<Group 05>

<SoulNote>
Software Architecture Document

Version <1.0>

Revision History

Date	Version	Description	Author
23/06/25	1.0	Architectural Goals and Constraints	Ly Quoc Thanh
24/06/25	1.0	Use-Case Model	Ly Quoc Thanh
24/06/25	1.0	Component: Controller	Ly Quoc Thanh
24/06/25	1.0	Architecture Overview	Pham Quang Thinh
24/06/25	1.0	Component: Database	Pham Quang Thinh
24/06/25	1.0	Introduction	Nguyen Tan Van
25/06/25	1.0	Component: View	Nguyen Tan Van
25/06/25	1.0	Layer Breakdown	Huynh Van Sinh
26/06/25	1.0	Component: Model	Nguyen Le Quang
11/07/25	1.0	Architecture Diagram	Ly Quoc Thanh
20/07/2025	1.1	Deployment	Pham Quang Thinh
20/07/2025	1.1	Change Firebase Storage to Cloudinary	Pham Quang Thinh
20/07/2025	1.1	Implementation View	Nguyen Tan Van

Table of Contents

Software Architecture Document

1. Introduction

The introduction of the Software Architecture Document for the **SoulNote** project provides an overview of the entire architectural framework of the system. It includes the purpose, scope, definitions, acronyms, references, and a general outline of the structure and goals of the document.

1.1 Purpose

The purpose of this document is to define the software architecture of the **SoulNote** project — a web-based memory journaling application. This document serves as a comprehensive blueprint for the system's design and implementation, offering a clear architectural vision to guide developers, testers, designers, and other stakeholders throughout the development lifecycle. It ensures consistency, scalability, and maintainability of the system.

1.2 Scope

This document covers all aspects of the software architecture of the **SoulNote** project. It includes detailed descriptions of the system components, their interactions, and the technologies used. The scope is limited to the architectural design and does not include detailed implementation or code-level specifics.

1.3 Definitions

- **API**: A protocol that lets different software systems communicate; SoulNote uses RESTful APIs.
- **DB (Database)**: A structured data store; SoulNote uses PostgreSQL to manage its data.
- **UI (User Interface)**: The visual components users interact with, like buttons and forms.
- **UX (User Experience)**: The overall feeling users get; SoulNote emphasizes emotional and reflective UX.
- **MVVM**: A frontend pattern separating View, ViewModel, and Model; used in the React client.
- **MVC**: A backend pattern splitting Model, View (JSON), and Controller; used in the Nest.js server.
- **React**: A JavaScript library for building modular and dynamic user interfaces.
- **Tailwind CSS**: A utility-first CSS framework for building consistent and responsive UIs.
- **Vite**: A fast frontend build tool with hot module replacement for React development.
- **Prisma ORM**: A type-safe ORM used to manage PostgreSQL data models in the backend.
- **PostgreSQL**: An open-source relational database storing users, memories, and metadata.
- **Firebase Authentication**: A service for user login, registration, and identity management.
- **Cloudinary**: Cloud storage for user-uploaded content like images and audio.
- **Firebase Cloud Messaging**: A service for sending notifications and reminders to users.
- **Jest**: A JavaScript testing framework used for unit testing the frontend.
- **React Testing Library**: A tool to test React components through user interactions.

- **Supertest**: A tool for testing backend API endpoints in Nest.js.
- **Cypress / Playwright**: End-to-end testing frameworks simulating real user behavior in the browser.
- **GDPR**: A regulation ensuring personal data protection; SoulNote encrypts data to comply.
- **JSON**: A lightweight data format used for client-server communication in REST APIs.
- **CRUD**: Basic operations (Create, Read, Update, Delete) performed on application data.

1.4 Abbreviations

Common acronyms used in this document include:

Abbreviation	Definition
API	Application Program Interface
DB	Database
UI	User Interface
UX	User Experience
CRUD	Create, Read, Update, Delete
MVC	Model-View-Controller
MVVM	Model-View-ViewModel

1.5 References

1. https://viblo.asia/p/tim-hieu-ve-mo-hinh-mvvm-maGK7vW95j2
2. https://youtu.be/pcM9xQiUt5g?list=PL3Bp9JDvkArbbb4KVB_Lk9QRGohSKt9hM

1.6 Overview

This document provides a comprehensive overview of the SoulNote project's software architecture. It includes descriptions of the system's major components, their interactions, and the guiding principles behind the architectural decisions. The overview serves as a roadmap for stakeholders to understand the overall structure and design approach of the system.

2. Architectural Goals and Constraints

- **Architectural Goals**
  - **User-Friendly Experience:** The system should offer an easy-to-use and emotional interface that helps users create, store, and share their memories with photos, audio, location, and feelings.
  - **Scalability:** The system must support an increasing number of users and memory data over time

without slowing down or crashing.
- **Security & Privacy:** User data such as memories, feelings, and locations must be stored and transferred securely. The system should follow privacy regulations like GDPR.
- **Maintainability:** The system should be built in separate modules (frontend, backend, database, storage) so that it's easy to fix bugs, update features, or add new ones in the future.
- **Performance:** Core features like uploading a memory, viewing the timeline, and editing should respond quickly. Aim for less than 3 seconds in normal usage.
- **Multi-device Support:** Users should be able to access the same data from different devices (phone, tablet, browser) without any issues.
- **Reliability:** The platform should be stable and work most of the time. Target system uptime is 99.9%. Backup and recovery mechanisms should prevent data loss.

- **Architectural Constraints**

  - **Technology Stack:**
    - **Frontend (Client):** Built with React using the MVVM pattern, along with TypeScript, Tailwind CSS, Vite, and React Router.
    - **Backend (Server):** Built with NestJS, a progressive Nest.js framework that uses **Express** under the hood. It provides a modular architecture and uses Prisma ORM for database operations with PostgreSQL.
    - **Authentication & Storage:** Uses Firebase Authentication for login/register, and **Cloudinary** to store memory media (photos, audio). Firebase Cloud Messaging is used for reminders.
  - **Platform Support:** The app must run smoothly on all modern browsers (Chrome, Firefox, Safari, Edge) and mobile devices. Responsive design is required.
  - **Data Privacy Compliance:** All personal data (like location, emotions, uploaded files) must be encrypted and securely stored. The system must follow privacy laws such as GDPR.
  - **Performance Requirements:** API response time should be under 500 milliseconds for common actions such as creating or viewing a memory. Page loading should be optimized with lazy loading and pagination.
  - **Resource Constraints:**
    - Uploaded files (images, audio) must be limited in size (e.g., max 10MB).
    - Use cloud storage and optimized data transfer to reduce server and bandwidth usage.
  - **Testing Requirements:** The system must support automated testing:
    - **Frontend:** Unit testing with Jest, component testing with React Testing Library.
    - **Backend:** API testing with Supertest.
    - **End-to-end:** Tests using Cypress or Playwright for full user workflows.
  - **Integration Needs:** The system must be able to connect with:
    - **Firebase services** (Authentication, Storage, Notifications)
    - **Map APIs** like Google Maps for location-based memories
    - **Optional** services like email or SMS for future reminders

3. Use-Case Model

4. Logical View

A. Architecture Overview

The architecture of the SoulNote system is designed following two complementary models:

1. **Client-Server Architecture**
   SoulNote adopts a classic Client-Server model that separates the user-facing interface from backend logic and data storage.

   - **Client**: A Single Page Application (SPA) built with **ReactJS + TypeScript**, running in the user's browser. It is responsible for rendering the UI, handling user interaction, and sending API requests to the server using **HTTPS**. The client interacts with the server via **RESTful API endpoints.**
   - **Server**: A Nest.js backend application using **NestJS** (or Express) that handles all business logic. It validates requests, performs operations via services, interacts with the database (**PostgreSQL**), and connects to third-party services like Firebase. The server processes requests from clients and sends appropriate JSON responses over **HTTPS.**

   This separation allows for modular development, improved maintainability, and the potential for independent scaling of frontend and backend.

