A Mood Diary Application for Emotion Management

AN YU XUAN，Shahrul Azman Mohd Noah

Faculty of Information Science & Technology

Universiti Kebangsaan Malaysia

43600 Bangi, Selangor

Abstract

Projek ini memberi tumpuan kepada pembangunan Aplikasi Diari Mood yang bertujuan untuk meningkatkan kesedaran diri emosi dan kesejahteraan mental. Ramai individu bergelut untuk mengesan perubahan emosi mereka kerana kekurangan alat peribadi dan berstruktur, menjadikannya sukar untuk menganalisis corak mood dengan berkesan. Untuk menangani isu ini, penyelesaian yang dicadangkan menyepadukan penandaan emosi, entri diari, penjejakan berasaskan kalendar dan visualisasi trend mood statistik. Projek ini mengikuti model pembangunan air terjun, merangkumi fasa seperti analisis keperluan, reka bentuk sistem, pelaksanaan, ujian dan penggunaan. Aplikasi ini dibangunkan untuk Android menggunakan Firebase Cloud Storage untuk pengurusan data yang cekap. Hasil yang dijangkakan termasuk platform yang mesra pengguna dan selamat yang membolehkan individu merekod dan menganalisis emosi mereka dengan mudah, sekali gus meningkatkan refleksi diri dan pengurusan kesihatan mental.

Copyright@FTSM UKM

Abstract

This project focuses on the development of a Mood Diary Application aimed at enhancing emotional self-awareness and mental well-being. Many individuals struggle to track their emotional changes due to the lack of private and structured tools, making it difficult to analyze mood patterns effectively. To address this issue, the proposed solution integrates emotion tagging, diary entries, calendar-based tracking, and statistical mood trend visualization. The project follows the waterfall development model, encompassing phases such as requirements analysis, system design, implementation, testing, and deployment. The application is developed for Android using the Firebase Cloud Storage for efficient data management. Expected results include a user-friendly, secure platform that allows individuals to record and analyze their emotions conveniently, thereby improving self-reflection and mental health management.

1.0 INTRODUCTION

As life speeds up, the need for self-awareness and emotional well-being has surged. Many people want to record their daily lives and emotional states as a means of reflection, but there are few solutions that can record and organize people's moods while protecting privacy.

In modern society, most people use Twitter, Facebook, TikTok, and other software to record and publish their lives, but these tools can leak people's privacy. According to the report "The Effect of Privacy Perception on Social Media on Attitude Towards Social Media Usage" published by Journal of Yaşar University (January 2022), a survey that studied global users' attitudes towards social media privacy, the results showed that many users are aware to privacy risks but continue to use these platforms.

According to the World Health Organization (WHO), approximately 970 million people worldwide suffer from mental disorders, reflecting the impact of the stress of modern life on people’s mental health (WHO, 8/6/2022)

According to Dr. Jeremy Sutton’s research report “5 Mental Health Benefits of Journaling.” Keeping a diary can increase self-awareness, reduce our anxiety, and help us critically analyze our own thinking.

Therefore, this project aims to solve these problems and meet user needs. As smartphones gradually become one of the essential tools in our daily lives, people rarely use pen and paper to write diaries, so the development of this project can also help people record themselves more conveniently.

Copyright@FTSM UKM

2.0 LITERATURE REVIEW

Recent advancements in emotion tracking applications have integrated user-centered design principles, digital data collection methods, and artificial intelligence to improve mental health monitoring, particularly among students. For instance, Hamre-Os (2021) developed an emotion tracking interface focused on student mental health, emphasizing user-centered design (UCD). This approach highlights the importance of tailoring interfaces to user needs by involving users in design and testing stages. While UCD significantly enhances user engagement and usability, it is time-consuming and may not fully address the diverse range of user experiences or emotional states.

Digital methods such as Mood 24/7, which employs SMS-based tracking, have shown improved accuracy and engagement compared to traditional paper-based logs. However, its generalizability to broader use scenarios remains limited. The use of Ecological Momentary Assessment (EMA) in this context has enabled real-time emotional data collection, capturing spontaneous emotional states and enriching the emotional dataset. Despite its strengths, EMA can interrupt daily routines and lead to user fatigue when frequent check-ins are required.

Personalization also plays a crucial role in enhancing user experience. Adjusting features such as color schemes, emoji options, and notification frequency can significantly increase user satisfaction and engagement with emotion tracking applications. Some applications are beginning to integrate real-time physiological data collection, using wearable sensors to monitor indicators like heart rate and sleep patterns. This method offers a more holistic view of the user's physical and emotional state, although it faces challenges with device compatibility, data accuracy, and sustained user participation.

Artificial Intelligence is emerging as a key component in the evolution of emotion tracking. Applications like Feeling Moodie have begun experimenting with AI-driven sentiment prediction, though this technology is still in its early stages. Furthermore, some platforms incorporate AI-powered virtual coaching to provide real-time psychological support and advice based on users' emotional data. These AI features represent promising directions for future development, though their current capabilities remain limited by the quality and scope of training data and algorithms.

In summary, past studies have demonstrated that combining UCD, EMA, real-time data collection, and AI technologies can significantly improve the design and functionality of emotion tracking applications. However, each approach comes with its own set of limitations that future developments must address to ensure scalability, inclusivity, and long-term user engagement.

Copyright@FTSM UKM

3.0 METHODOLOGY

This chapter presents a comprehensive system design framework that ensures the project meets user needs for emotion tracking, privacy, and usability. It provides an overview of system architecture, database design, algorithm functionality, and interface design, providing a solid foundation for application development and implementation.

In addition, this chapter discusses the object-oriented approach to system modeling, detailing how each module and component of the system interacts. By focusing on modularity and scalability, this approach ensures that applications can adapt to changing user needs while maintaining strong performance and security. The functional and non-functional requirements provided will guide the development process, ensuring that the application is reliable, secure, and easy to use.

3.1 USER NEEDS

3.1.1 Emotion and Diary record

Users can log their daily emotions using a set of predefined emotion icons and write diary entries to provide context or elaborate on the reasons for their emotional changes or daily life events.

3.1.2 Emotion tracking and management

The app generates visual charts showing sentiment trends over time. These charts can help users understand how their emotions change over time and help them analyze their emotions themselves.

3.1.3 Picture function

Allow users to upload images while recording to enrich their recordings.

Copyright@FTSM UKM

3.1.4 Calendar function

Users can find related records based on date.

3.1.5 Search

Users can search for entries using keywords, making it easier to find specific records.

3.1.6 Privacy protection

All data is stored securely, ensuring that only users can access it. The app uses a password-protected login to protect sensitive information.

3.2 SYSTEM MODEL

The system model section outlines the object-oriented approach used to build the project. The focus of this section is to represent the interaction, processes and functional requirements between visualization and the system through diagrams (Figures 1,2,3,4,5).

