Final Experience Report

Cover Page

- **Student Name**: Michael Roddy
- **Student Number**: A00326407
- **Assessment Component**: Final Work Placement Report

Introduction

The purpose of this report is to reflect on my work placement experience at the **Software Research Institute**, where I worked remotely between **June 2025 and August 2025**. The placement was an important opportunity to put into practice the technical and theoretical knowledge I had acquired through the MSc in Software Design with Cloud Native Computing.

During this period, I contributed to the development of an analytics service for an application called **MoodTrendz**. This app is designed to help users track their moods and medication intake over time. The analytics service was developed to provide deeper insights, such as identifying patterns, detecting mood trends, and comparing moods across different time periods. These features were especially valuable for doctors and healthcare professionals, who could then use the data to make more informed decisions about patient care and medication dosage adjustments.

The MoodTrendz application followed a **monolithic architecture**, with a **React Native frontend**, a **Spring Boot backend**, and a **MySQL database**. My main role was to design and implement an analytics component, which began as part of the monolithic backend but was structured in such a way that it could evolve into a standalone **microservice**. n.b Future Work.

This report will discuss literature and background research, my responsibilities and duties, detail the technical achievements I made, examine the challenges faced and how I resolved them, and provide a broader evaluation of what I learned from working in the software development industry. It will conclude with reflections on how this experience will shape my future professional development.

Background

Journaling and Mental Health

Journaling has long been recognised as a beneficial practice for self-reflection, emotional regulation, and stress management. Psychological studies suggest that expressive writing can reduce symptoms of anxiety and depression, while also improving overall well-being. By recording daily thoughts and feelings, individuals gain greater awareness of emotional triggers and behavioural patterns. This process creates a feedback loop: the act of writing provides immediate relief, while long-term analysis of entries enables recognition of recurring themes or stressors.

Digital journaling extends these benefits by reducing barriers to entry. Mobile applications remove the need for pen and paper, allow users to log experiences quickly, and often incorporate features such as reminders, data visualisation, and mood analysis. For younger generations in particular, digital-first solutions feel more

natural and accessible, ensuring consistency of use. However, digital tools also introduce challenges, such as data privacy, user engagement, and the risk of overwhelming users with complex features. Balancing ease of use with functionality becomes a central design concern in this space.

Mood Tracking and Emotional Awareness

While traditional journaling focuses on narrative expression, mood tracking offers a structured way to capture emotional states in a more quantifiable manner. Research in affective computing highlights the importance of tracking moods over time, as this data can reveal correlations between lifestyle habits and mental health outcomes. For example, studies demonstrate links between disrupted sleep patterns and increased irritability, or between physical activity and improved happiness ratings.

Mood-tracking systems typically allow users to select from a predefined list of emotions (e.g., happy, sad, anxious), sometimes supplemented with an intensity scale. More advanced systems integrate contextual metadata such as location, activity, or social interactions. This creates richer datasets that enable personalised insights, but at the cost of increased complexity in both data modelling and user experience.

Existing Applications and Industry Approaches

Several established applications provide useful benchmarks for this project:

- **Daylio**: A widely used mood-tracking app that allows users to log daily moods alongside activities. Its main strength lies in simplicity and minimal friction, users can complete an entry in seconds. However, Daylio's data export features are limited, and the mood model is relatively rigid, making it less suitable for users who want to customise emotions or add contextual depth or analytics.
- **Moodnotes**: This application blends journaling with cognitive behavioural therapy (CBT) principles, encouraging users to reflect on thinking patterns. While clinically informed, Moodnotes is more prescriptive, offering guidance rather than flexible self-expression. This makes it valuable for structured reflection but less adaptable to individual needs.

The project undertaken here attempts to bridge the gap by combining the ease of journaling with structured mood tagging, offering users both flexibility and data-driven analytical insights.

Data Modelling in Mental Health Applications

A technical challenge unique to this domain lies in the representation of emotional data. Unlike structured medical data (e.g., blood pressure readings, blood glucose etc..), emotions are inherently subjective, multidimensional, and often co-occurring. For instance, a person may feel both excited and anxious before a significant life event, making single-label models insufficient.

In database design, this necessitates many-to-many relationships between journal entries and mood tags. A single journal entry may reference multiple moods, while the same mood may appear across many entries. The chosen schema uses a junction table (`journal_entry_mood_tag`) to resolve this relationship, with an added intensity attribute to capture strength of feeling. This approach enables nuanced data analysis, such as comparing mood intensity across time periods.

Other possible modelling approaches include embedding moods as JSON arrays within journal entries or adopting a document-oriented database (e.g., MongoDB). While these may simplify schema design, they complicate querying and aggregation. In contexts where analytics are central, a normalised relational design remains the more scalable solution.

Ethical and Privacy Considerations

Mental health applications process some of the most sensitive categories of personal data. Journaling entries, mood logs, and medication records could all reveal intimate details about a user's psychological state. As a result, ethical handling of data is paramount. General Data Protection Regulation (GDPR) requirements in the European Union mandate explicit consent, minimisation of data retention, and the right to erasure. Similar principles apply in other jurisdictions under frameworks such as HIPAA in the United States.

Industry surveys indicate that lack of trust is a significant barrier to adoption of digital health tools. Users are often reluctant to record sensitive information if they are uncertain how it will be stored, shared, or monetised. To address this, applications must communicate transparently about data use, implement encryption both in transit and at rest, and offer users control over their records.

Summary of Gaps and Contribution

The literature and industry review highlights three main gaps:

1. **Flexibility vs Simplicity**: Many existing apps lean heavily towards one or the other, whereas users often need a middle ground.
2. **Multi-Mood Representation**: Single-label models fail to capture the complexity of human emotions, yet few tools offer rich support for multi-mood entries with intensity tracking.
3. **Data Ethics and Control**: Despite the sensitivity of mood and health data, many applications provide limited transparency or user autonomy over their records.

The project seeks to contribute by addressing these gaps. Through a relational database design, it supports nuanced multi-mood modelling. Through a streamlined journaling interface, it retains usability without sacrificing analytical depth.

Responsibilities and Duties

My responsibilities and duties during the placement covered mostly **technical development tasks**.

Summary of Responsibilities

- Designing and developing RESTful APIs in **Java (Spring Boot)** for the MoodTrendz analytics module.
- Designing and implementing a **MySQL schema** to support mood and medication data storage.
- Handling **data modelling**, to implement a normalised database schema for efficient storage.
- Implementing **trend analysis, pattern detection, and comparison algorithms** to process mood data.
- Ensuring the analytics code was **containerised using Docker** for consistency across environments.
- performing **manual tests (Postman)** to ensure API functionality.

- Writing **technical documentation** for other developers and stakeholders to understand the analytics APIs.
- Collaborating with supervisors to ensure the analytics module aligned with the work placement goals.

