DÚN LAOGHAIRE INSTITUTE OF ART, DESIGN AND TECHNOLOGY

FlexiCare: A Digital Patient Management System for Physiotherapy

Author: Alice Corry

Student Number: N00211635

Supervisor: John Montayne

Second Reader: Cyril Connolly

Dissertation submitted in partial fulfilment of the degree

of BSc (Hons) in Creative Computing

Declaration of Authorship

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

You should document this in your submitted work if you have received significant help with a solution from one or more colleagues. If you doubt what discussion/collaboration is acceptable, 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 leave copies of your files on a hard disk where others can access them. Remember that removable media used to transfer work may be removed and/or copied by others if left unattended.

Plagiarism is an act of fraudulence and an offence against the Institute's 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 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, with one individual giving a solution to another, who then makes some changes and hands it up as their work.

DECLARATION

I know the Institute’s policy on plagiarism and certify that this thesis is my work.

Alice Corry

Abstract

This paper presents the development of FlexiCare, a digital solution aimed at modernising physiotherapy management for both clinicians and patients. It focused on improving the management of patient data, treatment plans, communication and patient self-management. The system was developed using an iterative SCRUM approach, which divides a large project into a series of smaller sections called sprints. Key methods employed included existing product evaluation, key user surveys and feasibility testing.

The FlexiCare backend development utilised Microsoft's ASP.NET Core MVC framework, which is capable of intuitively and securely managing user data such as personal information, exercise routines, and appointments. The backend uses C# code and Entity Framework Core, Microsoft's Object-Relational Mapping (ORM) tool for mapping databases and SQLite as a lightweight but reliable data engine. Key components included the FlexicCare Manager for physiotherapists and administrators, role-based authentication using Microsoft Identity and a separately designed FlexiCare API, which used JSON Web Token (JWT) authentication and Swagger Support for the visual interface. Scaffolding was used for the rapid development of views and controllers, while data seeding was employed to populate the data with sample patient and exercise data for testing and development.

The frontend FlexiCare mobile development utilised React Native and Expo. Key tools included Visual Studio Code for coding, Insomnia for API testing and Figma for UI design. Views like the login screen, task list and appointment page were developed to provide an intuitive, yet visually pleasing user interface. Examples of features include secure authentication through the login view using AuthContext, and a task page displaying a list of assigned exercises with a daily streak feature for motivation. In addition, there was an appointment page developed to feature a data carousel to easily navigate through appointments and a feedback form that allows you to input data such as rep counts and pain levels, using metrics and sliders.

Comprehensive testing was conducted on both the front and backend, using Jest and MSTest, respectively, to ensure functionality across all systems and devices, various conditions, and user inputs. FlexiCares development incorporated modern technology and best practices in backend and frontend design, with secure data management, intuitive UI and detailed automated testing, This project demonstrates how careful research and an iterative design process can come together to create a solution that addresses the complex needs of the healthcare professional while still prioritising an efficient and supportive user experience.

Acknowledgements

I would like to express my sincere thanks to everyone who supported me throughout my thesis.

First, I am deeply grateful to my Project Supervisor, John Montayne, for his guidance, helpful feedback, and encouragement. His expertise and patience made a massive difference in this process.

I would also like to thank my lecturers at IADT for their valuable teachings and support throughout my studies. Their input has been essential in shaping my research and understanding.

A special thank you also to my family for their constant love, support, and encouragement. Their belief in me kept me going, especially during challenging times.

Finally, I acknowledge the Learning Support at IADT for providing valuable resources and assistance, helping me stay focused and on track with my work.

Thank you all for your support. This work would not have been possible without you.

Table of Contents

Declaration of Authorship **2**

Abstract **3**

Acknowledgements **4**

Table of Contents **5**

1. Introduction **7**

1.1. Admin Web App 7

1.2. Patient Engagement Mobile App 7

2. Research **9**

2.1. Introduction 9

2.2. Why Patient Engagement Matters 9

2.3. Integrating Healthcare Systems 10

2.4. Patient Self-Management and Outcomes 10

2.5. Keeping Healthcare Apps Secure 10

2.6. Review of Similar Applications 11

2.7. Conclusion 13

3. Requirements and Feasibility **15**

3.1. User Requirements 15

3.2. Functional Requirements 24

3.3. Non-functional requirements 24

3.4. System Model 24

3.5. Feasibility Study 25

3.6. Limitations 26

4. Design **28**

4.1. Program Design 28

4.2. User Interface Design 33

5. Implementation **41**

5.1. SCRUM Methodology 41

5.2. Development Environment..... 41
5.3. Sprint 1: Python Development..... 42
5.4. Sprint 2: Backend Development..... 43
5.5. Sprint 3: Frontend Development..... 54
6. Testing and Results..... 76
6.1. Usability Testing..... 76
6.2. Unit Testing..... 76
6.3. Evaluation..... 80
7. Project Management..... 82
7.1. Project Phases..... 82
7.2. Project Management Tools..... 83
8. Reflection..... 88
8.1 Your Views on the Project..... 88
8.2 Completing a Large Software Development Project..... 88
8.3 Working with a Supervisor..... 88
8.4 Technical Skills..... 88
8.5 Further Competencies and Skills..... 89
8.6. Future Plans..... 89
9. Conclusion..... 91
References..... 92

1. Introduction

FlexiCare is a multi-faceted digital platform designed to make physiotherapy more efficient and accessible for physiotherapists and patients. The project will be divided into two main components:

1.1. Admin Web App

This web application will help physiotherapists manage their work more effectively. The app will be built using ASP.NET Core, a C# framework designed for simplified web development. This app will allow physiotherapists to digitally:

- Organise and access patient records
- Schedule and manage appointments
- Create, update, and monitor treatment plans.

The user interface will be designed using HTML views, and data queries will be handled through REST to ensure fast and flexible access to information. With this system, physiotherapists can easily add, edit or delete patient information, helping them save time and reducing needless administrative tasks.

1.2. Patient Engagement Mobile App

This mobile app will help patients stay engaged and on track with their treatment plans. Developed using React Native, it can work seamlessly on both IOS and Android Devices. Patients will be able to:

- Track their daily exercises
- Receive reminders for upcoming appointments
- Access valuable educational resources related to their care

The app will feature a modern and simplistic design, using NativewindCSS, a tool for efficient styling. The app will be tested using Jest to ensure high performance across devices of various performance levels.

Flexicare will use SQLITE, a reliable and secure database system for storing essential data such as patient information, treatment plans, exercise logs, and progress tracking. This will ensure that both the web and mobile applications can manage data efficiently, while maintaining a high standard of security and accessibility. Both Physiotherapists and patients alike will be able to access and upload their information quickly and safely.

The primary goal of FlexiCare is to make physiotherapists' jobs easier by minimising time spent on administrative tasks, while helping patients stay actively engaged in their recovery. This project hopes to improve the efficiency of physiotherapy practices and the patient experience by building a web app dedicated to physiotherapists and a mobile app for patients.

FlexiCare is designed to grow with time. As more users join the platform, the system can handle increased data and traffic without compromising performance. Both applications will be intuitive and user-friendly, making them accessible to those with limited technical experience. While dealing with sensitive information, security is a top priority. Thus, the system will ensure that all patient data remains protected.

Ultimately, FlexiCare seeks to become a simple, trusted, all-in-one solution for physiotherapy, allowing therapists to do efficient work while supporting patients every step in their recovery journey.

2. Research

2.1. Introduction

Healthcare is constantly evolving, and technology plays a bigger role than ever. Systems like the proposed FlexiCare, a digital patient management tool, aim to enhance patient care, especially in physiotherapy. This literature review explores essential themes in designing and implementing such systems, focusing on information systems, communication, patient involvement, and safety in healthcare. The aim is to present the current landscape, highlight opportunities and challenges, and support the development of FlexiCare. Understanding these topics is essential for creating a reliable, user-friendly system that promotes communication, supports self-management, and ensures patient safety.

2.2. Why Patient Engagement Matters

In physiotherapy, patient engagement is key to determining treatment success and overall patient satisfaction. Engaged patients are more likely to follow their prescribed exercise regimen, actively participate in their rehabilitation and communicate openly with health professionals. This is crucial as it is what patient-centred care is about and it improves clinical outcomes. Studies show that the more a patient is involved in their treatment plan, the better the results and the lower the costs [1].

Introducing a physiotherapy tracking app to improve patient engagement could be an effective first step. But how is that to be done? Well, in this study ,it is suggested that using user-centered functions that address the patient's needs and preferences [1]. At a minimum, the app should offer educational tools that could explain the details and importance of the medical condition, the prescribed therapy, and why commitment to such a therapy plan is of the essence. Secure messaging and video consultation can further enhance the experience by allowing patients to communicate directly with the medical professional, fostering real-time support.

Technology can create interactive opportunities that strongly boost user motivation. Allowing for progress tracking and personalised feedback may let a patient visualise their progress over time. "Seeing is believing," whereby an individual will feel a greater sense of accomplishment and thereby may wish to continue trying. More importantly, they will feel confident that what they are doing helps them to improve their abilities. This development, when one comes to think about it, is not unlike that "aha" moment when patients sometimes suddenly realise that the therapy is working and that they are getting better.

Emphasising patient engagement in physiotherapy apps heightens their commitment to treatment and health overall. Patients who feel like active participants in their care are more likely to stay committed and achieve better outcomes. At the core of the FlexiCare approach, is the belief that a successful digital therapy application should educate, communicate, and motivate , three components of a successful user experience for any patient [2]. Moreover, while it is the app that is facilitating this, it is ultimately the thinking behind it, coming up with such innovative features that allows patient engagement to become possible in the first place [2].

2.3. Integrating Healthcare Systems

Integrating different healthcare systems is important but complex. The goal is to create a seamless system that improves care for patients and simplifies work for professionals. This means connecting health information systems (HIS) with everyday clinical practices in for example, nursing and physiotherapy [3]. There are a list of challenges that include interoperability issues, poor communication between systems, and restrictive regulations. Still, integration is vital for improving healthcare delivery.

2.4. Patient Self-Management and Outcomes

Supporting patients in managing their own health is a cornerstone of modern healthcare, especially those with chronic conditions [4; 5]. Physiotherapists are well-positioned to provide structured, evidence-based self-management supports for their patients [4]

These supports might include one-on-one sessions, follow-ups, and tailored information. It helps patients understand their condition, recognise warning signs, develop coping skills, and take control of their lifestyle and medications. Regular feedback and encouragement to make informed decisions are also important. Programs that use multiple strategies tend to lead to better outcomes and higher quality of life [4].

To evaluate treatment success, healthcare systems often use patient-reported outcome measures (PROMs). These tools gather data directly from patients about their health status. Despite their value, PROMs are not always used consistently in physiotherapy due to various barriers [6]. Mobile Health apps can incorporate PROMs and support self-management, but user perceptions and experiences play a key role in their effectiveness [7].

2.5. Keeping Healthcare Apps Secure

The prevention of cyber attacks on health information systems in healthcare services is fundamental, mainly because these systems handle highly sensitive patient data. Healthcare data, including electronic health records and personal health information, needs strong security measures [3]. Protecting sensitive patient information, preventing unwanted use, and stopping theft are all vital aspects of healthcare data security [3].

The increasing use of digital systems in healthcare, including connected medical devices and electronic health records, has unfortunately also introduced significant cybersecurity concerns [6]. Cybersecurity incidents are becoming more frequent and can seriously affect healthcare organisations [7]. These effects range from the accidental release of protected health information to disruptions in patient care [7].

In today's environment with advanced cyber threats, it is almost certain that successful attacks will occur [7]. Therefore, healthcare organisations must have effective strategies to respond to these incidents [7]. Recent events, such as ransomware attacks that have

taken parts of healthcare organisations offline, show the significant impact of these incidents [7]. Understanding how organisations respond and how to improve these responses is vital for maintaining safe and effective healthcare delivery globally [7]. Developers must implement a comprehensive set of multi-layered security measures, including encryption, secure access mechanisms, and regular software updates [8].

Encryption is one of the most effective and straightforward tools for safeguarding data. It ensures that information, whether it is being transferred or stored, is converted into unreadable code unless accessed without a key. This means that if the data is intercepted or accessed without permission, it remains unusable without the necessary keys to decrypt it [9]. Even basic encryption methods can offer strong protection.

Keeping the app regularly updated is another critical step. As cyber threats evolve, older versions of software may become more vulnerable. Developers must respond quickly patching security flaws and releasing updates. A strong developer will also regularly audit their own work to make sure no new bugs have introduced a problem into the app.[10]

Trust is the foundation of any healthcare relationship. If patients feel their data is not secure, the credibility of the physiotherapy app is compromised. Protecting user data is not optional, it is a fundamental responsibility. Developers must therefore incorporate strong encryption, secure authentication methods, and regular updates to build and maintain trust between patients and providers.[11]

2.6. Review of Similar Applications

Before beginning the development of FlexiCare, it is essential to analyse existing applications that offer similar features. Studying comparable systems is an efficient way to identify established best practices, common issues and gaps in the current functionality that offer opportunities for innovation. It ensures FlexiCare can build upon proven technologies and offer improvements where current solutions fall short. Table 1 compares two such applications: Helix and PhysiApp, focusing on their key functionality, algorithms, advantages and weaknesses.

Helix is a web-based application designed for physiotherapists. It helps manage patient records, track progress and create personalised treatment plans. Helix uses integrated algorithms to assist physiotherapists in monitoring patient outcomes and adjusting their treatment plans accordingly. As shown in Table 1, Helix offers several advantages, such as streamlining the physiotherapy workflow, providing customisable treatment plans and using data-driven insights to support physiotherapists in making informed decisions. However, two notable limitations were observed. It is a web application, not a permanently downloadable app, requiring a stable internet connection to function correctly. Its relatively complex design may also create a harsh learning curve for new users.

PhysiApp is a mobile application designed to support patients by tracking their exercises, sending reminders, and offering feedback on their progress. It helps patients stay engaged with their treatment plans by sending regular notifications and personalised exercise suggestions. PhysiApp uses algorithms to track exercise completion, deliver personalised reminders and feedback, and assist with goal setting and tracking. Its advantages include

keeping patients on track with their exercises through easy-to-follow progress tracking and reminders/feedback. While PhysiApp accessibility is an improvement over Helix, as a mobile app, it is limited to mobile devices. It is still noticeably complex and may not be suitable to less tech-savvy users.

Table 1: Review of Applications with Similar Functionality

Application	**Helix**	**PhysiApp**
Category	Physiotherapy management	Patient exercise tracking
Technology	Web App	Mobile App
Description and functionality	Helix helps physiotherapists manage patient records, track progress, and create personalised treatment plans. It uses data-driven tools to optimise treatment based on patient needs.	PhysiApp supports patients by tracking their exercises, sending reminders, and offering feedback on progress. It engages patients with their treatment plans through notifications and personalised exercise suggestions.
Algorithms	• Patient data analysis • Treatment plan optimisation • Progress monitoring	• Exercise completion tracking • Personalised reminders and feedback • Goal setting and tracking
Advantages	• Streamlines physiotherapy workflow • Customisable treatment plans • Data-driven insights for physiotherapists	• Keeps patients engaged and on track with exercises • Easy-to-follow progress tracking • Offers feedback and reminders
Disadvantages	• Requires internet access • It may be complex for new users	• Limited to mobile devices • It may not be suitable for all patients, especially those less tech-savvy

The images below display the user interfaces of Helix (Figure 1) and PhysiApp (Figure 2). These examples provide valuable insight into how these existing applications support physiotherapists and patients through their design choices. FlexiCare hopes to build on these insights, by analysing the strengths and weaknesses of these designs. FliexCare's interface will deliver a more intuitive, accessible and user-friendly experience.

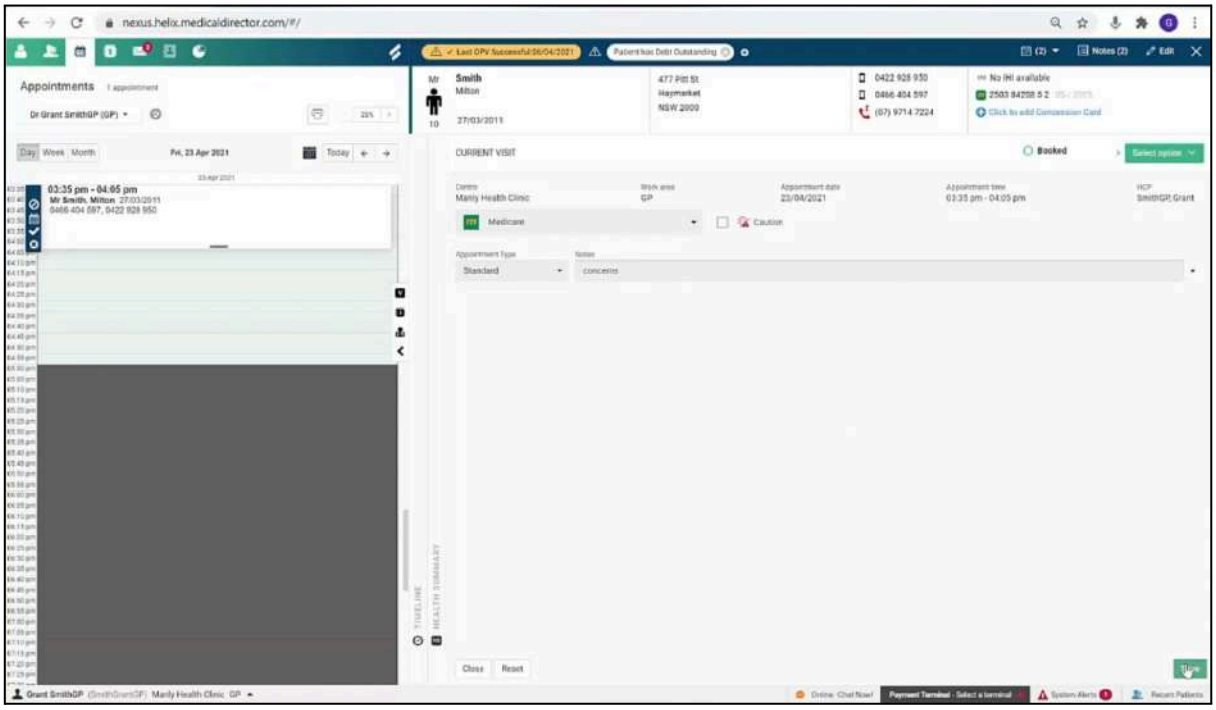


Figure 1: Helix Interface [12]

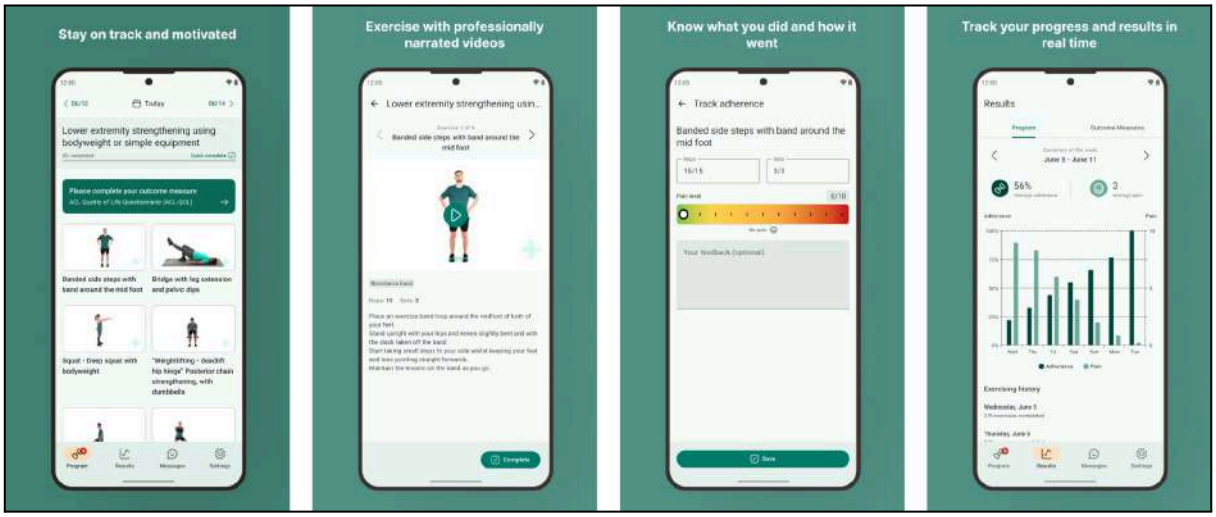


Figure 2: PhysiApp interface. [13].

2.7. Conclusion

This research has explored important themes involved in designing and developing a digital patient self-management system like FlexiCare within physiotherapy. As healthcare becomes increasingly more digital, patient engagement, integrated systems, data security and supporting self-management are highlighted as key areas of importance. Patient

engagement in particular is vital to improving treatment motivation and outcomes, and this should be supported through user-friendly, interactive technology. Integrating healthcare systems is a significant challenge, but it is necessary to streamline digital care. Encouraging patient self-management through feedback, educational content and other tools can lead to self-supportive patients and better recovery outcomes.

However, no digital solution is possible without strong cybersecurity. Within the world of healthcare, patient confidentiality is paramount above all else, and as cyber threats rise, safeguarding patient information through encryption, secure authentication and regular updates is crucial. The trust in any healthcare system depends directly on the strength of these protections.

The review of Helix and PhysiApp highlights important lessons for FlexiCare, highlights aspects that are successful and can be built upon, and areas of weakness such as intuitive design, that can be learned from.

In conclusion, the development of FlexiCare is not as simple as creating an app, it is about enhancing care, empowering patients to be self-supportive and self-managing, and supporting professionals through an efficient design. By learning from current challenges and addressing existing best practices, FlexiCare has the potential to set the standard for digital physiotherapy tools.

3. Requirements and Feasibility

A clear understanding of the system requirements and feasibility is essential for building a complicated, yet effective and user-intuitive application such as FlexiCare. This section will outline the key features required to meet the goals of the project, and will explore potential technical and practical challenges that may arise during development. It will also introduce the initial system model and outline the project's early roadmap. This will ensure that the development is structured and informed from the outset.

The requirements are divided into three primary categories:

- User requirements
- Functional requirements
- Non-functional requirements

3.1. User Requirements

User requirements specify what the end-user aims to achieve with the software. These are generally not technical requirements, but rather are focused on the wants and needs of the user, and serve as the foundation for designing a user-friendly and effective application. In the case of FlexiCare, gathering accurate user input prior to development was essential to ensure that the final product directly aligns with the needs and expectations of both physiotherapists and their patients.

A targeted user survey was conducted to obtain these insights.

3.1.1. Survey

A survey was conducted with practising physiotherapists to collect meaningful input on what they consider essential in a physiotherapy tracking and management application. This survey served as a critical tool, offering valuable insights into both common current issues and desired potential features. By capturing real-world experiences and expectations, it was possible to gather the understanding required for an application that can truly support and enhance physiotherapy care.

The survey was distributed anonymously via Google Forms, ensuring that participants could provide honest and uninfluenced feedback. Ten professionals from a range of age groups, practice settings and levels of experience participated. The hope of this mix was to provide a balanced perspective and have a general overview of a larger proportion of the potential user base

The questionnaire was designed to gather both quantitative data, in the form of multiple choice rankings, and qualitative feedback, such as questions with open-ended comment responses. Physiotherapists were asked to evaluate the usefulness of various proposed features, and their responses highlighted a value for simplified record-keeping and appointment management. The survey results will be used as the basis for the FlexiCare design, ensuring it meets the needs of physiotherapists and enhances patient care.

3.1.2. Why the Survey Matters

Conducting user-research is not just a formality but rather a vital step in the development of any application, particularly within healthcare. Direct input from users is indispensable for a variety of reasons:

1. Real-World Relevance

By consulting physiotherapists early in the process, it ensures that FlexiCare addresses actual needs rather than simple assumptions. This reduces the risk of developing features that could be unused or considered unhelpful.

2. User Validation

A survey can act as a validation tool for potential features or tools. If a concept ranks highly among the participating user-base, it signals that it is in high demand. This can help prioritise development resources and time.

3. Design Focus

It can help define the personas and use cases that guide UX design. Knowing who the user is and what they value means every interaction in the app can be shaped around their expectations.

The survey results offer valuable insights into the challenges physiotherapists face when supporting patients with exercise issues. As seen in Figure 3, a significant 85.7% of respondents identified low patient motivation as a major barrier, with 57.1% reporting that patients frequently forget to perform their prescribed exercises. Additionally, 42.9% highlighted physical discomfort, pain, or uncertainty about proper technique as common factors that hinder adherence. These insights point to a consistent need for tools that can provide both guidance and encouragement throughout the rehabilitation process.

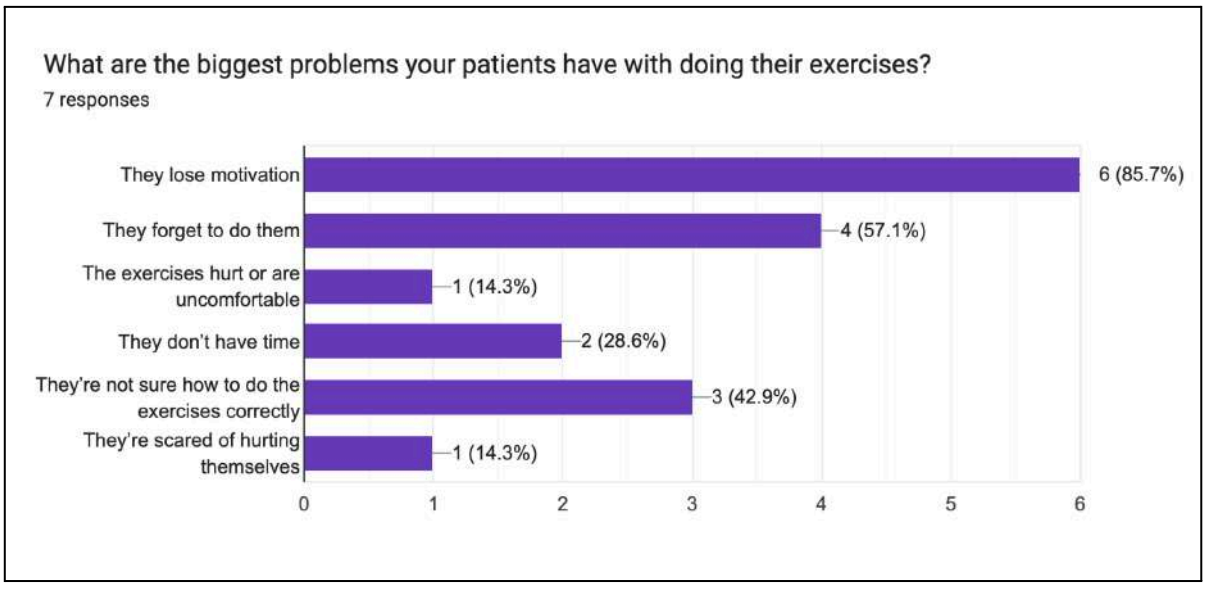


Figure 3: Survey Results - What are the biggest problems your patients have with doing their exercises?

As shown in Figure 4 when asked about desired features in a physiotherapy app, the vast majority (85.7%) of physiotherapists rated explicit exercise videos and timely reminders as the most valuable tools to assist patients. These were closely followed by progress-tracking features and adjustable exercise plans, with 57.1% of participants expressing strong interest in tools that allow for flexibility and individualisation of therapy routines.

