



Fall 2024
CPIT499 Senior Project 1
OBOUR: A Substance-Use Recovery System

Prepared By:

Alaa Khalid Asghar	2105843
Reema Samir Hesamudin	2106126
Bakirah Yasir Alseari	2107854

Supervised By:
Dr. Taghreed Bagies

Date:
5 May 2025

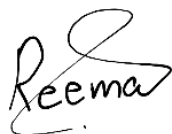
DECLARATION BY AUTHORS

“I/we certify that this work has not been accepted in substance for any degree and is not concurrently being submitted for any degree other than that of BS Information Technology being studied at King Abdulaziz University, Jeddah. I/we also declare that this work is the result of my/our own findings and investigations except where otherwise identified by references and that I/we have not plagiarized another’s work”.

Reema Samir Hesamudin

Alaa Khalid Asghar

Bakirah Yasir Alseari



DECLARATION BY SUPERVISOR

I, the undersigned hereby certify that I have read this project report and finally approve it with recommendation that this report may be submitted by the authors above to the final year project evaluation committee for final evaluation and presentation, in partial fulfillment of the requirements for the degree of BS Information Technology at the Department of Information Technology, Faculty of Computing and Information Technology, King Abdulaziz University, Jeddah.

Dr. Taghreed Bagies



Abstract

Substance use disorder is a growing concern in Saudi Arabia, with an increasing number of individuals struggling with addiction and facing significant challenges during recovery. While recovery tools exist, they predominantly cater to English speakers, neglecting the cultural and linguistic needs of Arabic-speaking communities. To address this gap, the *Obour* application is developed to support Arabic-speaking post-rehabilitation patients in their recovery journey. The platform offers culturally relevant features, including personalized recovery plans, progress tracking with achievement badges, activity reminders, emotional support through inspirational and religious quotes, and peer connection via SMS for additional support.

Additionally, *Obour* seeks to raise awareness about substance use disorder within communities, educate family members, and reduce stigma associated with addiction, fostering a supportive environment for patients. The project employs the Waterfall methodology to ensure a clear, structured, and high-quality development process, with initial focus on requirements gathering and system design. Targeted at post-rehabilitation patients in Saudi Arabia, *Obour* aims to improve long-term recovery outcomes, facilitate social reintegration, and empower individuals to build healthier lives.

Table of Contents

Abstract.....	3
Chapter 1: Introduction	11
1.1 Background and Introduction.....	11
1.2 Problem Definition	11
1.3 Aims and objectives.....	12
1.4 Scope	13
1.5 Suggested Solution.....	13
1.6 Project Stakeholders	14
1.7 Methodology.....	15
1.8 Schedule of Events Using Gantt Chart for The Project.....	16
1.9 Designing Tools.....	17
1.10 Contribution Breakdown.....	18
1.10.1 Individual Contributions	18
1.10.2 Shared Contributions	19
Chapter 2: Background and Similar Applications	20
2.1 Background.....	20
2.2 Similar Applications and Papers.....	20
2.2.1 First Paper: A Mobile App to Enhance Behavioral Activation Treatment for Substance Use Disorder (LETS ACT system).....	20
2.2.2 Second Paper: HealthCall delivered via smartphone to reduce co-occurring drug and alcohol use in HIV-infected adults (HealthCall system).....	21
2.2.3 WEconnect Application	22
2.2.4 Sober Time Application	23
2.2.5 I Am Sober Application	23
2.3 Literature Review:	24
2.4 Comparative Study.....	25
Chapter 3: Analysis.....	27
3.1 Background.....	27
3.2 Functional Requirements.....	27
3.3 Non-Functional Requirements	28
3.4 Initial Design	30
3.4.1 Use Case Diagram.....	30
3.4.2 Use Case Description	31
3.4.3 Mapping Requirements.....	43
3.5 Sequence Diagram.....	44
3.5.1 High level Sequence Diagram.....	44
3.5.2 Generate Activity Plan	45
3.5.3 Manage Activity Plan	46
3.5.4 Notify Activity	47
3.5.5 Add Emotional State	48

3.5.6 Generate Supportive Quote	49
3.5.7 Display Supportive Quote	50
3.5.8 Manage List of Peer's Contact Number	51
3.5.9 Send SMS to Peer's Contact Number	52
3.5.10 View Educational Information	53
3.5.11 View Achievement Badges	54
Chapter 4: System Design	55
4.1 Background	55
4.2 Class Diagram	56
4.3 ER Diagram	58
4.4 Normalized Relational Shema	59
Chapter 5: Implementation.....	60
5.1 Introduction	60
5.2 Tools	60
5.3 System Implementation	60
5.3.1 System Functionalities	60
5.3.2 Firebase	92
5.4 Code Debugging and Troubleshooting Issues	106
5.5 Packaging and Documentation	107
5.6 Conclusion	108
Chapter 6: Testing	109
6.1 Introduction	109
6.2 Functional Testing	109
6.2.1 Unit testing	109
6.2.2 Integration testing	116
6.2.3 System Testing	125
6.3 Non-functional testing	128
6.3.1 Compatibility Testing	128
6.4 Conclusion	128
Chapter 7: Results and discussions	129
7.1 Introduction	129
7.2 Results	129
7.2.1 Login and Signup (Generated activity plan)	129
7.2.2 Display Supportive Quote	130
7.2.3 Achievement Badges Screen	131
7.2.4 Peer Contact Screens	132
7.2.5 Media Screens	132
7.2.6 Profile Screens	133
7.3 Accomplished Objectives	134
7.4 Project Limitation	134
Chapter 8: Conclusion and future work	135

8.1 Project Conclusion	135
8.2 Future Work	136
8.3 Conclusion.....	136
Reference	138
Appendix A.....	139
A.1 Data Gathering Technique	139
A.1.1 Interview.....	139
A.1.2 Survey:.....	141
A.2 Changes from Report 1.....	144

List of Figures

Figure 1: Suggested Solution	14
Figure 2: Waterfall Model	16
Figure 3: Schedule of events using Gantt Chart for the project.....	16
Figure 4: Use Case Diagram	30
Figure 5: Sequence Diagram (High Level-up).....	44
Figure 6: Sequence Diagram (Generate Activity Plan)	45
Figure 7: Sequence Diagram (Manage Activity Plan)	46
Figure 8: Sequence Diagram (Notify Activity)	47
Figure 9: Sequence Diagram (Add Emotional State)	48
Figure 10: Sequence Diagram (Generate Supportive Quote)	49
Figure 11: Sequence Diagram (Display Supportive Quote)	50
Figure 12: Sequence Diagram (Manage List Of Peer's Contact Number)	51
Figure 13: Sequence Diagram (Send SMS to Peer's Contact Number)	52
Figure 14: Sequence Diagram (View Educational Information)	53
Figure 15: Sequence Diagram (View Achievement Badges)	54
Figure 16: Class Diagram	56
Figure 17: ER Diagram	58
Figure 18: Relational Schema	59
Figure 19: generateWeeklySchedule method	62
Figure 20: generateWeeklySchedule method	62
Figure 21: pushActivityLater method	63
Figure 22: buildTodayActivitiesHeader method	65
Figure 23: buildWeekDaysSelector method	66
Figure 24: buildActivitiesList method	67
Figure 25: buildSearchField method	68
Figure 26: class SmsHelper	69
Figure 27: Adding a Contact	70
Figure 28: Saving Contacts	71
Figure 29: EditContactScreen	71
Figure 30: Achievement Badges	72
Figure 31: fetchCompletedActivities	73
Figure 32: build	74
Figure 33: Notification remainder	75
Figure 34: scheduleSingleNotification	76
Figure 35: buildMediaScreen	77
Figure 36: streamBuilder(Video)	78
Figure 37: YTController	79
Figure 38: buildHeader(video)	80
Figure 39: playPodcasts	81
Figure 40: formatDuration	81
Figure 41: streamBuilder(podcasts)	82
Figure 42: buildHeader(podcasts)	83
Figure 43: _buildHeader(text)	85
Figure 44: streamBuilder(text)	86

Figure 45: onMoodSelected()	87
Figure 46: _buildMoodSelector()	88
Figure 47: ChatGPTService.generateQuote()	89
Figure 48: ChatGPTService	89
Figure 49: Request Details1	89
Figure 50: Request Details2	90
Figure 51: Request Body	90
Figure 52: Error Handling	91
Figure 53: Firebase Authentication	92
Figure 54: signup code	93
Figure 55: Firestore Patient collection	93
Figure 56: Creation of Activates collection code	94
Figure 57: Firestore Activates collection	95
Figure 58: Adding a New Scheduled Date	95
Figure 59: Updating a Scheduled Date	96
Figure 60: Firestore ScheduledDates collection	96
Figure 61: Creation of Availability collection code	97
Figure 62: Firestore Availability collection	97
Figure 63: Creation of Quotes collection code	98
Figure 64: Firestore Quotes collection	99
Figure 65: Creation of contactPeer collection	100
Figure 66: Edit contactPeer from firestore	100
Figure 67: Delete contactPeer from firestore	101
Figure 68: Fetch contactPeer from firestore	101
Figure 69: Firestore contactPeer collection	102
Figure 70: Fetch EducationalContent from Firestore	103
Figure 71: Firestore EducationalContent collection	103
Figure 72: Fetch Videos from Firestore	104
Figure 73: Firestore Videos collection	104
Figure 74: Fetch Podcasts from Firestore	105
Figure 75: Firestore Podcasts collection	105
Figure 76: Firebase Storage	106
Figure 77: Main and Screens	107
Figure 78: All Screens	107
Figure 79: All Screens	108
Figure 80: images and fonts	108
Figure 81: Signup test	109
Figure 82: signup test result	110
Figure 83: Signup test email	110
Figure 84: signup test email result	110
Figure 85: login test	111
Figure 86: login test result	111
Figure 87: Activity test	112
Figure 88: Activity test result	112
Figure 89: Priority test result	113
Figure 90: Priority test	113

Figure 91: Saving Free Time Availability test result.....	113
Figure 92: Saving Free Time Availability test.....	113
Figure 93: Saving Contacts to Firestore test result	114
Figure 94: Saving Contacts to Firestore test	114
Figure 95: Add peer contact test	115
Figure 96: Add peer contact test result	115
Figure 97: driver class.....	117
Figure 98: signup_test (initilization).....	117
Figure 99: signup_test (Run)	117
Figure 100: signup_test (UI interaction).....	118
Figure 101: signup_test (success)	119
Figure 102: add_activity_test(Flutter integ test binding).....	120
Figure 103: add_activity_test (init firebase)	120
Figure 104: : add_activity_test (wait by key)	120
Figure 105: : add_activity_test (wait firebase auth)	121
Figure 106: : add_activity_test (Run)	121
Figure 107: : add_activity_test (UI interactions)	122
Figure 108: : add_activity_test (firestore interactions).....	123
Figure 109: add_activity_test (UI validation).....	124
Figure 110: : add_activity_test (success).....	125
Figure 111: Login and Signup Process (Generated activity plan)	130
Figure 112: Display Supportive Quote	131
Figure 113: Achievement Badges Screen	131
Figure 114: Peer Contact Screens	132
Figure 115: Media Screens	133
Figure 116: Profile Screens.....	133
Figure 117: Appendix A (Survey Q1 & Q2)	141
Figure 118: Appendix A (Survey Q3 & Q4)	142
Figure 119: Appendix A (Survey Q5 & Q6)	142
Figure 120: Appendix A (Survey Q7 & Q8)	143
Figure 121: Appendix A (Survey Q9 & Q10)	143

List of Tables

Table 1: Designing tools, languages & libraries, and IT technology.....	17
Table 2: Literature Review	24
Table 3: Comparative Study	25
Table 4: Use Case Description (Register).....	31
Table 5: Use Case Description (Login)	32
Table 6: Use Case Description (Generate Activity Plan)	33
Table 7: Use Case Description (Manage Activity Plan)	33
Table 8: Use Case Description (Add Emotional State)	35
Table 9: Use Case Description (Generate Supportive Qoute)	35
Table 10: Use Case Description (Display Supportive Qoute)	37
Table 11: Use Case Description (Manage List of peer's contact numbers)	38
Table 12: Use Case Description (Send SMS to peer's contact number)	39
Table 13: Use Case Description (View Achievment Badges)	41
Table 14: Use Case Description (View Educational Information)	42
Table 15: Mapping Requirements.....	43
Table 16: System Testing.....	126
Table 17: Compatibility Testing	128
Table 18: Appendix A (Interview).....	139
Table 19: Changes from report 1	144

Chapter 1: Introduction

1.1 Background and Introduction

Drug abuse, or substance use disorder, is a condition that affects a person's ability to control their use of legal or illegal drugs, impacting both their brain and behavior [1]. Recently, the percentage of people falling into drug addiction has been increasing in Saudi Arabia, with many individuals facing significant psychological and physical challenges during their recovery. While various tools have been developed to help people manage their daily activities and support recovery, most are designed for English speakers [2].

To address this issue, our project, *Obour*, designed to support post-rehabilitation patients in Saudi Arabia during their recovery journey by providing a culturally and linguistically relevant platform. This initiative seeks to enhance recovery outcomes, reduce relapse rates, promote reintegration into society, and create a supportive environment for the patient.

1.2 Problem Definition

Post-rehabilitation patients face numerous challenges, including disruptions in body functions such as memory issues leading to forgetfulness, alongside psychological struggles like depression, anxiety, and stress. They often struggle with relapse after recovery and lack societal support to sustain their recovery plans. Many Arabic-speaking individuals search for an app that aligns with their language and beliefs but find that most recovery platforms cater to English-speaking users, neglecting their unique cultural, religious, and linguistic needs. Additionally, these individuals may face challenges in social reintegration and struggle to gain support from their community, which often lacks awareness of substance use and its associated risks.

1.3 Aims and objectives

Aim

The main aim of our project, *Obour*, is to provide an application that supports Arabic-speaking post-rehabilitation patients by addressing the unique challenges they face, such as memory issues, psychological struggles, and relapse prevention. The platform will encourage post-rehabilitation patients to continue their recovery journey by offering culturally relevant, accessible, and supportive resources. *Obour* also aims to raise awareness about substance use disorder within the community, reduce stigma, and promote understanding of its risks. Additionally, the platform will focus on improving long-term recovery outcomes, facilitating social reintegration, and ensuring patients have access to personalized, effective behavioral treatment strategies that align with their cultural and religious backgrounds.

Objectives

The following are the main objectives of our proposed system:

1. **Personalized Recovery Plans:** Develop individualized recovery plans that align with post-rehabilitation patient's cultural and religious backgrounds, providing structured guidance that encourages positive behavioral changes.
2. **Progress Tracking:** Motivate post-rehabilitation patients by awarding achievement badges for reaching specific sobriety milestones and completing recovery goals. These badges will serve as visual representations of progress, reinforcing post-rehabilitation patients' sense of accomplishment and inspiring them to continue their journey.
3. **Inspirational and Emotional Support:** Generate and display supportive religious messages tailored to post-rehabilitation patients' emotional states and preferred content types (Quotes, Affirmations, Advice, or Motivation), to provide encouragement and mental support.
4. **Notifications and Reminders:** Send notification reminders about scheduled activities helping post-rehabilitation patients stay engaged and mindful of their recovery goals.

5. **Public Awareness and Education:** Raise awareness about substance use disorder in Arabic-speaking communities, reduce stigma, and promote understanding of mental health and recovery.
6. **Connect with peer:** Send an SMS message to the peer's contact number to help build a support network for post-rehabilitation patients, sustaining their recovery and fostering social reintegration.

1.4 Scope

Obour's mission is to provide essential features that help Arabic-speaking post-rehabilitation patients in Saudi Arabia on their recovery journey from substance use disorder. These features include personalized recovery plans, activity reminders, educational resources, progress tracking with achievement badges, and peer connections for support. Additionally, *Obour* will offer emotional support through personalized quotes with religious significance, tailored to users' needs. The project is currently focused on Arabic-speaking post-rehabilitation patients in Saudi Arabia, with plans to expand and improve the platform in the future to enhance the recovery experience.

1.5 Suggested Solution

To address these issues, we've developed *Obour* application to help post-rehabilitation patients who are pursuing recovery from substance use disorder. *Obour* allow patients to enter their age and preferred activities. Subsequently, *Obour* will generate a customized weekly activity plan based on the activities the patients entered, specifying a duration time for each activity. The patient can mark the activities complete during the day, and *Obour* application will send the patient a remainder notification about the incomplete activities. To motivate users, *Obour* rewards achievement badges when a set number of activities have been marked as completed.

Furthermore, *Obour* allows patients to select their preferred types of supportive religious messages—quotes, affirmations, advice, or motivation—and displays them throughout the day, tailored to the patient's emotional state. The app also allows the patient to send an SMS message to a peer, requesting help when needed. Lastly, *Obour* application will provide educational information to enlighten the community, post-rehabilitation patients, and those surrounding them about substance use disorder and its risks, and many other rich information.



Figure 1: Suggested Solution

1.6 Project Stakeholders

- **Project owner:** Faculty of Computing and Information Technology at King Abdul- Aziz university.
- **Project Supervisor:** Dr. Taghreed Bagis.
- **End users:**
 - Post-rehabilitation patients in recovery who seek ongoing support to maintain sobriety and prevent relapse.

- Patients' family members and the broader community can benefit from the educational information provided.
- **Project team:** Alaa Khalid Asghar, Reema Samir Hesamudin, Bakirah Yasir Alseari.

1.7 Methodology

For our senior project, we have chosen to utilize the Waterfall model. This approach follows a linear and sequential method where each phase is completed before moving to the next one. Below are the key reasons for our choice:

1. Clear Structure
2. Defined Requirements
3. Focus on Documentation
4. Quality Assurance

Figure 1.2: Waterfall Model illustrates the sequential phases: *Gather Requirements*, *Design and Analysis*, *Build Prototype*, *Implement*, and *Test*. It highlights the linear flow from one stage to the next, emphasizing the importance of completing each phase before proceeding.

In the first semester, our focus was on the *Gather Requirements* and *Design and Analysis* phases. We gathered stakeholder needs and developed a detailed project design to establish a

solid foundation. In the second semester, we built the prototype, implemented the system, and conducted testing to ensure functionality and reliability.

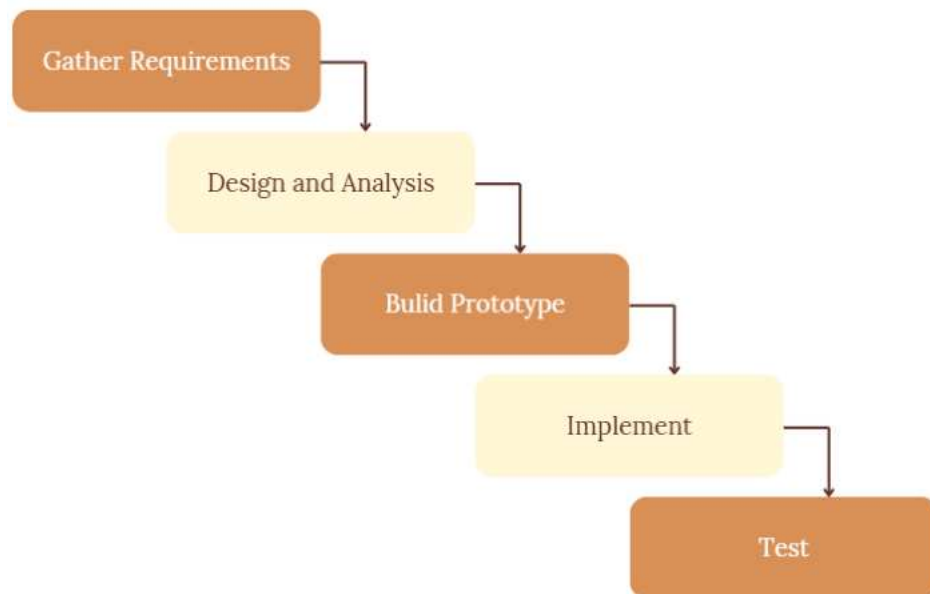


Figure 2: Waterfall Model

1.8 Schedule of Events Using Gantt Chart for The Project

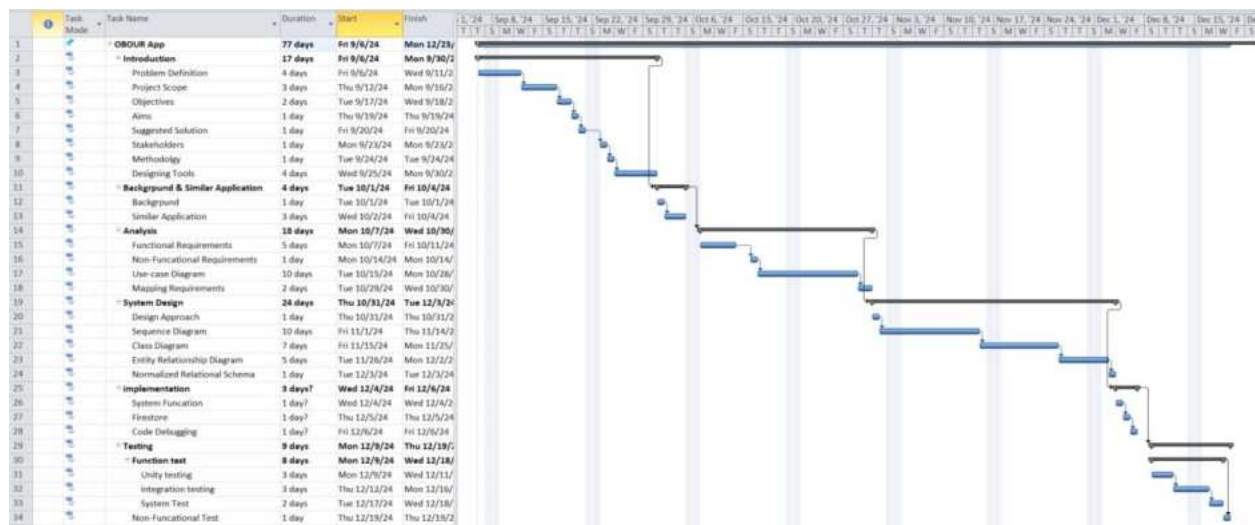


Figure 3: Schedule of events using Gantt Chart for the project

1.9 Designing Tools

Table 1: Designing tools, languages & libraries, and IT technology

Canvas	Utilized to generate exceptional graphics and illustrations for our project.
Figma	Utilized to develop a prototype and design for a collaborative interface
draw.io	Utilized to make diagrams (Use-case, UML, Sequence, ER, and Mapping ER)
Project Plan 365	It's a Project management tool. We utilized it to create Gantt charts.
Google Forms	Create an online form to analyze responses.
Flutter	It is a sophisticated UI toolkit for developing cross-platform mobile, web, and desktop apps that work on both iOS and Android devices. We utilize it to build the application.
Google Drive	A Cloud-based file storage and synchronization service. We used it for Storing, sharing files, and real-time collaboration on documents.
GitHub	A web-based platform for managing code repositories, collaborating on code, and controlling versions, which we use to facilitate code access and modification from any location
Git	A version control system is used to track code changes.
Firebase	A cloud-based platform for building and managing mobile and web applications, which we use for handling authentication, real-time data storage, and backend services that support the core functionality of our application.
Google Meet	A video conferencing application for online communication, which we used to hold real-time meetings.

Microsoft Office Word	A Word processing software allows you to create and modify documents. We used it to prepare reports for our projects.
ChatGPT Integration	A language model used in Obour to generate personalized Arabic motivational and religious quotes based on the user's emotion and preferred message type, providing real-time emotional support

1.10 Contribution Breakdown

The development of the **Obour app** by **Reema**, **Alaa**, and **Bakirah**, following the **Waterfall Model** methodology. This approach allowed the team to progress through clearly defined phases—**Requirement Analysis**, **System Design**, **Implementation** and **Testing**, in a structured and sequential manner.

To facilitate teamwork and coordination, the group used **GitHub** as a version control and collaboration platform. This ensured that all members could work on the codebase simultaneously, review each other's work, and maintain organized branches for different features. Weekly meetings were held throughout the project to align on progress, divide tasks, resolve blockers, and make joint decisions regarding the system's design and functionality. Communication was maintained through group chats and shared documentation tools, fostering a highly cooperative working environment.

While all team members contributed to every phase of the project, each had specific areas where they took the lead, based on their skills and interests.

1.10.1 Individual Contributions

Reema

Reema played a key role in both **design** and **technical integration**, particularly in connecting the app to AI-based services. She contributed to the **UI/UX design**, helping structure user flows and screen layouts. In the **implementation phase**, Reema worked on several Flutter frontend components and was responsible for integrating the app with the **ChatGPT API**, enabling it to analyze users' emotional states and generate personalized motivational quotes. She also participated in testing and interface refinements, ensuring the emotional logging feature was engaging and functionally correct.

Alaa

Alaa took the lead during the **Requirement Analysis** and **System Design** phases. She was responsible for translating user needs into technical specifications,

defining data structures, and mapping out system behavior. Alaa also contributed heavily to **backend development** by designing and implementing the **Firestore database** and configuring **Firestore Authentication**. In the implementation phase, she developed key features such as the **User Profile** screen and the **Activity Management** system, ensuring seamless interaction between frontend UI and the backend services. She also participated in writing and reviewing code across different parts of the application.

Bakirah

Bakirah was deeply involved in the **Implementation** and **Testing** phases, with a strong contribution to coding and feature development. She developed the logic for **achievement badge generation** and implemented the **weekly activity plan generator**, two of the core dynamic features in the app. She also contributed to backend integration tasks and worked closely with Alaa and Reema to ensure consistent functionality. In the **Testing phase**, Bakirah conducted **unit testing** to validate code components individually and verify the stability and accuracy of the logic implemented. She also helped document bugs and suggest improvements for user experience.

1.10.2 Shared Contributions

- **GitHub Collaboration:**

The team used **GitHub** throughout the project for source code management. Branching strategies were followed to ensure that each feature was developed in isolation before being merged into the main codebase. Pull requests and code reviews were part of the regular workflow to maintain code quality and avoid conflicts.

- **Weekly Meetings:**

The group held **weekly meetings**, where they discussed progress, redistributed tasks, and addressed technical challenges. These meetings were key to ensuring continuous alignment and momentum across all phases of the Waterfall Model.

- **Cross-Functional Support:**

Although each member had key areas of leadership, all members supported one another by participating in every phase. For example, Reema helped with database modeling, Alaa contributed to design feedback, and Bakirah participated in implementation testing. This **cross-functional approach** ensured well-rounded development and a shared understanding of the entire system.

Chapter 2: Background and Similar Applications

2.1 Background

The rise of digital health solutions has greatly improved addiction recovery. Mobile apps now play a key role in supporting individuals with substance use disorders, offering tools like activity plans, peer support, and educational resources.

While many apps aim to provide personalized recovery options, there is a noticeable lack of solutions designed for specific cultural needs, especially for Arabic-speaking communities. This chapter explores existing apps, their features, and how they compare to *Obour*.

Unlike others, *Obour* is tailored to its audience by incorporating cultural sensitivity, Islamic practices, and values. Through this comparison, we emphasize the importance of creating inclusive tools that better support diverse communities on their recovery journey.

2.2 Similar Applications and Papers

2.2.1 First Paper: A Mobile App to Enhance Behavioral Activation Treatment for Substance Use Disorder (LETS ACT system)

The LETS ACT study explored the use of a smartphone app designed to enhance behavioral activation treatment for individuals with substance use disorder (SUD). Integrated into the Life Enhancement Treatment for Substance Use (LETS ACT), this app aimed to boost engagement with treatment skills outside of clinician-led sessions. The study assessed the app's features, feasibility, and acceptability, ultimately finding it useful but noting a decline in user engagement over time. Recommendations included incorporating reminders and alerts to encourage long-term use.

In comparison, the *Obour* app offers a range of features designed to support individuals in their recovery journey. Key features include Arabic language support, activity planning for daily activities, daily reminders to stay on track, and supportive affirmations for motivation. *Obour* also includes achievement badges to celebrate

progress, peer contact for support, educational information to raise awareness about substance use and emotional track. While LETS ACT also focuses on supporting users, it lacks several features found in *Obour*, such as Arabic language support and peer connection capabilities. LETS ACT allows for activity planning, provides reminders and emotional support but does not include supportive affirmations, contact with peer asking for help, or achievement badges. Instead, it emphasizes manual engagement with coping strategies rather than offering the educational and multimedia resources that *Obour* provides. Regarding availability in Saudi Arabia, both apps aim to serve users in that region; however, *Obour*'s features and design make it potentially more accessible and relevant for Arabic-speaking users. Overall, *Obour* presents a more comprehensive and engaging platform for individuals seeking assistance in their recovery, complementing the insights gained from the LETS ACT study [3].

2.2.2 Second Paper: HealthCall delivered via smartphone to reduce co-occurring drug and alcohol use in HIV-infected adults (HealthCall system)

HealthCall-S is a smartphone-based intervention designed to reduce co-occurring drug and alcohol use among HIV-infected individuals. It combines Motivational Interviewing (MI) with daily self-monitoring, personalized feedback, and positive reinforcement, using multimedia features to engage socioeconomically disadvantaged minority groups with minimal clinical staff time.

In comparison, *Obour* app and HealthCall-S share several similarities as they both aim to support individuals in behavioral change through structured daily engagement and personalized features. Both apps provide daily reminders to encourage consistent usage, offer supportive affirmations or motivational feedback to reinforce positive behaviors. Additionally, both platforms include elements for track emotional well-being (*Obour*: Emotional Track, HealthCall-S: Mood and Stress Monitoring) and emphasize user engagement through interactive features.

However, the two apps differ in their scope and additional features. HealthCall-S focuses specifically on reducing co-occurring drug and alcohol use among HIV patients, incorporating personalized feedback through graphs and video-based guidance as part of its core functionality. In contrast, *Obour* has a broader focus on providing an activity plan and achievement badges to celebrate progress. Furthermore, *Obour* includes educational information and facilitates contact with peers, which is absent in HealthCall-S. Lastly, *Obour* is region-specific with availability in Saudi Arabia and support Arabic language, whereas HealthCall-S is designed for U.S.-based populations that support English language [4].

2.2.3 WEconnect Application

WeConnect is an app that offers peer support and resources for addiction recovery, providing both free and paid services. Users can access unlimited virtual support group meetings daily. For \$40 monthly or \$350 annually, members can receive one-on-one peer support and wellness planning, with some insurance plans accepted. Key features include tools for building self-care routines, tracking sobriety, and monitoring meeting attendance, all facilitated via Zoom with guidance from peer support specialists. The app allows for customizable self-care tasks and incorporates a rewards-based system to encourage positive behavior and engagement.

In comparison to the *Obour* app, both platforms share several similarities, including customized activity plans, daily reminders for those plans, educational information, achievement badges to celebrate progress and contact with peer. However, there are notable differences: WeConnect does not support the Arabic language or Islamic practices, cannot be downloaded in Saudi Arabia, and lacks features such as supportive affirmations, emotional track. Overall, while both apps aim to enhance support for individuals in recovery, *Obour* addresses specific needs that WeConnect does not [5].

2.2.4 Sober Time Application

The Sober Time app is designed to support individuals in their recovery from addiction by enabling them to track their sobriety journey. Users can log their sober days, set personalized goals, and view a clear visual representation of their progress. The app features a built-in community forum where users can connect, share experiences, and provide support to one another. Additionally, Sober Time offers supportive affirmation quotes, reminders, and resources to encourage users in their recovery, all within a user-friendly interface aimed at fostering accountability and motivation. Overall, Sober Time serves as a valuable tool for those looking to maintain their sobriety and build a supportive network.

When comparing Sober Time to the *Obour* app, there are notable similarities and differences. Both apps can be downloaded in Saudi Arabia, enable users to contacts with peer asking for help, and provide achievement badges to celebrate progress. Both also offer supportive affirmation quotes for inspiration. However, significant differences exist: Sober Time does not support the Arabic language or specific Islamic practices, lacks a feature for emotional track, does not provide activity plans, and offers no educational information regarding substance use. While both apps aim to assist individuals in their recovery, *Obour* addresses specific cultural needs that Sober Time does not [6].

