduling errors or missed notes. Eight respondents confirmed that their process had caused issues for a client, nine denied any issues, one chose not to disclose an answer and one additional respondent noted a scheduling error. Due to the sensitive nature of therapy practice, these figures highlight a serious risk associated with fragmented administrative workflows.

Lastly, ten participants responded to an open-ended question following the survey that asked for any additional comments or feature suggestions. The most common theme throughout the responses was security and confidentiality, noting that GDPR compliance was essential to a platform.

The answers for additional features included a desire for video calling compatibility, personal task reminders, end-of-year accounting functionality and an AI assistant. While some of these features fall outside the scope of a basic administrative platform, they suggest opportunities for future development.

Overall, these findings provide strong evidence that an all-in-one, affordable and intuitive administrative platform would address the core needs of this group of participants. Features like automation, simple set-up and GDPR compliance are highly desired and would increase the likelihood of application adoption.

3.2 Personas

Personas are fictional but realistic representations of target users, created using research data to help guide design decisions throughout development. A singular persona was created based on the analysis of survey results to represent the primary user of TherapEase (Figure 6). The reason for only one persona was that it would be designed exclusively for individual therapists. While future development could introduce additional user types, such as clients through a client-facing portal or administrators within a small practice, these were outside the scope of this project and a second persona was therefore not needed. Sarah Byrne reflects the main target user of the application, encapsulating all goals and pain points identified by the therapists surveyed.

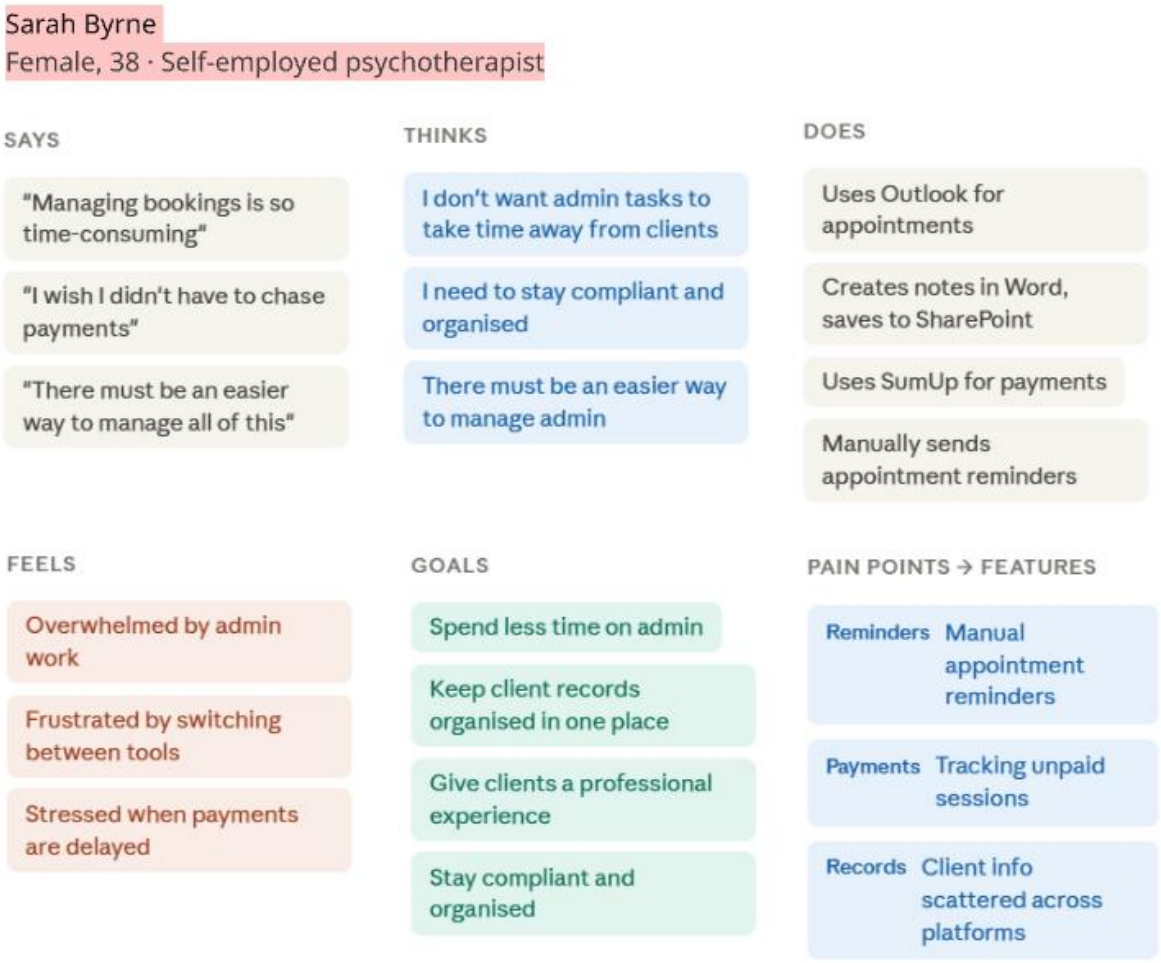

Figure 6 - Persona 1

Sarah is the primary user of TherapEase. Her pain points show how the application's core features, such as automated reminders, integrated payment tracking and client record management, are essential to the application and would improve workflow efficiency. TherapEase aims to replace the fragmented mix of tools she currently relies on.

3.3 Use Case Diagram

Use Case Diagrams show how users interact with a system and the system's automated processes. Figure 7 shows the primary functions of the application, showing how the user interacts with

it and the system's automation. Users can create and manage their own account, clients, appointments and payments, whereas the system automatically handles the email confirmations, SMS reminders, Zoom link generation and Stripe payment link generation.

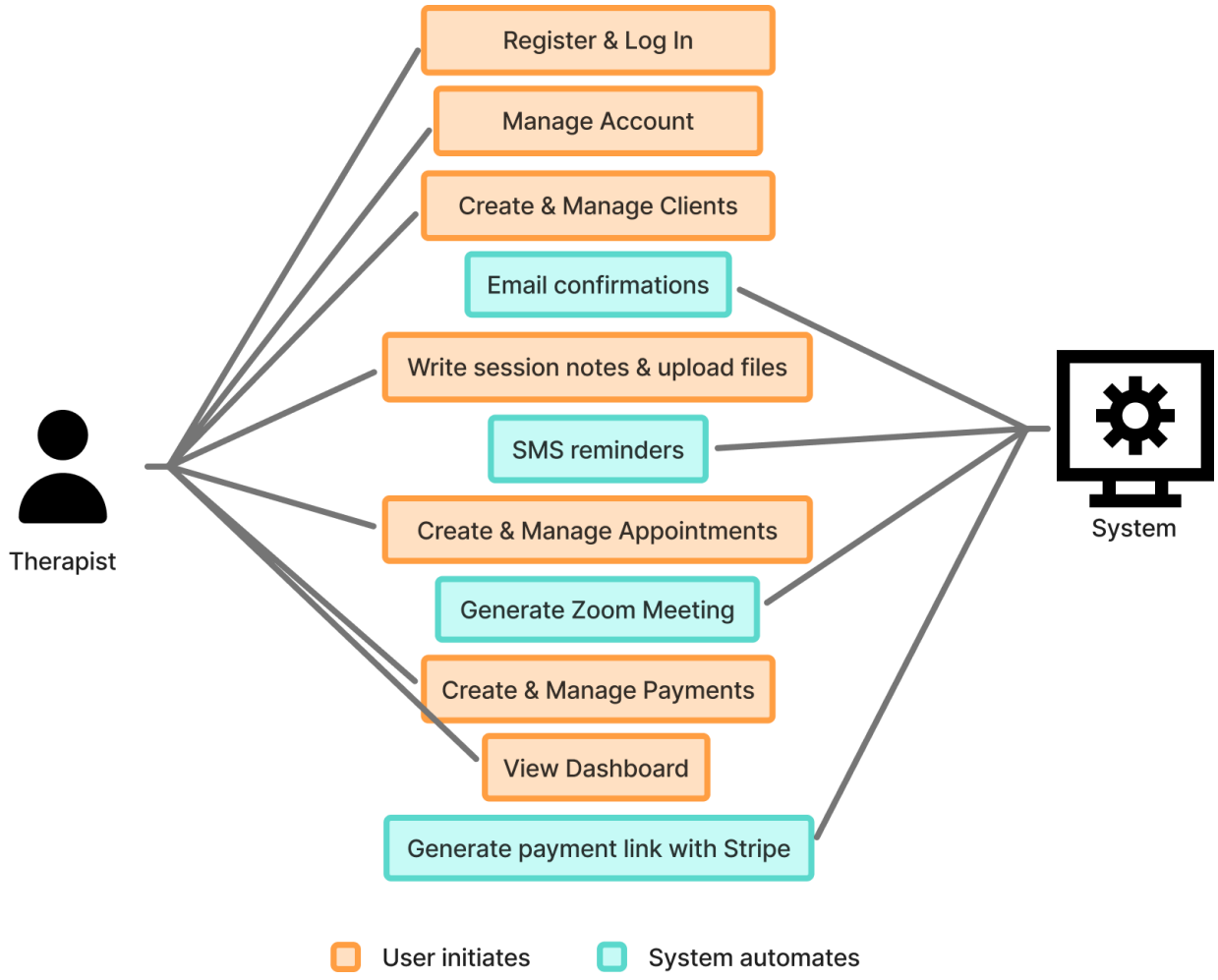

Figure 7 - Use Case Diagram of TherapEase

3.4 Use Cases

Use cases illustrate how users, based on the persona identified above, interact with TherapEase in realistic scenarios and is based on their needs, behaviours and pain points. They demonstrate how the

platform supports daily workflows for both therapists and clients, from managing appointments and payments to tracking practice metrics.

3.4.1 Use Case 1: Adding New Client

Persona: Sarah Byrne (Therapist)

Scenario: Sarah needs to add a new client and ensure all required documentation is uploaded before their first session.

Steps:

1. Sarah logs into TherapEase with secure 2FA authentication.
2. From the dashboard, she clicks “Add New Client.”
3. She enters personal details: name, contact information, date of birth and relevant notes about the client.
4. Sarah uploads intake forms, consent forms and any prior clinical notes.
5. TherapEase checks for missing mandatory fields and prompts Sarah if any are incomplete.
6. She saves the profile.
7. The system automatically confirms that the client profile has been created.
8. Optionally, Sarah can schedule the first session immediately after onboarding.

Outcome: The new client is fully onboarded with all sensitive data securely stored, reducing administrative errors and preparing the client for automated workflows like appointment reminders and payment processing.

3.4.2 Use Case 2: Scheduling & Managing Appointments

Persona: Sarah Byrne (Therapist)

Scenario: Sarah needs to schedule multiple types of sessions (online or in-person) while avoiding double bookings and ensuring clients receive timely reminders.

Steps:

1. Sarah logs into TherapEase with 2FA.
2. She navigates to the Calendar.
3. She clicks the “New Appointment” button.
4. She selects the client, date, time and session type (online or in-person) and opts to have a payment link sent before the session. She then enters the payment amount.
5. The system checks for errors and potential scheduling conflicts and notifies Sarah if any exist.
6. Sarah presses the “Submit” button to confirm the session.
7. TherapEase sends an automated email confirmation to the client with the appointment details and a Zoom link (if applicable).
8. The day before the session, TherapEase automatically sends an SMS reminder message to the client with a payment link attached.
9. If Sarah needs to reschedule or cancel, she can update the appointment in the system, triggering updated notifications to the client automatically.

Outcome: Sarah can manage appointments efficiently, avoid double bookings, reduce manual reminders and ensure clients receive accurate information, reducing stress and administrative workload.

3.5 Functional Requirements

Functional requirements define the core features and operations that the system must include and perform. These requirements were created based on the survey findings and persona analysis, ensuring that the system supports therapists' workflows and addresses the key usability challenges identified.

Category	Requirement
Security	The system shall allow users to register an account and log in securely
Security	The system shall implement authentication including login, logout, session handling and protected routes
Security	The system shall support two-factor authentication to enhance security and support GDPR compliance
Client Management	The system shall allow users to create, view, edit and delete client profiles
Client Management	The system shall allow users to create, edit and manage client notes
Client Management	The system shall allow users to link appointments to specific clients
Scheduling	The system shall allow users to create, view, edit and delete appointments
Scheduling	The system shall display appointments in daily and weekly calendar views
Scheduling	The system shall display appointment statuses (upcoming, cancelled, completed)
Dashboard	The system shall provide a dashboard displaying key information such as appointments and income summaries
Payments	The system shall support integrated payments through an external payment provider
Payments	The system shall automatically generate and send payment receipts
Automation	The system shall send automated notifications including confirmations, cancellations and reminders
Automation	The system shall support recurring appointments
Automation	The system shall generate online meeting links for remote appointments

3.6 Non-functional Requirements

Non-functional requirements define the quality standards that the system must meet, rather than specific features. These requirements describe how well the system performs the functional requirements, covering the three main categories of security, performance and user experience.

Category	Requirement

Security	The system shall ensure all sensitive user data is securely stored and encrypted
Security	The system shall comply with GDPR principles for data protection and privacy
Security	The system shall implement secure authentication mechanisms, including password hashing and two-factor authentication
Security	The system shall prevent common web vulnerabilities such as Cross-Site Scripting (XSS)
User Experience	The system shall provide a user-friendly and intuitive interface for non-technical users
User Experience	The system shall handle errors gracefully and display meaningful messages to the user rather than crashing
User Experience	The system shall be responsive and have functionality across different screen sizes, browsers and devices.
Performance	The system shall respond to user actions within an acceptable time frame
Performance	The database must be reliable so that data is stored safely and can be accessed quickly.

4. Design

This chapter outlines the design of TherapEase, beginning with the database design that structures the system, followed by the interface design and overall system architecture. The design is guided by user-centred principles, ensuring the application aligns with real-world therapist workflows while prioritising usability, efficiency and data organisation.

4.1 Database Design

An entity relationship diagram (ERD) is a visual representation of how items in a database relate to each other. The ERD for TherapEase (Figure 8) shows the database models used to support the main features of the application. These include Clients, Appointments, Payments, Reminders, ToDos, Notes, Files and User. The design centres around the User, who manages multiple clients, with each client linked to appointments, notes, files and reminders. Appointments then link clients to payments and notes, whereas ToDos, Reminders and Files support the administrative workflow. This structure allows the application to efficiently handle therapist workflows within an integrated system by connecting all key pieces of data.

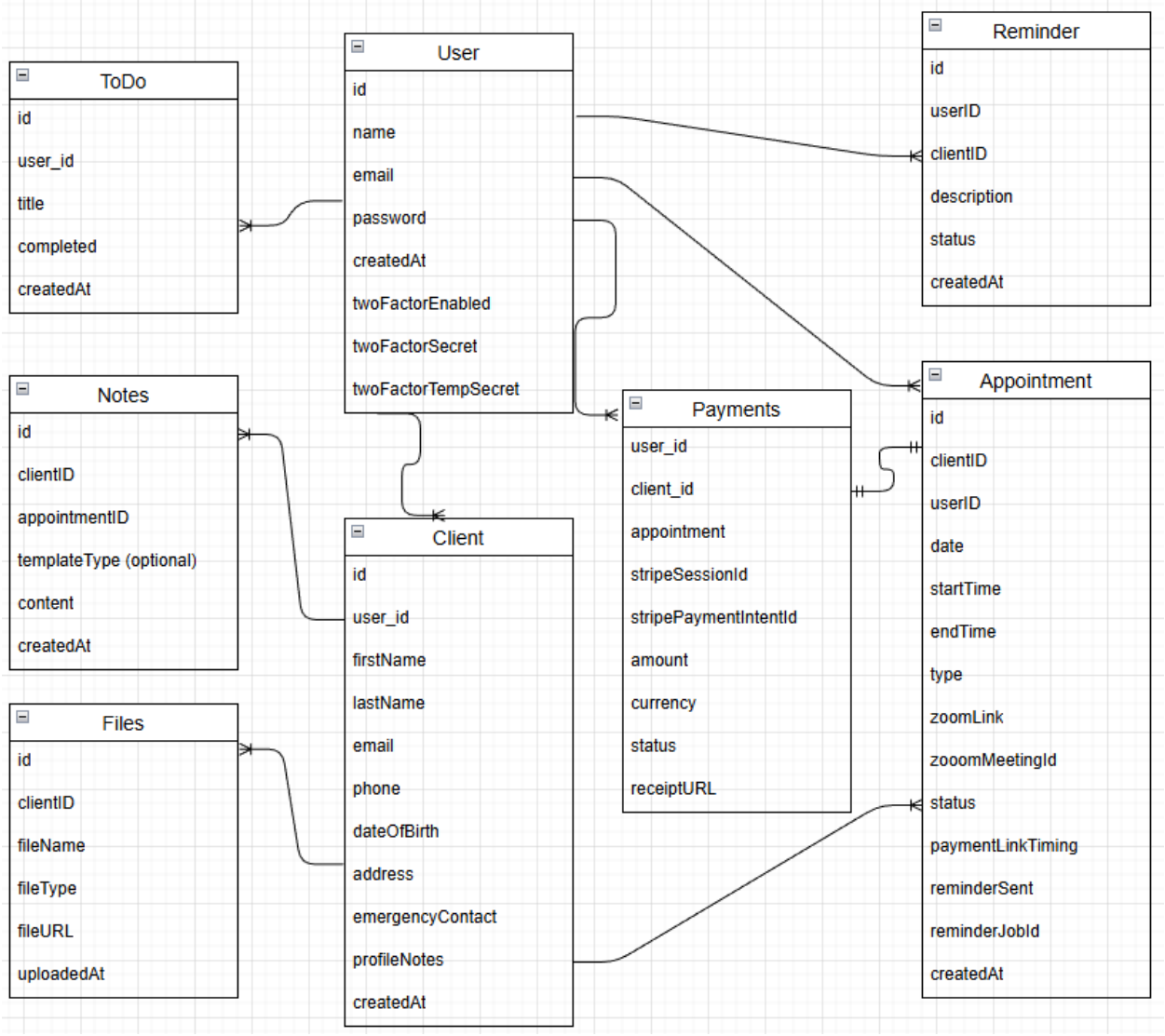

Figure 8 - Entity Relationship Diagram

At the centre of the database is the User collection, which represents the therapist or system user. A one-to-many relationship exists between the User and the Client, as a therapist manages multiple clients. The Client collection stores the personal information like name, address and emergency contact with an additional profile notes attribute where the user can write notes about the client that isn't specific to an appointment. The Appointment collection connects to the Client through a one-to-many relationship and includes attributes such as date, start and end time, session type and status.

Aside from these main collections, other collections support functionality in the system such as Payments, which is linked to appointments, for storing amount, currency, status and payment intent. Notes stores the clinical notes that are specific to each appointment and client. Files supports document uploads such as intake assessments and consent forms and is linked directly to Clients. Reminder and ToDo support the administrative workflows with the Reminders collection, allowing therapists to track outstanding actions to client profiles or session notes, while ToDo items help therapists track their own personal tasks.

4.2 Interface Design

4.2.1 User Flow Diagrams

Two user flow diagrams were created to represent the actions taken by the user when completing key tasks with TherapEase. The purpose of these diagrams is to ensure the task completion follows a logical and efficient path that minimises unnecessary navigation and cognitive load. The two diagrams depicted in

Figure 9 are the workflows for creating a client and scheduling an appointment. These workflows were chosen as they are the most frequent and essential administrative tasks for therapists.

Both workflows were intentionally designed to be consistent and follow the same process to reduce the learning curve and ensure familiarity across different functions. The user will always start out by logging in and being prompted to complete the two-factor authentication. This is an extra security measure that is implemented to ensure the protection of sensitive data and to be compliant with GDPR guidelines. Once authenticated, the user is redirected to the dashboard, where navigation to key areas such as Clients or Calendar is available through a sidebar. For both workflows, actions such as creating a new client or scheduling an appointment are handled through modals, allowing users to input required information without leaving the current page. This design reduces unnecessary navigation and supports

workflow continuity. Input validation is done before the user submits to minimise errors and prevent repetition from the user. Once submitted, a toast notification will confirm the client has been created or the appointment has been scheduled and that an email confirmation has been sent to the client. The differences in the workflows are intentionally minimal, with the only distinction being the type of information being entered.