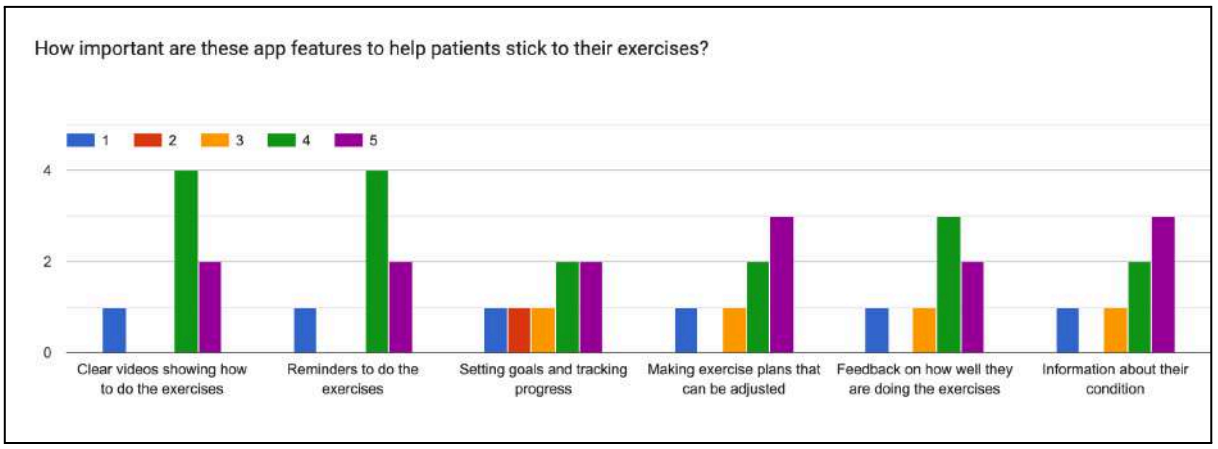


Figure 4: Survey Results - What are the biggest problems your patients have with doing their exercises?

On the clinical side, Figure 5 shows 71.4% of respondents indicated that chronic pain and arthritis are among the most prevalent conditions affecting their patients, while 57.1% noted that post-surgical recovery was a primary concern. These findings underscore the importance of tailoring exercise content to support long-term conditions and recovery-focused rehabilitation.

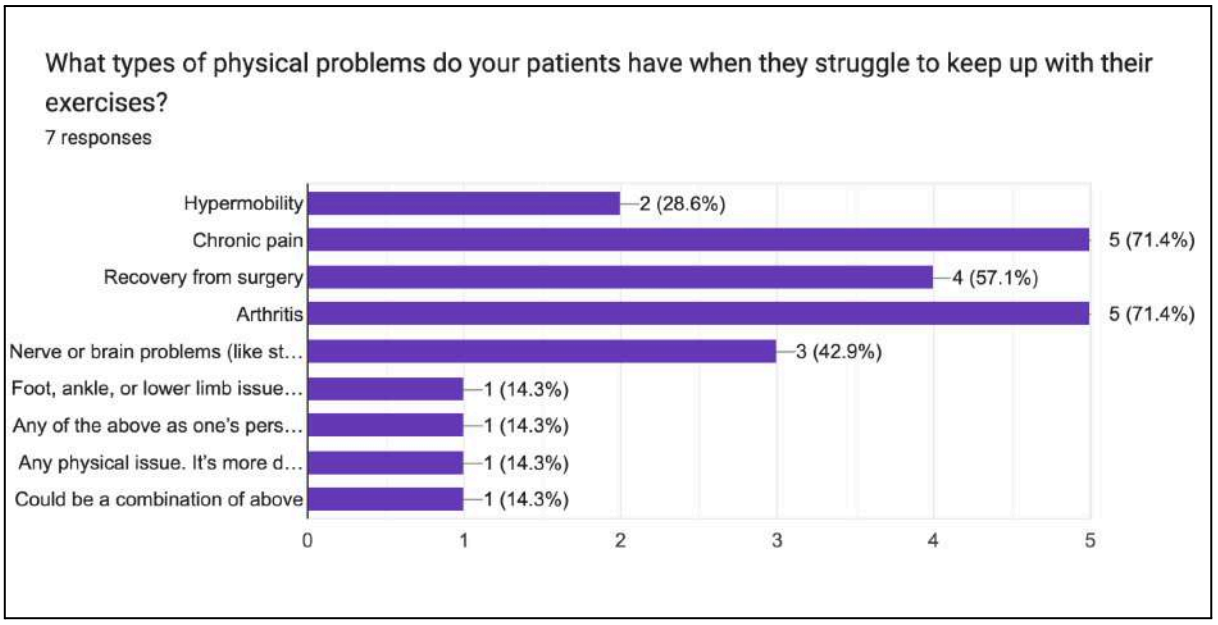


Figure 5: Survey Results - What types of physical problems do your patients have when they struggle to keep up with their exercises?

In terms of features and functionality, Figure 6 shows 85.7% of participants expressed a strong preference for the ability to customise exercise plans within the app, reinforcing the importance of adaptable treatment options. Moreover, as seen in Figure 7, the same percentage agreed that including clear exercise demonstrations and reminder features

would be particularly beneficial in helping patients stay on track. An additional 71.4% supported the inclusion of relevant patient health information within the app to provide context and improve care coordination.

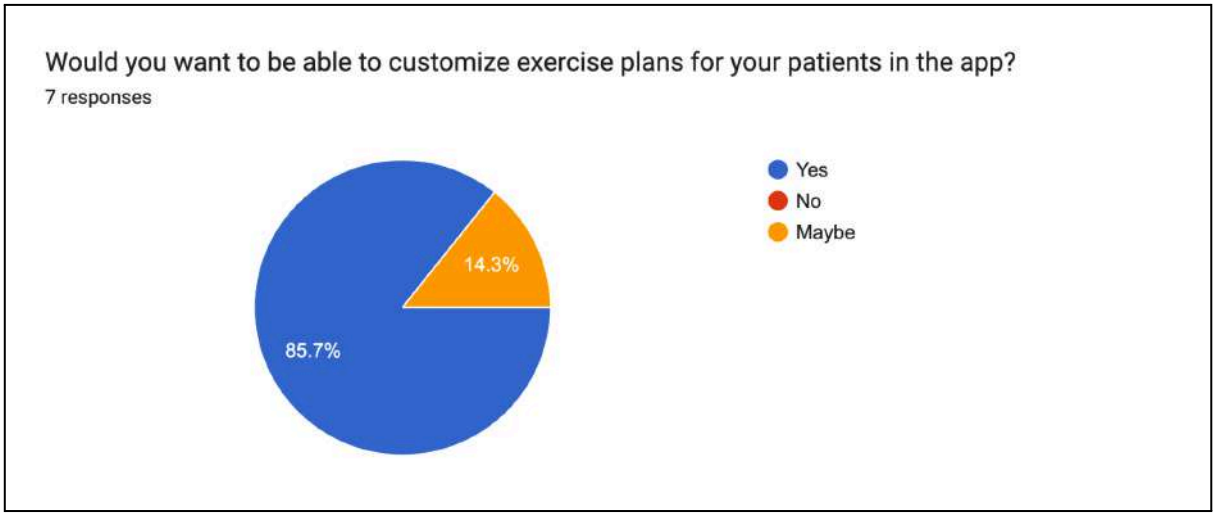


Figure 6: Survey Results - Would you want to be able to customise exercise plans for your patients in the app?

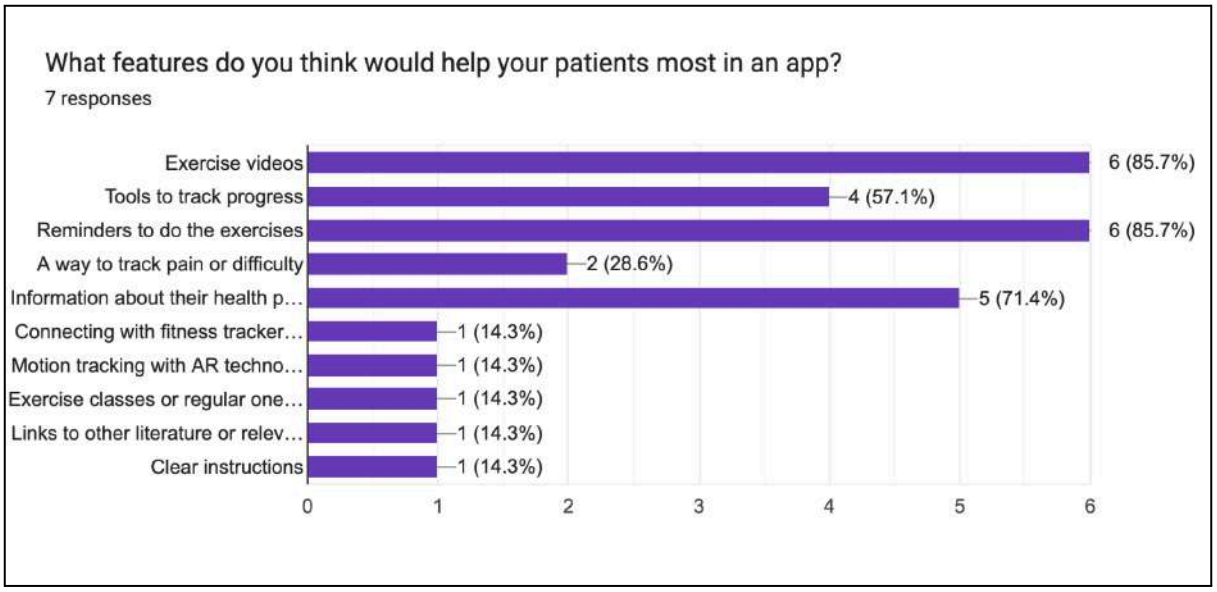


Figure 7: Survey Results - What features do you think would help your patients most in an app?

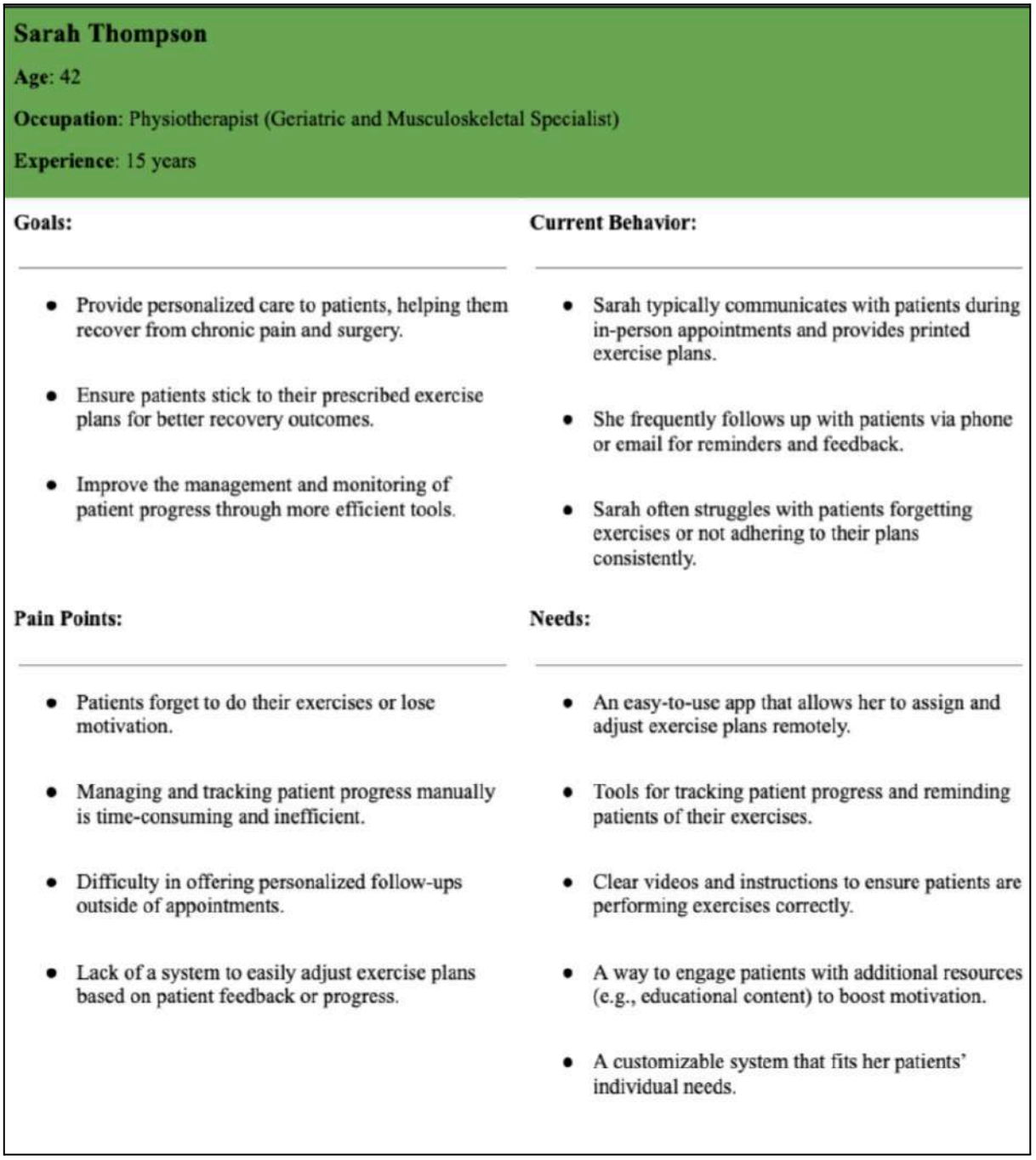
These findings highlight a clear direction for the FlexiCare app: it must prioritise intuitive, user-friendly features that promote motivation, enable personalisation, and reduce uncertainty around exercise routines. By addressing core issues such as motivation, memory, pain, and confidence in technique, the app can significantly improve patient engagement and hopefully support better outcomes in physiotherapy care.

3.1.3. Personas

Personas are fictional, but data-driven profiles are designed to represent the key user types who are likely to use a software application. The idea is to help developers better understand real users' motivations, behaviours, needs and potential frustrations. They are developed using user surveys, like that conducted above, with qualitative and quantitative

data, focused on user goals, behaviours, environment, pain points and needs. Personas serve as a powerful tool to guide design decisions and ensure that the final product remains focused on real-world usability and relevance.

The following is an illustration of the first persona, Sarah Thompson. Her profile reflects the needs and preferences of a key user group identified during the user research phase.



Sarah Thompson

Age: 42

Occupation: Physiotherapist (Geriatric and Musculoskeletal Specialist)

Experience: 15 years

Goals:

- Provide personalized care to patients, helping them recover from chronic pain and surgery.
- Ensure patients stick to their prescribed exercise plans for better recovery outcomes.
- Improve the management and monitoring of patient progress through more efficient tools.

Current Behavior:

- Sarah typically communicates with patients during in-person appointments and provides printed exercise plans.
- She frequently follows up with patients via phone or email for reminders and feedback.
- Sarah often struggles with patients forgetting exercises or not adhering to their plans consistently.

Pain Points:

- Patients forget to do their exercises or lose motivation.
- Managing and tracking patient progress manually is time-consuming and inefficient.
- Difficulty in offering personalized follow-ups outside of appointments.
- Lack of a system to easily adjust exercise plans based on patient feedback or progress.

Needs:

- An easy-to-use app that allows her to assign and adjust exercise plans remotely.
- Tools for tracking patient progress and reminding patients of their exercises.
- Clear videos and instructions to ensure patients are performing exercises correctly.
- A way to engage patients with additional resources (e.g., educational content) to boost motivation.
- A customizable system that fits her patients' individual needs.

Figure 8: Physiotherapist Sarah Thompson's Fictional Persona

While it's important to support physiotherapists like Sarah with tools to track progress and customise exercise plans, it's equally vital to meet the needs of patients like John, the second persona. Physiotherapists need efficient systems for monitoring, while patients require a simple, engaging platform that encourages regular participation and proper technique. Balancing both perspectives ensures the app supports effective care and sustained patient engagement.

John Miller

Age: 58

Occupation: Retired (Former office worker)

Condition: Chronic lower back pain due to arthritis

Goals:

- Relieve chronic pain and improve mobility through regular exercise.
- Follow the exercise plan provided by his physiotherapist to ensure long-term improvement.
- Stay motivated and track his progress in order to feel empowered in managing his health.

Current Behavior:

- John sometimes forgets to complete his exercises, especially on busy days.
- He struggles with maintaining the right form during exercises and often feels unsure if he's doing them correctly.
- He often experiences pain during exercises, which leads to frustration and occasional skipping of sessions.
- John typically uses printed instructions but finds them hard to follow without a visual guide.

Pain Points:

- Lack of motivation to complete exercises consistently, especially when he feels pain or discomfort.
- Difficulty remembering to do exercises, especially with a busy personal schedule.
- Uncertainty about performing exercises correctly, which sometimes leads to discomfort or fear of injury.
- Frustration with a lack of engagement or feedback from his physiotherapist between sessions.

Needs:

- An app that provides clear, easy-to-follow exercise videos and reminders.
- A simple system to track his progress and see improvements over time.
- Feedback on his form and how well he is doing the exercises.
- Information on how the exercises help with his specific condition, boosting motivation and understanding.
- An engaging and user-friendly platform that offers reminders and encouragement to stay consistent.

Figure 9: Patient John Miller's Fictional Persona

These personas help highlight the distinct needs, behaviours, and goals of both the physiotherapist and the patient. They will serve as a guide for designing the FlexiCare app to properly address their specific requirements.

3.1.3. Use Case Diagram

The use case diagram shown in Figure 10, based on the personas, represents the interaction between users and the system. This diagram outlines the system's key functions and how the physiotherapist and the patient interact with it. It highlights each user's various tasks, such as the physiotherapist managing patient records and

customising exercise plans. In contrast, the patient tracks progress, follows exercise instructions, and receives reminders. The diagram serves as a visual representation of how both users engage with the app to achieve their goals.

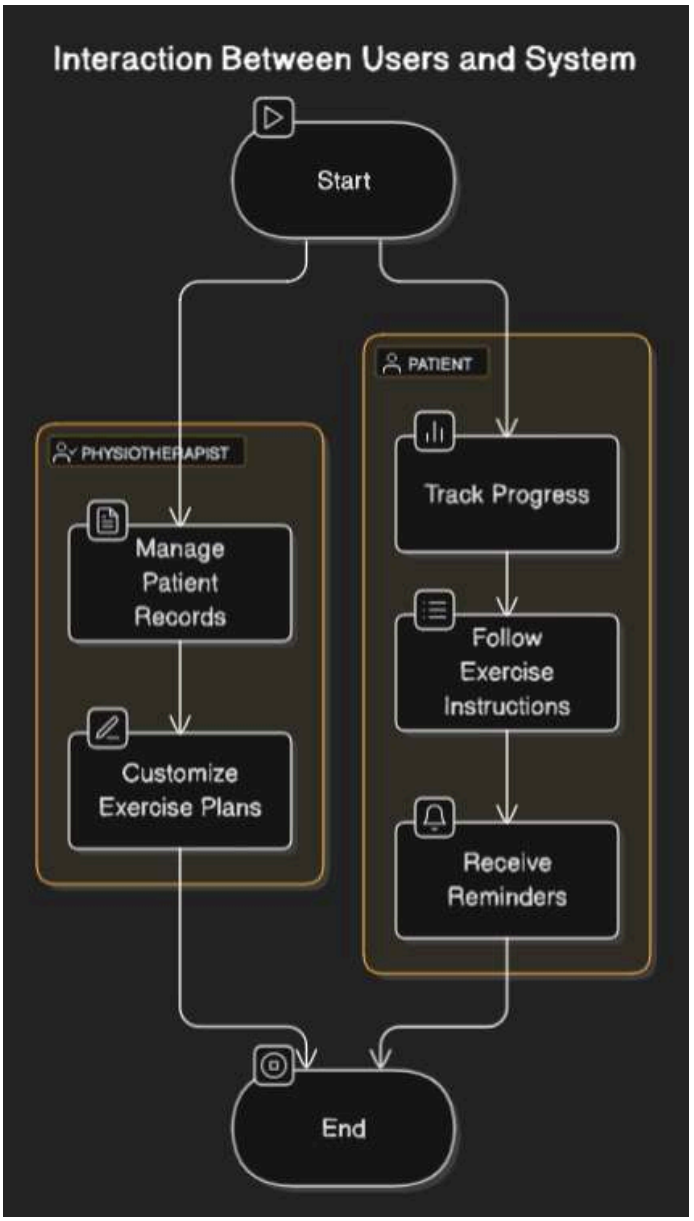


Figure 10: Use Case Diagram

3.1.4. Use Cases

The use cases represent possible scenarios or ways in which users interact with the software based on the use case diagram. These scenarios are based on the personas and are placed in typical situations where they would utilise the application. For example, a physiotherapist like Sarah might use the app to assign and adjust exercise plans for a patient, track their progress, and send reminders. On the other hand, a patient like John might use the app to receive exercise instructions, set personal goals, track his progress, and receive reminders to stay motivated. These use cases help illustrate how the application meets the needs of physiotherapists and patients in their daily interactions.

Physiotherapist Use Cases (Sarah Thompson)

Use Case 1: Assign and Customise Exercise Plan

Scenario:

Sarah must assign a personalised exercise plan to a new patient. She logs into the app and selects the patient's profile. Based on the patient's condition and progress, she customises the exercise plan, adding specific exercises and setting goals.

Steps:

1. Sarah logs into the app.

2. Sarah selects the patient from the list.

3. Sarah creates or adjusts an exercise plan based on the patient's needs.

4. Sarah assigns the exercise plan to the patient and sets progress goals.

5. Sarah can adapt the plan as needed based on patient feedback or progress.

Outcome:

The patient receives their customised exercise plan, and Sarah can track their progress remotely.

Use Case 2: Monitor and Track Patient Progress

Scenario:

Sarah wants to track how well her patients are doing with their exercise plans. She checks the app to review the progress reports, noting how often exercises are completed and whether the patient is meeting their goals.

Steps:

1. Sarah logs into the app and goes to the patient's profile.

2. She views the progress tracker, which includes completed exercises, time spent, and any adjustments made.

3. Sarah provides feedback or adjusts based on the patient's progress and performance.

4. Sarah can also send reminders or additional resources if necessary.

Outcome:

Sarah can track the patient's progress effectively and make necessary adjustments to the exercise plan.

Patient Use Cases (John Miller)

Use Case 3: Follow Exercise Instructions and Track Progress

Scenario:

John logs into the app to view his daily exercises. He watches the exercise videos and follows the instructions to ensure he performs the exercises correctly. After completing the exercises, he tracks his progress in the app.

Steps:

1. John logs into the app.

2. John views his assigned exercises for the day.

3. John watches the explicit exercise videos and follows the instructions.

4. After completing the exercises, John logs the results (e.g., number of reps, duration).

5. John checks his progress in the app to see how well he's sticking to his plan.

Outcome:

John stays on track with his exercises and can monitor his improvements.

Use Case 4: Receive Reminders and Set Goals

Scenario:

John often forgets to do his exercises. The app sends him daily reminders, encouraging him to complete his exercises. John also sets personal goals within the app to stay motivated and track his progress.

Steps:

1. John receives daily push notifications reminding him to complete his exercises.

2. John opens the app to check his exercises for the day.

3. He sets personal goals within the app (e.g., increase repetitions and reduce pain).

4. John works on his exercises, marking them as complete once done.

5. The app tracks his progress towards meeting his goals and provides motivational feedback.

Outcome:

Through reminders and goal tracking, John stays motivated and consistent with his exercises.

3.2. Functional Requirements

The functional requirements describe the core features and functions of the FlexiCare system. Derived from the survey results, personas, and case scenarios, these requirements ensure that the app meets the needs and expectations of both the physiotherapists and the patients. Functional requirements outline the elements of the user interface (UI), detailing the actions users can take and the system's responses to these interactions. For example, the UI may include features like customisable exercise plans for physiotherapists, detailed exercise videos, and reminders to ensure patients stay on track with their routines. Each feature is designed to enhance the user experience, providing efficient interaction with the app and helping users achieve their goals, whether managing patient progress or staying engaged with prescribed exercises.

This application has the following functional requirements:

- Web and Mobile App: FlexiCare will have both a web app for physiotherapists and a mobile app for patients.
- Manage Patient Profiles: Physiotherapists can create and update patient profiles, assign exercises, and track progress.
- Assign Exercises: Physiotherapists can set up and adjust exercise plans for patients.
- Track Exercises: Patients can log their completed exercises and note any pain-related issues.
- Exercise Videos: Clear exercise videos and instructions will guide patients in performing exercises correctly.
- Feedback: Physiotherapists can give feedback to patients on their performance.
- Customisable Plans: Physiotherapists can personalise exercise plans for each patient.

3.3. Non-functional requirements

The non-functional requirements focus on how the system should perform to meet user expectations. To meet the scope of this project, FlexiCare must be a responsive web application, ensuring smooth functionality across different screen sizes, devices and browsers. The database must be highly reliable, ensuring that data is stored safely and can be quickly accessed. Additionally, the login functionality must be secure, protecting user information through encryption and using previously discussed features like two-factor authentication to ensure only authorised users can log in. These requirements aim to ensure that FlexiCare remains dependable, accessible and secure for all users.

3.4. System Model

The system model for the FlexiCare application, shown in Figure 11, is divided into three layers: the client side, the server side, and the data layer.

The client-side layer manages the user interface for physiotherapists and patients, allowing interactions like viewing treatment plans, tracking exercises, and receiving feedback. For this application, this was built using React Native, which provides a seamless and responsive experience across mobile devices. NativewindCSS was utilised for styling, ensuring a clean, modern design, while Jest was used for testing, ensuring the app performs correctly on various platforms.

The server-side layer hosts the application, handles user management, processes data, and computes personalised exercise plans and progress tracking. This was developed using ASP. .NET Core, a C#-based micro-framework that simplifies web application development. ASP. .NET Core was paired with HTML views to render the physio web app interface and REST to handle data queries efficiently, allowing flexible database interaction.

The data layer consists of a database for storing patient information, exercise data, treatment history, and the recommendation model used to tailor treatment plans. For the data layer, SQLite was chosen as the relational database system. It stores all critical information, such as patient data, treatment plans, exercises, and progress tracking, offering a reliable and scalable solution for data management

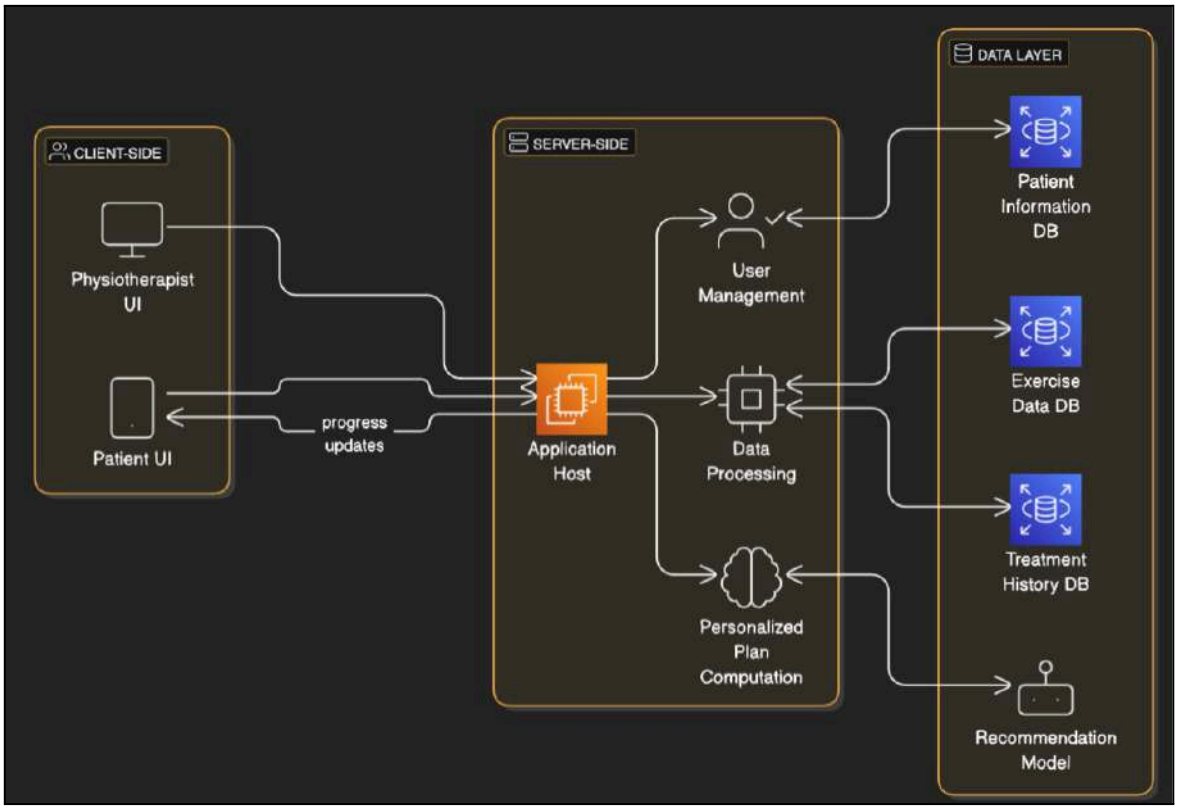


Figure 11: FlexiCare System model Diagram

During development, Git and GitHub were used for version control, enabling efficient tracking of code changes and ensuring secure backup management throughout the development process.