3.2.1 Casse Diagram

Case diagrams: The diagram describe the functional requirements of the system and illustrate the different actions a user can perform in the application.

Copyright@FTSM UKM

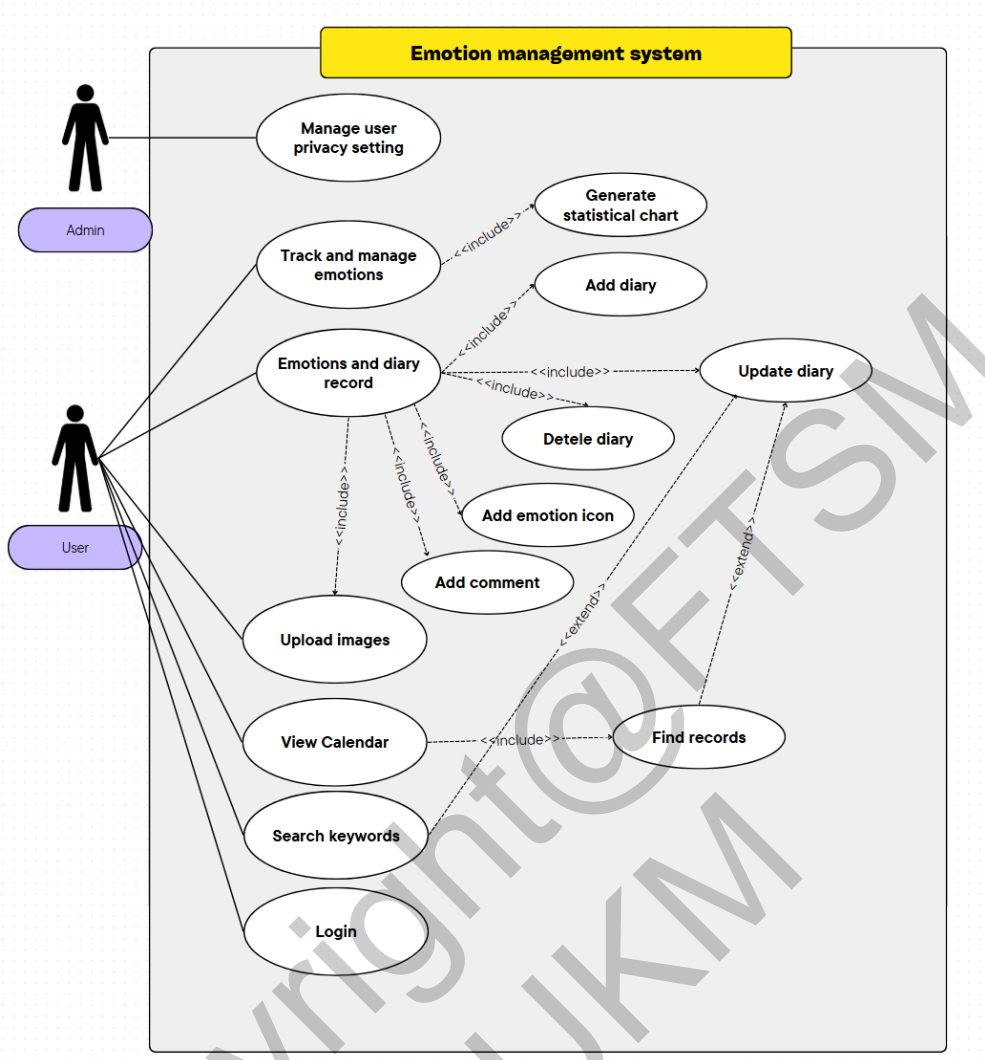
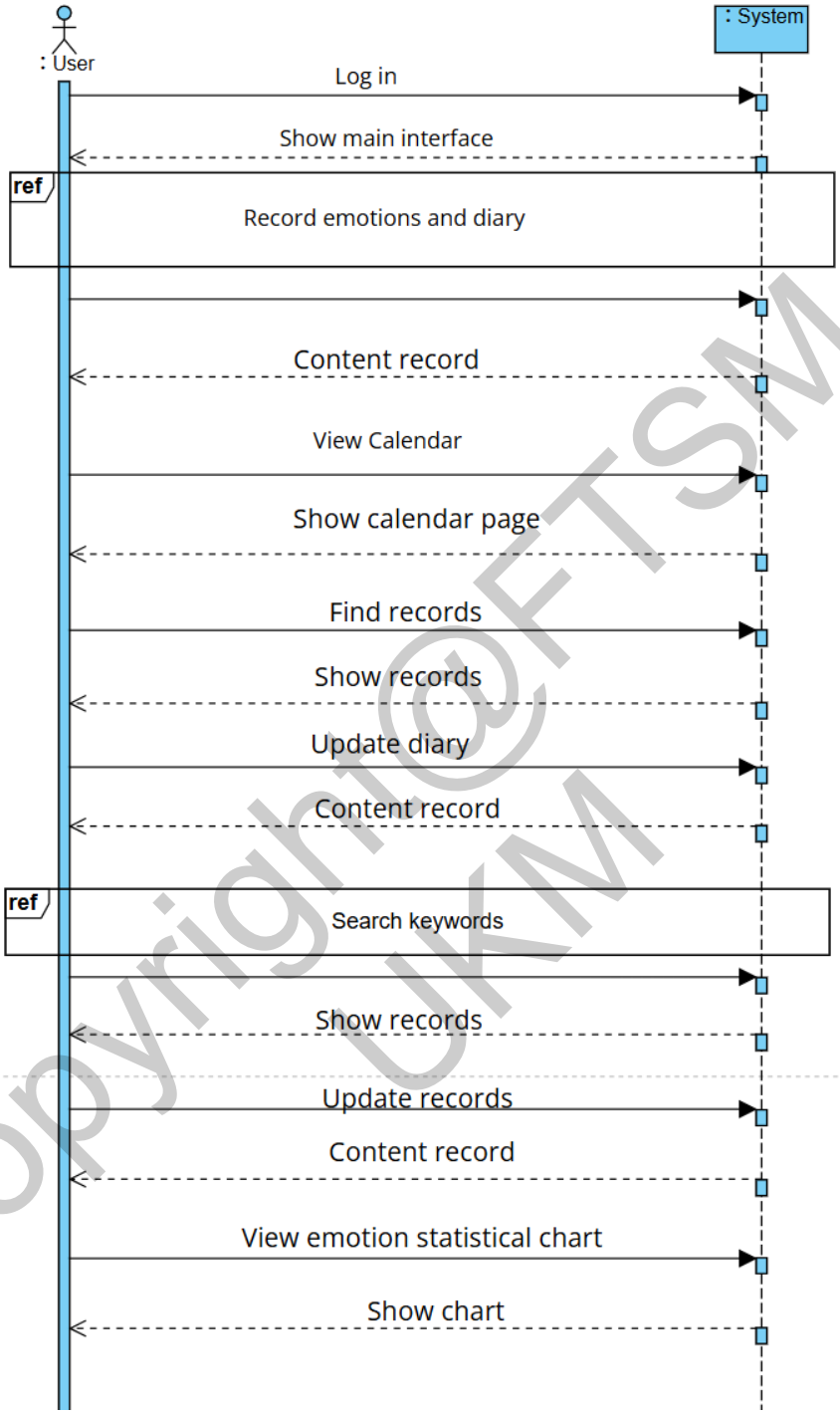


Figure 1 Functional requirements of the system

3.2.3 Sequence diagram

Sequence diagrams: These diagrams show the sequence of interactions between users and the system during key processes.

