GKChatty Application Architecture & Design Overview

This report provides a high-level overview of the GKChatty application's architecture and design, intended for a senior developer to quickly grasp its structure.

1. Overall Architecture & Technology Choices

Monorepo Structure

The project is organized as a monorepo, likely managed by PNPM Workspaces (`pnpm-workspace.yaml`). This facilitates code sharing and centralized dependency management.

Core Applications

- `apps/api` : The backend API server.
- `apps/web` : The frontend web application.

Shared Code

A `packages/types` directory exists for shared TypeScript type definitions between the frontend and backend, promoting consistency.

Language

Primarily TypeScript across both frontend and backend, ensuring type safety and better maintainability.

DevOps & Tooling

- **Containerization:** Docker is used (`Dockerfile.minimal` , `docker-compose.yml`), suggesting containerized deployments.
- **CI/CD:** GitHub Actions are utilized for continuous integration and deployment (inferred from `.github/workflows/`).

Deployment

- Frontend (`apps/web`) is likely deployed on Netlify (evidenced by `netlify.toml` and Netlify Next.js plugin).
- Backend (`apps/api`) has been discussed as being deployed on Render.com.

Code Quality

ESLint for linting and Prettier for code formatting are in place.

Testing

Jest is configured for testing, though the extent of test coverage isn't immediately clear from this overview.

2. Backend Architecture (`apps/api`)

Framework

Built with Node.js and the Express.js framework.

Entry Point

`apps/api/src/index.ts` initializes the server, sets up middleware, connects to the database, and mounts routes.

Middleware

- **Standard middleware:** CORS, JSON and URL-encoded body parsers, `cookie-parser`.
- **Logging:** `morgan` for HTTP request logging.
- **Custom:** `correlationIdMiddleware` for request tracing.

Routing

A modular routing system is in place (`apps/api/src/routes/`), with separate route files for different resources (e.g., `authRoutes.ts` , `chatRoutes.ts` , `documentRoutes.ts` , `systemKbRoutes.ts` , `adminRoutes.ts`).

Database

MongoDB is the primary database, interacted with via the Mongoose ODM (`apps/api/src/utils/mongoHelper.ts` , `apps/api/src/models/`). Models define schemas for users, chats, documents, settings, etc.

Authentication & Authorization

JWT-based authentication (`jsonwebtoken`) is implemented in `authRoutes.ts` and `middleware/authMiddleware.ts` (`protect` , `checkSession`). Passwords are hashed using `bcryptjs` .

Core Services & Business Logic

While there's a small `apps/api/src/services/` directory, much of the service-like logic (interactions with external systems, core operations) resides in utility helpers within `apps/api/src/utils/` .

AI Service Integration

- **OpenAI:** Extensive integration for chat completions and embeddings using the `openai` SDK (`apps/api/src/utils/openaiHelper.ts`).
- **Mistral AI:** Used as a fallback for chat completions via the `@mistralai/mistralai` SDK (`apps/api/src/utils/mistralHelper.ts`).
- **Pinecone:** Utilized as the vector database for semantic search, interacting via `@pinecone-database/pinecone` SDK (`apps/api/src/utils/pineconeService.ts`).
- *(Note: `chromadb` is a listed dependency but Pinecone appears to be the active vector DB solution).*

Document Processing

Handles file uploads (`multer`), PDF parsing (`pdf-parse`), and likely embedding generation before storage/indexing. Potentially uses AWS S3 for document storage (`@aws-sdk/client-s3`).

Resilience & Error Handling

- **Circuit Breakers:** Opossum circuit breakers are implemented to protect calls to external services (OpenAI, Pinecone, Mistral), preventing cascading failures.
- **Retries:** A custom `withRetry` utility (`apps/api/src/utils/retryHelper.ts`) using `async-retry` provides exponential backoff retry mechanisms for external calls.
- **Global Error Handler:** A global error handler is defined in `index.ts` for standardized error responses.

Logging

- Pino is used for structured, asynchronous logging (`apps/api/src/utils/logger.ts`).
- A `consolePatch.ts` utility redirects all global `console.*` calls to Pino, ensuring consistent structured logging with `correlationId` injection.

Configuration

Environment variables are managed using `dotenv` and are logged at startup for diagnostics.

3. Frontend Architecture (`apps/web`)

Framework

Built with React using the Next.js framework (App Router seems to be in use, e.g., `apps/web/src/app/layout.tsx`).

Structure

- Main layout defined in `apps/web/src/app/layout.tsx` .
- Pages/views are within `apps/web/src/app/` .
- Reusable UI components are in `apps/web/src/components/` .

Styling

- Tailwind CSS for utility-first styling.
- `clsx` and `tailwind-merge` for conditional and merged class names.
- `tailwindcss-animate` for animations.