Day-to-Day Duties

On a day-to-day basis, my work typically involved:

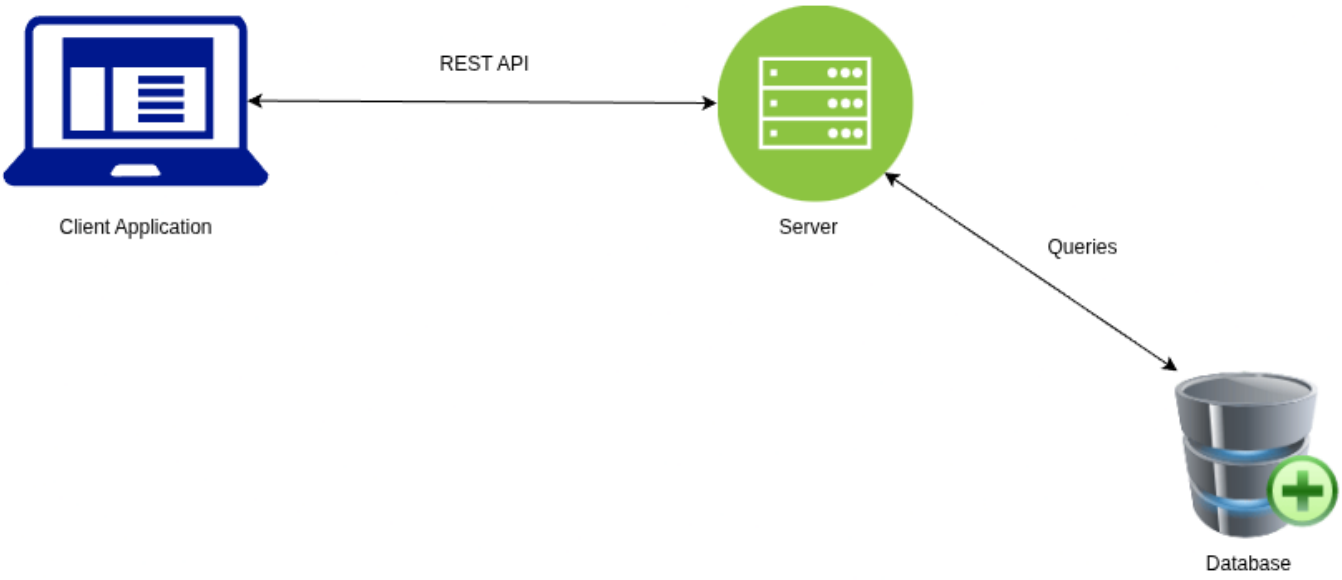
1. **Weekly stand-ups**: Attending virtual meetings with the supervisor to provide updates, discuss blockers, and plan priorities.
2. **Coding tasks**: Implementing new features for the analytics API, debugging issues, and refactoring existing code.
3. **Testing and validation**: Manually testing endpoints through Postman.
4. **Documentation**: Updating API specifications, writing READMEs, and maintaining clarity for future developers.
5. **Demonstrations**: Presenting progress to supervisor and demonstrating the analytics features.

Architecture Overview

The client–server architecture is a common three-tier model used in modern applications. The client (e.g., a mobile app or web browser) provides the user interface and sends requests. The server acts as the middle layer, handling business logic, processing client requests, and enforcing security rules. The database stores and manages persistent data, which the server queries and updates as needed. This separation improves scalability, maintainability, and security, since each layer focuses on its own responsibilities while working together to deliver a seamless user experience.

The client application in this case is a React Native (cross-platform) mobile application, with a Java Spring Boot backend REST API that quireies a MySQL database.

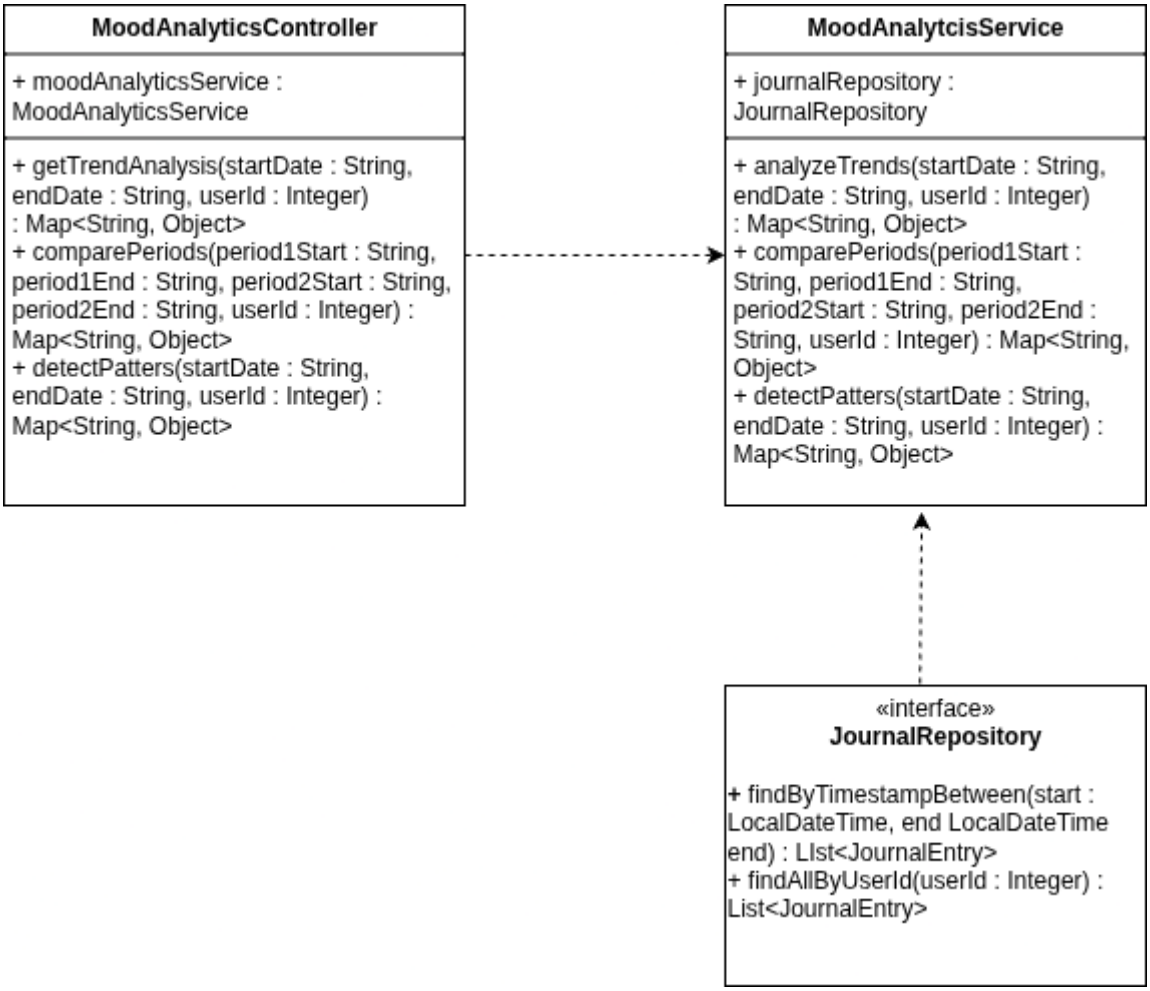
Client Server Model



Class Diagrams

Here is an example of the structure of the Java analytics service and interaction between classes.

- `MoodAnalyticsController` (entry point for REST-like calls)
- `MoodAnalyticsService` (business logic layer)
- `JournalRepository` (data access interface)



Class Diagram Explanation

MoodAnalyticsController

- **Role:** Acts as the entry point for client requests (e.g., API endpoints).
- **Responsibilities:**
  - Receives requests from the client containing date ranges and a `userId`.
  - Delegates the actual computation to the `MoodAnalyticsService`.
  - Returns structured results (`Map<String, Object>`) to the client.
- **Key Methods:**
  - `getTrendAnalysis(...)` → Calls `analyzeTrends` in the service.
  - `comparePeriods(...)` → Calls `comparePeriods` in the service.

