Expense Tracker

The Expense Tracker is a comprehensive web application designed to help companies efficiently manage and track their expenses. The application supports hierarchical roles, including Admin, Manager (department-wise), and User (Team Leader), and HR, ensuring that each role has the appropriate level of access and control over the data.

Table of Contents

Project Structure

```
app/
│
├── models/
│   ├── __pycache__/
│   ├── __init__.py
│   ├── admin.py
│   ├── all.py
│   ├── HR_model.py
│   ├── login_verify.py
│   ├── manager.py
│   ├── update_current_user.py
│   ├── user_expense.py
│   └── user.py
```

```
│
├── routes/
│   ├── __pycache__/
│   ├── __init__.py
│   ├── admin.py
│   ├── all.py
│   ├── HR_route.py
│   ├── manager.py
│   ├── reports.py
│   ├── user.py
│   ├── websocket_endpoint.py
│
├── templates/
│   └── __init__.py
│
├── auth.py
├── cloudinary_config.py
├── config.py
├── database.py
├── email_config.py
├── main.py
├── nginx.conf
├── transaction.py
├── update_department.py
├── websocket_manager.py
│
├── env/
│
├── profiles/
│
├── receipts/
│
├── static/
│
├── uploads/
│
├── .env
├── .gitignore
└── requirements.txt
```

Directory Structure and Contents

This report details the structure and organization of a Python-based project, likely a web application, that utilizes various technologies and frameworks. The project appears to be modular, with distinct directories and files dedicated to handling different aspects of the application, such as models, routes, templates, configuration, and WebSocket management:

1. app/

The `app/` directory serves as the main application directory, housing all core components of the project, including models, routes, templates, and various configuration files.

2. models/

The `models/` directory contains Python files defining the data models used in the application. These models likely represent the structure of the data stored in the database.

- **Files:**
    1. `__init__.py`**: Initializes the models module.**
    2. `admin.py`**: Contains models related to admin functionality.**
    3. `all.py`**: Potentially includes combined or shared models.**
    4. `HR_model.py`**: Specific models related to Human Resources (HR) functionality.**
    5. `login_verify.py`**: Models related to login verification processes.**
    6. `manager.py`**: Models specifically for manager-related data.**
    7. `update_current_user.py`**: Handles updates to the current user's information.**
    8. `user_expense.py`**: Manages models related to user expenses.**
    9. `user.py`**: General user-related models.**

3. routes/

The `routes/` directory holds Python files that define the API endpoints or routes for the application. These routes are likely connected to various parts of the application, such as admin, user, HR, and WebSocket functionalities.

- **Files:**
    1. `__init__.py`**: Initializes the routes module.**
    2. `admin.py`**: Routes related to admin operations.**
    3. `all.py`**: Possibly includes routes that handle multiple or general functions.**

4. `HR_route.py`: **Routes specific to HR-related operations.**
5. `manager.py`: **Routes for manager-related functions.**
6. `reports.py`: **Manages routes related to generating or viewing reports.**
7. `user.py`: **General routes for user operations.**
8. `websocket_endpoint.py`: **Defines WebSocket endpoints for real-time communication.**

3. templates/

The `templates/` directory is likely used for storing HTML or other template files that are rendered in the application. The presence of `__init__.py` suggests it could also include logic related to templating.

- **Files:**
    1. `otp_email_template.html`: **template for send otp on email.**
    2. `reset_password.html`: **template for reset password.**

4. Core Files

All core files here inside the `app/` like auth.py file for authentication , configuration file, main.py file that is the main gate of app etc.

1. `auth.py`: **Manages authentication-related logic and processes.**
2. `cloudinary_config.py`: **Configuration for Cloudinary, likely used for media storage and management.**
3. `config.py`: **General configuration settings for the application.**
4. `database.py`: **Handles database connections and operations.**
5. `email_config.py`: **Configuration settings for email functionality.**
6. `main.py`: **The main entry point of the application, where the app is likely initialized and run also login and forget password related endpoints and some other endpoints also included in this file.**
7. `nginx.conf`: **Configuration file for NGINX, possibly for serving the application or handling reverse proxy but currently not used in this project because we are doing it locally.**
8. `transaction.py`: **Manages transaction-related logic and operations basically in this file i writes some code to fetch transactions by custom date range or current month vise so it is basically for getting transactions on the basis of date range or current months.**
9. `update_department.py`: **Handles operations related to updating department information and i write here function for update user with department when user is creating. And this function is using in the manager and admin route because they both can create users but manager can only create user but admin can create user as well as manager and any other role like HR.**

