Design Document

Group 2

Rajvir Bhatti, Hana Hassan, Adeeb Hossain, Mariam Ibrahim, Junaid Khan, Usman Mahmood

Architecture:

Initially, in the early stages of our application design, we went for a microservices architecture approach, only because it was more optimal in terms of running the application (scalable, flexible and better performance). However, due to the simplicity and how small scale our project is, we eventually ended up doing a Layered Architecture. Our application follows the following Layer approach, Frontend (Presentation Layer), Backend (Application Layer) and the Database (Data Layer). Anytime a user makes a request on the website (frontend), that information gets sent to the code (backend), and depending on what's being asked, the code then sends a request to the database, which then in turn sends back the requested information in the reverse order.

Design Pattern:

Our project follows the **Model View Controller (MVC)** design pattern to create a scalable, maintainable, and modular architecture. The system consists of multiple services integrated into a cohesive frontend and backend setup, supporting a unified user experience.

Model:

The model layer handles the core data and backend logic. In our project, this includes:

- **Database Models:** MySQL tables representing user data, interests, and system components (e.g., preferences and user accounts).
- **LLMService & WeatherService Models:** These services interact with external APIs and databases to retrieve or update information.

The models represent our data layer, we are using these to collect, obtain and output data.

View:

The view layer represents the **frontend**, primarily built with **React**. It manages user interaction, displays data, and sends user actions to the backend through **Flask API routes**.

- **Interest Management Section:** Displays user interests, supports adding/removing interests, and updates dynamically.
- **Live Data Views:** Display weather and temperature data fetched from services.

The views interact with backend APIs to retrieve updated data, ensuring a responsive experience.

Controller:

The Flask backend acts as the **controller** — bridging the models and views. It handles incoming requests, processes data, and sends appropriate responses back to the frontend. Key example routes include:

- @app.route("/weather", methods=["GET"]): Fetches weather data from the WeatherService.
- @app.route("/get-recommendations", methods=["POST"]): Returns multiple activity recommendations

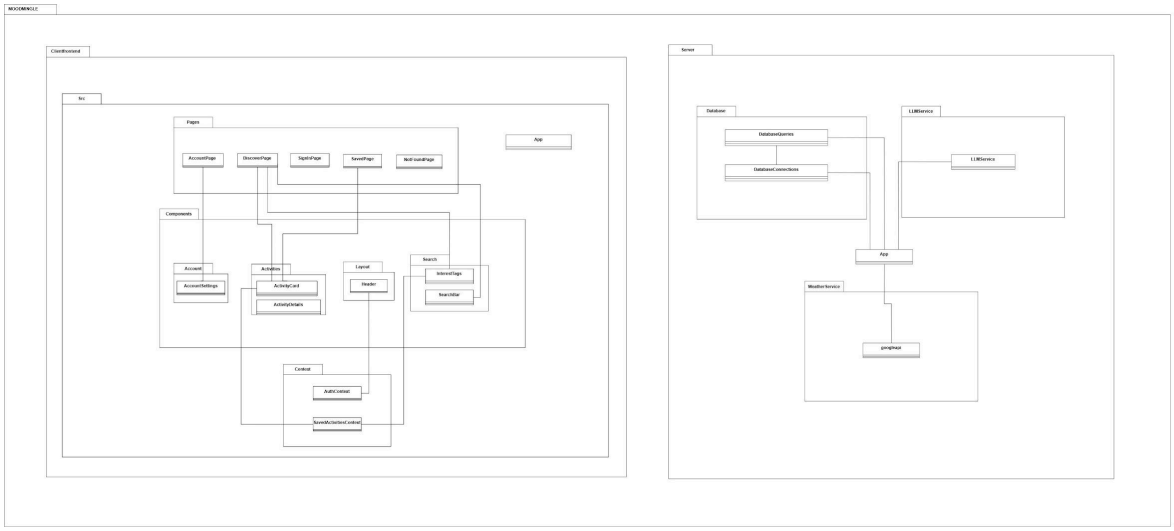
The controller ensures the business logic is centralized and separates the frontend from backend data handling.