2.2.5 I Am Sober Application

"I Am Sober" is a sobriety-tracking app designed to help individuals recover from addiction to alcohol and other substances, as well as to break bad habits and harmful behaviors. The app allows users to track progress, receive daily notifications for support, and reinforce their reasons for quitting. Additionally, it helps them measure how much money they've saved since their last drink or use, providing a tangible incentive for maintaining sobriety. A standout feature is the ability to connect with a community of people who are going through similar experiences, fostering a sense of belonging and support.

In comparison to the *Obour* app, both platforms share similarities such as affirmation motivational quotes, activity plans, and achievement badges for progress. However, there are

significant differences: "I Am Sober" does not support the Arabic language or specific Islamic practices, lacks a feature for emotional tracking, and does not provide educational information about substance use. Furthermore, it cannot be downloaded in Saudi Arabia and does not have the capability to connect with peer asking for help. While both apps aim to assist individuals in their recovery, *Obour* addresses certain cultural and practical needs that "I Am Sober" does not fulfill [7].

2.3 Literature Review:

Table 2: Literature Review

Topic No.	Topic Name	Relevance	Benefits	Limitations
1	LETS ACT (Paper)	<ul style="list-style-type: none"> -Activity planning. -Activity Reminders -Emotional tracking 	<ul style="list-style-type: none"> -Enhances behavioral activation, -Supports activity planning, -Tracks emotions -Provides reminders 	<ul style="list-style-type: none"> -No user engagement features -Lacks motivational tools like affirmations and badges -Falls short in cultural relevance and localized content.
2	HealthCall (Paper)	<ul style="list-style-type: none"> -Self-monitoring tools -Progress tracking -Emotional well-being tracking -Motivational feedback. 	<ul style="list-style-type: none"> -Provides personalized feedback using graphs and multimedia features. 	<ul style="list-style-type: none"> -Limited scope, focusing specifically on HIV-infected individuals with substance use issues.
3	WEconnect (App)	<ul style="list-style-type: none"> -Personalized activity plans -Daily reminders -Educational content -Peer support 	<ul style="list-style-type: none"> -Customizable self-care tasks, tailoring recovery plans to the user needs. 	<ul style="list-style-type: none"> -Requires payment for premium features like receiving one-on-one peer support and wellness planning -Not accessible in Saudi Arabia

4	Sober Time (App)	-Progress tracking -Achievement badges -Supportive affirmations -Accessibility in Saudi	-Offers supportive affirmations and reminders -Features a user-friendly interface	-Lacks Arabic language support and integration of Islamic practices
5	I Am Sober (App)	-Motivational affirmation quotes -Achievement badges	-Provides motivational affirmation quotes	-Not accessible in Saudi Arabia

2.4 Comparative Study

Table 3: Comparative Study

Papers/Apps	LETS ACT	HealthCall	 WEconnect	 Sober Time	 I Am Sober	OBOUR
Feature						
Arabic						✓
Activity Plan	✓		✓		✓	✓
Daily Reminders	✓	✓	✓			✓
Supportive Affirmations		✓		✓	✓	✓
Emotional State	✓	✓				✓
Achievement Badge			✓	✓	✓	✓
Contact with Peer			✓	✓		✓
Educational Info.			✓			✓

Availability in SA				✓		✓
--------------------	--	--	--	---	--	---

Chapter 3: Analysis

3.1 Background

In this chapter, we will explore the key components of the system's design and requirements. We will begin by discussing both functional and non-functional requirements, outlining what the system must accomplish and the constraints it operates under. Following this, we will present a use case diagram to visually represent the interactions between users and the system.

Additionally, we will provide detailed use case descriptions that clarify the various scenarios in which users will engage with the system. Finally, we will map these requirements to sequence diagrams, illustrating the flow of events and interactions over time. This comprehensive approach will ensure a clear understanding of how the system is intended to function and how it will meet the needs of its users.

3.2 Functional Requirements

Functional requirements outline the functions, features, and capabilities that a system must provide to meet patient needs. The functional requirements of the *Obour* system are specified below:

1. Account Registration

- The post-rehabilitation patient shall be able to register his info to *Obour* application.
- The post-rehabilitation patient shall be able to login with their username and password to the *Obour* application.
- The post-rehabilitation patient shall be able to add their favorite activity.

2. Managing Activity Plan

- The system shall generate a weekly activity plan.
- The post-rehabilitation patient shall be able to Edit/Delete/Add/Mark as done to their activity plan.
- The system shall send notification reminders about activities on a regular basis.

3. Achievement Badges

- The system shall generate achievement badges based on the patient's progress in marked activities.
- The post-rehabilitation patient shall be able to view their achievement badges.

4. Emotional state

- The post-rehabilitation patient shall be to log their emotional state throughout the day.
- The system shall use ChatGPT to generate messages based on the patient's emotional state and their preferred types of supportive religious messages—quotes, affirmations, advice, or motivation—.
- The system shall display a supportive message to the patient.

5. Peer Contact

- The post-rehabilitation patient shall be able to add/Edit their peer's contact number to request help via SMS.
- The system shall generate SMS text messages for the patient's peers, requesting help when needed.

6. Educational information

- The system shall display educational information, inspirational stories, YouTube videos, and podcasts.
- The post-rehabilitation patient shall be able to view educational information.

3.3 Non-Functional Requirements

1. Usability:

- The system's user interface shall be intuitive and easy to navigate for patients seeking support in their journey.
- The processes for creating an account and logging in shall be straightforward and user-friendly.

2. Reliability:

- The notification feature of the system shall be reliable, ensuring patients receive timely reminders.

- The system shall be capable of providing accurate updates about patient' activities and progress.
- The achievement badges system shall accurately reflect patient' progress to motivate continued engagement.

3. Scalability

- The system shall be scalable to accommodate a growing number of users without significant performance degradation.

4. Security

- The system should comply with relevant data protection regulations to safeguard patient information.

5. Data privacy

- The system shall collect only the minimum amount of personal data necessary to provide its services, reducing the risk of data exposure.

3.4 Initial Design

3.4.1 Use Case Diagram

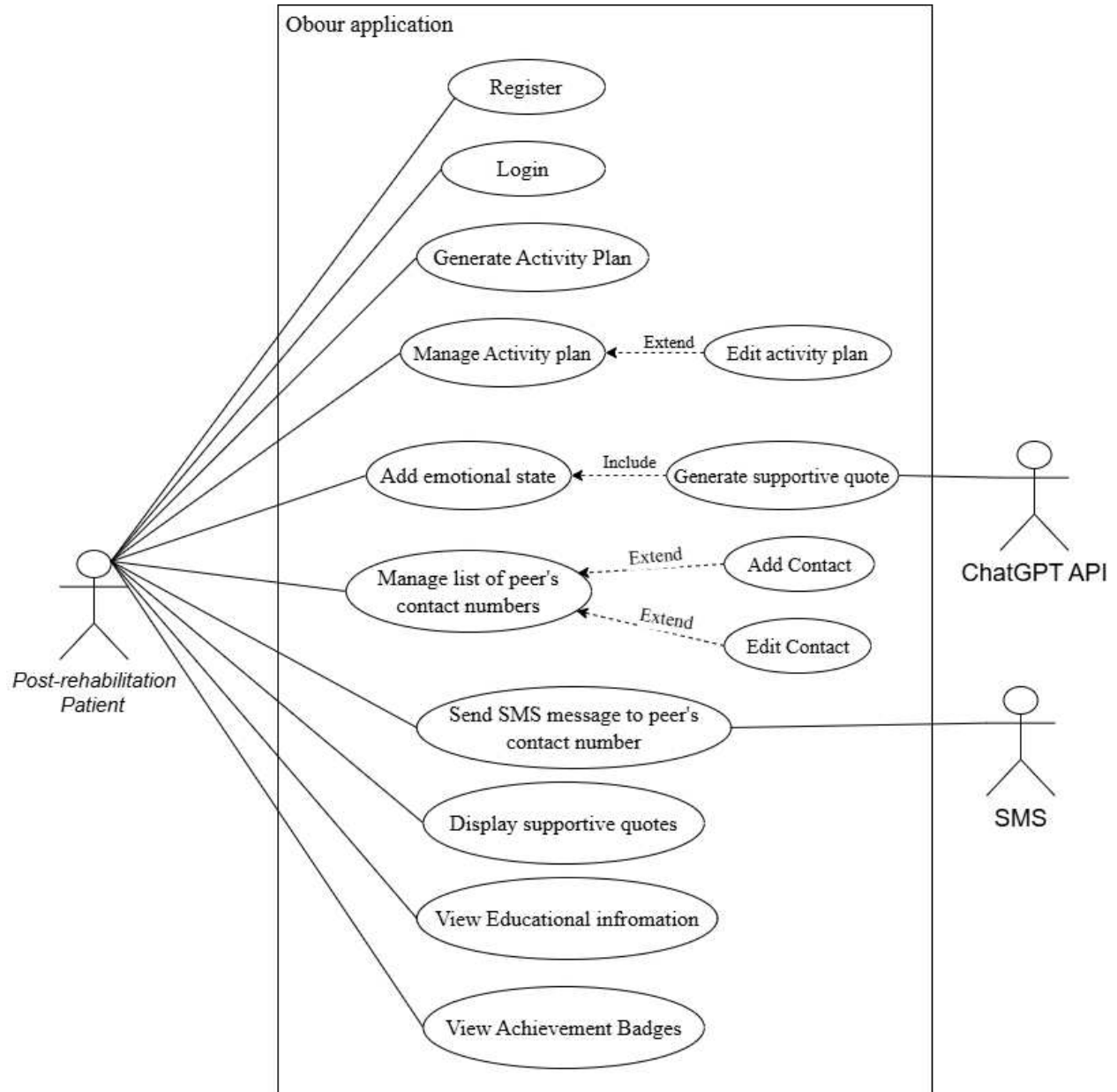


Figure 4: Use Case Diagram

3.4.2 Use Case Description

- **Register**

Table 4: Use Case Description (Register)

Use Case Name	Register
Primary Actor	Post-rehabilitation patient
Level	Kite (Summary)
Stakeholders	Post-rehabilitation patient
Trigger	Post-rehabilitation patient initiates the registration process by selecting the "Register" option on the app's home screen.
Precondition(s)	The post-rehabilitation patient must not already have an existing account with the app.
Success End Condition(s)	The post-rehabilitation patient is successfully registered and logged into the app, with an activity plan generated based on their interests and hobbies.
Normal Flow	<ol style="list-style-type: none">1. Post-rehabilitation patient opens the app and selects the "Register" option.1. Post-rehabilitation patient is prompted to enter their personal information, including:<ul style="list-style-type: none">– Name– Email address– Password– Date of birth– Gender2. Post-rehabilitation patient is prompted to enter their favorite activities (e.g., sports, art, music, reading, etc.).3. The system validates the information:4. Upon successful validation, the system creates a new user account.5. The system generates a customized activity plan based on the patient's favorite activities

Extensions	4a. Input Validation Errors: <ul style="list-style-type: none"> 4a1. If any fields are incomplete or invalid, the system highlights the errors and prompts the patient to correct them.
-------------------	---

- Login**

Table 5: Use Case Description (Login)

Use Case Name	Login
Primary Actor	Post-rehabilitation patient
Level	Kite (Summary)
Stakeholders	Post-rehabilitation patient
Trigger	Post-rehabilitation patient initiates the login process by selecting the "Login" option on the app's home screen.
Precondition(s)	The post-rehabilitation patient must have an existing account with valid credentials (username/email and password).
Success End Condition(s)	The post-rehabilitation patient is successfully logged into the app and directed to the main interface.
Normal Flow	<ol style="list-style-type: none"> 1. Post-rehabilitation patient opens the app and selects the "Login" option. 2. Post-rehabilitation patient enters their username/email and password. 3. Post-rehabilitation patient selects the "Submit" button. 5. The system validates the credentials against the database. 6. If the credentials are valid, the system logs the patient in and displays the dashboard. 7. The post-rehabilitation patient can now access their account features.
Extensions	4a. Invalid Credentials:

	<ul style="list-style-type: none"> 4a1. If the post-rehabilitation patient enters incorrect credentials, the system displays an error message prompting the patient to retry.
--	--

- **Generate Activity Plan**

Table 6: Use Case Description (Generate Activity Plan)

Use Case Name	Generate Activity Plan
Primary Actor	Post-rehabilitation patient
Level	Kite (Summary)
Stakeholders	Post-rehabilitation patient
Trigger	The post-rehabilitation patient enters their favorite activity.
Precondition(s)	The post-rehabilitation patient must enter their favorite activities so that the system can generate an activity plan.
Success End Condition(s)	The activity plan reflects the patient's personalized activities based on their selected favorite activities.
Normal Flow	<ol style="list-style-type: none"> 1. The post-rehabilitation patient chooses their favorite activities. 2. The system generates an activity plan based on the input provided.
Extensions	1a. If the post-rehabilitation patient does not enter any favorite activities: <ul style="list-style-type: none"> 1a1. The system displays a message: "There are no activities chosen." 1a2. The system suggests entering a new activity.

- **Manage Activity Plan**

Table 7: Use Case Description (Manage Activity Plan)

Use Case Name	Manage Activity Plan
Primary Actor	Post-rehabilitation patient
Level	Kite (Summary)
Stakeholders	Post-rehabilitation patient

Trigger	The post-rehabilitation patient decides to review or modify their activity plan.
Precondition(s)	The post-rehabilitation patient must be registered and logged into their <i>Obour</i> account, with their favorite activity chosen during generate activity plan.
Success End Condition(s)	The activity plan reflects the patient's personalized activities based on their chosen favorite activities, with the ability to add, edit, delete or marked the activities as done.
Normal Flow	<ol style="list-style-type: none"> 3. The post-rehabilitation patient registers and provides their favorite activity. 4. The system generates an activity plan based on the input provided. 5. The post-rehabilitation patient accesses the "Manage Activity Plan" feature to review the generated activities. 6. The post-rehabilitation patient can add new activity to the activity plan from mange activity plan. 7. The post-rehabilitation patient selects button edit on activity plan. 8. editing on plan, the post-rehabilitation patient updates the activity plan details (time, priority,activity) and save the changes. 9. If deleting, the post-rehabilitation patient uncheck the box of the activity. 10. The system updates the activity plan. 11. The system confirms the update with a success message
Extensions	<p>1a. If the post-rehabilitation patient cancels the editing process:</p> <ul style="list-style-type: none"> • 1a1. No changes are saved, and the patient is returned to the activity plan view. <p>2a. The successful confirming message does not appear.</p>

	<ul style="list-style-type: none"> • 2a1. The post-rehabilitation patient re-opens the app to check. • 2a2. The post-rehabilitation patient repeats the steps to edit or delete the activity
--	--

- **Add Emotional State**

Table 8: Use Case Description (Add Emotional State)

Use Case Name	Add Emotional State
Primary Actor	post-rehabilitation patient
Level	Kite (Summary)
Stakeholders	post-rehabilitation patient
Trigger	The post-rehabilitation patient wants to add their current emotional state
Precondition(s)	The post-rehabilitation patient must be logged into their <i>Obour</i> account.
Success End Condition(s)	The system saves the new emotional state entry
Normal Flow	<ol style="list-style-type: none"> 1. The post-rehabilitation patient enters their emotions state through a mood scale with emojis. 2. The post-rehabilitation patient also can change their emotion state through the day.
Extensions	1a. The post-rehabilitation patient does not enter the emotional state. <ul style="list-style-type: none"> • 1a1. No action is taken; the system allows the user to proceed without input.

- **Generate Supportive Quote**

Table 9: Use Case Description (Generate Supportive Quote)

Use Case Name	Generate Supportive Quote
Primary Actor	ChatGPT

Level	Kite (Summary)
Stakeholders	<ul style="list-style-type: none"> • Post-rehabilitation patient: Receives emotionally relevant and personalized motivational quotes (e.g., religious, affirmations, advice). • ChatGPTService: Generates quotes tailored to the patient's current mood and preferences.
Trigger	The patient selects an emotional state (emoji) on the Home screen, prompting the system to generate a quote.
Precondition(s)	<input type="checkbox"/> The patient is logged in. <input type="checkbox"/> The patient has selected an emotional state. <input type="checkbox"/> The ChatGPTService has access to the selected mood and stored quote type preferences (if any).
Success End Condition(s)	A supportive, culturally tailored motivational quote is successfully generated and displayed based on the selected emotional state and the patient's preferred quote types.
Normal Flow	<ol style="list-style-type: none"> 1. The patient selects a mood using the emoji-based mood selector on the Home screen. 2. The system saves the selected mood (<code>_selectedMood</code>) and shows a loading spinner. 3. The app retrieves the current user's ID using Firebase Authentication. 4. The system queries the patient's Quotes subcollection in Firestore to fetch preferred quote types (e.g., "نصائح", "تشجيع"). <ul style="list-style-type: none"> • If no preferences are found, it defaults to "نصائح". 5. The <code>ChatGPTService.generateQuote()</code> method is called with the selected emotion and preferred quote types. 6. ChatGPTService sends a POST request to the external GPT-4o model with a culturally appropriate system prompt and mood-based user input.

	<p>7. A personalized Arabic motivational quote is returned.</p> <p>8. The app decodes the response and displays the quote in a styled AlertDialog.</p>
Extensions	<p>1a. No preferred quote types are found in Firestore:</p> <ul style="list-style-type: none"> • 1a1. The system defaults to general type "نصائح". <p>2a. No emotional state is selected:</p> <ul style="list-style-type: none"> • 2a1. The quote generation feature is not triggered. <p>3a. Error occurs during API call or data retrieval:</p> <ul style="list-style-type: none"> • 3a1. The system logs the error and optionally shows a fallback message. • 3a2. The app dismisses the spinner without displaying a quote.

- **Display Supportive Quote**

Table 10: Use Case Description (Display Supportive Quote)

Use Case Name	Receive Supportive Quote After Emotion Selection
Primary Actor	Post-rehabilitation patient
Level	Kite (Summary)
Stakeholders	Post-rehabilitation patient
Trigger	The post-rehabilitation patient voluntarily selects their emotional state.
Precondition(s)	<input type="checkbox"/> The patient is logged in. <input type="checkbox"/> The patient manually selects a mood via the emoji scale.
Success End Condition(s)	The system generates and displays a supportive quote aligned with the selected emotional state and the patient's preferred quote type.
Normal Flow	<p>1. The patient opens the app and optionally selects an emotional state via the emoji-based mood selector.</p>

	<ol style="list-style-type: none"> The system retrieves the user's quote type preferences from Firestore (if available). A personalized quote is generated using ChatGPTService and displayed in a styled dialog.
Extensions	<p>1a. The user does not select an emotional state:</p> <ul style="list-style-type: none"> 1a1. No quote is generated; the app continues functioning without prompts or reminders.

- **Manage List of Peer's Contact Numbers**

Table 11: Use Case Description (Manage List of peer's contact numbers)

Use Case Name	Manage List of Peer's Contact Numbers
Primary Actor	Post-rehabilitation patient
Level	Kite (Summary)
Stakeholders	Post-rehabilitation patient
Trigger	Post-rehabilitation patient selects the option to view peer contacts from the app's main menu.
Precondition(s)	<p>The post-rehabilitation patient must be logged into the app.</p> <p>The post-rehabilitation patient must have an existing list of peer contacts or the ability to create one.</p>
Success End Condition(s)	The post-rehabilitation patient can view the list of peer contact numbers, add new contacts, and edit or delete existing contacts successfully.
Normal Flow	<ol style="list-style-type: none"> The post-rehabilitation patient selects the "Peer Contacts" option from the main menu. The system displays a list of existing peer contacts, showing names and phone numbers. The post-rehabilitation patient selects the "Add Contact" button. The system prompts the patient to enter the new contact's name and phone number.

	<ol style="list-style-type: none"> 5. The post-rehabilitation patient submits the information. 6. The system validates the input (e.g., checks for valid phone number format). 7. Upon successful validation, the new contact is added to the list. 8. The system confirms the addition with a success message. 9. The post-rehabilitation patient selects a contact from the list to edit or delete. 10. If editing, the post-rehabilitation patient updates the contact's details (name and phone number) and saves the changes. 11. If deleting, the post-rehabilitation patient confirms the deletion. 12. Upon successful validation, the contact details are updated in the list. 13. The system confirms the update with a success message.
Extensions	<p>6a. Input Validation Errors:</p> <ul style="list-style-type: none"> • 6a1. If the post-rehabilitation patient enters invalid data (e.g., non-numeric characters in the phone number), the system displays an error message and prompts the user to correct it.

- **Send SMS to Peer's Contact Number**

Table 12: Use Case Description (Send SMS to peer's contact number)

Use Case Name	Send SMS Message to Peer
Primary Actor	Post-rehabilitation patient
Level	Kite (Summary)
Stakeholders	Post-rehabilitation patient, peer

Trigger	The post-rehabilitation patient selects the "Connect with Peer" option and chooses to send an SMS to a peer's contact number
Precondition(s)	<ol style="list-style-type: none"> 1. The post-rehabilitation patient must be registered and logged into their Obour account. 2. At least one peer's contact number must be saved in the system. 3. The patient's device must have SMS functionality enabled.
Success End Condition(s)	The SMS message is generated in the device's default SMS messaging app, ready for the addict to review and send.
Normal Flow	<ol style="list-style-type: none"> 1. The post-rehabilitation patient logs into the Obour application. 2. The post-rehabilitation patient navigates to the "Connect with Peer" page from the main menu. 3. The system displays a list of saved peer contacts. 4. The post-rehabilitation patient selects a peer from the list. 5. The system generates a pre-written SMS message requesting help (e.g., <i>"Hi [Peer's Name], I need your support right now. Please contact me."</i>). 6. The system opens the device's default SMS messaging app and populates the selected peer's phone number and the generated message in the input box. 7. The post-rehabilitation patient reviews the message and decides whether to send it.
Extensions	<p>3a. No peer contacts saved:</p> <ul style="list-style-type: none"> • 3a1. The system displays a message: <i>"No peer contacts found. Please add a peer to your contact list."</i> • 3a2. The post-rehabilitation patient selects the option to add a new peer contact. <p>6a. The post-rehabilitation patient cancels the message:</p>

	<ul style="list-style-type: none"> 6a1. The post-rehabilitation patient closes the SMS messaging app without sending the message. 6a2. No further action is taken, and the post-rehabilitation patient returns to the Obour application. <p>6b. SMS app fails to open:</p> <ul style="list-style-type: none"> 6b1. The system displays an error message: <i>"Unable to open the SMS app. Please try again later."</i> 6b2. The post-rehabilitation patient is returned to the peer contact list.
--	---

- View Achievement Badges**

Table 13: Use Case Description (View Achievement Badges)

Use Case Name	View Achievement Badges
Primary Actor	Post-rehabilitation patient
Level	Kite (Summary)
Stakeholders	Post-rehabilitation patient
Trigger	The post-rehabilitation patient selects the "Achievements" section in the Obour app to view their earned badges.
Precondition(s)	The post-rehabilitation patient must be registered and logged into their <i>Obour</i> account.
Success End Condition(s)	The post-rehabilitation patient successfully views their earned achievement badges and associated progress details.
Normal Flow	<ol style="list-style-type: none"> 1. The post-rehabilitation patient logs into the Obour application. 2. The post-rehabilitation patient navigates to the "Settings" page and selects "View Achievement Badges." 3. The system retrieves the patient's activity progress and corresponding achievement badges. 4. The system displays a list of earned badges.
Extensions	4a. No badges earned yet:

	<ul style="list-style-type: none"> 4a1. The system displays a message: <i>"No achievement badges earned yet. Keep going—you're doing great!"</i> <p>2a. User is not logged in:</p> <ul style="list-style-type: none"> 2a1. The system redirects the user to the login page. 2a2. The post-rehabilitation patient logs in with their username and password, then repeats Step 2.
--	---

- View Educational Information**

Table 14: Use Case Description (View Educational Information)

Use Case Name	View Educational Information
Primary Actor	Post-rehabilitation patient
Level	Kite (Summary)
Stakeholders	Post-rehabilitation patient
Trigger	The post-rehabilitation patient selects the "Media" page to explore educational content.
Precondition(s)	<ol style="list-style-type: none"> 1. The post-rehabilitation patient must be registered and logged into their Obour account. 2. The system must have access to the database of educational materials.
Success End Condition(s)	The post-rehabilitation patient successfully views educational information, including articles, videos, or podcasts.
Normal Flow	<ol style="list-style-type: none"> 1. The post-rehabilitation patient logs into the Obour application. 2. The post-rehabilitation patient navigates to the "Media" page. 3. The system retrieves and displays a categorized list of educational materials (e.g., articles, inspirational stories, videos, and podcasts). 4. The post-rehabilitation patient selects a specific item to view or play. 5. The system loads the selected content for the patient to read, watch, or listen to.

Extensions	<p>3a. No educational content available:</p> <ul style="list-style-type: none"> 3a1. The system displays a message: <i>"No educational materials are currently available. Please check back later."</i> <p>2a. User is not logged in:</p> <ul style="list-style-type: none"> 2a1. The system redirects the post-rehabilitation patient to the login page. 2a2. The post-rehabilitation patient logs in with their username and password, then repeats Step 2. <p>5a. Content fails to load:</p> <ul style="list-style-type: none"> 5a1. The system displays an error message: <i>"Unable to load this content. Please try again later."</i> 5a2. The post-rehabilitation patient returns to the list and selects another item.
-------------------	--

3.4.3 Mapping Requirements

The mapping between each use case and the functional requirements is represented in Table 15.

Table 15: Mapping Requirements

Use Case \ Requirement	1	2	3	4	5	6	7	8	9	10	11
Account Registration	X	X									
Managing Activity Plan			X	X							
Achievement Badges											X
Emotional state					X	X			X		
Peer Contact							X	X			

Educational information.											X	
---------------------------------	--	--	--	--	--	--	--	--	--	--	---	--

3.5 Sequence Diagram

3.5.1 High level Sequence Diagram

This diagram shows how post-rehabilitation patients can add and manage their activities, log their emotional state to receive motivational quotes, send SMS messages to peers for support, earn achievement badges based on progress, and access educational media for guidance and information

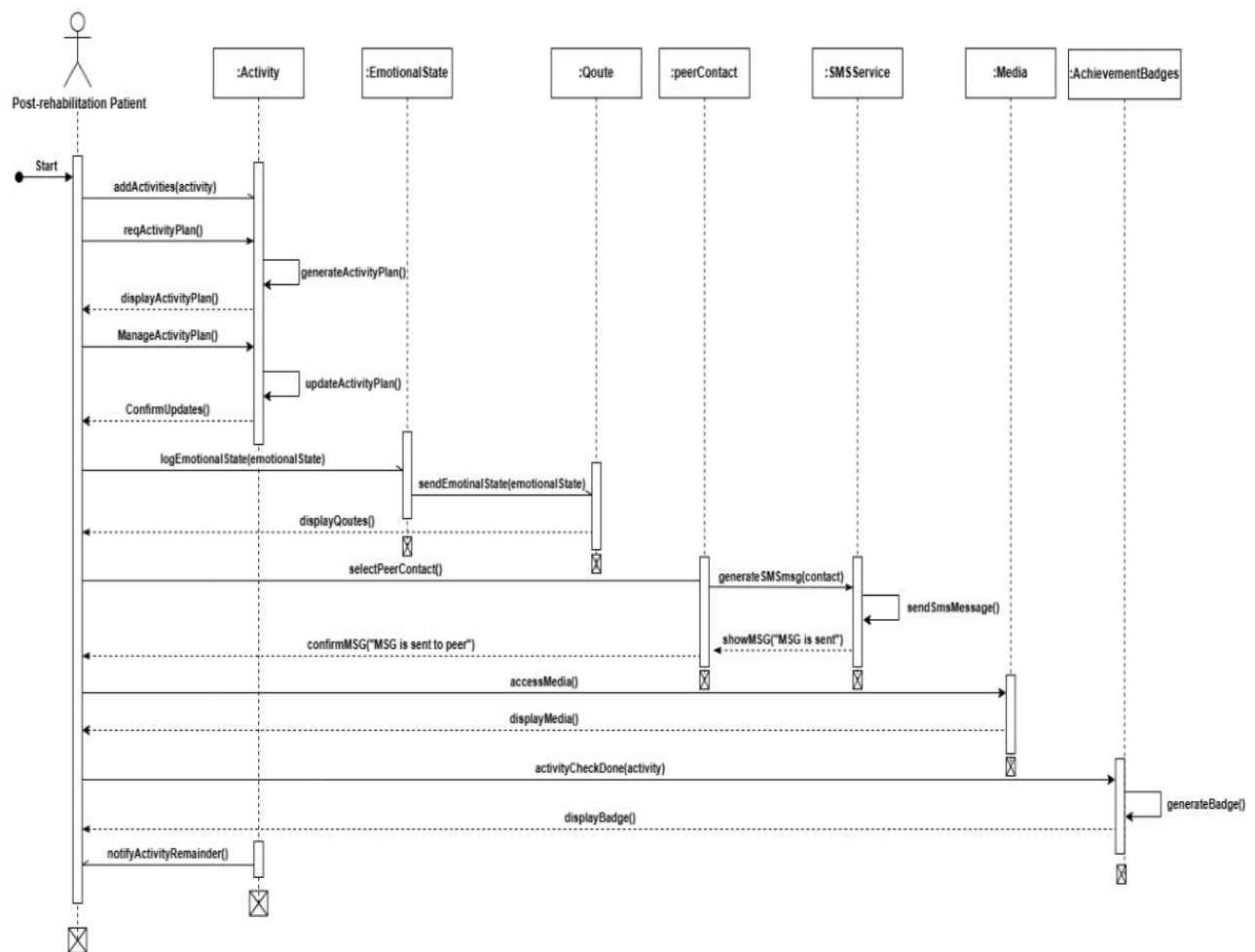


Figure 5: Sequence Diagram (High Level-up)

3.5.2 Generate Activity Plan

This diagram shows how the system generates activity plans for post-rehabilitation patients.

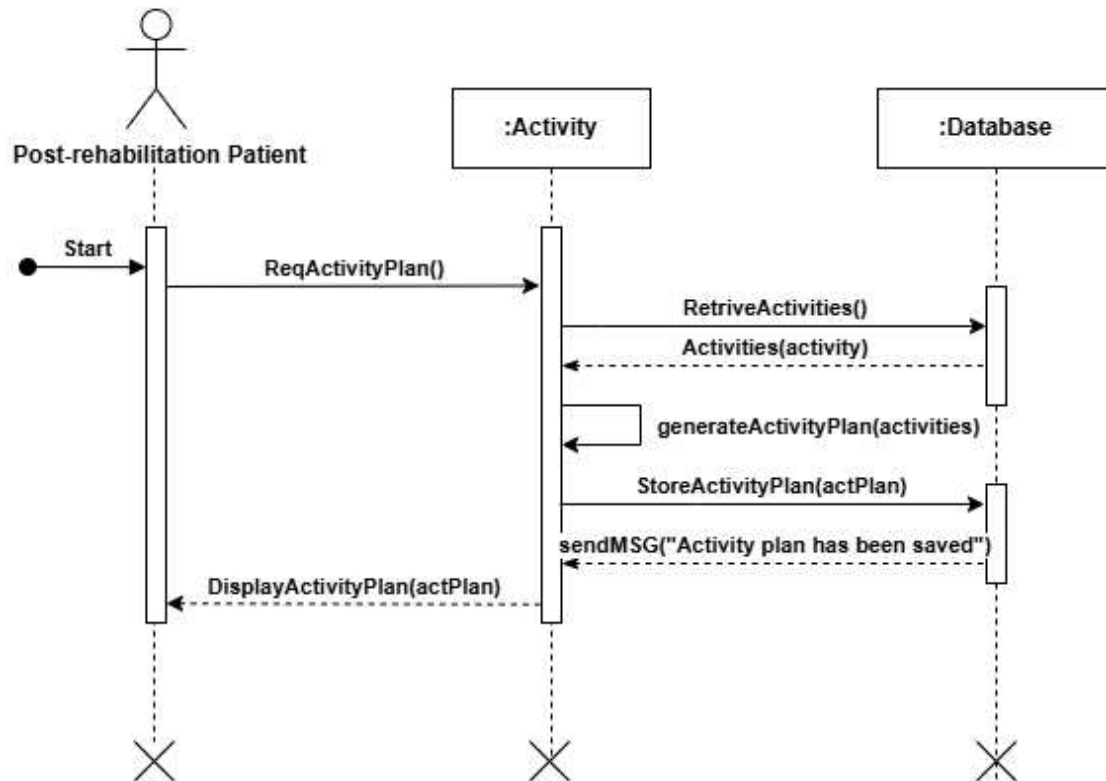


Figure 6: Sequence Diagram (Generate Activity Plan)

3.5.3 Manage Activity Plan

This diagram shows how post-rehabilitation patients manage their activity plan as they can add/delete/edit, and mark activities as done.

