Expense Tracker Backend API: Improvised Requirements Document

1. Introduction

This document specifies the functional and non-functional requirements for a robust backend API designed to enable users to record, categorize, and analyze their personal expenses over time. The system will leverage artificial intelligence to provide insightful summaries and comparisons, along with structured data for dynamic data visualizations.

2. Scope

This document covers the backend API development, including authentication, user management, expense management, dashboard data provision, and AI microservice integration. Frontend development and the AI microservice's internal logic are outside the immediate scope of this document but are acknowledged as dependent components.

3. Technology Stack

- **Backend API:** ASP.NET Core Web API
- **Database:** PostgreSQL
- **AI Microservice (Future Integration):** Python Flask
- **AI Service Integration:** Google Gemini API

4. Objective

The primary objective is to build a secure, scalable, and maintainable RESTful API using ASP.NET Core that facilitates comprehensive expense tracking. The API will empower users with:

- Efficient recording, viewing, editing, and deletion of expense records.
- Advanced filtering and sorting capabilities for expense data.
- Endpoints to provide aggregated data suitable for insightful frontend charts and dashboards.
- AI-driven textual summaries and comparative insights on spending patterns across various dimensions (e.g., time periods, categories).

5. Functional Requirements

5.1. Authentication and Authorization (FR-AUTH)

- **FR-AUTH-100:** The API **MUST** implement JWT (JSON Web Token) based authentication for all protected endpoints.

- **FR-AUTH-101:** The system **MUST** support user registration with a unique email address and a secure password.
- **FR-AUTH-102:** The system **MUST** support user login, issuing an access token upon successful authentication.
- **FR-AUTH-103:** The authentication mechanism **MUST** utilize JWT Bearer tokens, incorporating both Access Tokens (short-lived) and Refresh Tokens (long-lived) to enhance security and user experience.
- **FR-AUTH-104:** Password hashing **MUST** be implemented using a strong, industry-standard algorithm (e.g., BCrypt) to securely store user credentials.
- **FR-AUTH-105:** The system **MUST** support a single user role: User. No administrative roles are required at this stage.

5.2. User Management (FR-UM)

- **FR-UM-100:** Users **MUST** be able to register for an account.
- **FR-UM-101:** Users **MUST** be able to log in to their account.
- **FR-UM-102:** Users **MUST** be able to log out, invalidating their current session.

5.3. Expense Management (FR-EXP)

- **FR-EXP-100: Expense Entry:** Users **MUST** be able to add new expense records with the following minimum fields:
    - Title or Description (string)
    - Amount (decimal/numeric)
    - Date (date/datetime, representing when the expense occurred)
    - Category (string, e.g., "Food", "Travel", "Utilities", "Rent", "Other"). The system should allow for predefined or user-defined categories.
    - *(Additional fields such as Currency, PaymentMethod, Notes can be considered)*
- **FR-EXP-101: Expense Modification:** Users **MUST** be able to edit any of their existing expense records.
- **FR-EXP-102: Expense Deletion (Soft Delete):** Users **MUST** be able to delete their expense records. This operation **MUST** be implemented as a soft delete using an IsDeleted flag, rather than permanent removal from the database.
- **FR-EXP-103: View Expenses:** Users **MUST** be able to retrieve a list of their expense records.
- **FR-EXP-104: Expense Filtering:** The API **MUST** provide endpoints to filter expenses by:
    - Date range (e.g., startDate, endDate)
    - Category
    - *(Other suitable fields from the expense model)*
- **FR-EXP-105: Expense Sorting:** The API **MUST** provide endpoints to sort expense records by:
    - Amount (ascending/descending)
    - Date (ascending/descending)
    - *(Other suitable fields from the expense model as needed)*

- **FR-EXP-106: Pagination:** All list retrieval endpoints (e.g., viewing expenses) **MUST** support pagination to handle large datasets efficiently.