10. `websocket_manager.py`: **Manages WebSocket connections and possibly related logic.**

5. Additional Directories

- **env/: This directory may contain environment-specific files or scripts.**
- **profiles/: used for store profile image of users.**
- **receipts/: Likely holds receipt-related documents or files.**
- **static/: Contains static assets like CSS, JavaScript, or images and some HTML code to test the backend.**
- **uploads/: Directory for storing uploaded files.**

6. Other Files

- **.env: Environment variables file, likely storing sensitive information such as API keys, database credentials,cloudinary credentials, JWT SECRET KEY, algorithms and also Email configuration etc.**

- **.gitignore: Specifies files and directories to be ignored by Git version control.**

- **requirements.txt: Lists the Python packages and dependencies required for the project.**

Technologies Used

- **Frontend: React with Vite**
- **Backend: FastAPI**
- **Database: MongoDB**
- **Authentication: JWT (JSON Web Token)**
- **CSS: Tailwind CSS**
- **Build Tool: Vite**

Libraries and Tools Used

This project leverages a diverse set of libraries and tools to build a robust, scalable, and efficient web application. Below is an overview of the key libraries and tools used, categorized by their functionality.

1. Web Framework

- **FastAPI (fastapi==0.110.3): A modern, fast (high-performance), web framework for building APIs with Python 3.7+ based on standard Python type hints.**

2. Asynchronous Programming

- **Anyio (anyio==4.3.0): Provides a universal async API that is compatible with both `asyncio` and `trio`.**
- **Aiosmtplib (aiosmtplib==2.0.2): An asynchronous SMTP client library.**
- **Websockets (websockets==12.0): A library for building WebSocket servers and clients in Python with an asynchronous API.**

3. Authentication & Security

- **Python-Jose (python-jose==3.3.0): A JavaScript Object Signing and Encryption (JOSE) implementation in Python.**
- **Passlib (passlib==1.7.4): A comprehensive password hashing framework.**
- **Bcrypt (bcrypt==4.1.3): A library for hashing passwords using the bcrypt algorithm.**
- **Itsdangerous (itsdangerous==2.2.0): Provides various helpers to pass data securely to untrusted environments.**
- **Cryptography (cryptography==42.0.5): A library providing cryptographic recipes and primitives to Python developers.**
- **PyJWT (python-jose==3.3.0): A Python library that allows encoding and decoding of JWT tokens.**

4. Data Validation & Serialization

- **Pydantic (pydantic==2.7.4): Data validation and settings management using Python type annotations.**
- **Pydantic Extra Types (pydantic-extra-types==2.7.0): Additional types for Pydantic.**
- **Pydantic Settings (pydantic-settings==2.2.1): Pydantic-based settings management.**
- **Annotated Types (annotated-types==0.6.0): Adds annotation-based type validation.**

5. Database Management

- **Pymongo (pymongo==4.7.1): A Python driver for MongoDB.**
- **Motor (motor==3.4.0): The async Python driver for MongoDB, built on top of Pymongo.**

6. WebSocket Management

- **Websockets (websockets==12.0): A library for building WebSocket servers and clients in Python.**

7. Email Handling

- **FastAPI-Mail (fastapi-mail==1.4.1): A simple, customizable, and secure email handling package for FastAPI.**
- **Aiosmtplib (aiosmtplib==2.0.2): An asynchronous SMTP client library for sending emails**
- **Email Validator (email_validator==2.1.1): A robust library for validating email addresses.**

8. File Handling & Cloud Storage

- **Python-Multipart (python-multipart==0.0.9): A streaming multipart parser for file uploads**
- **Cloudinary (cloudinary==1.40.0): A tool for managing and delivering media assets (images, videos) in the cloud.**

9. Configuration & Environment Management

- **Python-Dotenv (python-dotenv==1.0.1): A tool for loading environment variables from a `.env` file.**
- **PyYAML (PyYAML==6.0.1): A YAML parser and emitter for Python.**

