Assignment: 03

SE - 322

Submitted By

Toqeer Ahmed — 2023728
Mian Muhammad Owais — 2023-316
Mustajaab Ul Fida Qadri — 2023-567

Submitted To

Instructor: Said Nabi
Department of Computer & Software Engineering

Ghulam Ishaq Khan Institute of Engineering Sciences & Technology
December 16, 2025

Contents

1 Component Diagram

1.1 Explanation of Componentization Strategy

For the Component Diagram, I have grouped the system classes into logical high-level components based on **Feature Domains**. This aligns with the "Package by Feature" or "Domain-Driven Design" (connected to Assignment 2's Layered Architecture).

1. **AuthComponent**: Encapsulates all user identity, registration, and security logic (UserService, JwtService). This is a foundational component required by all others.
2. **MoodTrackingComponent**: Handles the core student feature—logging and retrieving mood data (MoodService).
3. **BurnoutAnalysisComponent**: Contains the complex logic for analyzing student data (BurnoutDetectionEngine and Strategies). Separating this allows the analysis logic to evolve (e.g., adding ML) without touching the data logging components.
4. **NotificationComponent**: Abstraction for external communications. It uses the Factory and Adapter patterns to hide the complexity of different providers (Twilio, Firebase).
5. **AdvisorComponent**: Provides the interface for advisors to monitor student wellness.

1.2 Component Diagram

Figure 1 illustrates the componentization strategy with dependencies between components.

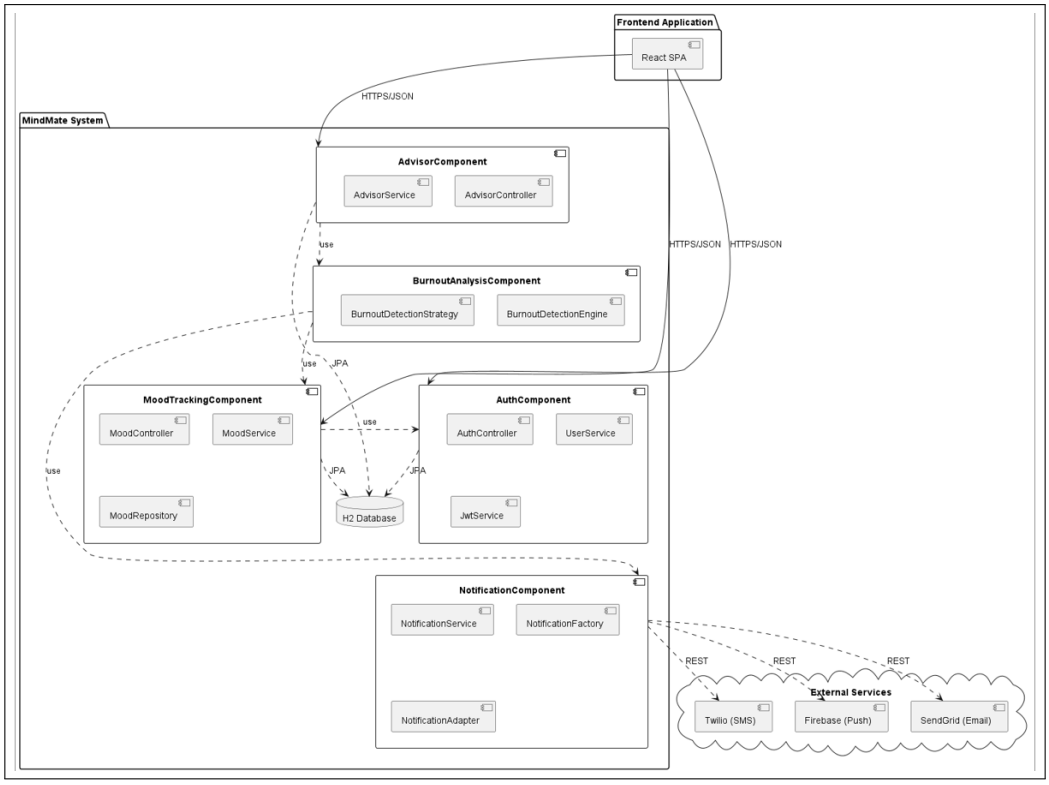

Figure 1: Component Diagram showing the system's modular architecture

2 Sequence Diagram

2.1 Use Case: High-Risk Mood Detection & Notification

This diagram models the flow where a student logs a generic mood entry that triggers a burnout warning. It demonstrates the interaction between the Mood Component, Analysis Component, and Notification Component.

