**🧠 模組說明（流程關鍵點補充）**

| **模組** | **負責內容** |  | **技術堆疊** |
| --- | --- | --- | --- |
| 1.Frontend | UI 操作與資料展示（筆記 CRUD、搜尋、圖譜） |  | Next.js + Tailwind |
| 2.API Server | 驗證、資料處理、API 對接 AI 模組 |  | tRPC / API Routes |
| 3.Embedding Handler | 將筆記轉換成向量存入 pgvector |  | OpenAI / LLM Embedding |
| 4.LangChain | AI 任務處理流程統籌（摘要、問題生成、語義搜尋） |  | LangChain Chains |
| 5.Database | PostgreSQL 儲存結構資料、pgvector 儲存向量 |  | Supabase / Railway |

**📈 Module Flow Diagrams**

**1. Notes CRUD Flow**

sequenceDiagram

participant U as User

participant F as Frontend (Next.js)

participant B as Backend API (tRPC)

participant DB as PostgreSQL

U->>F: Enter note (title, content, tags)

F->>B: call createNote mutation

B->>DB: INSERT note record

DB-->>B: return noteId

B-->>F: return new note data

F-->>U: render note in UI

**Explanation:**

1. **User Input:** The user fills in the note title, content, and optional tags in the NoteEditor component.
2. **Frontend Mutation:** The frontend uses tRPC’s createNote mutation to send a POST request containing the note payload.
3. **Database Write:** The backend handler validates input, then uses Prisma to insert a new record into the Note table.
4. **ID Return:** PostgreSQL returns the newly generated noteId, which Prisma includes in its response.
5. **UI Update:** The backend returns the complete note object; the frontend updates local state to display the newly created note immediately.

**2. AI Summarizer Flow**

sequenceDiagram

participant U as User

participant F as Frontend

participant B as Backend API

participant AI as OpenAI API

participant DB as PostgreSQL

U->>F: click Summarize on note

F->>B: call summarize mutation with content

B->>AI: send prompt for summary

AI-->>B: return summary text

B->>DB: update note.summary field

B-->>F: return summary

F-->>U: display summary

**Explanation:**

1. **User Action:** User clicks “Summarize” on an existing note, triggering the frontend to call the summarize mutation with the note’s text.
2. **Prompt Construction:** The backend constructs a prompt template that includes the note content for context.
3. **LLM Call:** Backend calls OpenAI’s Chat API (model gpt-4) with the prompt. Network latency and rate limits apply here.
4. **Persist Summary:** On receiving the summary, the backend updates the summary column in the Note record via Prisma.
5. **Response to Frontend:** The summarized text is returned in the mutation response; the UI replaces or appends the note’s summary section.

**3. Question Generator Flow**

sequenceDiagram

participant U as User

participant F as Frontend

participant B as Backend API

participant AI as OpenAI API

participant DB as PostgreSQL

U->>F: click Ask AI on note

F->>B: call generateQuestions mutation with content

B->>AI: send prompt for questions

AI-->>B: return questions

B->>DB: update note.questions

B-->>F: return questions list

F-->>U: render questions

**Explanation:**

1. **Trigger:** User invokes “Ask AI” which sends note content to the generateQuestions mutation.
2. **LLM Prompting:** Backend constructs a prompt instructing the model to generate multiple deep, reflective questions.
3. **Receiving Answers:** Once OpenAI responds, the backend parses the raw text into a structured array of questions.
4. **Storage:** Questions are saved in a JSON or text field in the Note table to allow future retrieval without re-calling the API.
5. **Display:** The UI renders the list of questions beneath the note for active recall.

**4. Semantic Search Flow**

sequenceDiagram

participant U as User

participant F as Frontend

participant B as Backend API

participant EMB as Embedding Service

participant DB as pgvector

U->>F: submit search query

F->>B: call semanticSearch mutation

B->>EMB: createEmbedding(query)

EMB-->>B: return queryVector

B->>DB: SQL <#> vector similarity search

DB-->>B: return top notes

B-->>F: return search results

F-->>U: display results

**Explanation:**

1. **Query Input:** User types a natural language query in the search bar.
2. **Embedding Creation:** Backend calls the embedding service (OpenAI or local LLM) to generate a high-dimensional vector representation of the query.
3. **Vector Search:** A raw SQL query using the <#> operator computes cosine distance between the stored note embeddings and the query vector, returning the top N closest matches.
4. **Result Packaging:** The backend returns note metadata (title, snippet) sorted by relevance score.
5. **UI Rendering:** Frontend displays search results with highlighted snippets or tag contexts.

**5. Chat with Notes Flow**

sequenceDiagram

participant U as User

participant F as Frontend

participant B as Backend API

participant DB as PostgreSQL & pgvector

participant AI as LangChain & OpenAI

U->>F: enter chat question

F->>B: call chatWithNotes mutation

B->>DB: fetch relevant note embeddings

B->>AI: send RAG prompt + context

AI-->>B: return chat response

B-->>F: return response

F-->>U: display chat answer

**Explanation:**

1. **User Query:** The user enters a question about their notes in the chat interface.
2. **Context Retrieval:** Backend runs a semantic search to fetch top K relevant notes and their content via embeddings.
3. **RAG Prompt Assembly:** Using LangChain, the backend constructs a Retrieval-Augmented Generation prompt that includes user question plus extracted note excerpts.
4. **LLM Generation:** The combined prompt is sent to OpenAI, which returns a synthesized answer grounded in the user’s own notes.
5. **Display Answer:** The answer is returned through the mutation and rendered in a chat-like UI, with links back to source notes.

**6. Knowledge Graph Flow**

flowchart TD

A[Frontend: fetchNotes] --> B[Backend: GET /notes]

B --> C[(PostgreSQL)]

C --> D[Prisma returns notes + embeddings]

D --> E[Frontend receives data]

E --> F[Graph Module: D3.js or React Flow]

F --> G[Rendered interactive graph]

**Explanation:**

1. **Data Fetch:** Frontend calls a secure API endpoint to retrieve all notes and their metadata (including embeddings/tags).
2. **Database Query:** Backend uses Prisma to fetch the note records including embedding vectors if needed for positioning.
3. **Data Transfer:** JSON payload of notes is sent to the frontend.
4. **Graph Rendering:** The frontend passes the data into a graph visualization component (using D3.js or react-flow) to render nodes and edges based on tag co-occurrence or vector similarity.
5. **User Interaction:** Users can click nodes to navigate to the full note or filter by tag clusters.