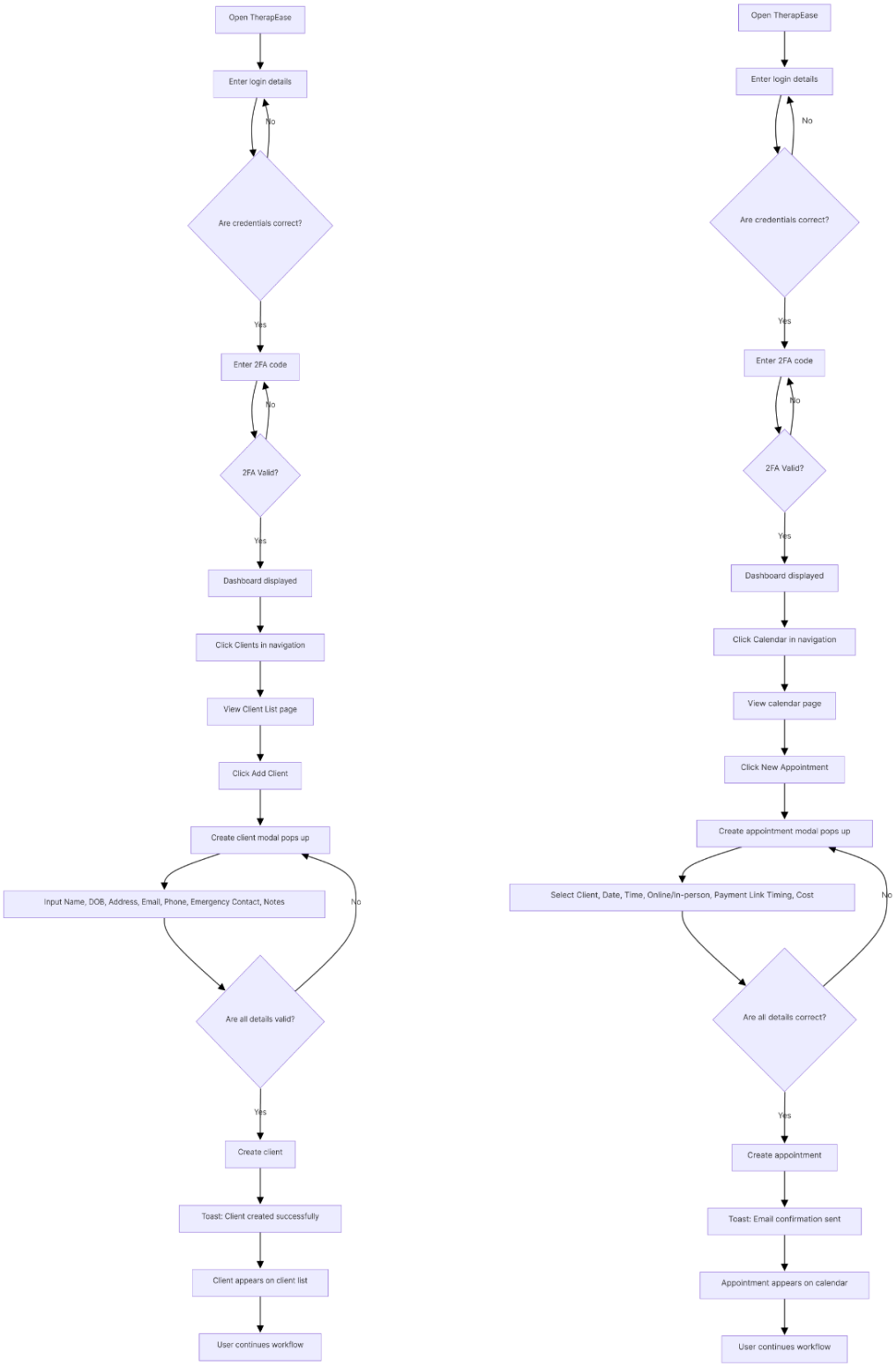

Figure 9 - User Flow Diagrams - Create Appointment (Left) & Add Client (Right)

4.2.2 Prototypes

Prototypes were created in Figma early in the design process to establish the application's overall layout and user interaction patterns. These designs were later refined before development to ensure usability and workflow alignment.

The dashboard in Figure 10 was chosen as the home page to provide users with immediate access to key information and quick actions. It includes elements such as upcoming appointments, key metrics, reminders and a to-do list, reducing the need for additional navigation. A sidebar navigation was chosen over a top navigation bar because it allows for easier scanning and offers better scalability as additional features are introduced.

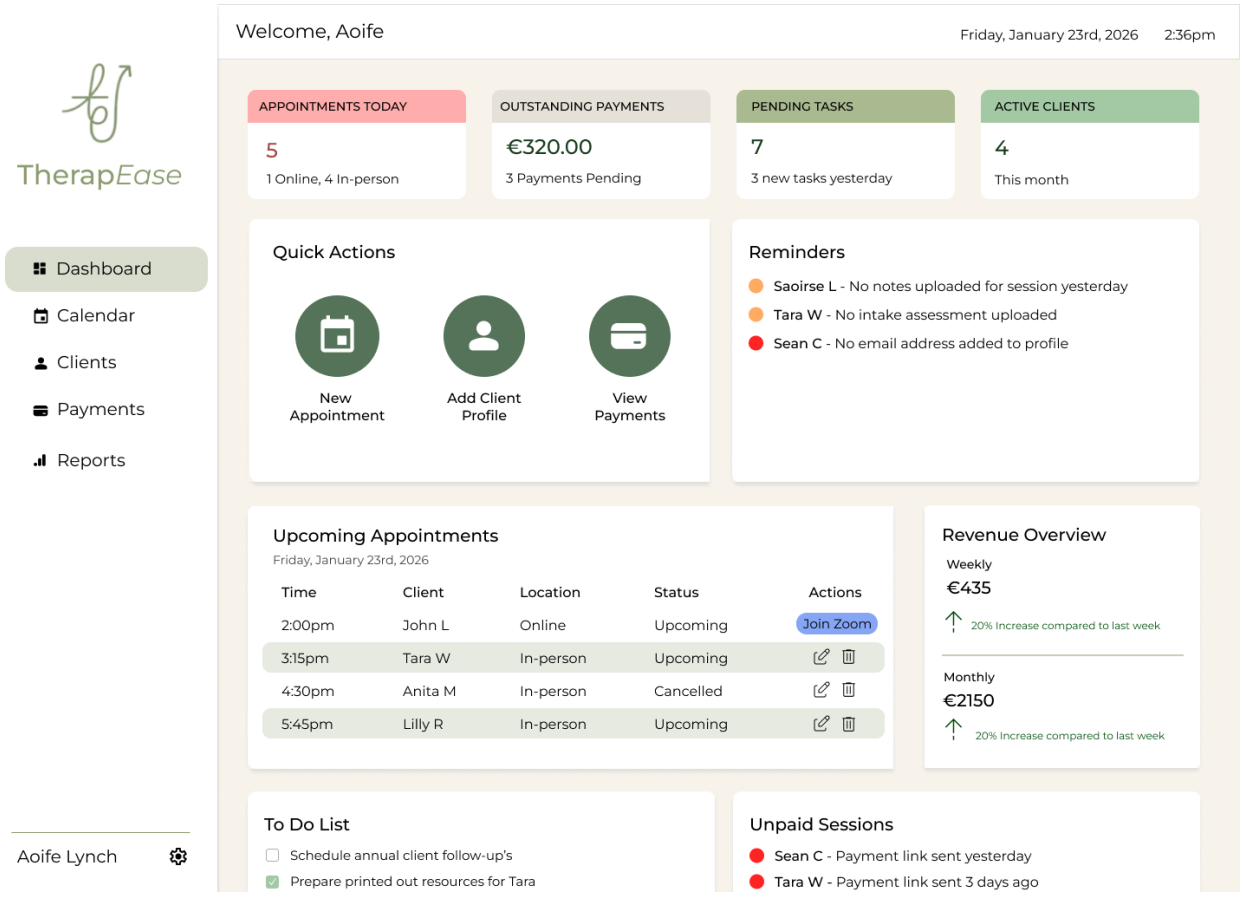

Figure 10 - Dashboard Prototype

The calendar interface, shown in Figure 11, was designed to provide a clear overview of appointments, with options to switch between daily, weekly and monthly views. Colour-coded appointment cards distinguish between online and in-person sessions. Filtering options were included to allow users to quickly identify relevant appointments, supporting efficient schedule management.

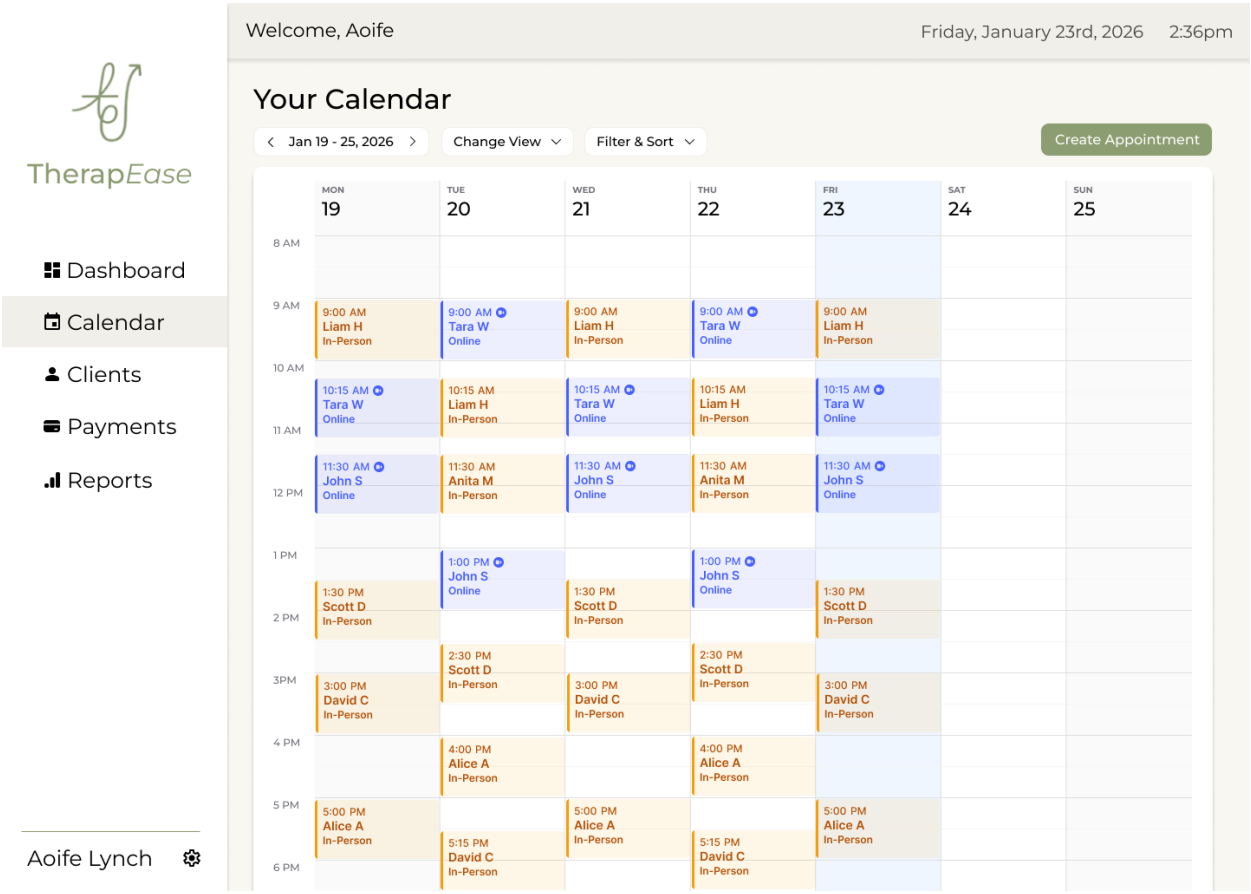

Figure 11 - Calendar Prototype

Client management is divided into a list view and an individual client profile (Figure 12). The list view provides a structured overview of all clients, with inline actions for viewing, editing and deleting profiles to minimise unnecessary navigation. The client profile's purpose is to show all relevant information for a client, including personal details, appointments, notes and file storage.

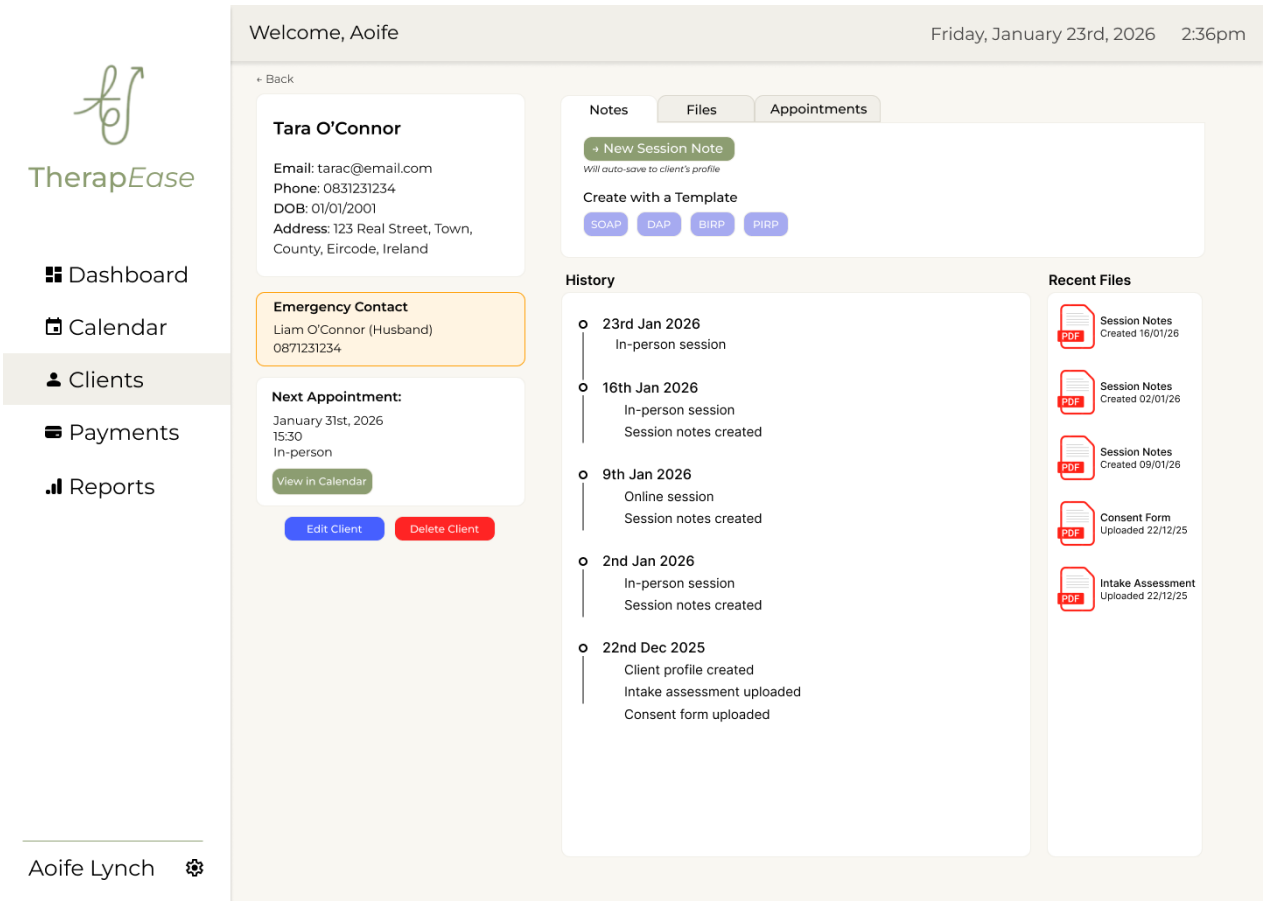

Figure 12 - Client Profile Prototype

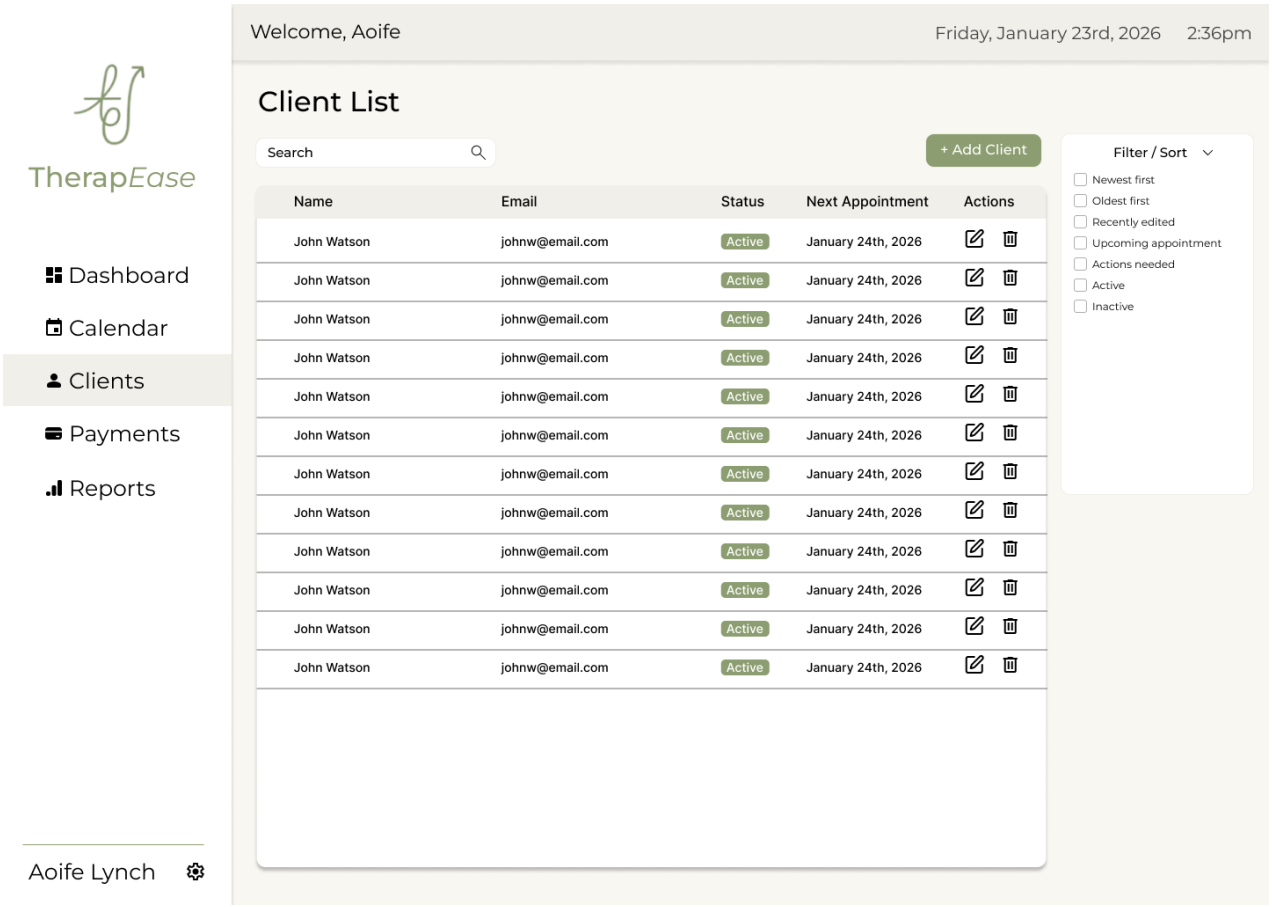

Figure 13 - Client List Prototype

The payments interface (Figure 14) is designed to provide therapists with a clear and structured overview of all financial transactions within the system. It is divided into a list view, displaying all payments and a detailed view linked to individual clients and appointments. The list view shows key information such as payment amount, status, date and client, allowing users to quickly monitor outstanding and completed payments. Integration with Stripe enables the generation of payment links and automatic receipt handling, removing the need for manual processing. The interface is designed to be consistent with other sections of the application, using tables for structured data and clear visual indicators for payment status, supporting efficient financial management and reducing administrative workload.

