IT314 – Software Engineering

Group 22 – Fingenie

AI – Powered Balance Sheet Analysis Tool

System Design

Team Members:

Name	Student ID
TANDEL DHRUVINEE DINESHKUMAR	202301203
MEET RUPESH GANDHI	202301219
PRIYANKA GARG	202301262
JAYADITYA SHAH	202301254
CHHABHAYA MANAN KETANBHAI	202301222
GAADHE JAYANSH MANUBHAI	202301232
VORA KRESHA MANOJBHAI	202301231
RATHOD AJAYKUMAR VALLABHBHAI	202301221
CHAUDHARI RUTU RAHUL	202301235
NAKUM AYUSH VIJAYBHAI	202301233

System Design Approach:

- We used a top-down method to design this system.
- We began by determining the main goal of the system, which was to create a financial analysis platform. We then listed the main inputs and outputs and the different features that our platform needed.
- We then split the system into four main parts: Presentation (Frontend), Interface (API), Business Logic, and Data.
- Lastly, we split the Logic layer into smaller, more manageable Django apps for each feature we chose.
- This method made sure that each small part we built had a clear role and place in the bigger system.

Design Goals:

- Understainability and readability: The system is designed to make subsystem responsibilities obvious.
- Maintainability and reuse of components: we prioritized modular architecture by keeping features segregated into distinct Django apps, making bugs easier to fix.
- Reliability and robustness: to ensure minimum number of errors, core logic is strictly separated from infrastructure code.

Interface Design (Black Box View)

1. Objective

The objective of the interface design is to define the system's boundaries by specifying valid inputs and expected outputs without exposing internal logic

2. External Interfaces

These define how the "Fingenie" system connects with third-party entities outside the application boundary.

- **User Interface (Client):** A web-based React application that sends HTTP requests and renders JSON responses.
- **LLM Service Provider (e.g., Google Gemini API):** Used for generating embeddings and natural language responses.
- **News Data Provider (News API):** Used to fetch real-time financial news feeds.

3. Functional Inputs & Outputs

Below is the black-box definition of data exchange for key system modules.

A. Authentication Module

- **Inputs:** User credentials (username/password) via POST /api/auth/login/.
- **Outputs:** HttpOnly Session Cookie (success) or 401 Unauthorized error.

B. Financial Report Processing

- **Inputs:** PDF or Excel files uploaded via POST /api/reports/.
- **Outputs**:
  Immediate: Upload success status.
  Processed: JSON object containing structured financial metrics and balance sheet data.

C. AI Chat & Insights

- **Inputs:** Natural language text queries (eg: "Explain the debt ratio") via POST /api/chat/.
- **Outputs:** AI-generated plain text responses or JSON-formatted "Word of the Day" content.

D. Semantic Search

- **Inputs:** Search query strings via GET /api/companies/search/?q=....
- **Outputs:** A ranked list of company profiles or documents matching the semantic context of the query.

E. External Service Integration (System-to-System)

- **Inputs (to External API):** Raw text prompts and API keys sent to the LLM provider.
- **Outputs (from External API):** Vector embeddings and text completions returned to the backend.

Typical endpoints(conceptual):

- POST /api/auth/login/ - login (session cookie)
- POST /api/auth/logout- logout
- GET /api/reports/ - list uploaded reports
- POST /api/reports/ - upload a financial report (PDF)
- GET /api/companies/search/?q=... - company semantic search
- POST /api/chat/ - chat message, returns AI response
- GET /api/insights/ - fetch AI-generated insights or summaries
- GET /api/blog/ - blog posts
- GET /api/news/ - news articles
- Auth & Security: Cookie-based sessions (Django sessions) with CSRF protection configured; CORS allowed origins are configured in settings for development.

Subsystem Decomposition
The system is organized into four logical layers, ensuring that the user interface is completely separated from the data processing logic.

1. Presentation Layer (The Client) : This is the "face" of the application that the user interacts with. Built using **React**, it handles all visual elements, such as the interactive charts for stock trends, the chat interface for the AI assistant, and the drag and drop zone for uploading financial reports.

2. Interface Layer (The Bridge) : Sitting between the user and the logic is the API layer. We use the **Django Rest Framework (DRF)** here. Its job is to act as a secure gateway, it receives requests from the React frontend (like a login attempt or a search query), validates them, and routes them to the correct backend function.

3. Business Logic Layer (The Core) : This is where the actual work happens. The backend is designed as a “modular monolith”, meaning distinct features are kept in separate Django apps:

- **Financial Module:** Handles the heavy lifting, processing uploaded PDFs, calculating financial ratios, and generating balance sheet comparisons (eg: apps.dataprocessor, apps.trends).
- **AI Module:** Manages the “brain” of the system. It handles the chatbot logic, processes vector embeddings, and generates insights (eg: apps.ai_insights, apps.chatbot).
- **Content Module:** Manages the educational aspects, such as the “Word of the Day”, blog posts, and news feeds (eg: apps.learning, apps.news).
- **Platform Services:** Handles standard utility functions like user login, authentication, and profile management (eg: apps.accounts).

4. Data & Infrastructure Layer : The foundation that supports the system. The system uses Django’s ORM(Object Relational Mapping) with SQLite at the database.
Data stored in SQLite:

- User account and authentication data
- AI generated summary
- Blog page articles

We also maintained a local vector store for the AI embeddings. This layer also manages the “handshake” with external services like the Google Gemini API and third-party News APIs.