Copyright@FTSM UKM



Copyright@FTSM UKM

Figure 2 Interactions between the user and the system

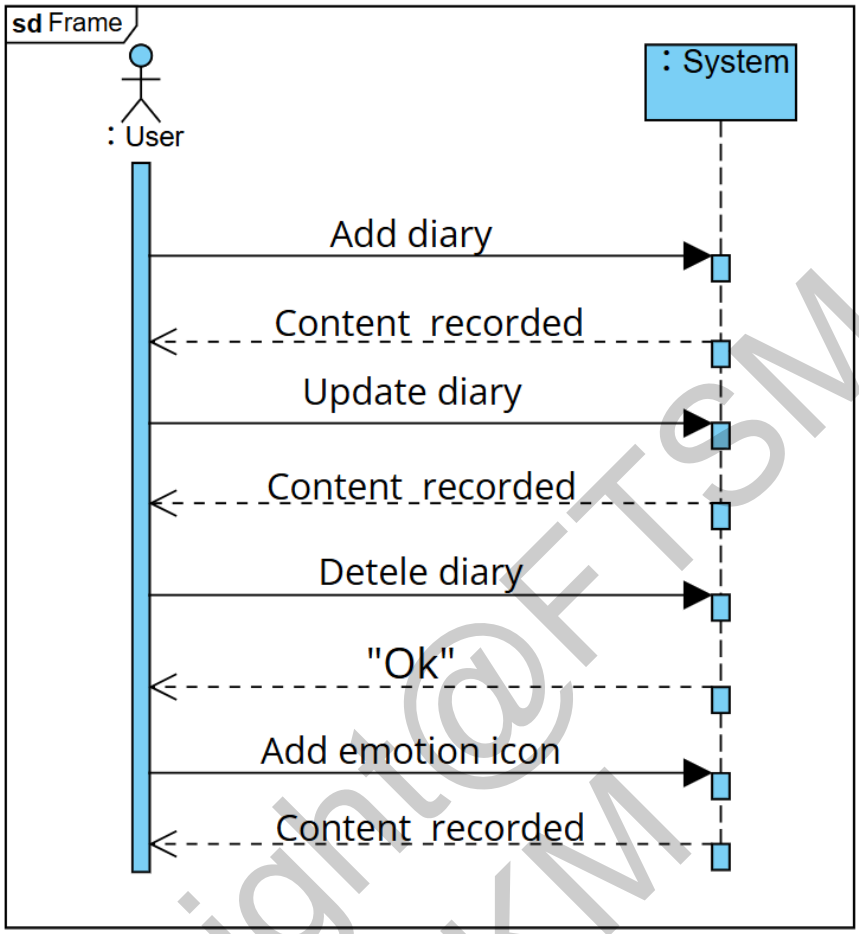
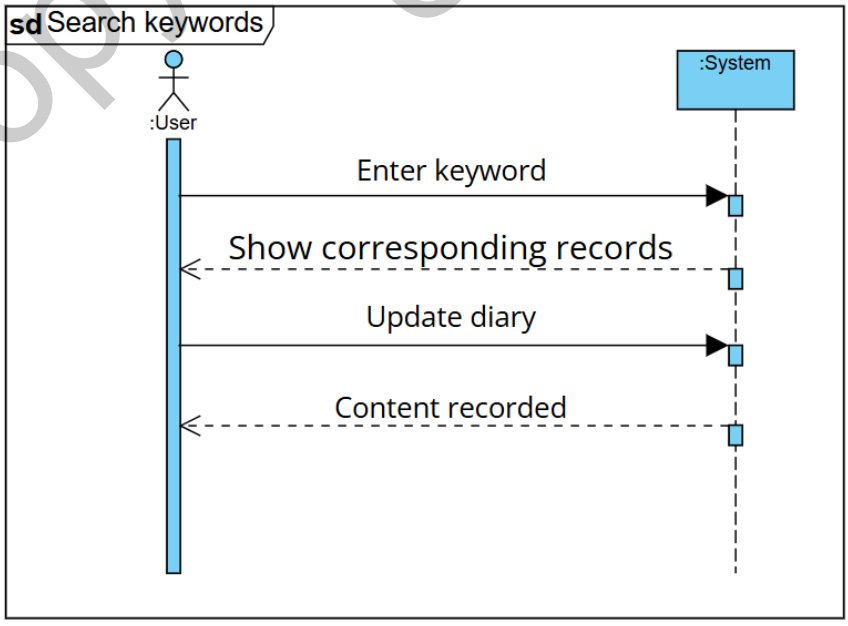


Figure 3 Record emotions and diary(Subfigure 1)



Copyright@FTSM UKM

Figure 4 Search keywords(Subfigure 2)

3.2.4 Activity diagram

Activity Diagrams: These diagrams depict the step-by-step flow of processes within a system.

