Oui, **absolument**. Une documentation technique professionnelle **doit** inclure des détails clairs sur le frontend si l'application en contient un. Cela facilite la compréhension pour d'autres développeurs, contribue à la maintenabilité, et permet une meilleure collaboration entre les équipes frontend/backend.

Tu trouveras ci-dessous **la documentation technique complète mise à jour**, intégrant **le frontend Streamlit** de manière structurée, tout en conservant la rigueur et le style de la version précédente.

**📘 Documentation Technique Détaillée**

**📚 Contexte du Projet**

Ce projet a pour but de créer un système conversationnel avancé reposant sur des LLMs (Claude Sonnet, OpenAI, ou locaux via LM Studio) et enrichi par un mécanisme RAG (Retrieval-Augmented Generation). Il intègre également un module de gestion de notes personnelles vectorisées et activables.

L'application est conçue pour fonctionner en local, avec des extensions hybrides vers des APIs payantes si configurées.

**1. 🗄️ Base de données (PostgreSQL)**

**1.1 Schéma ERD (Entity Relationship Diagram)**

**Entités principales :**

* conversations
* messages
* rag\_corpus, documents, document\_chunks
* notes, note\_chunks
* llm\_configs, conversation\_context

**Relations clés :**

* conversation 1<->n messages
* rag\_corpus 1<->n documents
* documents 1<->n document\_chunks
* notes 1<->n note\_chunks
* llm\_configs 1<->n conversations

**1.2 Exemple de tables**

**Table: conversations**

| **Champ** | **Type** | **Contraintes** |
| --- | --- | --- |
| id | SERIAL | PK |
| title | TEXT | NOT NULL |
| created\_at | TIMESTAMP | DEFAULT now() |
| llm\_config\_id | INTEGER | FK -> llm\_configs(id) |

**Table: messages**

| **Champ** | **Type** | **Contraintes** |
| --- | --- | --- |
| id | SERIAL | PK |
| conversation\_id | INTEGER | FK -> conversations(id) |
| role | TEXT | CHECK ('user', 'assistant') |
| content | TEXT | NOT NULL |
| created\_at | TIMESTAMP | DEFAULT now() |

**Table: document\_chunks**

| **Champ** | **Type** | **Détail** |
| --- | --- | --- |
| chunk\_text | TEXT | NOT NULL |
| embedding | VECTOR | Index vectoriel (CHROMADB) |

**1.3 Index**

* Index GIN/IVFFlat sur embedding
* Index temporel sur created\_at
* Index FK pour optimiser les jointures

**2. 📊 Architecture logicielle**

**2.1 Vue d'ensemble**

graph TD;

UI[Streamlit UI] --> API[FastAPI Backend];

API --> DB[PostgreSQL];

API --> VS[Vector Store (CHROMADB)];

API --> LLM[LLM (LM Studio / API)];

**2.2 Structure des dossiers**

/backend

├── api/

│ └── routes/ # Endpoints REST

├── rag/

│ ├── loader.py # Lecture PDF

│ ├── chunker.py # Split texte

│ ├── embedder.py # Embeddings

│ └── store.py # CHROMADB index

├── llm/

│ ├── router.py

│ └── providers/

│ ├── anthropic.py

│ ├── openai.py

│ └── local.py

├── conversations/

│ ├── controller.py

│ └── context\_manager.py

/frontend

└── app.py # Interface utilisateur Streamlit

**3. 🔍 RAG & Notes Vectorielles**

**3.1 Pipeline RAG**

1. Upload d’un document PDF
2. Extraction du texte
3. Découpage (chunking) par taille ou paragraphe
4. Embedding local (e5-base, MiniLM, Instructor, etc.)
5. Indexation vectorielle (CHROMADB)
6. Recherche contextuelle en fonction de la requête utilisateur
7. Activation et désactivation de chaque RAG depuis la conversation

**3.2 Notes personnelles**

* CRUD sur les notes (titre, contenu)
* Pipeline identique : chunk + embedding + index
* Activation et désactivation des notes (mini-rag) depuis la conversation

**4. 🔄 Appels LLM**

**4.1 Gestion dynamique des appels**

* Choix du provider : local (LM Studio) ou cloud (Claude, GPT)
* Prompt = Instructions système + contexte (notes & chunks RAG) + message utilisateur
* Configurations centralisées dans llm\_configs

**4.2 Exemple d’appel**

{

"model": "claude-3-sonnet",

"prompt": "Voici le contexte ...\nQuestion: ...",

"temperature": 0.7,

"max\_tokens": 1024

}

**5. 📊 Format JSON (Frontend <-> Backend)**

**5.1 Requête**

{

"conversation\_id": 42,

"content": "Que dit ce document ?",

"llm\_config\_id": 1,

"active\_rags": [2],

"active\_notes": [5, 8]

}

**5.2 Réponse**

{

"role": "assistant",

"content": "Voici un résumé...",

"sources": [

{"type": "rag", "id": 2, "score": 0.89},

{"type": "note", "id": 8, "score": 0.72}

]

}

## 🎨 6. Frontend — Interface Utilisateur (Streamlit)

### 6.1 Objectifs techniques du frontend

L’interface utilisateur est construite avec **Streamlit**, pour offrir un accès simple, local et interactif aux fonctionnalités du système conversationnel. Elle agit comme une **couche de présentation** au-dessus de l’API FastAPI, permettant aux utilisateurs de :

* Créer et gérer des conversations.
* Interagir avec les LLMs via chat enrichi.
* Activer/désactiver dynamiquement des corpus RAG ou des notes vectorisées.
* Gérer les configurations de modèles LLM et les documents.