- `detectPatterns(...)` → Calls `detectPatterns` in the service.

MoodAnalyticsService

- **Role:** Core **business logic** layer.
- **Responsibilities:**
  - Implements algorithms to process mood journal data.
  - Interacts with the repository to fetch entries for a specific user.
  - Encapsulates all analytical computations, keeping the controller clean.
- **Key Methods:**
  - `analyzeTrends(...)` → Groups entries by week, calculates average ratings, mood frequencies, and overall rating trend.
  - `comparePeriods(...)` → Compares two time periods (entry counts, intensity, mood frequency differences).
  - `detectPatterns(...)` → Detects mood patterns such as stability, streaks, dominant moods, and best/worst days.

JournalRepository (Interface)

- **Role:** Data access abstraction layer.
- **Responsibilities:**
  - Provides methods to query the persistence layer (e.g., database).
  - Decouples the service logic from database implementation.
- **Key Methods:**
  - `findByTimestampBetween(start, end)` → Retrieves entries within a date range.
  - `findAllByUserId(userId)` → Retrieves all entries for a given user.

Interactions

1. **Controller → Service:**
   - The controller does not perform calculations itself.
   - It delegates logic to the service, ensuring separation of concerns.
2. **Service → Repository:**
   - The service requests journal entries (all or filtered by time range) from the repository.
   - It then performs transformations, aggregations, and statistical analysis.
3. **Repository:**
   - Hides details of the persistence mechanism.
   - The service only depends on the interface, allowing easier substitution (e.g., JPA, in-memory DB, or mock repositories for testing).

RESTful Analytics Endpoints

I designed and implemented several key analytics endpoints that processed mood data:

- **Period Comparison**: Compared mood entries between two time periods, enabling users and doctors to see how moods changed over time.
- **Trend Analysis**: Generated time-based trends, identifying whether a user's moods were improving, deteriorating, or remaining stable.
- **Pattern Detection**: Detected recurring patterns (e.g., consistent drops or increases in mood during certain days).
- **Generate Report**: A higher-level endpoint that consolidated results from the other analytics endpoints into a structured report.
- **Get All Journal Entries**: Allowed retrieval of all mood journal entries for debugging, analysis, and validation.

These endpoints were implemented using **Spring Boot**'s controller and service layers, following clean architecture principles to separate concerns between data access, business logic, and presentation.

Endpoint Demonstration

Period Comparison

Example `/period-comparison` endpoint

```
http://localhost:8080/api/analytics/users/1/period-comparison?
period1Start=2025-06-16&period1End=2025-07-13&period2Start=2025-07-
14&period2End=2025-08-11
```

Description:

This algorithm compares a user's mood journal entries between two different time periods.

- It looks at how many entries were logged in each period.
- It checks the average mood intensity (overall rating) in both.
- It compares the frequency of different moods (e.g., "happy," "stressed," "tired") and shows which moods increased, decreased, or stayed the same.
- In short: it highlights how a user's mood patterns changed between two chosen time frames.

Example JSON response


```
{
  "period1Stats": {
    "entryCount": 28,
    "averageIntensity": 6.571428571428571,
    "moodFrequencies": {
      "CALM": 0.21428571428571427,
      "EXCITED": 0.125,
```

```
      "IRRITATED": 0.017857142857142856,
      "HAPPY": 0.26785714285714285,
      "ANXIOUS": 0.05357142857142857,
      "SAD": 0.07142857142857142,
      "ANGRY": 0.017857142857142856,
      "CONTENT": 0.19642857142857142,
      "STRESSED": 0.03571428571428571
    }
  },
  "period2Stats": {
    "entryCount": 29,
    "averageIntensity": 6.275862068965517,
    "moodFrequencies": {
      "CALM": 0.22413793103448276,
      "EXCITED": 0.1206896551724138,
      "HAPPY": 0.20689655172413793,
      "ANXIOUS": 0.08620689655172414,
      "SAD": 0.034482758620689655,
      "CONTENT": 0.25862068965517243,
      "STRESSED": 0.06896551724137931
    }
  },
  "differences": {
    "entryCountDifference": 1,
    "averageIntensityDifference": -0.29556650246305427,
    "moodFrequencyDifference": {
      "CALM": 0.009852216748768489,
      "EXCITED": -0.004310344827586202,
      "IRRITATED": -0.017857142857142856,
      "HAPPY": -0.06096059113300492,
      "ANXIOUS": 0.032635467980295575,
      "SAD": -0.03694581280788177,
      "ANGRY": -0.017857142857142856,
      "CONTENT": 0.062192118226601006,
      "STRESSED": 0.0332512315270936
    },
    "moodChangeSummary": [
      "HAPPY decreased by 6%",
      "CONTENT increased by 6%",
      "SAD decreased by 4%",
      "ANXIOUS increased by 3%",
      "STRESSED increased by 3%",
      "IRRITATED decreased by 2%",
      "ANGRY decreased by 2%",
      "CALM increased by 1%",
      "EXCITED stayed the same"
    ]
  }
}
```


Trend Analysis

Example `/trend-analysis` endpoint

```
http://localhost:8080/api/analytics/users/1/trend-analysis?startDate=2025-07-15&endDate=2025-08-12
```

Description:

This algorithm looks at a user's mood entries over time to identify longer-term trends.

- It calculates average mood ratings over days or weeks.
- It checks whether moods are getting better, worse, or staying stable.
- It highlights trends, such as "ratings are improving in the last two weeks" or "stress has been steadily increasing."
- In short: it gives a big-picture view of mood progression.

Example JSON response


```
{
  "startDate": "2025-07-15T00:00",
  "endDate": "2025-08-12T23:59:59",
  "averageRatingTrend": "increasing",
  "trend": [
    {
      "weekStart": "2025-07-14",
      "averageRating": 6.0,
      "moodFrequency": {
        "CALM": 3,
        "HAPPY": 2,
        "ANXIOUS": 2,
        "CONTENT": 3,
        "STRESSED": 2
      }
    },
    {
      "weekStart": "2025-07-21",
      "averageRating": 6.571428571428571,
      "moodFrequency": {
        "CALM": 3,
        "EXCITED": 2,
        "HAPPY": 3,
        "SAD": 1,
        "CONTENT": 4,
        "STRESSED": 1
      }
    },
    {
      "weekStart": "2025-07-28",
      "averageRating": 6.428571428571429,
      "moodFrequency": {
```

```
        "CALM": 3,
        "EXCITED": 2,
        "HAPPY": 3,
        "ANXIOUS": 1,
        "SAD": 1,
        "CONTENT": 4
      }
    },
    {
      "weekStart": "2025-08-04",
      "averageRating": 5.714285714285714,
      "moodFrequency": {
        "CALM": 3,
        "EXCITED": 2,
        "HAPPY": 2,
        "ANXIOUS": 2,
        "CONTENT": 4,
        "STRESSED": 1
      }
    },
    {
      "weekStart": "2025-08-11",
      "averageRating": 7.5,
      "moodFrequency": {
        "EXCITED": 1,
        "IRRITATED": 1,
        "HAPPY": 2,
        "ANXIOUS": 1,
        "ANGRY": 1,
        "CONTENT": 1
      }
    }
  ]
}
```


Pattern Detection

Example `/pattern-detection` endpoint

```
http://localhost:8080/api/analytics/users/1/pattern-detection?
startDate=2025-04-01&endDate=2025-05-31
```

Description:

This algorithm digs deeper into a single chosen time period to find specific patterns.