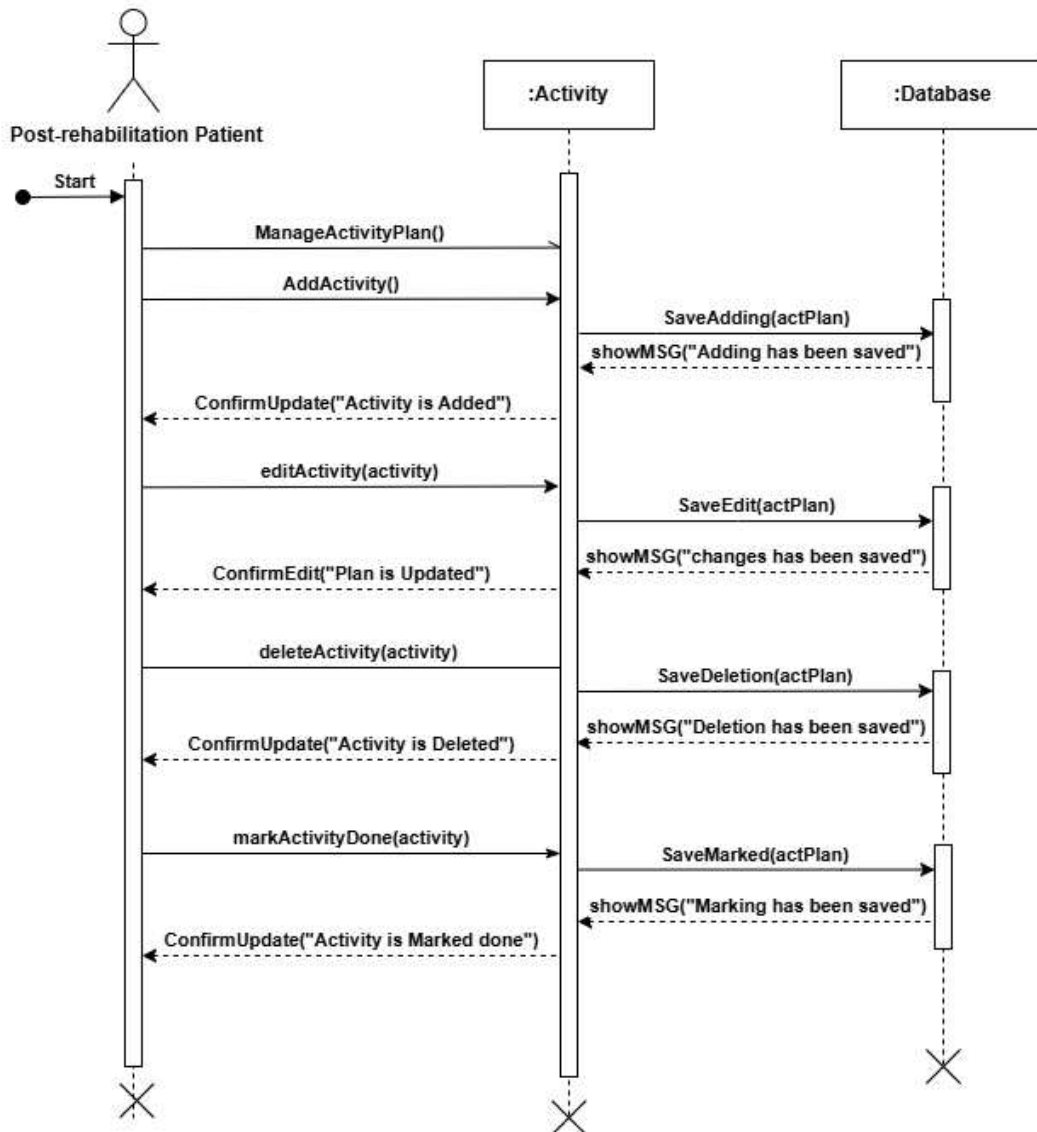


Figure 7: Sequence Diagram (Manage Activity Plan)

3.5.4 Notify Activity

This diagram shows how post-rehabilitation patients receive continuous notifications throughout the day about their scheduled activities, helping them stay on track and engaged in their recovery process.

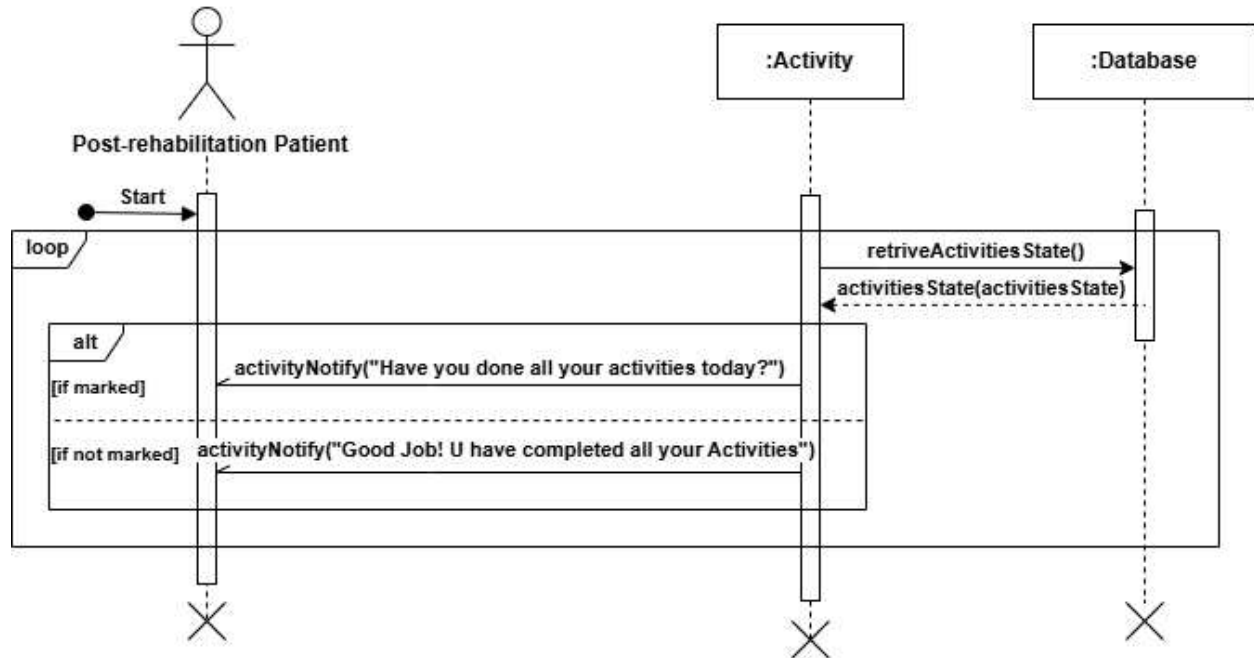


Figure 8: Sequence Diagram (Notify Activity)

3.5.5 Add Emotional State

This diagram shows how post-rehabilitation patients add their emotional state.

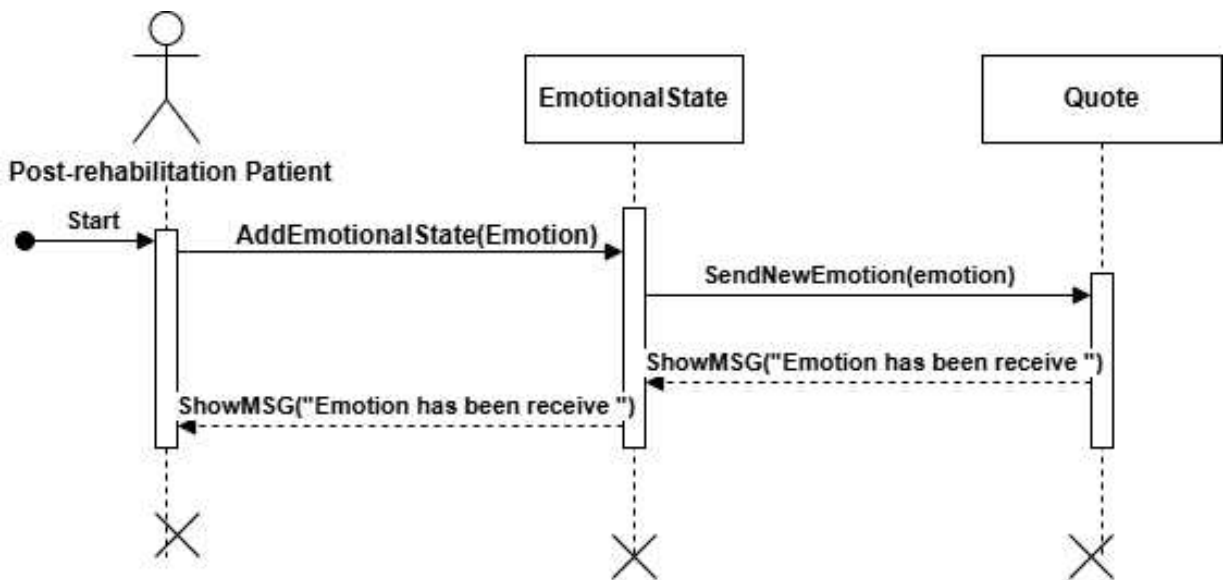


Figure 9: Sequence Diagram (Add Emotional State)

3.5.6 Generate Supportive Quote

This diagram shows how the chatGPT generates supportive quotes based on the emotional state logged by post-rehabilitation patients, offering personalized encouragement and motivation.

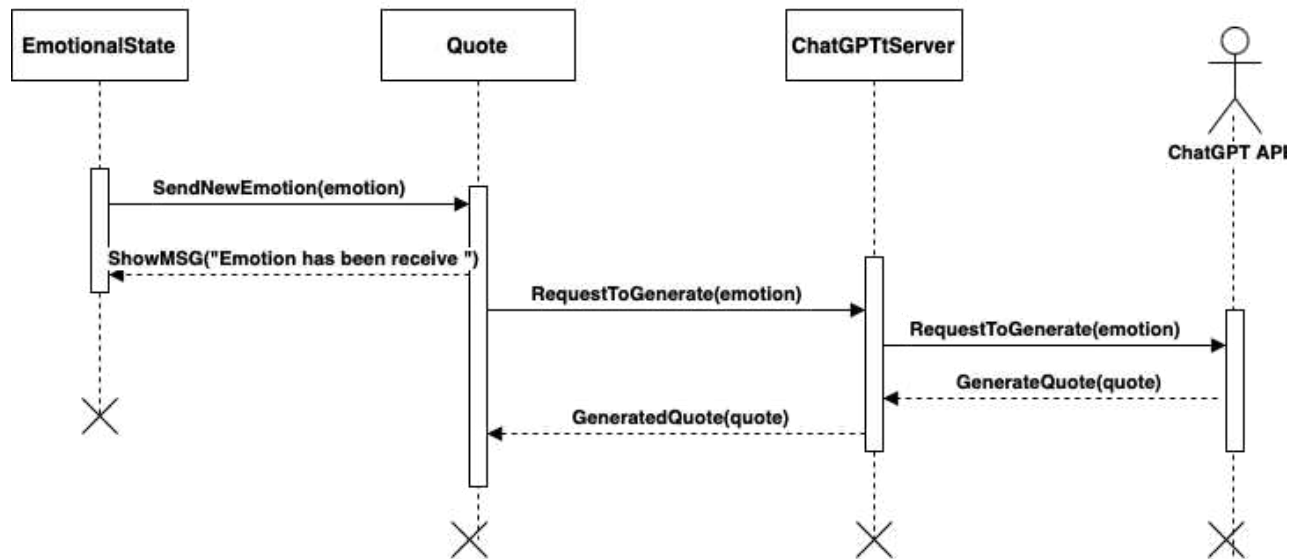


Figure 10: Sequence Diagram (Generate Supportive Quote)

3.5.7 Display Supportive Quote

This diagram shows how the system continuously displays supportive quotes throughout the day to post-rehabilitation patients, providing ongoing encouragement based on their emotional state.

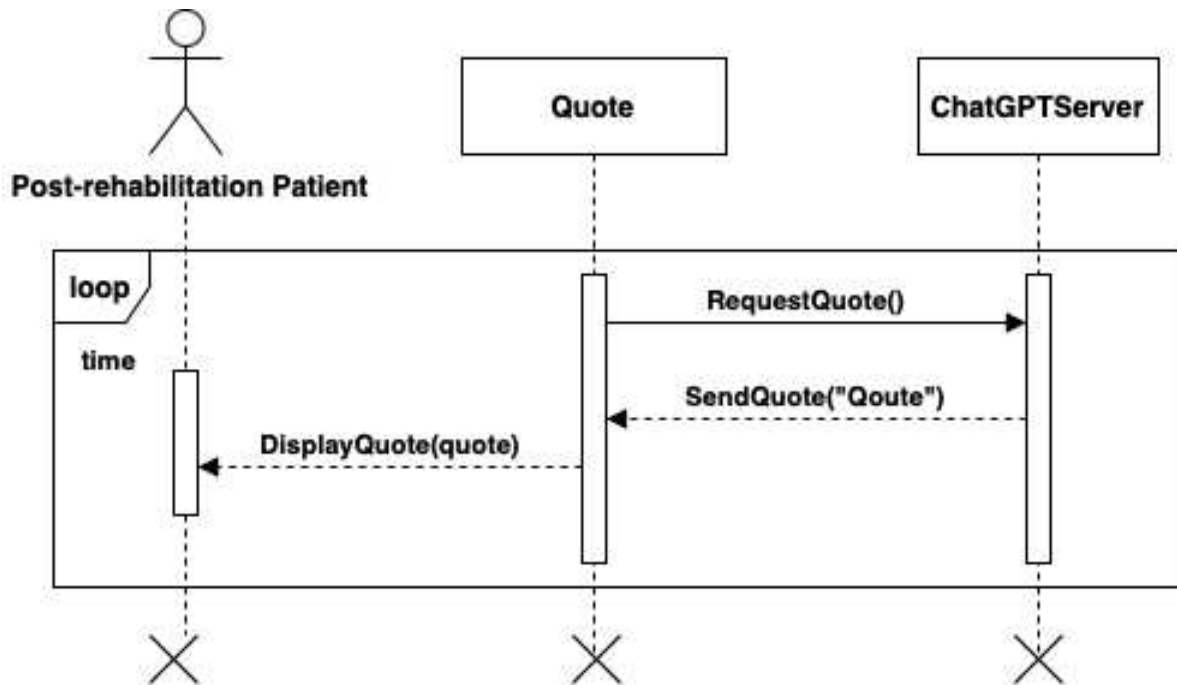


Figure 11: Sequence Diagram (Display Supportive Quote)

3.5.8 Manage List of Peer's Contact Number

This diagram shows how post-rehabilitation patients can manage list of peer's contact numbers as they can add/delete/edit contact.

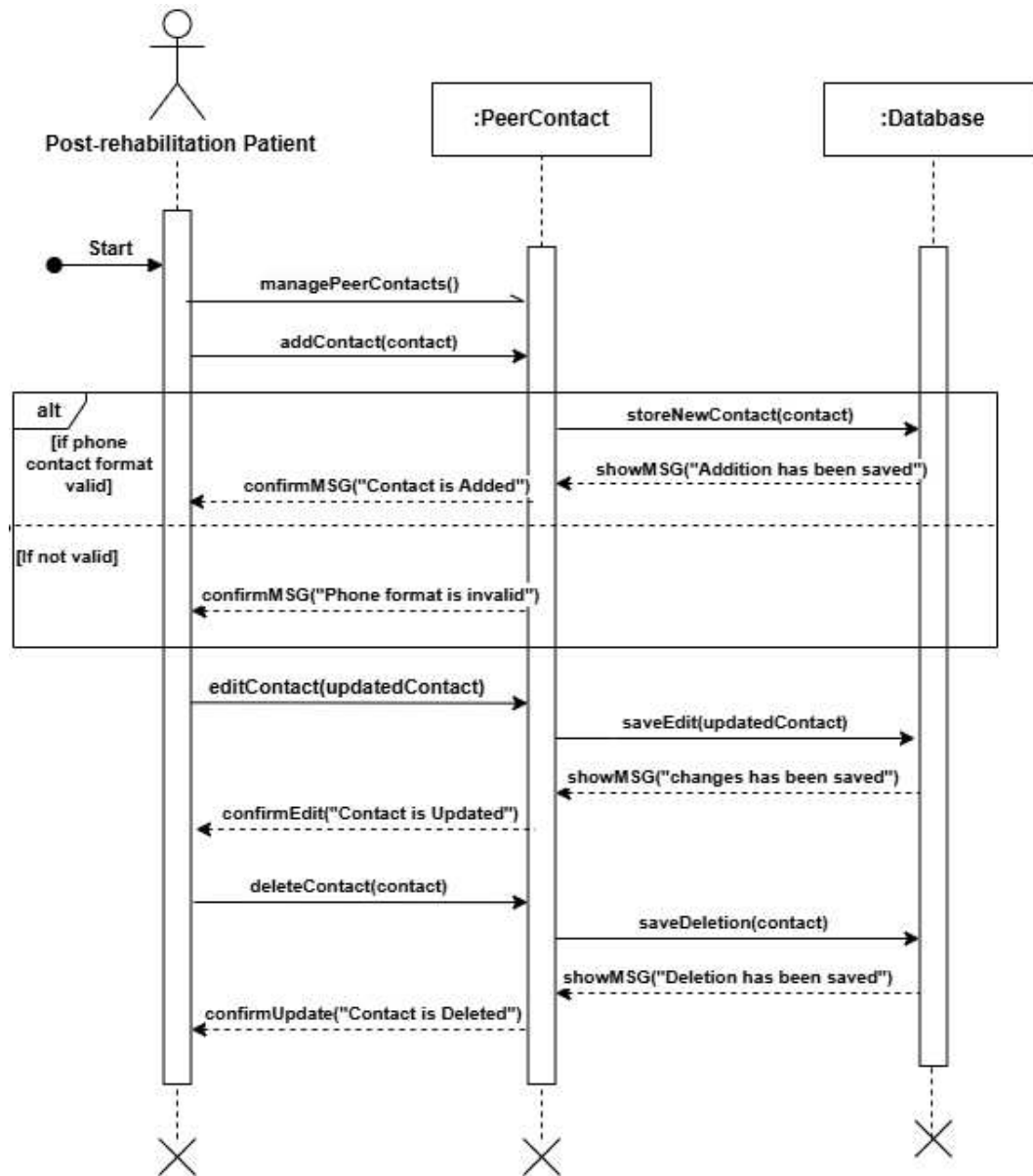


Figure 12: Sequence Diagram (Manage List Of Peer's Contact Number)

3.5.9 Send SMS to Peer's Contact Number

This diagram shows how post-rehabilitation patients can select a peer to contact, with the system automatically opening the SMS interface to facilitate communication.

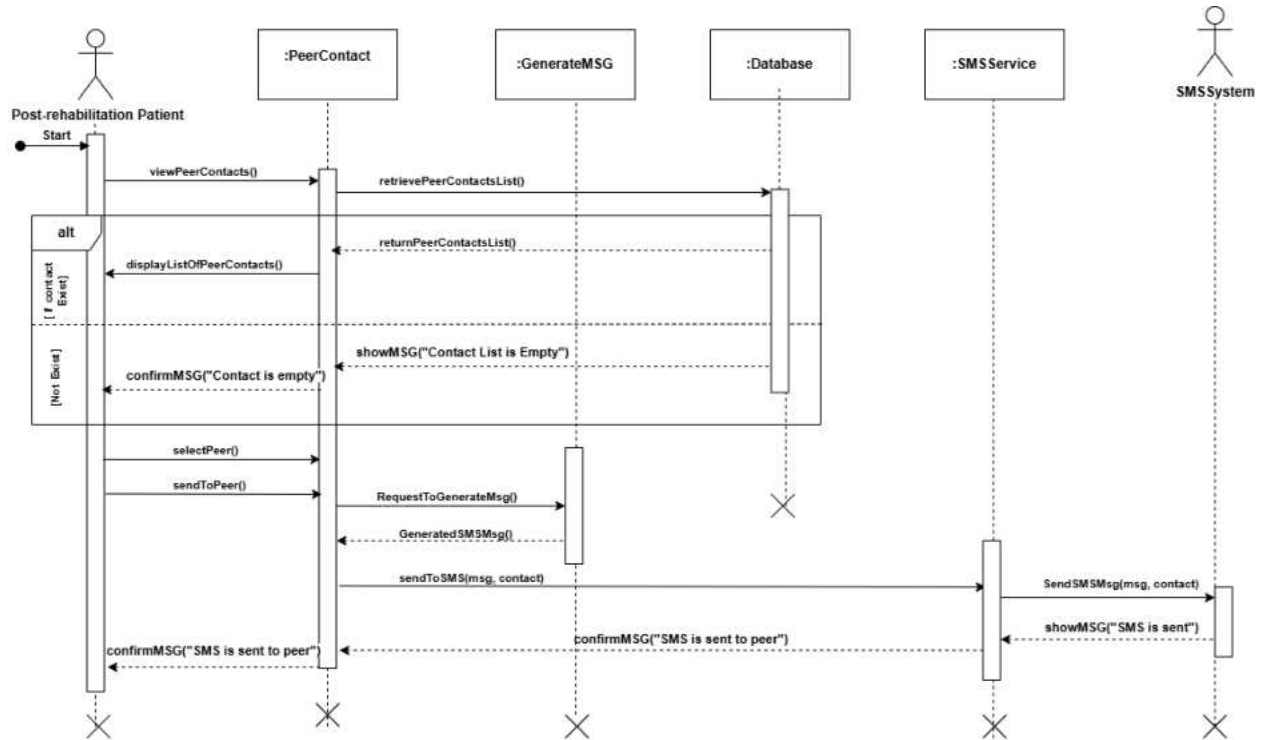


Figure 13: Sequence Diagram (Send SMS to Peer's Contact Number)

3.5.10 View Educational Information

This diagram shows how post-rehabilitation patients can access educational media, including text, videos, and podcasts, to support their recovery and enhance their knowledge.

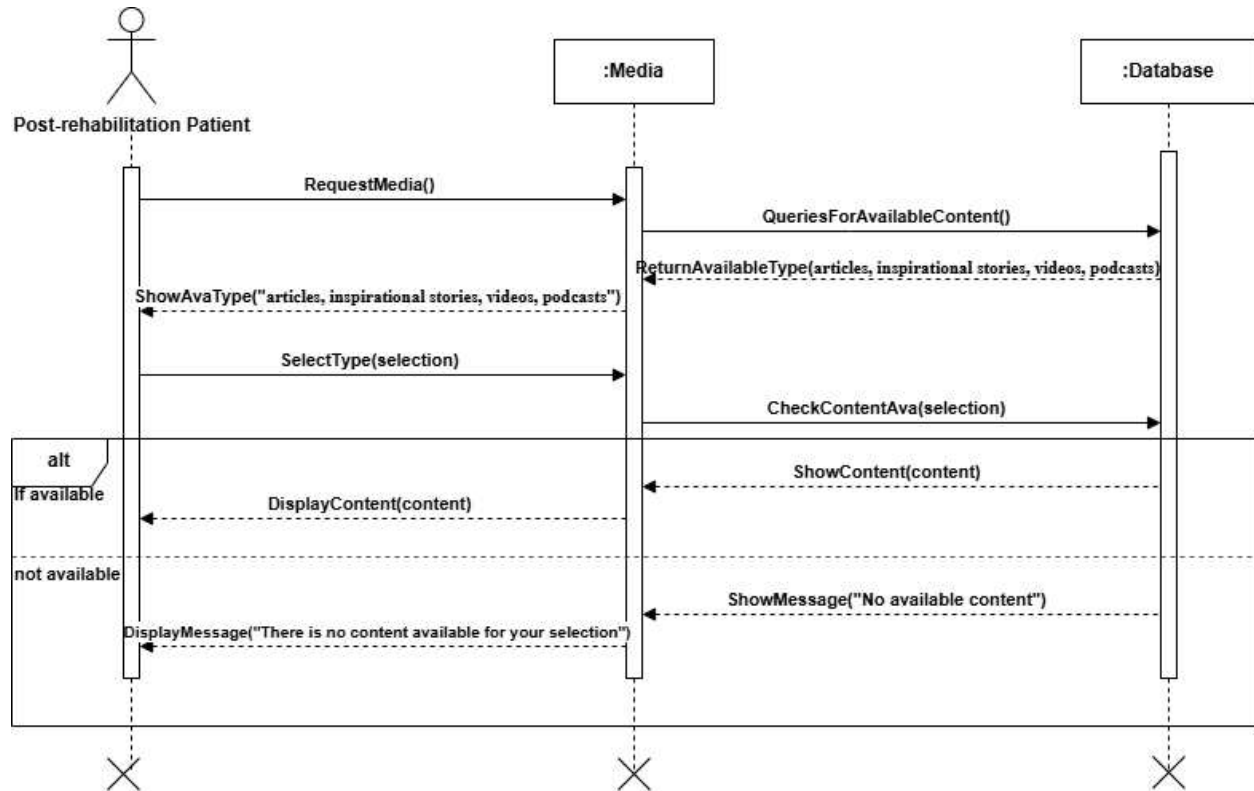


Figure 14: Sequence Diagram (View Educational Information)

3.5.11 View Achievement Badges

This diagram shows how post-rehabilitation patients earn achievement badges by completing and marking their activities as done, motivating progress and rewarding accomplishments.

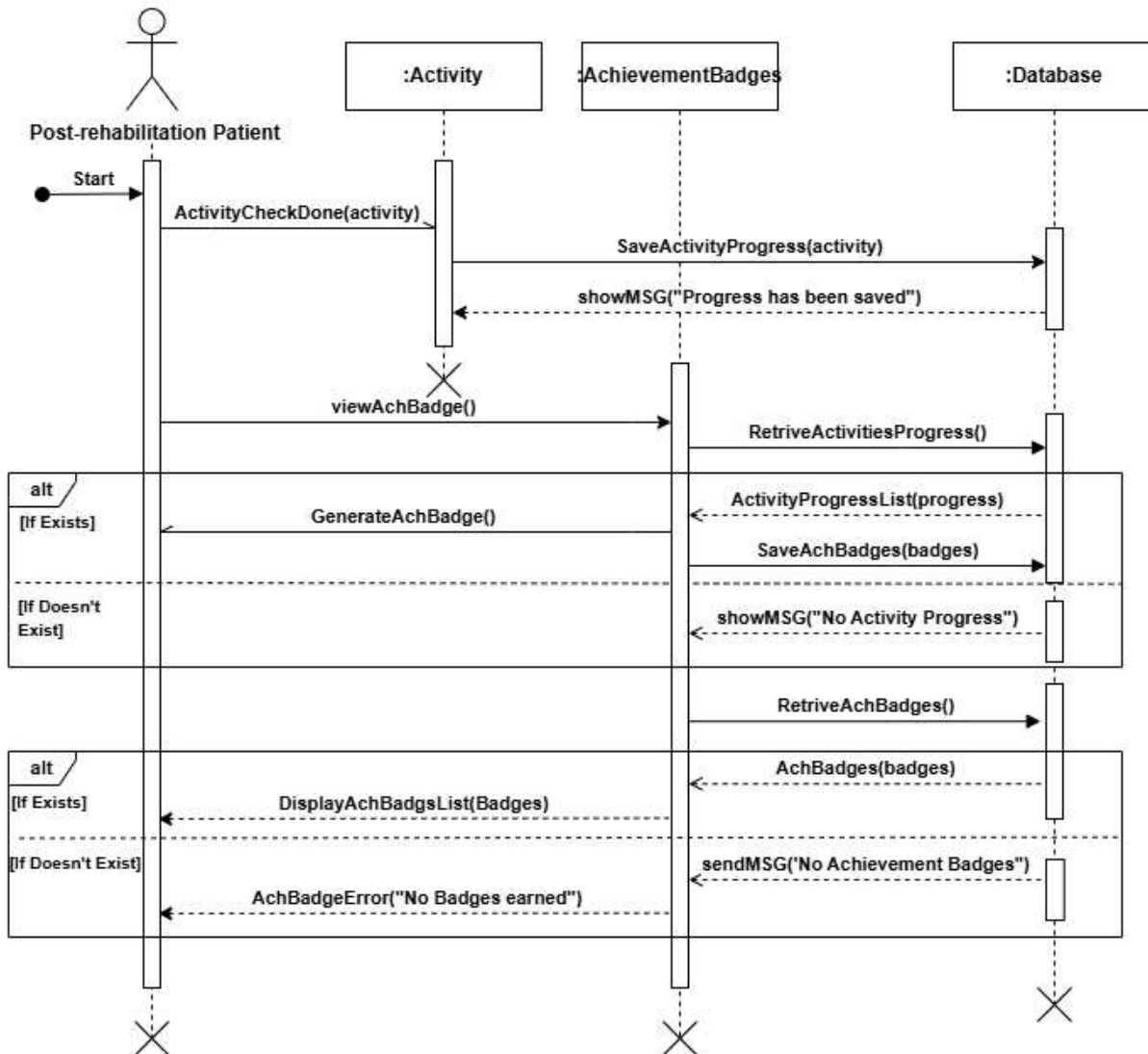


Figure 15: Sequence Diagram (View Achievement Badges)

Chapter 4: System Design

4.1 Background

In this chapter, we will design the Class Diagram and Entity-Relationship (ER) Diagram for the system. The Class Diagram will illustrate the structure of the system by depicting the classes, their attributes, methods, and the relationships among them. Following this, we will create the ER Diagram, which will provide a visual representation of the data entities, their attributes, and the relationships between these entities within the system.

Finally, we will map the relational schema, translating the ER model into a format suitable for implementation in a relational database. This mapping will ensure that the system's data is organized efficiently and can be effectively managed.

A class diagram is a fundamental component of the Unified Modeling Language (UML) that visualizes the structure of a software system by showing its classes, attributes, methods, and the relationships between them. It serves as a blueprint throughout the software development life cycle, supporting both system understanding and implementation.



The class diagram above represents the architectural structure of **Obour**, a mobile application aimed at supporting individuals in addiction recovery through activity planning, emotional support, and progress tracking. The system is composed of multiple interconnected classes, each designed to perform specific roles within the app's ecosystem:

- The main class initializes the app and navigates to the Obour_homepage if they log in or sign up successfully. From there, users can access various modules including activities, badges, profiles, and emotional support features.
- The auth_user and related Signup_* classes handle user registration and authentication, allowing users to sign up with personal data and preferences such as interests and hobbies. These classes enable navigation between different onboarding screens (Signup_new_user, Signup_select_Activity, etc.).
- The Activity_section and its related classes (Add_activity, Activity_edit, Activity_view) support the core functionality of planning, editing, and viewing personal recovery activities.
- The Emotion_entry class allows users to log their emotional states, which are sent to the chatgt_service class. This service class interfaces with an AI API to generate supportive quotes, which are then displayed through the Quote_display class.
- The Badges class generates motivational badges for users based on their completed activities and progress. These visual rewards are displayed in the Badge_display widget to encourage engagement.
- The Profile class and associated Profile_edit, Profile_contact_edit, and Profile_contact_add classes manage user profile information and emergency contact details. These screens allow users to update their data, connect with support contacts, and configure their profiles.
- The system also supports contact management for crisis intervention via the Sms_helper class, which handles message sending through the default SMS platform.

This class structure ensures a modular, maintainable, and user-centered application architecture that supports both the psychological and practical needs of individuals in recovery.