10. Command Line Interface (CLI) Tools

- **Click (click==8.1.7): A package for creating command-line interfaces in a composable way.**
- **Typer (typer==0.12.3): A library for creating CLI applications that draws inspiration from Click.**
- **FastAPI-CLI (fastapi-cli==0.0.4): A CLI tool for managing FastAPI projects.**

11. JSON Handling

- **Ujson (ujson==5.9.0): A ultra fast JSON encoder and decoder.**
- **Orjson (orjson==3.10.2): A fast, correct JSON library for Python.**

12. Templating

- **Jinja2 (Jinja2==3.1.3): A templating engine for Python, used for rendering HTML templates.**

13. Utilities

- **Certifi (certifi==2024.2.2): A Python package that provides Mozilla's CA Bundle.**
- **Cffi (cffi==1.16.0): A Foreign Function Interface for Python calling C code.**
- **DnsPython (dnspython==2.6.1): A DNS toolkit for Python.**
- **Ecdsa (ecdsa==0.19.0): An elliptic curve digital signature algorithm library.**
- **H11 (h11==0.14.0): A pure-Python HTTP/1.1 client and server implementation.**
- **Httpcore (httpcore==1.0.5): A low-level HTTP library that powers HTTPX.**
- **Httptools (httptools==0.6.1): A collection of low-level HTTP utilities.**
- **Starlette (starlette==0.37.2): A lightweight ASGI framework/toolkit, ideal for building high-performance async services.**
- **Typing Extensions (typing_extensions==4.11.0): Backports and extensions for the typing module.**

14. Development Tools

- **Uvicorn (uvicorn==0.29.0): An ASGI web server implementation for Python, often used to serve FastAPI applications.**
- **Watchfiles (watchfiles==0.21.0): A library for monitoring file changes in a directory.**

Authentication and Authorization

The project employs a robust authentication and authorization mechanism that ensures secure access and role-based control across various parts of the application. The following steps outline the authentication and authorization process, which includes two-factor authentication, role-based access control, and token management.

- **Login Process**
    1. **Email and Password Verification:**

       The initial step in the authentication process is to verify the user's credentials—email and password. The provided email and password are validated against the records in the database. This ensures that only registered users can proceed further.

2. **Role-Based Access Control (RBAC):**
    - **After successful email and password verification, the system checks the role of the user. There are four roles defined in the system:**
    - **User: Basic access, limited to user-specific functionalities.**
    - **Manager: Access to managerial dashboards and functions.**
    - **Admin: Full administrative control over the system.**
    - **HR: Access to HR-related functionalities.**
    - **Based on the user's role, access to specific dashboards and functionalities is either granted or denied. For instance, a user cannot access the manager dashboard, and vice versa.**

- **Two-Factor Authentication (2FA)**

OTP Generation and Sending:

Once the email and password are verified, and the user's role is identified, the system generates a One-Time Password (OTP). This OTP is sent to the user's registered email address as an additional layer of security.

OTP Verification:

- **The user is required to enter the OTP received via email. This step ensures that the person attempting to log in has access to the registered email account, adding an extra layer of security.**
- **Upon successful OTP verification, the user is granted access to their respective dashboard according to their role.**

- **Token Management**

1. Token Generation:

- **After successful OTP verification, an `access_token` is generated. This token is a critical part of maintaining session integrity and is used to authenticate subsequent requests made by the user during their session.**

2. Token Storage:

- **The generated `access_token` is stored in a secure cookie in the user's browser. Storing tokens in cookies helps manage user sessions without requiring the user to re-authenticate on every request.**

3. Token Expiration:

- **The `access_token` is configured with a one-day expiration time. This means that the user remains authenticated for a full day, after which they will need to log in again to continue using the application.**

4. Logout Process:

- **When the user logs out, the `access_token` is removed from the cookie, effectively ending the user's session. To log back in, the user must go through the entire authentication process again, including OTP verification.**

- **Security Considerations**

1. Secure Cookies:

- **The `access_token` stored in the cookie is secured to prevent unauthorized access. This ensures that even if a third party gains access to the user's browser, they cannot easily hijack the session.**

2. Token Validation:

- **On each request, the token is validated against the database to ensure it is still valid and has not expired or been tampered with.**

3. Role-Based Access:

- **The role of the user is continuously checked to prevent unauthorized access to restricted areas of the application.**

Routes and API Endpoints

The application is designed with a comprehensive routing system that ensures access and functionality are tailored according to different user roles. The routes are protected and role-based, providing appropriate access depending on the user's role within the system.

https://docs.google.com/spreadsheets/d/1zCBjkgeROYxmfIV4hGZEGNoGVH8Ny1JjTz8fwq6hk_E/edit?gid=0#gid=0

All API endpoints are mentioned in this excel link file.

