ChatGPT

Next.js Admin Dashboard for WhatsApp AI Chatbot

We build a self-hosted Next.js admin panel (frontend + backend API) that connects to a Postgres DB (with pgvector) to manage documents, chunks, and message logs. The UI is secured by email/password login (e.g. via NextAuth.js), and consists of pages for managing the knowledgebase, viewing chat history, and displaying analytics charts. It uses Tailwind (or a Tailwind-based UI kit like shadcn/ui) for styling and Charting libraries (Recharts/Chart.js) for graphs. The architecture is: **Next.js** (App Router or Pages) as frontend+API, **Postgres** with pgvector for data, and **n8n** handling the WhatsApp chatbot logic and memory. The DB has tables `documents(id, title, filename, content, created_at)`, `chunks(id, document_id, chunk_text, embedding_vector)`, and `messages(id, session_id, message JSON)` as given.

Tech Stack and Setup

We use **Next.js (v15+)** with Tailwind CSS or shadcn/ui components for the UI [1]. NextAuth.js handles authentication (Credentials provider for email+password) [2]. API routes (`/pages/api/*` or `/app/api/*`) implement backend endpoints. We use either **Prisma** or **node-postgres** for DB access. Because the `pgvector` extension is needed, we enable it in Postgres (e.g. `CREATE EXTENSION IF NOT EXISTS vector;`) and define vector columns via raw SQL migrations [3]. (Prisma does not yet natively support vector types, so we either use raw SQL or define it as an unsupported field [4].) Charts use **Recharts** (a React chart library) for line/bar graphs [5]. The app is deployed (e.g. on Render or Hostinger) with environment variables for DB connection, and NextAuth session secret.

Admin Authentication

A secure login page is required. We use NextAuth.js's **Credentials provider** for email/password sign-in. In NextAuth's `authorize()` callback we validate the credentials against a user table in DB [2]. For example:

```
providers: [
  CredentialsProvider({
    name: "Credentials",
    credentials: {
      email: { label: "Email", type: "text", placeholder: "admin@example.com" },
      password: { label: "Password", type: "password" }
    },
    async authorize(credentials) {
      // lookup user by email & verify password...
      if (user) return user;
      return null;
    }
```

```
  })
]
```

All admin pages are protected: on the client side, we use `useSession()` in components to check auth status and redirect or show "Access Denied" if unauthenticated [6]. We can also use NextAuth's middleware (`middleware.ts`) to guard routes. For example:

```
// middleware.ts
export { default } from "next-auth/middleware";
export const config = { matcher: ["/dashboard/:path*"] };
```

This ensures only logged-in users with a valid session can access the dashboard pages [7].

Knowledgebase Management

We maintain a document knowledgebase in the `documents` and `chunks` tables. The admin UI has a **Documents** page showing all rows from `documents`. Each document shows its title, filename, upload date, etc. For upload, the UI sends a file (PDF/DOCX/TXT) to `POST /api/documents`. In that API handler, we parse the file to extract text: for PDFs we can use a Node library like `pdf-parse` which returns the PDF's text content (e.g. `data.text` contains the full text) [8]. For Word documents, we can use `mammoth` or similar – e.g. `mammoth.extractRawText({path: filePath})` which yields the raw text of the DOCX [9]. The extracted full text and metadata (filename, title, etc.) are stored in the `documents` table.

Next, we split this text into smaller chunks (e.g. ~1000 characters each with overlap) before embedding. We can use a text-splitting library (such as llm-chunk or LangChain's splitters). For example, the `llm-chunk` library has a simple `chunk(text)` function that by default splits paragraphs into ~1000-char segments [10]:

```
import { chunk } from 'llm-chunk';
const chunks = chunk(fullText); // splits by paragraphs
```

Each chunk's text is then sent to the OpenAI Embedding API. Using the OpenAI Node SDK, we call `openai.createEmbedding({model: "text-embedding-ada-002", input: chunkText})` for each chunk [11]. We collect the resulting embedding vectors and insert each into the `chunks` table along with the `document_id`. For example:

```
INSERT INTO chunks (document_id, chunk_text, embedding_vector) VALUES ($1, $2,
$3);
```

(We create the `chunks.embedding_vector` column as type `vector(1536)` once pgvector is enabled [3].) All document chunks store their segment text and corresponding numeric vector.

On the **Documents** page, the admin can view each document's details (full content and metadata via `GET /api/documents/:id`) and delete it. Deletion is done via `DELETE /api/documents/:id`, which removes the document row and cascades to delete related chunks (either via a foreign-key ON DELETE cascade or explicitly deleting from `chunks` where `document_id = id`). This ensures the knowledgebase stays in sync.

