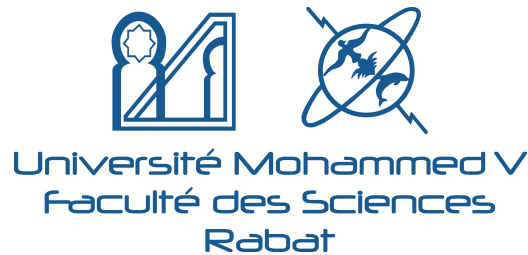


UNIVERSITÉ MOHAMMED V DE RABAT  
Faculté des Sciences



Département d'Informatique  
Filière Licence Fondamentale  
en Sciences Mathématiques et Informatique

---

PROJET DE FIN D'ÉTUDES

---

Intitulé :

Le titre du mémoire de Fin d'Etudes

Présenté par :

PRÉNOM NOM DU BINÔME1

PRÉNOM NOM DU BINÔME2

soutenu le xx Juin 2023 devant le Jury

M. Prénom Nom	Professeur à la Faculté des Sciences - Rabat	<i>Président</i>
M. Prénom Nom	Professeur à la Faculté des Sciences - Rabat	<i>Encadrant</i>
Mme Prénom Nom	Professeur à la Faculté des Sciences - Rabat	<i>Examineur</i>

Année Universitaire 2022-2023

# Remerciements

Au niveau de cette page, vous pouvez remercier votre encadrant, les membres du jury ainsi que toute personne qui vous a aidé à réaliser votre projet de fin d'études.

Voici un exemple de remerciement que vous pouvez adapter :

Au terme de ce travail, nous tenons à exprimer notre profonde gratitude et nos sincères Remerciements à notre encadrant Professeur ....., pour tout le temps qu'il nous a consacré, ses conseils précieux, et pour la qualité de son suivi durant toute la période de notre projet.

Nous tenons aussi à remercier vivement ..... (il faudra remercier aussi les membres de jury).

Nos plus vifs remerciements s'adressent aussi à .....(vous pouvez remercier aussi d'autres personnes).

Nos remerciements vont enfin à toute personne qui a contribué de près ou de loin à l'élaboration de ce travail.

# Résumé

Le projet consiste à développer un chatbot pour une application qui gère deux types d'utilisateurs : les administrateurs et les clients. Les administrateurs peuvent gérer les utilisateurs et leurs rôles, tandis que les clients peuvent utiliser les services fournis par l'application, uploader des fichiers (PDF, texte) et interagir avec le chatbot. L'application est développée en JavaScript avec le framework React et utilise une base de données MySQL.

**Mots clés :** chatbot, Langchain, supabase, SMI, FSR

# Abstract

The main objective of this ...

**Keywords :** TeX, LaTeX, PFE, SMI, FSR

# Table des matières

<b>Remerciements</b>	<b>i</b>
<b>Résumé</b>	<b>ii</b>
<b>Abstract</b>	<b>iii</b>
<b>Introduction</b>	<b>vi</b>
<b>1 Contexte Général du Projet</b>	<b>1</b>
1.1 Présentation du Projet . . . . .	1
1.2 Contexte du Projet . . . . .	2
1.3 Présentation de l'Application . . . . .	2
1.4 Objectif . . . . .	3
<b>2 Analyse et Spécification</b>	<b>4</b>
2.1 Étude de l'Existant . . . . .	4
2.2 Proposition de Différentes Solutions . . . . .	5
2.3 Enoncé du Besoin . . . . .	5
2.4 Fonctionnalités du Système . . . . .	6
2.5 Enoncé du Besoin . . . . .	6
<b>3 Conception du Système</b>	<b>8</b>
3.1 Architecture Générale du Système . . . . .	8
3.1.1 Utilisateur et Administrateur : . . . . .	8
3.1.2 Fonctionnement du Chatbot : . . . . .	8
3.1.3 Traitement des Fichiers Uploadés : . . . . .	9
3.1.4 Gestion des Questions : . . . . .	9
3.1.5 Utilisation du Modèle de Langage (LLM) : . . . . .	9
3.2 Modélisation UML (Unified Modeling Language) . . . . .	10
3.2.1 Diagramme de cas d'utilisation . . . . .	10

3.2.2	Diagramme de classe . . . . .	12
3.2.3	Diagramme de Séquence . . . . .	14
<b>4</b>	<b>implémentation</b>	<b>17</b>
4.1	Technologies et Outils Utilisés . . . . .	18
4.1.1	React . . . . .	18
4.1.2	Langchain . . . . .	18
4.1.3	Node.js . . . . .	19
4.1.4	Supabase . . . . .	20
4.2	Étape 1 (conception de la première version de notre applica- tion) : . . . . .	21
4.2.1	front-end : . . . . .	21
4.2.2	back-end : . . . . .	21
4.3	Étape 2 (conception de la 2eme version de notre application) :	26
4.4	Démonstration . . . . .	26
4.5	front-end : . . . . .	27
4.6	back-end : . . . . .	27
4.7	Étape 3 (conception de la 3eme version de notre application) :	44
4.7.1	front-end : . . . . .	44
4.7.2	back-end . . . . .	51
	<b>Conclusion</b>	<b>64</b>

# Introduction

Pour notre projet de fin d'études (PFE), nous avons entrepris de développer un chatbot avancé en utilisant Langchain, un framework puissant qui permet d'intégrer des modèles de langage large (LLMs) dans notre application. Node.js a été choisi pour l'implémentation backend en raison de sa performance et de sa flexibilité, tandis que Supabase a été utilisé comme base de données vectorielle pour gérer les embeddings de texte et améliorer la précision des réponses du chatbot.

