**User Management App(FastAPI) - Documentation**

As we discussed, This document explains the structure, code, and working of the User Management App. The project is built using FastAPI, a modern, fast web framework for building APIs with Python. It uses modular architecture for scalability and maintainability.

# Directory Structure

The directory structure is organized as follows:

1. `main.py`: Entry point of the application.
2. `app/controllers/`: Handles API requests.
3. `app/services/`: Contains business logic.
4. `app/repositories/`: Data storage and retrieval logic.
5. `app/middleware/`: Middleware for authentication, logging, and error handling.
6. `app/exceptions/`: Custom application-specific exceptions.
7. `app/models/`: Data models for request validation.
8. `app/utils/`: Utility functions.
9. `app/secrets/`: Directory to store sensitive files like `.pem` keys.
10. `logs/`: Stores application logs.

# About FastAPI

FastAPI is a web framework for building APIs with Python. It is:

* Fast: Built on Starlette and Pydantic, ensuring high performance.
* Easy to use: Automatic request validation and type checks.
* Scalable: Supports dependency injection and middleware.
* Automatic Documentation: Generates Swagger and ReDoc documentation out of the box.

# Detailed Code Explanation

1. main.py

This is the entry point of the application. It initializes the FastAPI app, adds middleware, and includes the user-related routes:

* `app.add\_middleware`: Adds logging middleware.
* `app.add\_exception\_handler`: Adds centralized error handling.
* `app.include\_router`: Modularizes and adds user-related routes defined in `user\_controller.py`.

2. user\_controller.py

Defines API endpoints for user management. It uses the APIRouter for modular routing:

* `@router.post('/')`: Endpoint to create a new user.
* `@router.get('/{user\_id}')`: Endpoint to fetch user details by ID.

Both routes call functions in `user\_service.py` to handle business logic.

3. user\_service.py

Contains business logic for user-related operations. It interacts with the repository layer:

* `create\_user(user)`: Calls `save\_user` in `user\_repository.py`.
* `get\_user(user\_id)`: Calls `fetch\_user\_by\_id` in `user\_repository.py`.

4. user\_repository.py

Acts as the data layer, storing and retrieving user information from an in-memory database:

* `users\_db`: A dictionary to store user data.
* `save\_user(user)`: Saves a user to the database.
* `fetch\_user\_by\_id(user\_id)`: Retrieves user details by ID.

5. Middleware

Middleware provides pre- and post-processing of API requests:

* `log\_middleware.py`: Logs all incoming requests and responses to `logs/app.log`.
* `error\_handler.py`: Catches and formats exceptions into JSON responses.

6. exceptions/custom\_exceptions.py

Defines custom exceptions for specific use cases:

* `UserAlreadyExistsError`: Raised when attempting to create a user that already exists.
* `UserNotFoundError`: Raised when a user is not found.

7. models/user\_model.py

Defines the structure of user data using Pydantic for validation:

- `User`: A Pydantic model with fields `id`, `name`, and `email`.

Automatically validates the request payload.

8. utils/helpers.py

Contains reusable utility functions:

- `generate\_user\_id()`: Generates a unique ID for new users.

9. secrets/private.pem

A placeholder for sensitive files such as private keys or API keys. Ensure this directory is excluded from version control using `.gitignore`.

10. logs/app.log

Stores application logs, including details of incoming requests and responses. Useful for debugging.

# Running the Application

1. Install dependencies: `pip install fastapi uvicorn`
2. Run the application:

`uvicorn main:app --reload`

1. Access the API documentation:

* Swagger UI: http://127.0.0.1:8000/docs
* ReDoc: http://127.0.0.1:8000/redoc

The application is now ready to handle user management tasks!