UI Components & Icons

- Radix UI headless components are used as building blocks for accessible UI elements.
- Lucide React for icons.

State Management

- Primarily relies on React's built-in state management (`useState` , `useEffect`) and React Context API (e.g., `AuthContext`).
- NextAuth.js (`next-auth`) is used for client-side session management and authentication state.
- *No indication of a global state library like Redux or Zustand from the initial overview, suggesting a preference for localized state or context.*

API Communication

API calls to the backend are made using `fetch` , with the base URL configured via environment variables (`@/lib/config`).

Specialized Features

- PDF viewing is implemented using `pdfjs-dist` and `react-pdf` .
- Dark mode/theming is supported via `next-themes` .
- Page transitions and animations using `framer-motion` .

Error Handling

An `<ErrorBoundary>` component is present, suggesting component-level error catching.

Client-Side Routing

Handled by Next.js router (`next/navigation`).

4. Shared Packages (`packages/types`)

A dedicated package `packages/types` houses shared TypeScript interfaces and type definitions (e.g., for Auth, Chat), ensuring type consistency across the API and web applications.

5. Key Data Flows (Conceptual)

User Authentication

1. Frontend (`apps/web`) captures credentials, calls `/api/auth/login` .
2. Backend (`apps/api`) validates credentials against MongoDB, generates a JWT.
3. JWT is typically sent back in an HttpOnly cookie. Subsequent requests from web include this cookie.
4. `authMiddleware` on the API verifies the JWT.

Chat Interaction

1. User inputs query in `apps/web` .
2. Frontend sends query and history to `/api/chats` or `/api/chat/stream` .
3. Backend (`chatRoutes.ts`):
   - Authenticates user.
   - Generates embeddings for the query (via `openaiHelper.ts`).
   - Queries Pinecone vector DB for relevant context (via `pineconeService.ts`).
   - Constructs a prompt with context and history.
   - Calls OpenAI (or Mistral fallback) for chat completion (via `openaiHelper.ts` / `mistralHelper.ts`), protected by circuit breakers and retries.
   - Persists chat messages and sources to MongoDB.
   - Returns response (or streams response) to the frontend.
4. Frontend displays the assistant's message and any retrieved sources.

Document Upload & Processing

1. User uploads a document via `apps/web` .

2. Frontend sends file to `/api/documents/upload`.
3. Backend (`documentRoutes.ts`):
   - Receives file using `multer`.
   - Parses text content (e.g., `pdf-parse` for PDFs).
   - Chunks the document content.
   - Generates embeddings for each chunk using OpenAI.
   - Upserts these embeddings and metadata to Pinecone.
   - Saves document metadata to MongoDB.
   - (Possibly uploads original file to S3).

6. Noteworthy Design Patterns and Principles

- **Modularity:** The codebase is broken down into modules for routes, utils, models, and components.
- **Service Abstraction (Partial):** External service interactions (OpenAI, Pinecone) are encapsulated within specific helper/utility modules, providing a degree of abstraction.
- **Resilience:** Explicit use of circuit breakers and retry mechanisms for external service calls is a strong point.
- **Structured Logging:** Centralized and structured logging with Pino and correlation IDs is critical for observability and debugging in a distributed environment.
- **Configuration Management:** Use of `.env` files for environment-specific configuration.
- **Type Safety:** TypeScript is used throughout, reducing runtime errors and improving developer experience.
- **Separation of Concerns:** Clear distinction between API (backend logic) and Web (presentation logic).

7. Areas for Potential Deeper Review by Senior Dev

- **Error Handling Consistency:** While global and component-level error handlers exist, ensuring consistent error propagation, logging, and user feedback across all layers.
- **Test Coverage:** The `package.json` includes Jest, but the actual extent and depth of unit, integration, and e2e tests would be important to assess.
- **Scalability Considerations:** Review database indexing (`db-index-optimization.js` exists, which is a good sign), query performance, and potential bottlenecks in high-traffic areas (e.g., chat processing, embedding generation).
- **Security Hardening:** Beyond authentication, review input validation, protection against common web vulnerabilities (OWASP Top 10), and secure handling of API keys/secrets (though `dotenv` is used, ensuring best practices in CI/CD and deployment).
- **State Management Complexity (Frontend):** For a growing application, assess if the current React Context/custom hook approach for state management remains scalable or if a more robust global state solution might be needed for certain complex features.
- **Completeness of `packages/types` :** Ensure all critical shared data structures are well-defined and utilized.

This report should provide a solid foundation for a senior developer to understand the GKChatty application. Further deep dives into specific modules or flows can be conducted based on areas of particular interest or concern.