Design Principles:

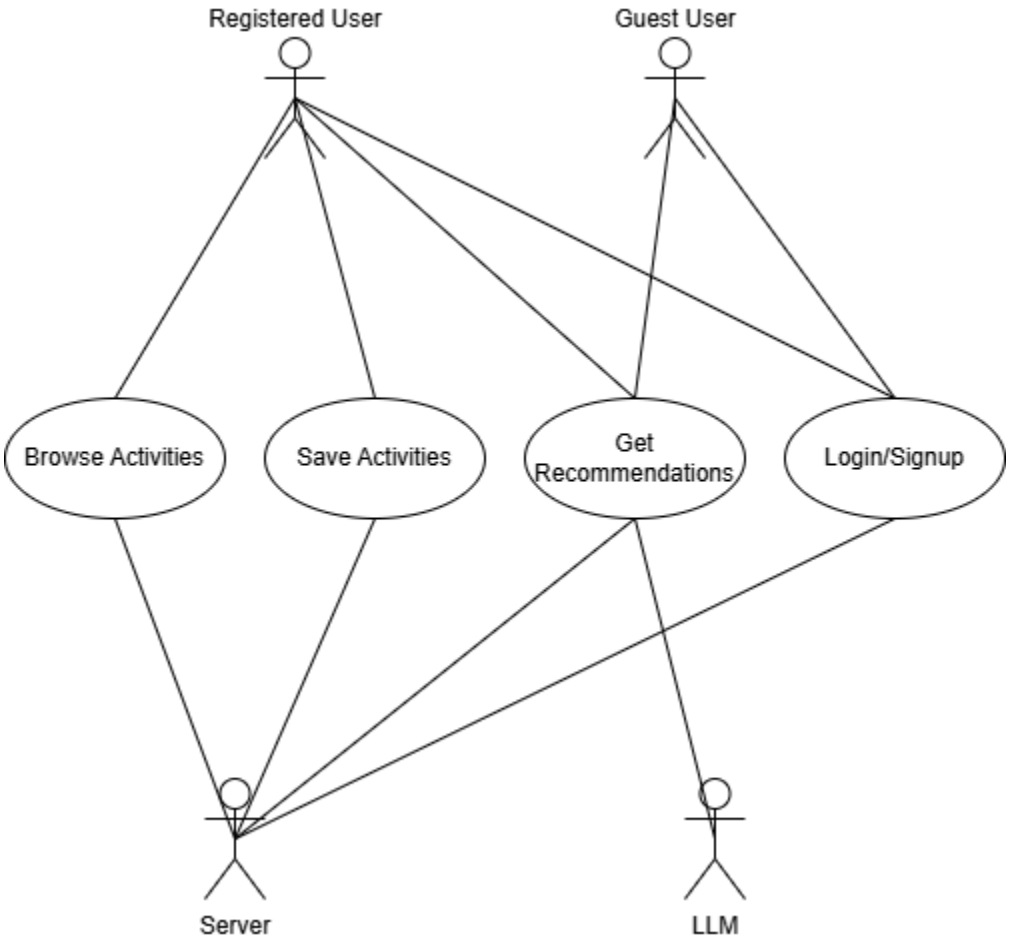
- **Single Responsibility:** Each service and component is responsible for one task remove_interest() strictly manages interest deletions, and React components are focused on UI rendering.
- **Open/Closed Principle:** Our system is built to allow extensions without modifying existing code. For example, adding new API routes or extending Wiegand support for different data formats is straightforward.
- **Interface Segregation:** Services like LLMService and WeatherService expose only necessary functionality to avoid unnecessary dependencies.
- **Dependency Inversion:** The frontend depends on the Flask API rather than directly accessing the database, making it adaptable to backend changes.

This architecture ensures our system remains **maintainable**, **scalable**, and **flexible** for future improvements. Whether that's expanding frontend features, or integrating additional external services.

UML



Use Case


Registered User
Guest User
Browse Activities
Save Activities
Get Recommendations
Login/Signup
Server
LLM