3.5. Feasibility Study

Developing the FlexiCare application involves addressing several potential challenges that could impact the project's success. Table 2 outlines the major challenges anticipated during development, along with descriptions and proposed solutions. Identifying these challenges early helps ensure that appropriate strategies are put in place to manage risks, maintain progress, and deliver a functional and user-friendly application.

Table 2: Assessment of Major challenges

Challenge	Description	Solution
Inexperience with C#	Limited experience in C# programming may slow development.	Use available tutorials and community support to learn and improve C# skills.
Understanding of System Design	Building a system with multiple layers (client, server, and data layers) can be complex.	Study similar systems and break the tasks into smaller, manageable parts for easier development.
Time Constraints	Limited time to complete the project may affect some features.	Plan the project well, set clear priorities, and focus on completing the most critical features first.
Data Accuracy	Ensuring the system gives accurate and reliable recommendations.	Carefully design the database and test the system regularly to improve the accuracy of recommendations.
Cold Start Problem	New users or physiotherapists may not have enough data for personalised recommendations.	Use demographic data and initial user input to create basic recommendations until more data is gathered.
User Engagement	Keeping patients engaged with their treatment plans is challenging.	Add reminders, progress tracking, and motivational feedback to keep patients motivated and involved.

3.6. Limitations

The development of the Physio web application faces some limitations. The project is being developed on consumer-grade hardware (desktop/laptop), which might limit the system's performance, particularly when handling large datasets or an expanding user base. The absence of specialised server infrastructure also poses scalability challenges, potentially affecting the system’s ability to manage increased traffic as the application grows.

Regarding the user interface (UI), the focus has been on building a basic but functional design. Both the Physio web app (developed using ASP.NET Core with HTML views) and the accompanying patient mobile app will be designed to meet essential usability requirements. More advanced features, aesthetic enhancements, or complex UI interactions will only be implemented if time allows within the project schedule.

Time constraints also represent a significant limitation for the project. Careful prioritisation of core features is necessary. Specific enhancements and additional features may be deferred to deliver a stable and fully functional product by the project deadline.

Additionally, API development will be essential for interacting with the patient's mobile app and the web-based physiotherapy management system. This may introduce some complexities in synchronising data across platforms and ensuring that the API functions correctly across different environments (web and mobile).

4. Design

This section outlines the design of the application. The objective of the design phase is to provide developers with a clear blueprint to build an application that meets the requirements outlined in the requirements section.

The design process divided into two key areas:

1. Program Design
2. User Interface Design

4.1. Program Design

4.1.1. Technologies

The FlexiCare application is built using the following two technologies, as outlined in Figure 12:

1. React Native (Mobile Application):

- Enables cross-platform compatibility for both IOS and Android with a single codebase.
- Support rapid development with a strong ecosystem and community support.
- Styled using NativewindCSS, providing a utility-first approach to responsive and customisable UI components.
- Previously used in related projects, allowing faster development through existing expertise.

2. ASP.NET Core (For web app, web server, and API):

- A new, modern framework, providing opportunities for growth and learning
- Supports both MVC (Model-View-Controller) and API architectures.
- Offers high scalability, flexibility and robust performance for web and backend systems.
- Utilises SQLite as the default database, simplifying setup and focusing efforts on application logic
- HTML files are used as part of the MVC views within ASP.NET Core to render content and display the web app interface.

This combination of technologies ensures efficient development and seamless integration across both mobile and web platforms, with a clean and maintainable codebase.

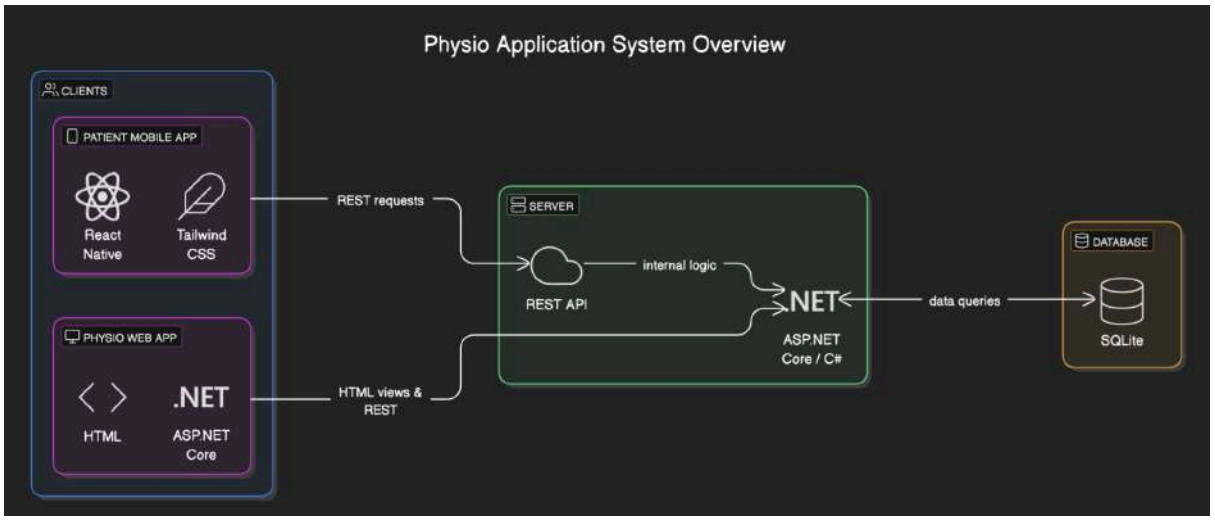


Figure 12: FlexiCare System architecture and technologies

4.1.2. Structure of Technologies

1. React Native (Mobile Application)

React Native is organized into a modular structure:

- **/node_modules:** Project dependencies.
- **/assets:** Static files (e.g., images, illustrations & icons).
- **/components:** Reusable UI components.
- **/constants:** Reusable constant values for icons
- **/app:** Contains the pages/views of the app.
    - **/(protected):** The pages of the app that require Authentication
        - **/(tabs):** Contains pages which can be navigated to using the TabBar
    - **/account:** Login & onboarding pages can be viewed without Auth
    - **/_layout:** Root layout file where AuthContext is in
- **/services:** Helper functions & API functions
- **/interfaces:** Typescript type interfaces for API models
- **context:** Contains the Auth context used to maintain Auth state
- **tailwind.config.js:** Configuration for NativewindCSS - based on TailwindCSS.

2. ASP.NET Core Structure (For Web App, Web Server, and API)

The ASP.NET Core solution is divided into two main area (MVC and API) within a single folder.

- **Controllers:** Handles HTTP requests (MVC returns views, and API returns data)
- **Models:** Shared data structures across MVC and API.
- **Services:** Business logic used by both projects.
- **Data:** Contains the database context.
- **Views:** Only in the MVC project, contains Razor views for UI.
- **wwwroot:** Only in the API project, status assets (JavaScript, CSS, images).

4.1.3. Design Patterns

1. Model-View-Controller (MVC)

Within the ASP.NET Core web application the MVC pattern is implemented:

- **Model:** Handles the data and business logic.
- **View:** Displays the user interface (UI) using Razor views (HTML + C#).
- **Controller:** Manages user input, updates the model, and returns the view.

This structure keeps the app organised and easy to maintain.

2. Client-Server (API)

The API project follows the Client-Server approach:

- **Client:** The React Native mobile app makes requests to the server.
- **Server:** The API processes these requests and responds, typically with JSON.

This allows the mobile app to interact with the backend.

3. Object-Oriented Programming (OOP)

Both the MVC and API parts of the application use Object-Oriented Programming (OOP) principles:

- **Classes and Objects:** Core parts of the system such as the Patient, Exercise, and Appointment are represented as classes.
- **Encapsulation:** Data and methods are bundled together inside classes, hiding complexity and making the code more manageable.
- **Inheritance:** Code reuse is achieved by allowing one class to inherit properties and methods from another. For example, the BaseController class sets standard

navigation behaviour, and all other controllers inherit from it, automatically applying consistent behaviour across the application.

- **Polymorphism**: Each controller inherits functionality from the ASP.NET controller class, while also implementing specific actions on top of this as required by each individual controller class.

OOP helps in building a modular, reusable, and easy-to-manage codebase.

4.1.4 Application Architecture

The application has a layered architecture, as shown in Figure 13, dividing the mobile and web apps into clear sections for better organisation and scalability.

1. ASP.NET Core Architecture (Web App)

- **MVC (Model-View-Controller):** Separates concerns into Models (data), Views (UI), and Controllers (logic).
- **API Layer: Handles** RESTful API requests for data exchange with the mobile app.
- **Database Layer:** Uses SQLite, the default database in ASP.NET Core, for local storage and database interaction. Entity Framework Core is used to interact with database.

2. React Native Architecture (Mobile App)

- **UI Layer:** Built with reusable components and screens (e.g., Home, Profile).
- **Navigation Layer:** Managed with React Navigation for screen transitions (e.g., stack, tab navigation).
- **State Management:** Manages app state with Context API.
- **API Layer:** Communicates with the backend using Axios or Fetch for API requests.
- **Styling:** Uses NativewindCSS or custom styles for responsive UI design.

This architecture ensures the application is modular, easy to maintain, and scalable for future development.

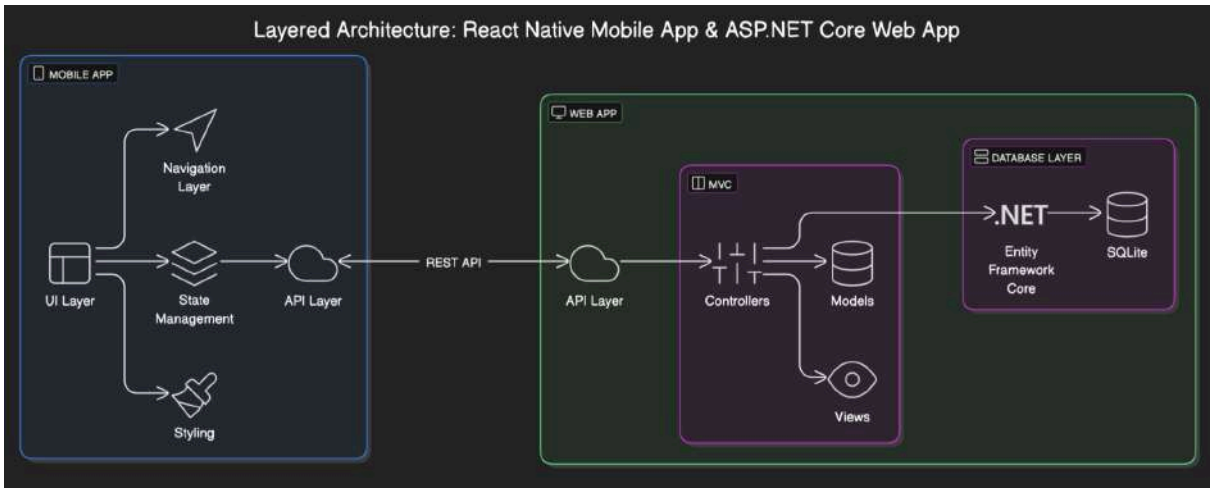


Figure 13 : Application Architecture Diagram

4.1.5 Database Design

The database schema is designed to handle the relationships between physiotherapists, patients, treatments, exercises, and appointments.

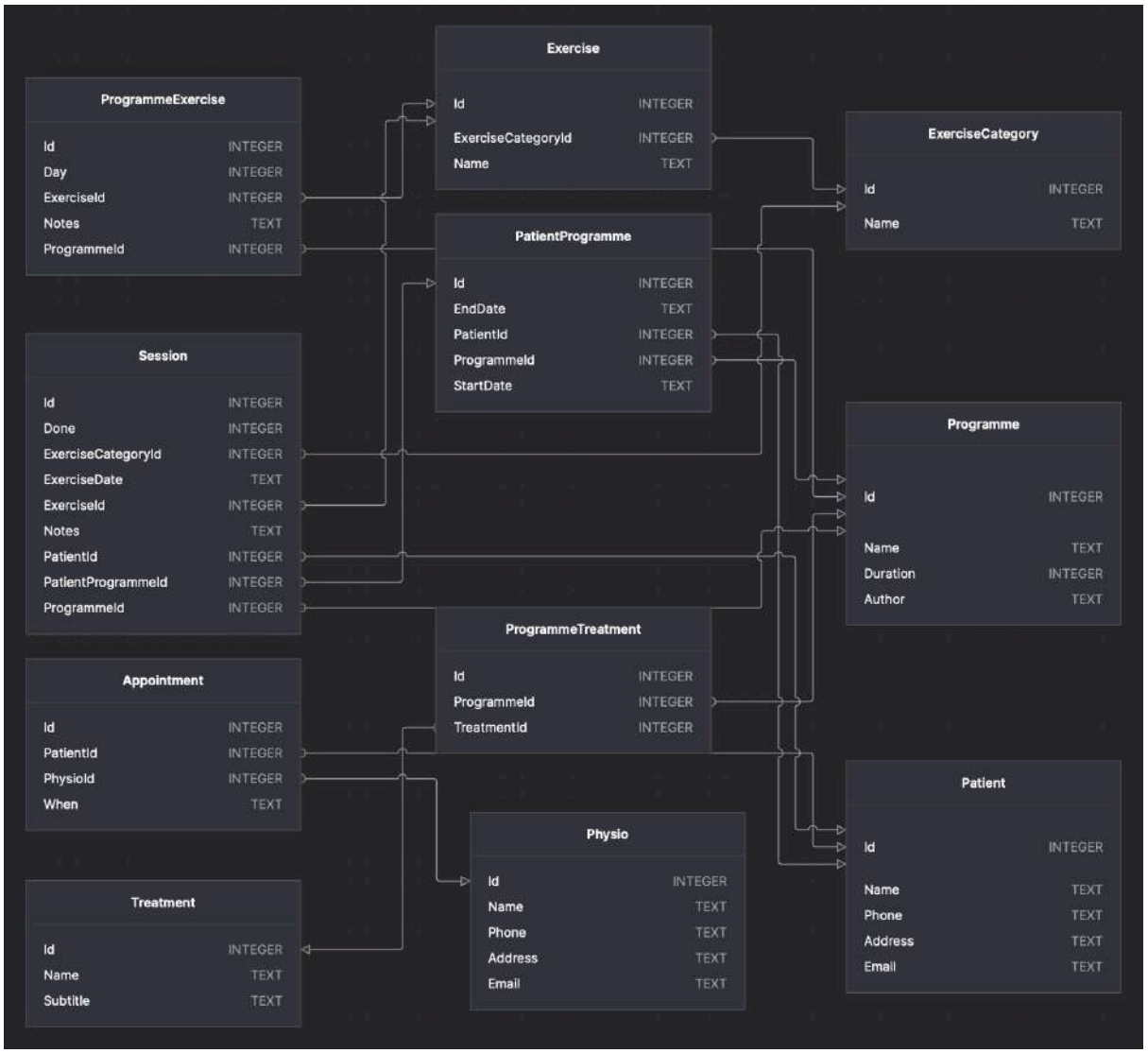


Figure 14 : Entity Relationship diagram

The Entity Relationship Diagram (ERD), shown in Figure 14. for the FlexiCare system highlights how key tables such as Physios, Users, Treatments, Programs, Exercises and Appointments are interconnected. Each table is connected using foreign keys, helping to keep the system organised and allowing the application to efficiently manage patient records, treatment plans, exercise schedules, and appointments.

- The Physios table stores information about physiotherapists. The physio_id acts as a primary key and is used in both the Users table (to link each patient to their physiotherapist) and the Programs table (to assign physiotherapists to patient programs).
- The Users table stores patient details. Each patient is linked to a physiotherapist through the physio_id foreign key.
- The Treatments table lists available treatment options. Each treatment can have one or more related exercises, which are stored in the Exercises table.
- The Programs table records the rehabilitation programs assigned to patients, and connects patients, physiotherapists, and exercises together.
- The Appointments table manages scheduling between patients and physiotherapists, storing information about upcoming and past sessions.

4.2. User Interface Design

This section outlines the planned user interface (UI) design for the FlexiCare mobile application. This initial version will use this design as a basic frame, and the mobile application UI will be subsequently developed based on this concept. These designs will follow a user-centered approach, aiming to deliver a clear, intuitive experience that supports patient engagement and effective treatment tracking.

4.2.1. UI Inspiration

Figure 15: Metrics Bar Charts [14]

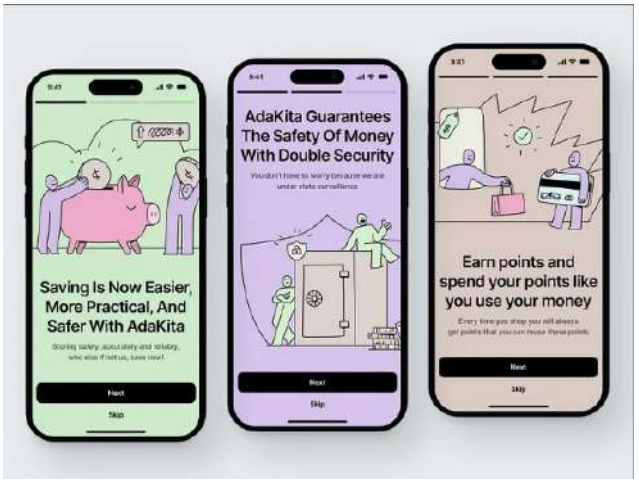


Figure 16: Modern Onboarding Design [17]

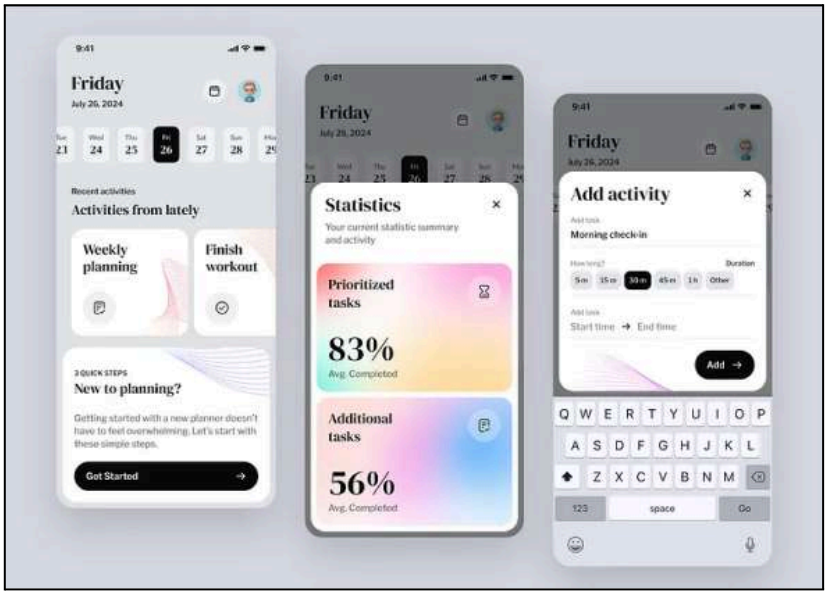


Figure 17: Home Page Statistics [15]

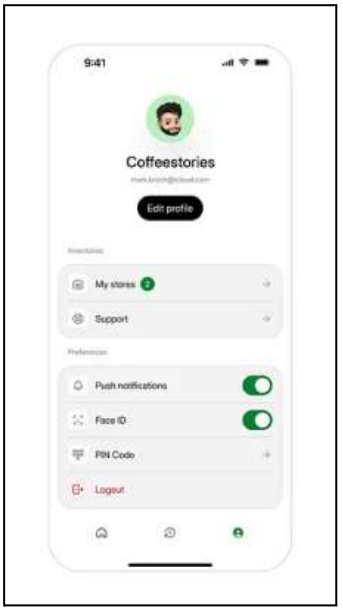


Figure 18: Account Settings [16]

Figures 15-18 show app designs from Dribbble that were used for inspiration. These designs helped shape the look and feel of the project, giving ideas for layout, color schemes, and user interface elements.

4..2.2. Wireframes

Figure 19: Homepage design

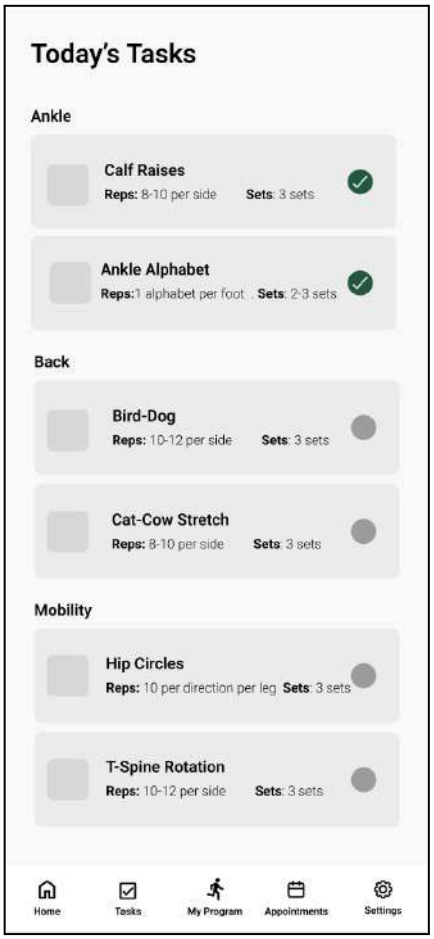


Figure 20: Today's Tasks design

Homepage

The homepage, shown in Figure 15, provides a snapshot of the user's treatment progress, displaying key statistics in a visually engaging format. It includes graphs showing pain levels and the number of reps completed over time. The page also provides stats on the total number of exercises completed and calculates the average pain level. This overview gives patients insight into their progress.

Today's Tasks

The "Today's Tasks" page, shown in Figure 16, lists the exercises that the user needs to complete for the day. The exercises are categorised by the specific treatment program, allowing users to quickly identify which exercises are part of each treatment. The page is designed to provide users with a clear, organised view of what is expected of them, with a simple layout that highlights the exercises they need to focus on today.

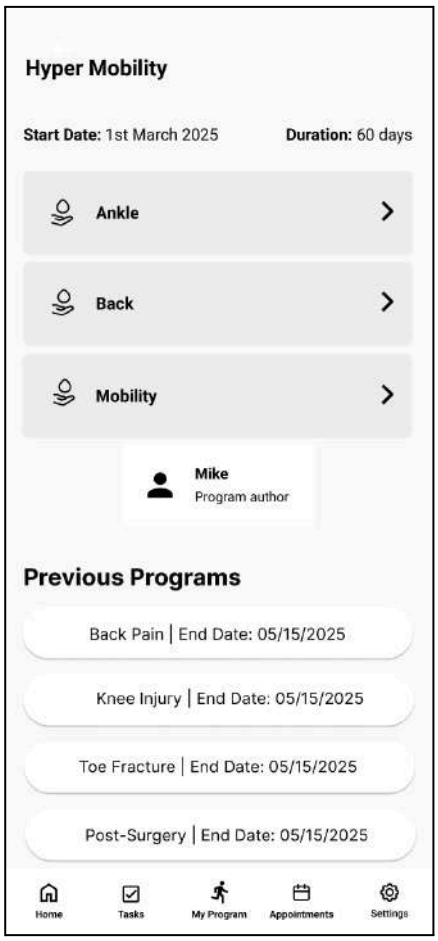


Figure 21: My Program design

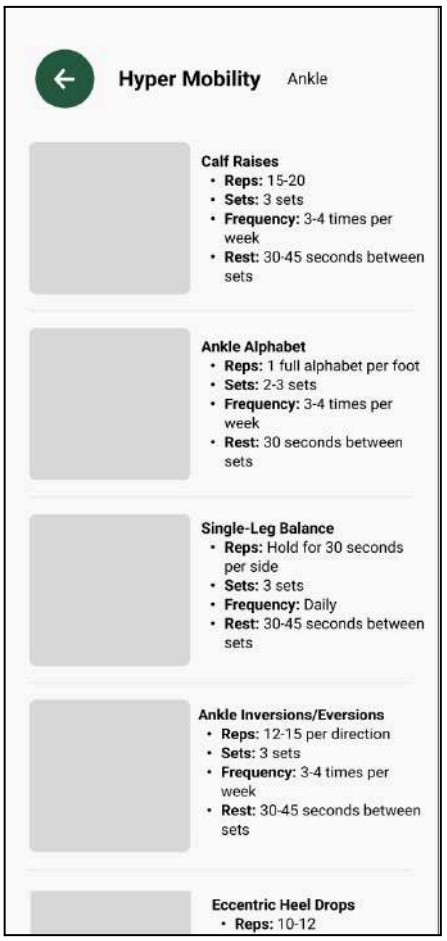


Figure 22: Treatment Exercise design

My Program

The "My Program" page, shown in Figure 17, provides detailed information about the user's current program, including an overview of the treatment plan, start and end dates, and specific goals. The page also includes a list of treatments being followed and a history of previous programs, offering users a clear view of their ongoing and past treatments. This helps patients track their progress and stay on top of their recovery journey.

Treatment Exercises

When the user selects a treatment from the “My Program” page, they are directed to the “Treatment Exercises” page, shown in Figure 18. This page lists all the exercises included in that specific treatment. Each exercise is presented with basic details such as the name, number of reps and sets, and any relevant notes or instructions. The layout ensures users can easily navigate through the exercises to find what they need to complete.