Relationships Between Subsystems

- Presentation Layer → API: The React frontend communicates with the API router (REST) for all user actions.
- API → Logic Layer Modules : API layer routes client requests to the appropriate Django app (data input → dataprocessor, chat → chatbot/ai_insights, search → company_search, content → blog/news).
- Logic Layer Modules → Data layer:

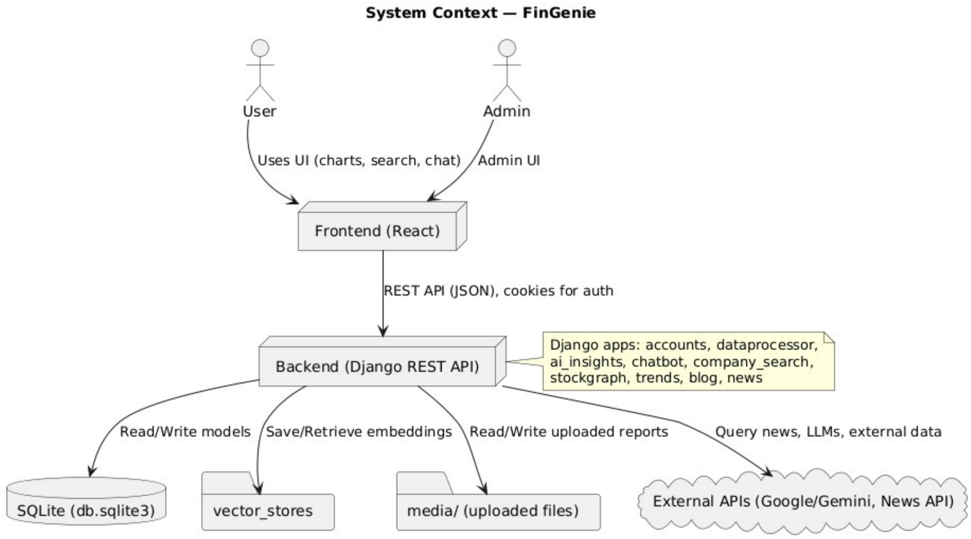
→dataprocessor, accounts, blog, news write/read from db.sqlite3.

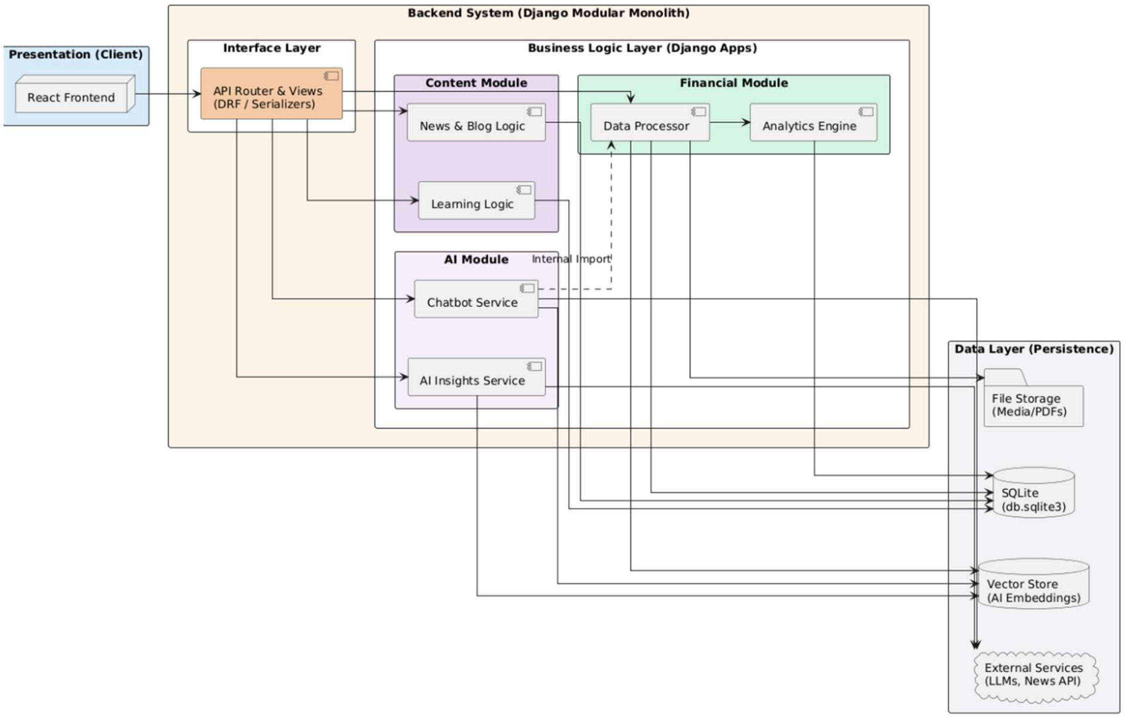
→ai_insights reads/writes embeddings to vector_stores and calls external LLMs.

→company_search performs semantic search using vector_stores.

Architectural Design

- Primary style: Client-Server + Layered architecture implemented as a Modular Monolith.
- Client: React frontend.
- Server: Single Django project with multiple apps (modular, but single deployable).
- Layers: Presentation (React), Interface/API (DRF views/serializers), Domain/Business (Django apps), Data/Infra (DB, vector store), AI/Integration (external LLMs).


System Context — FinGenie
User
Admin
Uses UI (charts, search, chat)
Admin UI
Frontend (React)
REST API (JSON), cookies for auth
Backend (Django REST API)
Django apps: accounts, dataprocessor, ai_insights, chatbot, company_search, stockgraph, trends, blog, news
Read/Write models
Save/Retrieve embeddings
Read/Write uploaded reports
Query news, LLMs, external data
SQLite (db.sqlite3)
vector_stores
media/ (uploaded files)
External APIs (Google/Gemini, News API)



Subsystem Decomposition — FinGenie