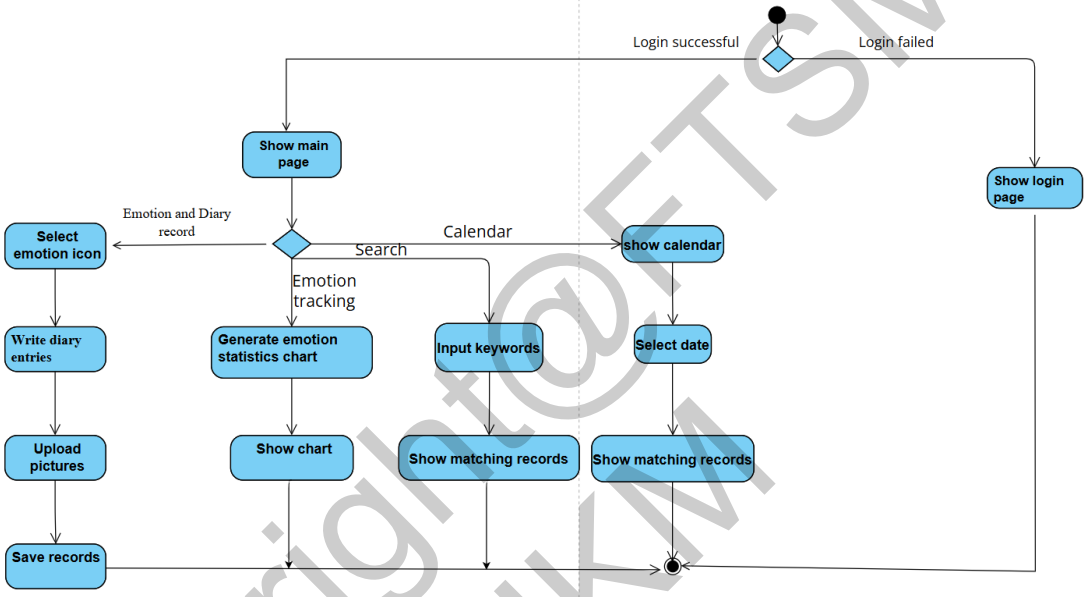


Figure 5 System processing steps

4.0 RESULTS

4.1 SYSTEM COMPONENTS

i. **User Registration**

Copyright@FTSM UKM

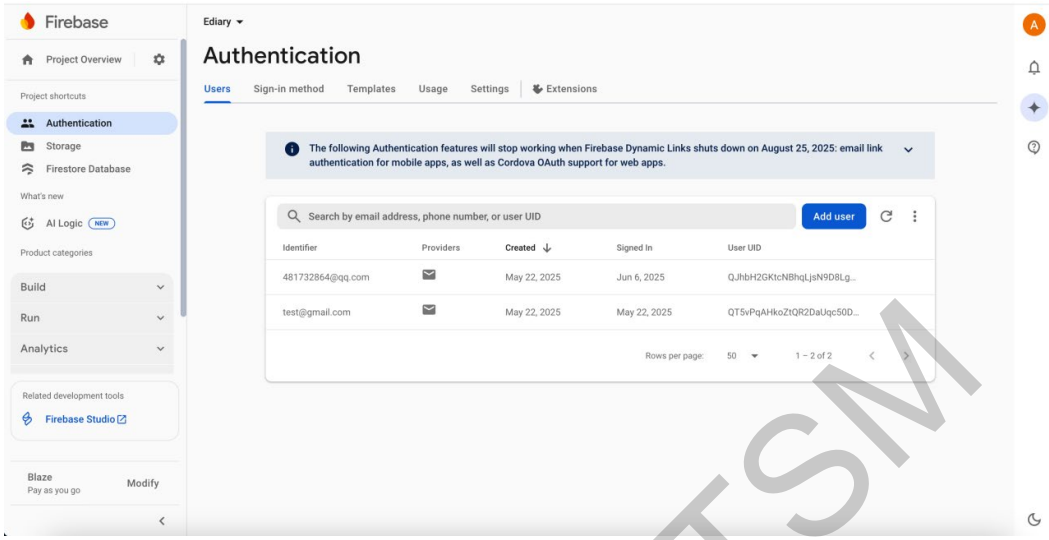


Figure6 Firebase-Authentication

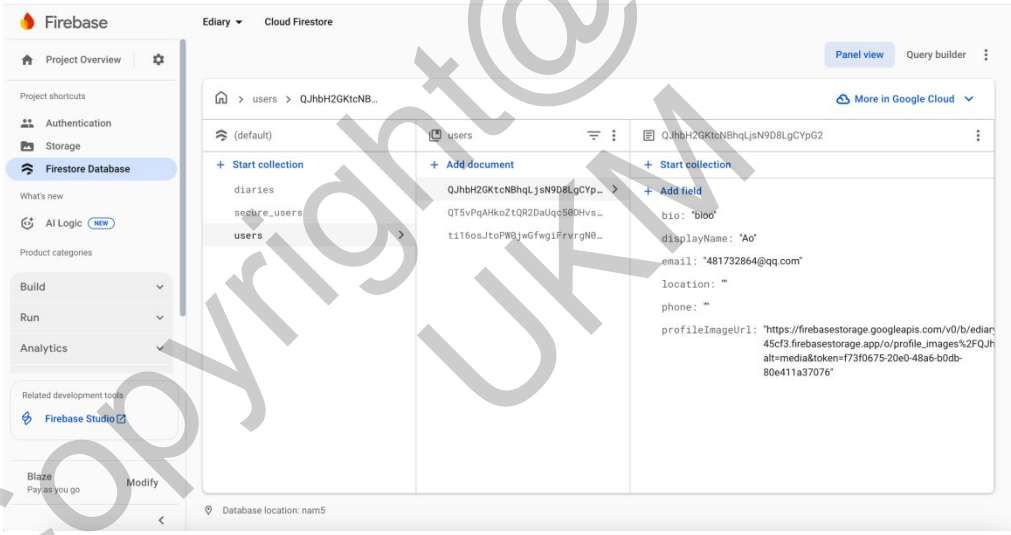


Figure7 Firebase-user set in Firestore

Copyright@FTSM UKM

```
suspend fun registerWithEmailPassword(
    name: String,
    email: String,
    password: String,
    onSuccess: (FirebaseUser) -> Unit,
    onError: (Exception) -> Unit
) {
    try {
        val result = auth.createUserWithEmailAndPassword(email, password).await()
        result.user?.let { user ->

            val profileUpdates = UserProfileChangeRequest.Builder()
                .setDisplayName(name)
                .build()

            user.updateProfile(profileUpdates).await()

            val saveResult = userRepository.saveSecureUserData(
                userId = user.uid,
                email = email,
                name = name,
                profilePhotoUrl = user.photoUrl?.toString()
            )

            if (saveResult.isSuccess) {
                onSuccess(user)
            } else {
                Log.e( tag: "FirebaseAuthManager",  msg: "Failed to save encrypted user data", saveResult.exceptionOrNull())
                onSuccess(user)
            }
        } ?: run {
            onError(Exception("Registration failed: User is null"))
        }
    } catch (e: Exception) {
        Log.e( tag: "FirebaseAuthManager",  msg: "Registration failed", e)
        onError(e)
    }
}
```

Figure8 Firebase Signup Integration Code

The code snippet (Figure8) first calls the createUserWithEmailAndPassword method of Firebase Authentication (Figure6) to create a basic account through the registerWithEmailAndPassword function; then uses UserProfileChangeRequest.Builder() to build a profile update request containing the display name, and synchronizes it to the user authentication information through user.updateProfile(profileUpdates).await() to complete the core process of identity authentication. Subsequently, the code encrypts the user's key information (UID, email, name, avatar link) and stores it in the users collection of Firestore Database (Figure7) through the userRepository.saveSecureUserData method, realizing the associated storage of identity information and business data. This process not only utilizes the security authentication mechanism of Firebase Authentication, but also

Copyright@FTSM UKM

expands the user data model through the structured storage capability of Firestore, forming a complete closed loop from identity authentication to data persistence.