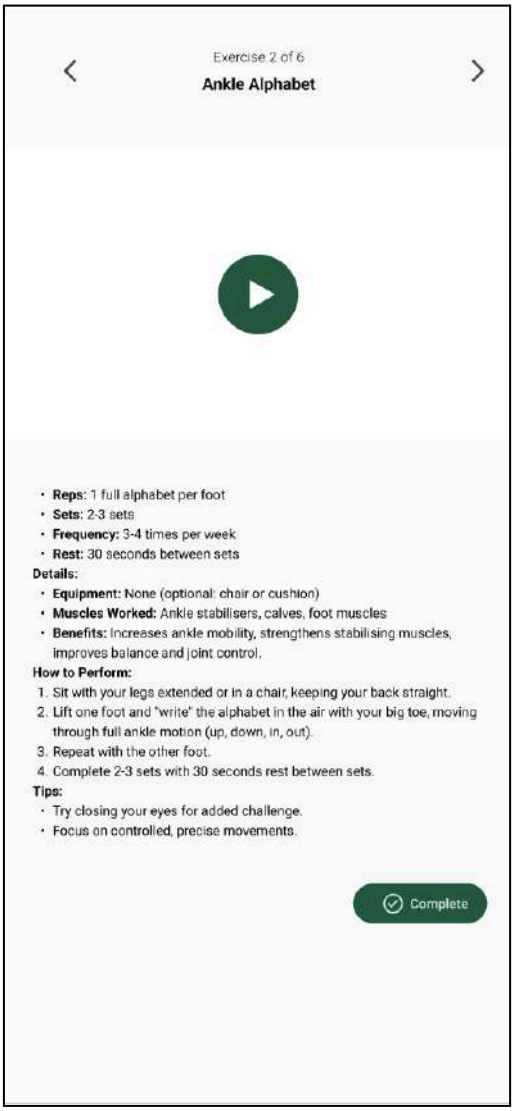


Figure 23: Individual Exercise design

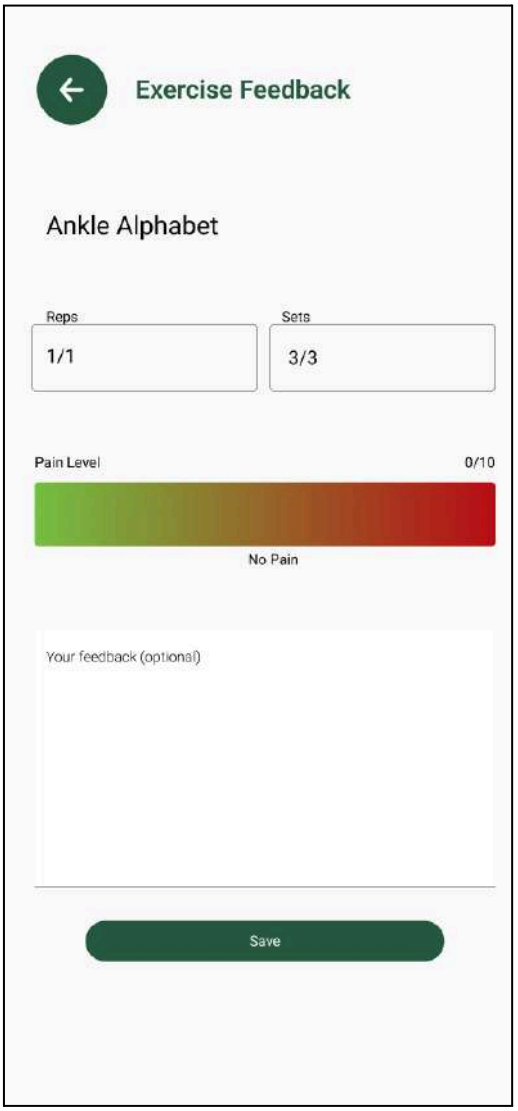


Figure 24: Feedback design

Individual Exercise

The “Individual Exercise” page, shown in Figure 19, provides in-depth information about a specific exercise, including a detailed description, equipment required, and step-by-step instructions. A button to mark the exercise as complete allows users to track their progress. This page is designed to provide users with all the information they need to perform the exercise correctly and stay on track with their treatment program.

Feedback

The “Feedback” page, shown in Figure 20, allows users to provide details on the exercise they just completed. Users can enter the number of reps and sets done, along with their pain level and any additional notes they might have. This feedback is essential for physiotherapists to assess the user’s progress and make any necessary adjustments to the treatment plan. It also helps users reflect on their experience with the exercises.

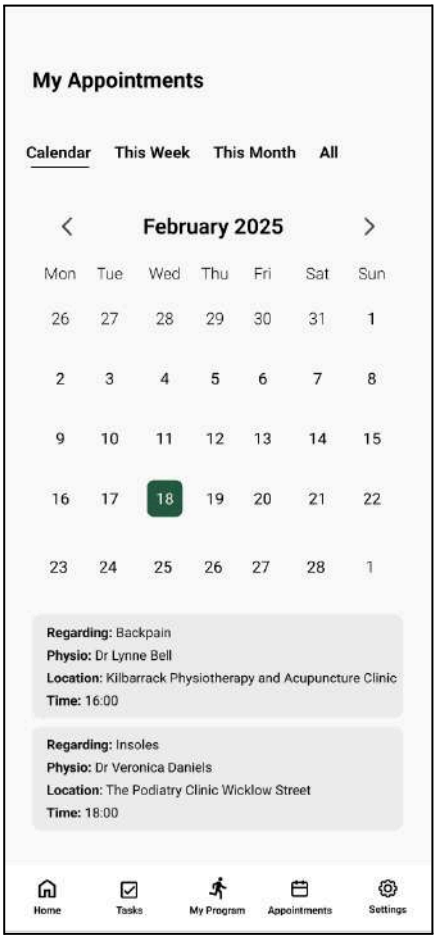


Figure 25: Appointments designs

Figure 26: Settings design

Appointments

The “Appointments” page presents a calendar view, where users can see their upcoming appointments and sessions. By clicking on a specific day, the user can view a detailed list of appointments scheduled for that day. This feature helps users stay organised by providing a clear, easy-to-read schedule of all their physiotherapy sessions and ensuring they never miss an appointment.

Utilities

The "Utilities" section of the application includes critical features such as the Login, Register, and Settings pages, which are vital for user account management. The Login

page provides users with secure access to their personal profiles, while the Register page allows new users to create an account, entering the necessary information to get started. The Settings page enables users to update their personal information, adjust notification preferences, and modify other aspects of their account, ensuring they have full control over their experience. These utilities provide essential functionalities for managing user accounts and ensuring smooth interaction with the app.

4.2.3. Refined Design Language

The wireframes have been updated to make the design easier to use and navigate. The layout is cleaner, with fewer distractions, so users can find what they need more quickly. Buttons and menus are more transparent and straightforward, making it easier for users to complete tasks with less effort. These changes help make the design more intuitive and user-friendly.

The colour scheme has also been slightly changed to improve the look and make things easier to read. The new colours help text stand out and make buttons more noticeable. These changes were made to improve accessibility for all users, including those with different vision levels. The spacing and arrangement of elements have been enhanced to make the design feel more balanced and pleasant. These updates focus on creating a better user experience while keeping the design consistent with the brand.

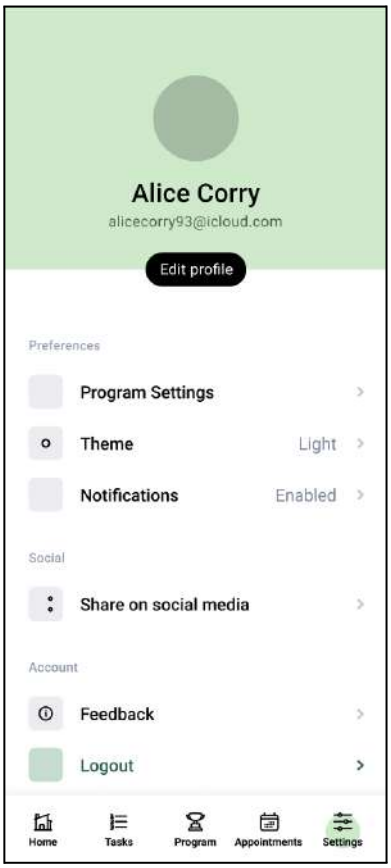


Figure 27: Settings designs

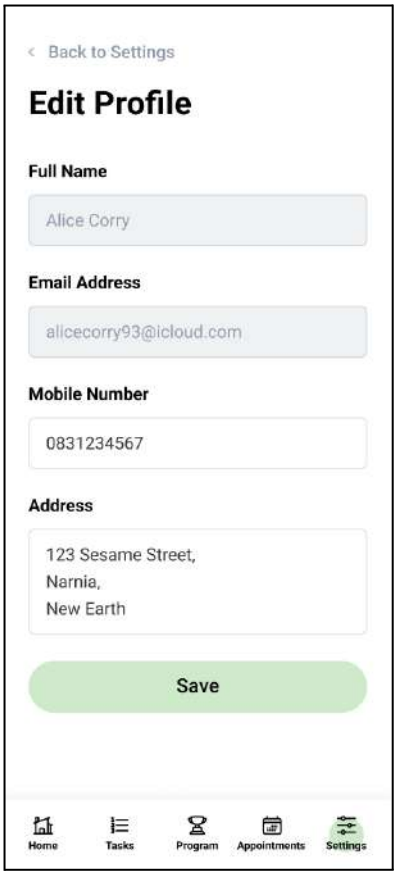


Figure 28: Edit profile design

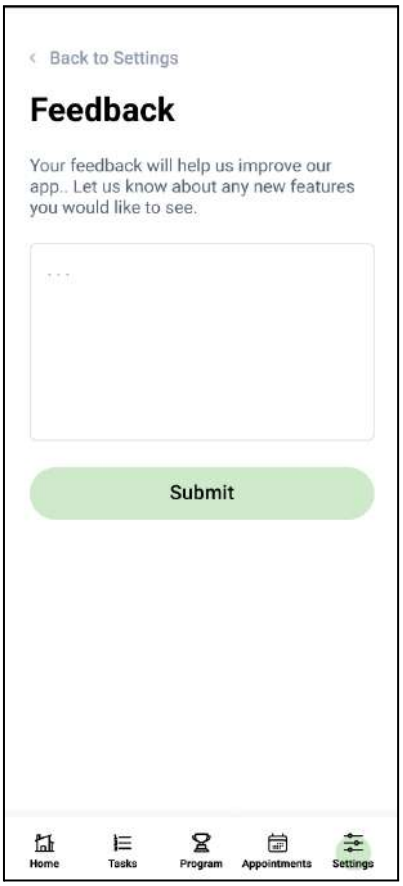


Figure 29: Feedback designs

Figure 30: Sign in design

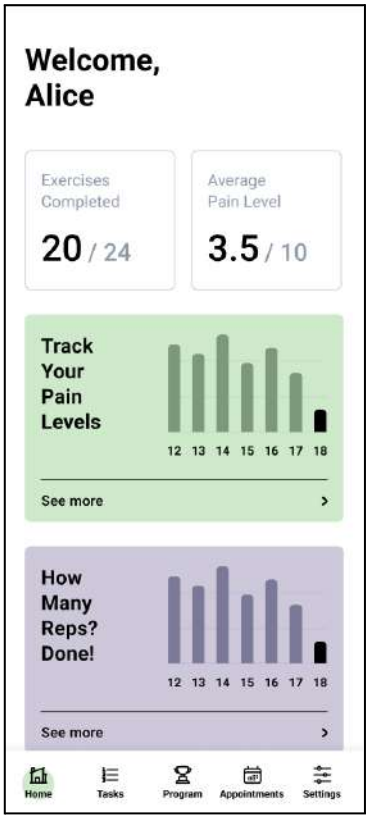


Figure 31: Dashboard designs

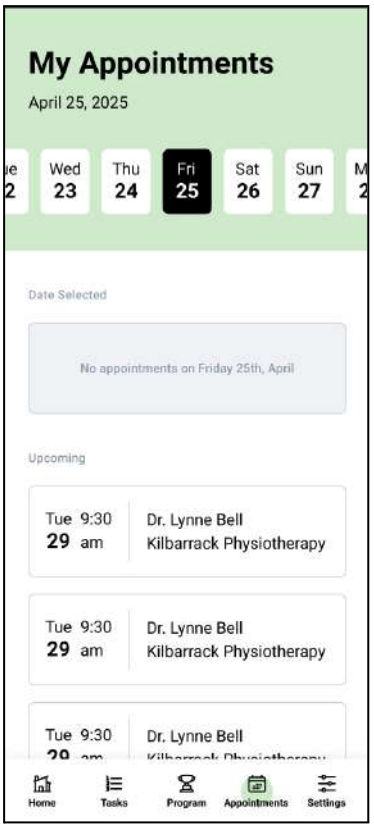
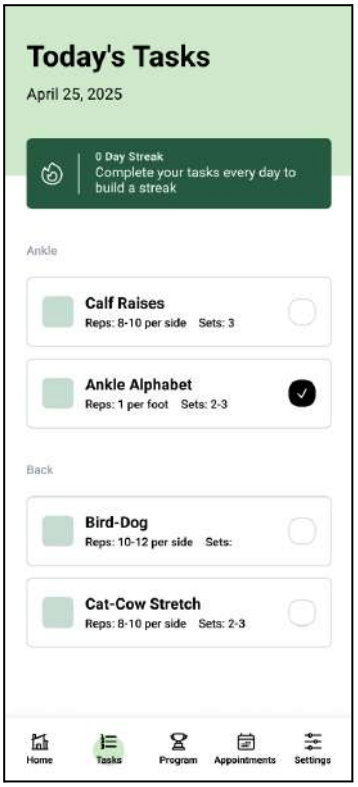
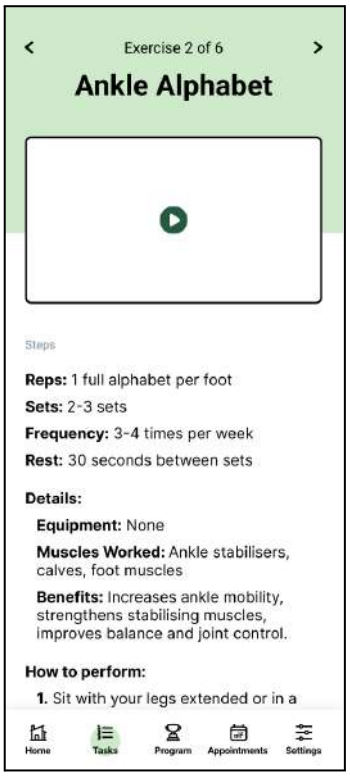


Figure 32: Sign in design





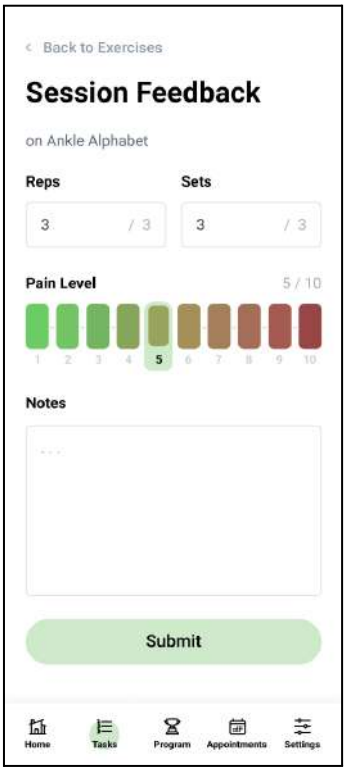


Figure 33: Tasks designs

Figure 34: Exercise design

Figure 34: Session feedback design

5. Implementation

This chapter discusses the implementation of the individual components of the application. While the previous chapter described the system's design, this section focuses on the development of the Web Server, API Server and Mobile Application.

5.1. SCRUM Methodology

A simplified version of the SCRUM methodology was used to manage the development process. The project was broken down into smaller, manageable tasks and completed over short development cycles, known as sprints.

Each sprint was focused on a specific area of the system:

- Sprint 1: Initial development attempt using Python.
- Sprint 2: Development of the backend infrastructure using C#, including the FlexiCare manager and API.
- Sprint 3: Development of the mobile frontend using Javascript, focusing on user interface design and client-side functionality.

The benefit of the SCRUM principles was that at the end of each sprint, it was possible to review the progress of and adjust the project priorities, such as timeline, to any challenges identified. .

5.2. Development Environment

Development was primarily carried out for this project using Visual Studio Code (VS Code) as the integrated development environment (IDE). This was selected for its flexible, extensive extension possibilities, and pre-configured tools for debugging, linting and version control.

During Sprint 1, for the Python-specific development, PyCharm was utilised to take advantage of its Python-specific features. Similarly, when starting working with C#, JetBrains Rider was used for a short time, as it provided helpful tools for .NET development. However, VS Code was eventually returned to, as it offered a more unified and comfortable environment for working across different parts of the project.

For any database-related development and testing, TablePlus was used, which provided a user-friendly interface for managing databases and running SQL queries efficiently.

Overall, the combination of these tools supported a smooth and productive development workflow throughout the project.

5.3. Sprint 1: Python Development

5.3.1. Early Attempt

Development was first started using C#, but there was also interest in trying out Python's Flask and GraphQL. JetBrains PyCharm was chosen as the development environment. Work during this spring was focused on building the API, and no effort was put into creating user interfaces or using MVC patterns yet. The goal was to understand how Flask and GraphQL could be used to handle backend tasks like routing, data handling, and queries.

First, the Flask project was set up and connected to GraphQL. A PostgreSQL database was also configured for use with the project. The database tables were created, and a connection between the app and the database was established. Models were created to represent the data needed by the application, such as users and authentication information. Seeders were written to fill the database with sample data

name	schema	kind	owner	estimated_row	total_size	data_size	index_size
appointments	public	TABLE	alicecorry	-1	8 KB	—	8 KB
exercises	public	TABLE	alicecorry	-1	16 KB	8 KB	8 KB
feedback	public	TABLE	alicecorry	-1	8 KB	—	8 KB
patients	public	TABLE	alicecorry	-1	16 KB	8 KB	8 KB
physios	public	TABLE	alicecorry	-1	16 KB	8 KB	8 KB
programs	public	TABLE	alicecorry	-1	16 KB	8 KB	8 KB
roles	public	TABLE	alicecorry	-1	8 KB	—	8 KB
sessions	public	TABLE	alicecorry	-1	8 KB	—	8 KB
tasks	public	TABLE	alicecorry	-1	8 KB	—	8 KB
treatments	public	TABLE	alicecorry	-1	8 KB	—	8 KB
users	public	TABLE	alicecorry	-1	16 KB	8 KB	8 KB

Figure 23: Flask Database in TablePlus

Next, GraphQL schemas were created to define the types of data that the API would use. These schemas were connected to the database models so that the data could be queried. Queries were added to fetch data, such as retrieving a user profile or getting a list of all users. Three key mutations were then created: register, login, and logout. The register mutation allowed new users to be created by accepting their emails, hashing their passwords, and saving the information in the database. The login mutation was used to authenticate users, while the logout mutation was created to log users out and invalidate their sessions.

5.3.2. Reflection of Sprint 1

After some development, it was decided that Flask and GraphQL were not the best fit for a large, long-term project. Because of this, the backend development was switched back to C#. Even though these tools were not continued, the experience was useful. A better understanding of how they work was gained, and it is likely that they will be used again in future, smaller projects.

5.4. Sprint 2: Backend Development

The backend is the part of an application that users do not directly see but is essential for making everything function properly. It manages the logic, database interaction and the communication between the database and the client interfaces (such as mobile or web apps). For FlexiCare, the backend is responsible for securely handling user data and storing important information such as fitness activities and exercise routines. Ensuring that both the physiotherapists and patients have a smooth and reliable experience when using the FlexiCare system.

Sprint 2 will outline the development of FlexiCares backend, outlining the key technology used and the structure of the system.

As previously mentioned, ASP.NET Core MVC, a powerful framework developed by Microsoft, will be utilised in this project. As seen in Figure 24, this framework is broken down into four projects: FlexiCareManger, FlexiCareManagerTest, FlexiCareAPITest and FlexiCareAPI. In addition, to make backend development more efficient and to extend the capability of C#, several essential extensions, shown in Figure 25, were added to the project.

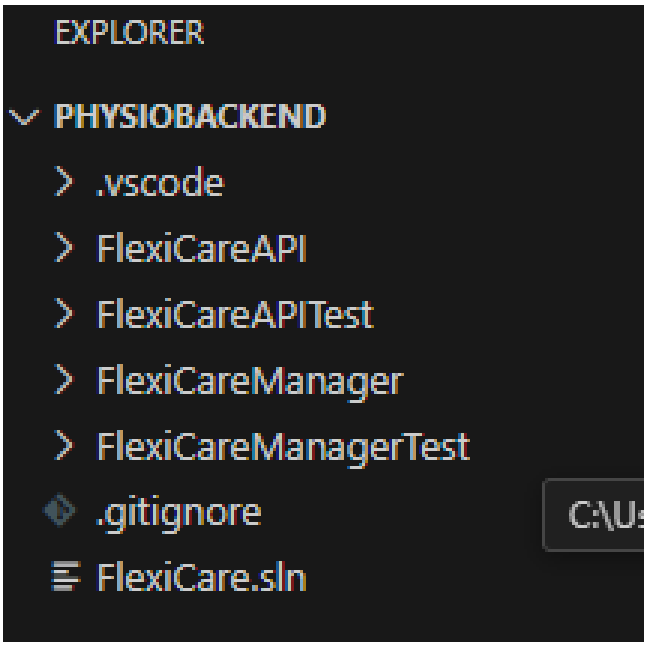


Figure 24: FlexiCare ASP.NET Core MVC *Projects*

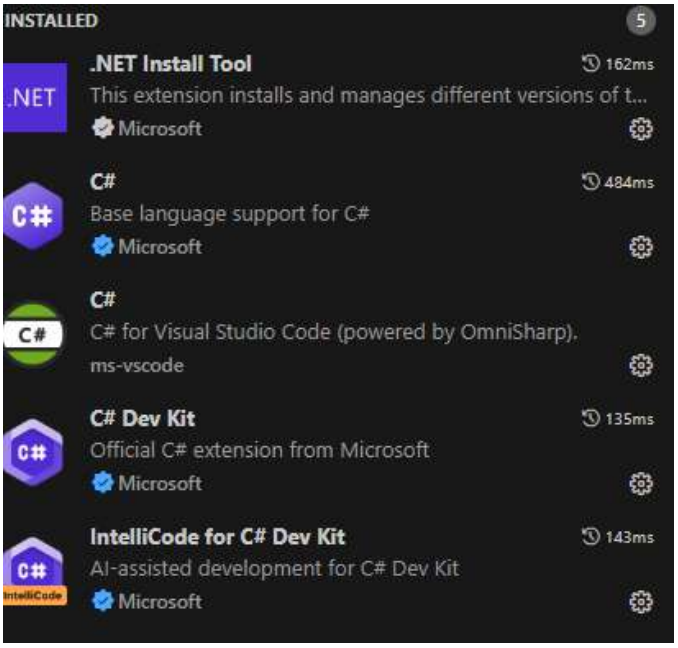


Figure 25: FlexiCare C# Extensions

In the C# Language package management is handled by NuGet, which works similarly to npm (Node Package Manager) in the JavaScript world. Using simple terminal commands, packages can be added directly to the project. When the project is built, these packages are automatically downloaded and available for the application. These include:

- dotnet add package Microsoft.EntityFrameworkCore.Design
- dotnet add package Microsoft.EntityFrameworkCore.SQLite
- dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design
- dotnet add package Microsoft.EntityFrameworkCore.SqlServer
- dotnet add package Microsoft.EntityFrameworkCore.Tools

5.4.1 FlexiCare Manager

The FlexiCare Manager project forms the core of the backend system. It is responsible for:

- Setting up the data model and database structure.
- Providing physiotherapists with the ability to manage patients, design personalised programmes, and schedule appointments.
- Allowing practice managers to control the system, including adding new exercises, registering new physiotherapists, and managing users.

This backend ensures that the physiotherapy workflows are streamlined and all administrative tasks are handled efficiently.

To begin, Dotnet provides a command that can create a shell project containing a blank folder for Models, Views, Controllers, etc. For example: “dotnet new mvc -o FlexCareManager” would create a structure similar to that seen in Figure 26.

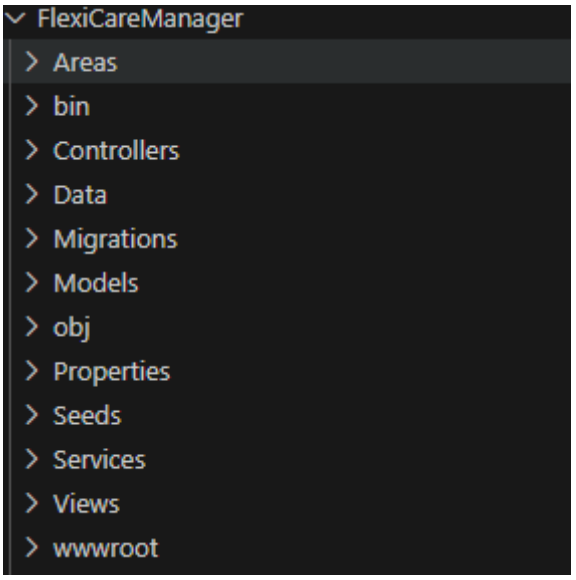


Figure 26: Example of FlexiCareManager Shell Folder

Data Model

The first step in developing the FlexiCare manager was to create the data model. This involved designing C# classes that represent the main entities of the system (e.g., Patient, Physio, Appointment, Exercise), as seen in Figure 27. These classes were later used to build the database. Relationships between entities were also established. For example, the Appointment entity has relationships with both Patient and Physio entities.

```
public class Physio
{
    10 references
    public int Id { get; set; }
    [Required]
    26 references
    public string? Name { get; set; }
    [Required]
    22 references
    public string? Phone  { get; set; }
    [Required]
    24 references
    public string? Address  { get; set; }
    [Display(Name = "Email address")]
    [Required(ErrorMessage = "The email address is required")]
    [EmailAddress(ErrorMessage = "Invalid Email Address")]
    2 references
    public string? Email { get; set; }
}
```

```
public class Appointment
{
    10 references
    public int Id { get; set; }
    [Display(Name = "Patient")]
    [Required]
    9 references
    public int? PatientId { get; set; }
    [Display(Name = "Patient")]
    9 references
    public virtual Patient? Patient { get; set; }
    15 references
    public DateTime? When  { get; set; }
    [Display(Name = "Physio")]
    [Required]
    8 references
    public int? PhysioId {get; set;}
    [Display(Name = "Physio")]
    13 references
    public virtual Physio? Physio  { get; set; }
}
```

Figure 27: Examples of C# Classes

Object-Relational Mapping

To map these classes to database tables, Entity Framework Core, Microsoft's Object-Relational Mapping (ORM) tool, was used. The mapping process was managed using a DbContext class, which specifies which class should be linked to which database table. This approach allows normal C# statements to easily access the database. For example, the statement shown in Figure 28 does the following:

- Accesses the Exercise database table
- Where the ID column matches my local ID field
- Links to the ExerciseCategory table for the ExerciseCategory linked to this Exercise
- Populates the result into a local variable exercise

```
var exercise = await _context.Exercise
    .Include(e => e.ExerciseCategory)
    .FirstOrDefaultAsync(e => e.Id == id);
```

Figure 28: Example C# Statement for Accessing the Database

Migration Building

Once the base data model was sufficiently developed (it is not necessary for it to be one hundred percent final initially), the entity framework was used to create a migration. A migration generates code to create or update the database based on the current model, such as in Figure 29. This process was relatively straightforward as the generated migration code could be reviewed if needed. However, in practice it tended to be accurate and did not require manual editing. This does not mean that the original model was correct. It was necessary to adjust the initial model, create another migration and update the model database again. For example, when the appointment model was added, the following commands were ran:

- dotnet ef migrations add Appointment
- dotnet ef database update

```
protected override void Up(MigrationBuilder migrationBuilder)
{
    migrationBuilder.CreateTable(
        name: "Client",
        columns: table => new
        {
            Id = table.Column<int>(type: "INTEGER", nullable: false)
                .Annotation("Sqlite:Autoincrement", true),
            Name = table.Column<string>(type: "TEXT", nullable: true),
            Phone = table.Column<string>(type: "TEXT", nullable: true),
            Address = table.Column<string>(type: "TEXT", nullable: true)
        },
        constraints: table =>
        {
            table.PrimaryKey("PK_Client", x => x.Id);
        });

    migrationBuilder.CreateTable(
        name: "ExerciseCategory",
        columns: table => new
        {
            Id = table.Column<int>(type: "INTEGER", nullable: false)
                .Annotation("Sqlite:Autoincrement", true),
            Name = table.Column<string>(type: "TEXT", nullable: true)
        },
        constraints: table =>
        {
            table.PrimaryKey("PK_ExerciseCategory", x => x.Id);
        });
```

Figure 29: Example of Migration Builder

Seeding

To make development and testing easier, seed data was added to the database. Seeding involved writing code that:

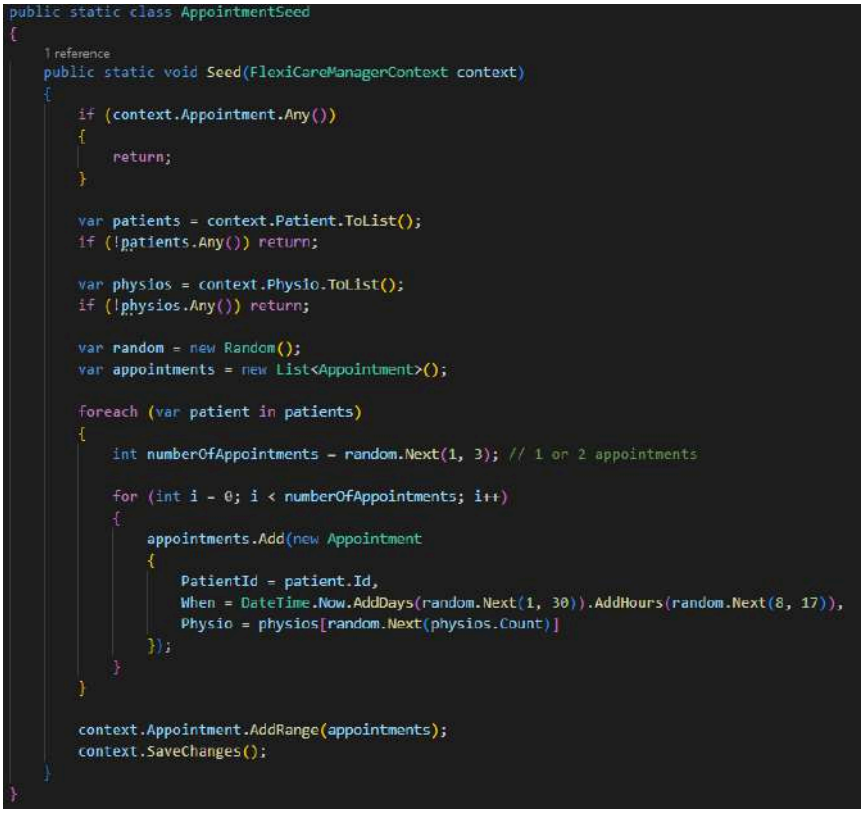
- Checked if specific tables (e.g. Patient) were empty.
- Inserted sample test data if no records were found.