- It identifies the most common moods (dominant moods).
- It calculates mood stability (whether ratings are steady, moderately variable, or very up-and-down).
- It finds the best and worst days of the week on average.

- It checks for streaks of positive or negative moods (e.g., “3 days in a row with high ratings”).
- In short: it summarises how consistent moods were and what stood out in that period.

Example JSON response


```
{
  "summary": {
    "entryCount": 31,
    "averageIntensity": 6.161290322580645,
    "moodFrequencies": {
      "CALM": 21.0,
      "DEPRESSED": 1.6,
      "EXCITED": 8.1,
      "HAPPY": 21.0,
      "IRRITATED": 1.6,
      "ANXIOUS": 4.8,
      "SAD": 4.8,
      "ANGRY": 1.6,
      "CONTENT": 29.0,
      "STRESSED": 6.5
    }
  },
  "patterns": {
    "moodStability": "Stable",
    "dominantMoods": ["CONTENT", "HAPPY"],
    "bestDayOfWeek": "SATURDAY",
    "worstDayOfWeek": "MONDAY",
    "longestPositiveStreak": 5,
    "longestNegativeStreak": 1
  }
}
```


Database Design and Modelling

The database schema was extended to support complex mood entries, including:

- **Multiple mood types** per journal entry (e.g., “anxious”, “happy”, “tired”).
- **Intensity values**, represented using integers 1 - 10.
- **Medication tracking**, enabling tracking between dosages and moods.

I implemented this using **MySQL**, ensuring the schema could support efficient querying for analytics.

The database schema is normalized up to Third Normal Form (3NF). Each table has a clear primary key, attributes depend only on that key, and many-to-many relationships (such as journal entries and mood tags) are handled through a linking table. This reduces redundancy and ensures data integrity.

Database Schema

```
users
 ├─ user_id (PK)
 ├─ email (UNIQUE)
 ├─ password_hash
 ├─ created_at
 └─ updated_at

journal_entry
 ├─ journal_entry_id (PK)
 ├─ user_id (FK → users.user_id)
 ├─ overall_rating
 ├─ note
 ├─ timestamp
 ├─ created_at
 └─ updated_at

medication
 ├─ medication_id (PK)
 ├─ name
 ├─ dosage
 ├─ journal_entry_id (FK → journal_entry.journal_entry_id)
 ├─ taken_at
 └─ notes

mood_tag
 ├─ mood_tag_id (PK)
 ├─ name (UNIQUE)
 └─ color

journal_entry_mood_tag
 ├─ journal_entry_id (FK → journal_entry.journal_entry_id, PK part)
 ├─ mood_tag_id (FK → mood_tag.mood_tag_id, PK part)
 └─ intensity
```

Database Normalisation vs. Query Complexity

Normalising the schema avoided duplication (so moods like “happy” or “calm” only existed once in mood_tag).

However, this also made queries more complex, since retrieving a single journal entry with its associated moods required joining across three tables (journal_entry, journal_entry_mood_tag, and mood_tag). For analytics queries, this extra step increased the risk of slower performance if not optimised correctly.

Data Processing and Analysis

Use of Modern Java Features

To build the analytics service in a way that was both **expressive** and **maintainable**, I made deliberate use of modern Java features introduced in recent language versions. These features helped simplify data handling, improve code safety, and make the analytics algorithms more efficient and readable.

Streams and Lambda Functions

Working with mood and journal data often meant filtering, grouping, or aggregating large collections. Traditionally, such operations required verbose `for` loops with nested conditionals. By using **Streams** and **lambda functions**, I was able to write concise, declarative code that directly expressed the intent of the operation.

For example:

- Filtering all journal entries within a given time range.
- Grouping mood tags by type and computing their average intensity.
- Mapping raw database entities into clean DTOs (data transfer objects) for API responses.

This functional style improved not just readability but also maintainability, since the logic could be expressed in fewer lines of code and was less prone to off-by-one or indexing errors.

Algorithm Design for Analytics

The core of the analytics service relied on custom **algorithm design**, particularly around:

- **Trend detection**: Identifying whether a user's mood was generally improving, worsening, or stable over a defined period.
- **Pattern matching**: Detecting repeated cycles or patterns, such as best day and worst day.
- **Period comparison**: Comparing two time windows (e.g., this month vs. last month) to surface changes in mood intensity.

To reduce the impact of inconsistent or noisy data, I implemented a **moving average algorithm** on mood ratings. This technique smoothed short-term fluctuations while preserving long-term trends, making it easier for users to see meaningful insights rather than being distracted by day-to-day variability. For example, instead of showing a sharp spike in mood based on a single anomalous entry, the moving average highlighted the broader pattern across several days.

Pseudocode Algorithms

Trend Analysis

Example pseudo `/trend-anlaysis` algorithm

```
### Pseudo-Algorithm: **Analyze Trends**

**Inputs:**

- `startDateStr` (string, optional)
- `endDateStr` (string, optional)
- `userId` (integer, required)

**Output:**

- A map containing:
```

```
  - Selected start and end dates
  - Overall rating trend (linear regression result)
  - Weekly summaries of ratings and moods

---

#### Steps:

1. **Set date range:**

   - If `startDateStr` is missing → default to 6 months ago.
   - Else → parse given `startDateStr` (start of the day).
   - If `endDateStr` is missing → default to now.
   - Else → parse given `endDateStr` (end of the day).

2. **Retrieve user data:**

   - Fetch all journal entries belonging to the given `userId`.

3. **Filter entries by date range:**

   - Keep only entries with timestamps between `startDate` and `endDate`.

4. **Group entries by week:**

   - For each entry, assign it to the Monday of that week.
   - Build a mapping: **weekStart → list of entries that week**.

5. **Prepare containers:**

   - `weeklyTrendData` → list of weekly summaries.
   - `averageRatings` → list of weekly average ratings (for trend
analysis).

6. **Process each week (sorted chronologically):**
   For each week:

   - Compute **average rating** across all entries that week.
   - Count occurrences of each **mood tag** across all entries.
   - Build a summary containing:

     - `weekStart` (date string)
     - `averageRating` (numeric)
     - `moodFrequency` (map of mood → count)

   - Add summary to `weeklyTrendData`.
   - Add the average rating to `averageRatings`.

7. **Detect overall trend:**

   - Use linear regression on `averageRatings` to determine if ratings are
trending upward, downward, or stable.
```

```
8. **Build result object:**

   - Store `startDate`, `endDate`, `averageRatingTrend`, and full
`weeklyTrendData`.

9. **Return result.**
```

Pattern Detection

Example pseudo `/pattern-detection` algorithm

```
### Pseudo-Algorithm: **Detect Patterns**

**Inputs:**

- `startDateStr` (string, optional)
- `endDateStr` (string, optional)
- `userId` (integer)

**Output:**

- A map containing:

  - **Summary stats** (entry count, average rating, mood frequencies)
  - **Patterns** (stability, dominant moods, best/worst days, streaks)

---

#### Steps:

1. **Parse date range:**

   - If `startDateStr` provided → parse into `startDateTime` (start of
day).
   - If `endDateStr` provided → parse into `endDateTime` (end of day).
   - If both provided → use them as filter range.
   - Otherwise → include **all entries**.

2. **Retrieve user entries:**

   - Fetch all journal entries for `userId`.
   - Filter by `startDateTime` and `endDateTime` if range is given.

3. **Initialize trackers:**

   - `entryCount` = number of entries.
   - `totalRating` = sum of ratings.
   - `ratings` = list of ratings.
   - `tagCounts` = map of mood tag → count.
   - `ratingByDay` = map of day of week → ratings.
```

4. **Process each entry:**

   - Add `overallRating` to `ratings` and `totalRating`.
   - Record rating under the entry's **day of week**.
   - For each mood tag → increment its count in `tagCounts`.

