AI/ML Fitness Application with Nudge Engine

1. Project Overview

1.1 Introduction

The mobile fitness application is designed to help users achieve their health goals by tracking activities, diet, and sleep patterns. The core feature of the application is the Nudge Engine, which analyzes user data and sends personalized nudges to encourage healthy behavior. These nudges may include reminders to take a walk, suggesting a bedtime routine, or other prompts aimed at improving the user's health and wellness.

1.2 Key Features

- Nudge Management: Create, update, and delete different types of nudges. Schedule nudges based on user behavior and predefined triggers.
- User Profile and History: Store user data such as activity logs, preferences, and historical interactions with the app.
- Personalization: Use AI/ML algorithms to predict which nudge will be most effective for each user based on their behavior patterns.
- Nudge Dashboard: A user interface where users can view their nudge history and adjust preferences. Admins can manage nudges and monitor AI/ML performance.
- Nudge Notifications: Design UI for displaying nudges via mobile push notifications or in-app messages.
- Backend Services: Develop RESTful APIs for managing nudges, user profiles, and data retrieval.
- Scalability & Extensibility: Ensure the system can scale to handle more users, nudge types, and additional features.

---

2. Setup Instructions

2.1 Prerequisites

- Python
- MySQL Database
- Django Framework
- Tkinter GUI
- DB-Api

2.3.1 Project Structure - 1

```
nudge_engine/
├── app.py
├── models.py
├── nudge_engine.py
├── ai_ml.py
├── templates/
│   ├── login.html
│   ├── dashboard.html
│   └── admin.html
├── static/
│   ├── css/
```

app.py

- **Purpose:** This is the main entry point for the Flask backend server. It handles routing, server configuration, and the initialization of core services, such as the database connection and the Nudge Engine.
- **Functions:**
  - Routes for login, dashboard, and admin interface.
  - API endpoints for nudge management and user data retrieval.
  - Initialization of the Flask application with configurations from config.py.

models.py

- **Purpose:** Defines the database models using SQLAlchemy ORM. These models represent the tables in the database, including user profiles, nudge data, and activity logs.
- **Classes:**
  - User: Stores user information like user ID, username, and activity logs.
  - Nudge: Contains data about nudges, including type, trigger conditions, and history.

nudge_engine.py

- **Purpose:** Contains the core logic for managing and triggering nudges. It includes functions for creating, updating, deleting, and scheduling nudges based on user data.
- **Functions:**

  - create_nudge(): Adds a new nudge to the system.
  - schedule_nudges(): Determines when and how to send nudges based on user activity.
  - trigger_nudge(): Sends nudges to users when conditions are met.

ai_ml.py

- **Purpose:** Implements AI/ML models to personalize nudges. It contains algorithms that analyze user data and predict the most effective nudges.
- **Functions:**
  - train_model(): Trains the AI/ML models on historical user data.
  - predict_nudge(): Predicts the best nudge for a user based on current data.
  - retrain_model(): Retrains models periodically to improve accuracy.

templates/

- **login.html**: The template for the user login page, featuring input fields for username and password, along with error messages for failed login attempts.
- **dashboard.html**: The main user interface where users can view their activity logs, nudge history, and set preferences.
- **admin.html**: The admin interface, allowing administrators to manage nudges, view user engagement statistics, and monitor AI/ML model performance.

static/

- **css/**: Contains stylesheets for the application, defining the look and feel of the web pages.

2.3.2 Project Structure - 2

```
nudge_engine/
|-- Build
    |-- AI
|-- Dist
    |-- AI    #Exe executable file
|-- AI.py
|-- frontend.py
|-- Nudges.py
|-- User_login_screen.py
```

1. nudge_engine/

This is the root directory of the project. It contains all files and subdirectories related to the Nudge Engine application.

2. Build/

The Build directory is typically used for compilation and build-related processes. It often contains generated files and outputs from the build process.

- **Build/AI/**
  - This subdirectory under Build likely contains build artifacts or files related to AI components. It might include compiled or processed AI models or scripts necessary for the deployment of AI functionality.