ii. User Login

```
suspend fun loginWithEmailPassword(
    email: String,
    password: String,
    onSuccess: (FirebaseUser) -> Unit,
    onError: (Exception) -> Unit
) {
    try {
        val result = auth.signInWithEmailAndPassword(email, password).await()
        result.user?.let { user ->
            onSuccess(user)
        } ?: run {
            onError(Exception("Login failed: User is null"))
        }
    } catch (e: Exception) {
        Log.e( tag: "FirebaseAuthManager",  msg: "Login failed", e)
        onError(e)
    }
}
```

Figure9 Firebase login authentication

The code snippet (Figure9) calls Firebase Authentication (Figure6) through auth.signInWithEmailAndPassword to complete the email and password login. Based on the login result, if successful, the FirebaseUser is returned through onSuccess, and if failed, the exception is captured and fed back through onError.

iii. User Information Encryption (AES-256 encryption)

a) AES-256 key management

Copyright@FTSM UKM

```
private fun getOrCreateSecretKey(): SecretKey {
    val keyStore = KeyStore.getInstance(ANDROID_KEYSTORE)
    keyStore.load( param: null)

    // 检查密钥是否已存在
    if (!keyStore.containsAlias(KEY_ALIAS)) {
        // 创建新密钥
        val keyGenerator = KeyGenerator.getInstance(
            KeyProperties.KEY_ALGORITHM_AES,
            ANDROID_KEYSTORE
        )

        val keyGenParameterSpec = KeyGenParameterSpec.Builder(
            KEY_ALIAS,
            purposes: KeyProperties.PURPOSE_ENCRYPT or KeyProperties.PURPOSE_DECRYPT
        )
            .setBlockModes(KeyProperties.BLOCK_MODE_GCM)
            .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_NONE)
            .setKeySize(KEY_SIZE)
            .build()

        keyGenerator.init(keyGenParameterSpec)
        return keyGenerator.generateKey()
    } else {
        // 获取已存在的密钥
        val entry = keyStore.getEntry(KEY_ALIAS, protParam: null) as KeyStore.SecretKeyEntry
        return entry.secretKey
    }
}
```

Figure10 AES-256 key management implementation

This code (Figure10) implements a secure AES-256 key management mechanism through the Android KeyStore. The getOrCreateSecretKey() method first checks whether the key with the specified alias exists. If not, it uses KeyGenerator to generate a 256-bit AES key, conFigure4.s the GCM mode and no padding, and stores it in the Android KeyStore, using the system security mechanism to protect the key; if the key already exists, it is directly obtained from the KeyStore. This mechanism prevents key export through hardware-level protection, combined with the encryption and authentication functions of the GCM mode, to ensure the security and integrity of data encryption, providing reliable underlying support for applications to store user sensitive information.

b) Encrypt

Copyright@FTSM UKM

```kotlin
fun encrypt(plainText: String): String {
    try {
        val cipher = Cipher.getInstance(TRANSFORMATION)
        cipher.init(Cipher.ENCRYPT_MODE, getOrCreateSecretKey())

        val encryptedBytes = cipher.doFinal(plainText.toByteArray(Charsets.UTF_8))
        val encodedEncryptedText = Base64.encodeToString(encryptedBytes, Base64.DEFAULT)
        val encodedIV = Base64.encodeToString(cipher.iv, Base64.DEFAULT)

        return "$encodedEncryptedText:$encodedIV"
    } catch (e: Exception) {
        // 在生产环境中应更好地处理这个错误
        e.printStackTrace()
        throw e
    }
}
```

Figure11 AES-256-Encrypt

This function(Figure11) implements the AES-GCM encryption process: get the key through getOrCreateSecretKey(), initialize Cipher to encryption mode, convert the plaintext into a byte array and encrypt it. The generated initialization vector (IV) and the encrypted byte array are Base64 encoded respectively, concatenated with ":" and returned. This format facilitates the extraction of IV and ciphertext during subsequent decryption, ensuring that a unique IV is used for each encryption, enhancing security

c) Dencrypt

```kotlin
fun decrypt(encryptedText: String): String {
    try {
        val parts = encryptedText.split( ...delimiters: ":")
        if (parts.size != 2) {
            throw IllegalArgumentException("Invalid encrypted text format")
        }

        val encodedEncryptedText = parts[0]
        val encodedIV = parts[1]

        val encryptedBytes = Base64.decode(encodedEncryptedText, Base64.DEFAULT)
        val iv = Base64.decode(encodedIV, Base64.DEFAULT)

        val cipher = Cipher.getInstance(TRANSFORMATION)
        val spec = GCMParameterSpec(TAG_LENGTH, iv)
        cipher.init(Cipher.DECRYPT_MODE, getOrCreateSecretKey(), spec)

        val decryptedBytes = cipher.doFinal(encryptedBytes)
        return String(decryptedBytes, Charsets.UTF_8)
    } catch (e: Exception) {
        // 在生产环境中应更好地处理这个错误
        e.printStackTrace()
        throw e
    }
}
```

Figure12 AES-256-Dencrypt

Copyright@FTSM UKM

This function(Figure12) handles the decryption of encrypted text: first split the input into Base64 strings of ciphertext and IV by ":", decode it, get the key through getOrCreateSecretKey(), and initialize Cipher to decryption mode with IV. Call doFinal() to decrypt the byte array and convert it to the original string to ensure data integrity and confidentiality.

iv. UI Components

```
fun DiaryDatePickerDialog(
    datePickerState: androidx.compose.material3.DatePickerState,
    onDismiss: () -> Unit,
    onConfirm: () -> Unit
) {
    DatePickerDialog(
        onDismissRequest = onDismiss,
        confirmButton = {
            TextButton(onClick = onConfirm) {
                Text( text: "OK")
            }
        },
        dismissButton = {
            TextButton(onClick = onDismiss) {
                Text( text: "Cancel")
            }
        }
    ) {
        DatePicker(state = datePickerState)
    }
}
```

Figure13 DiaryDatePickerDialog Component

DiaryDatePickerDialog(Figure13) manages the date selection state through DatePickerState. Based on the Material 3 dialog box specification, it uses onConfirm/onDismiss to connect "user selection of date" with "diary filtering logic". It is the interactive core for realizing accurate date filtering, allowing users to easily trigger "specified date diary query".