5. **Compute averages:**

   - `avgRating = totalRating / entryCount` (or 0 if none).

6. **Normalize mood frequencies:**

   - Calculate total number of tags.
   - For each mood → compute percentage frequency (rounded to 1 decimal).

7. **Assess mood stability:**

   - Compute variance and standard deviation of ratings.
   - Classify stability as:

     - `Stable` if std dev < 1.5
     - `Moderate` if std dev < 3
     - `Volatile` otherwise

8. **Find dominant moods:**

   - Sort moods by frequency.
   - Select top 2 most frequent.

9. **Find best and worst days of week:**

   - For each day → compute average rating.
   - Best day = highest average rating.
   - Worst day = lowest average rating.

10. **Detect streaks:**

    - Iterate through ratings:

      - Count consecutive positive ratings (`>= 6`) → update `maxPositiveStreak`.
      - Count consecutive negative ratings (`<= 4`) → update `maxNegativeStreak`.
      - Reset streaks otherwise.

11. **Build results:**

    - **Summary:** entry count, average rating, mood frequencies.
    - **Patterns:** mood stability, dominant moods, best/worst day, longest streaks.

12. **Return final result.**

Period Comparison

Example pseudo `/period-comparison` algorithm

```
### Pseudo-Algorithm: **Compare Periods**

**Inputs:**

- `period1Start` (string)
- `period1End` (string)
- `period2Start` (string)
- `period2End` (string)
- `userId` (integer)

**Output:**

- A map containing:

  - Stats for period 1
  - Stats for period 2
  - Differences in entries, intensity, and mood frequencies

---

#### Steps:

1. **Parse input dates:**

   - Convert `period1Start` / `period1End` into `p1Start` and `p1End` (with full-day time boundaries).
   - Convert `period2Start` / `period2End` into `p2Start` and `p2End` (with full-day time boundaries).

2. **Retrieve user data:**

   - Fetch all journal entries for the given `userId`.

3. **Filter entries for each period:**

   - `entriesP1` = all entries between `p1Start` and `p1End`.
   - `entriesP2` = all entries between `p2Start` and `p2End`.

4. **Analyze periods:**

   - Run `analyzePeriod` on `entriesP1` → produce `statsP1`.
   - Run `analyzePeriod` on `entriesP2` → produce `statsP2`.

   _(Each `stats` object includes entry count, average intensity, and mood frequencies.)_

5. **Compute differences between periods:**
```

```
   - Entry count difference = `statsP2.entryCount - statsP1.entryCount`.
   - Average intensity difference = `statsP2.averageIntensity -
statsP1.averageIntensity`.

6. **Compare mood frequencies:**

   - Collect all mood types present in either period.
   - For each mood:

     - Compute frequency in period 1 (`freq1`) and period 2 (`freq2`).
     - Calculate `change = freq2 - freq1`.
     - Save change into a `moodDiffs` map.
     - Build a summary sentence:

       - If change > 0 → "`mood` increased by X%".
       - If change < 0 → "`mood` decreased by X%".
       - If no change → "`mood` stayed the same".

     - Add summary to `moodChangeSummaries` list.

7. **Sort mood summaries:**

   - Order `moodChangeSummaries` by the largest absolute percentage change
(descending).

8. **Assemble differences:**

   - Store `entryCountDifference`, `averageIntensityDifference`,
`moodFrequencyDifference`, and `moodChangeSummary`.

9. **Build final result:**

   - Add `statsP1`, `statsP2`, and `differences` into the result map.

10. **Return result.**
```

Containerisation and Deployment

To ensure the analytics service could be deployed consistently across different environments, I **containerised the backend service using Docker**. Containerisation solves a common problem in software engineering, often described as the *"works on my machine"* issue, where applications behave differently across development, testing, and production due to environmental inconsistencies. By packaging the entire runtime environment including the JDK, dependencies, and application JAR into a Docker image, I ensured the service would run identically regardless of the host system.

The following `Dockerfile` was used to build the Spring Boot application into a lightweight image:

```
FROM eclipse-temurin:21-jdk-alpine
WORKDIR /app
```

```
# Copy the built JAR into the container
COPY target/*.jar app.jar

EXPOSE 8080
ENTRYPOINT ["java", "-jar", "app.jar"]
```

This minimal configuration leverages the **Eclipse Temurin OpenJDK 21 Alpine image**, which provides a secure and efficient base layer. By copying in the pre-built Spring Boot JAR, the container could be run locally with a single command, giving developers immediate access to the same execution environment used in production.

Advantages of Containerisation for Cloud Environments

Beyond simplifying local testing, containerisation provides clear benefits when deploying to cloud environments:

- **Portability**: Docker images are self-contained, meaning the service can be deployed seamlessly across cloud providers such as AWS, Azure, or Google Cloud without modification. This eliminates the need to reconfigure runtime environments manually.
- **Scalability**: Containers are lightweight compared to traditional virtual machines, making it easier to scale horizontally by running multiple instances behind a load balancer. This is particularly important for analytics workloads, which may see spikes in usage when users request reports or visualisations simultaneously.
- **Isolation and Reliability**: Each container runs independently, preventing conflicts between services. For example, the analytics backend could be deployed alongside other microservices (e.g., authentication or journaling services) without dependency clashes.

Testing and Validation

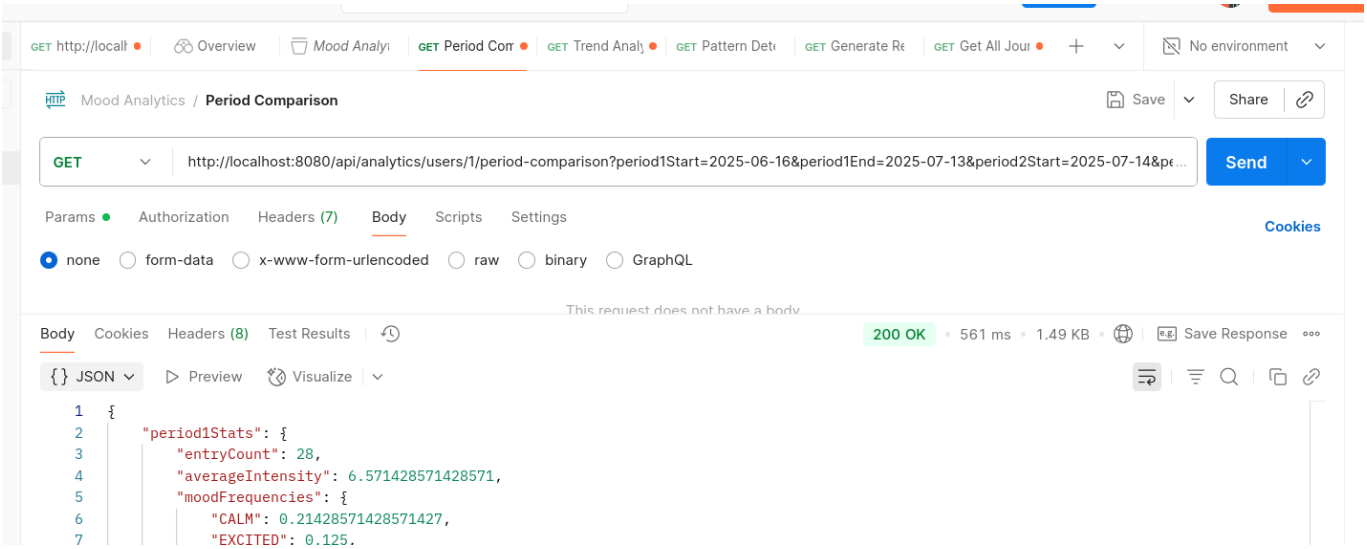
Ensuring the correctness and reliability of the analytics service was an important part of development. Since the system involved multiple interacting components user journaling, mood tagging, medication tracking, and analytics queries it was crucial to validate both the **business logic** and the integrity of database interactions.

