PDF Chat Source Code Documentation

Table of Contents

1. Project Overview
2. Technology Stack
3. Directory Structure
4. Backend Documentation
5. Frontend Documentation
6. Database Schema
7. API Documentation
8. Component Documentation
9. Implementation Details

Project Overview

PDF Chat is a Retrieval-Augmented Generation (RAG) application that enables users to have intelligent conversations with PDF documents. The application processes PDF files, extracts text, generates embeddings, and uses semantic search to provide context-aware responses.

Technology Stack

Backend

- **FastAPI**: Modern Python web framework for building APIs
- **LangChain**: Framework for developing LLM-powered applications
- **Cohere**: AI platform providing embeddings and chat capabilities
- **PostgreSQL**: Database with pgvector extension for vector similarity search
- **PyMuPDF**: PDF processing library

Frontend

- **React**: UI library with TypeScript support
- **Vite**: Build tool and development server
- **TailwindCSS**: Utility-first CSS framework
- **React Router**: Client-side routing

- **React Hot Toast**: Notifications system

Directory Structure

```
pdf-chat/
├── backend/
│   ├── routes/
│   │   ├── chat.py
│   │   ├── files.py
│   │   └── upload.py
│   ├── utils/
│   │   ├── chunker.py
│   │   ├── delete.py
│   │   ├── getmeta.py
│   │   ├── storedata.py
│   │   └── storemeta.py
│   └── main.py
└── frontend/
    ├── src/
    │   ├── components/
    │   │   ├── FeatureCard.tsx
    │   │   ├── Navbar.tsx
    │   │   └── NavItem.tsx
    │   ├── routes/
    │   │   ├── About.tsx
    │   │   ├── Chat.tsx
    │   │   ├── Contact.tsx
    │   │   ├── Feature.tsx
    │   │   ├── NotFound.tsx
    │   │   ├── Pricing.tsx
    │   │   ├── Synced.tsx
    │   │   └── Upload.tsx
    │   └── App.tsx
    └── package.json
```

Backend Documentation

Main Application (main.py)

The entry point of the FastAPI application, configuring CORS and routing.

Routes

1. **upload.py**
    - Handles PDF file uploads
    - Processes files and stores metadata
    - Creates document embeddings
2. **chat.py**
    - Manages chat interactions
    - Implements semantic search
    - Handles context retrieval and response generation
3. **files.py**
    - Manages file operations (list, delete)
    - Handles both database and filesystem operations

Utilities

1. **chunker.py**
    - Text splitting functionality
    - Chunk size: 500 characters
    - Overlap: 50 characters
2. **storedata.py**
    - Document embedding generation
    - Vector storage in PostgreSQL
    - Integration with Cohere API
3. **storemeta.py**
    - File metadata management
    - Database schema creation
    - Transaction handling

Frontend Documentation

Components

1. **Navbar.tsx**

```
// Navigation component with active route highlighting
const Navbar = () => {
  // Route-aware navigation with animated underlines
}
```

2. **FeatureCard.tsx**

```
// Reusable feature display component
interface FeatureCardProps {
  icon: React.ReactNode;
  title: string;
  description: string;
}
```

Routes

1. **Chat.tsx**
    - Real-time chat interface
    - Message history management
    - File upload integration
    - Markdown rendering support
2. **Upload.tsx**
    - Drag-and-drop file upload
    - File validation
    - Upload progress indication
    - Error handling
3. **Synced.tsx**
    - File management interface
    - Delete functionality
    - Loading states
    - Error handling

Database Schema

Files Table

```
CREATE TABLE files (
    file_id SERIAL PRIMARY KEY,
    file_location VARCHAR(255),
    file_name VARCHAR(255) UNIQUE,
    file_size INT,
    file_type VARCHAR(255),
    user_gmail VARCHAR(255)
);
```

Embeddings Table

```
CREATE TABLE embeddings (
    id SERIAL PRIMARY KEY,
    text TEXT,
    embedding vector(1024),
    file_id INT REFERENCES files(file_id) ON DELETE CASCADE
);
```

API Documentation

Upload Endpoints

- `POST /upload`
  - Accepts multipart/form-data
  - Returns file metadata and ID
  - Processes document for chat

Chat Endpoints

- `POST /chat`
  - Accepts JSON with message
  - Returns AI-generated response
  - Uses semantic search for context