# Chapitre 1

## Contexte Général du Projet

Le PFE ou Projet de Fin d'études est l'équivalent de deux modules Projets Tutorés : PT1 et PT2.

Le projet tutoré a pour objectif d'initier les étudiants à la recherche scientifique expérimentale, à la recherche bibliographique, aux stages en entreprises ainsi qu'à la rédaction de rapports scientifiques.

Votre mémoire de fin d'études devra être rédigé en LaTeX.

### 1.1 Présentation du Projet

Le développement des technologies de l'intelligence artificielle et du traitement du langage naturel a permis l'émergence de solutions innovantes dans divers domaines. Parmi celles-ci, les chatbots occupent une place importante en offrant des interactions automatisées et intelligentes entre les utilisateurs et les systèmes informatiques. Ce projet s'inscrit dans cette dynamique en proposant une application de chatbot multi-utilisateurs qui permet une interaction avancée avec les fichiers des utilisateurs stockés dans une base de données vectorielle.



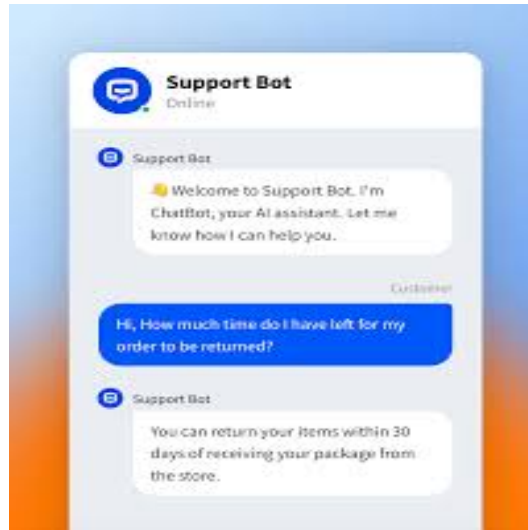


FIGURE 1.1.1 – chatbot

## 1.2 Contexte du Projet

Dans un monde de plus en plus connecté, les utilisateurs génèrent et stockent des quantités massives de données sous diverses formes de fichiers. La gestion et l'exploitation de ces données deviennent des tâches complexes et chronophages. Le besoin de solutions capables de comprendre et de manipuler ces données de manière efficace et intuitive est donc croissant. Le projet se base sur l'idée de fournir un outil intelligent qui facilite cette interaction, en permettant aux utilisateurs d'accéder et de traiter leurs fichiers de manière conversationnelle.

## 1.3 Présentation de l'Application

L'application développée dans le cadre de ce projet est un chatbot multi-utilisateurs. Elle utilise des technologies avancées de traitement du langage naturel et une base de données vectorielle pour stocker les fichiers des utilisateurs. Cette base de données vectorielle permet une recherche rapide et efficace, en utilisant des techniques de correspondance sémantique pour comprendre le contenu des fichiers et répondre de manière pertinente aux requêtes des utilisateurs.

## 1.4 Objectif

L'objectif principal de ce projet est de créer une application de chatbot capable de :

- Permettre aux utilisateurs d'interagir de manière intuitive avec leurs fichiers stockés.
- Utiliser une base de données vectorielle pour optimiser la recherche et l'accès aux fichiers.
- Fournir des réponses précises et pertinentes en utilisant des algorithmes avancés de traitement du langage naturel.

En résumé, ce projet vise à améliorer l'expérience utilisateur dans la gestion et l'exploitation de leurs données personnelles, en offrant une interface conversationnelle intelligente et performante.

# Chapitre 2

## Analyse et Spécification

### 2.1 Étude de l’Existant

#### Analyse de l’Existant

Avant de concevoir notre solution, il est crucial de comprendre les solutions actuelles disponibles sur le marché et leurs limitations. Les chatbots existants offrent principalement des services d’interaction basiques avec les utilisateurs, comme répondre à des questions simples, fournir des informations générales ou exécuter des tâches prédéfinies. Cependant, peu de solutions permettent une interaction approfondie avec les fichiers des utilisateurs, surtout en utilisant une base de données vectorielle pour une recherche sémantique avancée. Voici une analyse des principales technologies de chatbots existantes, notamment ChatGPT, Gemini et Claude.