5.4. Dashboard & Summary Endpoints (FR-DASH)

- **FR-DASH-100:** The API **MUST** provide an endpoint to retrieve the total expenses for the current month for the authenticated user.
- **FR-DASH-101:** The API **MUST** provide an endpoint to retrieve a breakdown of expenses by category for a specified period (e.g., current month). This data **MUST** be suitable for creating charts (e.g., pie chart, bar chart) on the frontend.
- **FR-DASH-102:** The API **MUST** provide an endpoint to highlight the highest single expense within a specified period.
- *(Optional: Additional dashboard metrics such as average daily spending, spending trends over time, or comparison with historical data can be considered.)*

5.5. AI Microservice Integration (FR-AI)

- **FR-AI-100:** The ASP.NET Core API **MUST** include a loosely coupled integration layer for interacting with an external AI Flask microservice. This integration should be robust and fault-tolerant.
- **FR-AI-101:** The API **MUST** expose an endpoint that, upon request, sends relevant expense data (e.g., expenses for a specific period, categorized expenses) to the AI Flask microservice to generate summary insights.
- **FR-AI-102:** The AI integration **MUST** provide textual summary insights based on specified time periods, categories, and other relevant expense attributes.
- **FR-AI-103: Monthly Comparison with AI:** The API **MUST** implement endpoints that facilitate monthly spending comparisons, showing the difference in total spending between the current month and the previous month, integrated with AI-driven commentary or insights on these differences.

6. Non-Functional Requirements

6.1. API Design & Protocol (NFR-API)

- **NFR-API-100:** The API **MUST** adhere to RESTful principles.
- **NFR-API-101:** All API communication **MUST** use HTTPS with TLS 1.2 or higher for secure data transmission.
- **NFR-API-102:** The API **MUST** implement CORS (Cross-Origin Resource Sharing) configuration to allow access from specified frontend client domains.

6.2. Security (NFR-SEC)

- **NFR-SEC-100:** All user input **MUST** undergo rigorous validation and sanitization to prevent common vulnerabilities (e.g., SQL injection, XSS).
- **NFR-SEC-101:** Sensitive data (e.g., passwords) **MUST** be stored using robust hashing algorithms.

6.3. Performance & Scalability (NFR-PERF)

- **NFR-PERF-100:** The API endpoints **MUST** be optimized for quick response times (e.g., < 500ms for typical requests).
- **NFR-PERF-101:** The system design **SHOULD** consider the potential for storing monthly aggregate summary tables in the database to improve performance for dashboard-related queries.
- **NFR-PERF-102:** The microservice architecture for AI integration **SHOULD** allow for independent scaling of the AI component.

6.4. Reliability & Maintainability (NFR-REL)

- **NFR-REL-100:** The API **MUST** implement comprehensive global exception handling middleware to gracefully manage and log errors without exposing sensitive information.
- **NFR-REL-101:** All significant data changes (create, update, delete) **MUST** be recorded through an audit logging mechanism.
- **NFR-REL-102:** The application **MUST** integrate Serilog for structured and configurable logging across all layers.
- **NFR-REL-103:** The codebase **MUST** adhere to standard ASP.NET Core design patterns, including:
    - Clean separation of concerns (Models, DbContext, DTOs, Controllers).
    - Use of Fluent API for database context configuration.
    - Implementation of interfaces for repositories and services to promote testability and modularity.
    - Use of AutoMapper for object-to-object mapping (e.g., DTO to Entity).
- **NFR-REL-104:** The primary keys for all entities **MUST** be GUID or long data types.
- **NFR-REL-105:** All relevant database tables **MUST** include standard audit columns: CreatedAt, CreatedBy, UpdatedAt, UpdatedBy.

7. Architectural Considerations

The backend will be developed as a modular ASP.NET Core Web API application. The AI component will be a separate Python Flask microservice, communicating with the main API via standard HTTP requests, ensuring loose coupling and independent deployment capabilities. This approach supports future scalability and allows for specialized development within each service's optimal technology stack.