File Management Endpoints

- `GET /files`
  - Lists all uploaded files

- `DELETE /files/{file_id}`
  - Removes file and associated vectors
- `GET /files/local`
  - Lists files in filesystem
- `DELETE /files/local/{filename}`
  - Removes local file

Component Documentation

State Management

```
// Message type definition
type Message = {
  id: number;
  text: string;
  sender: "user" | "bot";
};

// Chat state hooks
const [messages, setMessages] = useState<Message[]>([]);
const [inputMessage, setInputMessage] = useState("");
```

UI Components

```
// Feature card component with hover effects
<FeatureCard
  icon={<MessageSquare className="w-6 h-6 text-blue-600" />}
  title="Intelligent Chat"
  description="Natural conversations about PDF content"
/>

// Toast notifications
toast.success("PDF uploaded successfully!");
toast.error("Failed to upload PDF");
```

Utility Functions

```
// PDF chunk processing
const chunker = (data: string) => {
  const text_splitter = new RecursiveCharacterTextSplitter({
    chunkSize: 500,
    chunkOverlap: 50
  });
  return text_splitter.splitText(data);
};

// Vector similarity search
const searchSimilar = async (query: string) => {
  const embedding = await embeddings.embedQuery(query);
  // PostgreSQL similarity search
};
```

Implementation Details

PDF Processing Pipeline

1. Document Upload Flow

```
graph LR
    A[Upload PDF] --> B[Store File]
    B --> C[Extract Text]
    C --> D[Split into Chunks]
    D --> E[Generate Embeddings]
    E --> F[Store in PostgreSQL]
```

2. Chat Flow

```
graph LR
    A[User Query] --> B[Generate Query Embedding]
    B --> C[Vector Similarity Search]
    C --> D[Retrieve Context]
    D --> E[Generate Response]
    E --> F[Return to User]
```

Core Functions Documentation

Text Processing

```
# Text Chunking Configuration
CHUNK_SIZE = 500  # Characters per chunk
CHUNK_OVERLAP = 50  # Characters overlap between chunks

# Vector Dimensions
EMBEDDING_DIMENSIONS = 1024  # Cohere embedding size
```

Database Interactions

- **Cascade Deletes**: Files and their embeddings are automatically cleaned up
- **Vector Indexing**: Uses pgvector's HNSW indexing for fast similarity search
- **Connection Pooling**: Implements connection pooling for better performance

Security Measures

1. **File Validation**
    - Size limits
    - File type checks
    - Malware scanning (recommended addition)
2. **API Security**
    - CORS configuration
    - Rate limiting (recommended addition)
    - Input validation

Performance Optimizations

Backend Optimizations

1. **Database**
    - Indexed vector searches
    - Efficient chunking strategy
    - Batched embedding generation
2. **File Processing**
    - Async file handling
    - Streaming for large files

- Memory-efficient text extraction

Frontend Optimizations

1. **React Components**
   - Memoized components
   - Efficient re-renders
   - Lazy loading for routes
2. **UI/UX**
   - Optimistic updates
   - Progressive loading
   - Debounced input handling

Error Handling

Backend Error Handling

```
try:
    # Database operations
    with conn.cursor() as curr:
        curr.execute(...)
except Exception as e:
    conn.rollback()
    logger.error(f"Database error: {e}")
    raise HTTPException(status_code=500, detail="Internal server error")
finally:
    conn.close()
```

Frontend Error Handling

```
try {
  const response = await fetch("...");
  if (!response.ok) throw new Error("Request failed");
  // Handle success
} catch (error) {
  toast.error("Operation failed");
  // Handle error state
}
```

Testing Strategy

Unit Tests (Recommended)

- Backend route handlers
- Database utilities
- Text processing functions
- React components

Integration Tests (Recommended)

- PDF upload flow
- Chat interaction flow
- File management operations
- Database operations

End-to-End Tests (Recommended)

- Complete user journeys
- Cross-browser testing
- Performance testing

Deployment Considerations

Backend Deployment

- Use gunicorn for production
- Configure worker processes
- Set up SSL/TLS
- Configure PostgreSQL for production

Frontend Deployment

- Build optimization
- Asset compression
- CDN integration
- Environment configuration

Monitoring and Logging

Recommended Metrics

- API response times
- Database query performance
- PDF processing duration
- Error rates and types

Logging Strategy

- Structured logging
- Error tracking
- Performance monitoring
- User analytics