2. Layered Architecture

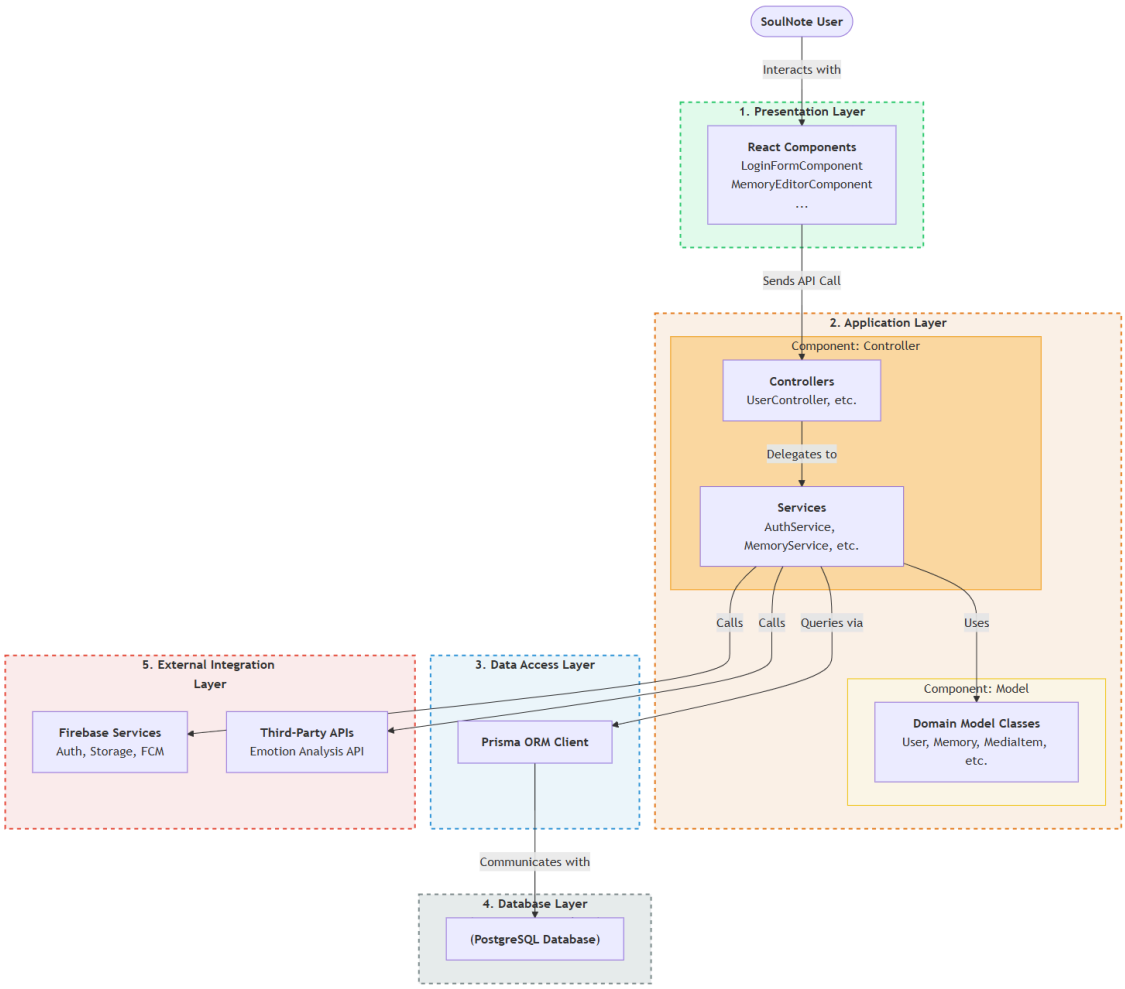
To ensure clarity and scalability, the system is logically organized into 4 layers, each with distinct responsibilities:

- **Presentation Layer:**
  - Located on the **client-side** (browser).
  - Built with **ReactJS** using the **MVVM** pattern.
  - Includes all UI components (**View**), state-handling logic (**ViewModel**), and data models used for display.
- **Application Layer:**
  - Located on the **server-side** (NestJS backend).
  - Applies a modularized **MVC** pattern tailored for APIs.
  - This layer includes **controllers** (handle HTTP requests), **services** (business logic), and **DTOs** (data transfer objects).
- **Data Layer:**
  - Uses **Prisma ORM** to interact with a **PostgreSQL** database that stores structured data (users, memories, reports, etc.).
  - Multimedia content (images, audio) is stored in **Cloudinary** for performance and scalability.
- **External Services Layer:**
  - Includes services integrated via HTTP/REST APIs:
    - **Firebase Authentication** (email/password & Google login)
    - **Firebase Cloud Messaging (FCM)** for push notifications
    - **Emotion Analysis API** (text/image emotion extraction)
    - **Geolocation API** for mapping memory locations

3. Communication Channels

Interaction	**Communication Method**
Client ↔ Server	HTTPS (REST API)
Server ↔ PostgreSQL	Prisma ORM over TCP
Server ↔ Firebase (Auth, Storage)	HTTPS (external API)
Server ↔ External APIs (Emotion, Map)	HTTPS (external API)

4. Architecture Diagram



The diagram above illustrates the architecture of the SoulNote application, structured using a multi-layered design pattern. This approach separates responsibilities across different layers to ensure maintainability, scalability, and clear data flow.

Presentation Layer (Component: View):

- Purpose: Serves as the user interface (UI) where users interact with the system.
- Technology: ReactJS
- Components: Includes UI components such as LoginFormComponent, MemoryEditorComponent, etc.
- Role: Captures user actions and sends API requests to the application layer.

Application Layer:

- Purpose: Handles application logic and coordinates actions between layers.
- Components:
    - Controllers: e.g., UserController, responsible for receiving and routing API requests.
    - Services: e.g., AuthService, MemoryService, implement business logic.
- Role: Orchestrates logic and data flow without managing persistent data.

Data Access Layer

- Purpose: Acts as a bridge between the business logic and the database.
- Technology: Prisma ORM
- Role: Provides an abstraction layer for secure and efficient database operations.

Database Layer (Component: Database)

- Purpose: Stores persistent application data.
- Technology: PostgreSQL
- Role: Responds to data access requests from the Data Access Layer.

External Integration Layer:

- Purpose: Integrates with external services and third-party APIs.
- Components:
    - Firebase Services: Handles authentication, media storage, and push notifications.
    - Third-Party APIs: Includes external tools such as Emotion Analysis APIs.
- Role: Enables extended functionality without internal implementation.

Data Flow Summary:

1. The user interacts with the interface (React Components).
2. UI components send API requests to Controllers in the Application Layer.
3. Controllers delegate processing to Services.
4. Services may:
    a. Use Domain Models to validate and manipulate data.
    b. Access data through the Prisma ORM.
    c. Call external services such as Firebase or third-party APIs.
5. Responses are returned upward through the layers to the user interface.

B. Layer Breakdown

The SoulNote system adopts a multi-layered architecture that separates concerns across user interaction, application logic, data access, and external integrations. The architecture supports modular development and maintainability, and is structured as follows:

1. **Presentation Layer (Frontend / Client)**
   This layer is a Single Page Application (SPA) running entirely in the user's web browser.

   Purpose:

   - To render the user interface (UI) and provide a rich, interactive user experience.
   - To manage client-side state (e.g., what the user is typing, which memory is selected).
   - To handle user interactions and communicate with the backend server via API calls.

