Life Copilot Project Structure

1. Introduction

This document aims to provide an overview of the proposed technological structure for the Life Copilot application, a comprehensive AI-powered solution designed to help users manage various aspects of their lives such as time, health, emotions, finances, and decision-making. This structure will cover the main components of the system, proposed technologies, data flow, and interactions between components to ensure scalability, performance, and security.

2. Key System Components

The Life Copilot system consists of several key components that work together to provide a seamless and intelligent user experience. These components can be divided into Frontend, Backend, Database, AI/ML Models, and Cloud Infrastructure.

2.1. Frontend

The frontend is the direct interaction point with users. It must be intuitive, responsive, and provide an excellent user experience across various devices (smartphones and tablets).

Proposed Technologies & Components:

- **Rich Text Editor (ProseMirror + Tiptap):**
    - ProseMirror manages browser inconsistencies in `contentEditable` and supports document nodes, selection, and transactions.
    - Tiptap acts as a high-level wrapper over ProseMirror for easier implementation.
    - Supports block types: paragraphs, headings, lists, images, videos, toggles, drag-and-drop, etc.
- **State Management Strategy:**
    - **Redux:** Used for synchronous client-side state like UI controls and time-travel debugging.
    - **React Query:** Handles asynchronous data: fetching, caching, mutations, optimistic updates, and background refetching.
    - Together, they provide a seamless and responsive collaboration experience with offline support.

- **Authentication Flow:**
    - Follows backend's Access/Refresh token strategy.
    - Axios interceptors handle automatic token refresh on expiration.
- **Design & Theming:**
    - **SCSS** combined with **React Context API** for light/dark mode support.
    - **Unsplash API** and **Firebase Storage** are integrated for uploading images (profile, cover, inline).
- **Project Structure & Tooling:**
    - Frontend built with **React** and **TypeScript**.
    - Tools used: **ESLint**, **Prettier**, **Husky** (pre-commit), **Jest** with code coverage.
    - Frontend code is open-source and available on GitHub.

Key Frontend Features:

- Interactive user interface for displaying schedules, health reports, expense summaries, and recommendations.
- Easy data entry for tasks, meals, expenses, and mood status.
- Customizable notifications and reminders.
- Personalized dashboards displaying summaries and insights.
- Dark Mode support and customization options.
- Rich text editing capabilities for notes and journal entries.

2.2. Backend

The backend is the brain of the application, handling business logic, data management, user authentication, and AI model integration.

Proposed Technologies & Components:

- **Clean Architecture Layers:**
    - **Domain Layer:** Contains the core business logic and entities. No external dependencies.
    - **Application Layer:** Coordinates business use cases. Interfaces with domain layer and infrastructure.
    - **Infrastructure Layer:** Handles external concerns like database access and third-party services. Implements repository interfaces and validation logic.
    - **Main Layer:** Entry point of the application (e.g., Express.js setup). Connects all layers and manages dependency injection.

- **Authentication Model:**
    - Uses **Access Tokens** and **Refresh Tokens**.
    - Tokens are stored in **HTTP-only cookies** for enhanced security.
    - Refresh mechanism is handled automatically via HTTP interceptors (e.g., Axios).
- **Separation of Concerns:**
    - Application logic is decoupled from technical implementation.
    - Promotes testability, maintainability, and clean structure.
    - Reduces tight coupling between layers.
- **Comprehensive Testing (TDD):**
    - **Unit Tests:** Test business logic in isolation (e.g., domain layer).
    - **Integration Tests:** Validate communication between components (e.g., application + infrastructure).
    - **End-to-End Tests:** Test full user flow through HTTP endpoints (e.g., using Supertest).
- **API Documentation:**
    - Every endpoint is fully documented.
    - Includes request/response samples.
    - Clearly outlines authentication flow and cookie usage.
    - Enhances team communication and client integration.

Key Backend Features:

- User management, authentication, and authorization.
- Processing and storing data from the frontend.
- Integration with AI models to generate recommendations and analyses.
- Providing RESTful / GraphQL APIs for the frontend.
- Logging and monitoring mechanisms.

2.3. Database

The database is the central data repository for the application, storing all user data, schedules, expenses, health data, and recommendations.

Proposed Technologies:

- **Relational Database:**
    - **PostgreSQL:** For its power, reliability, support for complex data (JSONB), and scalability. Suitable for storing structured data such as user profiles, schedules, and expenses.
- **NoSQL Database (for specific data):**
    - **MongoDB / Cassandra:** If there is a need to store large amounts of unstructured or semi-structured data quickly, such as mood tracking data or daily activity logs that may not fit perfectly into a strict relational model.
- **Caching Layer:**
    - **Redis:** For caching frequently accessed data, session storage, and real-time features.

Key Database Features:

- Secure and reliable storage of user data.
- Support for complex queries for data analysis and report generation.
- Horizontal and vertical scalability to handle data growth.
- Backup and recovery mechanisms to ensure data integrity.
- Optimized performance through caching strategies.

2.4. AI/ML Models

AI models are the core of the Life Copilot application, providing the intelligence needed to deliver personalized recommendations, analyses, and smart scheduling.

Proposed Technologies:

- **ML Frameworks:**
    - **TensorFlow / PyTorch:** For building and training deep learning models for emotional analysis, pattern recognition in health data, and recommendation generation.