4.3 ER Diagram

The ER diagram represents the data model of the OBOUR application, which is designed to support post-rehabilitation patients through personalized activity planning, motivational support, and peer interaction. At the center of the model is the **Post-rehabilitation Patient** entity, which stores essential personal information such as name, email, password, phone number, and date of birth. Each patient can choose multiple **activities**, with each activity having a name and a priority level. These activities are linked to **schedule dates**, which include specific dates, start and end of activities times, and a completion status, allowing for structured weekly plans. The patient can also define their **availability** by selecting specific days and time ranges they are free. To support motivation, the system provides **quotes** based on the patient's selected quote type and emotional state. Additionally, patients can engage with different types of **media**, such as videos or podcasts, which are stored with their type and URL. For social support, patients can manage a list of **peer contacts**, each with a name and phone number, enabling easy communication through the app. The relationships between these entities ensure a fully integrated experience, allowing the system to generate personalized schedules, display relevant content, and facilitate social encouragement—all aimed at improving patient adherence and recovery outcomes. **Figure 17** show ER diagram

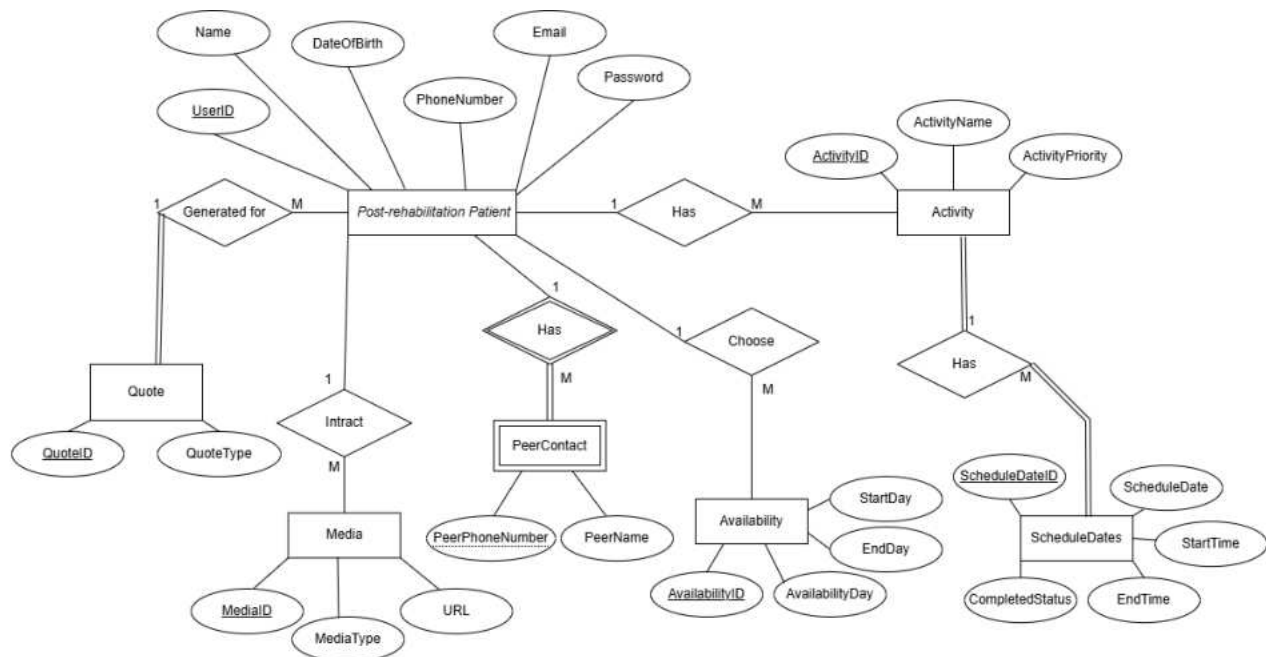


Figure 17: ER Diagram

4.4 Normalized Relational Shema

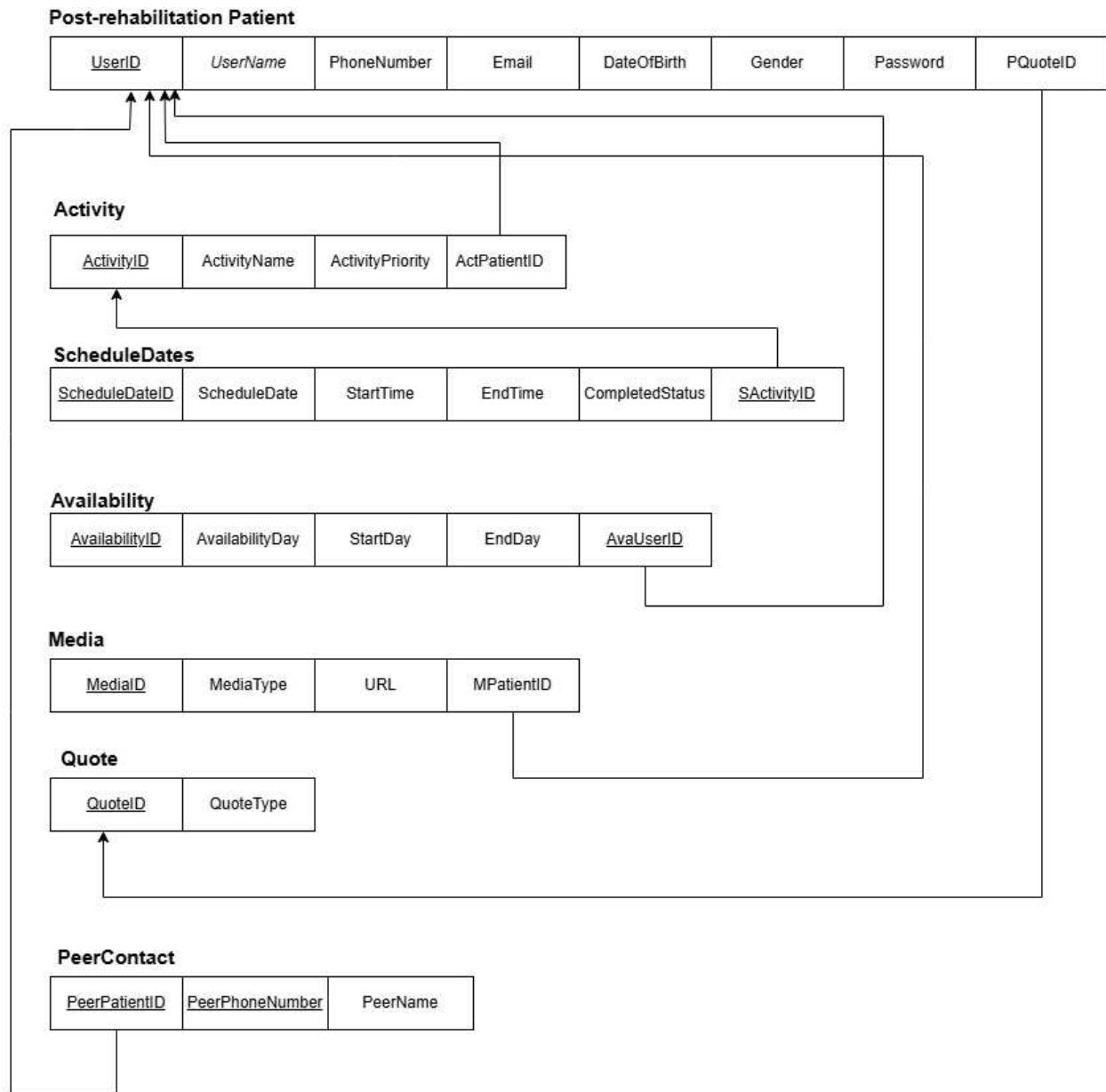


Figure 18: Relational Schema

Chapter 5: Implementation

This chapter presents the implementation phase of the Obour application, focusing on how the planned features and system architecture were translated into a fully functional platform. It provides a detailed account of the main system functionalities, the use of Firebase as the backend solution, code debugging efforts, and the final packaging and documentation process.

5.1 Introduction

Implementation represents the phase where the design and requirements of the Obour application were brought to life through actual coding and integration of various components. It involves the practical realization of system features intended to support individuals in recovery from substance use. This stage covers the development of core functionalities, backend integration using Firebase services, and the handling of challenges such as debugging and troubleshooting. Emphasis was placed on writing clean, maintainable code and ensuring smooth communication between frontend and backend components. Additionally, this chapter addresses the preparation of deployment-ready builds and comprehensive documentation to support future maintenance and scalability of the system.

5.2 Tools

- **Flutter UI Framework with Dart Programming Language:** Chosen for its compatibility with iOS and Android platforms.
- **Dart Programming Language:** Used to program the Obour system.
- **Git/GitHub:** Adopted by the team members for collaboration on projects.
- **FireBase:** Included as a back-end service platform in our project.
- **Android Studio:** Used to build Android applications.
- **Figma:** UI/UX design software.

5.3 System Implementation

5.3.1 System Functionalities

5.3.1.1 Customized weekly Activity plan

This feature is designed to help post-rehabilitation patients maintain structure and motivation in their daily lives through a personalized activity schedule. It includes two main capabilities: **automatic generation of a weekly plan** and **manual editing of the plan**.

5.3.1.1.1 Generate Weekly Activity

- **Purpose:**
 - Automatically generates a customized weekly activity plan based on the patient's available time, selected activities, and activity priority level.
- **How It Works:**
 - During sign-up or while editing the activity plan, patients provide:
 - Preferred activities
 - Activity priorities (e.g., 1 – exercise, 2 – reading, 3 – swimming)
 - Available days and time slots
 - The system processes this input and generates a full weekly activity schedule tailored to the patient's availability and preferences. The scheduled activities are:
 - Spread intelligently across different days
 - Assigned start and end times within the user's free time
 - Ordered by priority, so higher-priority activities are scheduled first
 - Stored securely in Firebase Firestore for later access and display
- **Supporting Methods and UI Behavior:**
 1. **generateWeeklySchedule(DateTime weekStartDate)**
 - **What it does:**

This is the **core scheduling function**. It:

 - Retrieves the current user's activities and availability from Firestore
 - Clears any previously scheduled activities for the current week, so no duplicated in the activity plan
 - Calculates free time slots based on availability
 - Assigns each activity to an available time slot, respecting priority
 - Resolves conflicts by moving lower-priority activities using `pushActivityLater`

```

9 Future<void> generateWeeklySchedule(DateTime weekStartDate) async {
10   final userId = FirebaseAuth.instance.currentUser?.uid;
11   if (userId == null) return;
12
13   final firestore = FirebaseFirestore.instance;
14   DateTime currentWeekStart = _findBeginningOfWeek(weekStartDate);
15
16   final activitiesSnapshot = await firestore
17     .collection('Patient')
18     .doc(userId)
19     .collection('Activities')
20     .orderBy('Priority')
21     .get();
22
23   List<Map<String, dynamic>> activities = activitiesSnapshot.docs.map((doc) {
24     return {
25       'id': doc.id,
26       'ActivityName': doc.data()['ActivityName'],
27       'Priority': doc.data()['Priority'],
28     };
29   }).toList();
30
31   final availabilitySnapshot = await firestore
32     .collection('Patient')
33     .doc(userId)
34     .collection('Availability')
35     .get();
36
37   Map<String, Map<String, String>> availability = {};
38   for (var doc in availabilitySnapshot.docs) {
39     availability[doc.id] = {
40       'start': doc.data()['start'],
41       'end': doc.data()['end'],
42     };
43   }
44
45   Map<String, List<Map<String, String>>> freeSlots = {};
46   for (var day in availability.keys) {
47     final start = availability[day]['start'];
48     final end = availability[day]['end'];
49     freeSlots[day] = _generateTimeSlots(start, end, 30);
50   }
51
52   List<DateTime> weekDates = List.generate(7, (index) => currentWeekStart.add(Duration(days: index)));

```

Figure 20: generateWeeklySchedule method

```

72   Map<String, List<Map<String, dynamic>>> dayUsedSlots = {};
73
74   for (var activity in activities) {
75     final activityId = activity['id'];
76     final activityRef = firestore.collection('Patient').doc(userId).collection('Activities').doc(activityId);
77
78     for (var date in weekDates) {
79       String arabicDay = _getArabicDayName(date.weekday);
80
81       if (!availability.containsKey(arabicDay) || (freeSlots[arabicDay]!.isEmpty)) continue;
82
83       if (!dayUsedSlots.containsKey(arabicDay)) {
84         dayUsedSlots[arabicDay] = {};
85       }
86
87       for (var slot in freeSlots[arabicDay]!) {
88         bool slotOccupied = dayUsedSlots[arabicDay]!.any((used) => used['start'] == slot['start'] && used['end'] == slot['end']);
89
90         if (!slotOccupied) {
91           var scheduledDocRef = await activityRef.collection('ScheduledDates').add({
92             'scheduledDate': Timestamp.fromDate(
93               DateTime(date.year, date.month, date.day, int.parse(slot['start']).split(":")[0], int.parse(slot['start']).split(":")[1]),
94             ),
95             'startTime': slot['start'],
96             'endTime': slot['end'],
97             'isCompleted': false,
98           });
99
100           dayUsedSlots[arabicDay]!.add({
101             'activityId': activityId,
102             'scheduledDate': scheduledDocRef.id,
103             'start': slot['start'],
104             'end': slot['end'],
105             'priority': activity['Priority'],
106           });
107
108           break;
109         } else {
110           var conflictActivity = dayUsedSlots[arabicDay]!.firstWhere((used) => used['start'] == slot['start'] && used['end'] == slot['end']);
111
112           if (conflictActivity['priority'] > activity['Priority']) {
113             await pushActivityLater(
114               arabicDay,
115               conflictActivity,
116               freeSlots,
117               dayUsedSlots,
118               date,
119               firestore,
120               userId
121             );
122
123             await firestore
124               .collection('Patient')
125               .doc(userId)
126               .collection('Activities')
127               .doc(activityId)
128               .collection('ScheduledDates')

```

Figure 19: generateWeeklySchedule method

- **Why it's important:**
 - Automates weekly scheduling for users, ensures no overlaps, respects user-defined time windows, and prioritizes tasks efficiently.

2. pushActivityLater(...)

- **What it does:**

When a higher-priority activity wants to occupy a time slot that's already taken, this method:

 - Finds the next available free slot on the same day
 - Moves (updates) the lower-priority activity to that new slot
 - Updates Firestore with the new schedule

```

155 Future<void> pushActivityLater(
156     String day,
157     Map<String, dynamic> conflictActivity,
158     Map<String, List<Map<String, String>>> freeSlots,
159     Map<String, List<Map<String, dynamic>>> dayUsedSlots,
160     DateTime date,
161     FirebaseFirestore firestore,
162     String userId
163 ) async {
164     List<Map<String, String>> slots = freeSlots[day]!;
165
166     for (var slot in slots) {
167         bool slotOccupied = dayUsedSlots[day]!.any((used) => used['start'] == slot['start'] && used['end'] == slot['end']);
168
169         if (!slotOccupied) {
170             await firestore
171                 .collection('Patient')
172                 .doc(userId)
173                 .collection('Activates')
174                 .doc(conflictActivity['activityId'])
175                 .collection('ScheduledDates')
176                 .doc(conflictActivity['scheduledDateId'])
177                 .update({
178                     'startTime': slot['start'],
179                     'endTime': slot['end'],
180                     'scheduledDate': Timestamp.fromDate(
181                         DateTime(date.year, date.month, date.day, int.parse(slot['start'].split(":")[0]), int.parse(slot['start'].split(":")[1])),
182                     ),
183                 });
184
185             dayUsedSlots[day]!.add({
186                 'activityId': conflictActivity['activityId'],
187                 'scheduledDateId': conflictActivity['scheduledDateId'],
188                 'start': slot['start'],
189                 'end': slot['end'],
190                 'priority': conflictActivity['priority'],
191             });
192
193             break;
194         }
195     }
196 }

```

Figure 21: pushActivityLater method

- **Why it's important:**
 - Enables **priority-based conflict resolution**, keeping all activities scheduled while honoring user preferences.

5.3.1.1.2 Edit an Activity

- **Purpose:**
 - To allow post-rehabilitation patients to manually modify the details of their activity plan — including available days, start time, and end time — so they can adapt their schedule as needed.
- **How It Works:**

- Patients can access this feature from the Home Page screen. Once they tap the Edit Activity Plan button, they are navigated to the SignupActivitySelection screen.
- The system fetches and displays the patient's current activity data. The patient can:
 - **Add, edit, or delete** activities
 - Reorder activities based on **priority**
 - Modify **available days** and **available time slots**
- All changes are immediately reflected in Firebase Firestore. On the final page of the edit activity plan flow, the system calls the method generateWeeklyActivityPlan() to regenerate the weekly activity schedule based on the newly updated data.

- **Supporting Methods and UI Behavior:**

1. **selectedDate**

- **What it does:**
 - This is a state variable of type DateTime that stores the currently selected day in the weekly calendar view. It is updated whenever the user taps on a different day in the horizontal weekday selector.
- **Why it's important:**
 - selectedDate ensures the app displays only the activities scheduled for that specific date, maintaining an accurate and responsive user interface.

2. **displayedWeekStartDate**

- **What it does:**
 - This variable holds the DateTime value for the first day (Sunday) of the currently visible week. When the user navigates to a different week using arrow buttons, this variable updates accordingly.
- **Why it's important:**
 - It determines which 7 days are visible in the weekly selector and keeps the interface aligned with the week being viewed.

3. `_buildTodayActivitiesHeader()`

- **What it does:**

This widget builds the header of the main weekly schedule screen. It contains:

- A localized Arabic title: "أنشطة اليوم" (Today's Activities)
- A button labeled "Edit Plan" that navigates the user to the `FavoriteActivitiesScreen`

```
192 Widget _buildTodayActivitiesHeader() {
193   return Row(
194     mainAxisAlignment: MainAxisAlignment.spaceBetween,
195     children: [
196       const Text(
197         "أنشطة اليوم",
198         style: TextStyle(fontSize: 18, fontWeight: FontWeight.bold),
199       ),
200       GestureDetector(
201         onTap: () {
202           Navigator.push(context, MaterialPageRoute(builder: (context) => const FavoriteActivitiesScreen()));
203         },
204         child: Row(
205           mainAxisAlignment: MainAxisAlignment.min,
206           children: [
207             const Text(
208               "تعديل الخطة",
209               style: TextStyle(fontSize: 15, color: Color(0xFF373A40)),
210             ),
211             const SizedBox(width: 9),
212             Image.asset(
213               'assets/images/Edit.png',
214               width: 14,
215               height: 14,
216             ),
217           ],
218         ),
219       ),
220     ],
221   );
222 }
```

Figure 22: `buildTodayActivitiesHeader` method

- **Why it's important:**

- Provides clear orientation to the user and a direct path to edit their plan.

4. `_buildWeekDaysSelector()`

- **What it does:**

This method constructs a horizontally scrollable row of 7 boxes, one for each day from Sunday to Saturday. Each box is generated using the `dateBox()` method and includes:

- A visible indicator for the selected day using `isSameDay()`
- A call to `generateWeeklySchedule()` when the week changes
- Navigation arrows that allow switching between weeks

```
224 Widget _buildWeekDaysSelector() {
225   return SizedBox(
226     height: 70,
227     child: Row(
228       children: [
229         IconButton(
230           padding: EdgeInsets.zero,
231           constraints: const BoxConstraints(),
232           icon: Image.asset('assets/images/arrowright.png', width: 18, height: 18),
233           onPressed: _goToPreviousWeek,
234         ),
235         const SizedBox(width: 0),
236         Expanded(
237           child: ListView.builder(
238             controller: _scrollController,
239             scrollDirection: Axis.horizontal,
240             itemCount: 7,
241             itemBuilder: (context, index) {
242               final date = displayedWeekStartDate.add(Duration(days: index));
243               return GestureDetector(
244                 onTap: () {
245                   setState(() {
246                     selectedDate = date;
247                   });
248                 },
249                 child: dateBox(
250                   DateFormat.E('ar').format(date),
251                   DateFormat.d().format(date),
252                   isSameDay(date, selectedDate),
253                 ),
254               );
255             },
256           ),
257         ),
258         const SizedBox(width: 0),
259         IconButton(
260           padding: EdgeInsets.zero,
261           constraints: const BoxConstraints(),
262           icon: Image.asset('assets/images/arrowleft.png', width: 18, height: 18),
263           onPressed: _goToNextWeek,
264         ),
265       ],
266     ),
267   );
268 }
```

Figure 23: *buildWeekDaysSelector* method

- **Why it's important:**
 - Offers intuitive week navigation and precise date selection, forming the core interaction mechanism for viewing daily plans.

5. `_buildActivitiesList()`

- **What it does:**

This method uses a `FutureBuilder` to retrieve and display the list of activities scheduled for `selectedDate`. It responds to different states:

- **Loading:** shows a spinner
- **Error:** displays an error message
- **Empty:** shows "لا توجد أنشطة لهذا اليوم"
- **Success:** maps each activity to a visual card via `activityCard()`

```

270   Widget _buildActivitiesList() {
271     return FutureBuilder<List<Map<String, dynamic>>>>(
272       future: getActivitiesForSelectedDate(selectedDate),
273       builder: (context, snapshot) {
274         if (snapshot.connectionState == ConnectionState.waiting) {
275           return const Center(child: CircularProgressIndicator());
276         } else if (snapshot.hasError) {
277           return const Center(child: Text('حدث خطأ أثناء تحميل الأنشطة'));
278         } else if (!snapshot.hasData || snapshot.data!.isEmpty) {
279           return const Center(child: Text('لا توجد أنشطة لهذا اليوم'));
280         }
281
282         final activities = snapshot.data!;
283
284         return Column(
285           children: activities.map((activity) => activityCard(activity)).toList(),
286         );
287       },
288     );
289   }

```

Figure 24: `buildActivitiesList` method

- **Why it's important:**
 - Handles asynchronous data fetching with real-time UI updates, ensuring data accuracy and responsiveness

5.3.1.2 Connect with peers through SMS msg

- **Purpose:**
 - This feature encourages patients to reach out and maintain supportive connections with peers using SMS messages. It plays a vital role in reinforcing social bonds, reducing isolation, and fostering a sense of community during the recovery process.
- **How It Works:**
 - When a user sends an SMS to a peer through the app, the system:
 - Sends the message using a platform channel (MethodChannel) to access native SMS APIs.
- **Supporting Methods and UI Behavior:**

1. buildSearchField()

- **What it does:**
 - Provides a search bar to filter contacts by name.
 - Updates searchQuery whenever the user types.

```
185   Widget buildSearchField() {
186     return TextField(
187       controller: _searchController,
188       textAlign: TextAlign.right,
189       onChanged: (value) {
190         setState(() {
191           searchQuery = value;
192         });
193       },
194       decoration: InputDecoration(
195         suffixIcon: Padding(
196           padding: EdgeInsets.all(10),
197           child: Image.asset('assets/images/ico-search.png', width: 20, height: 20),
198         ),
199         hintText: "البحث",
200         fillColor: Colors.white,
201         filled: true,
202         contentPadding: EdgeInsets.symmetric(vertical: 10, horizontal: 12),
203         enabledBorder: OutlineInputBorder(
204           borderRadius: BorderRadius.circular(12),
205           borderSide: BorderSide(color: Color(0xFFE6E6E6), width: 1.5),
206         ),
207         focusedBorder: OutlineInputBorder(
208           borderRadius: BorderRadius.circular(12),
209           borderSide: BorderSide(color: Color(0xFFE6E6E6), width: 2.0),
210         ),
211       ),
212     );
213   }
```

Figure 25: buildSearchField method

2. buildContactsList()

- **What it does:**
 - Filters contacts using the search query.
 - Displays them in a scrollable list.
 - Each contact appears as a card with:
 - Name and phone number
 - Menu to **edit** or **delete** the contact

3. SmsHelper.sendSms(phone, message)

- **What it does:**
 - Sends an SMS using platform-specific channels

```
429 class SmsHelper {
430     static const platform = MethodChannel('com.example.sms');
431
432     static Future<void> sendSms(String phone, String message) async {
433         try {
434             await platform.invokeMethod('sendSms', {
435                 'phone': phone,
436                 'message': message,
437             });
438         } on PlatformException catch (e) {
439             print("Failed to send SMS: '${e.message}'.");
440         }
441     }
442 }
```

Figure 26: class SmsHelper

5.3.1.2.1 Add new peer contact

- **Purpose:**
 - This screen allows the user to manually add one or more peer contacts (name and phone number), view them in a list, optionally delete them before saving, and finally store them in Firestore under the current patient's document.
- **How It Works:**
 - **Adding a Contact:**

When the add button is tapped:

 - The name and phone are trimmed and validated.
 - Phone must:
 - Start with **05** or **966**
 - Be **10 or more digits**

- If valid:
 - Contact is added to `_contacts`
 - Text fields are cleared
- If invalid:
 - A red SnackBar appears with a warning message
"رقم الهاتف يجب أن يبدأ بـ 05 أو 966 ويكون 10 أرقام أو أكثر !"

```

172         const SizedBox(height: 10),
173         TextField(
174           controller: _phoneController,
175           textDirection: TextDirection.rtl,
176           textAlign: TextAlign.right,
177           keyboardType: TextInputType.phone,
178           decoration: InputDecoration(
179             hintText: "أدخل رقم جهة الاتصال",
180             filled: true,
181             fillColor: Colors.white,
182             border: OutlineInputBorder(
183               borderRadius: BorderRadius.circular(8),
184               borderSide: BorderSide.none,
185             ),
186           ),
187         ),
188         const SizedBox(height: 15),
189         Center(
190           child: IconButton(
191             onPressed: () {
192               String name = _nameController.text.trim();
193               String phone = _phoneController.text.trim();
194
195               // Validation
196               bool isValidPhone = (phone.startsWith('05') || phone.startsWith('966')) && phone.length >= 10;
197
198               if (name.isNotEmpty && phone.isNotEmpty) {
199                 if (isValidPhone) {
200                   setState(() {
201                     _contacts.add({
202                       "name": name,
203                       "phone": phone,
204                     });
205                   });
206                   _nameController.clear();
207                   _phoneController.clear();
208                 } else {
209                   ScaffoldMessenger.of(context).showSnackBar(
210                     const SnackBar(
211                       content: Text(
212                         "رقم الهاتف يجب أن يبدأ بـ 05 أو 966 ويكون 10 أرقام أو أكثر !",
213                       textDirection: TextDirection.rtl,

```

Figure 27: Adding a Contact

- Saving Contacts:

Tapping "حفظ جهات الاتصال" (Save Contacts):

- Calls `_saveContactsToFirestore()` to upload all contacts

- Then navigates back (pops the screen)

```

17 Future<void> _saveContactsToFirestore() async {
18     final user = FirebaseAuth.instance.currentUser;
19     if (user == null) return;
20
21     final String userId = user.uid;
22     final CollectionReference contactsRef = FirebaseFirestore.instance
23         .collection('Patient')
24         .doc(userId)
25         .collection('contactPeer');
26
27     for (var contact in _contacts) {
28         await contactsRef.add({
29             'PeerName': contact['name'],
30             'PeerPhoneNumber': contact['phone'],
31         });
32     }
33 }

```

Figure 28: Saving Contacts

5.3.1.2.2 Edit on peer contact

- **Purpose:**
 - To enable users to edit an existing peer contact's name and phone number in the contactPeer subcollection within the current user's Firestore document.
- **How It Works:**
 - **Edit Contact Screen:**
 - Takes 3 required inputs:
 - initialName – the current name of the contact.
 - initialPhone – the current phone number.
 - contactId – the Firestore document ID of the contact.

```

64 padding: const EdgeInsets.all(16.0),
65 child: Column(
66   children: [
67     _buildStyledInputField("اسم جهة الاتصال", nameController),
68     const SizedBox(height: 16),
69     _buildStyledInputField("رقم الهاتف", phoneController, isPhone: true),
70     const SizedBox(height: 20),
71     ElevatedButton(
72       width: double.infinity,
73       child: ElevatedButton(
74         style: ElevatedButton.styleFrom(
75           backgroundColor: const Color(0xFF33AA00),
76           padding: const EdgeInsets.symmetric(vertical: 14),
77           shape: RoundedRectangleBorder(borderRadius: BorderRadius.circular(0)),
78         ),
79         onPressed: () async {
80             final user = FirebaseAuth.instance.currentUser;
81             if (user == null) return;
82
83             final docRef = FirebaseFirestore.instance
84                 .collection('Patient')
85                 .doc(user.uid)
86                 .collection('contactPeer')
87                 .doc(widget.contactId);
88
89             await docRef.update({
90                 'PeerName': nameController.text.trim(),
91                 'PeerPhoneNumber': phoneController.text.trim(),
92             });
93
94             Navigator.pop(context, true); // return true to trigger refresh
95         },
96       ),
97     const Text("تم", style: TextStyle(fontSize: 18, color: Colors.white)),
98   ],
99 )

```

Figure 29: EditContactScreen

5.3.1.3 Achievement badges

This feature visually rewards patients for their commitment to completing scheduled activities. It reinforces motivation by unlocking new badges as the user progresses through their recovery journey.

- **Purpose:**
 - Encourage continuous engagement by awarding visual badges based on the number of completed activities.
- **How It Works:**
 - Each badge has a predefined threshold of completed activities (e.g., 50, 100, 175). When users complete enough activities, the badge becomes unlocked and is displayed in full color.
- **Supporting Methods and UI Behavior:**

1. buildBadgesSection()

- **What it does:**
 - Displays the user's progress (total completed activities).
 - Shows a grid of reward badges.
 - Highlights the highest badge earned.
 - Navigates to activity history when clicking the notification icon.

```
205   Widget _buildBadgesSection() {
206     return Expanded(
207       child: Padding(
208         padding: EdgeInsets.symmetric(horizontal: 20),
209         child: Column(
210           crossAxisAlignment: CrossAxisAlignment.start,
211           children: [
212             Text(
213               "نقاط إنجاز",
214               style: TextStyle(
215                 fontSize: 20,
216                 fontWeight: FontWeight.bold,
217                 color: Color(0xFF292C30),
218               ),
219             ),
220             SizedBox(height: 8),
221             Expanded(
222               child: GridView.builder(
223                 gridDelegate: SliverGridDelegateWithFixedCrossAxisCount(
224                   crossAxisCount: 2,
225                   crossAxisSpacing: 12,
226                   mainAxisSpacing: 12,
227                   childAspectRatio: 1,
228                 ),
229                 itemCount: badges.length,
230                 itemBuilder: (context, index) {
231                   final badge = badges[index];
232                   bool isUnlocked = doneActivitiesCount >= badge["points"];
233                   return _buildBadgeItem(badge["image"], badge["name"], badge["points"], isUnlocked);
234                 },
235               ),
236             ),
237           ],
238         ),
239       ),
240     );
241   }
```

Figure 30: Achievement Badges

2. fetchCompletedActivities()

- What it does:

This is the core data-fetching function. It:

- Retrieves the current user's UID from Firebase Authentication.
- Queries Firestore for all "Activates" and their nested "ScheduledDates" collections.
- Counts how many ScheduledDates documents have isCompleted == true.
- Updates the state variable doneActivitiesCount, which controls badge unlocking and progress display.

```
34 Future<void> fetchCompletedActivities() async {
35   try {
36     final uid = FirebaseAuth.instance.currentUser?.uid;
37     if (uid == null) return;
38
39     final activatesSnapshot = await FirebaseFirestore.instance
40       .collection('Patient')
41       .doc(uid)
42       .collection('Activates')
43       .get();
44
45     int completedCount = 0;
46
47     for (var activityDoc in activatesSnapshot.docs) {
48       final scheduledSnapshot = await FirebaseFirestore.instance
49         .collection('Patient')
50         .doc(uid)
51         .collection('Activates')
52         .doc(activityDoc.id)
53         .collection('ScheduledDates')
54         .where('isCompleted', isEqualTo: true)
55         .get();
56
57       completedCount += scheduledSnapshot.docs.length;
58     }
59
60     setState(() {
61       doneActivitiesCount = completedCount;
62     });
63   } catch (e) {
64     print('Error fetching completed activities: $e');
65   }
66 }
```

Figure 31: *fetchCompletedActivities*

3. Visual Sections in build()

- Header Section:

- Displays a background image and a bell icon (linked to activity history).
- Shows the latest unlocked badge and its name dynamically using `_getLatestUnlockedBadge()` and `_getLatestUnlockedBadgeName()`.

- Activity Progress:

- Displays how many activities the user has completed (e.g., “لقد قمت بـ 120 نشاط”)

- **Badges Grid:**
 - Shows all available badges using a grid.
 - Each badge item includes an image, a name, and the required points (activities).
 - Locked badges are shown faded (lower opacity), and unlocked ones are colored.

```

70      @override
71      Widget build(BuildContext context) {
72          return Scaffold(
73              backgroundColor: Color(0xFFFDf1E9),
74              body: SafeArea(
75                  child: Column(
76                      children: [
77                          _buildHeader(),
78                          _buildActivityProgress(),
79                          _buildBadgesSection(),
80                      ],
81                  ),

```

Figure 32: build

5.3.1.4 Notification remainder for Activities

This feature automatically sends daily reminders to users at predefined times to encourage consistent engagement with their recovery activities.