   Technologies:

   - **Framework/Library:** React.js
   - **Build Tool:** Vite
   - **Language:** TypeScript
   - **Styling:** Tailwind CSS
   - **Routing:** React Router
   - **API Communication:** Axios or native fetch()

   Structure:

   - **Components:** Reusable UI blocks built with React (e.g., MemoryCard, Editor, Timeline).
   - **Views/Pages:** Compositions of components that represent a full page (e.g., HomePage, NewMemoryPage, ProfilePage).
   - **Services/Hooks:** Encapsulates logic for making API calls to the backend (e.g., a useMemories hook that fetches memories from the NestJS server).
   - **State Management:** (Optional) React Context, Zustand, or Redux for managing global application state.

   Communication:

   - Communicates exclusively with the Application Layer (Backend) via **RESTful API calls over HTTPS**. It sends JSON data in requests and receives JSON data in responses.

2. **Application Layer (Controller / Service)**
   This layer is a standalone server application that contains all the business logic.

   Purpose:

   - To expose a secure and efficient API for the Presentation Layer (Frontend).
   - To implement all core business logic, data validation, and processing.
   - To orchestrate communication between the client, the database, and external services.

Technologies:

- **Framework:** NestJS (running on Node.js with Express under the hood).
- **Authentication:** JWT (JSON Web Tokens) for securing API endpoints.
- **File Handling:** Multer for handling file uploads (images, audio).

Structure:

- **Modules:** The application is organized into modules based on features (e.g., AuthModule, UsersModule, MemoriesModule).
- **Controllers:** Handle incoming HTTP requests, validate request data (using DTOs), and call the appropriate services. (e.g., MemoriesController).
- **Services:** Contain the core business logic. They are responsible for processing data and interacting with the Data Layer. (e.g., MemoriesService).
- **Middleware:** Used for cross-cutting concerns like authentication checks, logging, and error handling.

3. **Data Access Layer (Prisma ORM + PostgreSQL)**
This layer is responsible for all aspects of data persistence and retrieval.

Purpose:

- To provide a consistent and abstract interface for the Application Layer to interact with data storage.
- To manage the structure, storage, and integrity of the application's data.

Technologies:

- **Database:** PostgreSQL (for structured, relational data like users, memories, etc.).
- **ORM (Object-Relational Mapping):** Prisma ORM, providing a type-safe client to query the database.

Structure:

- **Schema:** The entire database schema (tables, columns, relationships) is defined in a single schema.prisma file.
- **Prisma Client:** The Application Layer uses the auto-generated Prisma Client to perform all CRUD (Create, Read, Update, Delete) operations on the database. This abstracts away the raw SQL queries.

4. **External Integration Layer**
This layer consists of all third-party services that the SoulNote application depends on.

Purpose:

- To offload specific functionalities to specialized external providers, such as authentication, file storage, and notifications.

External Services:

- **Authentication:** Firebase Authentication (for user sign-up, sign-in, and social logins like Google).
- **File Storage:** Cloudinary (for storing user-uploaded media like images and audio).
- **Notifications:** Firebase Cloud Messaging (for sending push notifications and reminders).
- **Geolocation:** (Optional) Google Maps API or a similar service for tagging memories with location data.
- **Emotion Analysis:** (Optional) An external text/image analysis API.

Communication:

- The Application Layer (Backend) communicates with these services via their respective SDKs or REST APIs over HTTPS.

5. **Database Layer**

Purpose:

- Manages persistent, structured data storage and indexing.

Technology:

- PostgreSQL

Advantages:

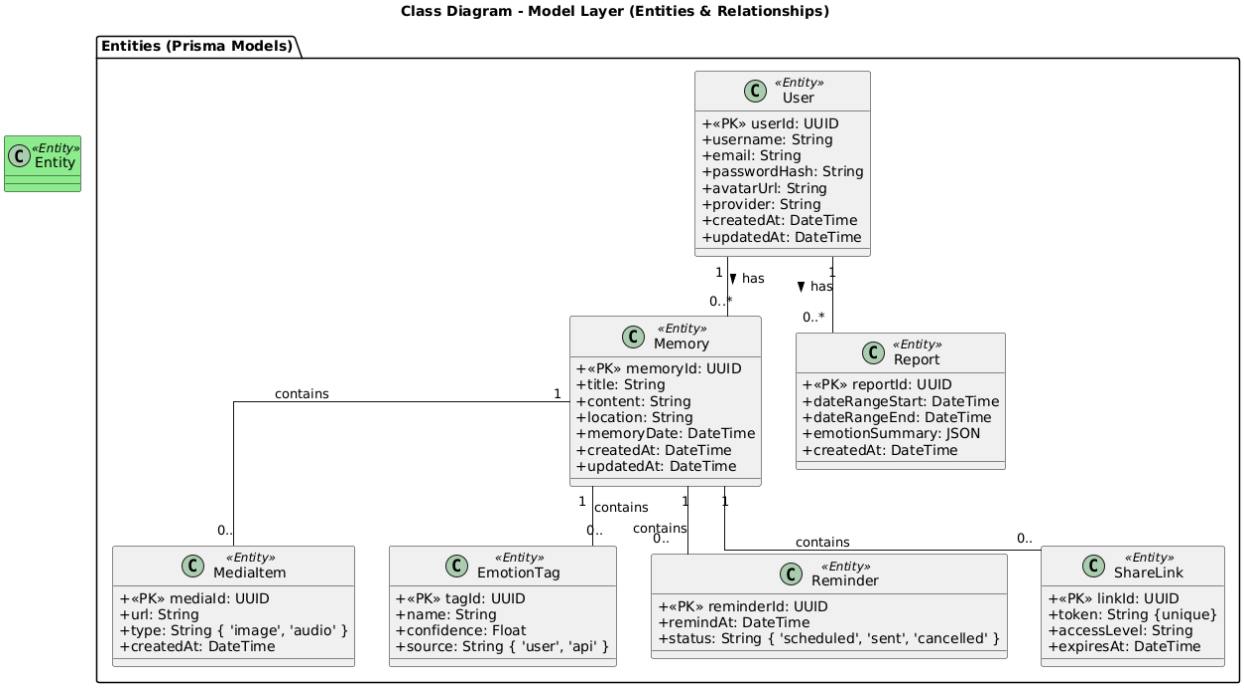
- Strong relational integrity and support for foreign keys
- Efficient SQL querying
- Compatibility with advanced features such as views, triggers, and functions
- Type-safe integration via Prisma ORM

Summary Table of Layer Responsibilities:

Layer	**Responsibility**
Presentation Layer	Renders the UI in the user's browser (SPA). Handles user interactions, client-side state, and makes API calls to the Application Layer.
Application Layer	Business logic, routing, validation, service coordination
Data Access Layer	Data schema definition and CRUD operations
External Integration	Connects with emotion, auth, and notification services
Database Layer	Stores persistent user and memory data in PostgreSQL

4.1 Component: Model

4.1.1 Class diagram overview



4.1.2 Description of key classes

a. User

- **Function:** Manages user accounts, including personal details, authentication methods, and related account activities.
- **Attributes:**
    - userId (UUID): Unique identifier for the user.
    - username (String): Display name or nickname chosen by the user.
    - email (String): Email address registered to the user’s account.
    - passwordHash (String): Secure hash of the user’s password.
    - avatarUrl (String): URL to the user’s profile picture.
    - provider (String): Authentication method used (e.g., "google", "password").
    - createdAt (DateTime): Date and time when the account was created.
    - updatedAt (DateTime): Last time the user’s profile information was updated.