- **ML Libraries:**
    - **Scikit-learn:** For traditional machine learning tasks such as classification, regression, and clustering for scheduling and expense data.
- **Natural Language Processing (NLP):**
    - **SpaCy / NLTK / Hugging Face Transformers:** For analyzing user text inputs (e.g., mood notes) and extracting information from them.

Key AI/ML Model Features:

- **Smart Scheduling:** Analyzing user habits, priorities, and commitments to create optimized schedules.
- **Health and Nutrition Recommendations:** Based on user data (activity, meals), providing personalized health and nutrition tips.
- **Emotion Analysis:** Analyzing mood entries to identify patterns and provide insights into emotional well-being.
- **Smart Expense Management:** Analyzing spending patterns and providing recommendations to improve budgeting.
- **General Life Recommendations:** Offering suggestions to improve productivity, decision-making, and achieving personal goals.

2.5. Cloud Infrastructure

Cloud infrastructure is essential for hosting all application components and ensuring scalability, reliability, and security.

Proposed Technologies:

- **Cloud Service Provider:**
    - **AWS / Google Cloud Platform (GCP) / Microsoft Azure:** These platforms offer a wide range of services (compute, storage, databases, AI services) that support building and operating scalable applications.
- **Compute Services:**
    - **Kubernetes (EKS/GKE/AKS):** For managing and deploying microservices in containers to ensure automatic scalability and high availability.
    - **AWS Lambda / Google Cloud Functions:** For serverless functions for specific tasks or event processing.

- **Storage Services:**
    - **Amazon S3 / Google Cloud Storage:** For storing large files such as images, videos, or trained models.
    - **Firebase Storage:** Integrated for frontend image uploads and management.
- **Managed Database Services:**
    - **Amazon RDS (PostgreSQL) / Google Cloud SQL (PostgreSQL):** For easy management of relational databases.
- **Managed AI/ML Services:**
    - **Amazon SageMaker / Google AI Platform:** For efficiently developing, training, and deploying AI models.
- **Third-Party Integrations:**
    - **Unsplash API:** For high-quality stock images.
    - **Firebase Services:** For authentication, storage, and real-time features.

3. Data Flow and Interactions

This section illustrates how data flows between the different components of the system and how they interact with each other.

1. **User Interaction with Frontend:** The user enters data (tasks, meals, expenses, mood) via the web/mobile application using React components.
2. **State Management:** Redux manages UI state while React Query handles server state and caching.
3. **Data Transmission to Backend:** The frontend sends data to the backend via secure APIs (HTTPS) with automatic token refresh.
4. **Clean Architecture Processing:** The backend processes requests through its layered architecture (Main → Application → Domain → Infrastructure).
5. **Data Processing and Storage:** The backend validates and stores data in the appropriate database (PostgreSQL for structured information, MongoDB for unstructured data).
6. **AI Integration:**
    - When recommendations or analyses are needed (e.g., smart scheduling, mood analysis), the backend calls the relevant AI models.
    - AI models process the data received from the database or backend and generate recommendations/analyses.
    - AI models return the results to the backend.

7. **Sending Recommendations to Frontend:** The backend sends recommendations and analyses to the frontend for display to the user.
8. **Real-time Updates:** React Query handles background refetching and optimistic updates for a responsive user experience.
9. **Notifications:** The backend sends notifications (via services like Firebase Cloud Messaging or Apple Push Notification service) to the frontend to remind users of tasks or provide updates.

4. Security and Scalability Considerations

4.1. Security

- **Encryption:** Encryption at Rest and Encryption in Transit using SSL/TLS.
- **Authentication and Authorization:**
    - HTTP-only cookies for token storage to prevent XSS attacks.
    - Access/Refresh token strategy with automatic renewal.
    - Secure authentication flow integrated between frontend and backend.
- **Access Management:** Implementing the Principle of Least Privilege for all components and services.
- **Vulnerability Scanning:** Conducting regular security scans and penetration testing.
- **Code Quality:** ESLint, Prettier, and Husky ensure code quality and security best practices.

4.2. Scalability

- **Modular Design:** Using microservices and clean architecture allows each service to be scaled independently.
- **Auto-scaling:** Utilizing cloud computing services that support auto-scaling based on load.
- **Scalable Databases:** Choosing databases that support horizontal scaling (sharding) or vertical scaling.
- **Caching:** Using Redis for caching frequently accessed data and reducing database load.
- **Frontend Optimization:** React Query provides intelligent caching and background updates to reduce server load.
- **Testing Strategy:** Comprehensive TDD approach ensures system reliability at scale.

5. Development and Deployment

5.1. Development Workflow

- **Version Control:** Git with GitHub for open-source frontend code.
- **Code Quality:** ESLint, Prettier, and Husky for consistent code formatting and pre-commit hooks.
- **Testing:** Jest for unit testing with code coverage requirements.

- **Documentation:** Comprehensive API documentation with request/response samples.

5.2. Deployment Strategy

- **Containerization:** Docker containers managed by Kubernetes for scalable deployment.
- **CI/CD Pipeline:** Automated testing and deployment pipeline.
- **Monitoring:** Comprehensive logging and monitoring across all layers.

6. Conclusion

This updated document presents a comprehensive technological structure for the Life Copilot application, incorporating modern frontend technologies like React with TypeScript, advanced state management with Redux and React Query, and a robust backend architecture following clean architecture principles. The system emphasizes security through HTTP-only cookies, comprehensive testing through TDD, and scalability through microservices and cloud infrastructure. This structure provides a solid foundation for building a production-ready, maintainable, and scalable life management application.