An example of the patient seed can be seen in Figure 31.

```
public static class PatientSeed
{
    1 reference
    public static void Seed(FlexiCareManagerContext context)
    {
        if (context.Patient.Any())
        {
            return;
        }

        var patients = new List<Patient>
        {
            new Patient { Name = "Alice Walker", Phone = "555-100-2001", Address = "12 Garden Lane, Roseville", AllowNotifications=true },
            new Patient { Name = "Ben Carter", Phone = "555-100-2002", Address = "34 Park Ave, Hilltown", AllowNotifications=false },
            new Patient { Name = "Catherine Lee", Phone = "555-100-2003", Address = "56 River Street, Lakeview", AllowNotifications=true },
            new Patient { Name = "Daniel Wright", Phone = "555-100-2004", Address = "78 Sunset Blvd, Westport", AllowNotifications=false },
            new Patient { Name = "Ella Johnson", Phone = "555-100-2005", Address = "90 Ocean Drive, Rivertown", AllowNotifications=false },
            new Patient { Name = "Frank Lin", Phone = "555-100-2006", Address = "21 Birch Road, Northside", AllowNotifications=true },
            new Patient { Name = "Grace Kim", Phone = "555-100-2007", Address = "43 Maple Crescent, Brookfield", AllowNotifications=true },
            new Patient { Name = "Henry Adams", Phone = "555-100-2008", Address = "65 Cedar Street, Springfield", AllowNotifications=false },
            new Patient { Name = "Isabella Nguyen", Phone = "555-100-2009", Address = "87 Elm Avenue, Eastview", AllowNotifications=true },
            new Patient { Name = "Jack Morales", Phone = "555-100-2010", Address = "109 Pine Hill Road, Southwood", AllowNotifications=false },
        };
        foreach (var patient in patients)
        {
            patient.Email = patient.Name!.Replace(' ', '.').ToLower() + "@patient.ie";
        }
        context.Patient.AddRange(patients);
        context.SaveChanges();
    }
}
```

Figure 31: Example of Patient Seed Code

Seeding also allowed for creating relationships between tables (such as linking patients with appointments) and inserting random values to simulate real ones. An example of this can be seen in Figure 32. This approach made it easy to reset the database and quickly repopulate it without manually entering test data each time.



```
public static class AppointmentSeed
{
    1 reference
    public static void Seed(FlexiCareManagerContext context)
    {
        if (context.Appointment.Any())
        {
            return;
        }

        var patients = context.Patient.ToList();
        if (!patients.Any()) return;

        var physios = context.Physio.ToList();
        if (!physios.Any()) return;

        var random = new Random();
        var appointments = new List<Appointment>();

        foreach (var patient in patients)
        {
            int numberOfAppointments = random.Next(1, 3); // 1 or 2 appointments

            for (int i = 0; i < numberOfAppointments; i++)
            {
                appointments.Add(new Appointment
                {
                    PatientId = patient.Id,
                    When = DateTime.Now.AddDays(random.Next(1, 30)).AddHours(random.Next(8, 17)),
                    Physio = physios[random.Next(physios.Count)]
                });
            }
        }

        context.Appointment.AddRange(appointments);
        context.SaveChanges();
    }
}
```

Figure 32: Example of Appointment Class Seed Using Link to Physio Class

SQLite

Following the setup of the models and basic database seeding, SQLite was chosen as the data engine for the development of FlexiCare. It was the logical choice for a range of practical reasons:

- VS Code is bundled with SQLite by default, offering built-in support.
- It is lightweight and gentle on the limited computing resources available during the project.
- All the data is stored in a single file (flexicare.db), which makes it easy to reset the database if necessary. Simply by deleting the flexicare.db file, re-running the migrations and re-seeding the database with test data.
- This was not a decision on the deployment platform. Entity Framework works with the most popular database platforms and insulates the application from the specific choice. Migrating to another platform would only involve changing a small configuration before re-running the migrations and seeding.

Views, Controllers and Scaffolding

Once the models and database were established correctly, the next step was to create Views and Controllers to allow users to interact with the system. To streamline development, Microsoft provides a tool that can automatically generate Controllers and Views based on a model using a single command, called scaffolding. For example, the following scaffolding command was used to create a controller and views for the ExerciseCategory model:

dotnet aspnet-codegenerator controller -name ExerciseCategoriesController -m ExerciseCategory -dc FlexiCareManager.Data.FlexiCareManagerContext --relativeFolderPath Controllers --useDefaultLayout --referenceScriptLibraries --databaseProvider sqlite

This lone command creates a controller and five views (Index, Details Create, Edit, Delete) automatically linked to the controller.

While in some cases the scaffolded code required slight modification, it offered a strong starting point and saved a lot of manual effort. These minor adjustments included styling tweaks, to ensure the layout and visuals better aligned to FlexiCare's desired design, and navigation updates. For example, Figure 32 shows the Programme page. On this page it was necessary to modify the New button to route to the ProgrammeExcercisesController to add exercises, rather than back to the main Programme controller. This was similarly done for the Edit, Delete and Back Buttons. After creating or editing, it had to be ensured that the controller was redirecting the user to the correct plate.

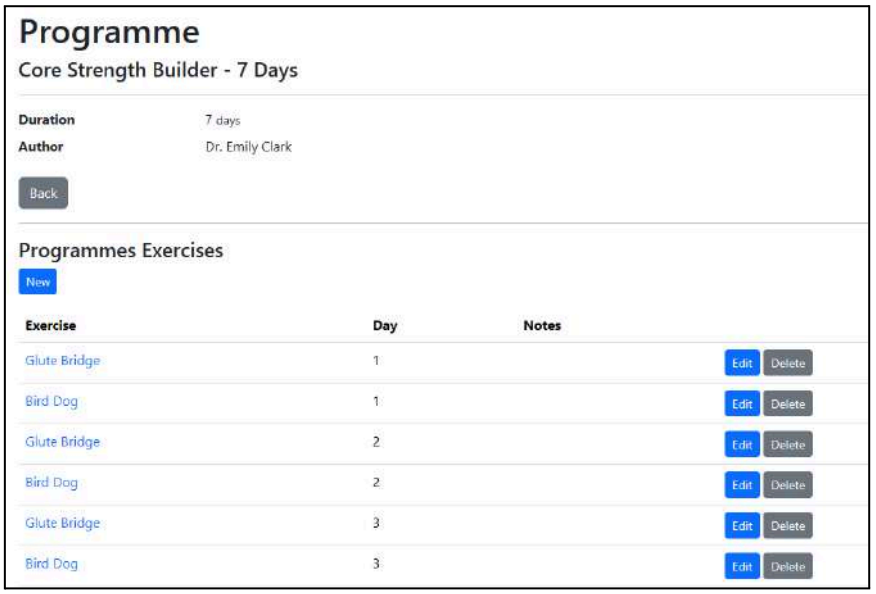


Figure 32: FlexiCare Web Application Programme Page

Layout and Design

Given that FlexiCare's web app was primarily intended for office use (physiotherapists and administrators), the visual design was kept simple and functional. The application was largely maintained with the original default Bootstrap appearance however, minor enhancements were necessary, and these included branding elements such as the logo and Favicon, and improved navigation. An example of the web app is shown in Figure 33, where the focus is firmly placed on functionality and inactivity rather than complex aesthetics.

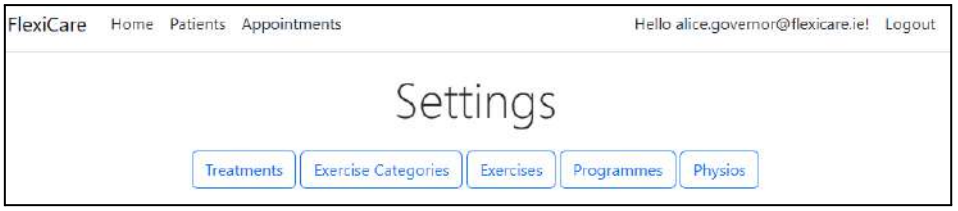


Figure 33: Example of FlexiCare Web Application Layout

User Roles and Navigation

From early in FlexiCare's development, three key user roles were identified: Administrators, Physiotherapists and Patients. However, at this point the web application is primarily focused on Administrators and Physiotherapists, since Patients are expected to interact mainly through the mobile app. Navigation was customised depending on the user role by setting a Navbar variable, such as that in Figure 34, with the potential to load different views depending on the currently logged-in user.

```
public override void OnActionExecuting(ActionExecutingContext filterContext)
{
    if (!User.Identity!.IsAuthenticated)
    {
        ViewData["Navbar"] = "_NotLoggedInNav";
        RedirectToAction("NotLoggedIn", "Home");
        return;
    }
    if (User.IsInRole("ADMINISTRATOR"))
    {
        ViewData["Navbar"] = "_AdministratorNav";
        return;
    }
    if (User.IsInRole("PHYSIO"))
    {
        ViewData["Navbar"] = "_PhysioNav";
        return;
    }
```

Figure 34: Example of Navibar Variables

Authentication and Authorization

To manage users and secure the system, Microsoft Identity was integrated into the backend. It added users and roles and allowed them to be easily integrated with controllers, while automatically providing functionality for secure user login/logout. The package only required some modifications, such as adjustments to the visuals to match FlexCare's style and the removal of some features that require a functional email service (like Forgot Password).

```
[Authorize(Roles = "ADMINISTRATOR, PHYSIO")]
1 reference
public class AppointmentsController : BaseController
```

Figure 35: Updated Controller User Role Requirement

Role-based access control was also implemented. Controllers had to be changed to account for the role restrictions to ensure that only users with the appropriate role could access certain parts of the application as shown in Figure 35. The database seeding was also updated to create user roles (Administrator, Physio, Patient) and sample users for each role, shown in Figure 36.

```
// Add Administrator Roles
await IdentityService.AddRoles(userManager, "alice.governor@flexicare.ie", [adminRole]);

// Add Patient roles
foreach (var patient in patients)
{
    var email = patient.Name!.Replace(' ', '.') + "@patient.ie";
    await IdentityService.AddRoles(userManager, email, [patientRole]);
}

// Add Physios roles
foreach (var physio in physios)
{
    var noTitle = physio.Name!.Replace("Dr. ", "");
    var email = noTitle.Replace(' ', '.') + "@flexicare.ie";
    await IdentityService.AddRoles(userManager, email.ToLower(), [physioRole]);
}
```

Figure 36: Updated Database Seeding for User Roles

5.4.2 FlexiCare API

The FlexiCare API was designed separately to the FlexiManager, due to its different functionality. By maintaining separate projects, it allowed for simpler configuration of identities and ensured the systems could operate and be modified, independently.

The integration between the FlexiCare API and Manger was managed by sharing the same database. As this stores all essential data such as user accounts, roles, patient data and progress tracking, it has everything necessary for the API to function. This shared data structure allows the two components to be fully synchronised, ensuring that both have access to the latest patient details.

```
app.UseSwaggerUi(options =>
{
    options.DocumentPath = "/openapi/v1.json";
});
```

Figure 37: Swagger Support Configuration Command

Swagger support came as part of the default Microsoft solution, it just needed to be configured as shown In Figure 37. It provides a visual interface for interacting with the API, such as in Figure 38. Swagger makes development much more efficient, allowing easy experimentation without requiring the full front-end interface, streamlining the testing and debugging process.

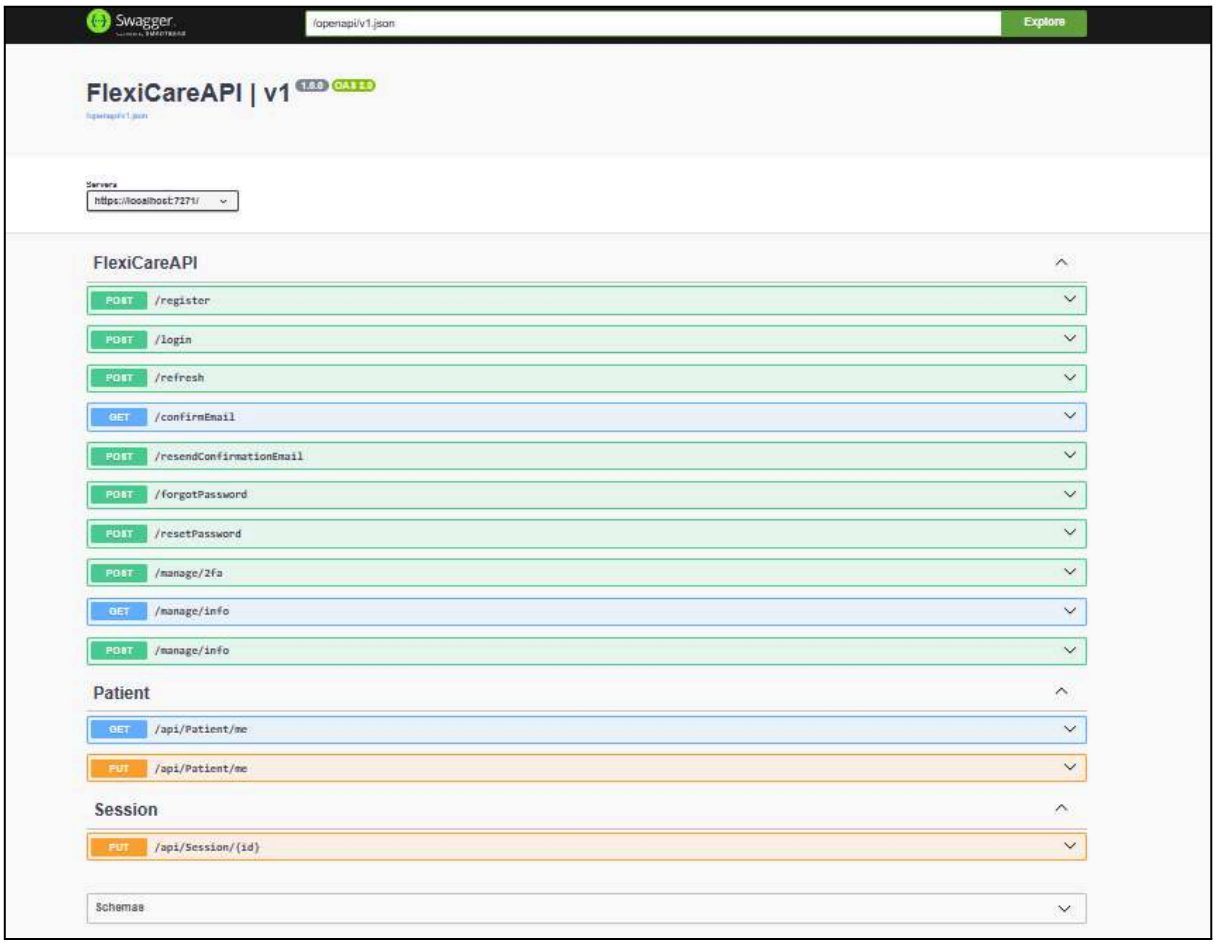


Figure 38: FlexiCare API Swagger Interface

Authentication and Authorisation

Once again, Microsoft Identity was used for authentication, but it was configured to use JWT (JSON Web Token) in this instance., shown in Figure 39. When a user logs in through the app, they receive a JWT access token. This token serves as proof of identity and authorises the user to perform actions on the system. Microsoft Identity uses this token to check if the user exists and then creates a user object. By using JWT, the API ensures that user credentials are verified in a secure manner.

```
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme).AddJwtBearer();
builder.Services.AddAuthorization();

builder.Services.AddIdentityApiEndpoints<IdentityUser>()
    .AddEntityFrameworkStores<FlexiCareManagerContext>();
```

Figure 39: JWT Configuration

Once that was completed, the controllers within the API were modified to require authentication via the access token, ensuring that each request was valid before allowing any access to sensitive data or operations. For example, a Patient would only be able to access and manage their own personal data, ensuring that no user could view or modify another patient's information.

Requests Exposed

In addition to the default requests provided by Microsoft Identity for user management, several custom API endpoints were developed for specific functions. For example:

PUT Patient, shown in Figure 40, allows the app to enable patients to update their details, such as contact information.

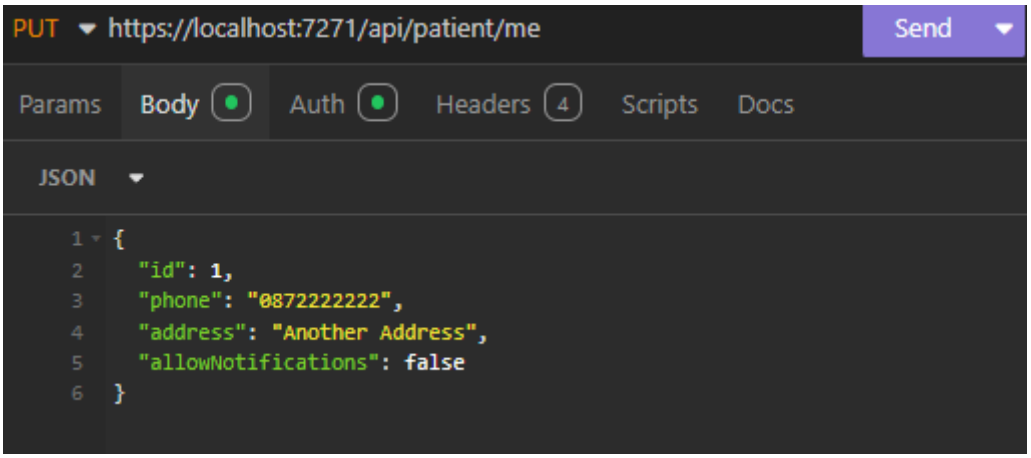


Figure 40: PUT Patient Request Code

GET Patient, shown in Figure 41, returns all the information allocated to a specific patient, such as their personal details, appointment data and scheduled exercise sessions.

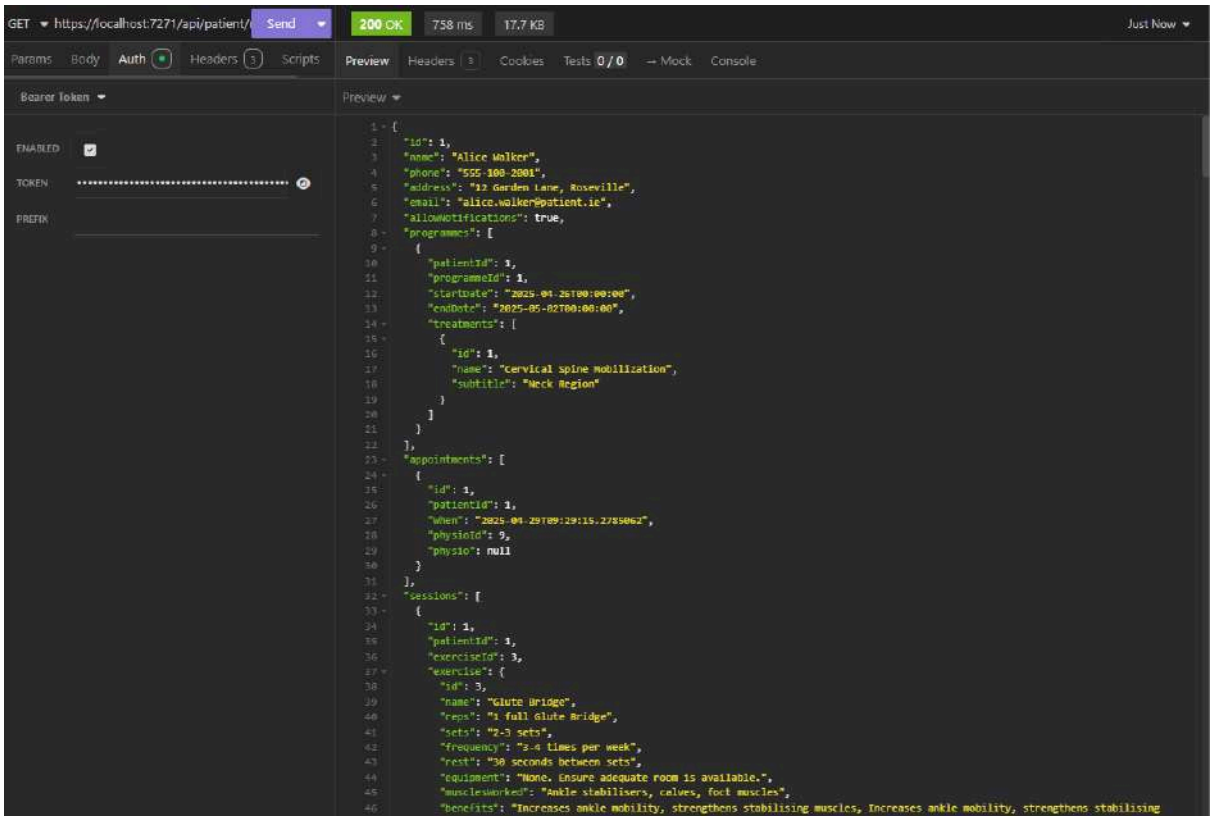


Figure 41: GET Patient Request Code

PUT Session, shown in Figure 42, is used by the patient app to mark an exercise session as completed. After a patient finishes their exercise routine, they can provide feedback on their session. This feedback helps physiotherapists monitor progress and adjust treatment if necessary.

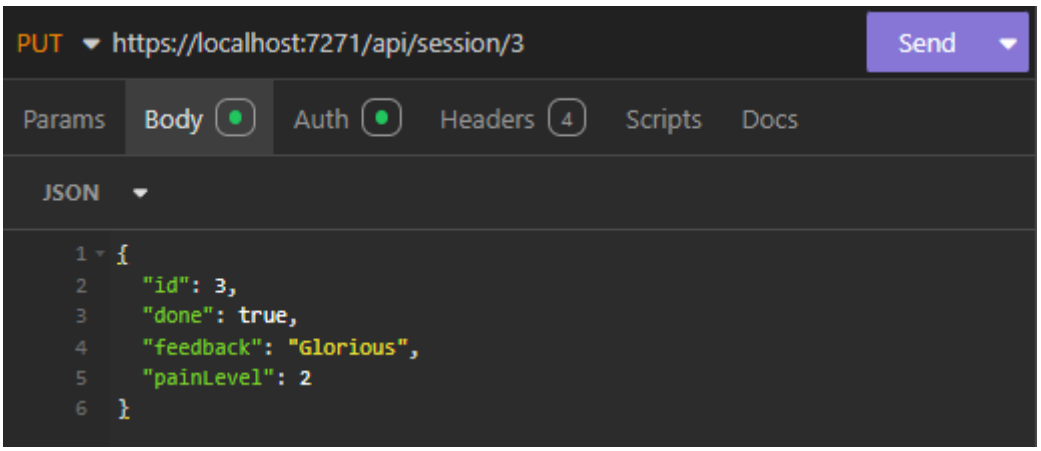


Figure 42: Put Session Request Code

5.4.3 Reflection of Sprint 2

The Development of FlexiCare's backend had several advantages and challenges.

On the positive side, using VS Code streamlined the process by offering an all-in-one solution, with the addition of Microsoft's extensive tutorials and resources, free and open source tools, and cross-platform support (Windows, Linux & Apple). With the addition of helpful features like scaffolding, migrations are on well-supported frameworks. In addition,

C# being a compiled language helped catch errors early and autocomplete suggestions improved efficiency.

However, on a negative standpoint, C#'s syntax was a bit overly complex and off-putting, making the code difficult to read and review. Additionally, the compiled language aspect brought problems. For example, saving a changed source typically refreshed the application, but on occasion, the server itself required restarting, and it was not overly explicit when this was necessary.

5.5. Sprint 3: Frontend Development

The frontend is the part of an application that users interact with directly, visually representing the data and functionality managed by the backend. Sprint 3 outlines the development of the FlexiCare mobile application using modern frontend technologies, including the tools, libraries, and frameworks employed to create an efficient, visually pleasing and intuitive user experience.

5.5.1. Fundamental Desgin

To initiate the frontend development process, the project employed the create-expo-stack initialiser, a standard tool for creating React Native applications using the Expo framework. During setup, configuration flags were used to include Expo, Expo Router with tab navigation, and NativeWindCSS, TailwindCSS adapted for React Native, as shown in Figure 43.

Figure 43: Expo Framework Setup

The development of the web application consisted of Visual Studio Code (VSC) as the primary code editor, Insomnia for testing against the backend API, and Figma for creating and iterating UI designs.