Manual Testing

Initially, I relied heavily on **manual testing** to validate functionality. Using **Postman**, I issued HTTP requests to the REST endpoints to confirm that new features behaved as expected. For example, I tested scenarios such as:

- Creating a new journal entry with multiple mood tags.
- Retrieving all entries for a given date range and checking that the correct JSON response were returned.
- Ensuring that invalid inputs (e.g., missing fields or invalid IDs) returned the appropriate error responses.

Example Postman request:



This screenshot shows one of the requests used to check that the period comparison endpoint was working correclty. By inspecting the JSON responses directly, I could confirm whether queries were returning the correct data structure.

Limitations of Manual Testing

While effective in the early stages, I recognised that manual testing has limitations. It is time-consuming, prone to human error, and does not scale well as the number of features increases. For instance, every time I modified the data model, I had to re-run multiple manual checks to ensure existing endpoints were still functioning correctly. This process highlighted the need for a more automated approach.

Unit Testing (Planned)

To improve maintainability and reliability, I plan to incorporate **unit tests** in future iterations. Using a testing framework such as **JUnit** for Java, I would write tests that validate individual components in isolation. For example:

- Verifying that the `JournalEntryService` correctly associates multiple mood tags with a single entry.
- Testing that the analytics functions (e.g., trend comparisons) return expected values for controlled input datasets.
- Mocking database interactions to ensure business logic holds even without a live database connection.

Automated unit tests would act as a **safety net**, catching regressions early in the development cycle. They would also enable faster iterations, since I could make changes with confidence that core functionality remained intact.

Future Expansion: Integration and Automated Testing

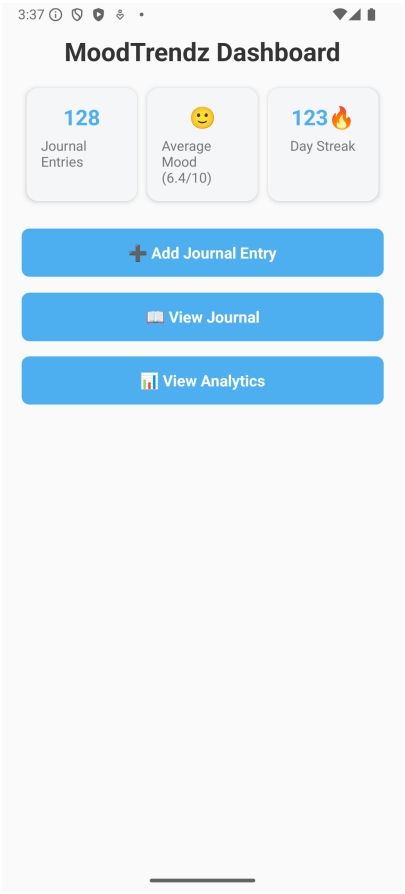
Beyond unit tests, I also see value in adding **integration tests** to verify that different parts of the system (backend, database, and analytics queries) interact correctly as a whole. Combined with a **CI/CD pipeline**, these tests could be run automatically whenever new code is committed, further reducing the risk of errors reaching production.

Frontend Development with React Native

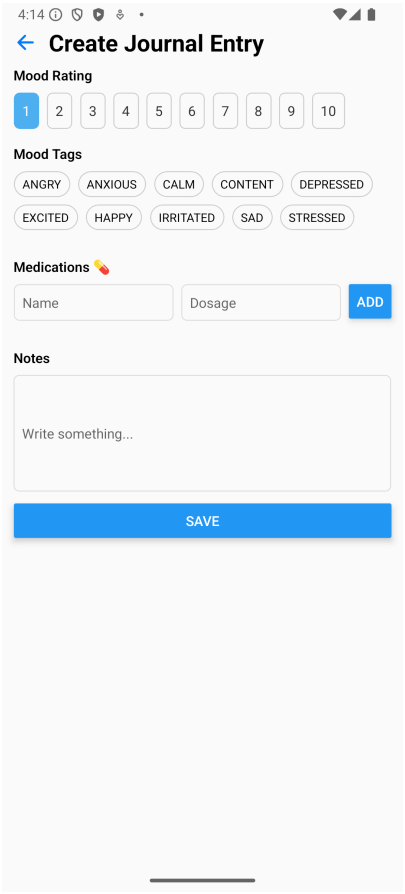
The frontend of the project was developed using **React Native**, chosen for its ability to deliver cross-platform mobile applications with a single codebase. This was an important decision because it allowed me to target both Android and iOS devices without maintaining separate native projects.

Features Implemented

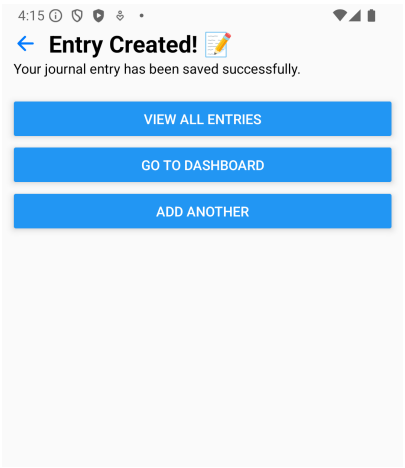
- **Journal Entry Management**: Users could create, view, and update journal entries, including mood ratings, notes, and medication logs.
- **Mood Tagging Interface**: A multi-select UI allowed users to associate multiple moods with varying intensities to each entry, reflecting the complexity of emotional states.