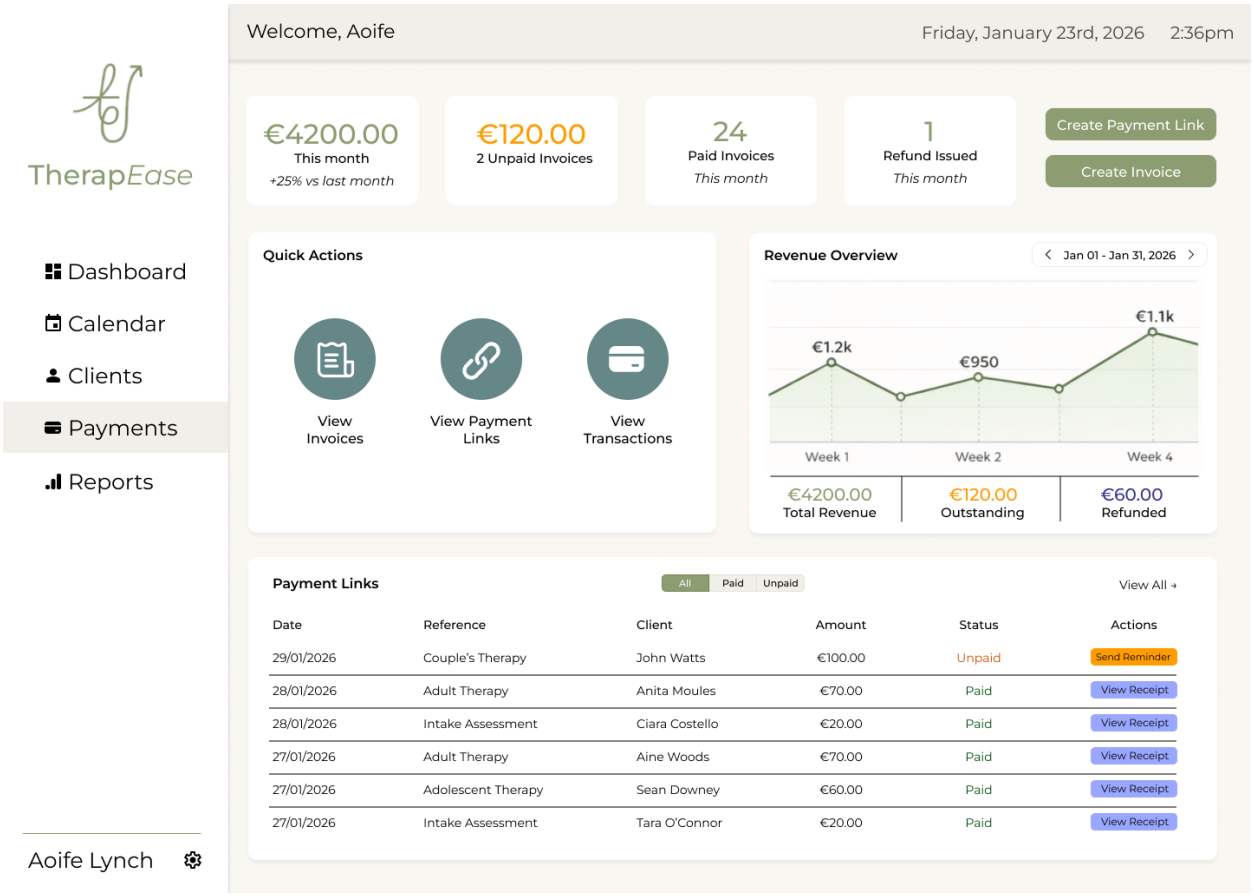

Figure 14 - Payments Page Prototype

Consistent UI patterns were applied throughout the application to improve usability. Modal interfaces are used for creating and editing clients and appointments, allowing users to complete tasks without navigating away from their current page. Tables are used for structured data such as client lists and payments, while card components are used to group related information and highlight key metrics.

Several refinements were made during development to improve usability. The colour scheme was adjusted to increase contrast between text and background elements, improving accessibility. Spacing and typography were standardised to ensure consistency across the interface.

4.2.3 UX Decisions

The goal for the user experience was to ensure easy navigation, reduce the need for switching between tools and make the administration work as seamless as possible to complete. A persistent sidebar was chosen so that users could quickly the core features, while remaining always visible so that the user has a clear sense of location within the application. This reduces unnecessary navigation and supports efficient task completion.

Automation is a key part of the application as it reduces the administrative burden placed on therapists by handling repetitive tasks on their behalf. By sending email confirmations and reminder SMS messages automatically, human error is reduced and time is saved for the user.

Using the Heuristics for User Interface Design by Nielsen (1994), the interface is designed to be simple and clearly structured, with important information prioritised to reduce cognitive load. By showing essential data, such as upcoming appointments and a revenue overview, users can quickly access important information without additional navigation. Users can also create an appointment or a new client profile from multiple pages, allowing them to pick whichever workflow works for them instead of being forced into a specific sequence of actions.

4.2.4 Styling

The colour palette chosen for TherapEase was curated to include green, beige and blue (Figure 15). Soft green tones are most commonly associated with mental health, well-being and calmness, making them an appropriate primary colour scheme for the application. These colours also contrast well with the secondary colours of beige and cream. The addition of blue creates a sense of professionalism and trust which is important for an application handling sensitive information.

Figure 15 - Application Colour Palette

The font family selected for TherapEase is Inter. Inter is a sans-serif font that conveys a clean, professional look for TherapEase (Figure 16). Inter was chosen due to its high readability across different screen sizes, making it suitable for a web application that aims to prioritise clarity, professionalism and creating a user-friendly experience.

Figure 16 - Application Typography & Logo

4.3 Process Design

TherapEase is built using the MERN stack (MongoDB, Express, React.js & Node.js) to create a full-stack application. Node.js and Express are used to implement the backend by creating a RESTful API that handles communication between the client and server. MongoDB was used as the database due to

its flexible document-based structure. The frontend is developed with React.js, using Tailwind CSS (Tailwind Labs Inc., 2025) as the styling framework, which allows for rapid UI development and consistent design. Docker is used to containerise and run the backend stack as a small, isolated development environment. This supports consistency across development and deployment

The application follows a client–server architecture, where the React frontend communicates with an Express-based backend via RESTful API requests. The backend is structured with routes, controllers, services and models to ensure that different parts of the system can be maintained and extended independently.

4.3.1 External APIs

TherapEase integrates several external APIs to support core application functionality and enable automation within the therapist's workflow. These integrations include Stripe, Zoom, Nodemailer, Twilio, Redis and BullMQ. Stripe is used to facilitate secure payment processing, allowing the application to generate payment links and automatically issue receipts following a successful transaction. Similarly, the Zoom API is integrated to automatically generate video call links for online appointments, reducing the need for manual setup. Automated emails and SMS messages are implemented using Nodemailer and Twilio. Asynchronous processing is incorporated to support time-sensitive operations, such as email confirmations and reminder SMS messages, ensuring these tasks do not affect overall system responsiveness.

4.3.2 Security

Security in TherapEase is designed to protect sensitive client data and ensure GDPR compliance. Authentication is implemented using JSON Web Tokens (JWT), ensuring that only authorised users can access protected routes within the system. To further enhance security, two-factor authentication (2FA)

is incorporated as an additional verification step during login, reducing the risk of unauthorised access. The system flow and user flow of 2FA is shown in Figure 17.

In addition to authentication, input validation and sanitisation are applied to prevent common vulnerabilities such as cross-site scripting (XSS). Secure communication between the frontend and backend is maintained through controlled API access and validation of incoming requests. This layered approach ensures that security is embedded into the overall system design, reducing the risk of unauthorised access and common security vulnerabilities.

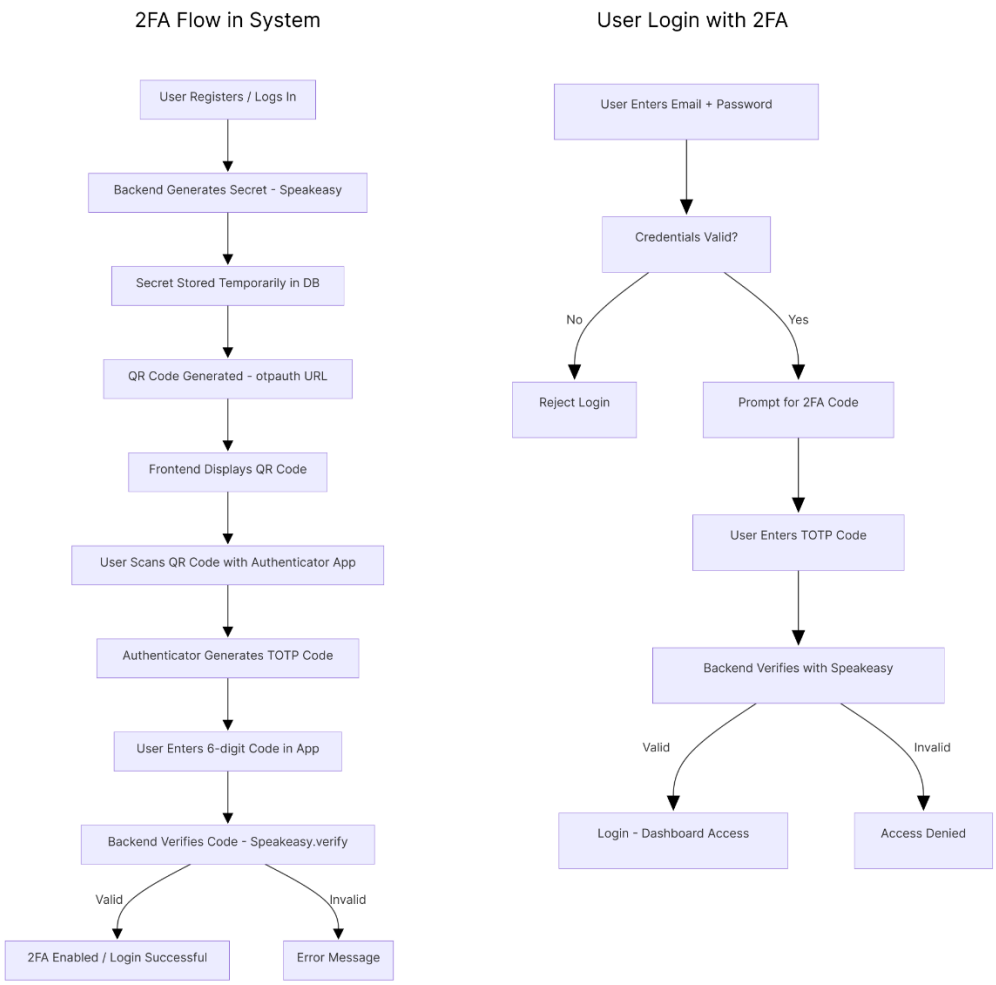

Figure 17 - Two-Factor Authentication Flow

4.3.3 Event Handling

The Model-View-Controller diagram in Figure 18 illustrates how the system handles user events. Actions come from the view layer, which is the React frontend. The frontend uses Axios to send HTTP requests to the backend, which the controller layer receives. The controller processes the request and delegates business logic to the service layer. This service layer can trigger asynchronous background tasks. Other operations are passed to the model layer which interacts with MongoDB to store and

retrieve data. Responses from this process are then sent back to the frontend to update the user interface.

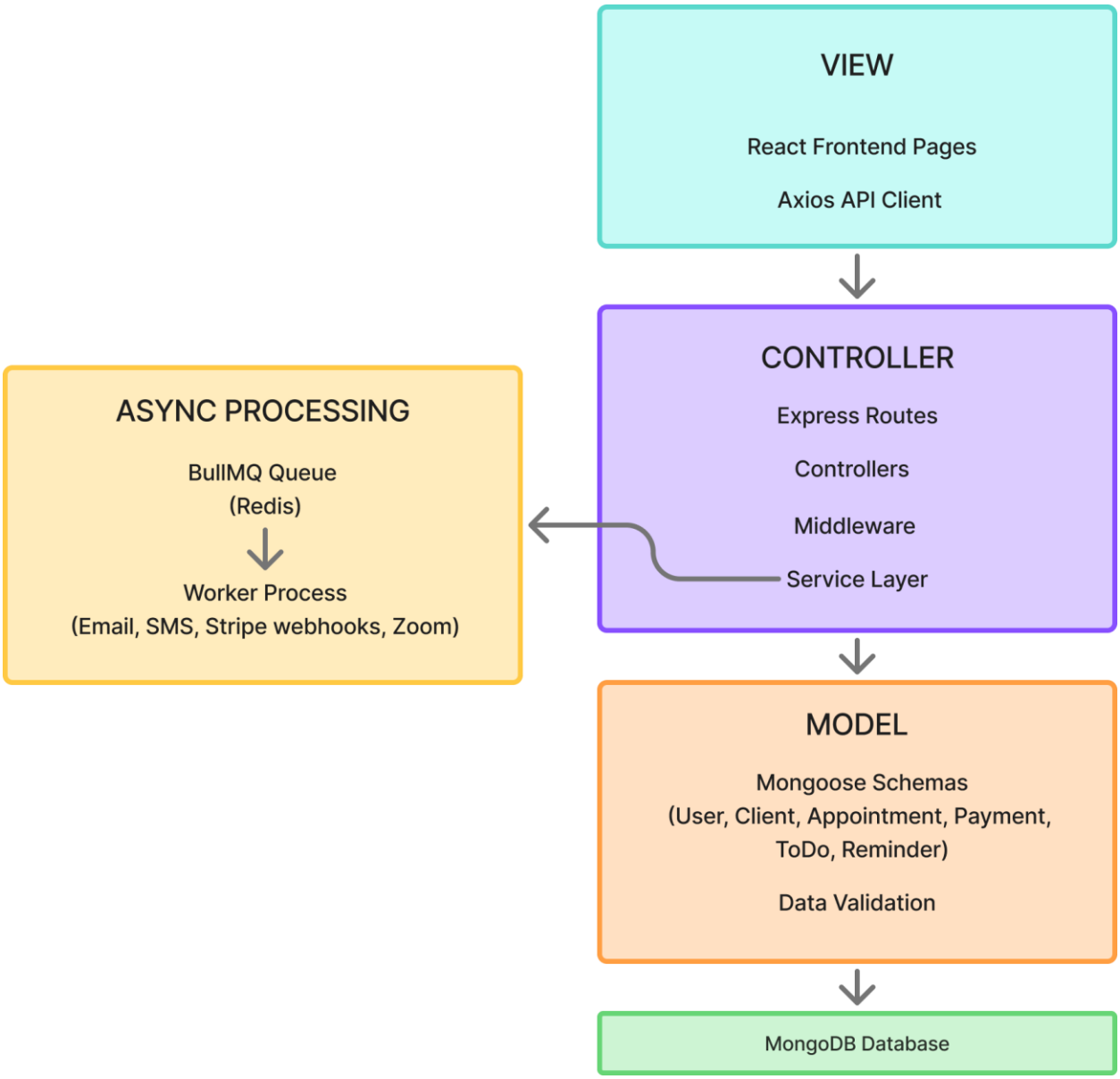

Figure 18 - Model-View-Controller Diagram

The main events handled by the system include viewing, creating, updating and deleting appointments and clients. Alongside that, the user can create payment links, view successful Stripe

payments, send appointment confirmation emails, appointment modification emails, Stripe receipt emails and SMS reminders. These events are a mixture of directly handled behaviour and asynchronous processing.

5. Implementation

This chapter outlines the six sprints undertaken to develop the application, including the goals, outcomes, challenges encountered and solutions implemented. It also explains how key features, such as Stripe payments and asynchronous email and SMS processing, were integrated using the technologies and design decisions discussed in the previous chapter.

5.1 Development Tools & Environment

The main development tools used were Visual Studio Code (Microsoft, 2025b), GitHub (GitHub, 2025a), Postman (Postman, 2025) and Docker (Docker, 2024). Visual Studio Code was used as the Integrated Development Environment due to its extensions, integrated terminal and debugging tools. GitHub was used for version control, allowing regular commits, rollbacks if needed and direct deployment to Render (Render, n.d.) from the repository (Appendix F). Postman was used within Visual Studio Code to test the database and API calls. GitHub Copilot (GitHub, 2025b) acted as a support tool for code suggestions, debugging assistance and implementing repetitive code (Appendix C). However, all outputs were reviewed and adapted to ensure compatibility with the project. Docker was used to containerise the application environment, allowing the backend, database and supporting services to run consistently throughout development.

5.2 Methodology

TherapEase was developed using an Agile approach, which prioritises iterative development, collaboration and responsiveness to change (Beck et al., 2001). This allowed features of the application to be built, tested and edited throughout the project, rather than developing the entire system in one go. The workload was divided across six sprints.

Iterative development was essential due to the complexity of the features being implemented. For example, appointment scheduling was initially developed as a basic feature, and email confirmations, Zoom link generation and payment integration were added once the core functionality was successfully implemented. The project backlog was refined across each sprint, with features reprioritised based on technical constraints and user feedback.

The six sprints and their goals are as follows:

Sprint	Goals
1	Choosing technologies, designing database & codebase setup
2	Creating backend application logic, creating API, JWT implementation and error handling.
3	Asynchronous processing of email confirmation, connecting the API to the frontend and implementing two-factor authentication
4	Zoom API integration, design of frontend pages and SMS message queue, setting up Stripe API
5	Improving frontend consistency, designing the client profile page and finishing the integration of Stripe API
6	Improving user feedback, revising dashboard design, enhancing payment workflow and deploying the application

5.3 Sprint 1

Prior to the first sprint, a discussion was held with a therapist to determine what was essential to the application and what could be left for future improvements. This was important to outline early so that a clear Minimum Viable Product (MVP) could be defined so development could focus on the core administration workflow. These features included calendar scheduling, a client profile, a section for writing session notes, and automation of time-consuming tasks, such as sending receipts and session reminder texts. With the MVP defined, the goals for the first sprint included creating initial prototypes, deciding which technologies to use, designing the database, and setting up the project environment.

5.3.1 Goal 1 – Technologies

Node.js with Express was chosen for the backend, React.js for the frontend and MongoDB for the database, forming the MERN stack. These technologies were selected to support a full-stack web application with asynchronous background processing. Docker was used for containerisation. Figure 19 shows the overall architecture and the relationships among the API, database, background worker, and external services.

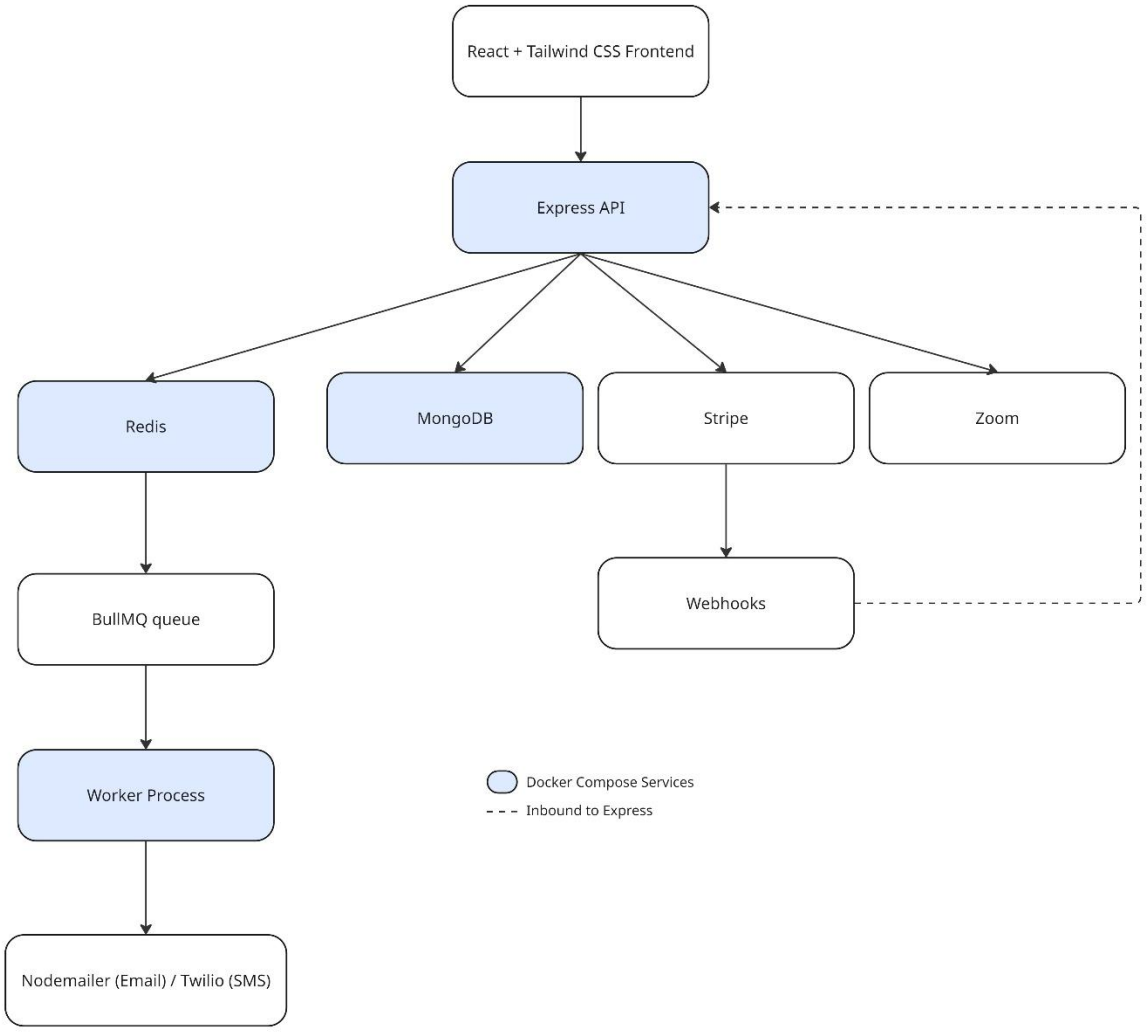

Figure 19 - System Architecture Diagram

5.3.2 Goal 2 – Database Design

An ERD was created to define the collections, their contents and the relationships between them. This was important at an early stage so the likelihood of major changes in later development could be reduced. Related data is connected through object references, such as linking appointments and notes to specific clients, which reduces duplication and supports retrieval across the system. A full list of collections is shown in Figure 20.

TherapEase > therapease

Collection name	Properties	Storage size	Data size
appointments	-	36.86 kB	3.52 kB
clients	-	36.86 kB	1.42 kB
files	-	36.86 kB	251.00 B
notes	-	36.86 kB	154.00 B
payments	-	36.86 kB	2.52 kB
reminders	-	4.10 kB	0 B
todos	-	36.86 kB	697.00 B
users	-	36.86 kB	1.39 kB

Figure 20 - MongoDB collections used to store application data

5.3.3 Goal 3 – Setup

Using GitHub to create a repository, the initial codebase was organised into frontend, backend, and end-to-end testing folders. A Docker Compose configuration (Appendix E) was created to run the API, MongoDB, Redis and a separate worker process together, ensuring a consistent development environment (Figure 21).