- **Purpose:**
 - Motivate users to complete their scheduled tasks and maintain progress on their recovery journey.
- **How It Works:**
 - The system utilizes the flutter_local_notifications plugin in combination with the timezone package to schedule three daily notifications:
 - Morning Reminder – 8:00 AM
 - Afternoon Reminder – 3:00 PM
 - Evening Reminder – 8:00 PM
 - Each notification includes a motivational Arabic message:
 - "تذكير من عبور 🏃: خطوه اليوم تصنع فرق غدا! أكمل نشاطاتك واستمر نحو أهدافك"

```

7 Future<void> scheduleDailyActivityReminder() async {
8   try {
9     // ☀ Morning: 8:00 AM
10    await _scheduleSingleNotification(
11      id: 1,
12      hour: 08,
13      minute: 28,
14    );
15
16    // ☀ Afternoon: 3:00 PM
17    await _scheduleSingleNotification(
18      id: 2,
19      hour: 15,
20      minute: 00,
21    );
22
23    // 🌙 Evening: 8:00 PM
24    await _scheduleSingleNotification(
25      id: 3,
26      hour: 20,
27      minute: 00,
28    );
29  }
30 }

```

Figure 33: Notification remainder

- **Supporting Methods and UI Behavior:**

1. **scheduleSingleNotification({required int id, required int hour, required int minute})**

- **What it does:**

- This is a helper method that schedules a single notification for a specified time. It is called by the scheduleDailyActivityReminder() method to schedule the three different daily reminders.
- **Input Parameters:**
 - id: A unique identifier for the notification (used to distinguish between multiple notifications).
 - hour: The hour (in 24-hour format) when the notification should be triggered.
 - minute: The minute when the notification should trigger.
- The method calculates the exact time when the notification should be sent using the _nextInstanceOfTime() helper method.
- The flutterLocalNotificationsPlugin.zonedSchedule() method is used to schedule the notification at the specified time, in the local timezone.
 - The notification contains the **title** "تذكير من عبور 🏠" and the body message "خطوه اليوم تصنع فرق غدا! أكمل نشاطاتك واستمر نحو أهدافك".
 - **Notification Details:** It uses high importance and priority (Importance.max and Priority.high) to ensure the notification is noticed by the user. It also ensures that the notification appears even if the device is idle.

```

37 Future<void> _scheduleSingleNotification({
38   required int id,
39   required int hour,
40   required int minute,
41 }) async {
42   final scheduledTime = _nextInstanceOfTime(hour, minute);
43   print("🔔 Scheduling notification (ID: $id) at $scheduledTime");
44
45   await flutterLocalNotificationsPlugin.zonedSchedule(
46     id,
47     'تذكير من عور',
48     'خطوة اليوم تملع فرق غدا! اكمل نشاطك واستمر نحو أهدافك',
49     scheduledTime,
50     const NotificationDetails(
51       android: AndroidNotificationDetails(
52         'activity_reminder_channel',
53         'Activity Reminders',
54         channelDescription: 'Multiple reminders to complete your activities',
55         importance: Importance.max,
56         priority: Priority.high,
57       ),
58     ),
59     androidScheduleMode: AndroidScheduleMode.exactAllowWhileIdle,
60     uiLocalNotificationDateInterpretation:
61       UILocalNotificationDateInterpretation.absoluteTime,
62     matchDateTimeComponents: DateTimeComponents.time,
63   );
64 }

```

Figure 34: scheduleSingleNotification

5.3.1.5 Educational information about substances (YT, Podcasts, and Articles)

This feature is designed to support post-rehabilitation patients in understanding substance use and recovery through accessible, engaging educational content. It includes three main capabilities: displaying curated YouTube videos, sharing relevant podcasts, and providing informative articles. These resources aim to enhance awareness, reinforce recovery knowledge, and empower patients with the tools to make informed decisions.

```

18  @override
19  Widget build(BuildContext context) {
20    return Scaffold(
21      backgroundColor: const Color(0xFFCECE),
22      body: SafeArea(
23        child: SingleChildScrollView(
24          child: Column(
25            children: [
26              _buildHeader(),
27
28              // Media Section
29              Padding(
30                padding: const EdgeInsets.all(16.0),
31                child: Column(
32                  crossAxisAlignment: CrossAxisAlignment.center,
33                  children: [
34                    const Text(
35                      "أخبار",
36                      style: TextStyle(
37                        fontSize: 22,
38                        fontWeight: FontWeight.bold,
39                      ), // TextStyle
40                    ), // Text
41                    const SizedBox(height: 20),
42
43                    _buildMediaBox(
44                      context,
45                      title: "أخبار",
46                      imagePath: 'assets/images/youtube.png',
47                      onTap: () => Navigator.push(
48                        context,
49                        MaterialPageRoute(builder: (context) => const VideoScreen()),
50                      ),
51                    ),
52                    const SizedBox(height: 15),
53
54                    _buildMediaBox(
55                      context,
56                      title: "أخبار",
57                      imagePath: 'assets/images/podcast.png',
58                      onTap: () => Navigator.push(
59                        context,
60                        MaterialPageRoute(builder: (context) => const PodcastScreen()),
61                      ),
62                    ),
63                  ], // Column
64                ), // Padding
65              ], // Column
66            ), // SingleChildScrollView
67          ), // SafeArea
68        ); // Scaffold
69      );
70    }

```

Figure 35:buildMediaScreen

5.3.1.5.1 View Educational Videos

- **Purpose:**
 - To allow post-rehabilitation patients to access a library of educational YouTube videos directly within the app, helping them expand their knowledge, stay motivated, and support their recovery journey.
- **How It Works:**
 - Patients can navigate to the *VideoScreen* from the media screen. Once on this screen, the app performs the following actions:
 - Fetches a list of video documents from the Firebase Firestore collection named '**Videos**'.
 - For each document, it extracts the videoTitle and videoUrl.
 - Converts the URL into a YouTube video ID and embeds the video using the YoutubePlayer widget.
 - Displays each video along with its title in a scrollable column layout.
 - The screen also includes a custom app bar and bottom navigation bar to maintain UI consistency.
- **Supporting Methods and UI Behavior:**

1. StreamBuilder

- Why it's important:

- Ensures the video list is always up-to-date and reflects the latest content without requiring a manual refresh.

```
// 📺 StreamBuilder to fetch all videos
StreamBuilder<QuerySnapshot>(  
  stream: FirebaseFirestore.instance.collection('Videos').snapshots(),  
  builder: (context, snapshot) {  
    if (snapshot.connectionState == ConnectionState.waiting) {  
      return const Center(child: CircularProgressIndicator());  
    }  
  
    if (snapshot.hasError) {  
      print("❌ Firestore error: ${snapshot.error}");  
      return const Center(child: Text('خطأ في تحميل الفيديوهات ⚠️'));  
    }  
  
    if (!snapshot.hasData || snapshot.data!.docs.isEmpty) {  
      return const Center(child: Text('لا توجد فيديوهات'));  
    }  
  
    final videos = snapshot.data!.docs;  
  
    print("✅ Videos fetched: ${videos.length}");  
  }  
)
```

Figure 36: streamBuilder(Video)

2. YoutubePlayerController

- Why it's important:

- Allows smooth video playback and control directly inside the app, keeping users engaged within the same screen.

```

return Column(
  children: videos.map((video) {
    final videoData = video.data() as Map<String, dynamic>;
    final videoTitle = videoData['videoTitle'];
    final videoUrl = videoData['videoUrl'];
    final videoId = YoutubePlayer.convertUrlToId(videoUrl);

    if (videoId == null) {
      print("✗ Invalid video URL: $videoUrl");
      return const SizedBox(); // Skip if URL is not valid
    }

    final controller = YoutubePlayerController(
      initialVideoId: videoId,
      flags: const YoutubePlayerFlags(autoPlay: false),
    ); // YoutubePlayerController

    return Padding(
      padding: const EdgeInsets.all(16.0),
      child: Column(
        children: [
          YoutubePlayer(controller: controller),
          const SizedBox(height: 10),
          Text(videoTitle ?? '', style: const TextStyle(fontSize: 18, fontWeight: FontWeight.bold)),
        ],
      ), // Column
    ); // Padding
  }).toList(),
); // Column

```

Figure 37: YTController

3. `_buildHeader()`

- What it does:

- A background image (Bg.png)
- A custom back button (backOrange.png)
- A right-aligned text column with the Arabic title “فيديوهات” and subtitle “وسّع مداركك”
- An illustrative YouTube logo (Y.png)

- Why it's important:

- Provides immediate context and aesthetic appeal, aligning the content with the app’s theme and RTL layout for Arabic-speaking users.

```

112 Widget _buildHeader(BuildContext context) {
113   return SizedBox(
114     height: 250, // Adjust height as needed
115     child: Stack(
116       children: [
117         Positioned(
118           child: Image.asset(
119             'assets/images/bg.png',
120             fit: BoxFit.fill,
121             width: double.infinity,
122           ), // Image.asset
123         ), // Positioned
124
125         // Custom Back Button (Top Right)
126         Positioned(
127           top: 30,
128           right: 20,
129           child: GestureDetector(
130             onTap: () {
131               Navigator.pop(context);
132             },
133             child: Image.asset(
134               'assets/images/backOrange.png',
135               width: 40,
136               height: 40,
137             ), // Image.asset
138           ), // GestureDetector
139         ), // Positioned
140
141         // Properly Aligned Row for Text & Image
142         Positioned(
143           left: 20,
144           right: 100,
145           top: 70, // Adjusts row position properly
146
147           child: Align(
148             alignment: Alignment.centerRight,
149             child: Align(
150               alignment: Alignment.centerRight, // Pushes entire Row to the right
151               child: Row(
152                 textDirection: TextDirection.rtl,
153                 mainAxisAlignment: MainAxisAlignment.min, // Prevents Row from expanding fully
154                 children: [
155                   // Text Section (Right side in RTL)
156                   Column(
157                     crossAxisAlignment: CrossAxisAlignment.end,
158                     mainAxisAlignment: MainAxisAlignment.min,
159                     children: const [
160                       Text(
161                         "About Us",
162                         style: TextStyle(
163                           fontSize: 35,
164                           fontWeight: FontWeight.bold,
165                           color: Color(0xFF632880),
166                           fontFamily: "Inter",
167                         ), // TextStyle
168                       ), // Text
169                       SizedBox(height: 4),
170                       Text(
171                         "About Us",
172                         textAlign: TextAlign.right, // Ensures RTL alignment
173                         style: TextStyle(
174                           fontSize: 20,
175                           color: Color(0xFF632880),
176                           fontFamily: "Inter",
177                         ), // TextStyle
178                       ), // Text
179                     ],
180                   ), // Column
181                 ],
182               const SizedBox(width: 12),
183             ),
184           ),
185         ),
186       ],
187     ),
188   );
189 }

```

Figure 38: buildHeader(video)

5.3.1.5.2 Listen to a Podcast

- **Purpose:**
 - To provide post-rehabilitation patients with access to curated podcast episodes that support mental well-being, motivation, and ongoing recovery education. This feature helps users gain insight and stay engaged through audio content.
- **How It Works:**
 - Patients can navigate to the Podcast screen from the bottom navigation bar. The app retrieves podcast episode data from the **Firestore** collection named 'Podcasts'. Each entry in the collection includes:
 - podcastTitle — the episode's title
 - audioUrl — the streaming URL of the podcast audio
 - Each episode is displayed as a card with a play button. When a user taps the play icon, the system
 - Sets the selected URL as the playback source using **just_audio**.
 - Begins playing the episode.
 - Shows an interactive player with:
 - A duration slider

- Elapsed and total time indicators
- Pause, play, and stop buttons
- The system listens for audio duration and playback position updates in real time, updating the UI accordingly.

- **Supporting Methods and UI Behavior:**

1. **`_playPodcast(url, title)`**

- **Why it's important:**

- Ensures the app responds immediately to user actions and visually indicates which episode is active.

```
Future<void> _playPodcast(String url, String title) async {  
  try {  
    await _audioPlayer.setUrl(url);  
    _audioPlayer.play();  
    setState(() {  
      _currentlyPlaying = title;  
    });  
  } catch (e) {  
    print("✗ Error playing podcast: $e");  
  }  
}
```

Figure 39: playPodcasts

2. **`_formatDuration(duration)`**

- **Why it's important:**

- Provides clarity on episode progress and helps users track playback time easily.

```
String _formatDuration(Duration d) {  
  return d.inMinutes.toString().padLeft(2, '0') +  
    ':' +  
    (d.inSeconds % 60).toString().padLeft(2, '0');  
}
```

Figure 40: formatDuration

3. StreamBuilder<QuerySnapshot>

- What it does

Connects to Firestore in real time to retrieve and display podcast episodes. Handles:

- Loading (shows spinner)
- Errors (displays a message)
- Empty state (shows "لا توجد حلقات بودكاست حالياً")
- Success (renders podcast cards)

- Why it's important:

- Allows seamless, live updates from the backend without requiring manual refreshes, keeping the podcast library current.

```
109 StreamBuilder<QuerySnapshot>{
110   stream: FirebaseFirestore.instance.collection('Podcasts').snapshots(),
111   builder: (context, snapshot) {
112     if (snapshot.connectionState == ConnectionState.waiting) {
113       return const Center(child: CircularProgressIndicator());
114     }
115
116     if (snapshot.hasError) {
117       return const Center(child: Text('⚠️ لم يتم تحميل البودكاست'));
118     }
119
120     final podcasts = snapshot.data!.docs;
121
122     if (podcasts.isEmpty) {
123       return const Center(child: Text('لا توجد حلقات بودكاست حالياً'));
124     }
125
126     return Column(
127       children: podcasts.map((doc) {
128         final data = doc.data() as Map<String, dynamic>;
129         final title = data['podcastTitle'] ?? '';
130         final audioUrl = data['audioUrl'] ?? '';
131
132         return Card(
133           margin: const EdgeInsets.symmetric(horizontal: 16, vertical: 16),
134           shape: RoundedRectangleBorder(borderRadius: BorderRadius.circular(16)),
135           color: Colors.white,
136           elevation: 4,
137           child: ListTile(
138             title: Text(title),
139             subtitle: Text('currentlyPlaying == title'),
140             trailing: Column(
141               crossAxisAlignment: CrossAxisAlignment.start,
142               children: [
143                 StreamBuilder<Duration>(
144                   stream: _audioPlayer.positionStream,
145                   builder: (context, snapshot) {
146                     final position = snapshot.data ?? Duration.zero;
147
148                     return Column(
149                       crossAxisAlignment: CrossAxisAlignment.start,
150                       children: [
151                         Text(
152                           position.inSeconds.toString(),
153                           style: TextStyle(fontWeight: FontWeight.bold),
154                         ),
155                         Text(
156                           'Duration: zero',
157                           style: TextStyle(fontWeight: FontWeight.bold),
158                         ),
159                       ],
160                     );
161                   },
162                 ), // IconButton
163               ],
164             ), // Row
165           ), // Column
166           : null,
167           trailing: IconButton(
168             icon: const Icon(Icons.play_arrow),
169             onPressed: () => _playPodcast(audioUrl, title),
170           ), // IconButton
171         ), // ListTile
172       ), // Card
173     );
174   },
175 }
```

Figure 41: streamBuilder(podcasts)

4. `_buildHeader(context)`

- **What it does**

Constructs the top visual section of the Podcast screen. Includes:

- A stylized background
- A back button
- A large podcast icon
- Arabic text titles: "بودكاست" and "وسّع مداركك"

- **Why it's important:**

- Enhances user engagement with culturally relevant and motivational visuals, and reinforces the purpose of the screen.

```
Widget _buildHeader(BuildContext context) {  
  return SizedBox(  
    height: 250,  
    child: Stack(  
      children: [  
        Positioned(  
          child: Image.asset(  
            'assets/images/Bg.png',  
            fit: BoxFit.fill,  
            width: double.infinity,  
          ), // Image.asset  
        ), // Positioned  
        Positioned(  
          top: 30,  
          right: 20,  
          child: GestureDetector(  
            onTap: () {  
              Navigator.pop(context);  
            },  
            child: Image.asset(  
              "assets/images/backOrange.png",  
              width: 40,  
              height: 40,  
            ), // Image.asset  
          ), // GestureDetector  
        ), // Positioned  
        Positioned(  
          left: 20,  
          right: 180,  
          top: 70,  
          child: Row(  
            crossAxisAlignment: CrossAxisAlignment.center,  
            mainAxisAlignment: MainAxisAlignment.spaceBetween,  
            children: [  
              Expanded(  
                child: Column(  
                  crossAxisAlignment: CrossAxisAlignment.start,  
                  mainAxisAlignment: MainAxisAlignment.min,  
                  children: const [  
                    Text(  

```

Figure 42: `buildHeader(podcasts)`

5.3.1.5.3 Access Educational Content (Articles)

- **Purpose:**
 - To provide post-rehabilitation patients with motivational and educational content through a dedicated screen that displays quotes and guidance retrieved from Firebase Firestore in real time.

- **How It Works:**
 - Patients can access this feature via the navigation bar by tapping on the "Media" icon. Once inside the `QuotesScreen`, the app displays a stylized interface with a header, an educational image, and a list of content cards pulled from Firestore's `EducationalContent` collection.
 - Each content card includes:
 - A **title** (e.g., a theme or topic of the quote)
 - A **description** (providing detailed educational or motivational text)
 - The screen is fully localized in Arabic and visually designed with rounded corners and a warm color palette to enhance readability and comfort.
 - All quote updates are shown **live**, using `StreamBuilder`, so any changes made in Firestore are immediately reflected in the UI without needing to refresh.

- **Supporting Methods and UI Behavior:**
 1. **`_buildHeader(BuildContext context)`**
 - **What it does:**
 - Constructs the top section of the screen with decorative elements and text.
 - **Includes:**
 - A background image (`Bg.png`)
 - A back button (custom image `backOrange.png`)
 - A title ("معلومات نصية") and subtitle ("وسّع مداركك")
 - A thematic image (`QuoteOrange.png`)

- **Why it's important:**
 - Provides context and emotional resonance through design, reinforcing the purpose of the content — mental and emotional support.

```
Widget _buildHeader(BuildContext context) {
  return SizedBox(
    height: 250, // Adjust height as needed
    child: Stack(
      children: [
        Positioned(
          child: Image.asset(
            'assets/images/Bg.png',
            fit: BoxFit.fill,
            width: double.infinity,
          ), // Image.asset
        ), // Positioned

        // Custom Back Button (Top Right)
        Positioned(
          top: 30,
          right: 20,
          child: GestureDetector(
            onTap: () {
              Navigator.pop(context);
            },
            child: Image.asset(
              'assets/images/backOrange.png',
              width: 40,
              height: 40,
            ), // Image.asset
          ), // GestureDetector
        ), // Positioned

        // Properly Aligned Row for Text & Image
        Positioned(
          left: 20,
          right: 70,
          top: 70, // Adjusts row position properly
          child: Row(
            crossAxisAlignment: CrossAxisAlignment.center,
            mainAxisAlignment: MainAxisAlignment.spaceBetween, //
            children: [
              // Text Section (Left Side)
```

Figure 43: `_buildHeader(text)`

2. `StreamBuilder<QuerySnapshot>`

- **What it does:**
 - Fetches and listens to live updates from the Firestore `EducationalContent` collection
 - **Handles 4 UI States:**
 - **Loading** – shows a spinner while waiting
 - **No Data** – displays a message "لا توجد معلومات حالياً"
 - **Error** – handled implicitly (not shown explicitly, but could be added)
 - **Success** – maps each Firestore document to a quote card

- **Why it's important:**
 - o Ensures the content is always fresh, personalized, and dynamically maintained by administrators without app updates.

```

71 child: StreamBuilder<QuerySnapshot>(< //live to the Firestore collection
72 stream: FirebaseFirestore.instance.collection('EducationalContent').snapshots(),
73 builder: (context, snapshot) {
74   if (snapshot.connectionState == ConnectionState.waiting) {
75     return const Center(child: CircularProgressIndicator());
76   }
77   if (!snapshot.hasData || snapshot.data!.docs.isEmpty) {
78     return const Center(child: Text('لا يوجد محتوى تعليمي'), style: TextStyle(fontSize: 18));
79   }
80   final documents = snapshot.data!.docs;
81   return ListView.builder(
82     padding: const EdgeInsets.all(16),
83     itemCount: documents.length,
84     itemBuilder: (context, index) {
85       final data = documents[index].data() as Map<String, dynamic>;
86       final title = data['title'] ?? 'عنوان تعليمي';
87       final description = data['description'] ?? 'وصف تعليمي';
88       return Container(
89         margin: const EdgeInsets.only(bottom: 16),
90         padding: const EdgeInsets.all(16),
91         decoration: BoxDecoration(
92           color: Colors.white,
93           borderRadius: BorderRadius.circular(12),
94           boxShadow: [
95             BoxShadow(color: Colors.black12, blurRadius: 6, spreadRadius: 3),
96           ],
97       ), // BoxDecoration
98       child: Column(
99         crossAxisAlignment: CrossAxisAlignment.end,
100        children: [
101          Text(
102            title,
103            style: const TextStyle(
104              fontSize: 18,
105              fontWeight: FontWeight.bold,
106              color: Color(0xFF632800),
107            ), // TextStyle
108            textAlign: TextAlign.right,
109          ), // Text
110          const SizedBox(height: 8),
111          Text(
112            description,
113            style: const TextStyle(fontSize: 14, color: Colors.black87),
114            textAlign: TextAlign.right,
115          ), // Text
116        ], // Column
117      ), // Container
118    ], // ListView.builder
119  ), // StreamBuilder
120  Expanded
121  ),
122  ),

```

Figure 44: streamBuilder(text)

5.3.1.6 Religious Supportive Messages Tailored to Patients' Emotions and Preferred Formats: Quotes, Affirmations, Advice, or Motivation

- **Purpose:**
 - o To provide post-rehabilitation patients with emotionally tailored motivational quotes that support their mental well-being and recovery. This feature engages users through a personalized emotional check-in and delivers content that aligns with their current state.
- **How It Works:**
 - Patients interact with a horizontal row of emojis representing five different moods (e.g., happiness, sadness, anxiety). When a mood is selected:
 - a. The index of the selected mood is stored in `_selectedMood`.

- b. **A modal loading spinner is shown**, indicating the system is retrieving and generating content.
- c. **Firestore Authentication** is used to get the current user's ID.
- d. **The app queries the user's Quotes subcollection** from Firestore, collecting any saved QuoteType values (e.g., "تشجيع", "نصائح").
- e. If no types are found, it defaults to a general type: "نماذج".
- f. The app calls `ChatGPTService.generateQuote()` with:
 - a. The selected emotion (from a predefined `emotions[]` list)
 - b. The list of quote types
- g. **An AI-generated motivational quote is returned** and shown in a custom-styled `AlertDialog` with Arabic typography, colors, and formatting.

- **Supporting Methods and UI Behavior:**

1. **`_onMoodSelected(index)`**

- **What it does:**
 - Handles tap interaction on a mood icon.
- **Why it's important:**
 - Orchestrates state updates, shows loading feedback, retrieves user preferences, and communicates with the AI service.

```

291 // tapping a mood
292 void _onMoodSelected(int index) async {
293   try {
294     setState(() {
295       _selectedMood = index; // UI updated to show which mood is selected
296     });
297
298     // loading spinner when fetching data
299     showDialog(
300       context: context,
301       barrierDismissible: false, // no dismissing the dialog by tapping outside it.
302       builder: (context) => const AlertDialog(
303         backgroundColor: Color(0xFFFFE0E0),
304         content: SizedBox(height: 80, child: Center(child: CircularProgressIndicator())),
305       ), // AlertDialog
306     );
307
308     // fetch the user id from DB
309     final uid = FirebaseAuth.instance.currentUser?.uid;
310
311     // store the selected motivational quote types in a list
312     List<String> quoteTypes = [];
313     if (uid != null) {
314       // Fetch all documents inside the Quotes
315       final snapshot = await FirebaseFirestore.instance
316         .collection("Patients")
317         .doc(uid)
318         .collection("Quotes")
319         .get();
320
321       for (var doc in snapshot.docs) { // Loop through each document
322         final data = doc.data();
323         if (data["QuoteType"] != null) {
324           quoteTypes.add(data["QuoteType"]); // Add QuoteType value to quoteTypes list.
325         }
326       }
327     }
328
329     if (quoteTypes.isEmpty) {
330       quoteTypes = ["نماذج"]; // make if a default is list is empty make it advice
331     }
332
333     // Call ChatGPT to generate a personalized quote based on mood and q.
334     final quote = await ChatGPTService.generateQuote(emotions[index], quoteTypes);
335
336     Navigator.pop(context); // Close the loading spinner
337
338     showDialog(
339       context: context,
340       barrierDismissible: false,
341       builder: (context) => AlertDialog(
342         backgroundColor: const Color(0xFFFFE0E0),
343         contentPadding: const EdgeInsets.fromLTRB(20, 20, 20, 30),
344         content: Stack(
345           children: [
346             Positioned(
347               top: 0,
348               right: 0,
349               child: IconButton(
350                 icon: const Icon(Icons.close, color: Colors.brown),
351                 onPressed: () => Navigator.pop(context),
352               ), // IconButton
353             ), // Positioned
354             Column(
355               mainAxisAlignment: MainAxisAlignment.min,
356               children: [
357                 const SizedBox(height: 40),
358                 Text(
359                   quote,
360                   textAlign: TextAlign.center,
361                   style: const TextStyle(
362                     fontSize: 18,
363                     color: Colors.brown,
364                     fontWeight: FontWeight.w800,
365                     fontFamily: 'Inter',
366                     height: 1.5,
367                   ), // TextStyle
368                 ), // Text
369               ], // Column
370             ), // Stack
371           ), // AlertDialog
372     );
  
```

Figure 45: `onMoodSelected()`

2. `_buildMoodSelector()`

- **What it does:**
 - Builds the UI component for mood selection using animated containers
- **Why it's important:**
 - Provides an intuitive and visually engaging way for users to express how they feel.

```
// building the UI for selecting the mood
Widget _buildMoodSelector() {
  return Align(
    alignment: Alignment.centerRight, // arabic layout
    child: Column(
      crossAxisAlignment: CrossAxisAlignment.start,
      children: [
        const Text(
          'كيف تشعر اليوم؟',
          style: TextStyle(fontSize: 18, fontWeight: FontWeight.bold),
        ), // Text
        SingleChildScrollView(
          scrollDirection: Axis.horizontal, // scrolling mood bar horizontally
          child: Row(
            children: List.generate(5, (index) { // Create 5 mood icons based on emotions
              return GestureDetector(
                onTap: () => _onMoodSelected(index), // tapping the mood
                child: AnimatedContainer(
                  duration: const Duration(milliseconds: 300),
                  margin: EdgeInsets.symmetric(horizontal: 4, vertical: _selectedMood == index ? 0 : 10), // ad
                  width: _selectedMood == index ? 80 : 70,
                  height: _selectedMood == index ? 80 : 70,
                  child: Stack(
                    alignment: Alignment.center,
                    children: [
                      if (_selectedMood == index)
                        Container(
                          width: 65,
                          height: 65,
                          decoration: BoxDecoration(
                            shape: BoxShape.circle,
                            boxShadow: [
                              BoxShadow(
                                color: Colors.yellow.withAlpha(153),
                                blurRadius: 15,
                                spreadRadius: 5,
                              ), // BoxShadow
                            ],
                          ), // BoxDecoration
                        ), // Container
                      Image.asset(
```

Figure 46: `_buildMoodSelector()`

3. AI Integration: ChatGPTService.generateQuote()

```
// Call ChatGPT to generate a personalized quote based on mood and quote types
final quote = await ChatGPTService.generateQuote(emotions[index], quoteTypes);
```

Figure 47: ChatGPTService.generateQuote()

4. Class: ChatGPTService

- What it does:

- Responsible for securely sending a POST request to an external AI model (GPT-4o) via HTTP and receiving a personalized Arabic motivational quote.

```
// returns asynchronous it waits for the AI to answer
static Future<String> generateQuote(String prompt, List<String> quoteTypes) async {
```

Figure 48: ChatGPTService

○ Request Details:

○ API URL:

<https://models.inference.ai.azure.com/chat/completions>

```
static const String apiUrl = "https://models.inference.ai.azure.com/chat/completions"; // url for sending the request
static const String apiKey = "ghp_sAEi7ZRzDAPx0AeRgJXzp96hTkgH51TCyPz"; // authorization token to access the AI model
```

Figure 49: Request Details I

- Model: gpt-4o
- Method: POST
- Headers:
 - Content-Type: application/json
 - Authorization: Bearer [API_KEY]

```
final response = await client.post(
  uri, // Authorization header
  headers: {
    "Content-Type": "application/json", // tell server request body is json format
    "Authorization": "Bearer $apiKey",
  },
```

Figure 50: Request Details2

- **Request Body Includes:**

- A warm and supportive system prompt instructing the AI to respond in a friendly Arabic tone.
- A user prompt that includes:
 - The selected **mood**
 - The preferred **quote types** (joined in Arabic using "،")

```
body: jsonEncode({ // msg converted from dart to json
  "messages": [
    {
      "role": "system",
      "content":
        "أنت صديق داعم ومحظوظ بتحدثك بأشياء شخصية وذاتية بالثقافة العربية الفصحى،  

        "لا تبتعد أبداً عن محادثات مثل (الضحك، التسلية،...)، فقط تحدث ببنية واضحة وتحدث إلى صديق يثق بك البشري،  

        "اجعل المحادثة متداولة تشتمل قصصك مرتبطة بواقعك،  

        "ثم مع التخييل تكتب قصصك ببنية الثقافة وتعد الاختصاص المطلوب (مثل: نصيحة، تذكيرات،...)".
    },
    {
      "role": "user", // users specific emotional mood and type of goufe
      "content":
        " - علمك تأنيق تعاطفه بشكل صادق وأصيل اختصاً ببنية متداولة لجانك بالثقافة العربية الفصحى. $typeString ويحد نوع الاختصاصات التالية: $prompt تلمي يحد بـ "
    }
  ],
  "model": "gpt-4o",
  "temperature": 1, // Creativity level
  "max_tokens": 4096, // maximum length of the response
  "top_p": 1 // considers all possible next word
}),
```

Figure 51: Request Body

- **Response Handling:**

- The quote is extracted from the `choices[0].message.content` field.
- UTF-8 decoding ensures proper Arabic character rendering.
- Errors are caught and returned as messages.

- **Why it's important:**

- Allows emotionally contextualized, high-quality quotes to be delivered in real-time, without requiring static or hardcoded content in

the app. This elevates the user experience and provides deeper psychological support.

5. Error Handling

- **What it does:**

- If any error occurs during data fetching or AI response generation, the spinner is dismissed and a fallback message is shown in the debug log.

- **Why it's important:**

- Ensures a smooth experience without app crashes, while enabling developers to track issues easily.

```
} catch (e) {  
  return "Error: $e";  
}
```

Figure 52: Error Handling

5.3.2 Firebase

The Firebase platform allows us to efficiently authenticate users, manage and sync data in real time, and improve the overall user experience. Using Firebase Authentication, we can securely control user access through email and password login. Cloud Firestore provides a scalable and real-time NoSQL database solution for effective data handling and retrieval. We began by creating the OBOUR DB in Firebase, then proceeded to implement the Authentication service, enabling users to sign in with their email and password through Firebase's built-in method. **Figure 1** shows users in our application

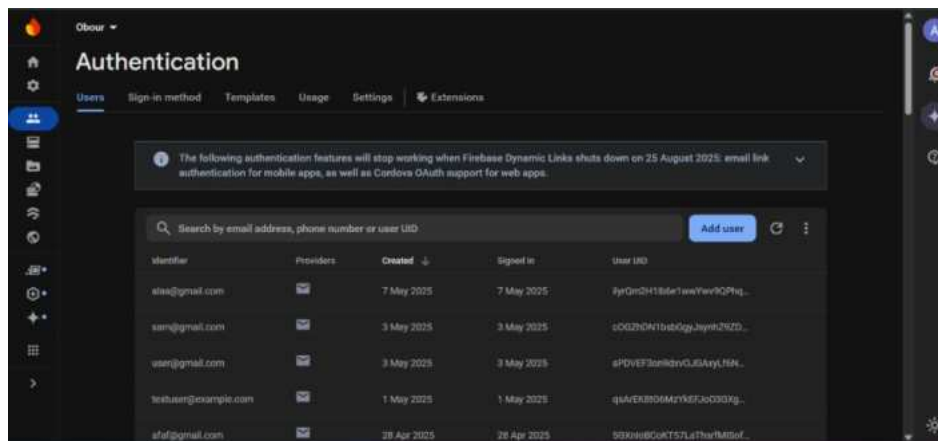


Figure 53: Firebase Authentication

Sing-up prosses and Patient collection:

During the sign-up process, when the user fills out the registration form and clicks the "continue" button, the app first validates all input fields using the `_formKey` and ensures there are no password-related errors. If all inputs are valid, the `_registerUser()` method is executed.