Copyright@FTSM UKM

```
FlowRow(
    horizontalArrangement = Arrangement.SpaceEvenly,
    verticalArrangement = Arrangement.spacedBy(16.dp),
    modifier = Modifier
        .fillMaxWidth()
        .padding(vertical = 16.dp)
) {
    Mood.values().forEach { mood ->
        MoodItem(
            mood = mood,
            isSelected = mood == selectedMood,
            onClick = { onMoodSelected(mood) }
        )
    }
}

TextButton(
    onClick = onDismiss,
    modifier = Modifier.align(Alignment.End)
```

Figure14 MoodPickerDialog Component

This code (Figure14) is in the custom component MoodPickerDialog. The core layout of the mood selector is implemented through the FlowRow component. Mood enumeration is traversed to dynamically generate MoodItem. The highlight effect is controlled by the isSelected state. When clicked, the selected mood is passed through the onMoodSelected callback, allowing users to filter diaries corresponding to the mood based on the emotional state (such as HAPPY, SAD).

v. Diary Management

a) Diary creation

```
val entry = DiaryEntry(
    id = diaryId ?: "",
    content = _content.value,
    date = _date.value,
    isBookmarked = _isBookmarked.value,
    isDraft = asDraft,
    weather = _selectedWeather.value?.name,
    mood = _selectedMood.value?.name,
    imageUrls = _existingImageUrls.toList()
)

// 使用Context保存日记并上传图片
val result = repository.saveDiaryEntry(
    context = context,
    entry = entry,
    images = selectedImages
)
```

Figure15 Diary creation(ViewModel layer)

Copyright@FTSM UKM

This code(Figure15) is responsible for assembling the diary data model and triggering the storage process. It encapsulates the user's input content, date, mood and other status information into the DiaryEntry object, and then passes it to the data layer through repository.saveDiaryEntry. This process completes the conversion from "user interaction data" to "storable data model" and is the data preparation stage of diary creation.

```
// 创建更新后的日记对象
val updatedEntry = entry.copy(
    userId = userId,
    imageUrls = imageUrls
)

// 保存到Firestore
val docRef = if (entry.id.isEmpty()) {
    // 新建日记
    diariesCollection.add(updatedEntry.toMap()).await()
} else {
    // 更新日记
    diariesCollection.document(entry.id).set(updatedEntry.toMap()).await()
    diariesCollection.document(entry.id)
}
```

Figure16 Diary creation(Repository layer)

This code(Figure16) focuses on the actual storage logic. Depending on whether the diary ID exists, it chooses to call add() to create a new document or set() to update the document, and directly operates Firestore to complete data persistence. It handles the branch logic of creation and update, ensuring that the diary data is accurately written to the database.

The two code(Figure15,16) segments work together. The first segment converts the user input into a DiaryEntry and triggers a storage request. The second segment receives the request and performs the corresponding storage operation based on the ID status. The two segments are connected through the repository to form a complete link of "data preparation → data storage", and together realize the core process from the user clicking "Save" to the diary being successfully stored in Firebase.

Copyright@FTSM UKM

b) Diary delete

```
fun deleteDiaryEntry(diaryId: String) {
    viewModelScope.launch {
        _isLoading.value = true

        try {
            // 保存当前日记状态，用于在发布事件时提供额外信息
            val currentDiary = when (val state = _diaryState.value) {
                is DiaryState.Success -> state.diary
                else -> null
            }

            val hasImages = currentDiary?.imageUrls?.isNotEmpty() == true
```

Figure17 Diary delete(ViewModel layer)

This code(Figure17) is the starting point of the deletion operation, which switches to background processing through coroutines. It obtains diary information from the current state, determines whether it contains images, and calls Repository to perform the deletion. The core logic is to prepare the deletion conditions and trigger data layer operations, while updating the UI status.

```
suspend fun deleteDiaryEntry(diaryId: String): Result<Unit> {
    return try {
        Log.d( tag: "DiaryRepository",  msg: "开始删除日记: $diaryId")

        // 先获取日记数据，包括图片URLs
        val entry = getDiaryEntry(diaryId)
        if (entry == null) {
            Log.e( tag: "DiaryRepository",  msg: "找不到要删除的日记: $diaryId")
            return Result.failure(Exception("Diary entry not found"))
        }

        val hasImages = entry.imageUrls.isNotEmpty()
        Log.d( tag: "DiaryRepository",  msg: "日记有 ${entry.imageUrls.size} 张图片")

        // 先删除Firestore中的日记文档
        diariesCollection.document(diaryId).delete().await()
        Log.d( tag: "DiaryRepository",  msg: "已删除Firestore中的日记文档")

        // 如果有图片，删除Storage中的所有图片
        if (hasImages) {
            Log.d( tag: "DiaryRepository",  msg: "开始删除 ${entry.imageUrls.size} 张图片")

            // 使用协程等待所有图片删除任务完成
            val deleteTasks = entry.imageUrls.map { url ->
                try {
                    Log.d( tag: "DiaryRepository",  msg: "删除图片: $url")
                    val imageRef = storage.getReferenceFromUrl(url)
                    imageRef.delete().await()
                    Log.d( tag: "DiaryRepository",  msg: "图片删除成功: $url")
                    true
```

Copyright@FTSM UKM

Figure18 Diary delete(Repository layer)

This code(Figure18) is the core implementation of the delete operation, which interacts directly with Firebase. First, get the diary entity, and then perform a two-step deletion: 1) delete the document in Firestore; 2) if there are Figure4.s, asynchronously delete all Figure4.s in Storage. The core logic is to ensure the atomic deletion of documents and associated resources to avoid data residue.

The ViewModel layer receives the UI deletion request, prepares the necessary parameters (such as the diary ID and whether there are any Figure4.s) and calls the Repository; the Repository layer first deletes the Firestore document and then cleans up the associated Figure4.s based on these parameters. The two work together through the MVVM architecture to achieve a complete closed loop from user interaction to data persistence deletion.

c) Diary Figure4. upload

```
val imageUrls = if (images.isNotEmpty()) {
    Log.d( tag: "DiaryRepository",  msg: "Start processing ${images.size} pictures")

    // 先压缩图片
    val compressedImages = ImageCompressor.compressImages(context, images)
    Log.d( tag: "DiaryRepository",  msg: "Image compression completed")

    // 上传压缩后的图片
    val newImageUrls = uploadImages(userId, compressedImages) { progress ->
        // 上传进度回调可以在这里处理
    }
```

Figure19 Image upload pre-processing

This code(Figure19) calls ImageCompressor.compressImages to compress the selected local image to optimize upload performance. After compression, the compressed image is uploaded through the uploadImages method to prepare for the subsequent association of the image with the diary. This is a transitional step to connect local image selection with Firebase storage.