- **Methods:**
    - register(): Registers a new user with provided credentials.
    - login(): Authenticates the user during sign-in.
    - updateProfile(data): Updates user information such as avatar or username.
    - deleteAccount(): Deletes or deactivates the user's account.

b. Memory

- **Function:** Manages the core content of a memory entry created by the user, including its metadata and associations.
- **Attributes:**
    - memoryId (UUID): Unique identifier for the memory.
    - title (String): Title of the memory.
    - content (String): Detailed description or narrative of the memory.
    - location (String): Location associated with the memory.
    - memoryDate (DateTime): The date and time when the memory occurred.
    - createdAt (DateTime): Timestamp when the memory was created.
    - updatedAt (DateTime): Timestamp of the last modification to the memory.
- **Methods:**
    - create(data): Adds a new memory to the system.
    - edit(data): Updates the details or content of an existing memory.
    - delete(): Removes the memory from the system.
    - getMediaItems(): Retrieves all media items attached to this memory.

c. MediaItem

- **Function:** Handles the metadata and storage details for media files (images or audio) attached to memories.
- **Attributes:**
    - mediaId (UUID): Unique identifier for the media item.

- url (String): Public or protected URL pointing to the media file.
- type (String): Media type, either 'image' or 'audio'.
- createdAt (DateTime): Timestamp when the media item was uploaded.

- **Methods:**
  - upload(file): Uploads a new media file to the system.
  - delete(): Deletes the media item.
  - getType(): Returns the type of the media item.

d. EmotionTag

- **Function:** Represents emotional labels associated with memories, and manages their origin and confidence level.
- **Attributes:**
  - tagId (UUID): Unique identifier for the emotion tag.
  - name (String): Name of the emotion (e.g., "joy", "fear").
  - confidence (Float): Confidence score from the detection engine.
  - source (String): Indicates the origin, either 'user' or 'api'.
- **Methods:**
  - attachToMemory(memoryId): Links this emotion tag to a memory.
  - updateConfidence(value): Updates the confidence score.
  - getSource(): Returns the source of the tag.

e. Reminder

- **Function:** Manages scheduled notifications to remind users of specific memories at defined times.
- **Attributes:**
  - reminderId (UUID): Unique identifier for the reminder.
  - remindAt (DateTime): Scheduled time when the reminder should trigger.
  - status (String): Status of the reminder ('scheduled', 'sent', or 'cancelled').

- **Methods:**
    - schedule(time): Sets a new reminder for a specific time.
    - cancel(): Cancels the reminder.
    - reschedule(newTime): Changes the reminder to a new scheduled time.

f. Report

- **Function:** Aggregates and presents emotion trends from user memories over a time period.
- **Attributes:**
    - reportId (UUID): Unique identifier for the report.
    - dateRangeStart (DateTime): Start date of the reporting period.
    - dateRangeEnd (DateTime): End date of the reporting period.
    - emotionSummary (JSON): Summary of emotional states during the period.
    - createdAt (DateTime): Timestamp when the report was generated.
- **Methods:**
    - generate(fromDate, toDate): Generates a report using data from a specific range.
    - export(format): Exports the report in a specified format (e.g., PDF, CSV).
    - getSummary(): Returns a structured overview of emotions in the report.

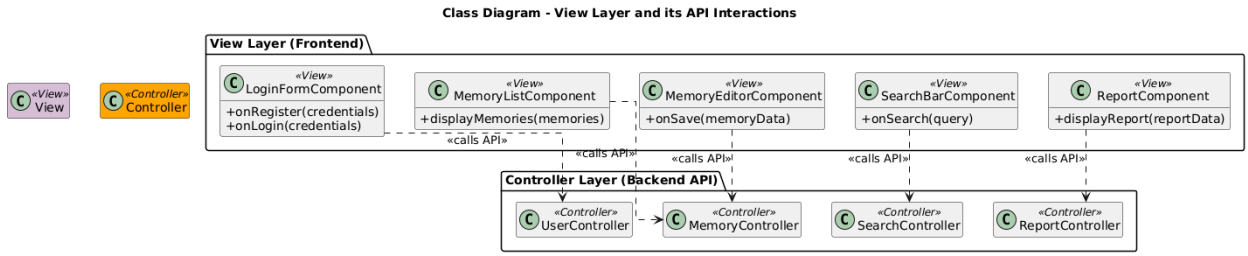
g. ShareLink

- **Function:** Creates and manages secure, time-limited shareable links for accessing specific memories.
- **Attributes:**
    - linkId (UUID): Unique identifier for the share link.
    - token (String): Unique secure token used for accessing the memory.
    - accessLevel (String): Defines who can view the memory ('public', 'private').
    - expiresAt (DateTime): Expiration date and time of the link.
- **Methods:**
    - generate(memoryId, options): Generates a new shareable link for a memory.

- validateToken(token): Validates the provided token to check access.
- revoke(): Disables the share link before its expiration.

4.2 Component: View

4.2.1 Class diagram overview



4.2.2 Description of key classes

a. LoginFormComponent

- **Function:** Functions as the user interface for authentication. It collects user credentials (email, password) for both registration and login processes.
- **Methods:**
  - onRegister(credentials): Triggered when the user submits the registration form. It gathers the input data and sends an API request to the UserController to create a new account.
  - onLogin(credentials): Triggered when the user submits the login form. It collects the credentials and sends an API request to the UserController to authenticate the user.

b. MemoryListComponent

- **Function:** Responsible for displaying a list or grid of memories to the user. It fetches this data by communicating with the MemoryController.
- **Methods:**
  - displayMemories(memories): Renders the list of memory objects received from the backend. Each item in the list is typically a summary (card view) that the user can click to see more details.

c. MemoryEditorComponent

- **Function:** Provides a form-based interface for users to create a new memory or edit an existing one. It

includes fields for text, images, emotions, and tags.

- **Methods:**
    - onSave(memoryData): Gathers all data from the memory form and sends it via an API call to the MemoryController to be saved or updated in the database.

d. SearchBarComponent

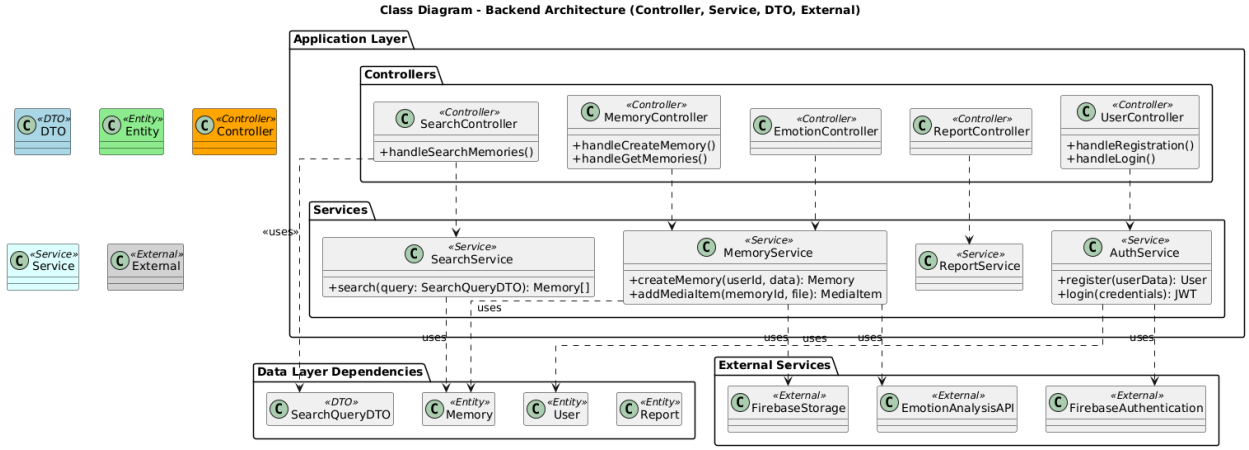
- **Function:** A dedicated UI component that allows users to input search queries or apply filters (like keywords, dates, or emotions) to find specific memories.
- **Methods:**
    - onSearch(query): Captures the user's search criteria and initiates an API request to the SearchController to retrieve a filtered list of memories.

e. ReportComponent

- **Function:** Responsible for rendering visual representations of user data, such as emotion trend charts. It fetches aggregated data from the ReportController.
- **Methods:**
    - displayReport(reportData): Receives processed report data (e.g., data points for a chart) from the backend and uses a charting library to display it graphically to the user.