In addition to this, several additional packages were integrated to support the required app functionality, as shown in Figure 44:

Figure 44: Additional Expo Packages Required

- **Jest** was installed to enable unit testing.
- **Expo Splash Screen** was used to provide a customised loading/splash screen, incorporating the FlexiCare logo and brand colours.
- **Expo Status Bar** allowed customisation of the phone's status bar (where battery and network carrier symbols are located) to align with the application's theme.
- **Expo Video** provided built-in support for a video player.
- **Expo System UI** enabled the configuration of global layout styling parameters.
- **Expo Secure Store** facilitated secure key-value storage on the device, which is used to manage sensitive data such as authentication tokens.

These packages were configured as plugins in the app.json file, establishing a splash screen with the FlexiCare logo and brand colours, as shown in Figure 45.

```
// app.json

"plugins": [
    "expo-secure-store",
    [
      "expo-splash-screen",
      {
        "backgroundColor": "#CEEACA",
        "image": "./assets/branding/logo.png",
        "dark": {
          "image": "./assets/branding/logo.png",
          "backgroundColor": "#CEEACA"
        },
        "imageWidth": 75
      }
    ],
    [
      "expo-video",
      {
        "supportsBackgroundPlayback": false,
        "supportsPictureInPicture": false
      }
    ]
  ],
```

Figure 45: Code Snippet Showing Plugin Integration and Splash Screen Design

To enhance performance and usability:

- Reanimated was used to implement smooth, customisable animations.
- Safe Area Context replaced the default React Native SafeAreaView component to ensure compatibility with various screen dimensions, camera notches, cutouts, etc.
- React Native Select Dropdown was integrated to support dropdown menu inputs.

For handling HTTP requests, Axios was selected over JavaScript's built-in Fetch API due to its more intuitive configuration. Axios allowed authentication headers (e.g. bearer

tokens), as shown in Figure 46, to be declared once globally within the AuthContext, enabling both secure and consistent API requests across the entire application without the need for repetitive code re-declaring them each time..

```
// account/login.tsx

// Set global headers to use token
axios.defaults.headers.common['Authorization'] = `Bearer ${accessToken}`
```

Figure 46: Global Axios Authentication Token

TypeScript interfaces were created to mirror the backend API models, for improved efficiency; three of these are shown in Figure 47.

```
// interfaces/interfaces.d.ts

export interface Physio {
  id: number;
  name: string;
  phone: string;
  address: string;
  email: string;
}

export interface Appointment {
  id: number;
  patientId: number;
  when: string;
  physioId: number;
  physio: Physio;
}

export interface Program {
  id: number;
  patientId: number;
  startDate: string;
  endDate: string;
  exerciseCategories: ExerciseCategory[]
}
```

Figure 47: TypeScript Interfaces for API Models

Additionally, an .env file was created to store variables that would be used throughout the entire application, as shown in Figure 48.

```
EXPO_PUBLIC_APP_NAME=Flexicare
EXPO_PUBLIC_APP_LANDING_URL=https://www.flexicare.ie
EXPO_PUBLIC_BACKEND_API_URL=http://localhost:5000
```

Figure 48: .env File Set-Up

TailwindCSS's configuration file was customised, as shown in Figure 49, to include the FlexiCare brand's colour palette, as defined in earlier Figma designs, to input the required visuals for the application.

```
// tailwind.config.js

module.exports = {
  content: ['./App.{js,ts,tsx}','./app/**/*.{js,ts,tsx}' , './components/**/*.
{js,ts,tsx}'],

  presets: [require('nativewind/preset')],
  theme: {
    extend: {
      colors: {
        primary: '#245A3F',
        'primary-lowlight': '',
        pastel: {
          green: '#CEEACA',
          purple: '#CEC8DC',
          highlight: {
            green: '#7C9A79',
            purple: '#7A799A'
          }
        },
        levels: {
          one: '#6BCD5E',
          two: '#73C35D',
          three: '#72B55A',
          four: '#88A557',
          five: '#99A557',
          six: '#A59057',
          seven: '#A57F57',
          eight: '#A56E57',
          nine: '#A45B51',
          ten: '#984646'
        }
      }
    },
  },
  plugins: [],
}
```

Figure 49: TailWind Configuration for FlexiCare Colour Pallete

To keep images organised, subfolders were created within the /assets directory. These included /branding, /icons, /images and /illustrations, each used to store the relative media files.

For authentication, a custom solution was implemented using AuthContext. A file named AuthContext.tsx was created to manage the authentication state, along with both login and logout functions, as shown in Figure 50. This allowed the login status and user data to be shared across the app.

The AuthState holds the user's information once they are logged in, which can be accessed easily from another screen. The onLogin and onLogout functions can be called from UI elements such as buttons to trigger login and logout actions.

```tsx
// contexts/AuthContext.tsx

interface AuthProps {
    authState?: {
        token: string | null;
        authenticated: boolean | null;
        user: { name: string; email: string } | null;
    };
    onLogin?: (email: string, password: string) => Promise<any>;
    onLogOut?: () => Promise<any>;
}
```

Figure 50: AuthProps Interface

The loadToken() function, shown in Figure 51, runs when the app starts. If the user has already previously logged in, it retrieves their saved authentication tokens from Expo Secure Storage and logs them in automatically if the tokens are still valid.

```tsx
// contexts/AuthContext.tsx

useEffect(() => {
        const loadToken = async () => {

            const token = await SecureStore.getItemAsync(TOKEN_KEY);
            const refreshToken = await SecureStore.getItemAsync(REFRESH_TOKEN_KEY);

            const userInfo = await SecureStore.getItemAsync(USER_KEY);

            if (token && refreshToken && userInfo) {

                setAuthState({
                    token: token,
                    authenticated: true,
                    user: JSON.parse(userInfo)
                });

                console.log("jwt loaded from: ", token, " | refresh from:", refreshToken)

                axios.defaults.headers.common['Authorization'] = `Bearer ${token}`;

            }


        }
        loadToken();
    }, [])
```

Figure 51: LoadToken() Function

The login() function, shown in Figure 52, takes an email and password, sends them to the API using Axios, and if the login is successful, saves the returned tokens, name, and email to Expo Secure Storage. The user's authentication state is then updated.

Since access tokens expire after sixty minutes, the refreshAccessToken() function, shown in Figure 53, uses the Refresh token to request a new updated Access token from the API. The new Access Token is also set as the Authentication Bearer token globally for all Axios

requests, ensuring that all HTTP requests to the API are authenticated automatically without additional configuration.

```
// contexts/AuthContext.tsx

const login = async (email:string, password:string) => {
    try {

        const response = await axios.post(`${BASE_URL}/login`, {email, password});

        const { accessToken, refreshToken } = response.data

        // Set global headers to use token
        axios.defaults.headers.common['Authorization'] = `Bearer ${accessToken}`

        const userResponse = await axios.get(`${BASE_URL}/api/Patient/me`);
        const { name, email: userEmail } = userResponse.data;

        setAuthState({
            token: accessToken,
            authenticated: true,
            user: {
                // eslint-disable-next-line object-shorthand
                name: name,
                email: userEmail
            }
        })

        await SecureStore.setItemAsync(TOKEN_KEY, accessToken)
        await SecureStore.setItemAsync(REFRESH_TOKEN_KEY, refreshToken)
        await SecureStore.setItemAsync(USER_KEY, JSON.stringify({ name, email: userEmail
}));

        return response;

    } catch (err) {

        console.log(err, "login error")

        return { error: true, msg: (err as any).response.data.msg}

    } finally {

    }
}
```

Figure 52: Login() Function

```
// contexts/AuthContext.tsx

const refreshAccessToken = async () => {

    const refreshToken = await SecureStore.getItemAsync(REFRESH_TOKEN_KEY);

    if (!refreshToken) {
        return null;
    }

    try {

        const response = await axios.post(`${BASE_URL}/refresh`, { refreshToken });
        const { accessToken } = response.data;

        // Update the access token in Secure Store
        await SecureStore.setItemAsync(TOKEN_KEY, accessToken);

        // Update the global headers
        axios.defaults.headers.common['Authorization'] = `Bearer ${accessToken}`;

        return accessToken;

    } catch (err) {

        console.log('Error refreshing accessToken:', err);

        return null;

    }

};
```

Figure 53: refreshAcessToken() Function

The logout() function, shown in Figure 54, removes the stored tokens and user information from Expo Secure Storage and resets the authentication state, logging the user out.

```
// contexts/AuthContext.tsx

const logout = async () => {

    // Delete JWT token from Local Store
    await SecureStore.deleteItemAsync(TOKEN_KEY);
    await SecureStore.deleteItemAsync(REFRESH_TOKEN_KEY);

    axios.defaults.headers.common['Authorization'] = '';

    setAuthState({
        token: null,
        authenticated: false,
        user: null
    });
}

const value = {
    onLogin: login,
    onLogOut: logout,
    authState
};
```

Figure 54: logout() Function

The login screen (login.tsx) was then implemented, shown in Figure 55. It includes state variables for the email and password fields to allow their value to be synced with the login form's inputs. useAuth is pulled in from AuthContext, and a login() asynchronous function is attempted. If login is successful, the user is redirected to the home screen.

```tsx
// app/account/login.tsx

const Login = () => {
    const [email, SetEmail] = useState('');
    const [password, SetPassword] = useState('');
    const { onLogin, authState } = useAuth();

    const login = async () => {
        const response = await onLogin!(email, password);
        if (response.error) {
            alert(response.msg)
        } else {
            return <Redirect href="/" />
        }
    }

    if (authState?.authenticated === true) {
        return <Redirect href="/" />
    }

    return (

        // JSX View Content
    )
}

export default Login
```

Figure 55: Log-In Page Implementation

To avoid repeating code, a component file was created to show links to the Terms of Service and Privacy Policy, as shown in Figure 56. These links are used on both the login and onboarding screens, so this code snippet was abstracted to be reused across both views.

Furthermore, the website URL was stored within the .env for future customisation. For example, if this product were to be deployed in production for a new customer, the .env data could easily be changed.

```tsx
// components/OnboardingPolicies.tsx

export const OnboardingPolicies = () => {
  return (
    <View className="w-full flex-row items-center justify-center gap-x-4 py-6">

        <TouchableOpacity
          onPress={() => Linking.openURL(`${process.env.APP_LANDING_URL}/privacy-policy`)}
        >
            <Text>Privacy Policy</Text>
        </TouchableOpacity>

        <View className="bg-black w-1 h-1 rounded-full" />

        <TouchableOpacity
          onPress={() => Linking.openURL(`${process.env.APP_LANDING_URL}/terms-of-service`)}
        >
            <Text>Terms of Service</Text>
        </TouchableOpacity>

    </View>
  );
};
```

Figure 56: unAuthenticated Views Component FIle

The layout file, one directory down within the (protected) directory, shown in Figure 57, checks if a user is authenticated by grabbing the authState from the AuthContext provider. If not, and the user attempts to visit any page within the (protected) directory, they are redirected back to the login page.

The parentheses on the directory name (protected) indicate that this is a grouping folder, used only to organise related pages. Pages inside this directory are served at the top level. For example, /app/(protected)/tasks has the URL of /tasks, whereas /app/account/login has the URL of /account/login.

A root layout file was added directly inside the /app directory, at the top level above all pages within the app, as shown in Figure 58. A <Stack> element was created to define the app's main navigation structure, including two subdirectories as <Stack.Screen> entries: (protected) for pages that require authentication, and account for the pre-login pages.

The global background colour was also set to white, affecting all screens throughout the app.

```tsx
// app/(protected)_layout.tsx

import { useAuth } from "context/AuthContext";
import { Redirect, Stack } from "expo-router";

export default function ProtectedLayout() {
    const { authState, onLogOut } = useAuth();

    if (!authState?.authenticated) {

        console.log('not authenticated!', authState?.authenticated)
        return <Redirect href="/account/login" />

    } else {

        return <Stack>
            <Stack.Screen
                name="(tabs)"
                options={{ headerShown: false }}
            />
        </Stack>;

    }
}
```

Figure 57: Protected Views Layout File

```tsx
// app/_layout.tsx

SystemUI.setBackgroundColorAsync("#FFFFFF");

export default function RootLayout() {
    return <AuthProvider>
        <Stack
            screenOptions={{
                contentStyle: {
                    backgroundColor: "#FFFFFF",
                },
            }}
        >
            <Stack.Screen
                name="(protected)"
                options={{
                    headerShown: false
                }}
            />
            <Stack.Screen
                name="account"
                options={{ headerShown: false }}
            />
        </Stack>
    </AuthProvider>;
}
```

Figure 58: Root Layout File

5.5.2. Pages

In the index (Home) page, several functions were written to calculate various metrics for the user, as shown in Figure 59. These are displayed in the view.

A call is made to the API to fetch all exercise session data for a user. This data is stored as sessionsData and is used across multiple functions.

For example, the averagePainLevel function filters all completed sessions, adds up their pain levels, and divides by the number of sessions to get the average. This value is then rounded to one decimal place and displayed as a score out of 10.

```tsx
// app/(protected)/index.tsx

// Calculate how many exercise sessions a user has completed
function getSessionsProgress(sessions: { done: boolean }[]) {
    const completed = sessions.filter(session => session.done).length;
    const total = sessions.length;
    return { completed, total };
}

// Calculate average pain level
const getAveragePainLevel = (sessions: Session[]): string => {

    const completedSessions = sessions.filter(session => session.done === true);

    if (completedSessions.length === 0) {
        return "0";
    }

    const totalPainLevel = completedSessions.reduce((sum, session) => sum + session.painLevel, 0);
    const averagePainLevel = (totalPainLevel / completedSessions.length).toFixed(1);

    return averagePainLevel;

};

// Get sessions data from the API
useEffect(() => {

    const fetchData = async () => {
        const sessions = await getSessions();

        if (!sessions.error) {
            setSessionsData(sessions);
        }
    };

    fetchData();

}, []);

// Configure the sessionsProgress state if available, else set to defaults
const sessionsProgress = sessionsData && Array.isArray(sessionsData)
    ? getSessionsProgress(sessionsData)
    : { completed: 0, total: 0 };

// Configure the painLevel state if available, else set to defaults
const avgPainLevel = sessionsData && Array.isArray(sessionsData)
    ? getAveragePainLevel(sessionsData)
    : "0"
```

Figure 59: Index (Home) Page Functions

For the appointments pages, several functions had to be written to handle the logic for displaying and interacting with user appointments, as shown in Figure 60. A call is made to the API to fetch all of the user's appointments, both upcoming and previous. This data is then used to populate the view.

Instead of a traditional calendar, a horizontal carousel of dates is shown at the top of the screen, just below the heading. This shows the selected date in the centre, with up to three days before and after on either side.

When a user taps on a date, the carousel slides to bring that date to the centre, and the appointment list updates to show only those scheduled for that day. The default selected date is today's date.

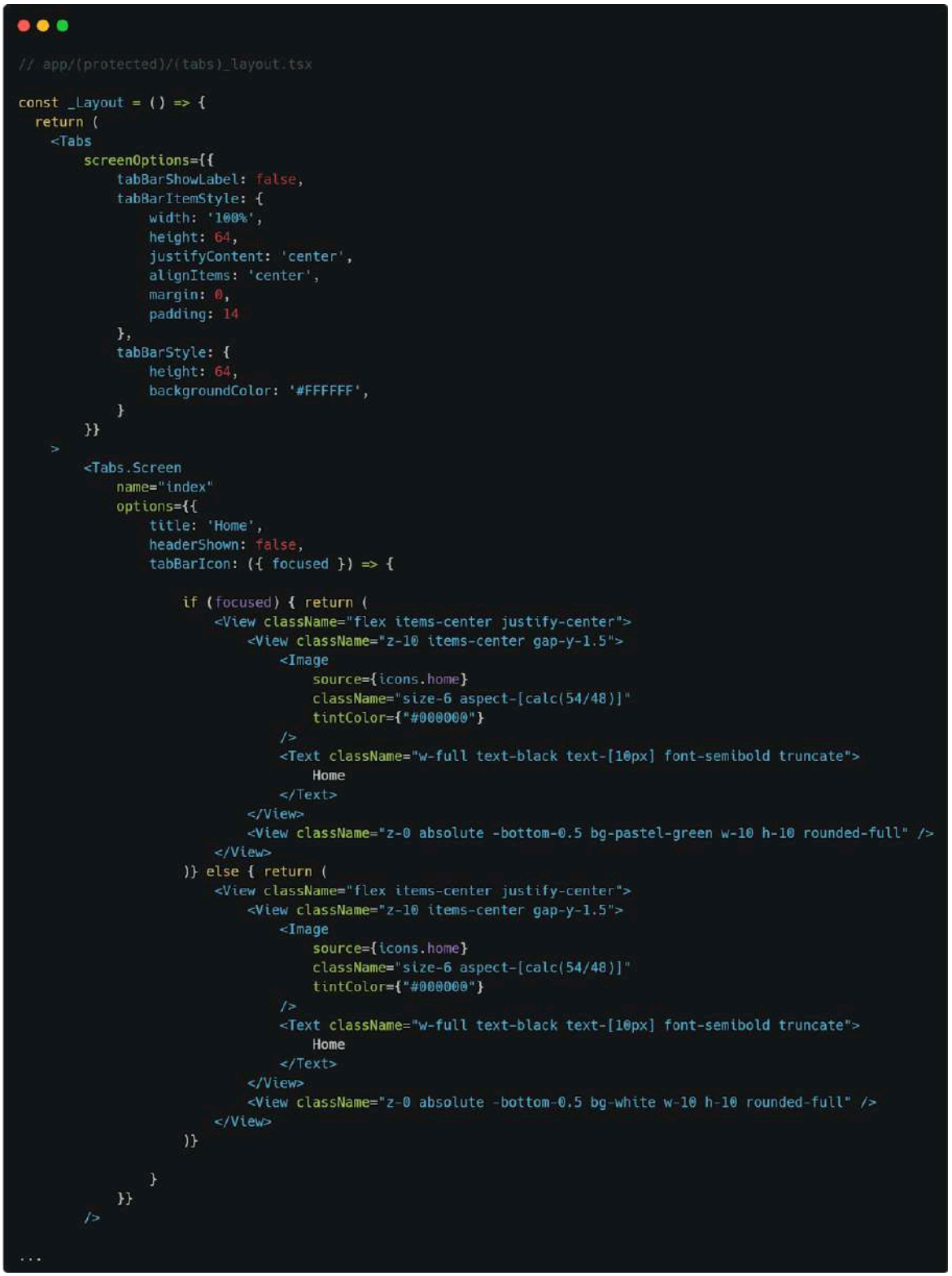
Below the date carousel, appointments for the selected date (or today if none is selected) are displayed first. Following this, a list of all upcoming appointments is shown in chronological order, with the most recent appointments displayed first.

```
// app/(protected)/appointments.tsx

// Filter for selected date (in carousel)
// Filter for upcoming appointments
useEffect(() => {

    // Filter appointments by selected Date
    if (appointmentData) {
        const appointments: Appointment[] = appointmentData as Appointment[];
        selectedDate.setHours(0, 0, 0, 0);

        // (Midnight of following day)
        const tomorrow = new Date(selectedDate);
        tomorrow.setDate(selectedDate.getDate() + 1);

        const selectedDateMap: Appointment[] = appointments
            .filter((appointment) => {
            const appointmentDate = new Date(appointment.when);
            // Check if the appointment is on selected date (regardless of time / if passed)
            return appointmentDate >= selectedDate && appointmentDate < tomorrow;
            })
            .sort((a, b) => new Date(a.when).getTime() - new Date(b.when).getTime());

        setSelectedAppointments(selectedDateMap);
    }

    // Filter appointments by upcoming (after now())
    if (appointmentData) {
        const appointments: Appointment[] =
            appointmentData as Appointment[];

        const upcomingAppointmentsMap: Appointment[] = appointments
            .filter((appointment) => new Date(appointment.when) > todaysDate)
            .sort((a, b) => new Date(a.when).getTime() - new Date(b.when).getTime());

        setUpcomingAppointments(upcomingAppointmentsMap);
    }

}, [appointmentData, selectedDate]);
```

Figure 60: Appointments Page Functions

The layout file under the (tabs) directory, shown in Figure 61, initialises the TabBar, which is located at the bottom of every authenticated page and contains buttons and icons to navigate to the main Home, Tasks, Program, Appointments, and Settings pages.

Considerable effort was made to match the styling of the TabBar as closely as possible to the Figma design.



```
// app/(protected)/(tabs)_layout.tsx

const _Layout = () => {
  return (
    <Tabs
        screenOptions={{
            tabBarShowLabel: false,
            tabBarItemStyle: {
                width: '100%',
                height: 64,
                justifyContent: 'center',
                alignItems: 'center',
                margin: 0,
                padding: 14
            },
            tabBarStyle: {
                height: 64,
                backgroundColor: '#FFFFFF',
            }
        }}
    >
        <Tabs.Screen
            name="index"
            options={{
                title: 'Home',
                headerShown: false,
                tabBarIcon: ({ focused }) => {

                    if (focused) { return (
                        <View className="flex items-center justify-center">
                            <View className="z-10 items-center gap-y-1.5">
                                <Image
                                    source={icons.home}
                                    className="size-6 aspect-[calc(54/48)]"
                                    tintColor={"#000000"}
                                />
                                <Text className="w-full text-black text-[10px] font-semibold truncate">
                                    Home
                                </Text>
                            </View>
                            <View className="z-0 absolute -bottom-0.5 bg-pastel-green w-10 h-10 rounded-full" />
                        </View>
                    )} else { return (
                        <View className="flex items-center justify-center">
                            <View className="z-10 items-center gap-y-1.5">
                                <Image
                                    source={icons.home}
                                    className="size-6 aspect-[calc(54/48)]"
                                    tintColor={"#000000"}
                                />
                                <Text className="w-full text-black text-[10px] font-semibold truncate">
                                    Home
                                </Text>
                            </View>
                            <View className="z-0 absolute -bottom-0.5 bg-white w-10 h-10 rounded-full" />
                        </View>
                    )}

                }
            }}
        />

...
```

Figure 61: Layout File under (Tabs) Directory

When users press the “Tasks” icon in the TabBar, they are redirected (using Expo Router) to the tasks/index.tsx file, shown in Figure 62, lists all tasks (exercise sessions) assigned for the current day.

All sessions are fetched from the API and filtered to include only those scheduled for today. These are then sorted by category, allowing them to be listed under each category header for ease of use.

A user can press on any exercise task (session) in the list to be brought to the relevant exercise page, which provides detailed instructions from the user's physiotherapist on how to complete the exercise, what equipment (if any) is required, and how often it should be performed.

Alternatively, they can click on the unmarked checkbox for any task to be directed straight to the feedback/mark-as-complete form if they are already familiar with how to perform the exercise.

```tsx
// app/(protected)/(tabs)/tasks/index.tsx

useEffect(() => {
    const fetchData = async () => {
        const sessions = await getSessions();

        if (!sessions.error) {
            setSessionsData(sessions);
        }
    };

    fetchData();
}, []);

type GroupedSessionsProps = {
    [categoryId: number]: {
        categoryName: string;
        sessions: Session[];
    };
};

const groupSessionsByCategoryId = (sessions: Session[]): GroupedSessionsProps => {

    return sessions.reduce((groups: GroupedSessionsProps, session: Session) => {

        const categoryId = session.exerciseCategory.id;
        const categoryName = session.exerciseCategory.name;

        if (!groups[categoryId]) {
            groups[categoryId] = {
                categoryName,
                sessions: []
            };
        }

        groups[categoryId].sessions.push(session);

        return groups;

    }, {});
};
```

Figure 62: Task/Index.tsx File

A "Streaks" feature was added to this page, which tracks the number of consecutive days a user has completed at least one assigned task, as shown in Figure 63.

This gamification feature motivates users to complete tasks to maintain their streak. If they approach the end of a day without completing any tasks, they receive a notification reminding them to complete a task to avoid losing their streak.

```tsx
// app/(protected)/(tabs)/tasks/index.tsx

// Calculate a user's unbroken streak of completing (any number of) tasks per day
// If they have no tasks to complete in a day their streak should remain active
const calculateStreak = (sessionsData: Session[]): number => {

    if (!sessionsData) return 0;

    // Step 1: Group all sessions by date
    const sessionsByDate: { [date: string]: Session[] } = {};

    sessionsData.forEach(session => {
        const sessionDate = new Date(session.exerciseDate);
        const formattedDate = sessionDate.toISOString().split('T')[0];

    if (!sessionsByDate[formattedDate]) {
        sessionsByDate[formattedDate] = [];
        }
        sessionsByDate[formattedDate].push(session);
    });

    let streak = 0;
    let currentDate = new Date();

    // Keep going back day by day
    while (true) {
        const formattedDate = currentDate.toISOString().split('T')[0];
        const sessions = sessionsByDate[formattedDate];

        if (!sessions) {
        // No sessions scheduled for that day => streak continues
        streak++;
        } else {
        const anyDone = sessions.some(session => session.done);

        if (anyDone) {
            // At least one session completed => streak continues
            streak++;
        } else {
            // Sessions scheduled but none done => streak breaks
            break;
        }
        }

        // Move to prev day
        currentDate.setDate(currentDate.getDate() - 1);
    }

    return streak;

};

const streak = sessionsData ? calculateStreak(sessionsData) : 0
```

Figure 63: Streak Feature

When a user clicks on a task from the Tasks list page, they get directed to the EachTask ([id].tsx) page, shown in Figure 64, which extracts an id from the dynamic route (e.g. /tasks/1, where id=1).

A call is then made to the API to retrieve all exercise sessions. This data is filtered to find the session matching the provided ID.

This page displays detailed information on how to perform the select exercise task (session). Step-by-step instructions written by the physiotherapist are shown clearly on the screen. In addition to written guidance, instructional videos, if available in the API, are also displayed using the Expo Video package, which was explicitly configured for this purpose.

```tsx
// app/(protected)/(tabs)/tasks/[id].tsx

const { id } = useLocalSearchParams();
const [sessionsData, setSessionsData]= useState<Session[] | null>(null);
const [eachSession, setEachSession]= useState<Session | null>(null);
const [sessionIndex, setSessionIndex] = useState<number | null>(null);
const [totalSessions, setTotalSessions] = useState<number>(0);

// get Sessions from API
useEffect(() => {
    const fetchData = async () => {
        const sessions = await getSessions();

        if (!sessions.error) {
            setSessionsData(sessions);
        }
    };

    fetchData();
}, []);

// When sessions are loaded, filter for the session by [id]
useEffect(() => {
    if (id && sessionsData !== null) {

        const sessionId = Number(id);

        const foundSession = sessionsData.find((session: { id: number; }) => session.id === sessionId);
        setEachSession(foundSession || null);

        const todaysSessions = getTodaysSessions(sessionsData);
        setTotalSessions(todaysSessions.length);

        const index = todaysSessions.findIndex((session) => session.id === sessionId);
        setSessionIndex(index !== -1 ? index + 1 : null);

    }
}, [id, sessionsData]);

// Add video player functionality using Expo Video
// Videos can be loaded from an external source e.g. YouTube
let videoSource = "";
if (eachSession !== null && eachSession.exercise.videoUrl !== null) {
    videoSource = eachSession.exercise.videoUrl
}
// Video player config
const player = useVideoPlayer(videoSource, player => {
    player.loop = true;
    player.play();
});
const { isPlaying } = useEvent(player, 'playingChange', { isPlaying: player.playing });
```

Figure 64: EachTask ([id].tsx) Page

Users can mark an individual exercise task as completed on the task feedback form. They can also provide optional feedback to their physiotherapist, select a pain level, and input the number of repetitions and sets performed, as shown in Figure 65. This data contributes to progress tracking and is later reflected in the metrics displayed on the Home page.