Inside this method, the user's input is retrieved and trimmed for clean formatting, including their name, email, password, phone number, and birth date. The app then attempts to create a new user account using Firebase Authentication via the `createUserWithEmailAndPassword` method, passing in the email and password.

If the account is successfully created, the app retrieves the newly created user's unique ID (`uid`) and stores the user's additional details—such as name, phone number, birth date, and registration timestamp—in a Firestore collection named 'Patient'.

After storing the user data, the app navigates the user to the `FavoriteActivitiesScreen` for the next step in the onboarding process.

However, if any error occurs during this process—for example, if the email is already in use—the app catches the exception and displays an error message to the user via a **SnackBar**, helping them understand what went wrong and prompting them to correct their input.

Figure 54: illustrates this sign-up process.

Figure 55: displays the structure of the **Patient** collection in Firestore, showing how each user's information is stored under their unique ID.

```
void _registerUser() async {
  if (_formKey.currentState!.validate() && _passwordError == null) {
    try {
      String name = _nameController.text.trim();
      String email = _emailController.text.trim();
      String password = _passwordController.text.trim();
      String phone = _phoneController.text.trim();
      String birthDate = _birthdateController.text.trim();

      UserCredential userCredential = await FirebaseAuth.instance
        .createUserWithEmailAndPassword(email: email, password: password);

      String uid = userCredential.user!.uid;

      await FirebaseFirestore.instance.collection('Patient').doc(uid).set({
        'name': name,
        'email': email,
        'phone': phone,
        'password': password,
        'birthdate': birthDate,
        'createdAt': FieldValue.serverTimestamp(),
      });

      Navigator.push(
        context,
        MaterialPageRoute(builder: (context) => FavoriteActivitiesScreen()),
      );
    } catch (error) {
      ScaffoldMessenger.of(context).showSnackBar(
        SnackBar(content: Text("Error: ${error.toString()}")),
      );
    }
  }
}
```

Figure 54: signup code

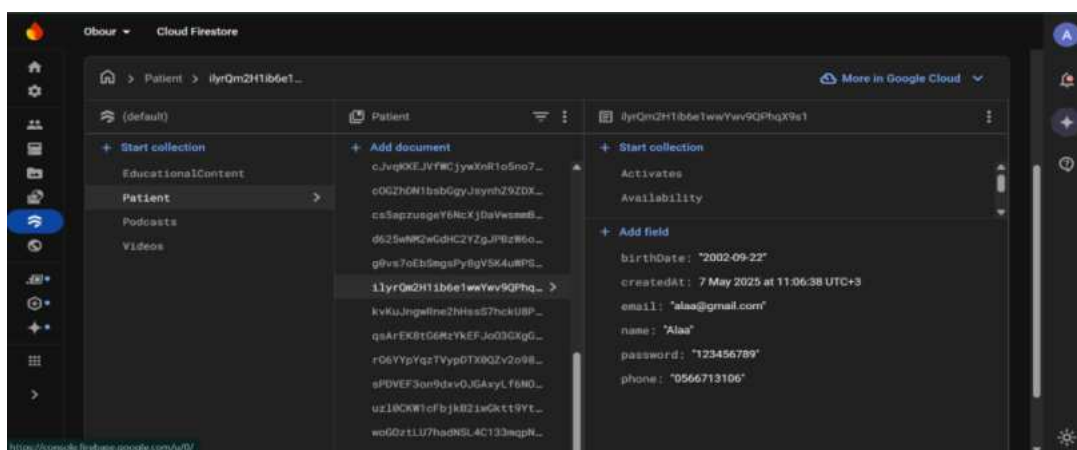


Figure 55: Firestore Patient collection

Activates collection:

This code snippet is responsible for creating the **Activates** subcollection and saving the user's selected activities. When a user selects one or more activities and submits them, the application

checks whether each selected activity already exists in Firestore under the path `Patient/{uid}/Activates`.

If an activity does not already exist, it is added as a new document to the `Activates` subcollection. Firestore automatically creates this subcollection when the first document is added, eliminating the need for manual setup. This process ensures that all selected activities are properly stored, easily retrieved, and remain associated with the correct user account.

Figure 56: Code snippet that creates the `Activates` collection and saves new user-selected activities to Firestore.

Figure 57: displays the structure of the `Activates` subcollection in Firestore, showing how each user's selected activities are stored as documents under their unique ID in the `Patient` collection. This collection is automatically created the moment the first activity is added, ensuring that the activities are properly stored and organized.

```
// إضافة الأنشطة الجديدة إن لم تكن موجودة
for (var entry in _selectedActivities.entries) {
  if (entry.value) {
    final querySnapshot = await activitiesRef
      .where('ActivityName', isEqualTo: entry.key)
      .limit(1)
      .get();

    if (querySnapshot.docs.isEmpty) {
      await activitiesRef.add({
        'ActivityName': entry.key,
      });
      print("تم إضافة النشاط الجديد: ✅ ${entry.key}");
    } else {
      print("النشاط موجود مسبقاً: ❌ ${entry.key}");
    }
  }
}
```

Figure 56: Creation of `Activates` collection code

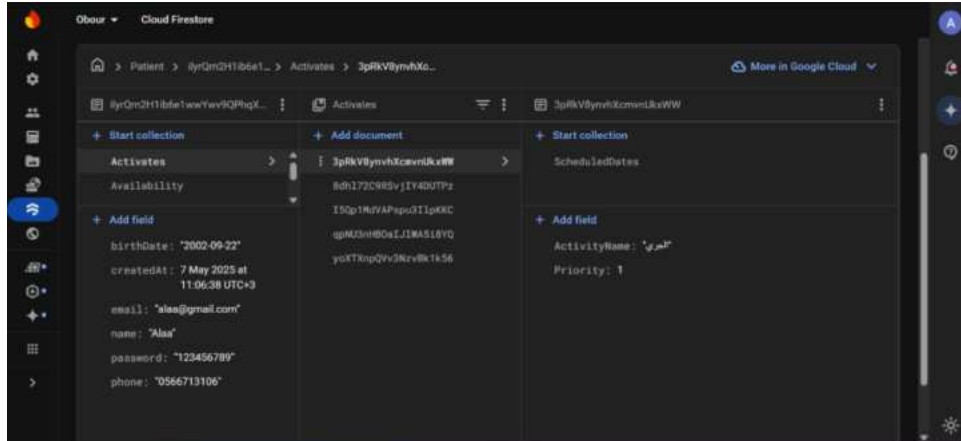


Figure 57: Firestore Activates collection

The **ScheduledDates** collection is created under each activity in the **Activates** subcollection. For each scheduled activity, a document is added to the **ScheduledDates** collection containing important information such as the scheduled date, start time, end time, and a flag indicating whether the activity is completed. The collection ensures that each scheduled instance of an activity is stored separately, with the possibility of having multiple scheduled dates for the same activity. If there is a scheduling conflict, the code will reschedule the activity to an available slot, updating the **ScheduledDates** collection accordingly. This allows for efficient tracking and management of all scheduled activities.

Figure 58 shows how a new document is created in the **ScheduledDates** collection for an activity. Each scheduled date contains the start time, end time, and a completion status.

Figure 59: demonstrates how an existing scheduled date is updated when there is a conflict with another scheduled activity. The start time, end time, and scheduled date are modified to resolve the conflict.

Figure 60: shows the structure of the **ScheduledDates** collection in Firestore, where each scheduled activity is stored with its start and end times, along with additional metadata.

```
var scheduledDocRef = await activityRef.collection('ScheduledDates').add({
  'scheduledDate': Timestamp.fromDate(
    DateTime(date.year, date.month, date.day, int.parse(slot['start']!.split(":")[0]), int.parse(slot['start']!.split(":")[1])),
  ), // Timestamp.fromDate
  'startTime': slot['start'],
  'endTime': slot['end'],
  'isCompleted': false,
});
```

Figure 58: Adding a New Scheduled Date

```

await firestore
  .collection('Patient')
  .doc(userId)
  .collection('Activates')
  .doc(activityId)
  .collection('ScheduledDates')
  .doc(conflictActivity['scheduledDateId'])
  .update({
    'scheduledDate': Timestamp.fromDate(
      DateTime(date.year, date.month, date.day, int.parse(slot['start']!.split(':')[0]), int.parse(slot['start']!.split(':')[1])),
    ), // Timestamp.fromDate
    'startTime': slot['start'],
    'endTime': slot['end'],
  });

```

Figure 59: Updating a Scheduled Date

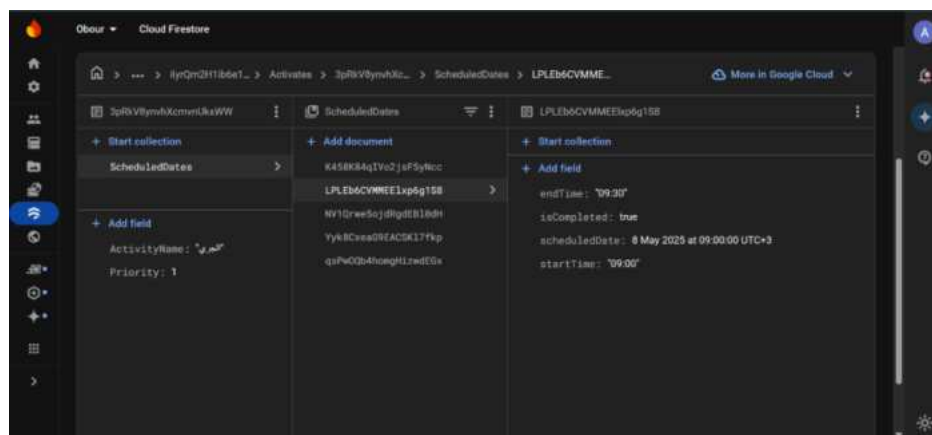


Figure 60: Firestore ScheduledDates collection

Availability collection:

saveAvailabilityToFirestore() this method handles saving the user's availability schedule to Firestore. It iterates through each selected day from the `_selectedDays` map. For each day that the user has marked as available and has set a start and end time, the method formats the times to a HH:mm format. Then, it saves the start and end times for each day to the **Availability** subcollection under the user's unique document in the **Patient** collection. Each day's availability is stored as a separate document in Firestore, using the day of the week as the document ID.

Figure 61: shows the code responsible for saving the user's availability schedule to Firestore. This method iterates through the selected days and saves the formatted start and end times for each day as documents under the **Availability** subcollection.

Figure 62: displays the structure of the **Availability** subcollection in Firestore, where each document represents the availability for a specific day, containing the start and end times of the user's availability.


```

Future<void> saveAvailabilityToFirestore() async {
  final user = FirebaseAuth.instance.currentUser;
  if (user == null) return;

  final userId = user.uid;
  final availabilityRef = FirebaseFirestore.instance
    .collection('Patient')
    .doc(userId)
    .collection('Availability');

  for (var day in _selectedDays.keys) {
    if (_selectedDays[day]! && _startTime.containsKey(day) && _endTime.containsKey(day)) {
      final start = _startTime[day]!;
      final end = _endTime[day]!;

      final startFormatted = "${start.hour.toString().padLeft(2, '0')}:${start.minute.toString().padLeft(2, '0')}";
      final endFormatted = "${end.hour.toString().padLeft(2, '0')}:${end.minute.toString().padLeft(2, '0')}";

      await availabilityRef.doc(day).set({
        'start': startFormatted,
        'end': endFormatted,
      });
    }
  }
}

```

Figure 61: Creation of Availability collection code

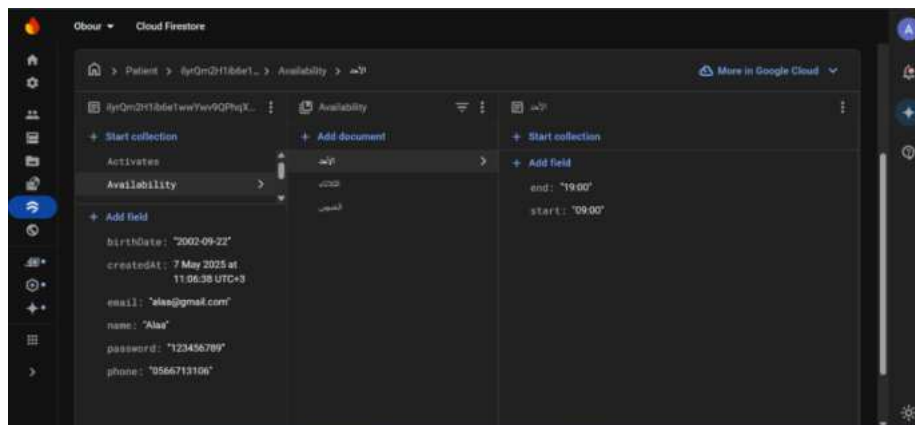


Figure 62: Firestore Availability collection

Quotes collection:

During the creation of the quotes collection, the `_saveChanges()` method manages the Quotes subcollection within the Patient collection in Firestore. It begins by checking the selected quote types from the phrases map, where each quote type is paired with a checked or unchecked status. For each quote type:

1. **If the quote type is checked** and it hasn't been previously saved (i.e., it doesn't exist in the previouslySaved list), the method adds a new document to the Quotes subcollection with the selected QuoteType.
2. **If the quote type is unchecked** and it was previously saved (i.e., it exists in Firestore), the method searches for the document in the Quotes subcollection and deletes it.

Firestore automatically creates the **Quotes** subcollection when the first document is added, so no manual setup is needed. This ensures that only the selected quote types are stored, and any unchecked quote types are removed from the Firestore database.

Figure 63: displays the code responsible for creating and managing the Quotes subcollection in Firestore. It handles adding new quote types when checked and removing them when unchecked, ensuring the Firestore database stays in sync with the user's selections.

Figure 64: illustrates the structure of the Quotes collection in Firestore, showing how each user's selected quote types are stored under their unique document in the Patient collection.

```
Future<void> _saveChanges() async {
  final user = FirebaseAuth.instance.currentUser;
  if (user == null) return;

  final quotesRef = FirebaseFirestore.instance
    .collection('Patient')
    .doc(user.uid)
    .collection('Quotes');

  for (String type in phrases.keys) {
    final isChecked = phrases[type]!;
    final wasSaved = previouslySaved.contains(type);

    if (isChecked && !wasSaved) {
      // Add new type
      await quotesRef.add({
        'QuoteType': type,
      });
    } else if (!isChecked && wasSaved) {
      // Delete unchecked
      final snapshot = await quotesRef.where('QuoteType', isEqualTo: type).get();
      for (var doc in snapshot.docs) {
        await doc.reference.delete();
      }
    }
  }
}
```

Figure 63: Creation of Quotes collection code

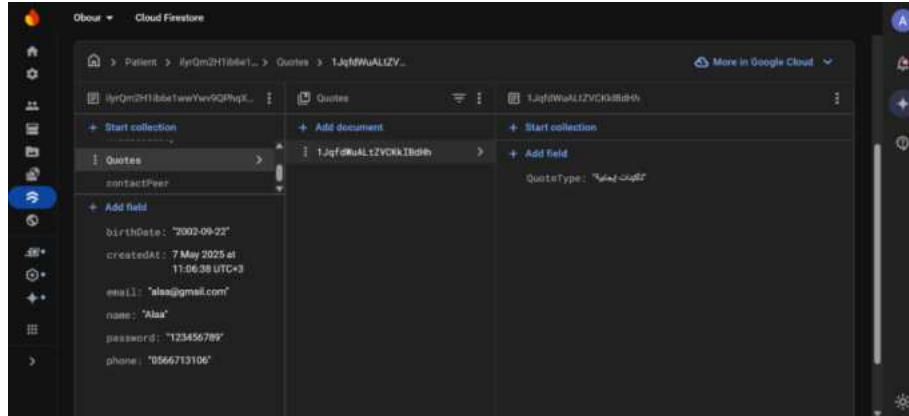


Figure 64: Firestore Quotes collection

ContactPeer collection:

This section of code manages the full lifecycle of the `contactPeer` subcollection in Firestore under each user in the `Patient` collection. During contact creation, the `_saveContactsToFirestore` method is used to add new peer contacts, storing their names and phone numbers. Once added, these contacts are retrieved using the `fetchContactsFromFirestore` method, which reads all documents from the `contactPeer` collection and updates the app's state to display them in the UI.

If a user wants to delete a contact, the `deleteContactFromFirestore` method is invoked. It identifies the specific document to remove based on its index in the list and deletes it from Firestore, then refreshes the contacts list by calling the `fetch` method again. For updates, the `onPressed` handler performs a Firestore `update` on a specific contact document using its ID, modifying the `PeerName` and `PeerPhoneNumber` fields based on user input.

Together, these operations ensure that peer contacts can be created, viewed, updated, and deleted seamlessly within the application, while keeping the Firestore database synchronized with the UI.

Figure 65: illustrates the code responsible for creating new contact entries in the `contactPeer` subcollection. Each contact includes a name and phone number and is saved under the current user's `Patient` document in Firestore.

Figure 66: shows the logic used to edit an existing contact. The contact's name and phone number are updated directly in Firestore using the document ID.

Figure 67: demonstrates the deletion process, where a selected contact is removed from the `contactPeer` subcollection, and the list is refreshed accordingly.

Figure 68: presents the method for fetching all saved contacts from Firestore. It retrieves the data, extracts document IDs, and updates the UI to display the contact list.

Figure 69: displays the structure of the `contactPeer` subcollection in Firestore, showing how peer contact information is stored for each user.

```
Future<void> _saveContactsToFirestore() async {
  final user = FirebaseAuth.instance.currentUser;
  if (user == null) return;

  final String userId = user.uid;
  final CollectionReference contactsRef = FirebaseFirestore.instance
    .collection('Patient')
    .doc(userId)
    .collection('contactPeer');

  for (var contact in _contacts) {
    await contactsRef.add({
      'PeerName': contact['name'],
      'PeerPhoneNumber': contact['phone'],
    });
  }
}
```

Figure 65: Creation of contactPeer collection

```
onPressed: () async {
  final user = FirebaseAuth.instance.currentUser;
  if (user == null) return;

  final docRef = FirebaseFirestore.instance
    .collection('Patient')
    .doc(user.uid)
    .collection('contactPeer')
    .doc(widget.contactId);

  await docRef.update({
    'PeerName': nameController.text.trim(),
    'PeerPhoneNumber': phoneController.text.trim(),
  });

  Navigator.pop(context, true); // return true to trigger refresh
},
child: const Text("حفظ", style: TextStyle(fontSize: 18, color: Colors.white)),
// ElevatedButton
```

Figure 66: Edit contactPeer from firestore

```

Future<void> deleteContactFromFirestore(int index) async {
    final user = FirebaseAuth.instance.currentUser;
    if (user == null) return;

    final docId = contactDocIds[index];

    await FirebaseFirestore.instance
        .collection('Patient')
        .doc(user.uid)
        .collection('contactPeer')
        .doc(docId)
        .delete();

    fetchContactsFromFirestore();
}

```

Figure 67: Delete contactPeer from firestore

```

Future<void> fetchContactsFromFirestore() async {
    final user = FirebaseAuth.instance.currentUser;
    if (user == null) return;

    final contactRef = FirebaseFirestore.instance
        .collection('Patient')
        .doc(user.uid)
        .collection('contactPeer');

    final querySnapshot = await contactRef.get();
    final fetchedContacts = querySnapshot.docs.map((doc) {
        final data = doc.data();
        return {
            "name": data["PeerName"] ?? "",
            "phone": data["PeerPhoneNumber"] ?? "",
        };
    }).toList();

    final ids = querySnapshot.docs.map((doc) => doc.id).toList();

    setState(() {
        contacts = fetchedContacts;
        contactDocIds = ids;
    });
}

```

Figure 68: Fetch contactPeer from firestore

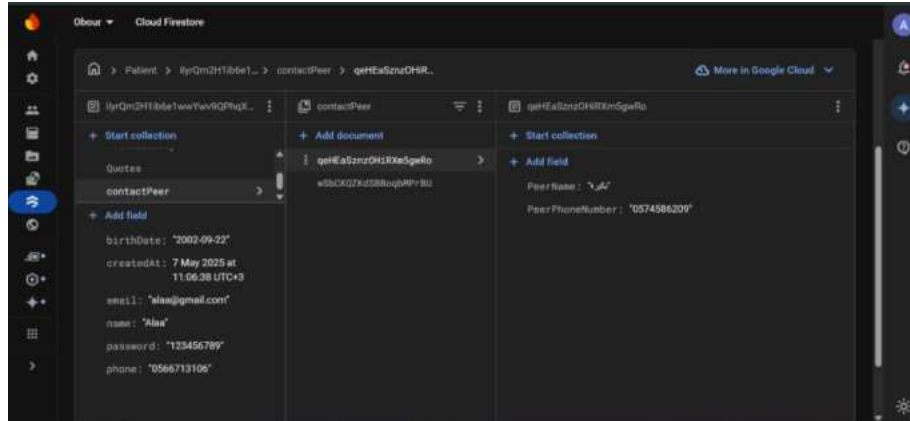


Figure 69: Firestore contactPeer collection

In our application, we utilized **Firebase** as a central platform to store, manage, and stream various types of media content including videos, educational text, and podcasts. We used **Cloud Firestore** to hold structured data for each media category and implemented real-time data rendering through **StreamBuilder**, allowing users to see updates instantly within the app.

For the **videos**, we created a Firestore collection called **Videos** where each document includes a video title and a YouTube URL. These URLs are processed using the **YoutubePlayer** plugin, enabling users to watch embedded YouTube content directly. **Figures 72 and 73** showcase both the Dart code for fetching and displaying video data and the Firestore **Videos** collection interface.

For **educational content**, a Firestore collection named **EducationalContent** stores entries as a title and its corresponding description. These are displayed in a stylized, scrollable view within the app, allowing users to easily read and benefit from Arabic educational content. **Figure 70** shows the code responsible for retrieving and displaying this content from Firestore, while **Figure 71** presents the structure of the Firestore collection.

When it comes to **podcasts**, we followed a slightly more advanced structure. We stored the actual audio files in **Firebase Storage** and linked them in Firestore under the **Podcasts** collection by saving each episode's title and its corresponding audio URL. The app then retrieves these URLs and streams the content using the **just_audio** package. Users are provided with intuitive playback controls including play, pause, seek, and stop features. **Figure 74 and 75** present both the code that handles podcast fetching and playback and a screenshot of the **Podcasts** Firestore collection. **Figure 76** shows the **Firebase Storage** interface.

```

child: StreamBuilder<QuerySnapshot>{
  stream: FirebaseFirestore.instance.collection('EducationalContent').snapshots(),
  builder: (context, snapshot) {
    if (snapshot.connectionState == ConnectionState.waiting) {
      return const Center(child: CircularProgressIndicator());
    }
    if (!snapshot.hasData || snapshot.data!.docs.isEmpty) {
      return const Center(child: Text('لا توجد معلومات حالياً', style: TextStyle(fontSize: 18)));
    }

    final documents = snapshot.data!.docs;

    return ListView.builder(
      padding: const EdgeInsets.all(16),
      itemCount: documents.length,
      itemBuilder: (context, index) {
        final data = documents[index].data() as Map<String, dynamic>;
        final title = data['title'] ?? 'بنتون عنبيل';
        final description = data['description'] ?? 'لا يوجد شرح';

```

Figure 70: Fetch EducationalContent from Firestore

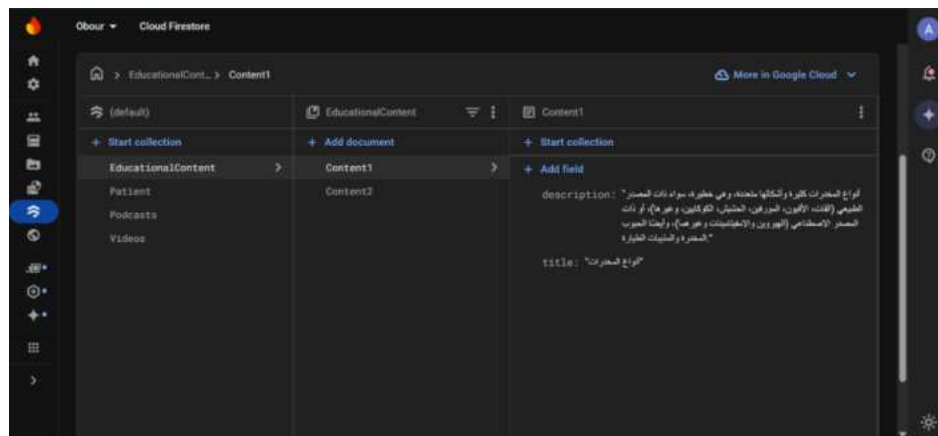


Figure 71: Firestore EducationalContent collection


```
// 🔍 StreamBuilder to fetch all videos
StreamBuilder<QuerySnapshot>{
  stream: FirebaseFirestore.instance.collection('Videos').snapshots(),
  builder: (context, snapshot) {
    if (snapshot.connectionState == ConnectionState.waiting) {
      return const Center(child: CircularProgressIndicator());
    }

    if (snapshot.hasError) {
      print("❌ Firestore error: ${snapshot.error}");
      return const Center(child: Text('⚠️ خطأ في تحميل الفيديوهات'));
    }

    if (!snapshot.hasData || snapshot.data!.docs.isEmpty) {
      return const Center(child: Text('لا توجد فيديوهات'));
    }

    final videos = snapshot.data!.docs;

    print("✅ Videos fetched: ${videos.length}");

    return Column(
      children: videos.map((video) {
        final videoData = video.data() as Map<String, dynamic>;
        final videoTitle = videoData['videoTitle'];
        final videoUrl = videoData['videoUrl'];
        final videoId = YoutubePlayer.convertUrlToId(videoUrl);

        if (videoId == null) {
          print("❌ Invalid video URL: $videoUrl");
          return const SizedBox(); // Skip if URL is not valid
        }
      })
    );
  }
}
```

Figure 72: Fetch Videos from Firestore

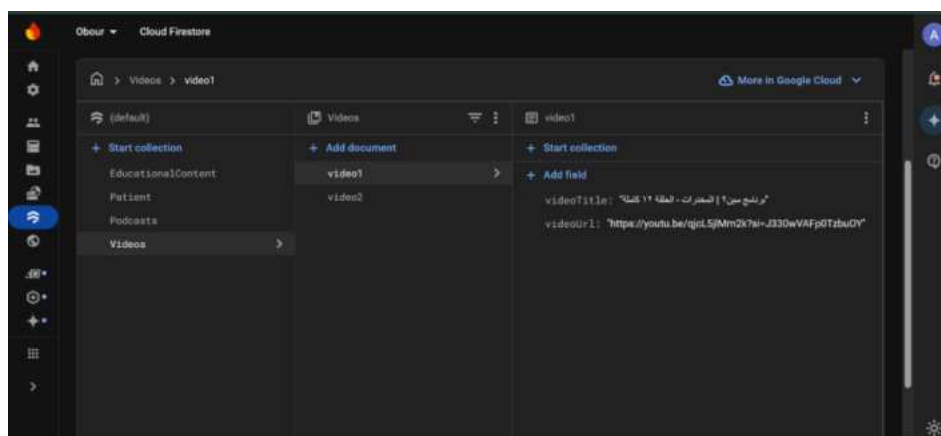


Figure 73: Firestore Videos collection


```

StreamBuilder<QuerySnapshot>(  
  stream: FirebaseFirestore.instance.collection('Podcasts').snapshots(),  
  builder: (context, snapshot) {  
    if (snapshot.connectionState == ConnectionState.waiting) {  
      return const Center(child: CircularProgressIndicator());  
    }  
  
    if (snapshot.hasError) {  
      return const Center(child: Text('⚠️ خطأ في تحميل البودكاست'));  
    }  
  
    final podcasts = snapshot.data!.docs;  
  
    if (podcasts.isEmpty) {  
      return const Center(child: Text('لا توجد حلقات بودكاست حالياً'));  
    }  
  
    return Column(  
      children: podcasts.map((doc) {  
        final data = doc.data() as Map<String, dynamic>;  
        final title = data['podcastTitle'] ?? '';  
        final audioUrl = data['audioUrl'] ?? '';  


```

Figure 74: Fetch Podcasts from Firestore

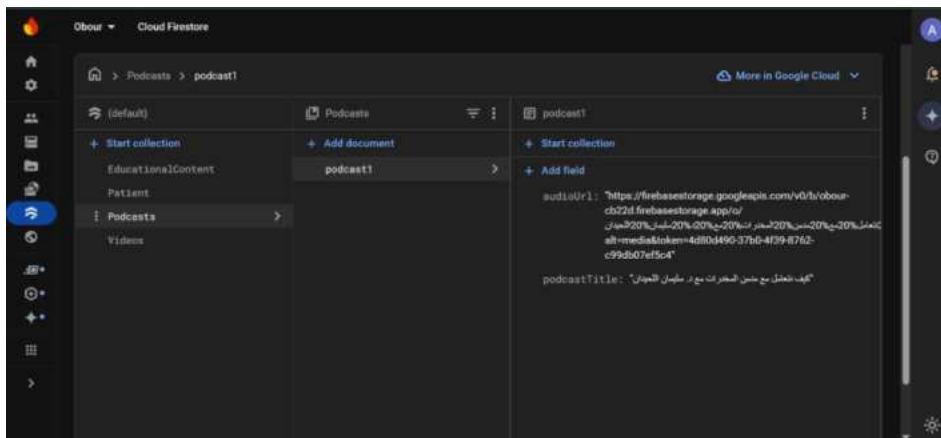


Figure 75: Firestore Podcasts collection

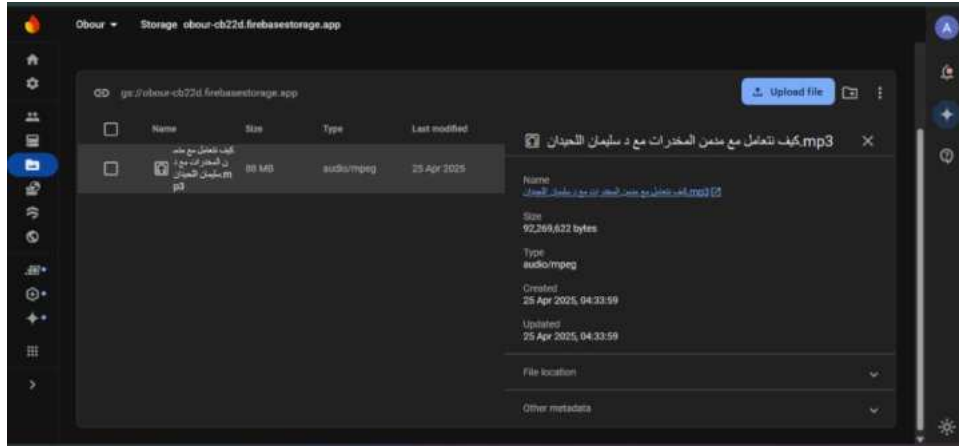


Figure 76: Firebase Storage

5.4 Code Debugging and Troubleshooting Issues

During the development of the Obour application, several technical challenges arose, primarily related to integrating Firebase with the Flutter framework. These issues required extensive debugging and troubleshooting to ensure seamless system functionality.