- It highlights the **Observer Pattern**: MoodService (Subject) notifying BurnoutDetectionEngine (Observer).
- It shows the **Strategy Pattern**: The Engine choosing a strategy to analyze data.
- It shows the **Factory/Adapter Pattern**: The Notification Service creating the right adapter to send the alert.

2.2 Sequence Diagram

Figure 2 shows the interaction flow between system components when a high-risk mood is detected.

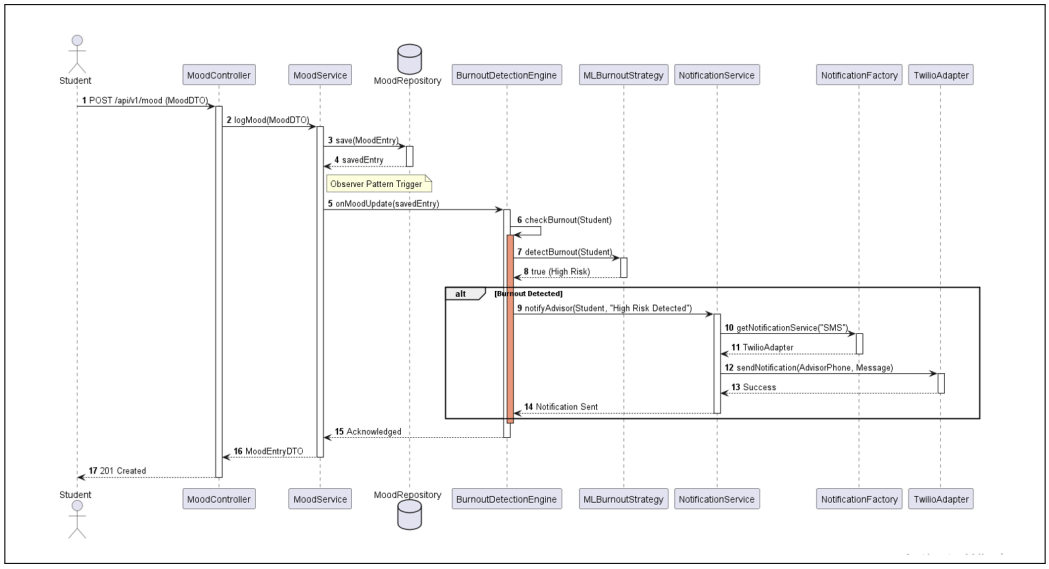

Figure 2: Sequence Diagram for High-Risk Mood Detection & Notification

3 Activity Diagram

3.1 Business Process: "Burnout Analysis & Intervention Workflow"

This diagram visualizes the complex decision-making process that occurs after a student logs their mood. It is chosen because it involves parallel activities (Fork/Join) and conditional logic that would be hard to capture in a sequence diagram.

- **Decisions**: Checking if the stress level exceeds the threshold.
- **Parallelism (Fork)**: If risk is detected, the system simultaneously notifies the advisor AND sends an immediate supportive message to the student, while also logging the incident.
- **Swimlanes**: Clearly separate the responsibilities of the Student, the System (Backend), and the Advisor.

3.2 Activity Diagram

Figure 3 depicts the workflow with decision points, parallel activities, and swimlanes.