3. Dist/

The Dist directory generally holds distribution files, such as executable files or packages, that are ready for deployment.

- **Dist/AI/**
  - This subdirectory contains executable files related to AI functionality. The AI component is packaged into a distributable executable format, which can be deployed or executed independently of the source code.

4. AI.py

This file contains the code for the AI/ML models used by the Nudge Engine. It might include:

- Definition and training of AI models
- Algorithms for nudge personalization and predictions
- Integration of pre-trained models for real-time use

5. frontend.py

This file likely contains the code for the frontend application or user interface. It might handle:

- Displaying data to the user
- User interactions and input
- Communication with backend services or APIs

6. Nudges.py

This file manages the core functionality of the nudge engine. It might include:

- Definition and scheduling of nudges
- Logic for sending nudges based on user activity and triggers
- Integration with the database to store and retrieve nudge data

7. User_login_screen.py

This file handles the user authentication and login screen. It includes:

- Code for the login UI

- Authentication logic to verify user credentials
- Navigation to other parts of the application based on successful login

API Documentation

The Nudge Engine provides several RESTful APIs to interact with the system, including managing nudges, user profiles, and retrieving data. Below is the comprehensive API documentation for the Nudge Engine.

1. API Endpoints

1.1 User Authentication

- Endpoint: /api/login
- Method: POST
- Description: Authenticates a user and returns a session token.
- Request Body:

```
{
"username": "string",
 "password": "string"
}
```

Response:

```
{
  "token": "string",
  "message": "string"
}
```

- Status Codes:
  - 200 OK - Successful login
  - 401 Unauthorized - Invalid credentials

1.2 User Profile

- Endpoint: /api/user/{user_id}
- Method: GET
- Description: Retrieves user profile information.

- Parameters:
    - user_id (Path parameter): The ID of the user.
- Response:

```
{
 "user_id": "integer",
 "username": "string",
 "preferences": {
   "step_goal": "integer",
   "sleep_target": "integer"
 },
 "activity_logs": [
   {
     "date": "string",
     "steps": "integer",
     "sleep_hours": "float"
   }
 ]
}
```

- Status Codes:
    - 200 OK - Profile retrieved successfully
    - 404 Not Found - User not found

1.3 Create Nudge

- Endpoint: /api/nudge
- Method: POST
- Description: Creates a new nudge.
- Request Body :

```
{
 "user_id": "integer",
        "nudge_type": "string",

        "trigger_condition": "string",

        "message": "string"

       }
```

Response :

```
{

  "nudge_id": "integer",

  "message": "string"

}
```

- Status Codes:
  - 201 Created - Nudge created successfully
  - 400 Bad Request - Invalid input

1.4 Update Nudge

- Endpoint: /api/nudge/{nudge_id}
- Method: PUT
- Description: Updates an existing nudge.
- Parameters:
  - nudge_id (Path parameter): The ID of the nudge.
- Request Body:

```
{

  "nudge_type": "string",

  "trigger_condition": "string",

  "message": "string"

}
```

Response:

```
{

  "message": "string"

}
```

- Status Codes:
  - 200 OK - Nudge updated successfully
  - 404 Not Found - Nudge not found
  - 400 Bad Request - Invalid input

1.5 Delete Nudge

- Endpoint: /api/nudge/{nudge_id}
- Method: DELETE
- Description: Deletes a nudge.
- Parameters:
  - nudge_id (Path parameter): The ID of the nudge.
- Response:

```
{
  "message": "string"
}
```

- Status Codes:
  - 200 OK - Nudge deleted successfully
  - 404 Not Found - Nudge not found

1.6 Nudge History

- Endpoint: /api/user/{user_id}/nudges
- Method: GET
- Description: Retrieves the history of nudges sent to a user.
- Parameters:
  - user_id (Path parameter): The ID of the user.
- Response:

```
{
  "user_id": "integer",
  "nudges": [
    {
      "nudge_id": "integer",
      "nudge_type": "string",
      "trigger_condition": "string",
      "message": "string",

      "timestamp": "string"

    }

  ]

}
```