Copyright@FTSM UKM

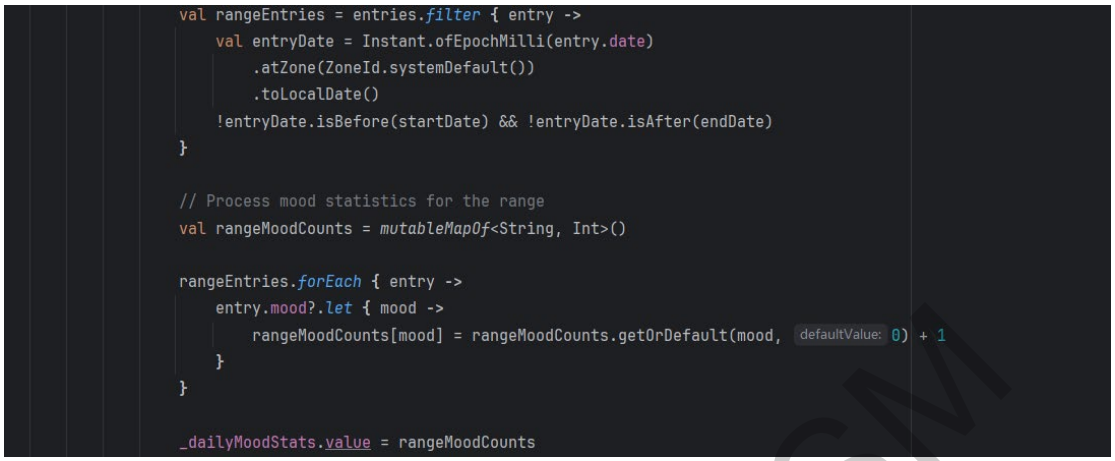
```
images.forEachIndexed { index, uri ->
    try {
        val imageName = "diary_images/${userId}/${UUID.randomUUID()}"
        val imageRef = storage.reference.child(imageName)

        Log.d( tag: "DiaryRepository",  msg: "Start uploading pictures ${index + 1}/${totalImages}")

        // 创建上传任务并监听进度
        val uploadTask = imageRef.putFile(uri)

        // 添加进度监听
        uploadTask.addOnProgressListener { taskSnapshot ->
            val progress = taskSnapshot.bytesTransferred.toFloat() / taskSnapshot.totalByteCount
            val overallProgress = (index.toFloat() + progress) / totalImages
            onProgress?.invoke(overallProgress)

            // 记录上传进度
            if (progress % 0.25f < 0.01f) { // 每25%记录一次
                Log.d( tag: "DiaryRepository",  msg: "picture ${index + 1} Upload progress: ${(progress * 100).toInt()}%")
            }
        }

        // 等待上传完成
        uploadTask.await()

        // 获取下载URL
        val downloadUrl = imageRef.downloadUrl.await().toString()
        imageUrls.add(downloadUrl)
```

Figure20 Firebase Image Upload Core

This code (Figure20) traverses the images to be uploaded, generates a unique storage path for each image (combining the user ID and random UUID), creates a Firebase Storage upload task, adds a progress monitor, calculates the overall upload progress in real time, and feedbacks the progress through onProgress. After the upload is complete, the image download URL is obtained and collected. These download URLs will be associated with the diary entity later. This is the key logic for implementing the image from local to Firebase storage and generating accessible links.

vi. Statistical Analysis

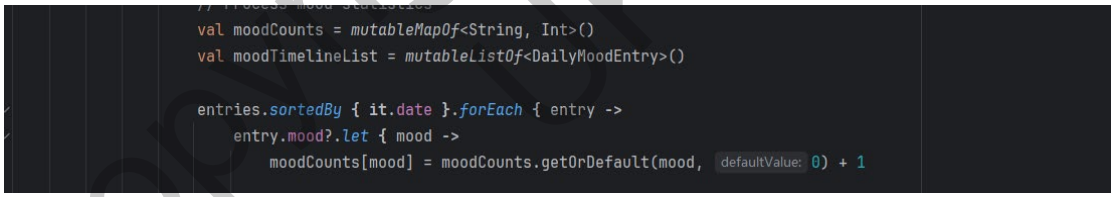
Copyright@FTSM UKM



```
val rangeEntries = entries.filter { entry ->
    val entryDate = Instant.ofEpochMilli(entry.date)
        .atZone(ZoneId.systemDefault())
        .toLocalDate()
    !entryDate.isBefore(startDate) && !entryDate.isAfter(endDate)
}

// Process mood statistics for the range
val rangeMoodCounts = mutableMapOf<String, Int>()

rangeEntries.forEach { entry ->
    entry.mood?.let { mood ->
        rangeMoodCounts[mood] = rangeMoodCounts.getOrDefault(mood, defaultValue: 0) + 1
    }
}

_dailyMoodStats.value = rangeMoodCounts
```

Figure21 Data filtering

This code(Figure21) implements "emotion statistics for a specified time range": first filter out the diaries in the target range by date (rangeEntries), then traverse these diaries, use mutableMap to count the number of occurrences of each emotion, and finally store the results in _dailyMoodStats to provide the UI with emotion distribution data for that time period.



```
val moodCounts = mutableMapOf<String, Int>()
val moodTimelineList = mutableListOf<DailyMoodEntry>()

entries.sortedBy { it.date }.forEach { entry ->
    entry.mood?.let { mood ->
        moodCounts[mood] = moodCounts.getOrDefault(mood, defaultValue: 0) + 1
```

Figure22 Global mood statistics

This code(Figure22) implements "global mood statistics": after sorting all diaries by date, it traverses and counts the number of occurrences of each mood through mutableMap (stored in moodCounts), providing full data support for subsequent analysis (such as mood trends and recommendations).

vii. Search and Calendar Filtering

Copyright@FTSM UKM

```
val filteredDiaries = remember(diaries, viewModel.searchQuery.value, viewModel.selectedDate.value) {
    diaries.filter { entry ->

        val matchesSearch = viewModel.searchQuery.value.isEmpty() ||
                entry.content.contains(viewModel.searchQuery.value, ignoreCase = true)


        val matchesDate = viewModel.selectedDate.value?.let { selectedDate ->
            val entryDate = Instant.ofEpochMilli(entry.date)
                .atZone(ZoneId.systemDefault())
                .toLocalDate()
            entryDate == selectedDate
        } ?: true

        matchesSearch && matchesDate
    }
}
```

Figure23 Search and calendar filtering logic

This code is the core filtering logic of the search and calendar filtering function: remember monitors the changes of diaries (diary list), searchQuery (search keyword), and selectedDate (filter date), first determines whether the diary content matches the search term (matchesSearch, default match when empty keyword), then converts the diary date and determines whether it matches the selected date (matchesDate, default match when no selected date), and finally implements diary filtering with the dual conditions of "search content + calendar date" through matchesSearch && matchesDate, accurately filtering out diary entries that meet user needs.

4.2 TEST CASE DESIGN

This chapter will provide an overall introduction to each functional module's test case design. The test cases will primarily test whether the system is behaving as expected, ensuring that the main functional modules can run stably in accordance with predetermined requirements. Each test case will include test goals, procedures, input parameters, and expected outputs, and verify the function according to actual users' requirements.