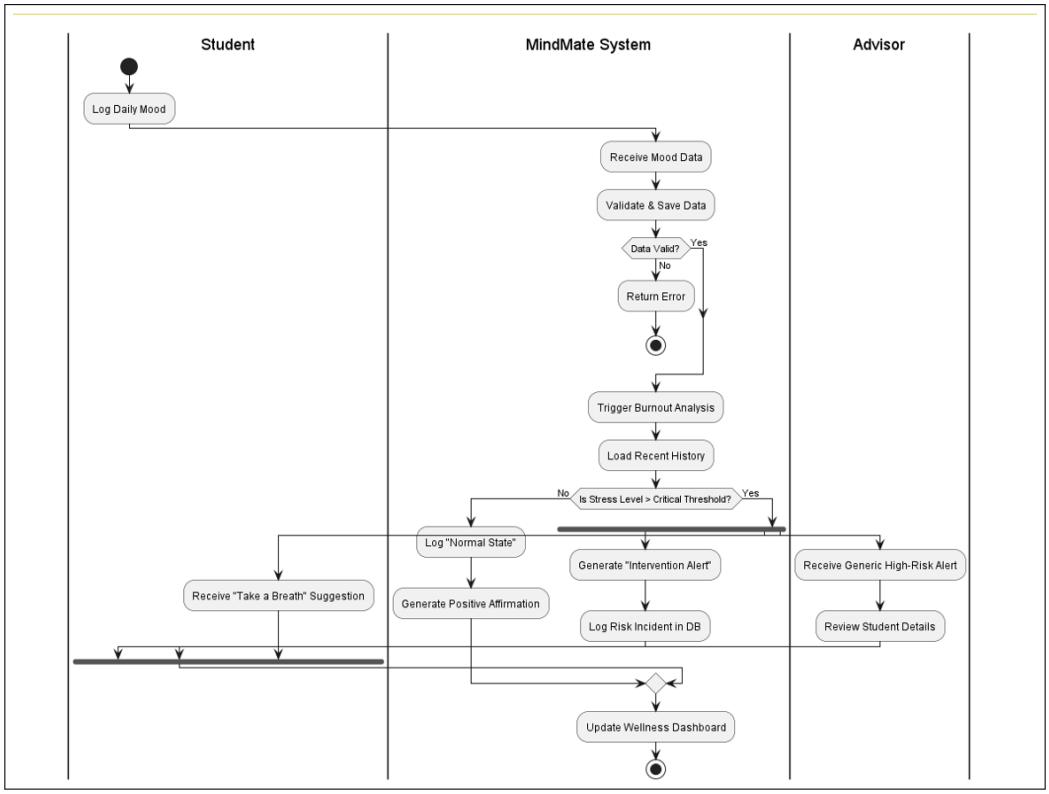

Figure 3: Activity Diagram showing the Burnout Analysis & Intervention Workflow

4 Architectural Style and Rationale

4.1 Architectural Style: Layered (N-Tier) Architecture

4.2 Rationale

We have chosen the **Layered Architecture** style for the MindMate system. This style organizes the system into horizontal logical layers, where each layer has a specific role and responsibility (Presentation, Business Logic, Data Access).

1. **Separation of Concerns & Maintainability**: This style perfectly fits our project needs because it enforces a strict separation of concerns. The *Controller Layer* handles HTTP requests and JSON serialization but knows nothing about business rules. The *Service Layer* contains all the complex business logic (like the Burnout Detection strategies) but knows nothing about SQL or database connections. The *Repository Layer* handles data persistence. This separation makes the system easier to maintain; for example, we could switch the specialized BurnoutDetectionStrategy in the Service Layer without changing a single line of code in the Controllers or Repositories.

2. **Testability**: The Layered Architecture directly supports our Non-Functional Requirement for high reliability through testing. Because the layers are decoupled (dependency injection), we can unit test the Business Logic Layer in isolation by mocking the Repository Layer. This is critical for our BurnoutDetectionEngine—we need to verify it detects risks correctly without needing to spin up a real database or send real SMS messages during every test run.

3. **Relationship to Design Patterns (Assignment 2)**: This architecture provides the "home" for the design patterns we implemented in Assignment 2:

   - The *Repository Pattern* provides the interface for the Data Access Layer, abstracting the database.
   - The *Factory* and *Strategy* patterns live within the Service Layer to encapsulate complex business logic decisions.
   - The *DTO (Data Transfer Object)* pattern is used to pass data between the Presentation and Service layers, ensuring that internal database entities (like User passwords) are never exposed directly to the API responses.

Summary

The Layered Architecture provides a robust, maintainable, and testable foundation for the MindMate system, supporting all the design patterns implemented and ensuring clear separation of concerns across the application.