Message History Viewer

The `messages` table logs all WhatsApp chat messages as they flow through n8n (via Postgres Chat Memory). Each row has an `id`, `session_id` (the WhatsApp number or user ID), and a `message` JSON blob with `type`, `content`, etc. We build a **Messages** page in the admin UI that lists sessions and their message histories. The UI calls something like `GET /api/messages?session_id=...` to fetch all messages for a given session in chronological order (ordered by `id`).

The chat history is rendered in a WhatsApp-style chat interface. We align user (human) messages on the left and AI responses on the right. Using Tailwind (or a component library like DaisyUI), we style chat bubbles: for example, DaisyUI provides classes `chat-start` (left-align) and `chat-end` (right-align) on a `div` to create chat bubbles [12]. Sample markup:

```
<div class="chat chat-start"><div class="chat-bubble">User: Hi!</div></div>
<div class="chat chat-end"><div class="chat-bubble">AI: Hello there!</div></div>
```

This produces left/right bubbles as in [29†L204-L211]. We iterate messages and wrap them with these classes depending on `message.type` or speaker. The admin can scroll through the conversation.

Sessions can be grouped: perhaps the UI shows a list of session IDs (clickable to open that session's chat). We ensure all API routes used here also check for a valid admin session.

Analytics Dashboard

An **Analytics** page computes live metrics via SQL queries. We run aggregate queries on the `messages` table to summarize usage. For example:
- **Total messages** in various time ranges: e.g., `SELECT COUNT(*) FROM messages WHERE created_at >= (current_date - interval '7 days')` for weekly count, similarly for today/month/year.
- **Messages per day/week/month**: use `date_trunc` to group by time. For example, to count messages per day:

```
SELECT date_trunc('day', created_at) AS day, COUNT(*)
FROM messages
GROUP BY date_trunc('day', created_at)
ORDER BY day;
```

(This pattern – using `date_trunc('day', ts)` in GROUP BY – is shown in SQL examples [13].) Similar queries with `date_trunc('week', ...)` or `'month'` produce weekly/monthly tallies. The top-5 active sessions is a query like:

```
SELECT session_id, COUNT(*) AS msg_count
FROM messages
GROUP BY session_id
ORDER BY msg_count DESC
LIMIT 5;
```

These results are rendered as charts. We use Recharts (a React chart library) to plot line or bar charts of these aggregates [5]. For example, a line chart of daily messages for the past month, or a bar chart of messages-per-week. Recharts provides easy components (e.g. `<LineChart>`, `<BarChart>`, etc.) to visualize our SQL results. The analytics page displays the charts and stats at a glance.

Backend API Routes

The Next.js app defines RESTful API routes under `/pages/api/*` (or `/app/api/*`). Key endpoints include:

- `GET /api/documents` – returns a list of documents (select * from `documents`).
- `GET /api/documents/:id` – returns full content/metadata of one document (select * where id).
- `POST /api/documents` – handles file upload (multipart/form-data). The handler reads the uploaded file (buffer or temp file), parses it (using pdf-parse/mammoth), then calls embedding generation and DB inserts.
- `DELETE /api/documents/:id` – deletes the document and its chunks (via cascade).
- `GET /api/messages?session_id=...` – queries `messages` table for that session sorted by id.
- `GET /api/analytics?range=daily|weekly|monthly|...` – runs the SQL aggregates described above and returns JSON results for the charts.

Each API route checks the user's session (using NextAuth's `getServerSession()` or middleware) to ensure only authenticated admins can call them. For example, a sample handler might look like:

```
export default async function handler(req, res) {
  // e.g. /api/documents (GET or POST)
  const session = await getServerSession(req, res, authOptions);
  if (!session) { return res.status(401).json({error: "Unauthorized"}); }
  if (req.method === 'GET') {
    const docs = await prisma.document.findMany();
    res.json(docs);
  } else if (req.method === 'POST') {
    // parse upload, extract text, split, embed, store
  }
}
```

Next.js API routes are simple functions exporting `handler(req, res)` and can be tested in dev easily [14].

Database Schema (Postgres + pgvector)

The assumed schema is as given. To recap:

- **documents**: `(id SERIAL PRIMARY KEY, title TEXT, filename TEXT, content TEXT, created_at TIMESTAMP DEFAULT now())`.
- **chunks**: `(id SERIAL PRIMARY KEY, document_id INT REFERENCES documents(id) ON DELETE CASCADE, chunk_text TEXT, embedding_vector VECTOR(1536))`. (We installed the `pgvector` extension so the `VECTOR` type is available [3].)
- **messages**: `(id SERIAL PRIMARY KEY, session_id TEXT, message JSONB, created_at TIMESTAMP DEFAULT now())`.