### 6.2 Structure technique

#### 📁 Arborescence indicative

/frontend

├── app.py # Entrée principale Streamlit

├── pages/

│ ├── 1\_Conversations.py

│ ├── 2\_RAG\_Manager.py

│ ├── 3\_LLM\_Configs.py

│ └── 4\_Notes.py

├── components/

│ ├── chat\_ui.py # UI du chat

│ ├── rag\_ui.py # Upload/gestion de documents

│ ├── notes\_ui.py # CRUD sur notes vectorielles

│ └── model\_selector.py # Dropdown LLM

├── services/

│ └── api\_client.py # Interfaçage avec l’API FastAPI

### 6.3 Pages et composants

#### 1. 🧵 Conversations

* Liste des conversations (appel GET /conversations)
* Création d’une nouvelle conversation (POST)
* Suppression (DELETE)
* Redirection vers l’interface de chat
* Composant principal : chat\_ui.display\_chat(conversation\_id)

#### 2. 📄 Gestion des Corpus RAG

* Visualisation des corpus (GET /rag)
* Upload de documents (POST /rag/upload)
* Chunking + embedding déclenchés côté backend
* Suppression de corpus
* Checkbox d’activation contextuelle d’un ou plusieurs RAG

#### 3. 🤖 Gestion des modèles LLM

* Liste des configurations LLM (GET /llm/configs)
* Ajout d’un nouveau provider (formulaire avec : nom, API, modèle, URL, temperature, max tokens)
* Sélection d’un modèle actif via dropdown pour chaque conversation

#### 4. 📝 Gestion des notes vectorisées

* Interface CRUD sur des notes textuelles
* Chunking + vectorisation côté backend (identique aux PDF)
* Activation dans la session de chat (checkbox)

### 6.4 Workflows UX/UI

#### ➤ Création de conversation

* Bouton ➕ → envoie une requête POST → redirection dynamique vers chat\_ui.

#### ➤ Envoi d’un message

* Saisie utilisateur → api\_client.send\_message()
* Contexte construit dynamiquement : prompt utilisateur + top-k chunks + notes activées
* Affichage des réponses + des sources (type RAG ou note)

#### ➤ Upload de document

* Sélection de fichiers → upload via Streamlit → /rag/upload
* Affichage du statut (progress bar ou toast)

### 6.5 Fonctionnalités spécifiques

| **Composant** | **Fonctionnalité** |
| --- | --- |
| chat\_ui.py | Interface de chat, affichage des messages, historique |
| rag\_ui.py | Upload de PDF, visualisation des corpus, suppression |
| model\_selector.py | Dropdown de modèles LLM configurables par utilisateur |
| notes\_ui.py | Ajout/édition/suppression de notes vectorisées |
| api\_client.py | Wrapper pour toutes les requêtes REST vers le backend |

### 6.6 Dépendances principales

* streamlit — UI réactive
* requests — communication avec l’API FastAPI
* pandas / plotly (optionnel) — visualisation de données
* dotenv — configuration locale
* uuid / json — gestion des identifiants et des payloads

### 6.7 Lancement du frontend

cd frontend

streamlit run app.py

### 6.8 Bonnes pratiques frontend

* Séparer logique UI (Streamlit) et appels API (dans /services)
* Utiliser st.session\_state pour le suivi des conversations et des configurations LLM
* Modulariser les composants pour réutilisation
* Utiliser st.cache\_data avec prudence pour éviter les erreurs sur les embeddings ou les résultats dynamiques

**7. 🤞 Tests recommandés**

**7.1 Unitaires**

* Tests sur vectorisation (mock)
* Vérification des routes API
* Parsing/Chunking PDF

**7.2 Fonctionnels**

* Upload + indexation document
* Dialogue avec RAG activé
* Notes vectorielles + recherche contextuelle
* Changement de modèle LLM (local vs. cloud)

**8. 🛠️ Technologies / Coûts**

| **Composant** | **Outil** | **Gratuit ?** |
| --- | --- | --- |
| LLM local | LM Studio + Mistral | ✅ |
| Embedding | SentenceTransformers | ✅ |
| Vector Store | CHROMADB | ✅ |
| API payante | Claude / OpenAI | Optionnel |
| Base de données | PostgreSQL | ✅ |
| Backend | FastAPI | ✅ |
| Frontend | Streamlit | ✅ |

**9. 📃 Instructions d’installation (local)**

# Installer les dépendances Python

pip install -r requirements.txt

# Démarrer CHROMADB et PostgreSQL (via Docker recommandé)

# Lancer LM Studio localement (http://localhost:1234)

# Lancer le backend FastAPI

uvicorn main:app --reload

# Lancer l’interface utilisateur

streamlit run frontend/app.py

**10. ⚖️ Bonnes pratiques**

* Séparer les providers LLM pour faciliter le switch (modularité)
* Logguer les erreurs API + LLM dans /logs
* Structurer les prompts : system + context + user
* Documenter les workflows d’indexation RAG + notes
* Reproductibilité via requirements.txt + Docker