Roles and Permissions

- **Admin: Full access to all resources, including user management and expense tracking across all departments.**
- **Manager: Access to manage expenses for their specific department and view reports.**
- **User (Team Leader): Can track and manage expenses at the team level within their department.**

Backend Configuration

The backend of the application is configured using various tools and libraries to ensure optimal performance, security, and maintainability. Below is an overview of the key configurations used in the backend of the project:

1. **FastAPI Configuration:**
    - **Main Configuration:**

      The core settings for the FastAPI application are managed in the `main.py` file. This includes initializing the FastAPI app, configuring middleware, and setting up routes.

    - **CORS Settings:**

      Cross-Origin Resource Sharing (CORS) is configured to allow or restrict access to the API from different origins. This is particularly important for securing API access in production environments.

2. **Database Configuration:**
    - **Database Connection:**

      The connection to the MongoDB database is managed in the `database.py` file. This includes setting up the connection string, initializing the database client using Motor, and ensuring a secure and stable connection to the database.

    - **Environment Variables:**

      Sensitive information such as database credentials, connection strings, and other environment-specific settings are stored securely in the `.env` file. These variables are loaded into the application using `python-dotenv`.

3. **Authentication and Authorization:**
    - **JWT Configuration:**

      JSON Web Tokens (JWT) are used for secure authentication. The tokens are generated and validated in the `auth.py` file. The configuration includes setting the token expiration time, signing algorithm, and secret keys, which are stored in environment variables.

    - **Role-Based Access Control (RBAC):**

      The roles (user, manager, admin, HR) are configured within the application to enforce access controls. These roles are checked in each route to ensure that users can only access resources appropriate to their role.

4. **Email Configuration:**
    - **SMTP Settings:**

      The `email_config.py` file contains the configuration for sending emails through an SMTP server. This includes setting up the SMTP host, port, and authentication credentials, which are also stored in the `.env` file for security.

    - **FastAPI-Mail:**

      The application uses the `FastAPI-Mail` library to handle email sending. This is configured to integrate seamlessly with the SMTP settings, enabling functionalities such as sending OTPs and password reset emails.

5. **WebSocket Configuration:**
    - **WebSocket Endpoints:**

      WebSocket functionality is configured in `websocket_manager.py` and `websocket_endpoint.py`. These files manage the real-time communication features of the application, such as live updates and notifications.

    - **ASGI Server (Uvicorn):**

      The application runs on an ASGI server, Uvicorn, which is configured to handle WebSocket connections efficiently alongside HTTP requests.

6. **Logging and Monitoring:**
    - **Logging Configuration:**

      The logging settings are configured to capture and store logs for both application events and errors. This helps in monitoring the application's health and debugging issues. The logging level and format are customizable via the configuration.

    - **Error Handling:**

      Custom error handlers are configured to manage and respond to different types of exceptions and errors, ensuring that users receive appropriate feedback and that critical issues are logged.

7. **Logging and Monitoring:**
    - **Security Middleware:**

      Middleware for security, such as HTTPS redirection, CORS, and protection against common vulnerabilities, is configured to enhance the application's security.

    - **Session Management:**

      Middleware is also configured to manage user sessions, handling tasks such as token validation on each request and managing token expiration.

8. **Environment Management:**
    - **Environment-Specific Configurations:**

      The application supports different configurations for development, testing, and production environments. These configurations are managed via the `.env` file and conditional logic within the application to ensure the correct settings are applied based on the environment.

Conclusion

The project is a robust, modular, and secure web application that leverages modern Python frameworks and libraries. It features a strong authentication and authorization system, comprehensive API endpoints, and well-managed configurations. The careful structuring and use of best practices in coding and security ensure that the application is scalable, maintainable, and currently in under modification and conectivity for production use.