Firestore Connectivity Issues with Flutter

One of the most significant obstacles encountered was establishing a stable and functional connection between Firebase and the Flutter application. Despite initially following the official Firebase documentation, the connection repeatedly failed due to the following reasons:

1. **Incorrect Firebase SDK Configuration**

The Firebase configuration files—**google-services.json** for Android and **GoogleService-Info.plist** for iOS—were not properly placed in their respective directories within the project. This misplacement caused initialization errors, preventing Firebase from launching correctly within the application.

2. **Outdated Firebase SDK**

The project was originally using an older version of the Firebase SDK, which conflicted with the latest Flutter framework updates. This incompatibility led to dependency conflicts and build errors. Resolving this issue required manually updating the SDK and revising the related dependencies listed in the `pubspec.yaml` file.

Solutions Implemented

- Revisited and followed the Firebase setup instructions meticulously, ensuring that all configuration files were correctly placed and referenced in the project.
- Updated all Firebase-related dependencies to the most recent stable versions that align with the Flutter SDK, resolving compatibility issues and restoring functionality.

5.5 Packaging and Documentation

To ensure long-term maintainability and scalability, the Obour application was packaged following a clean and modular project structure. This organization supports easier navigation, future updates, and collaborative development.

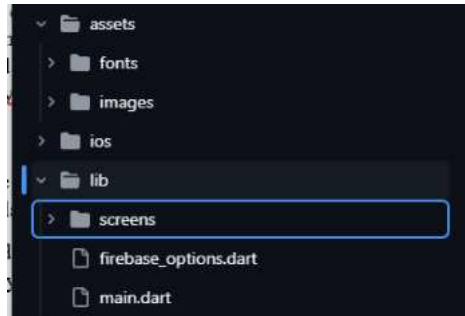


Figure 77: Main and Screens

This figure shows all the classes related to main :

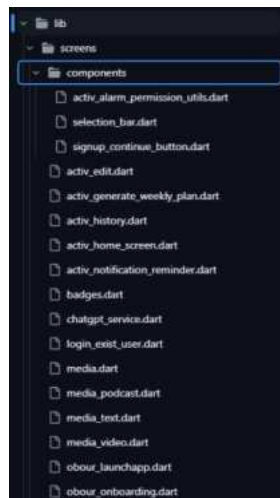


Figure 78: All Screens

This figure shows all the classes related to Screens

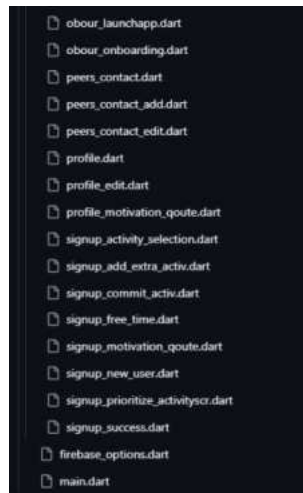


Figure 79: All Screens

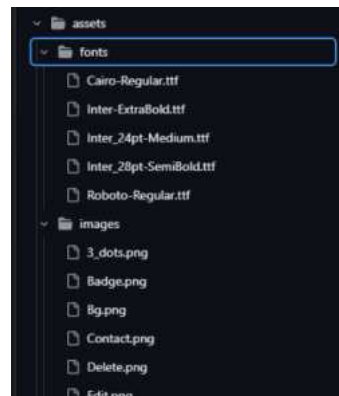


Figure 80: images and fonts

5.6 Conclusion

Completing the implementation of the Obour application was a major step in bringing the project to life. During this phase, the main features were developed and integrated, turning ideas and designs into a working system that can support users in their recovery journey.

While there were some challenges—especially when connecting Firebase with Flutter—these issues were solved through research, trial and error, and updating tools to ensure everything worked smoothly. The team took care to organize the codebase clearly, making it easier to manage and update in the future.

By the end of this phase, the Obour app had a solid structure, reliable functionality, and clear documentation, all of which help prepare it for future improvements and real-world use.

Chapter 6: Testing

6.1 Introduction

This chapter explores the **testing phase** of the OBOUR application. Section 6.2 focuses on **functional testing**, including unit testing, integration testing, and system testing. Section 6.3 addresses **non-functional testing**, such as **compatibility testing**, to ensure the application performs well across different devices and environments while Section 6.4 presents the **conclusion**.

6.2 Functional Testing

6.2.1 Unit testing

6.2.1.1 Signup test

This test suite verifies critical user authentication and input validation scenarios in the Flutter application using Firebase services. The tests are written with the help of the `flutter_test` and `mocktail` packages to simulate authentication and Firestore interactions without relying on real backend services.

The test class covers the following scenarios:

- **Successful Signup and Firestore Profile Saving**
This test validates the sign-up flow where a new user is registered via Firebase Authentication, and their profile is stored in Firestore. The test mocks `createUserWithEmailAndPassword()` to return a mock user credential and confirms that the `set()` method is called on the user document with the correct data structure.

```
test('Sign up and save user profile to Firestore', () async {
  const email = 'lolol@gmail.com';
  const password = '123456789';
  const name = 'lolol';
  const phone = '0566713106';
  const birthDate = '2002-5-5';
  print('👤 Starting sign up test...');

  // Step 1: FirebaseAuth signup
  final credential = await mockAuth.createUserWithEmailAndPassword(
    email: email,
    password: password,
  );
  final uid = credential.user!.uid;
  print('✅ FirebaseAuth.createUserWithEmailAndPassword called: UID = $uid');

  // Step 2: Firestore set user profile
  final userData = {
    'name': name,
    'email': email,
    'password': password,
    'phone': phone,
    'birthDate': birthDate,
    'createdAt': DateTime(2025, 4, 15, 21, 52, 54), // or DateTime.now()
  };
  // ... Firestore set call ...
});
```

Figure 81: Signup test

This ensures that the registration logic functions correctly and user details are saved as expected.

```
00:06 +4: C:/Users/baker/StudioProjects/Obour/test/auth_test.dart: Sign up and save user profile to Firestore
  Starting sign up test...
  ✓ FirebaseAuth.createUserWithEmailAndPassword called: UID = new-user-uid
  ✓ Firestore user profile saved.
  ✓ Sign up and Firestore user profile test passed.
```

Figure 82: signup test result

- **Signup Failure Due to Duplicate Email**

This test ensures robust error handling when attempting to register with an already-used email. The mock throws a FirebaseAuthException with the code email-already-in-use, and the test verifies that the exception is properly caught and handled. This helps maintain app stability by preventing crashes during registration.

```
test('✗ Signup fails when email already exists', () async {
  const email = 'duplicate@example.com';
  const password = '123456789';

  when(() => mockAuth.createUserWithEmailAndPassword(
    email: email,
    password: password,
  )).thenThrow(FirebaseAuthException(code: 'email-already-in-use'));

  try {
    await mockAuth.createUserWithEmailAndPassword(
      email: email,
      password: password,
    );
    fail('Expected FirebaseAuthException for email-already-in-use');
  } on FirebaseAuthException catch (e) {
    expect(e.code, 'email-already-in-use');
    print('✓ Detected email already in use during signup.');
  }
});
```

Figure 83:Signup test email

```
00:06 +5: C:/Users/baker/StudioProjects/Obour/test/auth_test.dart: ✗Signup fails when email already exists
  ✓ Detected email already in use during signup.
```

Figure 84: signup test email result

- **Login Failure with Incorrect Password**

This case simulates a login attempt using an incorrect password. The test mocks `signInWithEmailAndPassword()` to throw a `FirebaseAuthException` with the code `wrong-password`.

```
test('✗ Login fails with wrong password', () async {
  const email = 'user@example.com';
  const wrongPassword = 'wrongpass123';

  when(() => mockAuth.signInWithEmailAndPassword(
    email: email,
    password: wrongPassword,
  )).thenThrow(FirebaseAuthException(code: 'wrong-password'));

  try {
    await mockAuth.signInWithEmailAndPassword(
      email: email,
      password: wrongPassword,
    );
    fail('Expected FirebaseAuthException for wrong-password');
  } on FirebaseAuthException catch (e) {
    expect(e.code, 'wrong-password');
    print('✅ Detected wrong password during login.');
  }
});
```

Figure 85: login test

```
00:06 +6: C:/Users/baker/StudioProjects/Obour/test/auth_test.dart: ✗Login fails with wrong password
✅Detected wrong password during login.
```

Figure 86: login test result

Overall, this automated test file plays a key role in maintaining the reliability of the user onboarding process by verifying that both successful and erroneous states are correctly handled during sign-up and login procedures.

6.2.1.2 Activity test

This unit test file verifies core functionalities related to user activity selection, priority setting, and free-time availability within the app's Firestore-based data structure. It uses the `flutter_test` and `mocktail` packages to simulate Firebase Authentication and Cloud Firestore interactions in isolation, without connecting to real backend services.

The test suite covers the following main scenarios:

- **Selecting and Saving an Activity**

This test validates the ability to save a selected activity into the Firestore under the authenticated user's document. It constructs mock activity data with fields like activity name, date, and status, then confirms that the `set()` operation is correctly called. This ensures the reliability of the feature that lets users record their planned activities.

```
test('Select and save activity to Firestore', () async {
  final activityData = {
    'ActivityName': 'رياضة',
    'ActivityDate': '2025-05-01',
    'ActivityStatus': false,
  };

  when(() => mockActivityDoc.set(activityData)).thenAnswer((_) async {
    print('✅ Activity saved: $activityData');
  });

  await mockFirestore
    .collection('Patient')
    .doc('test-user-id')
    .collection('Activates')
    .doc('activity1')
    .set(activityData);

  verify(() => mockActivityDoc.set(activityData)).called(1);
});
```

Figure 87: Activity test

```
00:05 +0: C:/Users/baker/StudioProjects/Obour/test/activity_test.dart: Select and save ac
✅Activity saved: {ActivityName: رياضة, ActivityDate: 2025-05-01, ActivityStatus: false}
```

Figure 88: Activity test result

- **Setting Activity Priority**

To support personalized activity plans, this test checks whether a priority value (e.g., 1 for high priority) is successfully updated in the Firestore for a specific activity. The test mocks the `update()` method and verifies its invocation. This test ensures that user-defined priorities are stored correctly, which is essential for sorting and managing tasks.


```

test('Save priority for each activity', () async {
  final priorityData = {
    'Priority': 1,
  };

  when(() => mockActivityDoc.update(priorityData)).thenAnswer((_) async {
    print('✅ Priority updated: $priorityData');
  });

  await mockFirestore
    .collection('Patient')
    .doc('test-user-id')
    .collection('Activates')
    .doc('activity1')
    .update(priorityData);

  verify(() => mockActivityDoc.update(priorityData)).called(1);
});

```

Figure 90: Priority test

```

00:05 +1: C:/Users/baker/StudioProjects/Obour/test/activity_test.dart: Save priority for each activity
✅Priority updated: {Priority: 1}

```

Figure 89: Priority test result

- **Saving Free Time Availability**

The third test ensures that a user's availability (day and time range) is correctly saved into a subcollection called Availability under their document. It verifies that the set() operation is called with the correct time data for a given day (e.g., Sunday). This feature is essential for scheduling personalized activity plans based on user availability.

```

test('Save free time availability (day and time)', () async {
  final day = 'الأحد';
  final availabilityData = {
    'start': '07:00',
    'end': '20:00',
  };

  when(() => mockAvailabilityCollection.doc(day)).thenReturn(mockActivityDoc);
  when(() => mockActivityDoc.set(availabilityData)).thenAnswer((_) async {
    print('✅ Availability saved for $day: $availabilityData');
  });

  await mockFirestore
    .collection('Patient')
    .doc('test-user-id')
    .collection('Availability')
    .doc(day)
    .set(availabilityData);

  verify(() => mockActivityDoc.set(availabilityData)).called(1);
});

```

Figure 92: Saving Free Time Availability test

```

00:04 +2: C:/Users/baker/StudioProjects/Obour/test/activity
✅Availability saved for الأحد: {start: 07:00, end: 20:00}

```

Figure 91: Saving Free Time Availability test result

By thoroughly mocking and testing each key interaction with Firestore, this unit test file helps ensure that the core logic related to activity planning and scheduling behaves as expected, contributing to a stable and personalized user experience

6.2.1.3 Peer Contact test

This unit test validates the functionality for managing peer contact information in the Firestore database, allowing users to save and validate emergency or support contact details. The tests utilize the mocktail package to simulate Firebase Authentication and Firestore behavior, ensuring logic correctness without interacting with actual backend services.

The file includes the following key test cases:

- **Saving Contacts to Firestore**

This test confirms that valid contact data (peer name and phone number) is successfully saved under the contactPeer subcollection in the authenticated user's Firestore document. The test loops through a list of contacts and mocks the .add() method call for each one. It verifies that each call is executed exactly once. This functionality is critical for enabling users to store emergency or support contact information within the app.

```
test('Save contacts to Firestore', () async {
  final contacts = [
    {'name': 'Ali', 'phone': '0555555555'},
    {'name': 'Sara', 'phone': '966123456789'}
  ];
  // Mock .add() behavior
  for (var contact in contacts) {
    when(() => mockContactPeerCollection.add({
      'PeerName': contact['name'],
      'PeerPhoneNumber': contact['phone'],
    })).thenAnswer((_) async {
      print('✅ Firestore add called with: $contact');
      return MockDocumentReference();
    });
  }
  // Simulate saving each contact
  for (var contact in contacts) {
    await mockFirestore
      .collection('Patient')
      .doc('test-user-id')
      .collection('contactPeer')
      .add({
        'PeerName': contact['name'],
        'PeerPhoneNumber': contact['phone'],
      });
  }
}
```

Figure 94: Saving Contacts to Firestore test

```
00:05 ~1: C:/Users/baker/StudioProjects/Obour/test/add_contact_test.dart: Save contacts to Firestore
✅Firestore add called with: {name: Ali, phone: 0555555555}
✅Firestore add called with: {name: Sara, phone: 966123456789}
✅All contacts saved to Firestore successfully.
```

Figure 93: Saving Contacts to Firestore test result

- **Rejecting Invalid Phone Numbers**

This test simulates an attempt to add a contact with an improperly formatted phone number (e.g., containing letters). It uses a simple regular expression (`^\d{9,15}$`) to check for valid phone number patterns. If the format is invalid, the test ensures that the Firestore `.add()` method is never called by using `verifyNever()`. This validation helps maintain data integrity and prevents storage of unusable or malformed contact information.

```
test('Do not save contact with invalid phone number format', () async {
  final invalidContact = {'name': 'Invalid User', 'phone': '123abc'};

  bool isValidPhoneNumber(String phone) {
    final regex = RegExp(r'^\d{9,15}$');
    return regex.hasMatch(phone);
  }

  if (!isValidPhoneNumber(invalidContact['phone'])) {
    print('❌ Invalid phone number format: ${invalidContact['phone']}');
  } else {
    // This should NOT be executed in this test
    await mockFirestore
      .collection('Patient')
      .doc('test-user-id')
      .collection('contactPeer')
      .add({
        'PeerName': invalidContact['name'],
        'PeerPhoneNumber': invalidContact['phone'],
      });
  }

  verifyNever(() => mockContactPeerCollection.add(any()));
});
```

Figure 95: Add peer contact test

```
00:08 +1: Do not save contact with invalid phone number format
❌Invalid phone number format: 123abc
00:09 +2: All tests passed!
```

Figure 96: Add peer contact test result

These tests provide confidence in the app's ability to handle peer contact management reliably and securely. By enforcing format validation and confirming successful Firestore integration, the unit tests contribute to a robust and user-friendly safety net for individuals seeking support during recovery.

6.2.2 Integration testing

Integration testing verifies that multiple components of a Flutter app work together as expected. It simulates real user interactions with the UI—from launching the app to navigating between screens, filling forms, tapping buttons, and validating end-to-end behavior.

- **Purpose:**
 - Ensures different modules (UI, business logic, backend calls, etc.) work together cohesively.
 - Catches bugs that unit or widget tests may miss.
 - Simulates real user scenarios for quality assurance before deployment.
- **In Flutter:**
 - Integration tests run on real devices or emulators.
 - They use the `integration_test` package.
 - Consist of two parts: a test driver (`driver.dart`) and a test file (`*_test.dart`).

6.2.2.1 driver.dart

- **Purpose:**
 - Acts as the entry point that launches the app and connects it with the integration test framework.
- **How It Works:**
 - `integrationDriver()` sets up the environment for test execution.
 - It is required to pair with the actual test script so that Flutter can drive the app and report results.
- **Why it's important:**
 - Without this file, integration tests can't be executed using flutter drive, as it enables the test runner to connect with the running app instance.

```

1  import 'package:integration_test/integration_test_driver.dart';
2
3  >> Future<void> main() => integrationDriver();
4

```

Figure 97: driver class

6.2.2.2 signup_test.dart

- **Purpose:**
 - This script runs a full end-to-end test that simulates the **user signup process** and verifies successful navigation to the FavoriteActivitiesScreen.
- **Initialization:**
 - Ensures the test environment is fully connected to Flutter's widget system.

```
IntegrationTestWidgetsFlutterBinding.ensureInitialized();
```

Figure 98: signup_test (initilization)

- **Running the App:**
 - Runs the entire app inside a guarded zone to catch any uncaught runtime errors.

```

// Run the app in a guarded zone
runZonedGuarded(() {
  app.main();
}, (error, stackTrace) {
  print('🔴 Uncaught error: $error');
});

```

Figure 99: signup_test (Run)

- **UI Interactions:**
 - 1. Launch Screen**
 - Taps the "التالي" button.
 - Waits for UI to settle.
 - 2. Onboarding Screen**
 - Taps the "ابدأ الآن" button.
 - 3. Signup Form**
 - Finds fields using Key identifiers.
 - Enters:
 - Name
 - Email
 - Password
 - Phone
 - Selects a birthdate (via date picker).
 - Taps "متابعة" to submit the form.

```

1 import 'package:flutter_test/flutter_test.dart';
2 import 'package:integration_test/integration_test.dart';
3 import 'package:ghayr_1/main.dart' as app;
4 import 'package:flutter/material.dart';
5 import 'dart:async'; // This gives access to runZonedGuarded
6
7
8 void main() {
9   IntegrationTestWidgetsFlutterBinding.ensureInitialized();
10
11   testWidgets('User signs up successfully and navigates to FavoriteActivitiesScreen', (tester) {
12     print("🚀 Starting full app from main()");
13
14     // Run the app in a guarded zone
15     runZonedGuarded(() {
16       app.main();
17     }, (error, stackTrace) {
18       print("🔴 Uncaught error: $error");
19     });
20     await tester.pumpAndSettle(const Duration(seconds: 10));
21
22     // 🟡 Tap "التالي" in Launchpage screen
23     final launchButton = find.text("التالي");
24     expect(launchButton, findsOneWidget);
25     await tester.tap(launchButton);
26     await tester.pumpAndSettle(const Duration(seconds: 1));
27     print("🟢 التالي 🟢");
28
29     // 🟡 Tap "ابدأ الآن" in Onboarding screen
30     final onboardingButton = find.text("ابدأ الآن");
31     expect(onboardingButton, findsOneWidget);
32     await tester.tap(onboardingButton);
33     await tester.pumpAndSettle(const Duration(seconds: 1));
34     print("🟢 ابدأ الآن 🟢");
35
36     // 🟢 Now we're on SignupScreen
37     // Use Keys
38     final nameField = find.byKey(const Key('name_field'));
39     final emailField = find.byKey(const Key('email_field'));
40     final passwordField = find.byKey(const Key('password_field'));
41     final phoneField = find.byKey(const Key('phone_field'));
42     final birthdateField = find.byKey(const Key('birthdate_field'));
43
44     // Ensure each field exists before interacting
45     expect(nameField, findsOneWidget);
46     await tester.enterText(nameField, 'Test');
47
48     expect(emailField, findsOneWidget);
49     await tester.enterText(emailField, 'testuser@example.com');
50
51     expect(passwordField, findsOneWidget);
52     await tester.enterText(passwordField, '12345678');
53
54     expect(phoneField, findsOneWidget);
55     await tester.enterText(phoneField, '0533234547');
56
57     // Tap birthdate field to open date picker
58     await tester.tap(birthdateField);
59     await tester.pumpAndSettle(const Duration(seconds: 2));
60     await tester.tap(find.textContaining('2023')); // this only works if the date picker is open
61     await tester.pumpAndSettle(const Duration(seconds: 2));
62
63     // 🟡 Tap "متابعة" in sign new user screen
64     final continueButton3 = find.text("متابعة");
65     expect(continueButton3, findsOneWidget);
66     await tester.tap(continueButton3);
67     await tester.pumpAndSettle(const Duration(seconds: 10));
68     print("🟢 متابعة 🟢");
69
70   });
71 }

```

Figure 100: `signup_test` (UI interaction)

- **Why this test matters:**
 - Validates that critical navigation and form input logic works.
 - Verifies text inputs and buttons respond correctly.
 - Simulates real user behavior from app start to post-signup.

- **Best Practices Followed:**
 - expect(..., findOneWidget) ensures UI elements are present before interaction.
 - await tester.pumpAndSettle(...) is used for proper wait timing.
 - Print logs (🔴, ✅) help track test progress.
- **Test Outcome: Passed Successfully**
 - All expected UI elements were found.
 - User inputs were entered without errors.
 - Navigation to the next screen (FavoriteActivitiesScreen) occurred as intended.
 - No exceptions were thrown during execution.
- This confirms that the signup flow works correctly in the app under real user conditions.

```
I/flutter ( 8385): 00:00 +0: User signs up successfully and navigates to FavoriteActivitiesScreen
```

```
I/flutter ( 8385): 🟡 اختبار
D/EGL_emulation( 8385): app_time_stats: avg=100
I/flutter ( 8385): 00:56 +1: (tearDownAll)
I/flutter ( 8385): 00:56 +2: All tests passed!
All tests passed.
```

Figure 101: signup_test (success)

6.2.2.3 add_activity_test.dart

- **Purpose:**
This script runs a full **integration test** that validates the workflow of:
 - Logging into an existing user account.
 - Creating an activity programmatically for **this Sunday and the next two Sundays**.
 - Confirming that the activity appears correctly in the **HomeScreen UI** after login.

- **Initialization:**
 - Initializes the **Flutter integration test binding**.

```
IntegrationTestWidgetsFlutterBinding.ensureInitialized();
```

Figure 102: `add_activity_test`(Flutter integ test binding)

- Initializes **Firestore** before launching the app.

```
// ✅ Initialize Firestore and run app
await Firestore.initializeApp();
```

Figure 103: `add_activity_test` (init firestore)

- Uses helper functions to:
 - Wait for a specific widget by Key.

```
Future<void> waitForActivityByKey(WidgetTester tester, String keyValue,
    {Duration timeout = const Duration(seconds: 10)}) async {
  final endTime = DateTime.now().add(timeout);
  while (DateTime.now().isBefore(endTime)) {
    await tester.pump(const Duration(milliseconds: 500));
    final found = find.byKey(Key(keyValue));
    if (found.evaluate().isEmpty) return;
  }
  throw Exception('❌ Widget with Key("$keyValue") not found in UI within timeout.');
```

Figure 104: `add_activity_test` (wait by key)

- Wait for Firebase Authentication to confirm the user is signed in.

```
Future<User> waitForFirebaseUser({Duration timeout = const Duration(seconds: 10)}) async {
  final endTime = DateTime.now().add(timeout);
  while (DateTime.now().isBefore(endTime)) {
    final user = FirebaseAuth.instance.currentUser;
    if (user != null) return user;
    await Future.delayed(const Duration(milliseconds: 500));
  }
  throw Exception('❌ FirebaseAuth.currentUser is still null after waiting.');
```

Figure 105: : add_activity_test (wait firebase auth)

- **Running the App:**
 - Runs the main app entry point using app.main() after Firebase is ready.
 - Waits for the app to settle before continuing.

```
app.main();
await tester.pumpAndSettle(const Duration(seconds: 10));
```

Figure 106: : add_activity_test (Run)

- **UI Interactions:**
 - 1. Launch Screen**
 - Taps the "التالي" button.
 - Waits for UI to settle.
 - 2. Onboarding Screen**
 - Taps the "قم بالتسجيل الدخول" RichText link.
 - 3. Login Screen**
 - Finds and fills the **email** and **password** fields via their Key.
 - Taps the **login button**.
 - 4. Authentication Check**
 - Waits for Firebase to return a valid signed-in user.
 - Prints the user's UID and email as confirmation.

```

39 // 🟡 Tap "التسجيل" on launch screen[
40 final launchButton = find.text('التسجيل');
41 expect(launchButton, findsOneWidget);
42 await tester.tap(launchButton);
43 await tester.pumpAndSettle(const Duration(seconds: 3));
44 print("🟡 Pressed 'التسجيل'");
45
46 // 🟡 Tap "قم بالتسجيل الدخول" on onboarding screen
47 final loginLink = find.byWidgetPredicate(
48   (widget) => widget is RichText && widget.text.toPlainText().contains('قم بالتسجيل الدخول'),
49 );
50 expect(loginLink, findsOneWidget);
51 await tester.tap(loginLink);
52 await tester.pumpAndSettle(const Duration(seconds: 3));
53 print("🟡 Pressed 'قم بالتسجيل الدخول'");
54
55 // 🟡 Fill login fields
56 const testEmail = 'sama@gmail.com';
57 const testPassword = '123456789';
58
59 final emailField = find.byKey(const Key('email_field'));
60 final passwordField = find.byKey(const Key('password_field'));
61
62 expect(emailField, findsOneWidget);
63 await tester.enterText(emailField, testEmail);
64
65 expect(passwordField, findsOneWidget);
66 await tester.enterText(passwordField, testPassword);
67
68 // 🟡 Tap login button
69 final loginButton = find.byKey(const Key('login_button'));
70 expect(loginButton, findsOneWidget);
71 await tester.tap(loginButton);
72 await tester.pumpAndSettle(const Duration(seconds: 5));
73 print("🟡 Pressed 'تسجيل الدخول'");
74
75 // ✅ Wait for Firebase user to be available after login
76 final user = await tester.runAsync(waitForFirebaseUser);
77 final uid = user!.uid;
78 final email = user.email ?? 'no-email-available';
79 print('✅ User is signed in with UID: $uid, Email: $email');

```

Figure 107: : `add_activity_test` (UI interactions)

- **Firestore Interaction:**
 - Creates an activity named "النشاط" for the current and upcoming two Sundays.
 - Each scheduled activity includes:
 - A scheduledDate timestamp.
 - Start and end times.

- isCompleted: false.
- This is inserted under the user's Activates subcollection in Firestore.

```

81 // 📌 Add test activity to Firestore for this Sunday and upcoming Sundays
82 final now = DateTime.now();
83 final int daysSinceSunday = now.weekday % 7; // Sunday = 0
84 final thisSunday = now.subtract(Duration(days: daysSinceSunday));
85 final thisSundayMidnight = DateTime(thisSunday.year, thisSunday.month, thisSunday.day);
86
87 // Add for this Sunday + next 2 Sundays
88 final upcomingSundays = List.generate(3, (i) {
89     final nextSunday = thisSunday.add(Duration(days: 7 * i));
90     return DateTime(nextSunday.year, nextSunday.month, nextSunday.day);
91 }); // List.generate
92
93 final firestore = FirebaseFirestore.instance;
94 final activityRef = firestore
95     .collection('Patient')
96     .doc(vid)
97     .collection('Activates')
98     .doc('integration-activity');
99
100 await tester.runAsync(() async {
101     await activityRef.set({'ActivityName': 'النشاط'});
102
103     for (int i = 0; i < upcomingSundays.length; i++) {
104         await activityRef
105             .collection('ScheduledDates')
106             .doc('auto-test-${i + 1}')
107             .set({
108                 'scheduledDate': Timestamp.fromDate(upcomingSundays[i]),
109                 'startTime': '09:00',
110                 'endTime': '10:00',
111                 'isCompleted': false,
112             });
113     }
114 });
115
116 print('✅ Activity added to Firestore for this and upcoming Sundays');

```

Figure 108: : add_activity_test (firestore interactions)

- **UI Validation:**

1. Tap on "الأحد"

- Simulates the user selecting the **Sunday tab** to view that day's activities.

2. Wait for Activity to Appear

- Waits for a widget with key "النشاط" to be visible.
- Confirms that the activity appears as expected in the UI.

```
// 🔄 Wait for UI to show the new activity
await tester.pumpAndSettle(const Duration(seconds: 5));

// 📅 Tap on Sunday
final dayBox = find.byKey(Key('الأحد'));
expect(dayBox, findsOneWidget);
await tester.tap(dayBox);
await tester.pumpAndSettle();

// ✅ Wait for activity widget with Key('النشاط')
await waitForActivityByKey(tester, 'النشاط');
expect(find.byKey(Key('النشاط')), findsOneWidget);
print('✅ Activity is visible on HomeScreen with key "النشاط"');
});
```

Figure 109: add_activity_test (UI validation)

- **Why this test matters:**

- **End-to-end flow** that connects UI, authentication, and database layers.
- Ensures the system displays backend-scheduled content on the frontend.
- Confirms Firebase operations are correctly handled during integration testing.

- **Best Practices Followed:**

- expect(..., findsOneWidget) confirms UI elements are present before interaction.
- await tester.pumpAndSettle (...) is used for proper UI wait timing.
- Helper functions handle asynchronous waiting for Firebase and widgets.
- Print logs (🔴, ✅, ❌) help trace and debug each stage of execution.

- **Test Outcome: Passed Successfully**
 - All expected UI elements were found.
 - Firebase login completed successfully.
 - Firestore data was written without error.
 - UI correctly rendered the scheduled activity under the proper day.
 - No exceptions were thrown during execution.
- This confirms that the **activity scheduling and visibility flow works reliably** for users signed in through Firebase.

```

I/flutter ( 7480): 00:00 +0: Add activity on this and upcoming Sundays, and verify it appears on HomeScreen
VMServiceFlutterDriver: Connected to Flutter application.
I/example.obour_1( 7480): Background concurrent mark compact GC freed 5606KB AllocSpace bytes, 39(1716KB) LOS obje
D/EGL_emulation( 7480): app_time_stats: avg=11146.45ms min=10100.40ms max=12192.49ms count=2
W/WindowOnBackDispatcher( 7480): OnBackInvokedCallback is not enabled for the application.
W/WindowOnBackDispatcher( 7480): Set 'android:enableOnBackInvokedCallback="true"' in the application manifest.
D/EGL_emulation( 7480): app_time_stats: avg=889.27ms min=10.53ms max=2748.30ms count=4
I/flutter ( 7480): 🟡 Pressed 'لوجن'
I/flutter ( 7480): ✅ Login Screen is building...
I/flutter ( 7480): ✅ Login Screen is building...
D/EGL_emulation( 7480): app_time_stats: avg=356.78ms min=160.38ms max=734.35ms count=3
I/flutter ( 7480): 🟡 Pressed 'لوجن'

I/flutter ( 7480): 🟡 Pressed 'لوجن'
I/flutter ( 7480): ✅ User is signed in with UID: g0vs7oEbSmgsPy8gVSK4uWPSqHg1, Email: sama@gmail.com
I/flutter ( 7480): ✅ Activity added to Firestore for this and upcoming Sundays
D/EGL_emulation( 7480): app_time_stats: avg=5409.71ms min=5409.71ms max=5409.71ms count=1
I/flutter ( 7480): ✅ Activity is visible on HomeScreen with key "طابق"
I/flutter ( 7480): 00:53 +1: (tearDownAll)
I/flutter ( 7480): 00:53 +2: All tests passed!
All tests passed.

```

Figure 110: : add_activity_test (success)

6.2.3 System Testing

System testing is conducted to validate the complete and integrated functionality of the OBOUR application. It ensures that all modules work together as intended and that the app meets its functional requirements. This phase tests the end-to-end flow of real user scenarios, verifying that user inputs are properly handled, data is correctly stored and retrieved, and the navigation between features is seamless. The tests also ensure consistency of data, accuracy of feedback, and user interface reliability across different sections of the app.