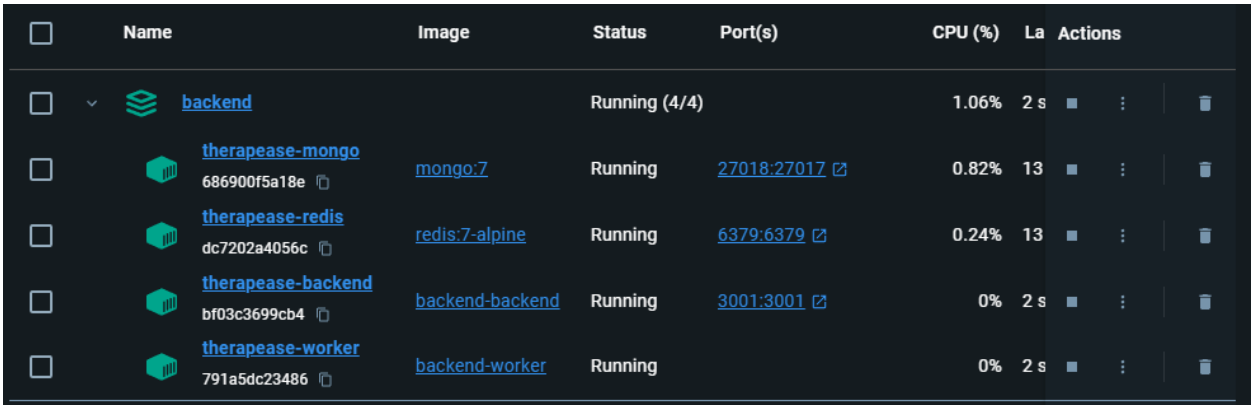

Figure 21 - Docker Container Environment showing Backend API, MongoDB, Redis and Worker service

5.3.4 Sprint Outcome

This sprint defined the MVP, selected the core technologies and established the development environment. The system architecture and database design were also completed, providing a clear blueprint for later development. While the system design was fully defined, no functional features were implemented at this stage as the sprint focused on planning and setup rather than development.

A key challenge during this sprint was narrowing down the scope of the application to a realistic MVP as initially, there was a risk of including too many features, which would not have been achievable within the project timeline. As a result of defining the MVP, several lower-priority features such as advanced reporting, invoice generation and a client portal were moved to the backlog as future improvements.

5.4 Sprint 2

The goal for sprint two was to begin building the application's backend. This included creating the models, controllers, routes, and services for the API, implementing authentication, and building the

API. Authentication setup included not only login and registration but also the transition from session-based IDs to JSON Web Tokens (JWTs). Error handling was also introduced to improve debugging.

5.4.1 Goal 1 - Application Logic

To create the database, models were first created to define the collection schema. For example, in the Client model, relevant fields were defined, along with logic for cascading deletes so that if the user deletes a client, all associated files, appointments, notes and reminders will also be deleted. Originally, the backend had all database functionality, including routing, validation service logic in the controller. This approach was based on prior experience with a much smaller project, but as more functionality was added, the controller files got too large and difficult to debug. Therefore, it was restructured, with controllers refined to handle only incoming requests, the overall application logic, and task delegation to the service layer. Service files are created in the service layer to execute tasks sent by the controller and return a response (Stafford, 2003). Route files were created to map HTTP requests to the controller functions. Validation was also separated and initially minimal to confirm functionality, but when this eventually led to invalid data being sent to the database, more extensive validation was added in later sprints to prevent invalid inputs and improve usability. The complete folder structure of the backend is shown in Figure 22.

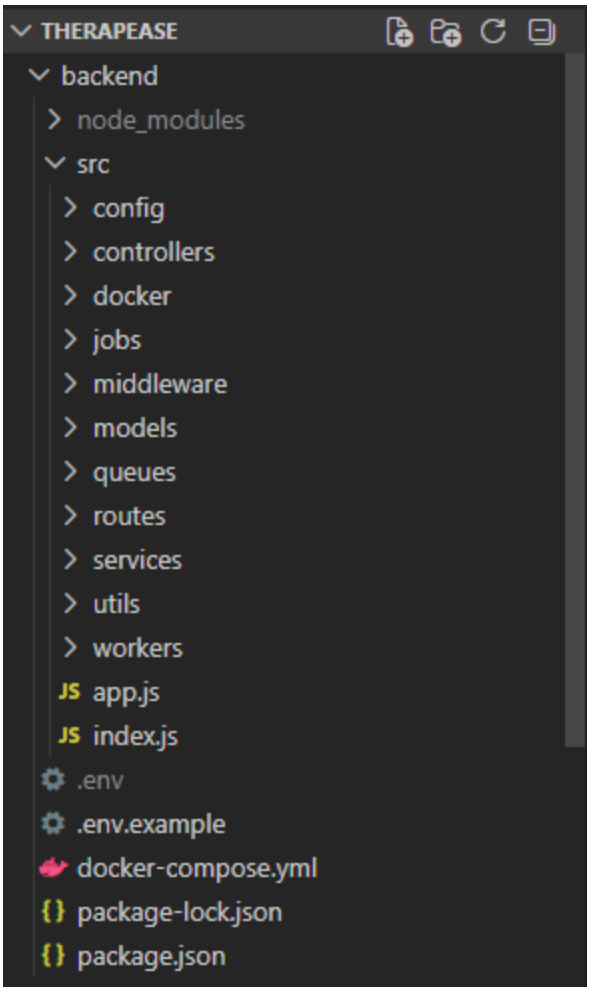

Figure 22 - Backend Folder Structure in Visual Studio Code

5.4.2 Goal 2 – API Structure

A RESTful API was created with Express to enable communication between the frontend and backend, allowing the frontend to retrieve data from the database and send updates using HTTP requests (Fielding, 2000). The API endpoints followed a consistent /api routing structure and were created for all key features of the application. This included clients, appointments, payments, notes, reminders, to-dos and authentication. The payments endpoint remained incomplete until Stripe was integrated in a later sprint and the reminders endpoint had no functionality until later in development, as the workflow was not fully defined. Therefore, payments and reminders were both moved to the

backlog. Authentication endpoints were responsible for the user registration, login and token verification. Postman was initially used to test the API routes and was then published using Swagger (SmartBear Software, 2021), which lists endpoints clearly and allows testing (Appendix D). These endpoints are shown below in Figure 23.

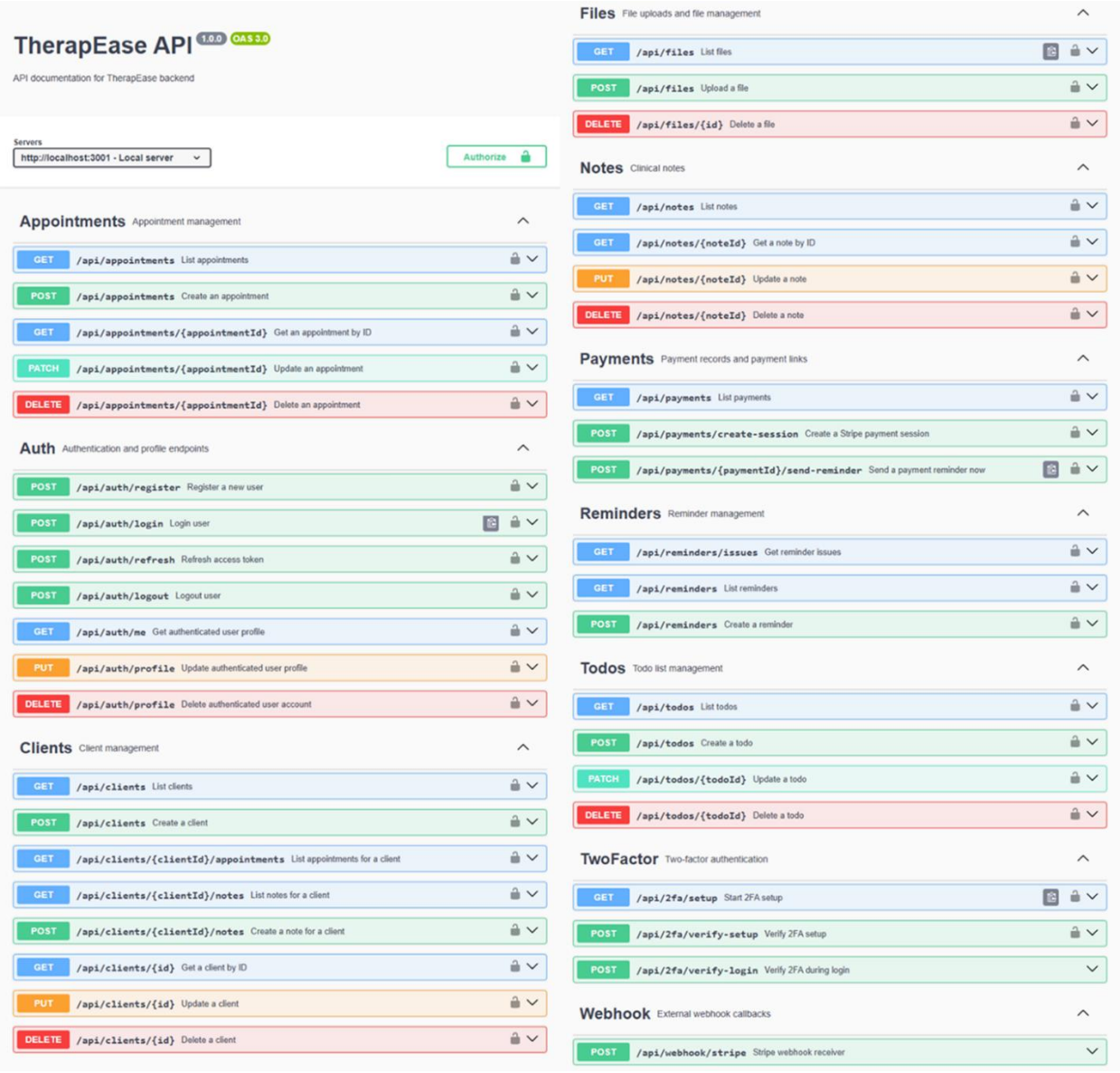

Figure 23 - Swagger Documentation of API Routes

5.4.3 Goal 3 – JWT Implementation & Error Handling

The authentication system was originally using session-based IDs as it was more familiar from previous work and was suited for basic login protection. However, this was then changed to JSON Web Tokens (JWT), which was better suited to the stateless nature of the API. Login and registration had to be redesigned to issue tokens after successful authentication and protected routes were updated to validate bearer tokens on each request through centralised middleware rather than server-side session storage.

Switching to JWT-based authentication increased the use of asynchronous operations, particularly for token verification and external API calls, which initially caused errors in async controllers that were not passed to Express middleware. To resolve this, a reusable asyncHandler wrapper was implemented to automatically forward rejected promises, removing the need for repetitive try/catch blocks and ensuring consistent error capture (Figure 24, Figure 25). Error handling was further structured across the backend, where validation middleware rejected invalid input early, and the service layer used a HttpError class to return appropriate status codes for expected issues such as authentication failures or missing data. Centralised error middleware then handled all remaining errors, including database and JWT issues and returned consistent responses. On the frontend, an Axios interceptor was used to attach tokens to requests and handle authentication errors, such as retrying requests when a token expired or redirecting the user to login if needed. This approach improved consistency, reduced duplicated code and made the application more reliable. Improving the usability of error displays is discussed in Section 5.8.1.

```
export const asyncHandler = (fn) => (req, res, next) => {
  Promise.resolve(fn(req, res, next)).catch(next);
};
```

Figure 24 - asyncHandler Code Snippet

```
router.get('/', asyncHandler(appointmentsController.getAllAppointments));
```

Figure 25 – Example of asyncHandler wrapping a route to handle asynchronous errors

5.4.4 Sprint Outcome

This sprint successfully established the core backend architecture. Authentication using JWT was implemented and error handling was introduced to improve system reliability. Some features, such as payments and reminders endpoints, were only partially completed.

A significant challenge during this sprint was transitioning from session-based authentication to JWT, which required reworking authentication logic across multiple parts of the application. More challenges arose when handling asynchronous errors in Express, requiring the asyncHandler wrapper to ensure consistent error handling. This caused additional tasks to be added to the backlog, including refining authentication flows and improving validation. The incomplete payment and reminder features were deferred to later sprints, where external integrations would be implemented.

5.5 Sprint 3

This sprint marked the beginning of frontend UI development and the implementation of asynchronous email processing and two-factor authentication. After the core API endpoints were established, the next priorities were to implement asynchronous email confirmations, connect the frontend to the backend and introduce 2FA. These goals were closely linked, as both API integration and 2FA required working a working frontend before they could be fully tested.

5.5.1 Goal 1 – Email Confirmation Queue (Asynchronous Processing)

To add asynchronous processing for sending email confirmations, BullMQ and Redis were first added to manage the job queue, with Nodemailer used to send the emails. This process was structured around a shared Redis connection, a queue for email jobs, and a worker process to execute the jobs. Redis was also added to the Docker Compose configuration so it could run alongside the API and database as its own service. The Redis connection file was used so both the main application and the background worker could connect to the same queue. The emailQueue file allowed the main application to add email jobs, while the worker process was configured to listen for and process them. The appointment service file was updated to send a confirmation email to the client when an appointment has been successfully created by adding a job to the queue using the emailQueue.add() function (Figure 27). The worker then takes the email details, such as the client's email address, subject and HTML, it then sends the email using Nodemailer, shown in Figure 26. Console logs were added to confirm whether emails were sent successfully and to detect any errors. The first version of this email confirmation is shown in Figure 28. This process became the template for asynchronous processing in the project for future SMS reminder development.

```
const worker = new Worker(
  "emailQueue",
  async (job) => {
    console.log("Processing job:", job.name);

    const { to, subject, html } = job.data;

    await transporter.sendMail({
      from: `"TherapEase" <${process.env.EMAIL_USER}>`,
      to,
      subject,
      html,
    });

    console.log(`Email sent to ${to}`);
  },
  { connection: redisConnection }
);
```

Figure 26 – Worker processing the email queue

```
const html = appointmentEmail({ client, appointment });

await emailQueue.add("appointmentConfirmation", {
  to: client.email,
  subject: "Appointment Confirmation - TherapEase",
  html
});
```

Figure 27 - Job Creation in the Appointment Service file

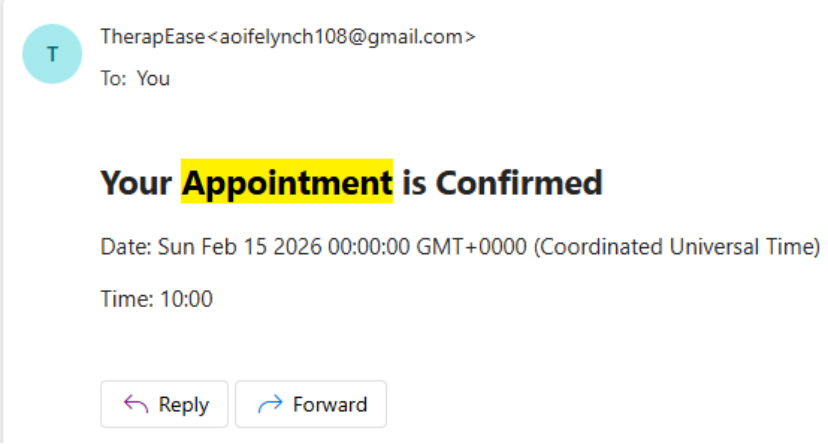

Figure 28 - Example of the first version of Email Confirmations

5.5.2 Goal 2 – Connecting API to Frontend

To start implementing features on the frontend, the API needed to be connected so data from the database could be displayed and CRUD could be performed. To complete this, an api.js file was created on the frontend and a single Axios configuration was set up (Figure 29). This was done so that all API requests follow the same structure and the base URL would only have to be defined once, as seen in Figure 30.

During this process, a Cross-Origin Resource Sharing (CORS) error occurred as the backend and frontend were running on different localhost ports, which caused the browser to block the connection. To resolve this, CORS middleware was added in the backend to allow requests from the frontend, permit the usage of HTTP methods and the Content-Type and Authorization headers, shown in Figure 31. On the frontend, the API base URL was set through an environment variable, so requests were consistently sent to the correct backend address. These changes allowed CRUD operations to function correctly without being blocked by the browser.

```
const api = axios.create({
  baseURL: import.meta.env.VITE_API_URL || 'http://localhost:3001/api',
  timeout: 10000,
  headers: {
    'Content-Type': 'application/json'
  }
});
```

Figure 29 - Axios API Client

```
export const clientsAPI = {
  async getAll() {
    const response = await api.get('/clients');
    return response.data;
  },

  async getById(clientId) {
    const response = await api.get(`/clients/${clientId}`);
    return response.data;
  },
```

Figure 30 - Example of API Routes for Clients

```
app.use(cors({
    origin: process.env.CORS_ORIGIN || ['http://localhost:5173',
    'http://localhost:3000', 'http://localhost:3001'],
    credentials: true,
    methods: ['GET', 'POST', 'PUT', 'DELETE', 'PATCH', 'OPTIONS'],
    allowedHeaders: ['Content-Type', 'Authorization']
}));
```

Figure 31 - CORS issue solution in app.js file

5.5.3 Goal 3 – Two-factor Authentication

A second layer of security was added through two-factor authentication using Time-based One-Time Passwords (TOTP). The Speakeasy library was used for TOTP generation and verification, along with the QRcode library to display the QR code. The backend used Speakeasy to generate a shared secret between the server and the user's authenticator app. This secret is stored in a twoFactorTempSecret field in the User model. An OTPAuth URL was generated from this secret and converted into a QR code

for display on the frontend. The totp.verify() function was used to check the 6-digit code the authenticator app provides against the temporary secret, if the code is valid, the secret gets moved to the twoFactorSecret field in the User model and the twoFactorEnabled field gets set to true. The 2FA setup can be seen in Figure 32.

The original login logic was modified so that, after the user's email and password are verified, the system checks the twoFactorEnabled field. If set to true, it would redirect to a 2FA verification page where the user can scan the QR code and enter the 6-digit code in their linked authenticator app. To prevent the user from accessing protected routes immediately after login, before 2FA verification, the JWT tokens were withheld, and the user was assigned a temporary user ID while completing 2FA (Figure 33). To test this entire process, simple frontend pages were created to display the QR code and provide an input field for the 6-digit code.

```
export const setup2FA = async (req, res) => {
  const user = await User.findById(req.user._id);
  if (!user) {
    throw new HttpError(UNAUTHORIZED, "User not found");
  }

  // Generate secret
  const secret = speakeasy.generateSecret({
    name: `TherapEase (${user.email})`,
    issuer: "TherapEase",
  });

  // Save temp secret to user (not confirmed yet)
  user.twoFactorTempSecret = secret.base32;
  await user.save();

  // Generate QR code
  const qrCode = await QRCode.toDataURL(secret.otpauth_url);

  res.status(200).json({
    message: "2FA setup initiated",
    secret: secret.base32,
    qrCode,
  });
};
```

Figure 32 - 2FA setup

```
export const verifyLogin2FA = async (req, res) => {
  const { token, tempUserId } = req.body;

  if (!token || !tempUserId) {
    throw new HttpError(BAD_REQUEST, "Authentication code and user ID
    are required");
  }

  const user = await User.findById(tempUserId);

  if (!user || !user.twoFactorEnabled) {
    throw new HttpError(BAD_REQUEST, "Invalid request");
  }

  const verified = speakeasy.totp.verify({
    secret: user.twoFactorSecret,
    encoding: "base32",
    token,
    window: 1,
  });

  if (!verified) {
    throw new HttpError(BAD_REQUEST, "Invalid authentication code");
  }

  // Issue real tokens
  const tokens = await authService.generateTokens(user);

  res.status(200).json({
    message: "2FA verification successful",
    user: safeUser(user),
    accessToken: tokens.accessToken,
    refreshToken: tokens.refreshToken,
  });
};
```

Figure 33 - Verifying 2FA on login and issuing tokens

5.5.4 Sprint Outcome

This sprint implemented asynchronous email processing using BullMQ and Redis, allowing email confirmations to be handled in the background. The frontend was also successfully connected to the backend API, which enabled data to be retrieved and displayed. Two-factor authentication (2FA) was implemented at a functional level but required further refinement in later sprints, particularly regarding user flow and frontend integration.

A major challenge during this sprint was resolving CORS issues when connecting the frontend to the backend. The browser initially blocked API requests due to different localhost ports, preventing communication between the two systems. The configuration of CORS middleware and environment variables resolved this. Additionally, implementing 2FA caused issues around withholding JWT tokens until verification was complete. As a result, refinement for 2FA and frontend authentication flows were added to the backlog.