Session data is retrieved by id via an API call. After form submission, a PUT request is sent using the updateSession function from the api.ts helper module, which updates the session with the user's input.

```tsx
// app/(protected)/(tabs)/tasks/feedback/[id].tsx

const { id } = useLocalSearchParams();
const sessionId = Number(id);

const [sessionData, setSessionData]= useState<Session | null>(null);
const [repsCompleted, setRepsCompleted]= useState<number>(0);
const [setsCompleted, setSetsCompleted]= useState<number>(0);
const [painLevel, onChangePainLevel]= useState<number>(0);
const [feedbackNotes, onChangeFeedbackNotes]= useState<string>("");

// get Sessions from API
useEffect(() => {
    const fetchSession = async () => {
        const session = await getSessionById(sessionId);
        if (session !== null && !('error' in session)) {
            if (session.done) {
                router.replace('/tasks');
            }
            setSessionData(session);
        }
    };

    fetchSession();
}, []);

// Submit changes (PUT request) to the API
const submitFeedback = async () => {
    try {
        const submit: { error: boolean; msg: string } =
            await updateSession({
                id: sessionId,
                done: true,
                painLevel,
                feedback: feedbackNotes
        });

        if (submit.error) {
            alert(`Error: Feedback not submitted - ${submit.msg}`);
        } else {
            alert (submit.msg);
            router.replace('/tasks');
        }
    } catch {
        alert("Error: Feedback not submitted");
    }
}
```

Figure 65: Feedback/{id} .tsx

A custom view component was built to help with pain level selection, as shown in Figure 66. This component displays ten pain levels as clickable buttons, styled on a colour gradient from green (low pain) to red (high pain). The selected pain level is visually highlighted in response to user input.

In addition to clicking, the component has swiping functionality, allowing users to slide across the pain level scale to adjust their selection. The prop is then returned as state to the task feedback form.

```tsx
// components/PainLevelSlider.tsx

const [selectedPainLevel, setSelectedPainLevel]= useState<number>(1);

useEffect(() => {

    onChangeLevel(selectedPainLevel);

}, [selectedPainLevel]);

return (
<View className="flex gap-y-3 -mx-1">

    <View className="flex flex-row justify-between px-1">
        <Text className="text-black font-bold">
            Pain Level
        </Text>
        <Text className="text-gray-400">
            { selectedPainLevel } / 10
        </Text>
    </View>

    <View className="w-full flex flex-row">

        {Array.from({ length: 10 }, (_, i) => (
        <TouchableOpacity
            className="w-[10%] h-24 px-0.5"
            onPress={() => setSelectedPainLevel(i + 1)}
            key={i + 1}
        >
            {i + 1 === selectedPainLevel ? (
            <View className="bg-pastel-green w-full h-20 px-1 pt-1 rounded-lg flex items-center">
                <View className="w-full h-12 rounded-md" style={{ backgroundColor: painLevelColours[i] }} />
                <Text className="text-black text-sm font-bold">{i + 1}</Text>
            </View>
            ) : (
            <View className="w-full h-20 px-0.5 pt-1 rounded-md flex items-center">
                <View className="w-full h-12 rounded-md" style={{ backgroundColor: painLevelColours[i] }} />
                <Text className="text-gray-400 text-sm font-bold">{i + 1}</Text>
            </View>
            )}

            </TouchableOpacity>
            ))}
        </View>
...
```

Figure 66: PainlevelSlider Component

When the “Settings” icon in the TabBar is selected, navigation is directed to the settings/index.tsx page, as shown in Figure 66. This page displays an overview of the user's profile, including name, email address, and an identicon generated based on user data. Below the profile section, a list of available settings options is available to the user.

Figure 67 shows how two of those options are implemented: the social media sharing functionality and the logout, which uses the onLogOut function defined in the AuthContext file to handle logging the user out easily.

```tsx
// app/(protected)/(tabs)/settings/index.tsx

// Process logout when user presses the log out button
const handleLogout = async () => {
    if (onLogOut) {
        await onLogOut();
        router.replace('/account/login'); // 🚀 redirect them back to login page (optional)
    }
};

// Handle social media sharing onPress
const onShare = async () => {
try {
    const result = await Share.share({
        message:
            'FlexiCare | An app to track your Physio exercise progress',
        url:
            `${process.env.EXPO_PUBLIC_APP_LANDING_URL}/app`,
        title:
            'FlexiCare'
    });
} catch (error: any) {
  Alert.alert(error.message);
}
```

Figure 67: Setting/index.tsx Page

For the form pages within the setting directory, which can be accessed by selecting an option from the settings list on the index page, multiple code snippets were abstracted into view components for easy reuse over each form. An example of this is shown in Figure 68 for the feedback form.

Elements like form containers, text inputs, text areas, email fields, and section headers were converted into components.

```tsx
// app/(protected)/(tabs)/settings/feedback.tsx

const Feedback = () => {
    return (
    <ScrollView>

        <FormPageHeader
            backPageName="Settings"
            backPageURI="settings"
            headerText="Feedback"
            descriptionText="Your feedback will help us improve our app..
                             Let us know about any new features you would like to see."
            descriptionVisible={true}
        />
```

Figure 68: Setting/feedback.tsx Page

Figure 69 shows the implementation of a header component used across multiple pages in the app. It accepts a set of typed props, allowing it to be customised based on the specific requirements of each page where it is used.

```tsx
// components/FormPageHeader.tsx

type FormPageHeaderProps = {
    backPageName: string
    backPageURI: string
    headerText: string
    descriptionText: string,
    descriptionVisible: boolean
    children?: React.ReactNode;
};

export const FormPageHeader = (
    { backPageName, backPageURI, headerText, descriptionText, descriptionVisible, children }
    : FormPageHeaderProps) => {
  return (
    <View className="w-full flex gap-y-6 pt-9 px-6">

        <TouchableOpacity
            className="text-gray-400 font-medium flex flex-row items-center gap-x-3"
            onPress={() => router.navigate(`/${backPageURI}`)}
        >
            <Image
                source={icons.chevronLeft}
                className="size-2 aspect-[calc(6/9)]"
                tintColor="#BCC2CC"
            />
            <Text className="text-gray-400 font-medium">Back to { backPageName }</Text>
        </TouchableOpacity>

        <Text className="text-black text-3xl font-bold tracking-wide">{ headerText }</Text>

        {descriptionVisible &&
        <Text className="text-gray-500 leading-6">
            { descriptionText }
        </Text>

    </View>
  );
};
```

Figure 69: FormPageHeader Component

The simpler FormPageView component, shown in Figure 70, enforces consistent spacing across all form pages. It allows child elements, such as form field components, to be passed from the page.

```tsx
// components/FormPageView.tsx

type FormPageViewProps = {
    children?: React.ReactNode;
};

export const FormPageView = (
    { children }
    : FormPageViewProps) => {
  return (
    <View className="w-full flex gap-y-6">

        { children }

    </View>
  );
};
```

Figure 70: FormPageView Component

The FormButton component, shown in Figure 71, allows the onPress element parameter to be passed from the View as a prop.

```tsx
// components/FormButton.tsx

type FormButtonProps = {
    text: string
    onPress: any
    children?: React.ReactNode;
};

export const FormButton = (
    { text, onPress, children }
    : FormButtonProps) => {
  return (
    <TouchableOpacity
        className="bg-pastel-green w-full h-14 rounded-full
                   flex items-center justify-center"
        onPress={onPress}
    >
        <Text className="text-black text-lg font-medium tracking-wide">
            { text }
        </Text>
    </TouchableOpacity>
  );
};
```

Figure 71: FormButton Component

Figure 72 shows an example of the simpler View in the settings/edit.tsx page is made possible by the Form components.

```
// app/(protected)/(tabs)/settings/edit.tsx

return (
    <ScrollView>
        <FormPageHeader
            backPageName="Settings"
            backPageURI="settings"
            headerText="Edit Profile"
            descriptionText=""
            descriptionVisible={false}
        />

        <View className="w-full py-9 px-6">
            <FormPageView>

                <TextInputField
                    placeholder="Alice Walker"
                    value=""
                    onChangeText={undefined}
                    keyboardType="default"
                    autoComplete="off"
                    labelText="Full Name"
                    labelVisible
                    isDisabled
                />
                <TextInputField
                    placeholder="alice.walker@patient.ie"
                    value=""
                    onChangeText={undefined}
                    keyboardType="default"
                    autoComplete="off"
                    labelText="Email Address"
                    labelVisible
                    isDisabled
                />
                <TextInputField
                    placeholder="555-100-2001"
                    value={ String(mobileNumber) }
                    onChangeText={onChangeMobileNumber}
                    keyboardType="number-pad"
                    autoComplete="tel"
                    labelText="Mobile Number"
                    labelVisible
                    isDisabled={false}
                />
                <TextAreaField
                    placeholder="12 Garden Lane, Roseville"
                    value={ String(address) }
                    onChangeText={onChangeAddress}
                    numberOfLines={6}
                    maxLength={240}
                    labelText="Address"
                    labelVisible
                />
                <FormButton
                    text="Save"
                    onPress={saveProfile}
                />

            </FormPageView>
        </View>
    </ScrollView>
)
```

Figure 72: Simple View within the Setting/edit.tsx Page

When the user presses the “Save” button to confirm changes to their profile on the edit page, the saveProfile function, shown in Figure 73, is triggered. This function sends the updated data to the API using the updatePatient helper function defined in the api.ts file.

```
// app/(protected)/(tabs)/settings/edit.tsx

const saveProfile = async () => {

    try {

        const submit: { error: boolean; msg: string } =
            await updatePatient({
                id: 1,
                phone: "1234",
                address,
                allowNotifications: userData.allowNotifications,
        });

        if (submit.error) {
            alert(`Error: Feedback not submitted -
${submit.msg}`);
        } else {
            alert (submit.msg);
            router.replace('/settings');
        }

    } catch {

        alert("Error: Feedback not submitted");

    }

}
```

Figure 73: saveProfile Function

5.5.3. Reflection of Sprint 3

React Native and TypeScript made development faster by using a single codebase for both iOS and Android. This saved time and kept the app consistent across devices. The component-based architecture of React Native made it easy to develop and update the user interface. Plus, the extensive collection of libraries and tools available for JavaScript and React Native helped speed up adding new features.

However, there were also some downsides. The app sometimes faced performance issues with complex animations or native features, requiring extra coding. JavaScript's dynamic typing also led to errors that were harder to spot until later in development. JavaScript's dynamic typing also led to mistakes that were harder to spot until later in development. Finally, features like hot reloading didn't always work perfectly, which made testing and debugging slower. These issues made the development process a bit more challenging.

6. Testing and Results

6.1. Usability Testing

Usability testing would allow the physiotherapists and their patients to try the application and provide feedback. Physiotherapists would use it to manage patient information, and patients would use it to follow their treatment plans. While not included within the scope of this current project, this testing would be a priority for further development of the application. It would include testing of simple tasks like creating an account and navigating the app, with user feedback collected on the ease and effectiveness of usage. This would ensure that the application continues to meet the needs of all users.

6.2. Unit testing

Unit testing is performed to check small parts of a program and ensure they work correctly. This helps identify issues early by testing individual functions or components. For this project, unit testing was completed on both the frontend and backend. MSTest with C# was used to test the backend, while Jest with JavaScript was used to test the frontend, ensuring that both parts of the application worked as expected.

6.2.1. Backend Testing

Automated backend testing was again developed using VS Code. Dedicated test projects were created within the code to ensure separation from the production code. MSTest (MSUnit) was chosen for the testing framework, which is Microsoft's recommended approach for C# unit tests.

The first step involved adding a reference from the test project to the main project under test, shown in Figure 74.

```
<ProjectReference Include="..\FlexiCareAPI\FlexiCareAPI.csproj" />
```

Figure 74: Project Reference Snippet

VS code automatically searches the solution for test cases and generates a test explorer window, indicating passing tests, those that have yet to be run and failed tests, as shown in Figure 75. Tests can be run individually or collectively, with support for debugging during test execution

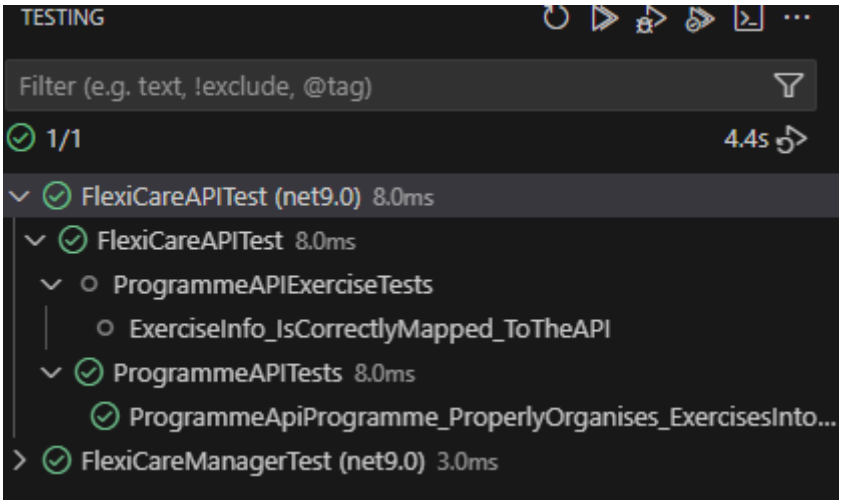


Figure 75: Test Explorer Window

Each test case was structured using the standard MSTest convention, which divides the test logic into three key sections, as shown in Figure 76:

- **Arrange -** Set up the necessary data and variables.
- **Act -** Runs the test
- **Assert -** Verify that the results meet the expected outcomes.

```
public void ExerciseInfo_IsCorrectlyMapped_ToTheAPI()
{
    // ARRANGE

    // Create a FlexiCare Exercise class
    var exercise1 = new Exercise()
    {
        Id=1,
        Name = null,
        Reps = "5 - 10",
        Sets = "4 or less",
        Frequency = "once or twice"
    };

    // ACT
    var result = new ProgrammeApiExercise(dayNumber: 1, exercise: exercise1);

    // ASSERT
    Assert.IsNull(result.Name);
    Assert.AreEqual("5 - 10", result.Reps);
    Assert.AreEqual("4 or less", result.Sets);
    Assert.AreEqual("once or twice", result.Frequency);
}
```

Figure 76: Arrange, Act, Assert MSTest Convention.

VS Code highlights any failing tests directly, as shown in Figure xxx. It also visually marks the related code, as shown in Figure 77, making it easier to navigate and address.

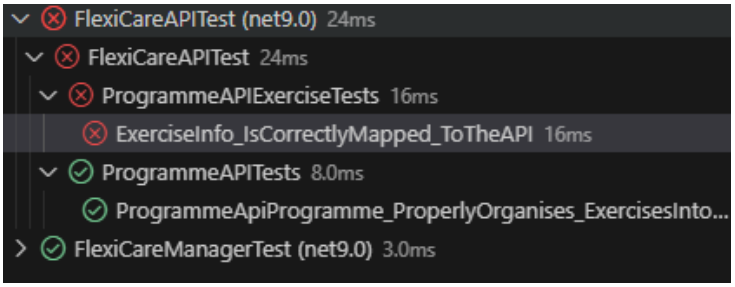


Figure 77: Failed Test Flagging

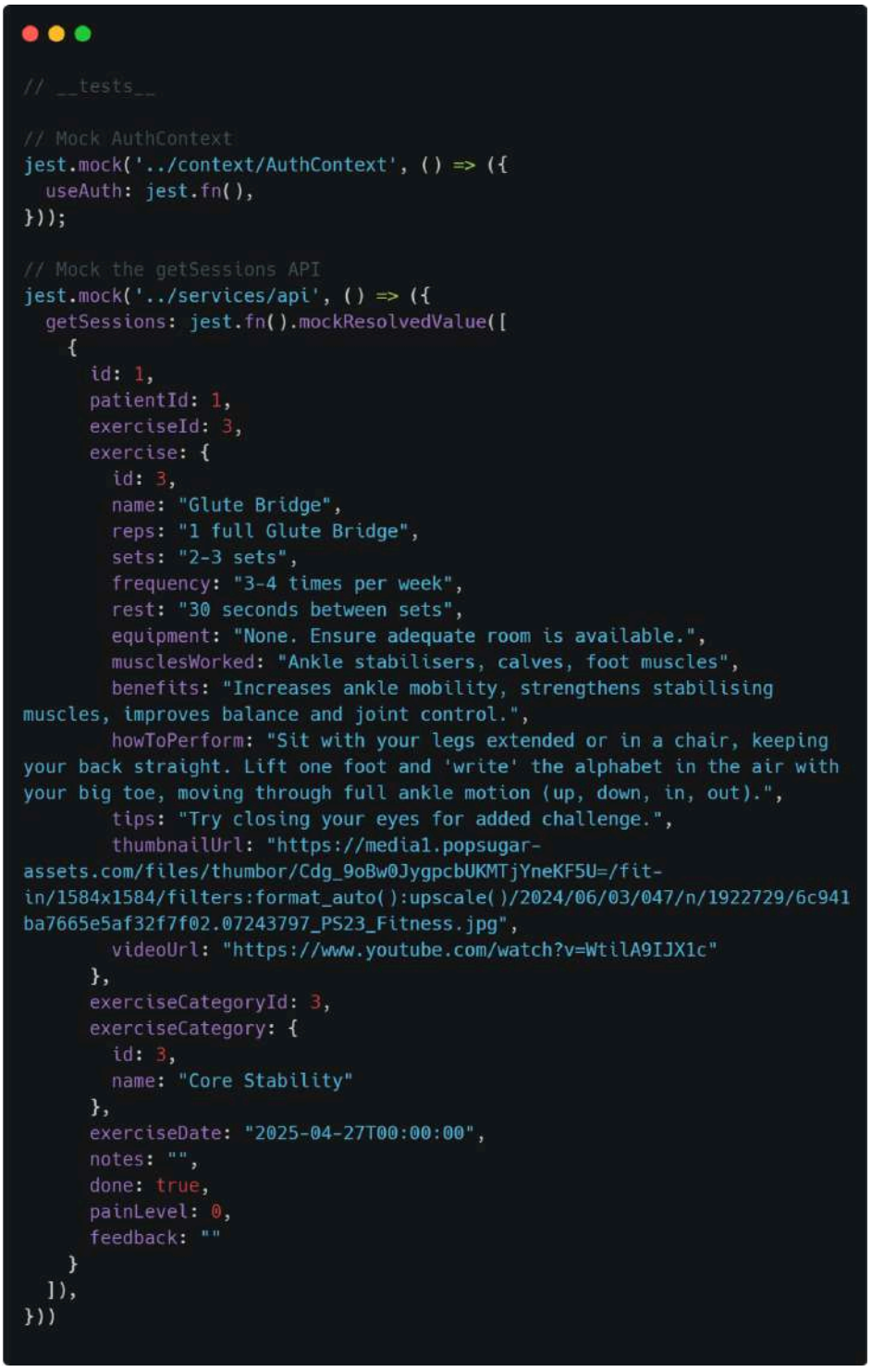
```
public void ExerciseInfo_IsCorrectlyMapped_ToTheAPI()
{
    // ARRANGE

    // Create a FlexiCare Exercise class
    var exercise1 = new Exercise()
    {
        Id=1,
        Name = null,
        Reps = "5 - 10",
        Sets = "4 or less",
        Frequency = "once or twice"
    };

    // ACT
    var result = new ProgrammeApiExercise(dayNumber: 1, exercise: exercise1);

    // ASSERT
    Assert.IsNull(result.Name);
    Assert.AreEqual("5 - 10", result.Reps);
    Assert.AreEqual("4 or less", result.Sets);
    Assert.AreEqual("this is wrong", result.Frequency); ⊗ Assert.AreEqual failed. Expected:<this is wrong>. Actual:<once or twice>.
}
```

Figure 78: Failed Test Code Highlighting

6.2.2. Frontend Testing

Automated backend testing was again developed using VS Code. For the testing framework, Jest was chosen, which is the most popular package for JavaScript unit testing. Additionally, the jest-expo package was installed for closer integration with Expo and the ability to use .tsx files for tests.

Jest's mocking capabilities were used to simulate API responses and authentication context in tests. Figure 79 is an example of how to mock an API call and the AuthContext in a test file.



```
// __tests__

// Mock AuthContext
jest.mock('../context/AuthContext', () => ({
  useAuth: jest.fn(),
}));

// Mock the getSessions API
jest.mock('../services/api', () => ({
  getSessions: jest.fn().mockResolvedValue([
    {
      id: 1,
      patientId: 1,
      exerciseId: 3,
      exercise: {
        id: 3,
        name: "Glute Bridge",
        reps: "1 full Glute Bridge",
        sets: "2-3 sets",
        frequency: "3-4 times per week",
        rest: "30 seconds between sets",
        equipment: "None. Ensure adequate room is available.",
        musclesWorked: "Ankle stabilisers, calves, foot muscles",
        benefits: "Increases ankle mobility, strengthens stabilising
muscles, improves balance and joint control.",
        howToPerform: "Sit with your legs extended or in a chair, keeping
your back straight. Lift one foot and 'write' the alphabet in the air with
your big toe, moving through full ankle motion (up, down, in, out).",
        tips: "Try closing your eyes for added challenge.",
        thumbnailUrl: "https://media1.popsugar-
assets.com/files/thumbor/Cdg_9oBw0JygpcbUKMTjYneKF5U=/fit-
in/1584x1584/filters:format_auto():upscale()/2024/06/03/047/n/1922729/6c941
ba7665e5af32f7f02.07243797_PS23_Fitness.jpg",
        videoUrl: "https://www.youtube.com/watch?v=WtilA9IJX1c"
      },
      exerciseCategoryId: 3,
      exerciseCategory: {
        id: 3,
        name: "Core Stability"
      },
      exerciseDate: "2025-04-27T00:00:00",
      notes: "",
      done: true,
      painLevel: 0,
      feedback: ""
    }
  ]),
}))
```

Figure 79: Mocking API and AuthContext with Jest

A test for the homepage is written using mocked API data and the authentication context to check if the correct elements are displayed on the UI. The test ensures that the displayed content matches the expected result based on the authentication status and the fetched data. Figure 80 shows an example of how to write a test for the homepage using mocked data and AuthContext.

```tsx
// __tests__/HomeScreen-test.tsx

describe('HomeScreen (Index)', () => {
  beforeEach(() => {
    (useAuth as jest.Mock).mockReturnValue({
      authState: {
        user: {
          name: 'Jane Doe',
        },
      },
    });
  });

  it('renders welcome text and metrics correctly', async () => {
    const { getByText } = render(<Index />);

    await waitFor(() => {
      expect(getByText('Welcome,')).toBeTruthy();
      expect(getByText('Jane')).toBeTruthy(); // Should be first name only
      expect(getByText('Exercises\nCompleted')).toBeTruthy();
      expect(getByText('Average\nPain Level')).toBeTruthy();
      expect(getByText('0.0')).toBeTruthy();
      expect(getByText('10')).toBeTruthy();
    });
  });
});
```

Figure 80: Testing the homepage with mocked data and AuthContext

```
PASS __tests__/HomeScreen-test.tsx
  HomeScreen (Index)
    ✓ renders welcome text and metrics correctly (228 ms)
```

Figure 81: Running the HomeScreen test using Jest

This runs the HomeScreen test and shows the results in Figure 81. If the test passes, it confirms the component is working correctly with the mock data and authentication context. Figure 82 shows how to run the test and the result when it passes.

6.3. Evaluation

For the backend, tests were completed using MSTest in VS Code. These tests checked if everything worked as expected, using a simple setup: Arrange (set up), Act (run), and Assert (check results). The Test Explorer in VS Code helped track and fix any issues.

For the frontend, Jest was used to test the app's interface. It made sure the app displayed the right information based on user login and data. Mocking was used to simulate real-life data, like API responses and login states.

In summary, the technical testing worked well in confirming that both the frontend and backend were performing as expected. The inclusion of usability testing, while out of scope for this project, has been identified as a priority step in the further development of the application. This will provide insight into how easy the app is to use by both the physiotherapist and the patients and how well it meets their needs .This knowledge will be used to further improve the app’s design and overall user experience.

7. Project Management

This chapter explains how the project was managed and how well the student followed the project guidelines. It covers the different phases, starting from the project idea and moving through to gathering requirements and creating the project specification, design, implementation, and testing. It also discusses how tools like Notion, GitHub, and Miro helped manage the project.

7.1. Project Phases

The project life cycle is typically divided into distinct phases that help organise and manage the process from start to finish. Below is an explanation of each phase and the common issues that arose during each.

7.1.1. Requirements

In the Requirements phase, sufficient reliable data was difficult to find, and the constantly changing nature of the project made the requirements quickly outdated or irrelevant. As a result, constant adjustments and reassessments were required. These adaptations ensured that the project aligned with evolving goals and user needs.

7.1.2. Research

During the research phase, the abundance of available information was initially overwhelming, making it difficult to focus on the key topics and narrow them down. Sorting through various sources took considerable time, and isolating key insights was a challenge. However, it laid a solid foundation for more informed decision-making and development.

7.1.3. Design

In the design phase, iteration played a key role. Early uncertainty around the design, along with difficulties in finalising a clear plan for the project, led to significant experimentation. Revisions ultimately focused on ensuring that the basic concept of a user-centred design, with a strong emphasis on intuitivity, was achieved before handling more complex ideas in later iterations.

7.1.4. Implementation

In the Implementation phase, issues with changing tech stacks caused delays and required adjustments. Technical challenges, such as compatibility problems, arose from switching to new tools and technologies, requiring parts of the project to be rethought. This made the development process slower and more complicated than initially planned.

7.1.5. Testing

In the Implementation phase, issues with changing tech stacks caused delays and required adjustments. Technical challenges, such as compatibility problems, arose from switching to new tools and technologies, requiring parts of the project to be rethought. This made the development process slower and more complicated than initially planned.

7.2. Project Management Tools

To stay organised and keep track of progress throughout the project, a few different tools were used at different stages.

7.2.1. Notion

Notion became the main tool for organising notes and saving useful code snippets, like those shown in Figure 82, especially in the later stages of the project. It helped quickly find and reuse information, saving time and keeping things clear.

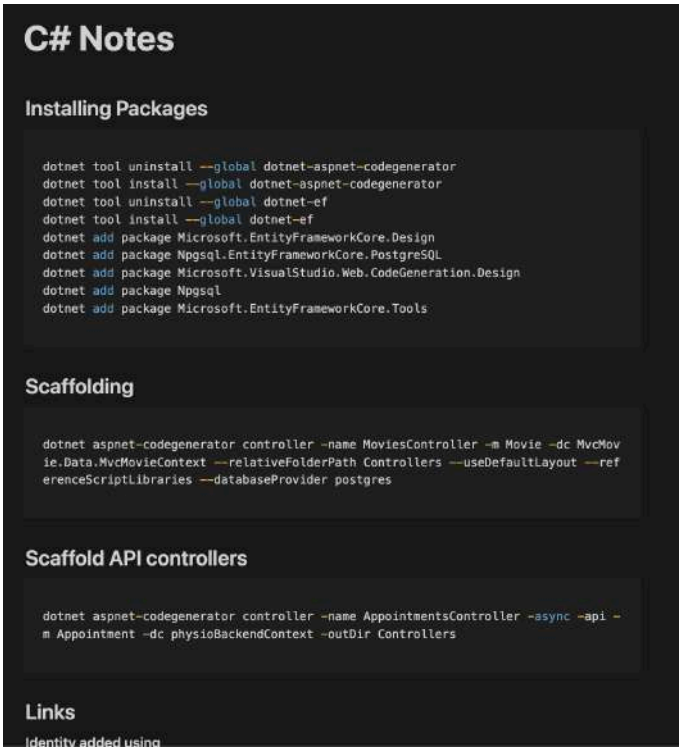


Figure 82: Notion Page

7.2.2. GitHub

GitHub was used only at the end of the project to upload and store the final version of the code. During development, only local work was done, and GitHub was not used for version control regularly.

