Billsight Backend Architecture & Work Structure

The Billsight backend is a modern, robust API built with Python and the FastAPI framework. Its primary purpose is to handle user authentication, manage receipt data, process uploaded files using Optical Character Recognition (OCR), and provide aggregated data for analytics and dashboard displays. The architecture is designed to be modular, scalable, and maintainable.

**Core Technologies**

* **Framework**: FastAPI (for high-performance, asynchronous API development)
* **Database**: PostgreSQL (managed via SQLAlchemy)
* **Object-Relational Mapper (ORM)**: SQLAlchemy
* **Data Validation**: Pydantic (used natively by FastAPI)
* **Authentication**: JWT (JSON Web Tokens) with OAuth2
* **OCR Engine**: Tesseract (via pytesseract)
* **Server**: Uvicorn (as the ASGI server)

**Architecture**

The backend follows a layered, modular architecture that separates concerns, making the codebase clean and easy to manage. The flow of a typical request is as follows:

**API Request -> FastAPI Routers -> Dependencies -> CRUD Functions -> Database Models -> Database**

1. **API Routers (routers/)**: The entry point for all incoming HTTP requests. Each router is responsible for a specific feature (e.g., receipts.py, auth.py) and defines the API paths, methods, and response models.
2. **Dependencies (dependencies.py)**: Reusable functions that are injected into the API endpoints. The most critical dependency is get\_current\_user, which handles token validation and ensures that endpoints are secure.
3. **CRUD Layer (crud.py)**: The bridge between the API logic and the database. This file contains all the functions that perform Create, Read, Update, and Delete (CRUD) operations. It houses the core business logic and database queries.
4. **Database Models (models.py)**: Defines the structure of the database tables using SQLAlchemy's ORM. These Python classes map directly to the tables in your PostgreSQL database.
5. **Data Schemas (schemas.py)**: Defines the shape of the data for API requests and responses using Pydantic. This ensures that all data entering and leaving the API is validated and has the correct structure.
6. **OCR Processing (ocr.py)**: A specialized module that runs as a background task. It handles the heavy lifting of text extraction from uploaded receipt files without blocking the API.

**Folder and File Structure Breakdown**

The backend/app/ directory is organized to reflect the separation of concerns:

* **main.py**: The main application file. It initializes the FastAPI app, sets up CORS middleware, and includes all the feature-specific routers from the routers directory.
* **database.py**: Handles the database connection setup, creating the SQLAlchemy engine and the session maker. It also provides the get\_db dependency for managing database sessions.
* **security.py**: Contains all security-related helper functions, such as password hashing/verification and JWT token creation.
* **dependencies.py**: Defines dependencies used across multiple endpoints, primarily the get\_current\_user function for authenticating API requests.
* **models.py**: Defines the SQLAlchemy ORM models (User, Receipt, Export), which represent the database tables.
* **schemas.py**: Contains the Pydantic models that define the data shapes for API requests and responses (e.g., UserCreate, ReceiptUpdate). This is crucial for data validation.
* **crud.py**: The "database logic" layer. All functions that directly interact with the database (querying, creating, updating, deleting records) are located here.
* **ocr.py**: Contains the logic for processing uploaded receipt files. It extracts text using Tesseract and parses it to find key details like vendor, date, and amount.
* **routers/**: A directory containing individual router files for each major feature:
  + **auth.py**: Handles user registration (/register) and login (/token).
  + **receipts.py**: Manages all receipt-related actions (/upload, /, /{receipt\_id}).
  + **user.py**: Handles user-specific data like goals and the main dashboard summary.
  + **analytics.py**: Provides endpoints for the analytics charts and KPIs.
  + **export.py**: Manages the creation and history of data exports.

**Data Flow Example: Uploading a Receipt**

1. A user uploads a file to the frontend, which sends a POST request to **/api/receipts/upload**.
2. The request is received by the upload\_receipt function in **routers/receipts.py**.
3. FastAPI uses the **dependencies.py** module to run get\_current\_user, which validates the user's JWT token.
4. The upload\_receipt function performs initial validation (file type, size) and saves the file to the server.
5. It then calls the create\_user\_receipt function in **crud.py**.
6. The crud.py function uses the Receipt model from **models.py** to create a new record in the database with a "processing" status.
7. The upload\_receipt function schedules a background task to run the process\_receipt\_file function from **ocr.py**.
8. The API immediately returns a 201 Created response to the user.
9. In the background, the **ocr.py** task extracts and parses the text, then calls the update\_receipt\_details function in **crud.py** to update the receipt's database record with the extracted data and a "processed" status.