5.6 Sprint 4

With the frontend now connected to the API and the core backend in place, Sprint 4 focused on expanding the application's functionality through external integrations and frontend page development. Additional key features implemented included the SMS message queue and Stripe setup.

5.6.1 Goal 1 – Zoom API Integration

To facilitate online appointments, the Zoom API was added to automatically generate Zoom meeting links for the appointment date and time. A Zoom service was created to communicate with the Zoom API and generate meetings using appointment details such as date, time and duration. In the Appointment service, a condition was added that, if the appointment type is set to online, it fills in the parameters and generates the link and meeting ID (Figure 34). The email confirmation content was changed to add the Zoom link if the appointment was online.

The Appointment service was further revised to manage Zoom links when appointments were updated, cancelled or deleted. The logic for handling these scenarios is outlined in pseudocode below, while the delete function is shown in Figure 35.

Update Appointment:

Find the existing appointment

If the appointment is changed to in-person or cancelled:

Delete the Zoom meeting if a Zoom meeting ID exists

Clear the Zoom link field

Clear the Zoom meeting ID field

If the appointment is rescheduled and remains online:

Delete the old Zoom meeting if a Zoom meeting ID exists

Create a new Zoom meeting with the updated date and time

Store the new Zoom joining URL

Store the new Zoom meeting ID

Save the updated appointment

Return the updated appointment

Delete Appointment:

Find the appointment

If the appointment is online and has a Zoom meeting ID:

Delete the Zoom meeting

Then delete the appointment from the database

Return a success message

```
if (appointmentPayload.type === "online") {
  const zoomMeeting = await createZoomMeeting({
    date: appointmentPayload.date,
    startTime: appointmentPayload.startTime,
    endTime: appointmentPayload.endTime,
    topic: `TherapEase Session - ${client.firstName} ${client.lastName}`
  });

  appointmentPayload.zoomLink = zoomMeeting.joinUrl;
  appointmentPayload.zoomMeetingId = zoomMeeting.id;
}
```

Figure 34 - Zoom Meeting Creation in Appointment Service

```
async deleteAppointment(appointmentId, userId) {

  const appointment = await Appointment.findById(appointmentId);

  if (!appointment) {
    throw new HttpError(NOT_FOUND, "Appointment not found");
  }

  if (appointment.user.toString() !== userId.toString()) {
    throw new HttpError(FORBIDDEN, "Forbidden");
  }

  // Delete Zoom meeting if online appointment
  if (appointment.type === "online" && appointment.zoomMeetingId) {
    await deleteZoomMeeting(appointment.zoomMeetingId);
  }

  await Appointment.findByIdAndDelete(appointmentId).exec();

  return { success: true };
}
```

Figure 35 - Delete Appointment function showing Delete Zoom Meeting

5.6.2 Goal 2 – Frontend Page Design

The dashboard page was initially developed to match the prototype, using rows of cards with equal heights to create a consistent layout. However, this caused issues when certain cards, such as upcoming appointments, contained more information than the card beside them, forcing the other card to expand to match and creating an unbalanced design. An example of this is shown in Figure 36. As this did not work well in practice, the decision was made to pause the dashboard design and revisit it in a later sprint, so that the other pages could be created and the dashboard requirements could be clarified.

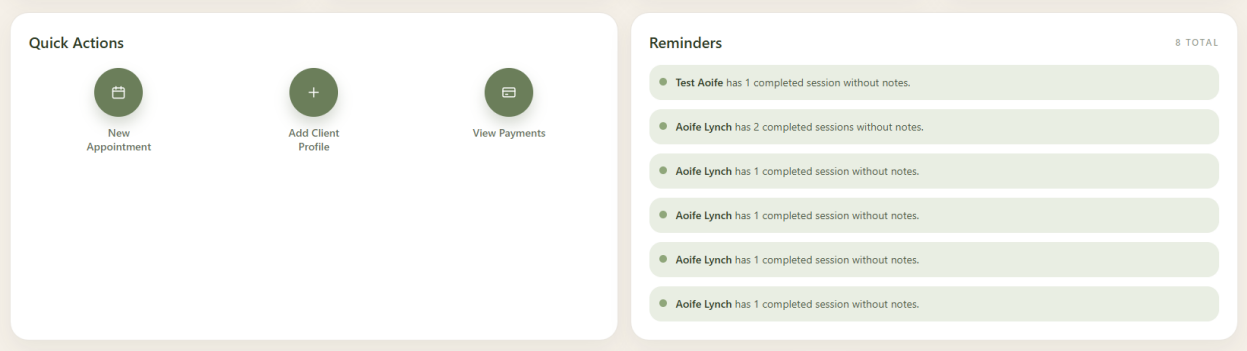

Figure 36 - Example of card height expansion on Dashboard page

The calendar was implemented using the full-calendar package, which allowed it to be viewed monthly, weekly, or by day. Key features of the calendar include colour-coded appointment components to differentiate in-person from online sessions, a filter feature, and a create appointment modal that opens on the calendar page, reducing the need for navigation. The final calendar page design is shown in Figure 37.

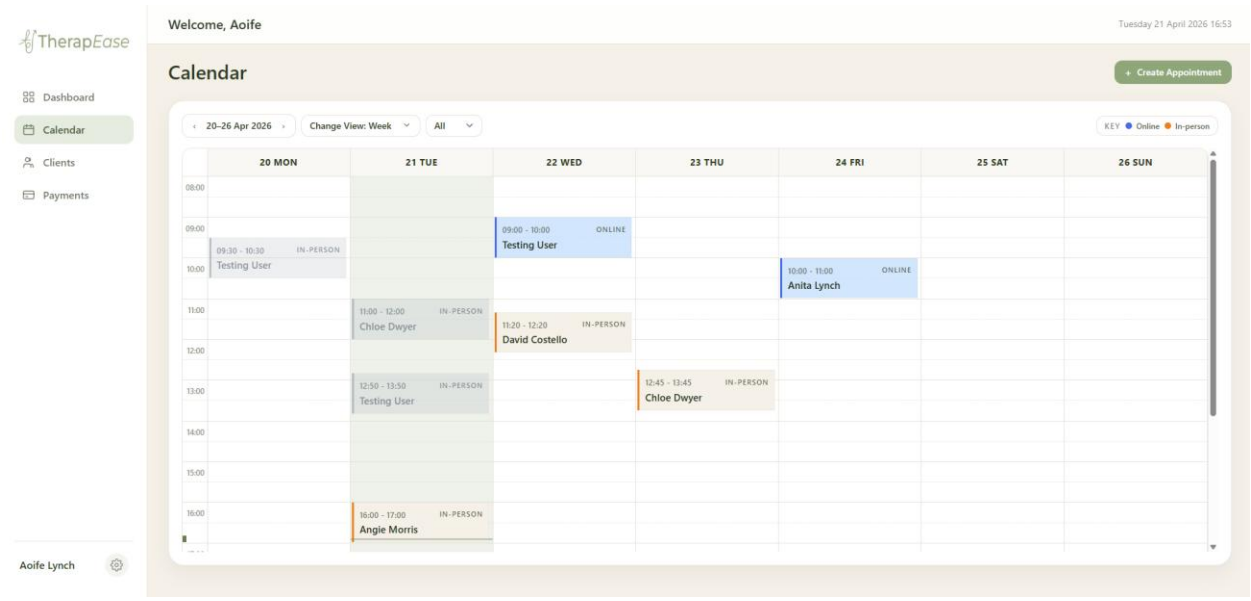

Figure 37 - Calendar Page

The client list page was next to be implemented, following a design similar to the prototype, with minor changes to the layout (Figure 39). In the client list table, the “Actions Needed” column is linked to the reminders, displaying if the client needs session notes or profile details added. The actions needed column originally listed all actions that expanded the line height, making the table layout uneven. To prevent this, a condition was created that adds a (+n more) label (Figure 38). The remaining actions can be hovered over to reveal them, but this change cleans up the page's style while remaining functional.

```
const actionsNeededDetails = actionsNeeded.join(' • ');
const actionsNeededLabel = actionsNeeded.length === 0
    ? 'None'
    : actionsNeeded.length === 1
      ? actionsNeeded[0]
      : `${actionsNeeded[0]} +${actionsNeeded.length - 1} more`;
```

Figure 38 - Actions Needed "+ more" label

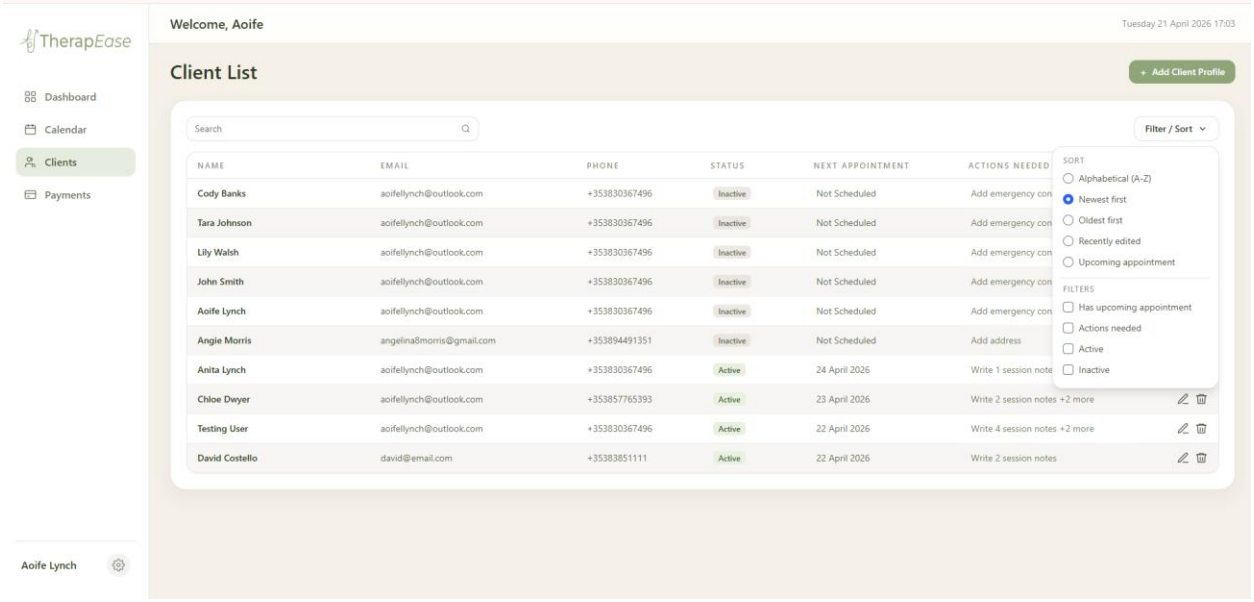

Figure 39 - Client List Page with Filter/Sort button

5.6.3 Goal 3 – SMS Message Queue (Asynchronous Processing)

Following the creation of the email confirmation queue, a new queue was required to send SMS reminders for upcoming appointments. Instead of Nodemailer, this was implemented using Twilio.

While integrating Twilio initially seemed straightforward, issues arose when attempting to send messages with the free trial account. Twilio requires recipient phone numbers to be verified when using a trial account. Despite verifying the test phone number, messages consistently failed to send, and the cause of the issue remained unclear. After some debugging and research, a normalisation function was introduced to automatically format Irish mobile numbers by adding the +353 prefix where required (Figure 40), as Twilio requires numbers to include an international country code. This change resolved the initial error but resulted in a different error response from the API. Significant time was spent trying to fix this issue, including reviewing API credentials, request structure and Twilio configuration, but the problem could not be fully resolved in the free trial environment. Upgrading to a paid Twilio plan resolved the issue immediately and allowed messages to be sent without the verification restrictions.

A new queue for reminders was then created using the same Redis connection as the email confirmation system. As SMS messages act as reminders, they were scheduled using node-cron (Merencia, 2025). This scheduler runs daily at 9:00 and queries the database for appointments scheduled for the following day with the reminderSent field set to false, preventing duplicate messages. Appointments that fit those parameters are added as jobs to the reminder queue and processed by a worker. The worker retrieves the appointment and associated client, constructs the SMS message and sends it via Twilio using the sendSMS function. This process is shown in Figure 41. Once the message is successfully sent, the reminderSent field is updated to true. The phone number normalisation was kept as part of the final implementation to ensure all numbers are stored and processed in a valid format for Twilio, reducing the likelihood of formatting-related errors in future use.

```
const normaliseIrishPhoneNumber = (value) => {
  if (typeof value !== 'string') return value;

  const compact = value.trim().replace(/[\s()-]/g, '');
  if (!compact) return compact;

  if (/^08\d+$/u.test(compact)) {
    return `+353${compact.slice(1)}`;
  }

  if (/^3538\d+$/u.test(compact)) {
    return `+${compact}`;
  }

  if (/^003538\d+$/u.test(compact)) {
    return `+${compact.slice(2)}`;
  }

  return compact;
};
```

Figure 40 - Phone number normalisation in the validator file

```
cron.schedule("0 9 * * *", async () => {
    try {
        console.log("Running reminder scheduler...");

        const tomorrowStart = new Date();
        tomorrowStart.setDate(tomorrowStart.getDate() + 1);
        tomorrowStart.setHours(0, 0, 0, 0);

        const tomorrowEnd = new Date(tomorrowStart);
        tomorrowEnd.setHours(23, 59, 59, 999);

        const appointments = await Appointment.find({
            date: {
                $gte: tomorrowStart,
                $lte: tomorrowEnd
            },
            status: "upcoming",
            reminderSent: false
        }).populate("client");

        for (const appt of appointments) {

            const job = await reminderQueue.add("send-reminder", {
                appointmentId: appt._id,
                phone: appt.client.phone,
                clientName: appt.client.firstName,
                appointmentTime: appt.startTime
            });

            await Appointment.findByIdAndUpdate(appt._id, {
                reminderJobId: job.id
            });
        }

        console.log(`${appointments.length} reminder jobs queued`);
    } catch (error) {
        console.error("Error running reminder scheduler:", error);
    }
});
```

Figure 41 - Cron SMS reminder scheduler

5.6.4 Goal 4 – Stripe Setup

Stripe was implemented to provide secure payment processing and automatic receipt generation. This was set up in a sandbox environment so all parts of the integration could be tested safely. As the application was running locally, the Stripe Command Line Interface (CLI) was installed, as Stripe webhooks normally require a deployed application. The CLI was then configured to listen to a local port, which generated a signing secret that was added to the environment file along with the API keys.

After this initial setup, the Stripe Software Development Kit (SDK) was installed to enable the backend to create checkout sessions and verify incoming webhook requests. The overall integration quickly became more complex than expected as managing multiple components such as checkout sessions, webhooks, database updates and linking payments to appointments was difficult to undertake in a single sprint. At this point, progress on the Stripe integration slowed and became inconsistent, as changes in one area often caused issues in another.

Towards the end of the Sprint, a more structured approach was taken to try to achieve as much of the implementation as possible. A Stripe service file was created to handle the core logic, including generating checkout sessions with key details such as client email, appointment ID, client ID, user ID and payment amount (Figure 42). The Payment model was updated to include fields for the Stripe session ID and payment intent ID, enabling payment tracking and preventing duplicate transactions. A dedicated API route and controller were then created to handle session creation and store a pending payment record before payment was completed. Finally, a webhook endpoint was implemented to act as the connection between Stripe and the application. This endpoint listens for events such as successful or failed payments, verifies the webhook signature and updates the corresponding payment record in the database. This ensured that payment status was handled securely and did not rely solely on frontend confirmation.

The further integration of Stripe was paused at this stage as full testing required a completed frontend payment flow and additional updates to the SMS notification system. This iterative approach allowed the feature to be developed in stages without blocking overall progress.

```
const session = await stripe.checkout.sessions.create({
  payment_method_types: ["card"],

  mode: "payment",

  customer_email: clientEmail,

  line_items: [
    {
      price_data: {
        currency: "eur",
        product_data: {
          name: "Therapy Session"
        },
        unit_amount: unitAmount
      },
      quantity: 1
    }
  ],

  success_url: `${process.env.FRONTEND_URL}/payment-success?session_id={CHECKOUT_SESSION_ID}`,

  cancel_url: `${process.env.FRONTEND_URL}/payment-cancelled`,
```

Figure 42 – Stripe checkout session creation in the Stripe Service file

5.6.5 Sprint Outcome

This sprint integrated the Zoom API for automatic meeting generation, implemented key frontend pages such as the calendar and client list and added SMS reminder functionality using Twilio and background processing. The Stripe integration was only partially completed during this sprint. While the backend setup, including webhook handling and payment models, was implemented, the full payment flow could not be tested due to missing frontend components.

A key challenge during this sprint was frontend design, particularly with the dashboard layout. The initial card-based design caused layout issues with dynamic content, resulting in an unbalanced interface. As a result, the dashboard design was paused for later review. Another challenge was the difficulty of integrating Stripe due to the configuration's complexity. Due to these challenges, the dashboard redesign and full Stripe implementation were moved to the backlog for later sprints, along with additional testing and refinement tasks for SMS and external API integrations.

5.7 Sprint 5

With the core frontend pages created and external services implemented, this sprint focused on expanding the user interface's functionality. The goals were to improve the consistency of the user interface by creating reusable components and themes, and to build the client profile page with functionality for creating session notes and uploading files. Following the earlier Stripe setup, the payments page was needed to complete the payment workflow.

5.7.1 Goal 1 – Components, Modals & Themes

To reduce repetitive code and improve consistency across the application, three main changes were made to the frontend: the creation of reusable components, the use of modals for user interactions and the implementation of a global Tailwind CSS theme. Initially, modals were coded directly into each page, but they were later refactored into reusable components to improve consistency and reduce repeated code. The create and edit modals for clients are shown in Figure 43. Additional reusable components were created for elements such as the navigation sidebar, logo, tables, toast notifications and confirmation modals for deleting or discarding changes. Each of these was stored in a components folder and then imported where needed. However, when transitioning tables from hard-coded implementations to reusable components, formatting issues arose due to differences in

data structure and layout requirements across pages. This required additional props and conditional rendering to ensure the tables remained flexible while maintaining a consistent design. A global Tailwind CSS theme was also implemented to set the chosen brand colours, text styles and button designs for TherapEase, ensuring consistency throughout the interface.

Client Modals - Create & Edit

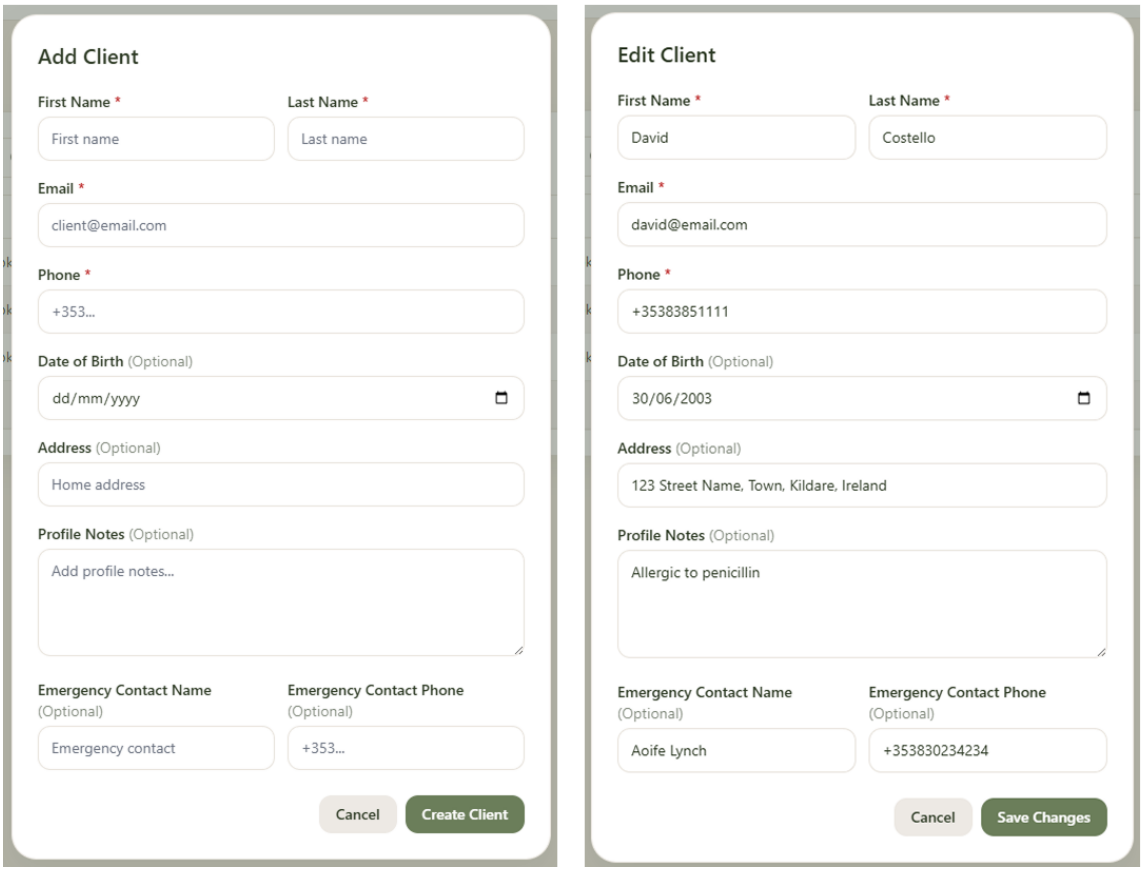

Figure 43 - Client Create & Edit modals

5.7.2 Goal 2 – Client Profile Page

The client profile page needed to display client details, interaction history, session notes, file uploads, appointment history and payment history. A note editor was created and controlled within the page to handle creating, editing and viewing notes (Figure 44). Two options were added for creating

session notes: the first is a completely custom note, which appears when a user clicks “New Session Note” and the second is session note templates. Common note templates, including SOAP, DAP, BIRP, and PIP, were included to accommodate different therapists' preferences. Session notes have to be linked to an existing appointment, so a dropdown was added with the client’s appointments that do not yet have a session note. An autosave feature was added for notes, which detects changes and saves them after a short delay to prevent data loss. After some testing, additional controls were added to allow users to cancel or discard changes and to display the saved notes in a modal.