7.2.3. Miro

In the early planning stages, Miro was used to brainstorm ideas and lay out the basic requirements for the project. The visual layout helped organise thoughts. As the project progressed, Notion was used for planning and note-taking.

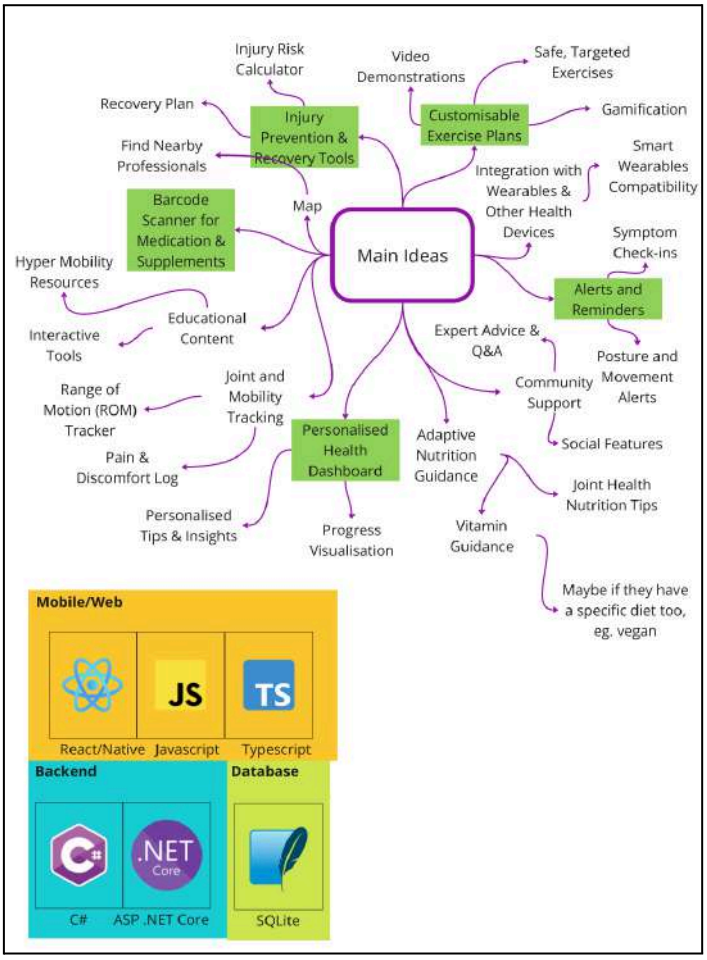


Figure 83: Miro Board - Brainstorming Frame

In the Brainstorming Frame, shown in Figure 83, different ideas are shared and looked at, helping to explore various possibilities for the project. This frame is used to ensure that many options are considered before deciding on a direction for the project.

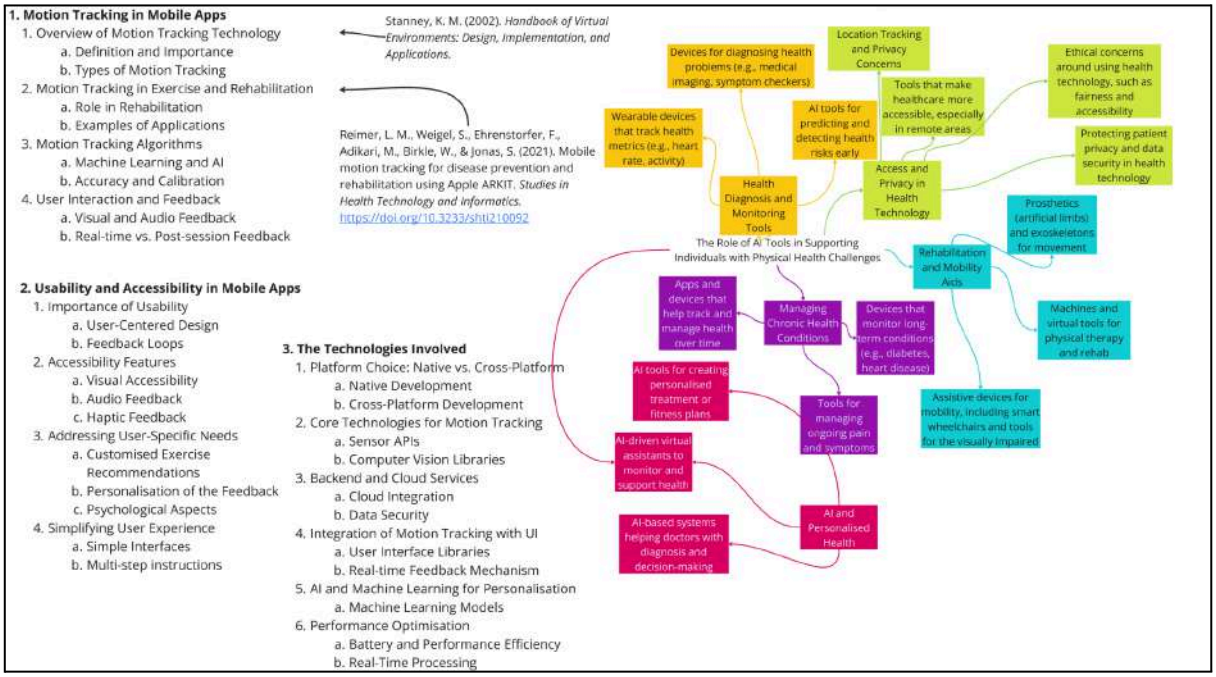


Figure 84: Miro Board - Researching Frame

The Research Frame, shown in Figure 84, organises and shows important information that has been collected during the research process. This information is put together in a way that makes it easy to understand, helping to ensure the project is based on solid facts and insights.

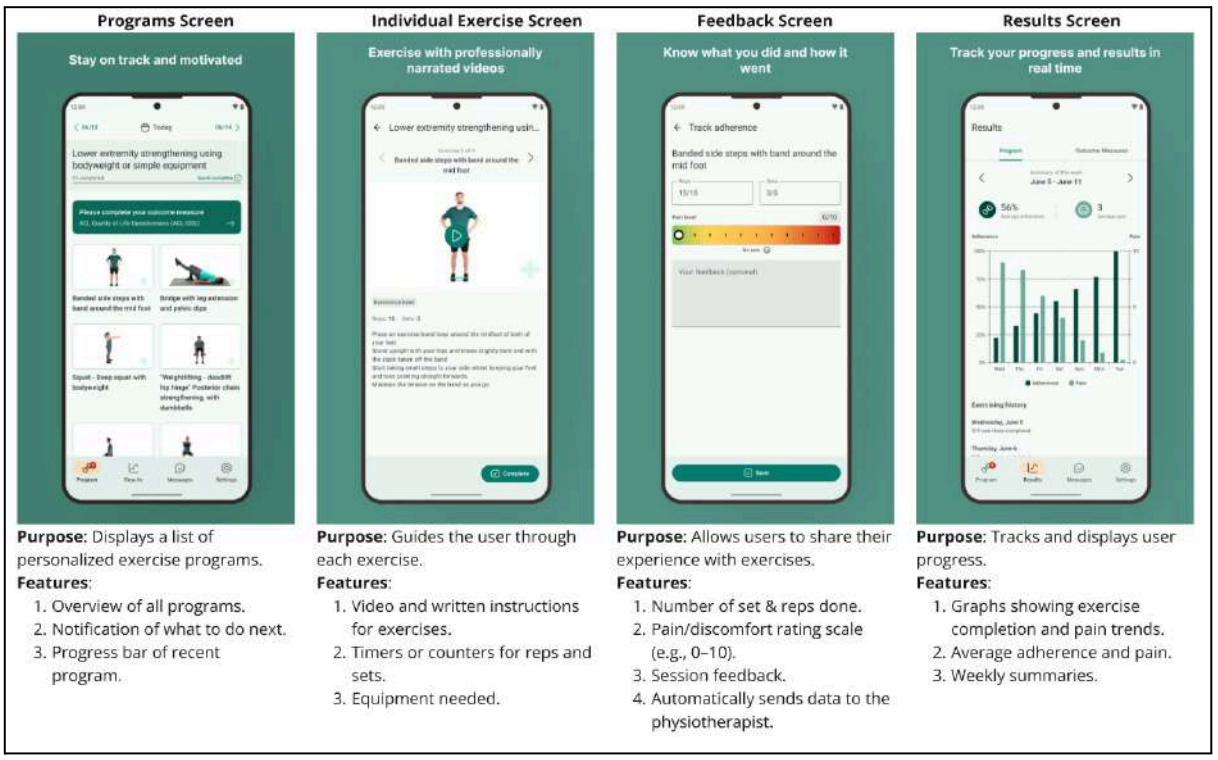


Figure 85 Miro Board - PhysiApp Frame

The PhysiApp Frame, shown in Figure 85, focuses on the features and design elements of the PhysiApp, which has been reviewed for its approach to fitness and rehabilitation. This frame looks at how the app's functionality and user experience are designed to support users in achieving their health and fitness goals.

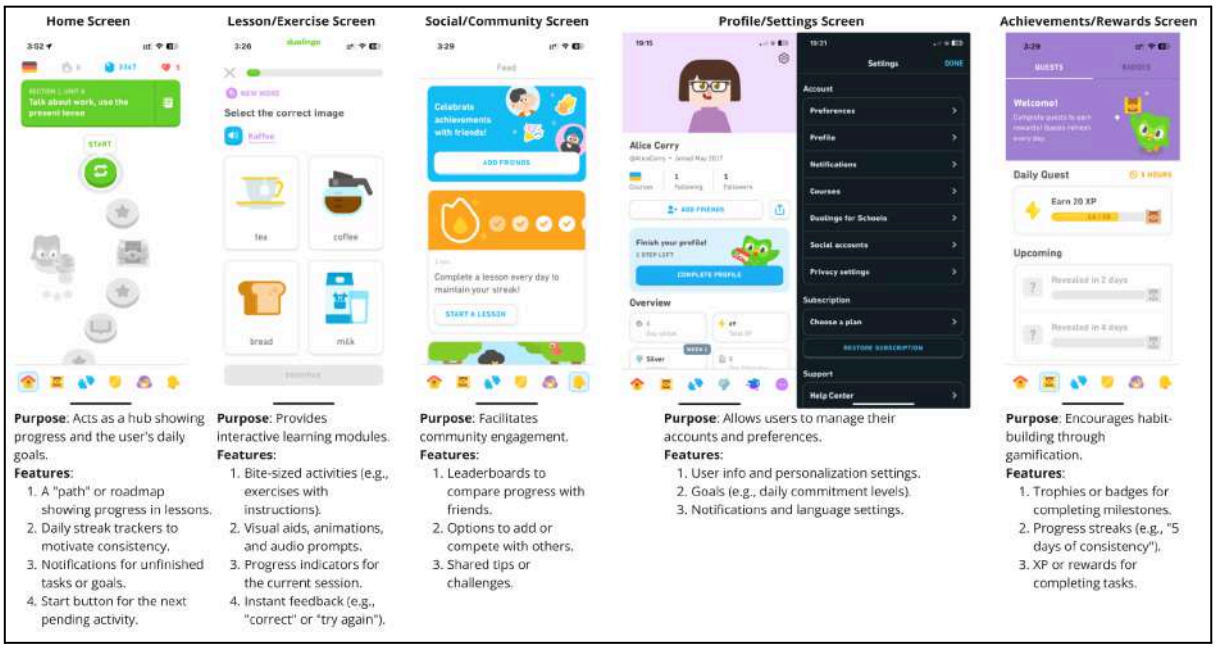


Figure 86: Miro Board - Duolingo

The Duolingo Frame, shown in Figure 86, examines the features of the Duolingo app, analysing how it keeps users engaged through fun, game-like elements. It highlights the app's effective use of gamification to motivate users while they enjoyably learn a new language.

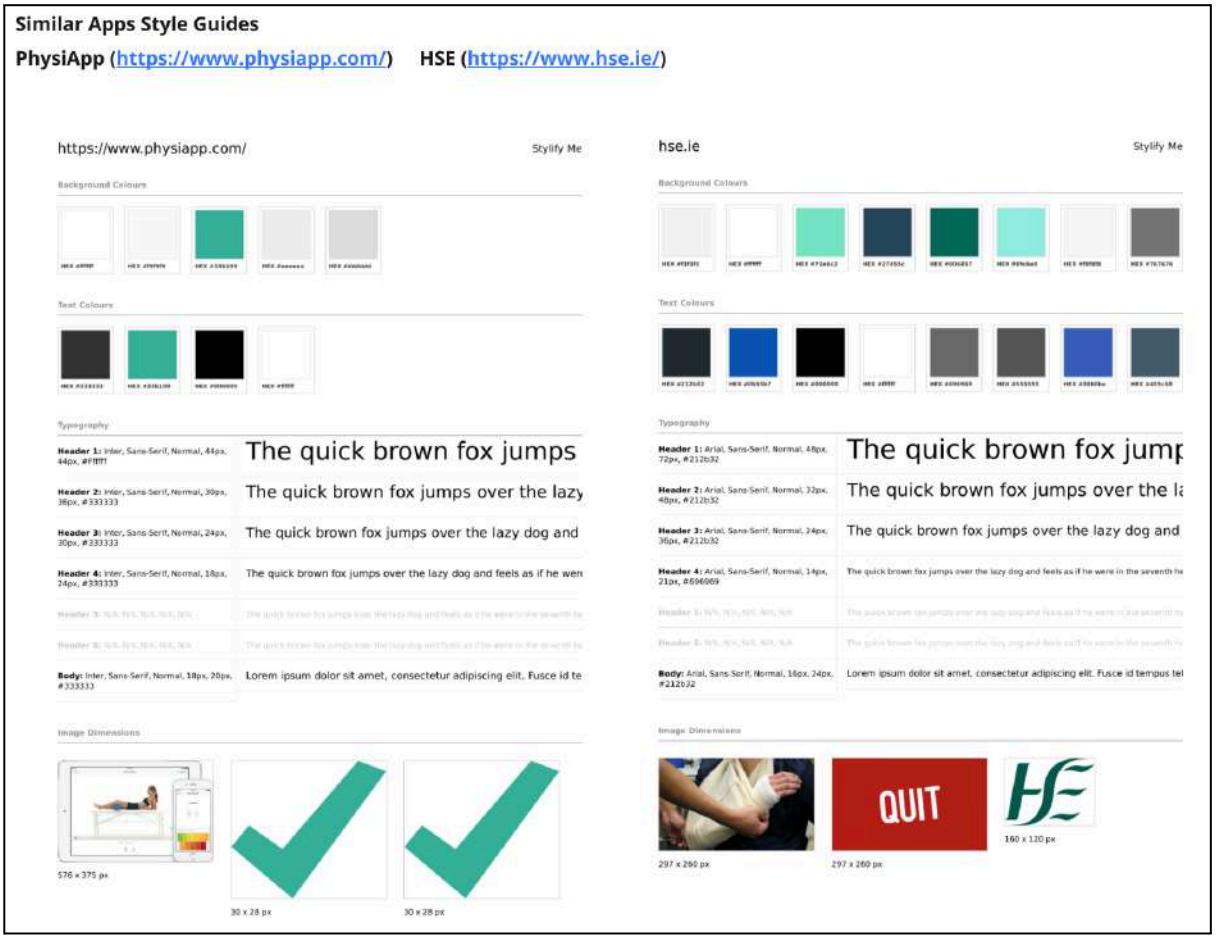


Figure 87: Miro Board - Designing Frame

The Design Frame, shown in Figure 87, reviews style guides from existing apps and websites, focusing on design elements like colour schemes, typography, and layout. This frame provides insight into design trends and best practices that influence the overall look and feel of the final product.

8. Reflection

8.1 Your Views on the Project

Working on this project was both challenging and rewarding. Given the chance to experience the whole software development process, from gathering requirements to deploying the final product. What stood out most was how much personal growth was experienced, not just in technical skills, but also in confidence, adaptability, and problem-solving. The comfort zone was pushed, mainly when new technologies were used, and pride was taken in the achievements.

8.2 Completing a Large Software Development Project

This project demonstrated the largest scope and complexity of anything attempted before. It showed how important it is for a clear plan to be in place, for organisation to be maintained, and for flexibility to be exercised when changes occur. Working on a long-term, multi-phase project taught me how big tasks can be broken down into smaller, manageable ones. This approach helped keep things on track, despite unexpected problems.

Many challenges were faced along the way, including significant changes in the technologies used. C# and ASP.NET Core were initially chosen for the backend, then Python and Flask were tried for a simpler framework. JavaScript and Express were also tested to see how they might work for full-stack development. Ultimately, the decision was made to return to C# and ASP.NET Core, as it offered the best support for both MVC and API structures, which worked best for the project. PostgreSQL was also switched to SQLite, as it was easier to set up with ASP.NET Core and allowed more focus to be placed on coding rather than configuration.

These changes helped realise the importance of flexibility and made decisions based on what was best for the project.

8.3 Working with a Supervisor

Working with a supervisor was very helpful, particularly at the start of the project. Advice on how to make the system user-friendly was given by their specialisation in user experience (UX) design. Although they did not focus on the development side, the guidance provided by the early stages helped create a strong foundation for the project. As the project became more technical, more freedom was given to take on responsibility, which was considered a great learning experience.

8.4 Technical Skills

One of the biggest challenges during the project was learning a new backend language and framework, C# and ASP.NET Core. At first, it was difficult, but now, a much better understanding of server-side development has been gained. Both MVC patterns and RESTful APIs were learned to be used effectively, and experience was gained with integrating databases, user authentication, and deployment.

Switching technologies during the project was not easy, but it offered experience on evaluating different tools and deciding what would work best for the project. This helped encourage strategic thinking about the technical choices that were made.

8.5 Further Competencies and Skills

This project also helped to improve other vital areas. Better research skills in new technologies were developed, and more informed choices were made. Time management, self-motivation, and writing clear documentation were all improved skills. The ability to explain complex ideas, whether in code comments or written reports, also became stronger.

In addition to technical skills, problem-solving abilities were improved, especially when unexpected bugs were encountered or new features were designed under pressure. These technical and non-technical skills are expected to be valuable in future projects and roles.

8.6. Future Plans

Due to the nature of this project and limitations such as time constraints, limited resources, and the focus on building basic features first, some advanced features couldn't be added at this stage. However, with more time, resources, and future development phases, FlexiCare has the potential to grow significantly. The following sections outline key areas that are planned for improvement to make the app more engaging for patients, improve treatment results, and provide a better overall experience.

8.6.1. Improving Exercise Tracking and Patient Engagement

In future versions of FlexiCare, improvements are planned for tracking exercises. Better and more engaging exercise tracking is crucial for keeping patients motivated and helping them achieve the best results from their rehabilitation.

One idea is to add a system that provides real-time feedback during exercises. For example, the app could give feedback through sound, images, or vibrations to help patients correct their technique in the moment. This would help patients stay motivated and ensure they are performing exercises correctly.

Additionally, plans include connecting the app with fitness trackers or smartwatches. These devices can track metrics like heart rate, steps, and calories burned, providing more data to personalise rehabilitation plans for each user. This data can create more specific and individualised plans, helping patients feel more accountable for their progress. Goal-setting features (such as SMART goals) are also planned to help patients track progress and stay engaged. For example, the app could track how many push-ups a patient does, and as they complete more, they could earn rewards.

Another planned feature is the integration of social elements, such as forums or group challenges. These would allow patients to connect with others experiencing similar journeys, fostering a sense of support and motivation. [18]

8.6.2. Using Gamification to Increase Motivation

FlexiCare will incorporate gamification in future versions to make rehabilitation more enjoyable and rewarding. Gamification involves adding game-like elements to non-game situations to engage users and enhance their experience. At the same time, FlexiCare has some aspects of this, such as the streak system, with a longer timeline, and further developments could be made. [19]

One idea is to introduce a reward system where users can earn points, badges, or other rewards for reaching specific milestones. In addition, features like leaderboards and community challenges are planned, allowing patients to compete or collaborate with others, which could boost motivation. These elements would foster a sense of teamwork and friendly competition, helping patients feel more connected to others on the same recovery journey. [20]

Finally, personalising the app will be a key focus. Real-time feedback will be provided, progress will be shown through visual indicators, and goals will be adjusted based on individual patient performance. This will ensure that patients are consistently challenged but not overwhelmed, helping them stay on track with their rehabilitation. [21]

8.6.3. Making the App Accessible for Everyone

FlexiCare must be accessible to all users, regardless of their physical abilities or familiarity with technology. The app was designed with ease of use in mind for everyone, including people with disabilities.

However, in the future, to improve accessibility, features like voice commands, switch controls, and options for high-contrast or monochrome displays will be included. These features will support users with visual impairments. Additionally, the app will be compatible with adaptive devices, allowing users to interact with it in ways that best suit their needs. [22]

The app will also be designed to be intuitive, ensuring users can easily navigate it without confusion. By gathering feedback from a diverse group of users during development, any potential accessibility issues can be identified and addressed.

Furthermore, educational resources will be offered in multiple languages to help users understand how to perform their prescribed exercises correctly. These resources will be available in various formats to accommodate the needs of a broad range of users. [23]

9. Conclusion

The Flexicare application was developed as an electronic management system for physiotherapy. It incorporates an intuitive web application for physiotherapists and a mobile app for patients, transforming more traditional examples of physiotherapy practice.

The application was developed using an iterative SCRUM approach, which divides a large project into smaller sections called sprints. Key methods employed included existing product evaluation, key user surveys and feasibility testing.

The FlexiCare backend development utilised Microsoft's ASP.NET Core MVC framework, C# code and Entity Framework Core, Microsoft's Object-Relational Mapping (ORM) tool for mapping databases and SQLite. Key components included the FlexicCare Manager for physiotherapists and administrators, role-based authentication using Microsoft Identity and a separately designed FlexiCare API, which used JSON Web Token (JWT) authentication and Swagger Support for the visual interface.

The frontend FlexiCare mobile development utilised React Native and Expo. Key tools included Visual Studio Code for coding, Insomnia for API testing and Figma for UI design.

The benefits of FlexiCare to the physiotherapist include the simplification of administrative tasks, the provision of secure and convenient data management, and enhancing patient engagement. The research and development process for Fleixcare has incorporated user input from the physiotherapist, with questionnaires utilised to provide recommendations that both directed the design and functionality of the application.

For the patient, Flexicare delivers improved coordination and communication with the physiotherapist. It also allows patients to become active participants in their rehabilitation process. Incorporation of features such as exercise tracking, progress monitoring, and instant feedback highlights the focus on improving patient outcomes and satisfaction.

Further enhancements of Flexicare will include an increased level of gamification features and improved accessibility for patients. Future ongoing engagement with the physiotherapists will be required to ensure that the application is adjusted on a continuous basis to meet their changing needs, as well as incorporating the latest technologies and methodologies to continue refining their administrative and clinical services.

References

1. Anwar, N., Maratis, J., Adhy, D. R., Hermawan, R., & Hadi, M. A. (2022). Mobile Application Design for Online Physiotherapy Services. Atlantis Press.

2. Richardson, J., Letts, L., Sinclair, S., Chan, D., Miller, J., Donnelly, C., Smith-Turchyn, J., Wojkowski, S., Gravesande, J., & Loyola Sánchez, A. (2021). Using a Web-Based App to Deliver Rehabilitation Strategies to Persons With Chronic Conditions: Development and Usability Study. JMIR Rehabilitation and Assistive Technologies, 8(1).

3. AL Anazi Fayez Khalaf, Bin Thari Razan Rashed, AL Aloula Ali Suliman, AL Azmiy Barakat Shumilan and AL Jarallah Majed Khalid. OPTIMIZING CARE THROUGH UNIFIED SYSTEMS: A CRITICAL REVIEW OF INTEGRATED MANAGEMENT ENHANCEMENTS BETWEEN HEALTH INFORMATION SYSTEMS AND NURSING PRACTICE. International Journal of Development Research. 2023.

4. Dineen-Griffin, S., Garcia-Cardenas, V., Williams, K., & Benrimoj, S. I. (2019). Helping patients help themselves: a systematic review of self-management support strategies in primary health care practice. *PloS one*, *14*(8), e0220116.

5. Martínez, N., Connelly, C. D., Pérez, A., & Calero, P. (2021). Self-care: A concept analysis. *International journal of nursing sciences*, *8*(4), 418-425.

6. Santos, R., & Pires, D. (2024). Current use of patient-reported outcome measures by musculoskeletal physiotherapists in Portugal. *Journal of Back and Musculoskeletal Rehabilitation*, *37*(6), 1479-1488.

7. Jalali, M. S., Russell, B., Razak, S., & Gordon, W. J. (2019). EARS to cyber incidents in health care. *Journal of the American Medical Informatics Association*, *26*(1), 81-90.

8. Hiller, A., & Delany, C. (2018). Communication in physiotherapy: challenging established theoretical approaches. *Manipulating Practices: A Critical Physiotherapy Reader. Oslo: Cappelen Damm Akademisk*.

9. Fenyuk, A. (2024). How to Develop a Physical Therapy App: Key Steps for Success. Stormotion.

10. Jane Patterson (2023, June 5). The Importance of Regular Software Updates in Cybersecurity

11. Dawson-Rose, C., Cuca, Y. P., Webel, A. R., Báez, S. S. S., Holzemer, W. L., Rivero-Méndez, M., et al. (2016). Building trust and relationships between patients and providers: an essential complement to health literacy in HIV care. *J Assoc Nurses AIDS Care*, 27(5), 574–584.

12. App Store. (2015, December 1). PhyiApp.

https://apps.apple.com/us/app/physiapp/id1047722007

13. MedicalDirector Helix - Software features. (n.d.). MedicalDirector. https://www.medicaldirector.com/products/helix/features

14. Strangehelix. (n.d.). freud v2: AI Mental Health App - Mindfulness Metrics UIUX. Dribbble. https://dribbble.com/shots/25513079-freud-v2-AI-Mental-Health-App-Mindfulness-Metrics

15. Paperpillar. (n.d.). Task Management App. Dribbble. https://dribbble.com/shots/24600588-Task-Management-App

16. Kubalczyk, M. (n.d.). *Inventory app: Profile*. Dribbble. https://dribbble.com/shots/24566846-Inventory-app-Profile

17. Asal Design. (n.d.). *AdaKita Onboarding Screen*. Dribbble. https://dribbble.com/shots/19942973-AdaKita-Onboarding-Screen

18. Chen, J., & Wang, Y. (2021). Social media use for health purposes: Systematic review. *Journal of Medical Internet Research*, *23*(5), e17917.

19. Al-Rayes, S., Al Yaqoub, F. A., Alfayez, A., et al. (2022). Gaming elements, applications, and challenges of gamification in healthcare. *Journal Name*, *Volume*(Issue).

20. Polskii, M. (2024, May 24). Building a reward system in mobile apps: Best practices for gamification. *InAppStory*.

21. EPR Staff. (2025, March 27). The future of app digital marketing: Personalization is the key to success. *EPR*.

22. Skynet Technologies. (2024, November 5). Mobile accessibility trends: Best practices for inclusive app design. *Skynet Technologies*.

23. Council of Europe. (n.d.). Platform of resources and references for plurilingual and intercultural education. *Council of Europe*.

Appendices

Appendix A – App Code Repository

A GitHub repository containing the frontend of the application.

https://github.com/ac-png/physioApp

Appendix B – Backend Code Repository

A GitHub repository containing the backend of the application

https://github.com/ac-png/physioServer

Appendix C – Miro Board

Miro board project management link

https://miro.com/app/board/uXjVI85lj98=/?share_link_id=509385298488