- Status Codes:
  - 200 OK - Nudge history retrieved successfully
  - 404 Not Found - User not found

1.7 Admin: Nudge Management

- Endpoint: /api/admin/nudges
- Method: GET
- Description: Retrieves a list of all nudges for administrative purposes.
- Response:

```
{

 "nudges": [

  {

    "nudge_id": "integer",

    "user_id": "integer",

    "nudge_type": "string",

    "trigger_condition": "string",

    "message": "string"

  }

 ]

}
```

- Status Codes:
  - 200 OK - Nudges retrieved successfully

1.8 Admin: User Engagement Statistics

- Endpoint: /api/admin/engagement
- Method: GET
- Description: Retrieves statistics on user engagement with nudges.
- Response:

```
{
  "engagement_stats": [
    {
      "user_id": "integer",
      "nudge_id": "integer",
      "engagement_rate": "float"
    }
  ]
}
```

Assumptions Made

When designing and implementing the project structure for the Nudge Engine application, several assumptions have been made. These assumptions guide the development process and ensure that the project components work together seamlessly.

1. **nudge_engine/ Directory**
    - **Assumption:** This is the root directory where all project files are centralized for easy management and access.
2. **Build/ Directory**
    - **Assumption:** The Build directory is used for storing intermediate files or artifacts created during the build process. It is assumed that files in this directory are generated automatically and not manually edited by developers.
    - **Build/AI/ Subdirectory:**
        - **Assumption:** Contains build-related artifacts or output files specifically for AI components. This may include compiled versions of AI models or scripts. It is assumed that these files are required for deploying the AI functionality but are not part of the source code.
3. **Dist/ Directory**
    - **Assumption:** The Dist directory is intended for distribution files, which are ready for deployment. It contains executable files or packaged components.
    - **Dist/AI/ Subdirectory:**

        - **Assumption:** This subdirectory contains the executable file for the AI component. It is assumed that the AI functionality needs to be distributed in a compiled or executable format to be run independently.

4. **AI.py File**
    - **Assumption:** This file contains the core AI/ML code, including model definitions, training algorithms, and predictions. It is assumed that this script handles the entire AI workflow required by the Nudge Engine.
5. **frontend.py File**
    - **Assumption:** This file is responsible for the frontend interface of the application. It is assumed to handle user interactions, display data, and integrate with backend services. This script is expected to manage the UI elements and user experience aspects.
6. **Nudges.py File**
    - **Assumption:** This file manages the nudge engine's core functionality. It is assumed to handle the creation, scheduling, and delivery of nudges based on user data and triggers. It interacts with the database and other components to ensure that nudges are appropriately managed.
7. **User_login_screen.py File**
    - **Assumption:** This file handles user authentication and the login interface. It is assumed to include the logic for verifying user credentials and navigating users to different parts of the application upon successful login.

General Assumptions:

- **Modular Architecture:**
    - The project is designed with a modular approach, where different components are isolated into separate files and directories. This modularity assumes that each component can be developed, tested, and maintained independently.
- **Build and Distribution Processes:**
    - The Build and Dist directories assume a standard build and deployment process, where build artifacts and executables are generated automatically. It is assumed that the build process is managed by build tools or scripts.
- **Executable Formats:**
    - The assumption is that certain components, especially AI-related files, need to be packaged into executable formats for deployment. This implies that the AI component requires a compiled form for execution.
- **File Dependencies:**
    - It is assumed that each file and directory is dependent on others as outlined in the structure. For example, frontend.py relies on the backend functionality provided by Nudges.py, and the executable files in Dist/AI/ are generated from AI.py.
- **Configuration and Setup:**

- The project assumes that necessary configurations, such as database credentials and environment settings, are managed through configuration files or environment variables. It is assumed that these configurations are correctly set up for the application to function properly.