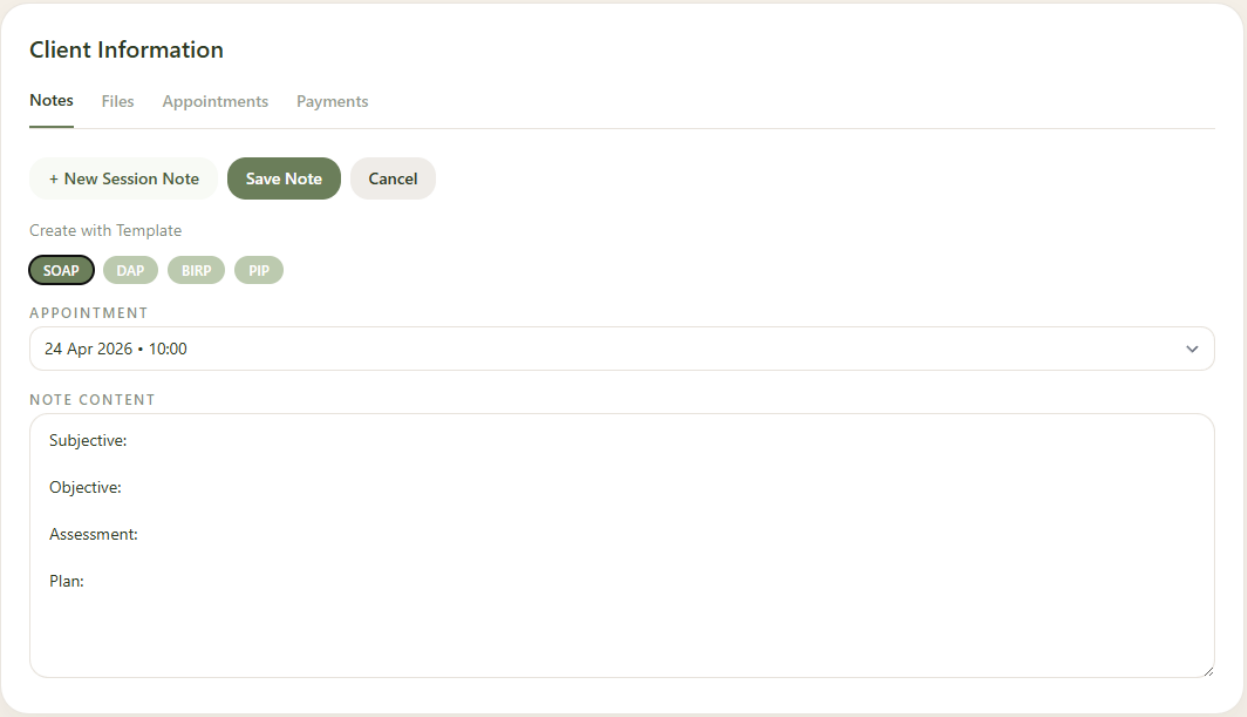

Figure 44 - Note editor on Client Profile page

As therapists often need to store additional documents for a client, such as service agreements or intake forms, a file upload feature was added to the client profile page (Figure 45). The selected file was processed on the frontend, assigned a temporary browser URL, and sent to the backend via filesAPI.upload. The stored record included metadata such as the client reference, file name, file type and file URL. Once uploaded, the file list refreshed immediately, and a confirmation message was displayed.

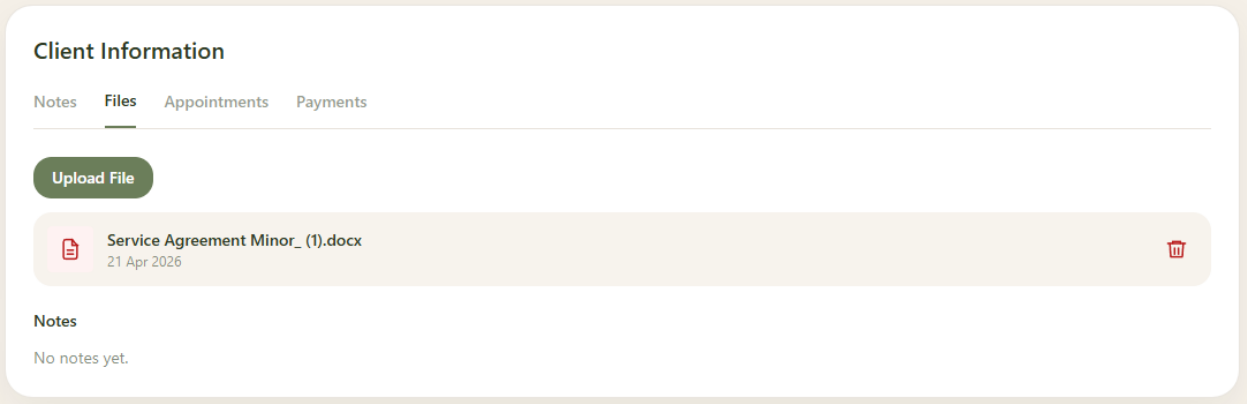

Figure 45 - File upload on Client Profile page

5.7.3 Goal 3 – Stripe Integration

The completion of Stripe integration was required to generate payment links and receipts. To achieve this, the backend payment logic was expanded to allow payment links to be created and linked to the correct therapist, client, and appointment. The payment service was responsible for checking that the appointment and client belonged to the logged-in user, normalising the amount and preventing duplicate links from being created for the same appointment. These checks were added to prevent incorrect payments, duplicate links, and unnecessary Stripe sessions. The webhook logic was then extended so that when a payment was completed, Stripe could notify the backend, the payment record table, shown in Figure 46, could be updated and the Stripe receipt URL could be stored. A new email worker was added to send payment receipts once the webhook confirmed the payment was successful.

Once the backend payment flow was complete, the frontend was updated to allow therapists to use payment links. A payments page was created to display payment records, statuses, and receipt links in one place, making it easier to track client payments without checking Stripe directly. Payment result pages were also added for successful and cancelled payments so that the client would be given clear feedback after checkout.

On the Calendar page, payment options were added to the create and edit appointment forms, including a quoted amount and payment timing settings. This was done so payment links could be connected directly to an appointment rather than being created separately later. The appointment collection was also extended to include fields such as paymentLinkTiming, autoSendPaymentLink, and quotedAmount, allowing the system to decide whether a payment link should be sent before or after a session. This was then connected to the SMS reminder process, where before-session payment links could be added to reminder messages, and after-session links could be triggered automatically when an appointment status changed to completed. Finally, a “Payments” tab was added to the client profile page so payment history, status and receipt links could also be viewed within the client record.

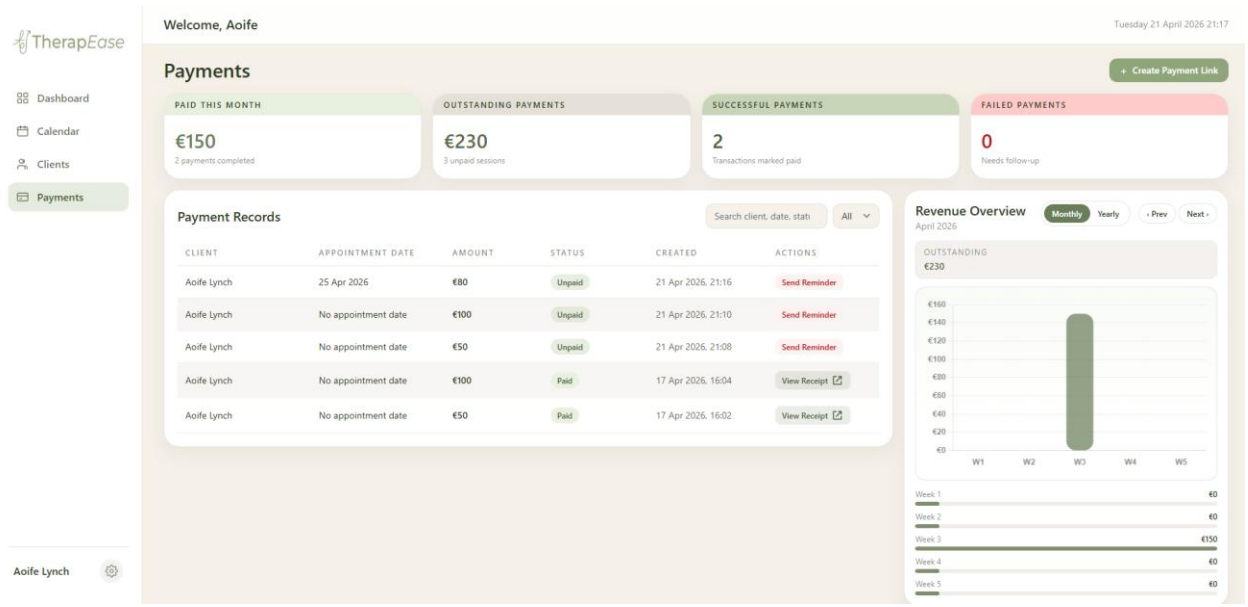

Figure 46 - Payments Page showing payment record table and graph created with Chart.js (chart.js, 2019)

5.7.4 Sprint Outcome

This sprint improved the consistency and usability of the frontend by introducing reusable components, modals, and a global theme. The client profile page was also fully implemented, including session note creation, note templates, autosave functionality and file uploads. In addition, the core payment workflow was completed, with Stripe fully integrated across both the backend and frontend, enabling payment links, receipts, and payment tracking.

A key challenge during this sprint was transitioning frontend elements, particularly tables and modals, into reusable components. Differences in data structure across pages caused formatting and rendering issues, requiring the introduction of conditional logic and additional props to maintain flexibility without breaking layout consistency. Another challenge emerged during the implementation of the payments page. While payment links could be generated and tracked, the presence of pending payments revealed a usability issue: there was no direct way for therapists to follow up with clients who had not yet paid. As a result, a “Send Reminder” button for pending payment links was identified as necessary but not implemented in this sprint. This was added to the project backlog.

5.8 Sprint 6

With most core functionality implemented, the final sprint focused on refinement after user testing, redesigning the dashboard, completing the payment workflow and deploying the application on Render.

5.8.1 Goal 1 – Improving User Feedback

Following the first round of user testing, one significant user feedback issue was the lack of notification when session notes are saved. Notes are stored under the Files tab, but the user was not clearly informed of this. To address this, a pop-up was implemented to notify the user that their note has been saved and to redirect them to the files tab. This implementation added a new condition to the

Client Profile page to show the pop-up when a note was saved successfully and to remove it when a new note was started, to prevent confusion. The pop-up could then be dismissed or used to redirect the user to the Files tab.

To address inconsistencies in error handling, a reusable error component was created to ensure all errors display real user feedback rather than HTTP error codes. A cleanErrorMessage function that strips the HTTP codes and labels was created in a utility file on the backend. This function was then imported into all files, where the error message could be edited to fit each page to create a more user-friendly error message and improve usability

When a user created a payment link on the payments page, the new row on the payment record table would not show until the page had refreshed. Similarly, when a client paid through the payment link, it wouldn't update the table, revenue graph or feature cards until the page was refreshed. To fix this, an auto-refresh feature was implemented by creating a refresh function that fetches the latest payments and clients (Figure 48). This function is called after creating a new payment link so the table will update to show the new information. On the frontend, a function with a useEffect hook requests fresh data from the server every fifteen seconds (Figure 47). If any new data is found, the entire page reloads to show this. It also checks to see if the page is visible, if the user is on a different browser window and then returns to the page, it automatically updates.

```
useEffect(() => {
    const pollInterval = window.setInterval(() => {
        if (document.visibilityState === 'visible') {
            refreshPayments();
        }
    }, 15000);

    const handleFocus = () => {
        refreshPayments();
    };

    window.addEventListener('focus', handleFocus);
    document.addEventListener('visibilitychange', handleFocus);

    return () => {
        window.clearInterval(pollInterval);
        window.removeEventListener('focus', handleFocus);
        document.removeEventListener('visibilitychange', handleFocus);
    };
}, [refreshPayments]);
```

Figure 47 - Refresh Page Function

```
const refreshPayments = useCallback(async () => {
    try {
        const [paymentsResponse, clientsResponse] = await Promise.all([
            paymentsAPI.getAll(),
            clientsAPI.getAll(),
        ]);
        setPayments(paymentsResponse.data || []);
        setClients(clientsResponse.data || []);
    } catch (requestError) {
        setError(requestError.response?.data?.message || requestError.message ||
        'Unable to load payments');
    }
}, []);
```

Figure 48 - Refresh Payments Table Function

The final issue addressed for the goal of improving user experience and feedback was the issue of incorrect email validation, where dots were being removed from the client's email addresses. This happened as the default behaviour of the normalizeEmail function in express-validator would remove

the dots, as Gmail treats email addresses with and without dots as the same email. However, this is not consistent across other email providers and created confusion during user testing. To resolve this, the validation logic was updated so that email normalisation no longer removes dots (Figure 49). This ensured that email addresses input would be stored exactly as the user entered them.

```
email: {
  in: ['body'],
  notEmpty: { errorMessage: "'email' field is required" },
  isEmail: { errorMessage: "'email' must be a valid email address" }
  normalizeEmail: {
    options: {
      gmail_remove_dots: false,
    }
  }
},
```

Figure 49 - Remove Gmail Dots in Validator

5.8.2 Goal 2 – Dashboard Revisions

As the final major page left to design, the dashboard was completed in this sprint as a backlog item carried over from Sprint 4. Originally, the “Quick Actions” buttons took up a significant amount of space. These were moved to the top header, where they open modals on the calendar, client and payments pages. The number of cards on the page was reduced from six to four to display the key information better. The reminders section was built to show tasks the user still needs to finish, such as writing session notes or filling in missing client profile details. An issue occurred in which the total reminder count remained the same after tasks were completed. To fix this, a function was created to build the reminders from a set of rules, and another function was added to check the latest client and appointment data again. If a task has already been completed, it is removed from the reminders list, so the total number updates correctly.

To combat the issue of card heights expanding, cards now have a fixed height and become scrollable when the information exceeds the card height. The upcoming appointments and revenue overview cards also have “View More” buttons that redirect them to the relevant page. The completed page design of the dashboard is shown in Figure 50.

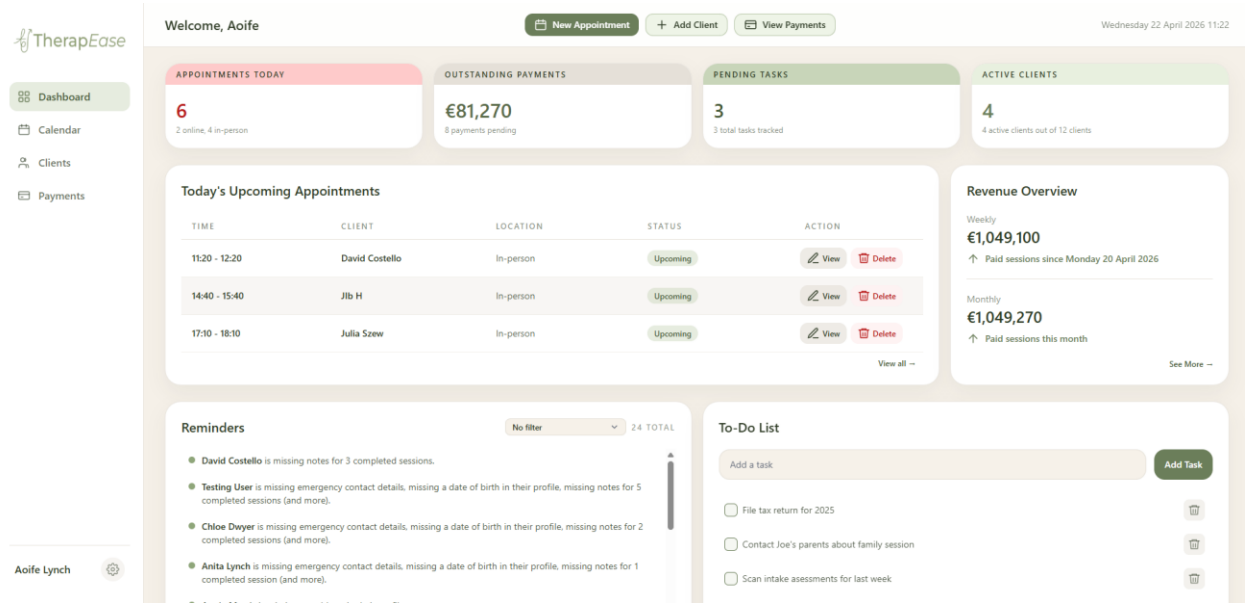

Figure 50 - Updated Dashboard Page Layout

5.8.3 Goal 3 – Payment Workflow Enhancements

To accommodate all types of workflows, a “Send Now” button was added as an option when sending a payment link through appointment creation. This was first added as an option in the modal and then reused the same payment link setup as the Payments page. On the backend, the system checks the payment link timing and if the user picked “now”, it creates the Stripe link and sends the SMS message immediately. The “Send Reminder” button on the payment record table works the same way in reverse. The page sends the payment ID to the backend, the backend verifies that the payment has not been paid, it tries to reuse the same link or creates a new one if the original is invalid, then sends the

reminder text (Figure 51). This improved flexibility in the payment workflow and reduced the need for therapists to manually follow up on unpaid sessions.

```
let paymentLink = null;

if (payment.stripeSessionId) {
  const checkoutSession = await getCheckoutSessionById(payment.stripeSessionId);
  paymentLink = checkoutSession?.url || null;
}

if (!paymentLink) {
  const refreshedSession = await createCheckoutSession({
    clientEmail,
    amount: payment.amount,
    appointmentId,
    clientId,
    therapistId,
  });

  payment.stripeSessionId = refreshedSession.id;
  await payment.save();

  paymentLink = refreshedSession.url;
}

if (!paymentLink) {
  throw new HttpError(BAD_REQUEST, 'Unable to resolve the payment link');
}

const appointmentDate = payment.appointment?.date || payment.createdAt;
const formattedDate = formatReminderDate(appointmentDate);
const linkText = `Reminder that payment is due for your therapy session ${paymentLink}`.trim();

await sendSMS({
  to: payment.client.phone,
  message: linkText,
});

return {
  sent: true,
  paymentId: payment._id,
};
```

Figure 51 – Code snippet of the Send Reminder function

5.8.4 Goal 4 – Deployment & Hosting

TherapEase was deployed using Render, which involved splitting the application into four separate services, as seen in Figure 52. The backend was configured as a Web Service, the asynchronous processing was set up as a Background Worker, a Redis instance was created and the frontend was deployed as a Static Site. MongoDB Atlas was used for the database, which required network access configuration to allow the database to be used by Render. Core deployment was successful, but several issues arose after testing the deployment. Stripe integration required a complete reconfiguration as it needed a live webhook endpoint, new environment variables and the correct handling of raw requests for webhook signature verification. This also improved the deployment setup, as Stripe's development tools could now be used instead of the Stripe CLI, providing more detailed error handling and reporting. Following the new Stripe setup, the payment links stopped working and would only trigger with the "Send Reminder" button. This was traced back to a missing import on the payments controller, which prevented the sending of the SMS messages in the calendar and payments pages. On the frontend, an issue occurred when the page was refreshed and gave an error saying "Failed to load resource". This happened because React Router handled navigation client-side, while Render attempted to resolve each route as a static file. A rewrite rule was added so that all pages are routed through the React app instead, which fixed the issue. After implementing these fixes, the deployed application had full end-to-end functionality.