4.3 Component: Controller

4.3.1 Class diagram overview



4.3.2 Description of key classes

a. UserController

- **Function:** Handles user authentication-related requests, such as registration and login.
- **Methods:**
  - handleRegistration(): Receives new user data from the request, then calls the AuthService to create the account.
  - handleLogin(): Receives login credentials from the request, then calls the AuthService to authenticate the user and returns an access token (JWT).

b. MemoryController

- **Function:** Handles core operations for memories, such as creating and retrieving them.
- **Methods:**
  - handleCreateMemory(): Receives data for a new memory from the request and calls MemoryService to save it.
  - handleGetMemories(): Calls MemoryService to retrieve a list of the user's memories and returns it to the client.

c. SearchController

- **Function:** Handles all incoming search requests for memories.
- **Methods:**
  - handleSearchMemories(): Receives search criteria encapsulated in a SearchQueryDTO and passes it to the SearchService to execute the search.

d. EmotionController

- **Function:** Handles requests related to emotion analysis for memories. It delegates the complex logic to the MemoryService.

e. ReportController

- **Function:** Handles requests to generate and retrieve user-specific reports. It calls the ReportService to perform the actual data aggregation.

f. AuthService

- **Function:** Contains the core business logic for user authentication. It interacts with the User entity and external authentication providers.
- **Methods:**
    - register(userData): Validates user data, hashes the password, creates a new User record in the database, and returns the created user.
    - login(credentials): Verifies user credentials against the database and returns a JSON Web Token (JWT) upon success.

g. MemoryService

- **Function:** Manages all business logic related to memories, including database interactions and communication with external services for media and emotions.
- **Methods:**
    - createMemory(userId, data): Creates and saves a new Memory entity to the database for a specific user.
    - addMediaItem(memoryId, file): Uploads a media file to an external storage service (like Cloudinary) and associates its URL with a specific memory.

h. SearchService

- **Function:** Implements the complex logic for searching through memories based on various criteria.
- **Methods:**
    - search(query: SearchQueryDTO): Constructs and executes a database query based on the SearchQueryDTO and returns a list of matching Memory entities.

i. ReportService

- **Function:** Contains the business logic for aggregating user data (like memories and emotions) over a

period to generate insightful reports.

j. SearchQueryDTO

- **Function:** Acts as a structured data container (Data Transfer Object) to pass search parameters from the SearchController to the SearchService.
- **Attributes:**
    - keyword: The search term to match in memory content.
    - tags: An array of emotion tags to filter the results.

k. User

- **Function:** Represents a user's record in the database. It is used by AuthService for authentication and authorization.

l. Memory

- **Function:** Represents a single memory's record in the database. It is the core data entity managed by MemoryService and SearchService.

m. Report

- **Function:** Represents a generated report's record in the database. It stores the summary of a user's activity for a specific period.

n. FirebaseAuthentication

- **Function:** Represents an external, third-party service for handling user authentication, such as sign-in with Google or email/password verification. It is used by AuthService.

o. Cloudinary

- **Function:** Represents an external, third-party service for storing and retrieving files, such as images and audio clips. It is used by MemoryService.

p. EmotionAnalysisAPI

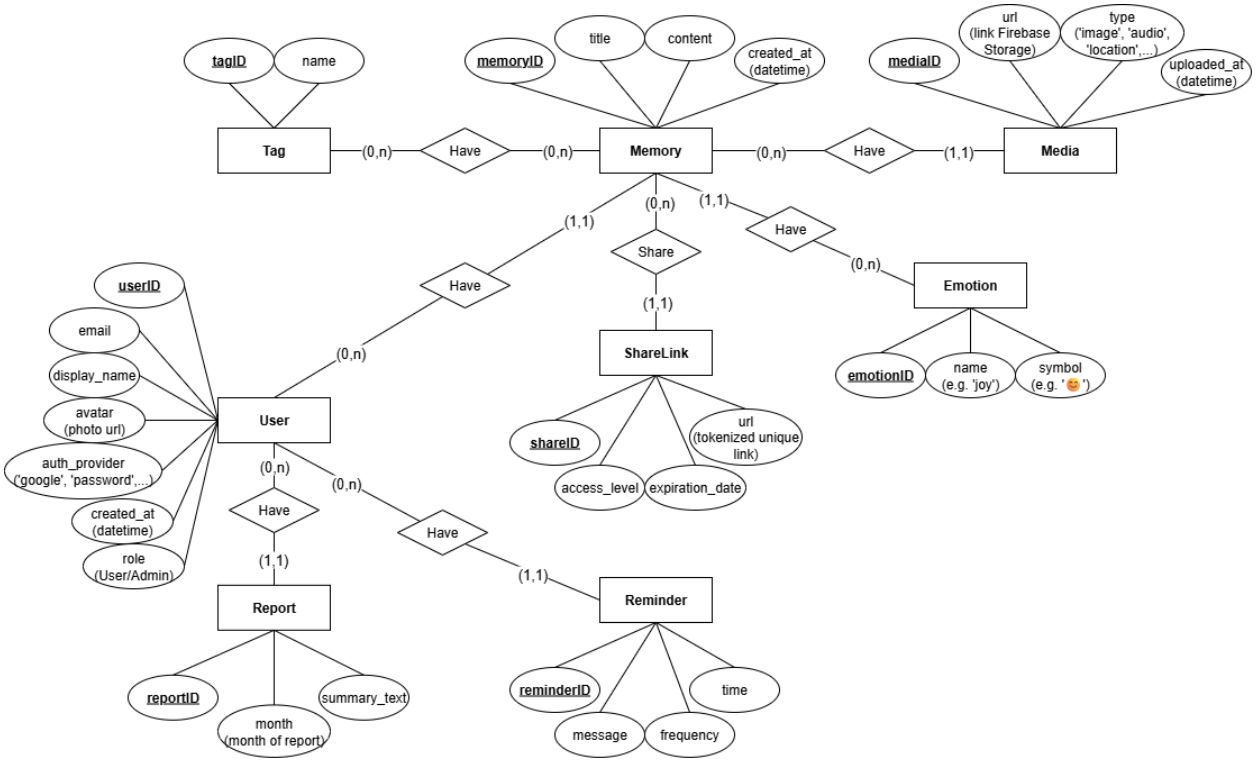
- **Function:** Represents an external, third-party API that analyzes text or images to detect emotions. It is used by MemoryService for automatic tagging.

4.4 Component: Database

This section describes the database design of the SoulNote application. The database was designed based on the documented requirements and user stories, focusing on capturing personal memories, organizing them with emotional tags, and supporting reminders and analytics.

The database follows a **relational schema** using **PostgreSQL** and is managed via **Prisma ORM** on the backend (NestJS). The schema ensures data consistency, normalization, and scalability.