The following table summarizes key system test scenarios executed during this phase, along with their expected results and outcomes:

Table 16: System Testing

ID	Test Scenario	Expected Result	Pass/Fail
1	Start from the Obour onboarding page → Tap “Start” → Navigate to the Sign-Up page → Enter patient name, email, password, phone number, and birth date → Tap “Continue” → On the Activity Selection page, choose favorite activities → Tap “Add Extra Activity”, enter an activity name, and tap “Add” → Tap “Continue” → On the Commit Activities page, tap “Commit Activities” → On the Prioritize Activities page, drag to reorder activities by preference → Tap “Continue” → On the Free Time Selection page, select available days and time slots → Tap “Continue” → On the Quote Type page, select a preferred quote type → Tap “Continue” → On the Success page, tap “Let’s Start” → Navigate to the Home page.	User account is successfully created. All entered information is saved. Based on the selected favorite activities, priorities, and free time slots, a personalized weekly activity plan is generated and displayed on the Home page.	Pass
2	Log in page → enter Email → enter password → press log in button → navigate to home page	User is authenticated using email and password. If credentials are valid, the app navigates to the Home page with the user’s data loaded from Firestore. If invalid, an error message is displayed.	Pass
3	Home page → Tap the checkbox to mark an activity as done → Tap the "Edit Activity Plan" button → Navigate to the Activity Selection page → Select favorite activities → Tap “Add Extra Activity”, enter the activity name, and tap “Add” → Tap “Continue” → On the Commit Activities page, tap “Commit Activities” → On the Prioritize Activities page, reorder activities by dragging → Tap “Continue” → On the Free Time Selection page, select preferred days and times → Tap	Updated activity preferences are saved. A new weekly activity plan is generated based on selected activities, their priorities, and available time slots. The Home page reflects the updated plan accordingly.	Pass

	“Continue” → On the Success page, tap “Let’s Start” → Return to the Home page.		
4	Home page → navigate to Peer Contact page from tab bar → add peer contact → select peer contact → send message → open SMS app	Contact saved, SMS intent triggered	Pass
5	Home page → navigate to Badges page from tab bar → view badges and counts of completed activates	Badge screen shows unlocked achievements and counts of completed activates	Pass
6	Home page → press Emoji icon → view motivational quote	Quote matches emotion and quote type	Pass
7	Home page → navigate to Media page from tab bar → view media categories → select a media category → navigate to the related media page	Selected media content loads	Pass
8	Home page → Navigate to Profile page from the bottom tab bar → Tap “Activities History” → Navigate to Activities History page → Tap the notification bell → Enable activity reminders → Tap the history calendar icon → Select a date → View completed activities → Tap Save → Tap Back → Tap “Edit Profile” → Edit name, email, password, birth date, and phone number → Tap Save → Return to Profile page → Tap the motivation quote icon → Navigate to Quote Type page → Update quote preferences → Tap Save → Return to Profile page → Tap the Logout icon → Confirm logout → Return to Onboarding page	Views activity history, updates profile and quote preferences, enables notifications, logs out, and returns to the onboarding screen.	Pass

6.3 Non-functional testing

6.3.1 Compatibility Testing

Compatibility testing was performed to ensure that the OBOUR application runs smoothly across different devices, Android versions, and hardware configurations. This type of testing is essential to verify that the app maintains its functionality, responsiveness, and visual consistency regardless of the device model.

The application was tested on multiple Android phones with varying hardware specifications and storage availability. All tested devices successfully installed, launched, and ran the app without issues such as crashes, display errors, or lag. This confirms that the OBOUR application is compatible with a range of Android environments.

The table below summarizes the devices used during compatibility testing and their outcomes:

Table 17: Compatibility Testing

Phone Release	Android Version	Memory Space	Available Space	Pass/Fail
Galaxy S9+ (SM-G965F)	10	6GB	2.9GB	Pass
Samsung Galaxy S22	14	8GB	2.9GB	Pass
Google Pixel 8	15	8GB	3GB	Pass

6.4 Conclusion

The testing phase was essential in ensuring that the Obour application performs reliably and meets both functional and non-functional requirements. Through system testing, we verified that all features worked together as intended, providing a seamless experience for users such as individuals in recovery, mentors, and administrators.

In addition to functionality, non-functional testing—particularly compatibility testing—confirmed that the application runs smoothly across different devices, screen sizes, and operating systems. This was especially important given the diverse user base Obour aims to support. Overall, the testing process helped identify and resolve potential issues before deployment, ensuring a more stable and accessible application. It also laid the foundation for ongoing quality assurance as the system evolved.

Chapter 7: Results and discussions

This chapter begins with an introduction in Section 7.1. Section 7.2 showcases the OBOUR application's user interfaces designed for patients. In Section 7.3, we evaluate the primary goals of the OBOUR application and assess their fulfillment. Lastly, Section 7.4 outlines the limitations encountered during the development or implementation process.

7.1 Introduction

This section offers a thorough summary of the results of the OBOUR application, showcasing its user interfaces along with clear usage instructions. It also assesses how well the project objectives were met and outlines the key challenges and limitations faced during the development process.

7.2 Results

There is one main interface designed for patients. Through this interface, the patient can customize a weekly plan, communicate with peers via SMS, view media content, and track earned achievement badges. The application also displays supportive quotes tailored to the user's emotional state and selected quote type. Additionally, patients can enable notifications for scheduled activities and more. The following subsections will discuss each feature in detail.

7.2.1 Login and Signup (Generated activity plan)

The patient begins by accessing the **Login Page**, where they are required to enter their credentials. The system then checks whether the account exists. If the credentials are valid, the patient is granted access to their personalized interface. If the account does not exist, the patient must click the **Sign-Up** button to create a new account.

During the **Sign-Up** process, the patient is prompted to provide personal information. The system validates the entered data, including email address and phone number, to ensure it meets the required criteria. After entering the necessary information, the patient clicks the **Continue** button.

Next, the patient is asked to select their **favorite activities** from a list. If an activity they enjoy is not listed, they can click the **"Add New Activity"** button to include it manually. Once activities are selected, the patient proceeds to a **Confirmation Page**, where they are asked to confirm their chosen activities.

Following confirmation, the patient can **prioritize** the activities by ranking them from most to least preferred. Then, the patient selects their **available days** and sets a **start time and end time** for each selected day.

After scheduling, the patient is directed to the **Quotes Page**, where they choose the types of motivational or supportive quotes they would like to receive throughout their journey.

Finally, the patient is taken to the **Home Screen**, where their personalized **activity plan** is displayed based on all the preferences and inputs provided during the sign-up process.

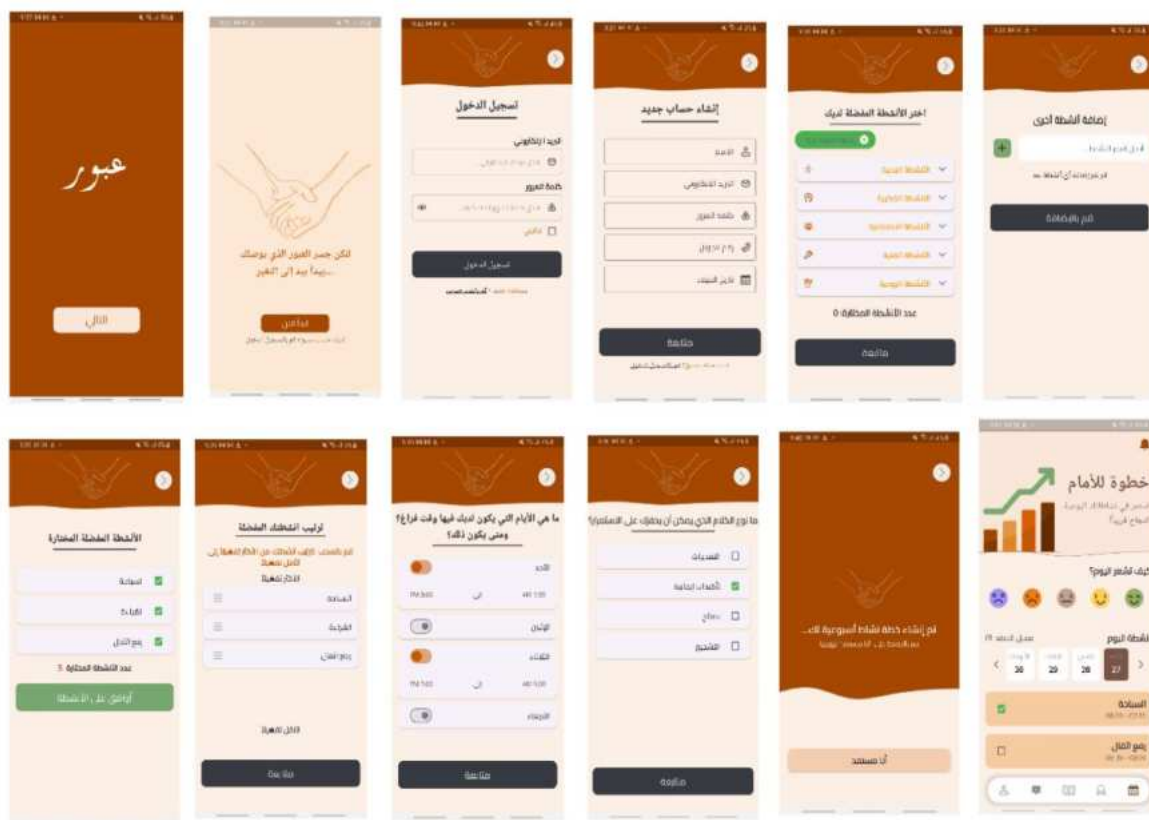


Figure 111: Login and Signup Process (Generated activity plan)

7.2.2 Display Supportive Quote

On the **Home Screen**, the patient has the option to express their current emotional state by selecting one of several emoji icons, such as **happy**, **sad**, **angry**, and **others**. Based on the selected emotion and the **preferred quote types** the patient chose earlier during setup, the system automatically displays a **supportive quote** that aligns with both the emotion and the

chosen motivational style. This feature helps provide personalized emotional support as the patient navigates their wellness journey.



Figure 112: Display Supportive Quote

7.2.3 Achievement Badges Screen

On the **Achievement Badges** screen, the patient can view the **number of completed activities** along with the **badges earned** as a result of their progress. This feature allows patients to track their accomplishments and stay motivated by visually seeing their achievements over time.



Figure 113: Achievement Badges Screen

7.2.4 Peer Contact Screens

On the **Peer Contact** page, the patient can add a new contact by entering the person's **name and phone number**, then clicking the **"Add Contact"** button. Patients can also **search** for specific contacts from their saved list. Once a contact is selected and the **"Send Message"** button is clicked, the system launches the default **SMS app** with a **pre-written support message**, ready for sending. This feature provides a quick and convenient way for patients to connect with peers for encouragement and emotional support.

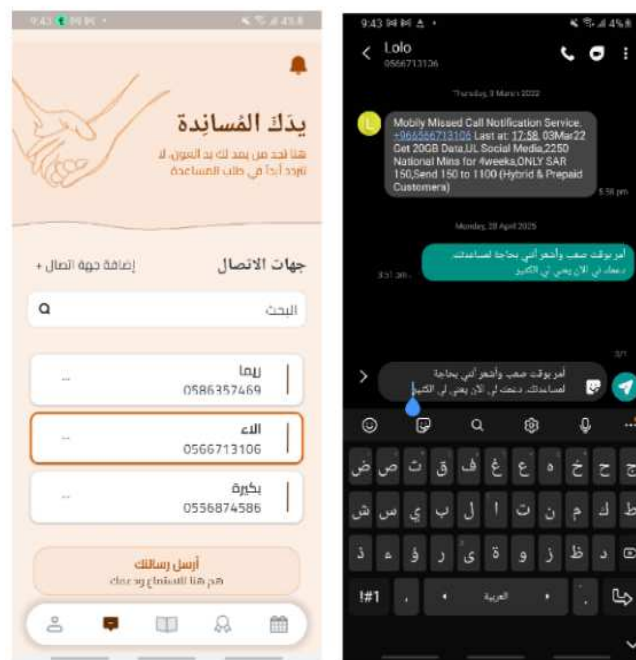


Figure 114: Peer Contact Screens

7.2.5 Media Screens

On the **Media** screen, the patient can select their **preferred type of media**, such as **YouTube videos**, **educational content**, or **podcasts**. This allows the application to tailor media recommendations to the patient's interests and support their engagement throughout the journey.

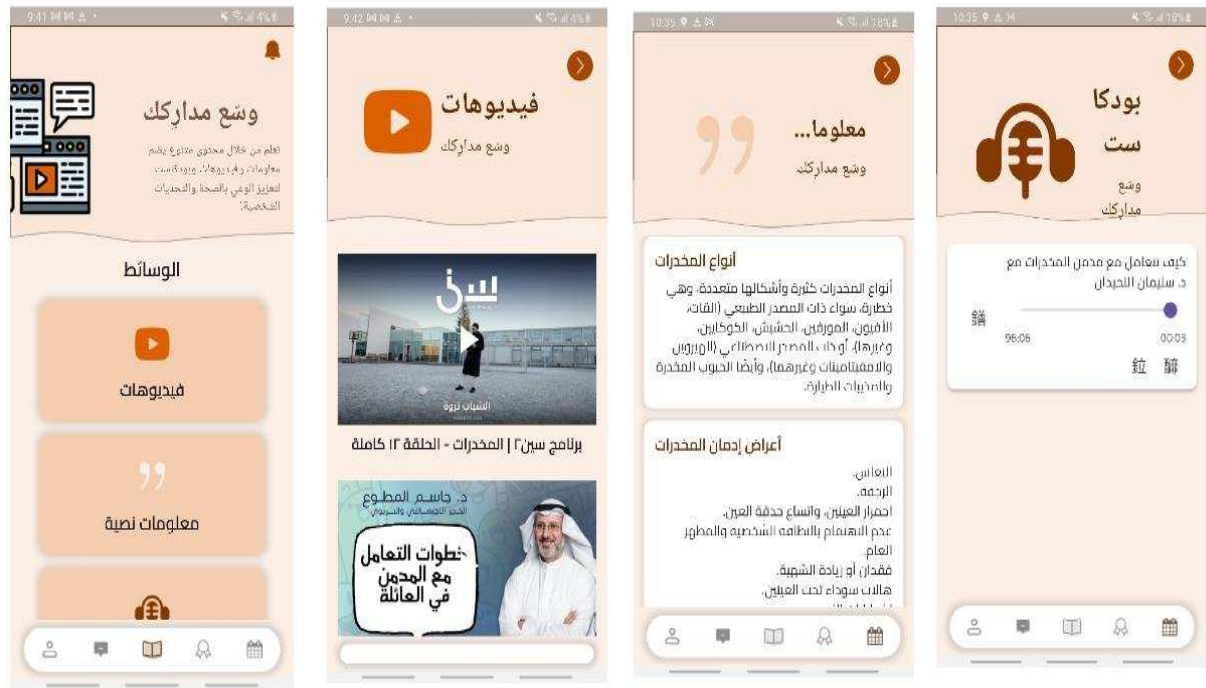


Figure 115: Media Screens

7.2.6 Profile Screens

On the **Profile** screen, the patient can navigate to several key sections. They can access the **Settings** page to edit their personal information, and the **History of Activities** page to enable **notifications** for activities and view a record of all **completed activities**. Additionally, the patient can go to the **Quote Preferences** page to update their preferred types of motivational quotes. The **Profile** page also provides the option to **log out** of the application.

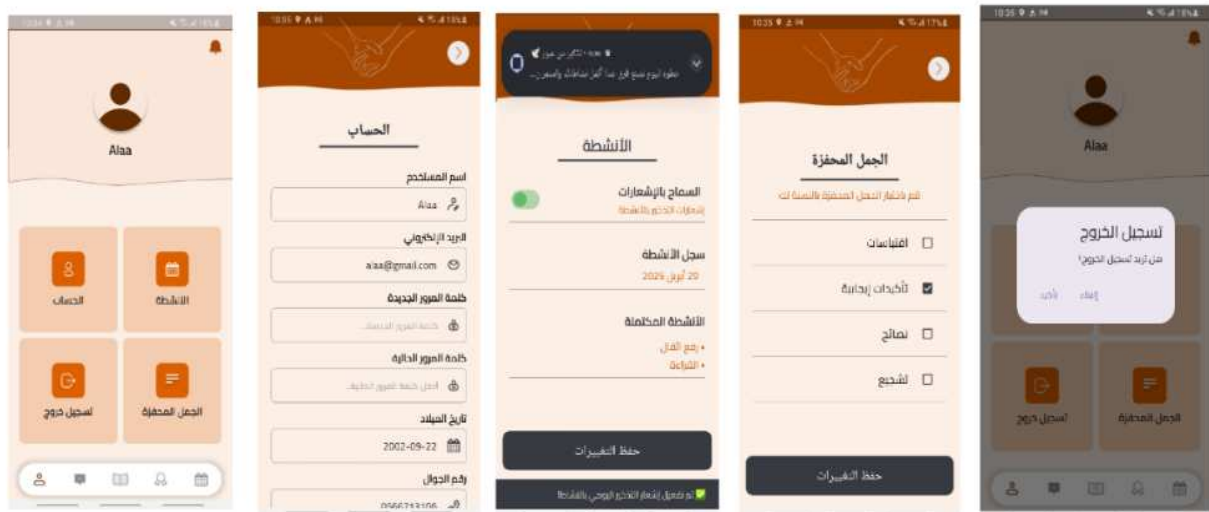


Figure 116: Profile Screens

7.3 Accomplished Objectives

The OBOUR app successfully fulfilled the following core objectives:

- **Personalized Recovery Plans:** Created structured activity schedules tailored to users' cultural, religious values and routines.
- **Progress Tracking:** Added achievement badges to motivate users through visible recovery milestones.
- **Inspirational Support:** Displayed daily religious and motivational messages based on users' emotions and preferred quote type.
- **Reminders:** Sent notifications for scheduled activities to keep users engaged.
- **Awareness & Education:** Provided Arabic educational content on substance uses and recovery.
- **Peer Support:** Enabled users to add peers and send SMS for support.

7.4 Project Limitation

- No real-time peer chats.
- Internet required for most features.
- Emotions selected manually; no automatic detection.
- No integration with professional recovery services.
- AI training for Arabic emotional expressions was challenging.
- Adapting to cultural and linguistic needs required extra effort.
- Limited access to recovering users restricted firsthand research.
- Difficulty consulting experts for medical and motivational content.

Chapter 8: Conclusion and future work

This chapter presents the final evaluation of the OBOUR project. Section 8.1 outlines the overall project conclusion, summarizing its goals, achievements, and societal impact. Section 8.2 proposes strategic areas for future development to enhance the application's effectiveness and scalability. Finally, Section 8.3 concludes the chapter and reflects on the broader implications of this work.

8.1 Project Conclusion

The *OBOUR* application is a pioneering digital health initiative aimed at supporting Arabic-speaking patients in Saudi Arabia who are recovering from substance use disorders. It addresses a critical gap in post-rehabilitation care by providing culturally and linguistically tailored tools that empower individuals to maintain sobriety, build structure, and strengthen their emotional resilience during recovery.

Throughout the development process, the team conducted extensive research, consulted addiction recovery specialists, and designed an intuitive interface that meets the specific needs of the target audience. OBOUR integrates a wide range of features such as personalized activity scheduling and a reminder for the activities, AI-generated emotional support messages, SMS-based peer support, educational content, and motivational achievement badges—all designed to reinforce healthy behaviors and reduce relapse risk.

The development phase of the project was followed by a comprehensive testing process, which included:

- **Unit Testing:** To verify the correctness of individual components such as user sign-up, activity setup, activity prioritization, peer contact management, and saving contact information to Firestore, among other core functionalities.
- **Integration Testing:** To ensure that connected modules work together as expected. This included validating the integration of the sign-up process and the add activity feature to ensure smooth user experience across screens and data flow.
- **System Testing:** To validate the entire application's functionality, stability, and user flow across all screens and features.
- **Compatibility Testing:** To confirm that the app performs consistently across different Android devices, screen sizes, and system versions.

This thorough testing approach guaranteed a stable, user-friendly experience, essential for a sensitive application designed for individuals in recovery.

The results demonstrate the success of OBOUR in meeting its initial objectives: providing a reliable and culturally sensitive digital companion for recovery. It empowers users to take control

of their rehabilitation journey with support tools that are emotionally responsive, customizable, and clinically informed. More importantly, the application promotes awareness, combats stigma, and facilitates reintegration into society.

8.2 Future Work

While the current version of *OBOUR* offers essential support features for post-rehabilitation patients, several enhancements can be implemented in future releases to increase its therapeutic value and reach:

- **AI for Diagnosis and Severity Assessment**
Leveraging artificial intelligence to assess the severity of substance use and diagnose related mental health issues can dramatically improve treatment precision. Predictive analytics and emotional pattern recognition could help detect early signs of relapse, enabling timely intervention.
- **AI-Based Personalized Activity Planning**
Enhancing the current scheduling system with machine learning can allow the app to create dynamic, personalized activity plans based on users' evolving preferences and behaviors. This approach can increase user engagement, improve adherence to recovery routines, and reduce dropout rates.
- **Medical Follow-Up Integration**
Introducing features that allow medical professionals to monitor patient progress through dashboards or periodic updates will support more effective recovery. Doctors could provide personalized recommendations, track symptom changes, and intervene when necessary, creating a stronger link between technology and professional care.

These improvements will not only boost the therapeutic value of the app but also position **OBOUR** as a scalable, clinically integrated solution capable of contributing meaningfully to national mental health and addiction recovery strategies.

8.3 Conclusion

In conclusion, *OBOUR* represents a significant step forward in combining digital technology with culturally informed addiction recovery. It offers a comprehensive set of tools that directly address the psychological, social, and behavioral challenges faced by individuals in post-rehabilitation. From personalized plans and AI-generated quotes to peer support and educational content, the application serves as a holistic support system rooted in empathy, usability, and effectiveness.

Our journey in building OBOUR was grounded in understanding the complex realities of recovery, the cultural context of our users, and the potential of modern technology to drive real-world impact. Through structured development, expert consultation, and rigorous testing, we delivered a tool that not only meets its intended purpose but lays the groundwork for future innovation in digital health for addiction recovery.

OBOUR is more than an app—it is a digital companion for hope, healing, and sustained change. With further development and clinical integration, it can become a cornerstone of personalized, tech-driven recovery solutions in the Arab world and beyond.

Reference

[1] Substance use and Co-Occurring mental disorders. (n.d.). National Institute of Mental Health (NIMH).

[https://www.nimh.nih.gov/health/topics/substance-use-and-mental-health#:~:text=Substance%20use%20disorder%20\(SUD\)%20is,most%20severe%20form%20of%20SUD.](https://www.nimh.nih.gov/health/topics/substance-use-and-mental-health#:~:text=Substance%20use%20disorder%20(SUD)%20is,most%20severe%20form%20of%20SUD.)

[2] I Am Sober | Sobriety App for Android & iOS. (n.d.). I Am Sober.

<https://iamsobber.com/en/site/home>

[3] JMIR Formative Research. (2021). Design and evaluation of digital tools for behavioral health. *JMIR Formative Research*. Retrieved from

<https://formative.jmir.org/2021/11/e25749/>

[4] PubMed. (2017). The impact of health apps on patient outcomes: A systematic review.

PubMed. Retrieved from

<https://pubmed.ncbi.nlm.nih.gov/29129192/>

[5] WeConnect Health Management. (n.d.). Free recovery support app for addiction and mental health. Retrieved from

<https://www.weconnecthealth.io/free>

[6] Sober Time. (n.d.). App to track sobriety and support addiction recovery. Retrieved from

<https://sobertime.app/>

[7] I Am Sober | Sobriety App for Android & iOS. (n.d.). I Am Sober.

<https://iamsobber.com/en/site/home>

Appendix A

A.1 Data Gathering Technique

A.1.1 Interview

We interviewed Dr. Mohamed Alghamdi, a psychologist specializing in addiction treatment at Al-Amal Hospital, to better understand the needs of individuals recovering from addiction. Our discussion covered key topics such as general topics of addiction, treatment plans, motivation, relapse, and family involvement. His insights helped shape the core features of the *Obour* app, including activity planning to support long-term recovery and prevent relapse, emotional support to provide motivation, contact tools to seek help from family or friends, and educational resources to raise awareness about substance use.

Table 18: Appendix A (Interview)

Interview Outline	
Interviewee: Dr.Mohamed Alghamdi	Interviewer: Reema Hesamudin - 2106126 Alaa Asghar - 2105843 Bakerah Alseari -2107854
Location: Al Amal Hospital	Appointment Date: Monday, September 9, 2024 Start Time: 12:00 PM End Time: 12:43 PM
Objectives: Gathering Requirements of new system	Reminders: The interviewee is psychologist in Al Amal Hospital
Introduction Background on Project Overview of Interview	Approximate Time:

Question 1	3 min
Question 2	2 min
Question 3	1 min
Question 4	1 min
Question 5	2 min
Question 6	4 min
Question 7	4 min
Question 8	3 min
Question 9	2 min
Question 10	3 min
Summary of Major Points	
Questions from Interviewee	
Closing	
General Observations:	
Interviewee: Dr.Mohamed Alghamdi	
Date: Monday, September 9, 2024	
Question 1 How is addiction diagnosed?	Diagnosis is based on symptoms like continued substance use, social and psychological harm, withdrawal symptoms, and difficulty stopping the substance.
Question 2 How is addiction level determined?	Addiction level is determined through urine and blood tests to identify the type of substances used.
Question 3 Does gender affect diagnosis?	No, the diagnosis depends on symptoms, though some social factors may differ between males and females.
Question 4 What age groups are most vulnerable to addiction?	Addiction typically starts around age 12, peaks during adolescence, and decreases after age 30.
Question 5 Does addiction affect brain function?	Yes, drugs can impair the brain, leading to memory loss, concentration issues, and other cognitive effects.
Question 6 What is the treatment plan for addiction?	The plan includes therapy sessions, lifestyle changes, medications, and behavioral therapy.
Question 7 How is motivation for treatment assessed?	Motivation is assessed when the patient recognizes their issues and wants to change for themselves and their family.

Question 8 What are relapses?	Relapse is returning to drug use after a period of abstinence. It's a normal part of the recovery process.
Question 9 What motivates addicts to follow the treatment plan?	Motivation comes from internal desire to recover and external support from family and friends.
Question 10 How is the family involved in the treatment?	Family is involved through psychoeducation, learning how to support the patient and provide encouragement during recovery.

A.1.2 Survey:

We conducted a survey with 60 participants to gain insights into public perceptions and needs related to addiction recovery.

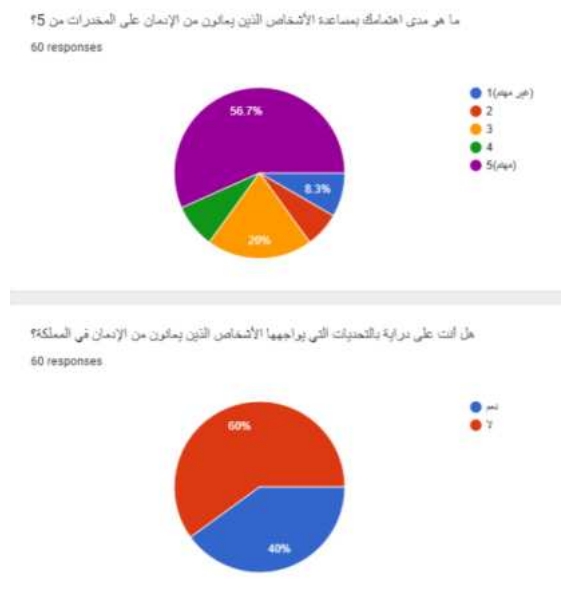


Figure 117: Appendix A (Survey Q1 & Q2)

As shown in **Figure 19**: 56.7% of participants expressed interest in helping individuals struggling with addiction. 60% were aware of the challenges addiction presents.

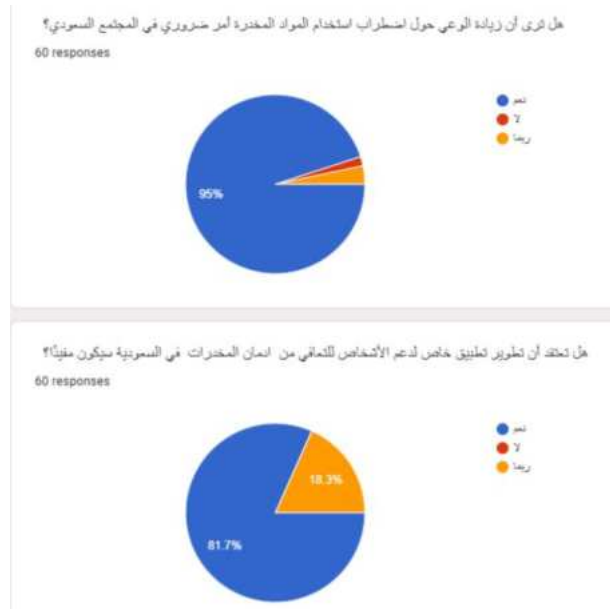


Figure 118: Appendix A (Survey Q3 & Q4)

As shown in **Figure 20**, 95% of respondents agree that apps could play a crucial role in supporting recovery, and 81.7% supported the development of a region-specific app for addiction recovery in Saudi Arabia, highlighting a positive reception to tailored solutions.



Figure 119: Appendix A (Survey Q5 & Q6)

As shown in **Figure 21**, An overwhelming 95% emphasized the importance of tailored behavioral therapy plans, while 66.7% highlighted the potential role of technology in improving recovery outcomes.

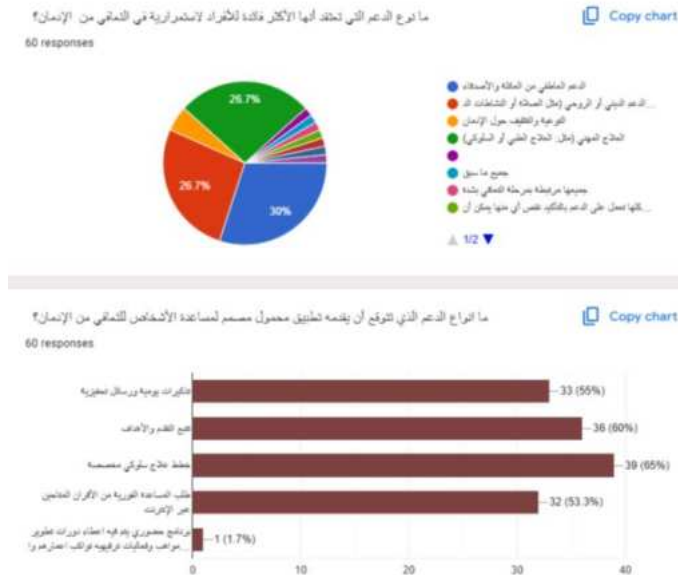


Figure 120: Appendix A (Survey Q7 & Q8)

As shown in **Figure 22**: The top factors contributing to addiction were identified as a lack of emotional support (30%) and financial or life stressors (26.7%). Participants expressed a strong preference for app features such as goal setting (60%), tailored therapy plans (65%), peer support tools (55%).

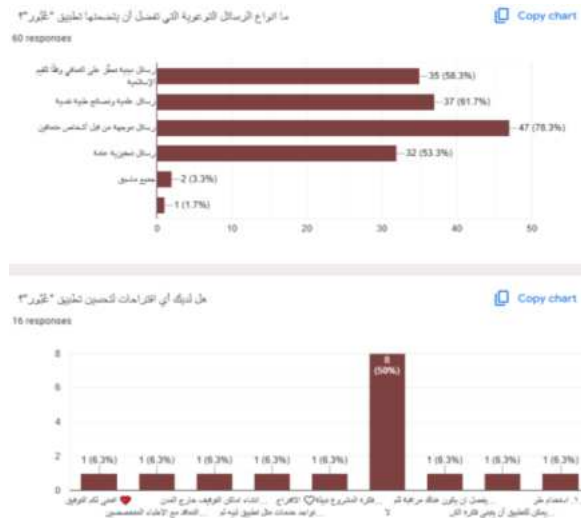


Figure 121: Appendix A (Survey Q9 & Q10)

As shown in **Figure 23**: impactful messages (78.3%) and practical guidance (61.7%). Suggestions for improvement included adding interactive features and expert advice.

A.2 Changes from Report 1

A.2.1 Modifications in report 1

Table 19: Changes from report 1

No.	The Tasks Modification	Section	PageNo.
1	Methodology: update the waterfall model	1.7	15 - 16
2	Designing Tools: update and remove from section	1.9	17 - 18
3	Analysis: update from Use case section	3.4.1	30
4	Design: change the UML structure	4.2	56 - 57
5	Design: Modify ER diagram	4.3	58
6	Design: Modify relational schema	4.4	59