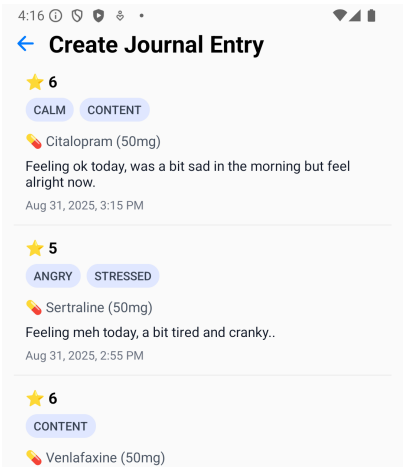


1

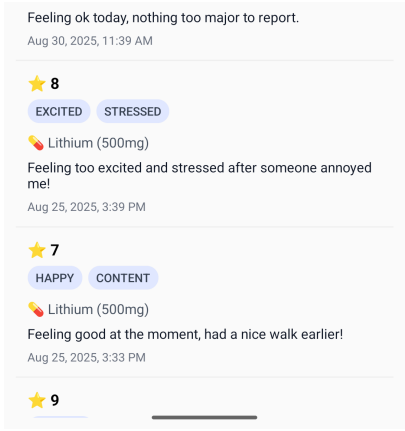


2



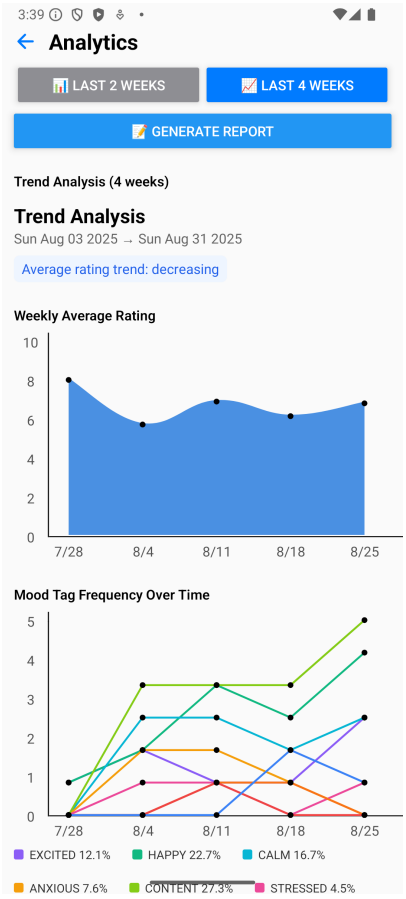


3

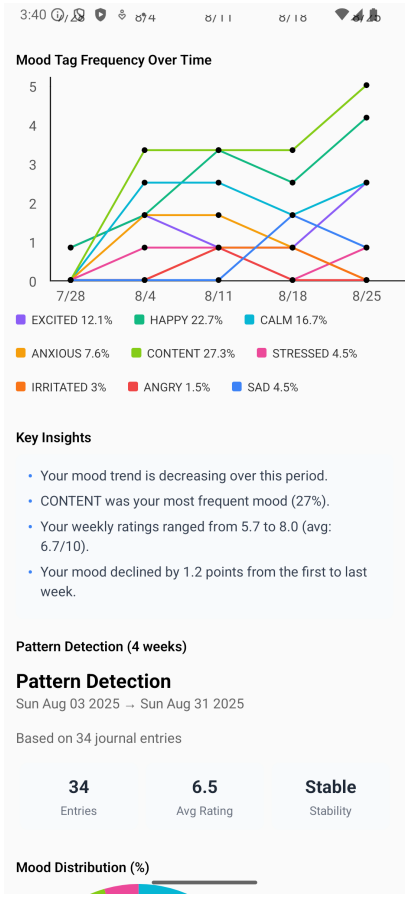


4

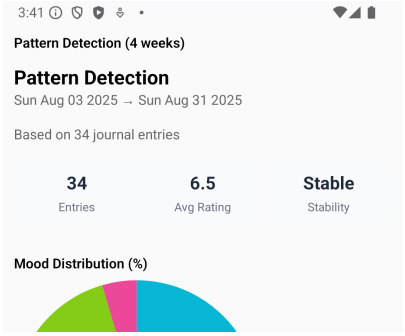
- **Data Visualisation**: Implemented charts to display mood trends over time, enabling users to reflect on their emotional wellbeing. Thiis an example of the trend analysis, pattern detection and period comparison algorithms in action.

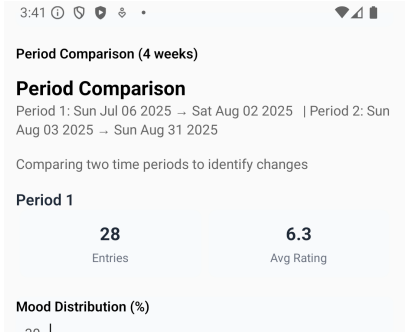


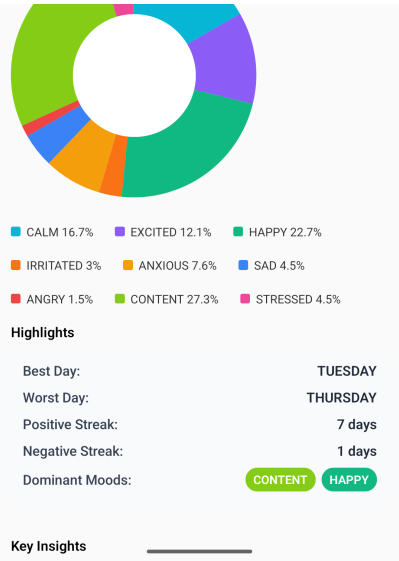
1



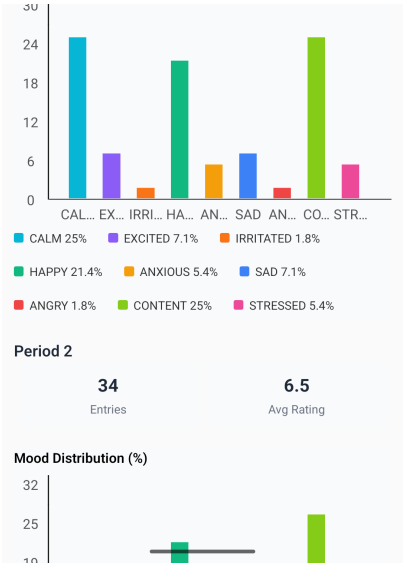
2



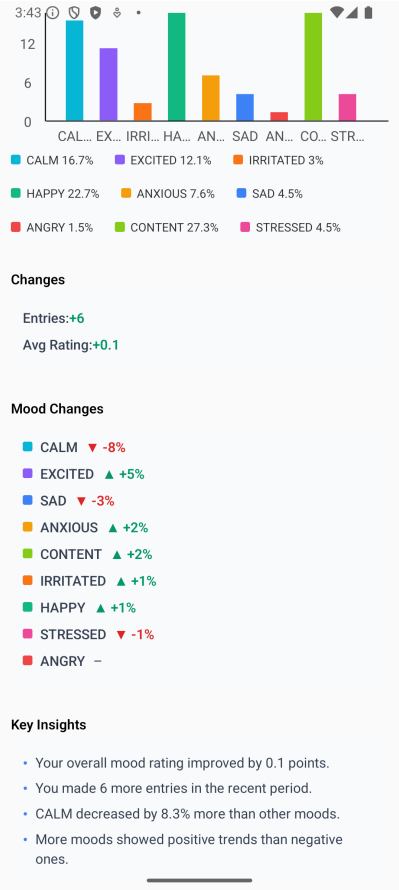




3



4



5

Technical Implementation

- Built **modular components** for reusability, such as mood selectors, input fields, and data cards.
- Used **React Navigation** to handle screen transitions and create a smooth, intuitive user experience.
- Leveraged **Axios** for API communication, ensuring a consistent method for making requests and handling responses.
- Applied **AsyncStorage** for local caching of user sessions, allowing offline access to journal entries before syncing with the backend.

Challenges and Solutions

- **State Management**: Managing complex state, was challenging. I addressed this using React hooks such as `useState` and `useEffect`, while keeping components as stateless as possible.
- **UI Consistency Across Platforms**: React Native components sometimes render differently on iOS and Android. Careful styling and testing on emulators helped maintain consistency.
- **Performance**: Rendering charts with larger datasets could cause lag. To address this, I optimised data-fetching and limited the number of rendered points via smoothing algorithms.