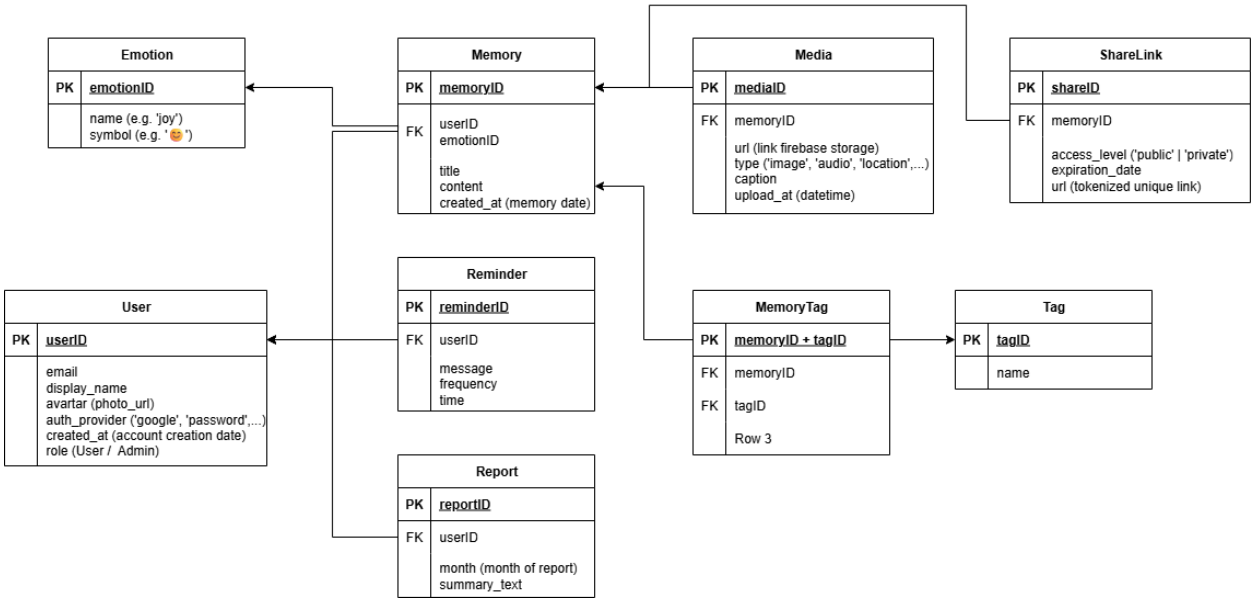
4.4.1 Entity-Relationship Model



The ER model of SoulNote includes the following core entities:

- `User`
- `Memory`
- `Tag`
- `Emotion`
- `Media`
- `Reminder`
- `Report`
- `ShareLink`
- `MemoryTag` (association entity for many-to-many)

Each entity contains relevant attributes and is connected to others through well-defined relationships, reflecting real-world interactions within the application.



4.4.2 Entity Descriptions

- **User**
  Represents a registered account in SoulNote.
  - **Primary Key:** `userID`
  - **Attributes:** `email`, `display_name`, `avatar (photo URL)`, `auth_provider`, `created_at`, `role (User/Admin)`

  Users can authenticate via email/password or third-party login (e.g., Google) using Firebase Auth

- **Memory**
  Represents a single memory created by a user.
  - **Primary Key**: `memoryID`
  - **Foreign Key**: `userID → User`
  - **Attributes**: `title`, `content`, `emotion` (e.g., happy, sad), `created_at`

  Each memory may have multiple media files and tags.

- **Emotion**
  Represents an emotion associated with a memory.
  - **Primary Key**: `emotionID`
  - **Attributes**: `name` (e.g., "joy", "sadness"), `symbol` (e.g., "😊"), `color` (for UI display)

  This table allows for standardized emotional labeling and visual representation

- **Tag**
  Represents a category or label assigned to a memory.
  - **Primary Key:** `tagID`
  - **Attributes:** `name`

- **MemoryTag**
  A bridge (junction) table that establishes a many-to-many relationship between `Memory` and `Tag.`
  - **Composite Primary Key**: `memoryID + tagID`

- **Foreign Keys :**
  - `memoryID → Memory`
  - `tagID → Tag`

This structure allows a memory to have multiple tags, and a tag to be reused by many memories

- **Media**

Represents media files (images, audio, location previews...) attached to a memory.

- **Primary Key:** `mediaID`
- **Foreign Key:** `memoryID → Memory`
- **Attributes:** `url` (**Cloudinary** link), `type` (image/audio/location), `caption`, `uploaded_at`

Each media belongs to exactly one memory.

- **Reminder**

Represents optional reminders set by a user to revisit or create new memories.

- **Primary Key:** `reminderID`
- **Foreign Key:** `userID → User`
- **Attributes:** `message`, `frequency` (e.g., daily, weekly), `time`

Each user may configure personal reminders

- **Report**

Represents monthly summaries and emotional statistics.

- **Primary Key**: `reportID`
- **Foreign Key**: `userID → User`
- **Attributes**: `month`, `summary_text`

These reports are generated by the system based on user memory activities in the corresponding month.

- **ShareLink**

Represents a shareable link generated for a memory.

- **Primary Key:** `shareID`
- **Foreign Key:** `memoryID → Memory`
- **Attributes:** `url` (tokenized link), `access_level` (`public`, `private`), `expiration_date`

This allows a memory to be shared externally in a secure and controlled manner, similar to platforms like Notion or TikTok.

4.4.3 Relationship Summary

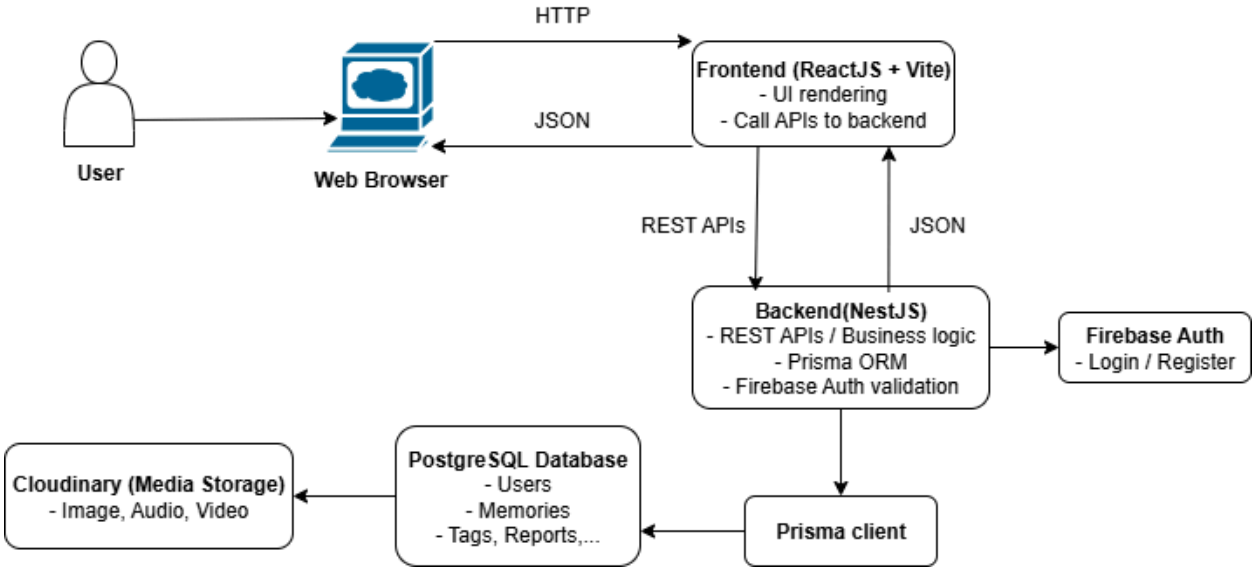
Entity	**Relationship**	**Type**
User–Memory	One-to-Many	(1:N)
Memory–Media	One-to-Many	(1:N)
Memory–Tag	Many-to-Many *(via MemoryTag)*	(M:N)
Memory-Emotion	Many-to-One *(via FK emotionID)*	(N:1)
Memory–ShareLink	One-to-Many (if multiple links allowed)	(1:N)

User–Reminder	One-to-One or One-to-Many *(optional)*	(1:1) or (1:N)
User–Report	One-to-Many *(monthly summaries)*	(1:N)