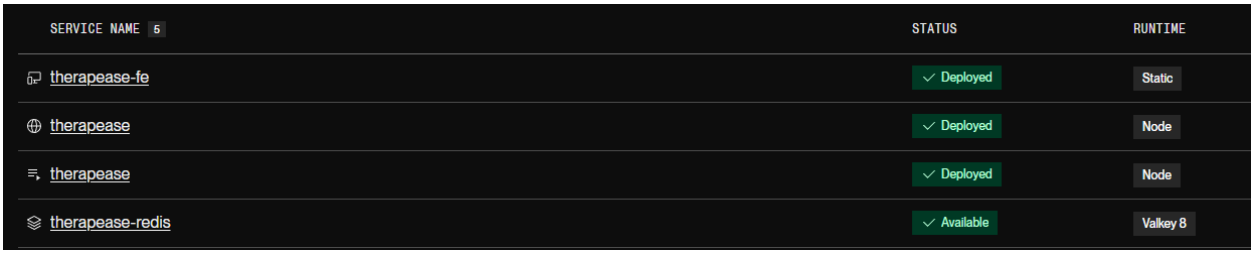

Figure 52 - Services deployed on Render

5.8.4 Sprint Outcome

This sprint refined the overall user experience following user testing and completed the final major features of the application. Improvements to user feedback, including clearer error messages, confirmation pop-ups and automatic data refresh, significantly improved usability. The dashboard was fully redesigned and implemented, resolving earlier design limitations. The payments workflow was also completed, with full functionality for sending payment links and reminders directly from the interface. Finally, the application was successfully deployed on Render, achieving full end-to-end functionality in a live environment.

A key challenge during this sprint was addressing issues uncovered during deployment. Stripe integration required reconfiguration for a live environment, which caused payment links to fail. This was later traced to a missing import in the payments controller that prevented SMS messages from being sent correctly. Another issue arose with frontend routing on Render, where page refreshes resulted in “Failed to load resource” errors due to the mismatch between React Router and server-side routing. This required the implementation of rewrite rules to ensure all routes were correctly handled by the frontend application. Additionally, refining the dashboard required reworking earlier design decisions, particularly around dynamic data such as reminders, which initially did not update correctly after tasks were completed.

As this was the final sprint, all backlog items were completed, including the dashboard redesign and payment workflow enhancements that had been carried forward from previous sprints.

6. Testing & Evaluation

Two types of testing were completed to evaluate the functionality and usability of the application. Technical testing was conducted to ensure that system components operated correctly, while usability testing was carried out with real users to assess workflow efficiency and ease of use.

The results from both testing approaches are used to evaluate whether TherapEase successfully addresses the research question outlined in Chapter Two, specifically whether an integrated, user-centred web application can streamline administrative workflows.

6.1 Testing Strategy

The testing strategy was divided into two main areas: technical testing and usability testing. This approach ensured that the system was both functionally reliable and effective for its intended users.

Technical testing focused on validating the functionality of individual components and full workflows. Unit testing was carried out using Jest to test backend functionality, while end-to-end testing using Playwright ensured that the frontend and backend communicated correctly and would operate as expected across different browsers.

Usability testing focused on evaluating the user experience of the system. Task-based testing was conducted where participants completed realistic administrative tasks using both their existing workflow and TherapEase. Metrics collected included time on task, number of errors and System Usability Scale (SUS) scores. This allowed for both quantitative and qualitative results to evaluate the system's effectiveness.

6.2 Technical Testing

Jest and Playwright were used to carry out tests on the functionality of the system. Both integration and end-to-end testing were completed to ensure the application functioned as intended before user testing was conducted.

6.2.1 Unit & Integration Testing

Jest was used to individually test functions and logic on the frontend. 7 test suites, containing 24 tests, were created and executed. The Jest tests focused on smaller functional units that are essential to the stability of the application. These included:

- Formatting helpers for currency, dates, times, colour values and client names
- Client profile functions such as filtering notes by appointment ID and formatting profile-related data
- API behaviour, including attaching tokens, handling success responses and refreshing tokens
- Error message handling to ensure technical errors are transformed into clearer user-facing messages
- Toast notification behaviour to ensure valid feedback messages are shown correctly

The full test run completed in 0.674 seconds and all 24 tests passed successfully.

6.2.2 End-to-End Testing

Playwright was used to complete user workflows across the application. These tests simulated user interactions and verified that the backend and frontend worked together correctly. Playwright tests are done across three browsers: Chromium, Firefox and WebKit. The tested workflows included:

redirecting unauthenticated users away from protected routes

- Validating the login form when submitted empty
- Registering from the landing page and reaching the dashboard
- Logging in using two-factor authentication
- Forcing first-time 2FA setup and completing it successfully

- Creating, editing and deleting clients
- Creating appointments from the calendar
- Displaying appointment conflict errors without closing the modal
- Dismissing unsaved calendar changes using the discard modal
- Editing and deleting appointments
- Creating and sending payment links
- Sending payment reminders for pending payments
- Loading client profile details
- Creating and deleting client notes
- Updating profile settings and logging out
- Deleting the account and confirming logout

These tests were performed successfully across all browsers, which indicated that all features in the application worked correctly from the frontend to the backend.

6.2.3 Technical Evaluation

Overall, the technical testing done on TherapEase shows that it is functional across both the frontend and backend, including all core features. The Jest results confirmed that the smaller pieces of logic were being used correctly, while the Playwright results showed that complete workflows could be completed across multiple browsers.

These findings suggest that the application was reliable enough to support the usability testing that followed and confirmed that TherapEase met the technical objective of functioning as an integrated platform.

6.3 Usability Testing

Usability testing was conducted with three participants who regularly perform therapy administration tasks. The testing was done within-subjects, meaning that each participant tested the

entire interface. The survey completed before development showed the three most common tasks were performed by almost all therapists and these three tasks were then used for testing to compare TherapEase with their existing workflow:

- Creating a new client profile
- Schedule an online appointment and create a payment link
- Write and save session notes

The participants' existing workflow involved a combination of tools such as Outlook, Word, SharePoint, Zoom and SumUp. They then repeated the same tasks using TherapEase. The primary result of this testing is the time-on-task comparison between the original workflow and TherapEase, which will determine whether the application improves efficiency. While performing the tasks, the participants' mistakes were also noted to see whether any improvements could be made.

6.3.1 Time on Task

The time participants took to complete each task was recorded and compared across both workflows.

Task 1: Creating a client profile

For this task, participants were asked to create a profile that included the client's name, email address and phone number. For the original workflow, Outlook was used for its contact feature which allowed for the input of these details. The steps for creating a new client in TherapEase were very similar, with the form allowing for name, email address, phone number, along with other client details such as date of birth, emergency contact and address.

Participant	Original Workflow	TherapEase
A	40 seconds	1 minute 5 seconds
B	49 seconds	23 seconds
C	32 seconds	20 seconds

For participants B and C, TherapEase reduced task time. Participant A's time increased due to TherapEase having more fields for client information, which the original workflow lacked. Although the input time increased, it results in more complete and organised client records. With Outlook's contact feature, there are many fields available that are irrelevant when creating a client profile and therefore took slightly more time to navigate.

Task 2: Creating an online appointment with a payment link

To create an online appointment with a payment link in the original workflow, Outlook was used for the calendar feature, Zoom was used to create the link for the online call and SumUp was used to create the payment link. With TherapEase, all these features were connected to a single modal when creating the appointment, with the user needing to use a drop-down to select an online appointment and opt in to have the payment link sent to the client.

Participant	Original Workflow	TherapEase
A	1 minute 55 seconds	32 seconds
B	2 minutes 29 seconds	25 seconds
C	2 minutes 40 seconds	32 seconds

Significant time reductions were observed across all participants when using TherapEase to schedule an appointment compared with the original workflow. This is due to the original workflow requiring switching between three different systems to create the appointment, then schedule an online call and finally create a payment link to send to the client. Whereas TherapEase completes all of this in a single flow.

Task 3: Writing and saving session notes

To write notes in the original workflow, the participants used Microsoft Word and were asked to include the session date, time, client name and subheadings for the session note layout. To save the session notes, they were asked to create a folder on SharePoint for the client and then save the file inside that folder. When using TherapEase, participants were asked to create a note on the client profile page, attach the previously created appointment to it, select a note template and save it.

Participant	**Original Workflow**	**TherapEase**
A	1 minute 46 seconds	28 seconds
B	1 minute 40 seconds	14 seconds
C	1 minute 33 seconds	10 seconds

This task showed the greatest improvement in workflow efficiency when participants were using TherapEase. As the client notes are written and saved directly on the client profile, with pre-made templates available, the users don't need to use any external tools or file storage.

On average, the original workflow took 4 minutes 37 seconds, whereas TherapEase reduced this to 1 minute 23 seconds, representing a 70% reduction in task completion time.

6.3.2 Errors Made

During user testing, the main error seen from participants was trying to schedule an appointment directly in the client profile page. When asked to create an appointment for the client they just made, they navigated to the "Appointments" tab on the profile page, which was just a display of previous appointments. This was seen by 2 separate participants and therefore, the functionality of a "Create Appointment" button and modal directly in the client profile page was added after. One other minor error was observed during the time-on-task testing with participant A, who clicked the "Edit Client" button while trying to navigate to the client profile page from the client list page. No other major errors or task failures were found during the user testing stages of TherapEase.

6.3.3 System Usability Scale

To determine the level of usability of TherapEase, a System Usability Scale (SUS) questionnaire was completed by participants after testing the application. The SUS is a popular ten-item scale developed by Brooke (1996) that produces a score out of 100. Scores above 85 are considered excellent. The respondents use a 5-point scale (1=Strongly Disagree, 5=Strongly Agree) to answer the following questions:

1. I think that I would like to use this system frequently.
2. I found the system unnecessarily complex.
3. I thought the system was easy to use.
4. I think that I would need the support of a technical person to be able to use this system.
5. I found the various functions in this system were well integrated.
6. I thought there was too much inconsistency in this system.
7. I would imagine that most people would learn to use this system very quickly.
8. I found the system very cumbersome to use.
9. I felt very confident using the system.

10. I needed to learn a lot of things before I could get going with this system.

The results were high across all participants, with the positive statements about ease of use and confidence in the system scoring a consistent 4 and 5, while the negative statements of complexity and inconsistency scored a 1 or 2. The full results from all participants are in Appendix G. TherapEase scored a 97.5 on the System Usability Scale.

One participant who gave question 2 a score of 2 stated that the system contains some complexities, but it was easy to figure out on their own.

6.3.4 User Feedback

Participants were asked to think aloud during the duration of the user testing. Requests made by the participants included the want for a "Create Appointment" button on the client profile page, a function to open a full calendar to select a different week instead of having to use arrow buttons and the addition of the Zoom link to the appointment details modal.

Alongside helpful feedback on their thought process when using the application, highly positive feedback about the functionality and design was also provided by the participants. Participants stated "That's it done? That's so much less work to do", "It's so handy that everything is there in one place", "It's all so easy, it doesn't give me anxiety to use" and "That is amazing, that is exactly what's needed". The feedback shows not only improvements in efficiency but also an overall more positive user experience.

6.4 Evaluation Against Objectives

The project's objectives were to streamline administrative workflows for therapists, improve efficiency and improve user experience.

TherapEase successfully reduced administrative workload by integrating multiple systems into one platform. Tasks that previously required several tools were completed within a single interface, eliminating duplication and unnecessary steps. The complete workflow of creating a client, scheduling an appointment and writing session notes was reduced by 70% when using TherapEase compared to the original fragmented workflow.

The application significantly improved efficiency across the most common workflows. Appointment creation and note-taking tasks showed 80-90% reductions in completion time.

High SUS scores and positive user feedback confirm that the system is easy to use, intuitive and creates an overall positive user experience.

6.5 Limitations

While the results of usability testing were highly positive, some limitations should be considered. The testing had a small sample size of only three participants. While this was sufficient for identifying major usability issues, a larger sample size would provide more accurate results for time-on-task testing. All participants used very similar workflow tools originally, which made the time-on-task results very similar due to the common flow. A more diverse range of tools would need to be compared to TherapEase to more accurately compare the difference in workflows. When performing time-on-task testing, all three participants completed the tasks in the same order: first the original workflow, then TherapEase. As there was no randomisation of the sequence, the results could potentially be skewed in favour of TherapEase. Additionally, all participants were personally known to the author, which may have introduced bias when completing the SUS questionnaire.

Additionally, all users were tested on the same device, which may not be reflective of the time it takes to complete tasks in a different environment due to differences in device performance and speed.

7. Project Management

Project management tools and methodologies were used throughout all project phases of TherapEase. The six phases of development include proposal, research, requirements, design, implementation and testing. The SCRUM methodology was implemented during the implementation phase to split the workload into sprints. Multiple tools were used to manage the project, including Trello to create a kanban to-do list, Miro to create diagrams and plan the different phases of development, GitHub for version control of the application code and Notion for planning user testing scripts and creating daily to-do lists and timelines. This chapter also presents a reflection on the positive and negative aspects of the project, the technical skills gained and the experience of working with a supervisor.

7.1 Project Phases

This project was completed in six key phases, each one built on the previous to support the development of TherapEase.

7.1.1 Proposal

The initial project proposal was where the first idea for TherapEase was created with the sole purpose of implementing as many administrative tasks into one platform as possible. The scope of the project, initial features and objectives were outlined in this phase. Conversations with the project supervisor helped to try to narrow down these features and explore how these features would actually be implemented, more specifically, the email and SMS messages being sent. Supervisor feedback was also important in ensuring the project was achievable in the given time frame.

7.1.2 Research

The research phase was focused on identifying the key problems with existing therapists' workflows and completing an analysis of competitor applications. Academic research was done to highlight the importance of usability and user-centred design. Technical research was done to figure out how to best implement authentication to abide by GDPR, as well as research on asynchronous processing to determine what technologies would be best suited for the application.

7.1.3 Requirements

After creating a survey for therapists and examining the responses, a persona was created to show the goals and pain points of the therapists. The functional and non-functional requirements were then created from these personas to determine what core features were needed and which were not essential. Use cases were then created to show the workflow the users would have. This phase was critical in determining the pain points of therapists and turning them into features in an application.

7.1.4 Design

The design phase involved decisions on system architecture, database structure and user interface design. After completing research on Redis, BullMQ and 2FA integrations, the system architecture diagram was created to reflect how the system would work. Followed by prototypes and user flows to show how the user interface would look and how users would interact with the system. Prototype design was refined after consulting with the second reader to make tables clearer, add more line space and ensure the colours chosen had a readable contrast.

7.1.5 Implementation

An interactive approach was used when building the application to divide the development into sprints. These sprints allowed for the features to be implemented gradually, as well as providing a clear

timeline for development. The development started with core features being implemented first, such as the ability to create appointments, and then expanded to implement external APIs, payment processing and background job queues. Many challenges were faced throughout this phase and forced some features to be added to a backlog to be completed in later sprints, such as dashboard design and Stripe implementation. However, all features outlined in the functional requirements list and MVP were successfully implemented by the final sprint. Weekly meetings were held with the project supervisor to ensure the project wasn't falling behind and to discuss challenges that arose. These conversations were essential to get another opinion on the implementation of features and to decide when to add features to the backlog.

7.1.6 Testing

Although there was continuous testing throughout the project development, with Postman to test the API and user testing throughout frontend development, additional testing was conducted as the final phase of project development. Functionality and usability testing were completed to ensure the project aligned with therapists' workflows, was easy to use and functioned as intended. Technical testing was achieved with Playwright and Jest, so both complete workflows and smaller components were functional across all web browsers. Usability testing was done with real users to compare TherapEase to their original workflow. This allowed for a time-on-task analysis, System Usability Scale score and to assess any errors made. The results of these tests were positive, with the end-to-end testing and unit testing passing all tests, along with a 70% reduction in time when comparing the time it took to complete tasks in the original workflow versus on TherapEase. The results of testing confirmed the system successfully improved workflow efficiency and usability.

7.2 SCRUM Methodology

The workload was divided across six sprints following the Scrum framework, which structures development into short, goal-focused iterations (Schwaber & Sutherland, 2020). This made the workload more manageable and allowed for tracking the progress of the application.

Sprints were divided into 2-3 week blocks from January 2026 to the end of April 2026. Each sprint consisted of a backlog and a set of goals. Some backlog tasks were functionality waiting to be implemented, while others were tasks from the previous sprint that couldn't be completed and need to be revisited.

7.3 Tools Used

A range of tools was used throughout the project to support organisation, development and debugging.

7.3.1 Trello

Trello is a kanban-style project management tool that allowed for the tracking of tasks under custom headings. The headings were the same as the project phases, but with implementation divided between backend development, frontend development and integrations. These headings had a list of tasks to be completed in each (Figure 53). This helped form a clear picture of what tasks need to be achieved in each phase of project development.

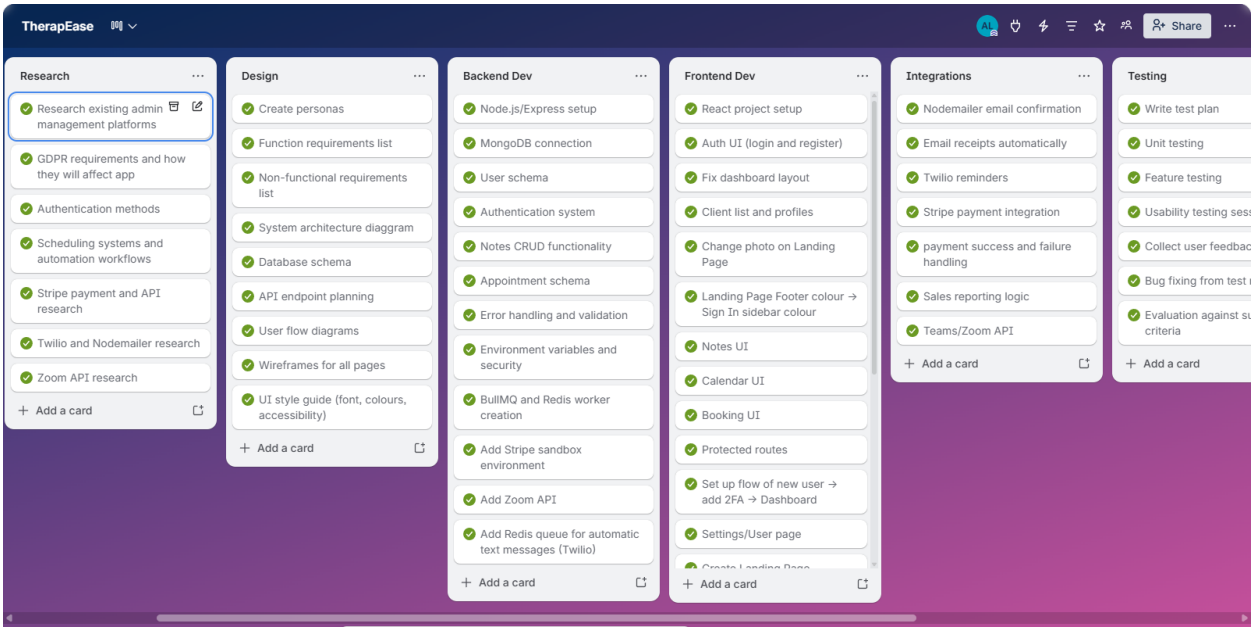

Figure 53 - Trello Board for TherapEase

7.3.2 GitHub

As the codebase was in a GitHub repository, regular commits were made to update the repository after every coding session or feature implementation. This allowed for rollbacks to the previous version of the project if needed, along with access for the project supervisor to see the progress made. Being able to see previous commits to the repository was helpful when documenting the implementation process, as commits were made with a message describing all changes made, along with a comparison of files to clearly see changes.

7.3.3 Miro

Miro is designed to be an online whiteboard, allowing for freestyle creation of brainstorms, workflows and plans. In the project development, it was used to create diagrams to explain the system architecture, personas and flow charts (Appendix B). Miro is highly customisable and includes pre-made templates, which were used to structure the creation of the personas and flow charts. It was also used

when collecting research about external APIs, allowing for decisions on application workflows and planning.

7.3.4 Notion

Notion is a customisable workspace with templates to create pages. A page was created specifically for this project that featured the project timeline and a more detailed to-do list that was updated daily. It also has a feature to connect to Teams calls and create an overview with a to-do list based on the call. This was used during user testing to create a list of action items and quotes taken from the call to be referred back to later (Figure 54, Figure 55).