Copyright@FTSM UKM

4.2.1 User Registration and Login Module

Table 1 Use Case TC-001

Use Case TC-001	**Successful User Registration**
Test Objective	Verify that the user can successfully register an account with a valid email and password.
Pre-conditions	The user is not registered and is connected to the network.
Test steps	1. Open the registration page. 2. Enter a valid user name, email address and password. 3. Click the 'Register' button.
Input data	User name: test_user, e-mail: test@domain.com, password: password123.
Expected result	Registration is successful, and you are redirected to the login page.
Actual result	Registration succeeded, jumped to the login page. Actual result: As expected.
Test Status	Passed

Table 2 Use Case TC-002

Use Case TC-002	**User Login Successful**
Test Objective	To verify that the user can successfully login with the correct email and password. Pre-conditions: The user has already registered an account.
Pre-conditions	The user has registered an account and is connected to the network.
Test steps	1. Open the login page. 2. Enter the registered email and password. 3. Click the 'Login' button.
Input data	Email: test@domain.com, password: password123.
Expected result	Successful login and entry into the main application interface.
Actual result	As expected.
Test Status	Passed

Table 3 Use Case TC-003

Use Case TC-003	**User Exit Login Successful**
Test Objective	To verify that the user can log out successfully.
Pre-conditions	The user is logged in.
Test steps	1. Click the 'Logout' button. 2. Confirm the logout operation.

Copyright@FTSM UKM

Input data	None.
Expected result	The user successfully logs out and returns to the login page.
Actual result	The user logs out successfully and returns to the login page.
Test Status	Passed

4.2.2 Diary Management Module

Table 4 Use Case TC-004

Use Case TC-004	**User Diary Created Successfully**
Test Objective	To verify that a user can successfully create and save a diary.
Pre-conditions	The user is logged in and enters the main interface.
Test steps	1. Select 'Create New Diary' button. 2. Input diary content and select the emotion tag. 3. Click 'Save' button.
Input data	Content: 'I am in a good mood today', emotion tag: happy.
Expected result	The diary is successfully saved and displayed in the diary list.
Actual result	Same as expected.
Test Status	Passed

Table 5 Use Case TC-005

Use Case TC-005	**User diary deleted successfully**
Test Objective	To verify that a user can successfully delete a diary that he/she has already created.
Pre-conditions	The user has at least one diary.
Test steps	1. Select the diary to be deleted. 2. Click the 'Delete' button. 3. Confirm the deletion.
Input data	Diary ID: 12345.
Expected result	The diary is deleted and removed from the list.
Actual result	The diary is deleted and removed from the list.
Test Status	Passed

Copyright@FTSM UKM

Table 6 Use Case TC-006

Use Case TC-006	**User Modifies Diary Successfully**
Test Objective	To verify that a user can successfully modify the contents of a saved diary.
Pre-conditions	The user has created and saved at least one diary.
Test steps	1. Select the diary to be modified. 2. Modify the diary content and save it.
Input data	Modified content: 'I am in a very good mood today'.
Expected result	The modified diary content is successfully saved.
Actual result	The modified diary content is successfully saved.
Test Status	Passed

Table 7 Use Case TC-007

Use Case TC-007	**User Uploads Pictures Successfully**
Test Objective	To verify that a user can upload a picture and associate it with a diary entry. Pre-conditions: The user has created at least one diary entry.
Pre-conditions	The user has created at least one diary.
Test steps	1. Create a new diary entry. 2. Select Upload Picture and Save.
Input data	Picture: an emotionally relevant photo.
Expected result	The picture is successfully uploaded and displayed in the diary.
Actual result	The picture is successfully uploaded and displayed in the diary.
Test Status	Passed

4.2.3 Sentiment Analysis Module

Table 8 Use Case TC-008

Use Case TC-008	**Sentiment Trend Graph Generation Correct**
Test Objective	To verify that the system is able to generate accurate sentiment trend graphs based on diary entries.
Pre-conditions	The user has created multiple diaries with sentiment tags.
Test steps	1. Click the 'Sentiment Trend' button to view the graph. 2. Verify the generated sentiment trend graph.
Input data	Emotion tags: Happy, Tired, Excited.

Copyright@FTSM UKM

Expected result	The emotion trend graph accurately shows the distribution of each emotion.
Actual result	Consistent with expectations.
Test Status	Passed

4.2.4 Data Security Module

Table 9 Use Case TC-009

Use Case TC-009	**Data Encryption and Decryption**
Test Objective	To verify that user data is correctly encrypted and stored, and that it can be correctly decrypted.
Pre-conditions	User is registered and logged in, data encryption is enabled.
Test steps	1. Create a diary and save it. 2. Get the encrypted stored data in the background and decrypt it. 3. Check whether the decrypted data is complete.
Input data	Diary content: 'I feel good today'.
Expected result	The data is successfully encrypted and decrypted, and the content of the diary is consistent with the input.
Actual result	The data is successfully encrypted and decrypted, and the content of the diary is consistent with the input.
Test Status	Passed

4.2.5 Search and Calendar Module

Table 10 Use Case TC-010

Use Case TC-010	**Search Function (Search by Date)**
Test Objective	To verify that users can search for relevant diaries by date.
Pre-conditions	The user has created several diaries with different date records.
Test steps	1. Open the search interface. 2. Select a date range and perform a search.
Input data	Date: 28th June 2025.
Expected result	Display all diary entries for that date.
Actual result	Display all diary entries for that date.
Test Status	Passed

Copyright@FTSM UKM

4.6 TEST EXECUTION AND RESULT

4.6.1 TEST PROGRESS DESCRIPTION

The testing will be done in phases, with the test cases of every functional module being executed one after the other. Any issues that will be revealed at the end of every phase will be addressed based on the test result. The outcomes generated for every function will be recorded and matched with the expected results to ensure that all functions meet the design's requirements. Throughout the testing phase, special emphasis will be put on the stability and security of the data in the system, so the system would be running in good performance and stability even when the load and concurrency are high.

4.6.2 Test Results Summary

By running the aforementioned test cases, most of the functional modules have yielded results that are in line with expectations. The system is stable, and the key functions are comparatively error-free. Some minor issues concerning device compatibility were observed, including charts not loading completely on certain lower-version Android devices. Apart from this, the data encryption and decryption processes passed the tests without any issues, the trend graph of emotions is correctly generated, and the search and calendar operations demonstrate good speed and accuracy.

4.7 ISSUE SUMMARY AND FIXES

Table 11 BUG

No.	Problem Description	Discovery Stage	Remediation Method	Result

Copyright@FTSM UKM

BUG01	Some users cannot login successfully when using non-standard passwords (e.g. special characters).	User Login Module	Adjusted the password encryption module and enhanced the support for special characters to ensure that users can log in properly with any type of password	Fixed
BUG02	Chart display is not normal on some devices	Chart Rendering Module	Adjusted the chart rendering code to ensure compatibility with all Android devices	Fixed

5.0 CONCLUSION

EDiary is a personal journal app designed to help users record daily experiences, track moods, and reflect on their emotional journeys. The project encompasses features like diary creation with mood and weather tagging, search and date - based filtering, mood analysis visualizations, and user profile management. Over the development process, it evolved from initial concept to a functional system addressing personal journaling needs, with a focus on usability and emotional insight.

Copyright@FTSM UKM