4.4.4 Technology Stack

- **Database System**: PostgreSQL
- **ORM**: Prisma (Type-safe query builder)
- **Storage for Media**: Cloudinary Cloud (links stored in `Media.url`)
- **Authentication Data**: Managed via Firebase Auth (only essential fields stored locally)

5. Deployment

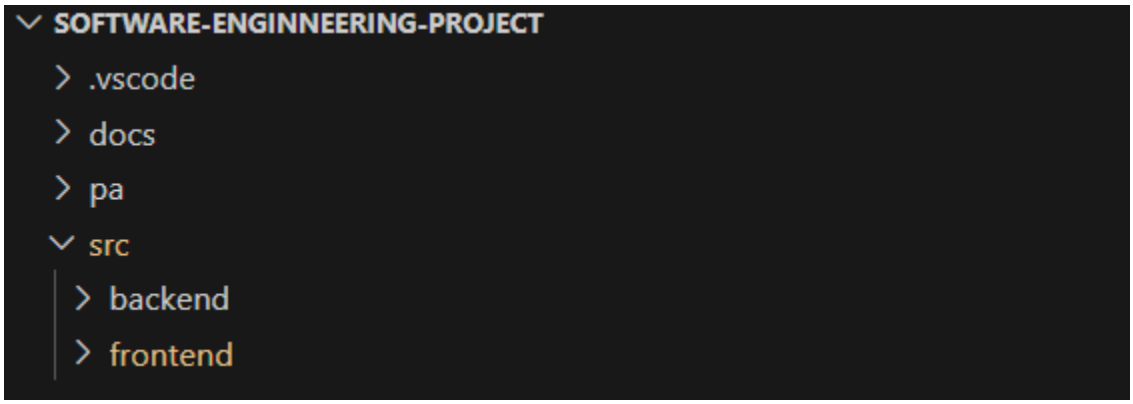


- **Frontend (Client-side)**
  - **Technology**: ReactJS (using Vite)
  - **Deployed on**: Web browsers (Google Chrome, Firefox, Edge, Safari, etc.)
  - **Devices supported**: Laptops, PCs, Tablets, and Mobile phones
  - **Platform**: The frontend is hosted on a static web hosting service
  - **Interaction**: Users access the SoulNote application through a browser, interacting with the user interface to manage memories, upload media, and generate reports.
- **Backend (Server-side)**
  - **Technology**: NodeJS with NestJS framework
  - **Deployed on**: Cloud server or VPS
  - **Responsibilities**:
    - Handle REST API requests from frontend
    - Authenticate users (via Firebase Auth)
    - Connect to the database (PostgreSQL)
    - Handle file-related logic (in coordination with Cloudinary)
- **Database**
  - **Technology**: PostgreSQL with Prisma ORM
  - **Deployed on**: Cloud database service (e.g., Supabase PostgreSQL, NeonDB, or Railway DB)
  - **Usage**:
    - Store user accounts, memories, emotions, media metadata, reminders, and reports

        - Perform queries, joins and manage relationships via Prisma Client
- **Cloud Storage**
    - **Technology:** Cloudinary
    - **Deployed on:** Cloudinary Cloud
    - **Purpose:**
        - Store media files (images, audio, video)
        - Allow secure download/upload via signed URLs
- **Authentication**
    - **Technology:** Firebase Authentication
    - **Deployed on:** Firebase Auth cloud service
    - **Purpose:**
        - Authenticate users via email/password or third-party methods (Google, Facebook, etc.)
        - Issue access tokens to be used in frontend-backend communication

6. Implementation View

6.1 Folder Structure Overview

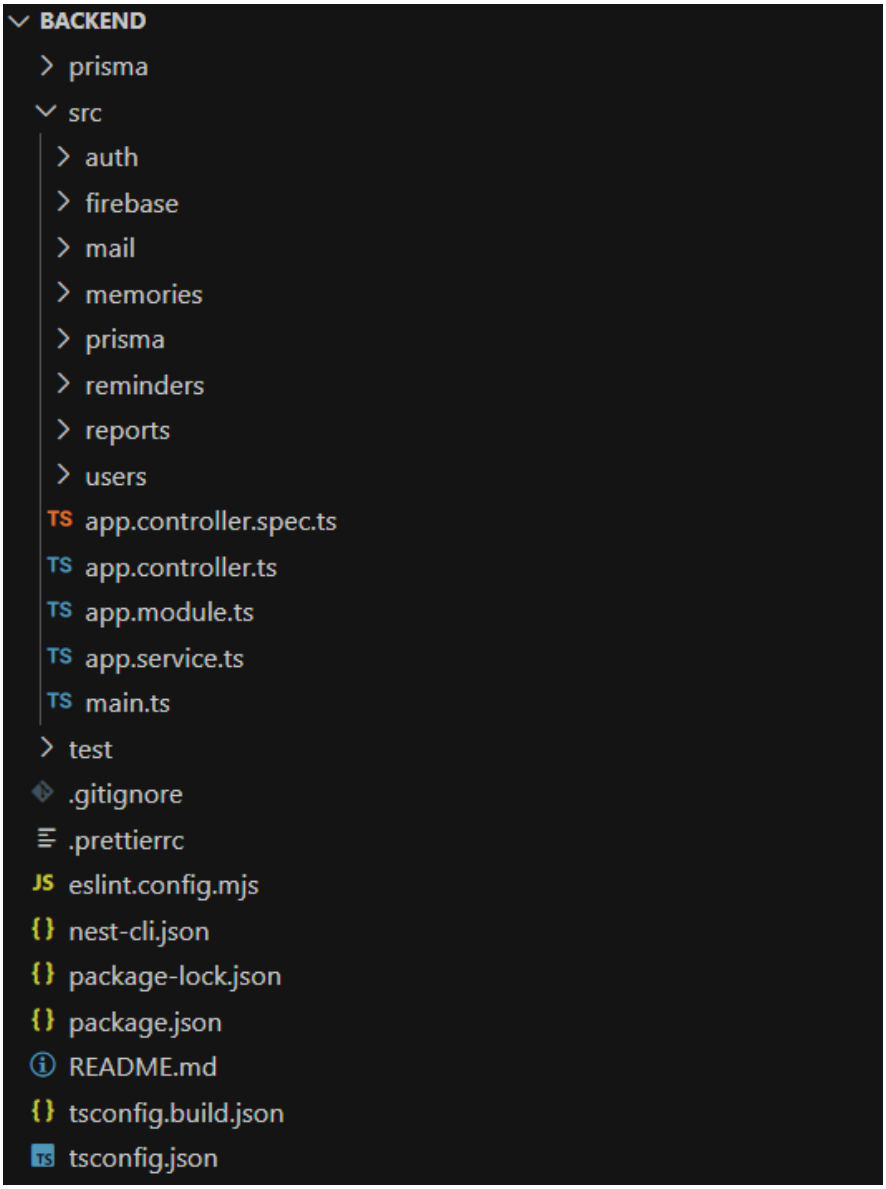


- **`.vscode/`**: Contains editor configuration and workspace settings to help the development team maintain consistent code style and tools.
- **`docs/`**: Contains project documentation, including architecture documents, user manuals, diagrams, etc.
- **`pa/`**: Be used for planning artifacts (e.g., prototypes, analysis documents, revised documents, weekly report, planning report).
- **`src/`**: Contains all source code, divided into two main parts:
    - **`frontend/`**: The client-side **React** application responsible for rendering the user interface and managing user experience.
        - Built with **React**, **TypeScript**, **Tailwind CSS**, and **React Router**.

- Handles client-side routing, user interaction, and local state management.
- Communicates with backend via **REST APIs** and Firebase services.
- Implements responsive design to support both desktop and mobile devices.