Key Design Challenges

Language Choice

One of the early architectural challenges was deciding which programming language to adopt for the analytics component of the system. At first glance, **Python** appeared to be the natural choice. It has a mature ecosystem for data science (pandas, NumPy, scikit-learn), is widely taught in academic settings, and would have simplified the implementation of exploratory data analysis features. However, the rest of the system particularly the backend service was being developed in **Java**, and diverging into a polyglot codebase introduced risks.

Using two languages in the same system would mean additional **infrastructure complexity**, such as managing separate runtimes, packaging dependencies for both Java and Python, and designing integration points between the services. While feasible, this increases the cognitive load and complexity.

After weighing these trade-offs, I decided to keep the **analytics service in Java**. Although this came at the cost of missing out on Python's rich data science tooling, it delivered **consistency and maintainability**:

Algorithm Complexity

Another significant challenge was designing algorithms for **trend analysis, pattern detection, and period comparison** across journal data. Unlike financial or sensor data, which tend to be quantitative and precise, **mood data is highly subjective and inconsistent**.

For example, one user might rate themselves as "happy = 7/10" on one day, and then "content = 6/10" the next, even though both ratings reflect a very similar emotional baseline. Another user may never use numerical ratings consistently at all, instead relying heavily on qualitative text notes. This inherent **variability and noise** makes it difficult to extract meaningful trends.

Some of the key difficulties included:

- **Accuracy vs Over-Interpretation**: It was easy for algorithms to overfit, spotting "trends" in what was essentially random variance. For example, a slight drop from 7/10 to 6/10 may not indicate a genuine decline in mood, but simply day-to-day fluctuation.
- **Temporal Comparisons**: Comparing moods across different time periods (e.g., "this month vs last month") had to account for irregular logging behaviour. If a user skipped several days, naive averaging could produce misleading results.
- **Multiple Data Types**: The algorithm needed to handle both structured inputs (mood tags, intensity values, ratings) and unstructured data (free-text notes). Balancing these required compromises, such

as focusing trend analysis only on numerical components initially.

Ultimately, the algorithms aimed to be **supportive rather than prescriptive** providing users with broad insights while acknowledging the subjectivity and noisiness of self-reported data. This trade-off reinforced the importance of **user-centred design** in mental health applications: the goal is to empower reflection, not deliver clinical diagnoses.

Representing Multiple Emotions

Perhaps the most interesting data modelling challenge was how to represent multiple simultaneous emotions within a single journal entry. Emotional states are rarely singluar; a user might feel **both excited and anxious** before a big presentation, or **happy yet nostalgic** after meeting old friends. Capturing this richness was central to the project.

Several approaches were evaluated:

1. **Single mood per entry**
    - Simplest to model and query.
    - However, it **misrepresents reality**, forcing users to oversimplify complex experiences.
2. **Embedded arrays (JSON storage)**
    - Allowed flexible storage of multiple moods in a single field.
    - Querying and aggregating across entries became inefficient, requiring custom JSON parsing.
    - Violated principles of relational normalisation.
3. **Relational many-to-many schema (chosen)**
    - Implemented through a junction table (`journal_entry_mood_tag`).
    - Each entry can link to multiple mood tags, each with an **intensity value**.
    - Enables powerful queries, e.g., “Find all entries where anxiety and excitement co-occur,” or “Average intensity of happiness when medication X was taken.”

Team and Process Challenges

Remote Supervision and Communication

One challenge of working largely independently was adapting to the supervision model, which relied on weekly meetings over Microsoft Teams. Unlike daily collaboration or in-person lab sessions, this meant that questions or uncertainties could not always be addressed immediately. If I encountered a technical blocker early in the week, I often had to either find a solution myself or wait until the next scheduled meeting to discuss it.

This required me to become more **self-reliant and resourceful**. I developed the habit of maintaining a running log of issues, questions, and design decisions throughout the week. By structuring my notes in this way, I could make the most of supervision time, ensuring meetings were efficient and focused. At the same time, this process helped me to reflect critically on my own work before asking for guidance, which often led me to resolve issues independently.

Another adaptation was the use of short demos when communicating with my supervisor. Short demos proved invaluable in making remote discussions more precise. This experience highlighted the importance of presenting technical problems in a concise but structured manner when working in asynchronous or limited-contact environments.

Balancing Deadlines and Scope

Time management was another significant challenge. With a wide range of possible features to implement —journaling, mood tagging, analytics, visualisations, and data modelling, it was easy to become overambitious. Initially, I attempted to plan for advanced features such as machine learning-driven mood predictions and fully customisable dashboards. However, the reality of the timeline forced me to reassess what could realistically be achieved.

I adopted a **minimum viable product (MVP)** mindset, focusing first on building a stable journaling system with core functionality before attempting enhancements. By prioritising essentials, such as ensuring the database schema was robust and the journaling workflow intuitive, I avoided situations where the project could risk being incomplete. This approach also reduced stress by giving me clear milestones to track progress.

Maintaining Motivation and Focus

Another challenge of working independently, particularly in a remote setting, was maintaining consistent motivation and focus. Without the natural accountability of working alongside peers, it was sometimes difficult to sustain momentum over longer periods. This was especially noticeable when I encountered complex bugs or setbacks that were time-consuming to resolve.

To address this, I adopted strategies such as breaking tasks into **smaller milestones** and tracking them on a Kanban-style board. This gave a sense of tangible progress and prevented larger goals from feeling overwhelming. I also made use of short reflection sessions at the end of each week to review what I had accomplished and what needed to be improved. These techniques helped maintain productivity and reduced the frustration that often comes with solo development work.

Conclusion

Reflecting on the placement, I believe it was a highly valuable experience for both my technical and personal development. My placement at the Software Research Institute gave me the opportunity to contribute to the development of the **MoodTrendz analytics service**, a project that combined technical innovation with meaningful healthcare use cases. Working on a system designed to help users track and understand their emotional wellbeing highlighted how software can make a tangible difference in people's lives, which was both motivating and rewarding.

Through the process of **designing APIs**, **modelling complex data**, and **implementing algorithms**, I gained first hand experience of applying academic knowledge to real world challenges. I developed a stronger appreciation for the importance of thoughtful design decisions such as how data is structured, how algorithms handle noisy input, and how deployment practices influence reliability in production. These experiences not only enhanced my technical skill set but also improved my problem-solving ability and professional confidence.

Although the project presented challenges ranging from technical hurdles like balancing database normalisation with performance, to personal ones like adapting to remote supervision I learned to navigate them by remaining adaptable, seeking out resources, and maintaining clear communication with my supervisor. These habits will continue to serve me in future professional environments where problem-solving and collaboration are key.

References

- M. Fowler, Microservices: A Definition of This New Architectural Term. martinfowler.com, 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html
- Carnell, J. (2017). Spring Microservices in Action (1st ed.). Manning Publications.
- Docker Inc., Docker Documentation. [Online]. Available: https://docs.docker.com/
- Spring.io, Spring Boot Reference Documentation. [Online]. Available: https://spring.io/projects/spring-boot
- MySQL, MySQL 8.0 Reference Manual. Oracle Corporation. [Online]. Available: https://dev.mysql.com/doc/
- OpenAPI Initiative, OpenAPI Specification (Swagger). [Online]. Available: https://swagger.io/specification/
- Oracle, Java SE 21 Documentation. [Online]. Available: https://docs.oracle.com/en/java/javase/21/