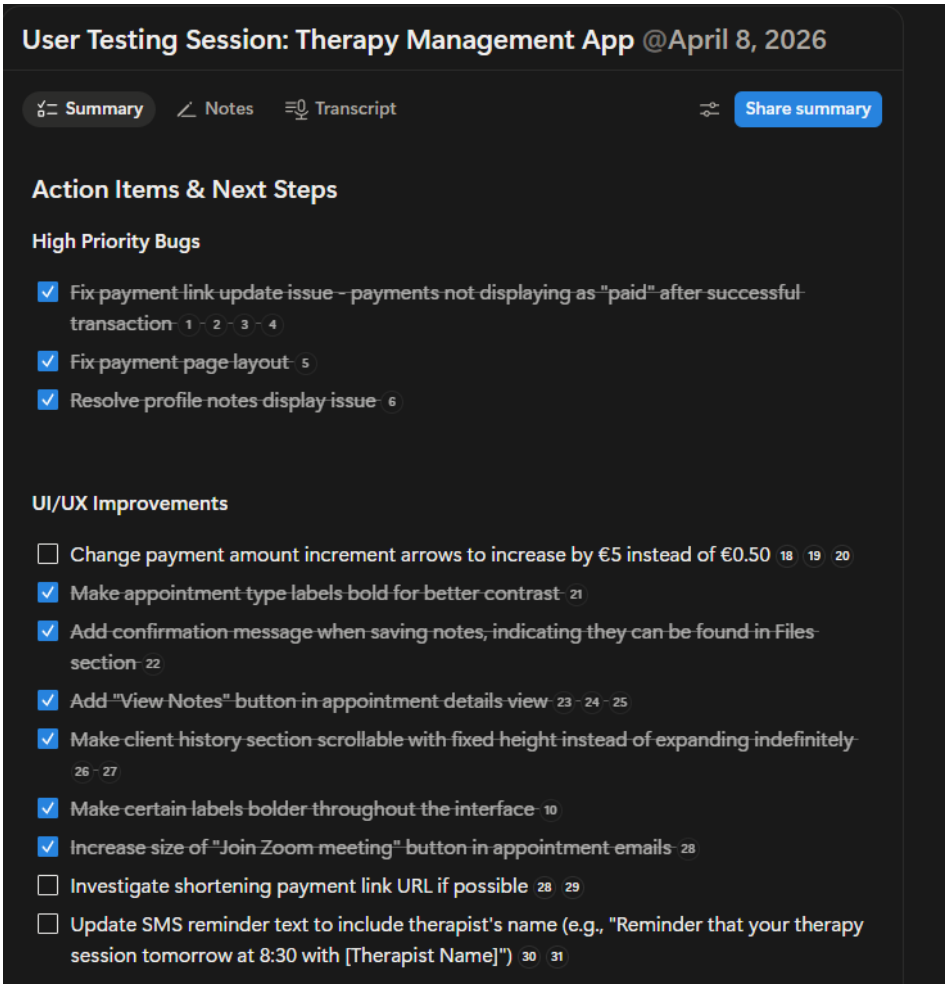

Figure 54 - Example of User Testing with Notion

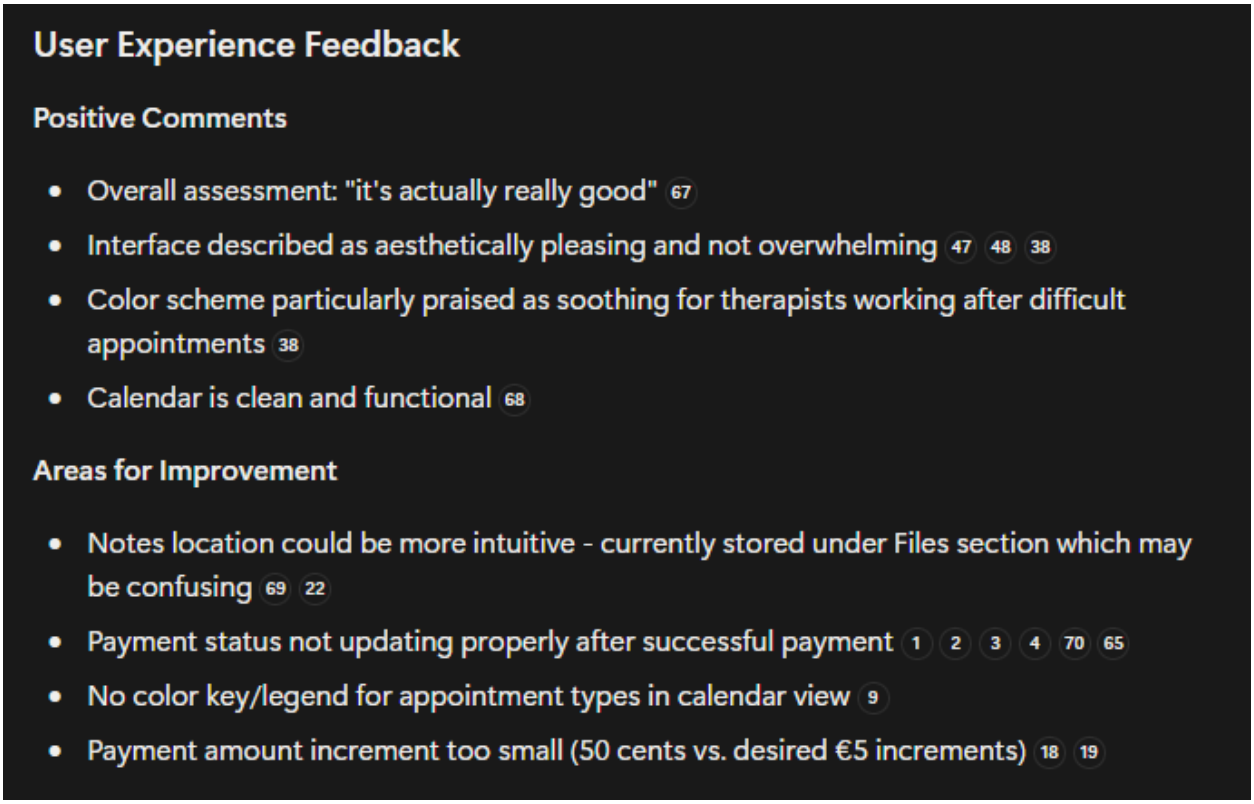

Figure 55 - Example 2 of User Testing with Notion

7.3.5 External Resources

Debugging was a significant part of the development of TherapEase, with multiple tools used to source errors and solutions. GitHub Copilot was a helpful tool when debugging small issues, as it had access to view the entire project and could scan files for issues such as missing imports or locating errors in files when the system returned unclear error messages. Console logs were used throughout development to ensure the right data was being received on each page and that the workers were running, shown in Figure 56. YouTube videos were used when integrating 2FA and Redis/BullMQ asynchronous processing to understand how the system works and how it could be implemented in the project. These videos were key to understanding the different components of the system and preventing errors.

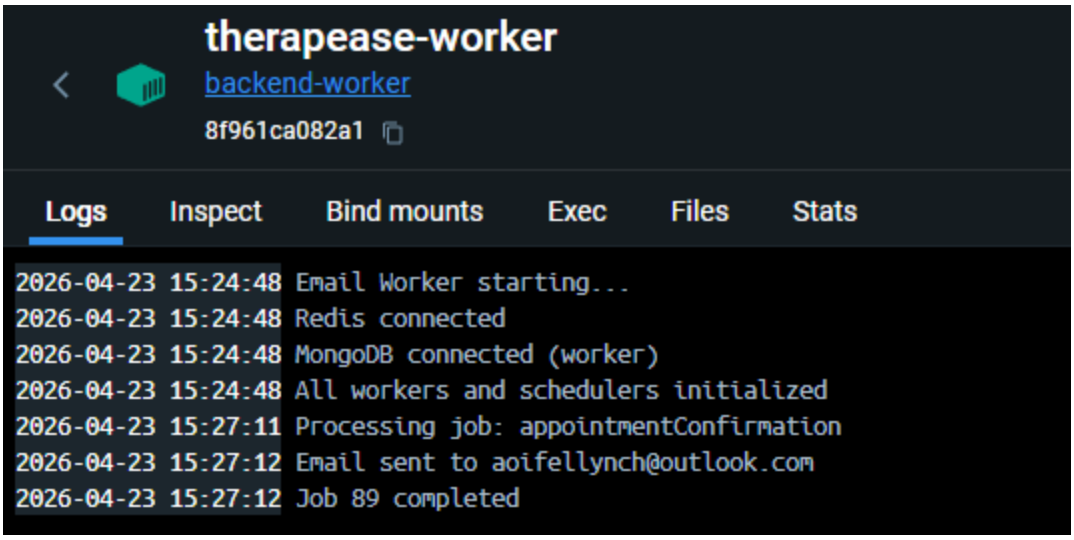

Figure 56 - Example of console logs in the asynchronous process

7.4 Reflection

7.4.1 Views on the Project

The project started with a very large scope of features and even after reducing this, it was an overwhelming list of features to implement. Due to this, the start of the development was very slow and, at times, felt as if it wouldn't be completed in time. Towards the end of the implementation phase, when many rounds of user testing were being performed, minor errors kept popping up and while the user feedback was very positive, each round brought on a long list of new to-dos. That being said, it was a very enjoyable application to develop as it addressed a real-world problem and with every new function, a sense of accomplishment was felt, especially with the more time-consuming integrations of Stripe and asynchronous processing. As one of the core aims for the application was usability and user-centred design, the frontend UI took more time than expected to get correct and some of the prototype designs were not successful in development, but once components had been created, the development of the UI sped up significantly, which allowed for more time to be put into exploring other ways around the issues faced on the dashboard, while still keeping the general layout the same. If the

implementation had to be done again, the only thing that would've been changed is getting the most difficult functionality finished first so that it wouldn't linger throughout other sprints and perhaps could've saved time for implementing the features that were taken out of the MVP, such as invoice generation or more advanced reporting and analytics.

7.4.2 Working with a Supervisor

Working with a project supervisor was a valuable experience throughout the course of the project. Regular meetings allowed for feedback and guidance with challenges. These meetings also helped ensure sprints were progressing and that deliverables each week were completed.

7.4.3 Technical Skills

This project helped develop technical skills with both familiar and unfamiliar technologies. Prior to the development of TherapEase, experience using Node.js and MongoDB was limited. This project required a deeper understanding of database design and managing application logic across controllers, services, routes and models. Because of the size of the project, confidence in using these technologies has grown substantially.

Multiple technologies were entirely new, including the Stripe API, Zoom API and asynchronous processing with Redis and BullMQ. Implementation of these technologies required extensive learning, especially with webhook handling and the concept of background queues. The successful implementation of these technologies, although time-consuming, represented a technical achievement and strengthened the ability to work with third-party services.

Deployment using Render was familiar, but configuring the application across multiple services was a learning curve. As the application had to be split into the backend, frontend, Redis and

background worker, new errors arose involving additional configuration and debugging. A greater understanding of the deployment of large applications has been gained from this.

A key area of improvement throughout the project was debugging and problem-solving. As the application grew larger and more complex, issues such as authentication errors, API failures and CORS configuration also required more work than in previous projects. This led to stronger skills in reading documentation, testing and creating solutions.

Overall, the project resulted in a strong increase in technical confidence, particularly in backend development, API integration and deployment.

Conclusion

The aim of this project was to address the usability challenges therapists face when using fragmented administrative systems by designing and developing an integrated web application. The final version of this application successfully combines scheduling, client management, payments and automated communication into a single platform that reduces the need for multiple tools such as Outlook, SharePoint, and SumUp. By implementing automated features such as email confirmations, SMS reminders, Zoom link generation and Stripe payment processing and receipts, the application achieved its goal of streamlining administrative workflows and reducing manual workload.

The research question explored how user-centred design principles could be applied to improve workflow efficiency in therapist administration tools. The results of the user testing show that using TherapEase reduces task switching, simplifies common tasks, and improves therapists' workflow efficiency. Overall, the application proves that an integrated, user-centred approach can significantly improve usability compared to fragmented workflows.

Despite successful test results, several limitations remain in testing and with extra features. The application was tested on a limited number of users and devices, which may not accurately represent all environments and workflows. Some advanced features, such as Stripe invoicing and more detailed reporting, were not implemented due to time constraints. While the system is functionally complete, further refinement is needed to improve scalability and support larger practices.

Future improvements would focus on implementing more enhanced reporting tools for both client statistics and revenue statistics to give the user a clearer view of their practice. Support for larger practices with an administrator user to oversee all calendars and payments would also be implemented to support a wide range of practices. The use of AI features, as requested in the initial survey, would be explored to assist with therapy administration. Finally, a version of TherapEase that allows clients to

self-book and contact their therapist could be implemented to further reduce the administrative workload.

In addition to future improvements to the application, other factors need to be implemented to make it ready for daily use by therapists. A second version of the application on Render needs to be created for staging so that testing the new version can be done without affecting the live one. The connection to Zoom would be changed from the Direct connection it's currently using to a Server-to-Server OAuth app, as it's more secure and won't expire. Extensive unit, integration, and end-to-end testing would need to be conducted to ensure all error scenarios are covered, as well as penetration testing to ensure there are no back doors when accessing data or processing payments. For GDPR audit trails, a log would need to be made to show who accessed what data and when. Finally, the live production keys would need to be swapped out for the test keys to make it a usable application.

This project significantly strengthened skills in full-stack development using the MERN stack, API integration, asynchronous processing and user interface design. Technologies such as Stripe, Zoom, Twilio, Nodemailer, Redis and BullMQ introduced new challenges that improved problem-solving and debugging skills that will transfer to other future projects and improvements to TherapEase. Due to the scope of the project and its many features, time management skills were also greatly improved. The project showed the importance of user-centred design and of aligning real-world workflows with an application.

References

Auth0. (2025). *JSON Web Tokens - jwt.io*. JSON Web Tokens - Jwt.io; Auth0. https://www.jwt.io/

Bao, M. (2016). *Speakeasy*. Npm. https://www.npmjs.com/package/speakeasy

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., & Thomas, D. (2001). *Manifesto for Agile Software Development*. Agile Manifesto. https://agilemanifesto.org/

Brooke, J. (1995, November). *SUS: a Quick and Dirty Usability Scale*. ResearchGate. https://www.researchgate.net/publication/228593520_SUS_A_quick_and_dirty_usability_scale

BullForce Labs AB. (2018). *BullMQ - Background Jobs Processing and Message Queue for NodeJS*. Bullmq.io. https://bullmq.io/

chart.js. (2019). *Chart.js | Open source HTML5 Charts for your website*. Chartjs.org. https://www.chartjs.org/

Docker. (2024). *Enterprise Application Container Platform | Docker*. Docker. https://www.docker.com/

Ensora Health. (2025, December 12). *Theranest*. Ensora Health. https://ensorahealth.com/product/theranest-mental-health/

European Data Protection Supervisor. (2018, May 25). *The History of the General Data Protection Regulation | European Data Protection Supervisor*. Www.edps.europa.eu. https://www.edps.europa.eu/data-protection/data-protection/legislation/history-general-data-protection-regulation_en

Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures* [Dissertation]. https://roy.gbiv.com/pubs/dissertation/fielding_dissertation.pdf

GitHub. (2025a). *GitHub*. GitHub. https://github.com/

GitHub. (2025b). *GitHub Copilot · Your AI pair programmer*. GitHub. https://github.com/features/copilot

Gould, J. D., & Lewis, C. (1985). Designing for usability: key principles and what designers think. *Communications of the ACM*, *28*(3), 300–311. https://doi.org/10.1145/3166.3170

Gulliksen, J., Göransson, B., Boivie, I., Blomkvist, S., Persson, J., & Cajander, Å. (2003). Key principles for user-centred systems design. *Behaviour & Information Technology*, *22*(6), 397–409. https://doi.org/10.1080/01449290310001624329

Jest. (2017). *Jest · Delightful JavaScript Testing*. Jestjs.io. https://jestjs.io/

Kruse, C. S., Kothman, K., Anerobi, K., & Abanaka, L. (2016). Adoption factors of the electronic health record: A systematic review. *JMIR Medical Informatics*, *4*(2), e19. https://doi.org/10.2196/medinform.5525

Mazur, L. M., Mosaly, P. R., Moore, C., & Marks, L. (2019). Association of the Usability of Electronic Health Records With Cognitive Workload and Performance Levels Among Physicians. *JAMA Network Open*, *2*(4), e191709. https://doi.org/10.1001/jamanetworkopen.2019.1709

Melnick, E. R., Harry, E., Sinsky, C. A., Dyrbye, L. N., Wang, H., Trockel, M. T., West, C. P., & Shanafelt, T. (2020). Perceived Electronic Health Record Usability as a Predictor of Task Load and Burnout Among US Physicians: Mediation Analysis. *Journal of Medical Internet Research*, *22*(12), e23382. https://doi.org/10.2196/23382

Merencia, L. (2025). *Node-cron*. Npm. https://www.npmjs.com/package/node-cron

Meta Open Source. (2025). *React*. React.dev. https://react.dev/

Microsoft. (2025a). *Fast and reliable end-to-end testing for modern web apps | Playwright*. Playwright.dev. https://playwright.dev/

Microsoft. (2025b). *Visual Studio Code*. Visualstudio.com; Microsoft. https://code.visualstudio.com/

MongoDB, Inc. (2024). *MongoDB*. MongoDB. https://www.mongodb.com/

Mongoose. (2011). *Mongoose ODM v5.8.2*. Mongoosejs.com. https://mongoosejs.com/

Nielsen, J. (1994, April 24). *10 Heuristics for User Interface Design*. Nielsen Norman Group. https://www.nngroup.com/articles/ten-usability-heuristics/

Nodemailer. (2010). *Nodemailer*. Nodemailer.com. https://nodemailer.com/

Olakotan, O., Samuriwo, R., Ismaila, H., & Atiku, S. (2025). Usability Challenges in Electronic Health Records: Impact on Documentation Burden and Clinical Workflow: A Scoping Review. *Journal of Evaluation in Clinical Practice*, *31*(4). https://doi.org/10.1111/jep.70189

OpenJS Foundation. (2017). *Express - Node.js web application framework*. Expressjs.com. https://expressjs.com/

OpenJS Foundation. (2025). *Node.js*. Node.js. https://nodejs.org/en

Postman. (2025). *Postman | The Collaboration Platform for API Development*. Postman. https://www.postman.com/

Redis, Inc. (n.d.). *Redis*. Redis.io. Retrieved 2026, from https://redis.io/

Render. (n.d.). *Cloud Application Hosting for Developers | Render*. Cloud Application Hosting for Developers | Render. https://render.com/

Saparamadu, A. A. D. N. S., Fernando, P., Zeng, P., Teo, W. M. H., Goh, X. T. A., Lee, J., & Lam, C. W., Leslie. (2020). A User-centered Design Process of an mHealth Application for Health Professionals: A Case Study (Preprint). *JMIR MHealth and UHealth*, *9*(3). https://doi.org/10.2196/18079

Schwaber, K., & Sutherland, J. (2020, November). *The 2020 Scrum Guide*. Scrumguides.org. https://scrumguides.org/scrum-guide.html

SimplePractice. (n.d.). *Practice Management Software and EHR Made Simple*. SimplePractice. https://www.simplepractice.com/

SmartBear Software. (2021). *API Documentation Made Easy - Get Started | Swagger*. Swagger.io. https://swagger.io/solutions/api-documentation/

Stafford, R. (2003). *P of EAA: Service Layer*. Martinfowler.com. https://martinfowler.com/eaaCatalog/serviceLayer.html

Stripe, Inc. (n.d.). *Stripe documentation*. Docs.stripe.com. Retrieved 2026, from https://docs.stripe.com/

Tailwind Labs Inc. (2025). *Tailwind CSS - Rapidly build modern websites without ever leaving your HTML.* Tailwindcss.com. https://tailwindcss.com/

Twilio. (2024). *Communication APIs for SMS, Voice, Video & Authentication | Twilio*. Www.twilio.com. https://www.twilio.com/en-us

Zoom Video Communications. (n.d.). *Introduction to Zoom APIs*. Developers.zoom.us. Retrieved 2026, from https://developers.zoom.us/docs/api/

Appendix

Appendix A

Survey on Admin Workflow and Tool Usage

This appendix contains the full results from the Microsoft Forms survey conducted with therapists

Link: Microsoft Forms – Survey Results

Appendix B

Project Miro Board

This appendix includes the Miro board used for project planning, diagram creation and research documentation.

Link: https://miro.com/app/board/uXjVJj9jbiU=/?share_link_id=850906214813

Appendix C

GitHub Copilot Chat

This appendix provides selected examples of interactions with GitHub Copilot, demonstrating its use in development, problem-solving and code refactoring.

Link: Copilot Chat Examples.pdf

Appendix D

API Documentation

This appendix contains the full Swagger API documentation for TherapEase that shows all endpoints.

Link: https://therapease-j72q.onrender.com/api/docs/

Appendix E

Docker Compose File

This appendix includes the Docker Compose configuration used to manage the development environment, including services for the API, database, Redis and worker.

Link: Docker Compose File.txt

Appendix F

GitHub Repository

This appendix links to the full source code repository for the TherapEase application.

Link: https://github.com/aoifelynch/therapease.git

Appendix G

System Usability Scale Results

This appendix links to the results of the System Usability Scale questionnaire, completed by all testing participants.

Link: System Usability Scale Results.pdf

Appendix H

TherapEase Frontend Screenshots

This appendix links to the screenshots of all the pages in the TherapEase application.

Link: TherapEase Live Application Screenshots.docx

Appendix I

Trello Board

This appendix links to the Trello Board used for tracking the to-do list across project phases

Link:https://trello.com/invite/b/6961358906cc87cabfe7781f/ATTIdfdf9caa13ec6886c07267469997b2eb17FE7A5A/therapease

Appendix J

Figma Prototypes

This appendix links to the Figma page where the application's prototypes were created.

Link: https://www.figma.com/design/TFVTm5Sur19HYAi6v4qIZb/TherapEase?node-id=0-1&t=XBiAd49PrMXADlR8-1

Appendix K

Minimum Viable Product

This appendix links to the description of the Minimum Viable Product for TherapEase

Link: Minimum Viable Product.pdf

Appendix L

Technical Testing Results

This appendix links to the images showing the results of end-to-end testing and unit/integration testing

Link: Appendix L - Technical Testing Results