- **`backend/`**: The server-side **Node.js** application, implemented with **NestJS** and **Prisma ORM**.
  - Provides RESTful APIs for the frontend to fetch, create, update, and delete data.
  - Handles core business logic, authentication, authorization, and integration with Firebase (e.g., for notifications).
  - Connects to a **PostgreSQL** database using Prisma for data persistence.
  - Follows modular architecture for scalability and maintainability.

6.2 Backend



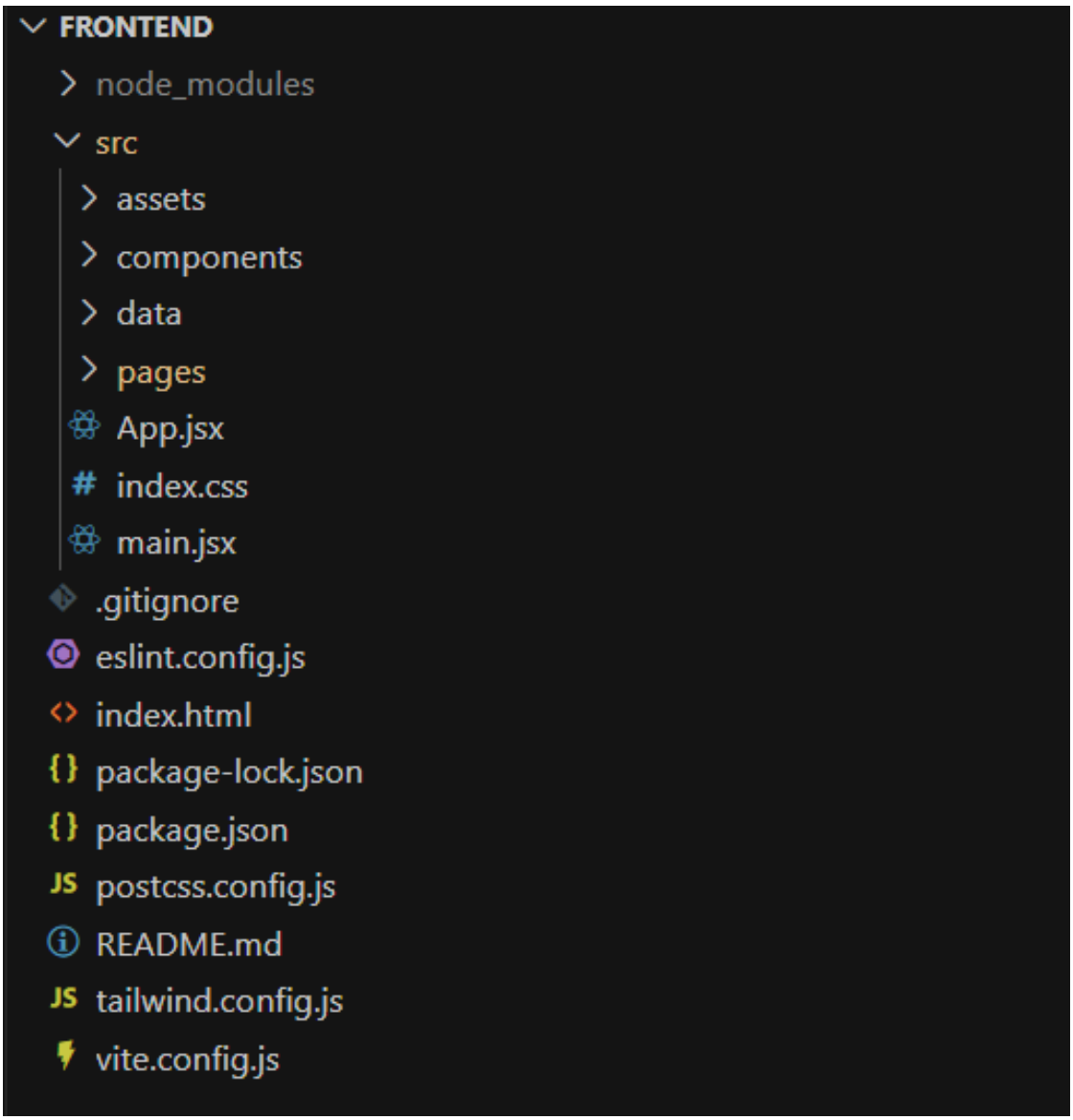
- **prisma/**
  Contains Prisma schema files and migration files.
  Used for defining the database models, relationships, and running migrations.

- **src/**
  Main source code of the backend application, organized into modules following NestJS's modular architecture:

  - **auth/**: handles user authentication logic (login, register, JWT strategies, guards).

    - **firebase/**: integrates Firebase services, such as Authentication, Storage, or Cloud Messaging.
    - **mail/**: manages sending emails for verification, reminders, or notifications.
    - **memories/**: handles CRUD operations and business logic for user-created memories (text, photo, audio).
    - **prisma/**: defines the Prisma client and database service, used to query PostgreSQL.
    - **reminders/**: manages scheduled reminders and notifications.
    - **reports/**: handles reporting features or data analytics.
    - **users/**: manages user profiles and related user data.
- **app.controller.ts**: root controller handling basic routes like health checks or landing endpoints.
- **app.module.ts**: root module that imports and configures all feature modules.
- **app.service.ts**: root service providing basic shared business logic.
- **main.ts**: the application entry point. It starts the NestJS server and applies global middleware and settings.
- **app.controller.spec.ts**: test file for the root controller, using Jest.
- **test/**: contains additional automated tests.
- **.gitignore**: specifies files and folders that should be ignored by Github.
- **.prettierrc**: Prettier configuration for code formatting.
- **eslint.config.mjs**: ESLint configuration to enforce coding standards.
- **nest-cli.json**: configuration file for Nest CLI commands and build options.
- **package.json**: lists project metadata, scripts, and npm dependencies.
- **package-lock.json**: locks dependency versions to ensure consistent installations.
- **tsconfig.build.json** & **tsconfig.json**: TypeScript compiler settings for development and build.
- **README.md**: project documentation, setup guide, and contribution notes.

6.3 Frontend



- **assets/**: contains static assets such as images, icons, fonts, and other media used in the user interface.
- **components/**: holds reusable React components like **`NavigationBar`**, **`Search`**,, etc.These components help break down the interface into manageable, maintainable parts.
- **data/**: stores static data, JSON files, mock data, or constants used within the frontend.
  Useful for development, testing, or displaying sample content.

- **pages/**: includes top-level pages of the application, each file typically represents a screen or route, e.g., **`Dashboard`, `SettingPage`, `AboutPage`**,**DeleteAccountPage,** ...
- **App.jsx**: the root React component. Defines the overall layout and sets up page routing (using React Router, if applied).
- **main.jsx**: the entry point of the React application. Initializes and renders **`App.jsx`** into the root DOM element.
- **index.css**: contains global CSS, typically includes Tailwind CSS base styles and additional custom styling.
- **index.html**: the base HTML file provided by Vite, containing the **`<div id="root">`** where the React app mounts.
- **tailwind.config.js**: Tailwind CSS configuration file, where custom colors, fonts, and breakpoints are added.
- **vite.config.js**: Vite configuration file. Defines build settings, plugins, and other project-specific options.
- **postcss.config.js**: configuration for PostCSS, required for Tailwind CSS and other CSS plugins.
- **eslint.config.js**: configuration for ESLint, enforcing code style and helping to catch potential issues early.
- **package.json**: contains project metadata, dependencies, and npm scripts like **`npm run dev`, `npm run build`.**
- **package-lock.json**: automatically generated to lock dependency versions, ensuring consistent installs across environments.
- **README.md**: project documentation including setup instructions, usage, and deployment notes.