We can use Prisma to model documents and messages. For `chunks.embedding_vector`, Prisma doesn't support vector type, so we either mark it as an unsupported type or use raw SQL for that field [4]. Alternatively, use `node-postgres` to query/inser vectors directly. In any case, PGVector allows fast similarity search if needed (though for admin UI we may not need similarity queries, only storing them).

UI and Styling

We organize the Next.js app with clear pages and components. Common layout components (navbar, sidebar) are built with Tailwind/shadcn-ui components [1]. For example, a sidebar with links to **Knowledgebase**, **Messages**, **Analytics**, and **Settings**. The **Knowledgebase** page shows a table/list of documents (with columns Title, Date, actions). The **Messages** page might first show a list of session IDs; clicking one fetches the chat history which we display using styled chat bubbles (as noted above with `chat-start`/`chat-end`). The **Analytics** page shows cards or sections with total counts and Recharts graphs. Tailwind utility classes (or DaisyUI classes) give a clean modern look.

For the chat UI specifically, DaisyUI's chat component demonstrates how to align bubbles [12]. For other forms (file upload), we use nice input components (e.g. file picker styled by Tailwind). We ensure responsiveness and clarity, with simple routing via Next.js links (`<Link>`). The **Sign In** page uses NextAuth's signIn form.

Putting It All Together

In summary, the admin dashboard is a Next.js app using NextAuth for secure login [2] [6], and Tailwind/shadcn for styling [1]. The frontend calls our custom API routes to manage docs and view data. File parsing is done server-side in Node (e.g. `pdf-parse` or `mammoth` to extract text [8] [9]), and text is chunked (e.g. with llm-chunk [10]) and embedded via OpenAI API [11]. We store everything in Postgres with pgvector (extension enabled via `CREATE EXTENSION vector;` [3]). The message history is queried from the same DB and shown in a WhatsApp-like UI (using chat bubble CSS as per [29]). Analytics charts use SQL aggregates (e.g. `date_trunc` grouping [13]) and Recharts [5] for visualization. All pages and API routes require a valid session, as shown in NextAuth tutorials [6].

This design yields a complete Next.js admin dashboard: clean routes (e.g. `/documents`, `/messages`, `/analytics`), secure access, and a polished UI for the knowledge base and conversation logs, all wired up to the Postgres backend with vector embeddings as specified.

Sources: We draw on official docs and examples for NextAuth credentials auth [2] [6], file parsing libraries [8] [9], text splitting strategies [10], embedding with OpenAI/pgvector [11] [3], and UI patterns like chat bubbles [12] and dashboards [5] [1]. These informed our architecture and code approach.

---

[1] GitHub - Kiranism/next-shadcn-dashboard-starter: Admin Dashboard Starter with Nextjs 15 and Shadcn ui
https://github.com/Kiranism/next-shadcn-dashboard-starter

[2] Credentials | NextAuth.js
https://next-auth.js.org/providers/credentials

[3] [4] Postgres extensions | Prisma Documentation
https://www.prisma.io/docs/postgres/database/postgres-extensions

[5] How to use Next.js and Recharts to build an information dashboard
https://ably.com/blog/informational-dashboard-with-nextjs-and-recharts

[6] [7] Securing pages and API routes | NextAuth.js
https://next-auth.js.org/tutorials/securing-pages-and-api-routes

[8] Parsing PDFs in Node.js - LogRocket Blog
https://blog.logrocket.com/parsing-pdfs-node-js/

[9] mammoth - npm
https://www.npmjs.com/package/mammoth

[10] How to Chunk Text in JavaScript for Your RAG Application | DataStax
https://www.datastax.com/blog/how-to-chunk-text-in-javascript-for-rag-applications

[11] How to Use PGVector: Making Queries with OpenAI Embeddings in Node.js | by Vadim Pryakhin | Medium
https://medium.com/@lostargon/how-to-use-pgvector-making-queries-with-openai-embeddings-in-node-js-ba5c27332c03

[12] Tailwind Chat bubble Component — Tailwind CSS Components (version 5 update is here)
https://daisyui.com/components/chat/?lang=en

[13] postgresql - For each day, get query row count for that day - Database Administrators Stack Exchange
https://dba.stackexchange.com/questions/308322/for-each-day-get-query-row-count-for-that-day

[14] Routing: API Routes | Next.js
https://nextjs.org/docs/pages/building-your-application/routing/api-routes