**ChatGPT** Développé par OpenAI, ChatGPT utilise des modèles de langage naturel pour générer des réponses pertinentes. Cependant, la version gratuite présente certaines limitations :

- **Précision limitée** : Réponses parfois inexactes ou hors sujet.
- **Pas d’accès aux fichiers** : Pas d’interaction directe avec les fichiers des utilisateurs.
- **Personnalisation restreinte** : Capacités de personnalisation limitées.

**Gemini** Développé par Google DeepMind, Gemini est conçu pour des interactions complexes mais présente aussi des limitations :

- **Accès et intégration complexes** : Nécessite une expertise technique pour l’intégration.

- **Fonctionnalités limitées en version gratuite** : Fonctionnalités avancées réservées aux versions payantes.
- **Pas d'interaction avec les fichiers** : Principalement axé sur les réponses textuelles.

**Claude** Développé par Anthropic, Claude offre des capacités de conversation sophistiquées mais présente des limitations similaires :

- **Performance variée** : Réponses parfois imprécises.
- **Interaction limitée avec les fichiers** : Ne gère pas les fichiers des utilisateurs.
- **Personnalisation insuffisante** : Adaptation limitée aux besoins spécifiques des utilisateurs.

## 2.2 Proposition de Différentes Solutions

L'étude de l'existant nous a permis de dégager plusieurs anomalies. Plusieurs approches peuvent être envisagées pour développer notre application de chatbot multi-utilisateurs :

- **Solution Basique** : Un chatbot qui permet de répondre à des questions simples et d'effectuer des tâches limitées, sans interaction avec les fichiers des utilisateurs.
- **Solution Intermédiaire** : Un chatbot intégré avec une base de données traditionnelle, capable de répondre à des questions et d'effectuer des tâches basiques de gestion de fichiers.
- **Solution Avancée** : Un chatbot utilisant une base de données vectorielle pour offrir une recherche sémantique avancée et des interactions complexes avec les fichiers des utilisateurs. Cette solution inclut des fonctionnalités de gestion des utilisateurs, d'authentification et de personnalisation des réponses.

## 2.3 Enoncé du Besoin

L'application doit répondre aux besoins suivants :

- Faciliter l'accès et la gestion des fichiers des utilisateurs.
- Offrir une interface utilisateur intuitive et conversationnelle.

- Utiliser des techniques avancées de traitement du langage naturel pour comprendre et répondre aux requêtes des utilisateurs de manière pertinente.
- Gérer plusieurs utilisateurs avec des niveaux d'accès et des rôles différents.

## 2.4 Fonctionnalités du Système

### Utilisateurs

- **Admin :**
  - Gérer les utilisateurs (ajout, suppression, modification des utilisateurs).
- **Visiteur :**
  - S'inscrire sur la plateforme.
- **Client :**
  - Interagir avec leurs fichiers stockés dans la base de données vectorielle.
  - Interagir avec le chatbot.
  - Gérer le profil.

## 2.5 Enoncé du Besoin

### Fonctions Principales

- **Interaction Conversationnelle :** Permettre aux utilisateurs d'interagir avec le chatbot via une interface conversationnelle.
- **Recherche Sémantique :** Utiliser une base de données vectorielle pour offrir des capacités de recherche avancées.
- **Gestion des Fichiers :** Faciliter l'accès, la modification et la gestion des fichiers des utilisateurs.
- **Gestion des Utilisateurs :** Fournir des outils pour l'administration des utilisateurs et la gestion des accès.

### Fonctions Contraintes

- Sécurité des données et confidentialité des interactions.
- Scalabilité pour gérer un nombre croissant d'utilisateurs et de fichiers sans perte de performance.
- Précision dans les réponses fournies par le chatbot.

### Fonctions Complémentaires

- **Personnalisation** : Adapter les réponses et les interactions en fonction des préférences et de l'historique des utilisateurs.
- **Notifications** : Envoyer des notifications aux utilisateurs pour les informer des mises à jour ou des actions requises.
- **Support Multilingue** : Permettre des interactions dans plusieurs langues pour atteindre un public plus large.

# Chapitre 3

## Conception du Système

Les soutenances de PFE peuvent être effectuées durant les semaines du 06 et du 13 juin 2016, qui correspondent aux semaines de Délibérations et de Rattrapage.

### 3.1 Architecture Générale du Système

Le système de chatbot développé comporte plusieurs composants clés et flux de travail permettant une interaction efficace avec les utilisateurs et les administrateurs. Voici une description détaillée de ces éléments :

#### 3.1.1 Utilisateur et Administrateur :

Les utilisateurs peuvent s'inscrire via une page de registration et se connecter via une page de login. Une fois connectés, ils peuvent interagir avec le chatbot. Les administrateurs, quant à eux, peuvent également se connecter via une page de login. Ils disposent de fonctionnalités supplémentaires telles que l'ajout de clients pour les promouvoir en administrateurs, la suppression de clients et la modification des données des clients.

#### 3.1.2 Fonctionnement du Chatbot :

Le chatbot utilise le framework LangChain pour intégrer et utiliser un modèle de langage (LLM) dans l'application.

**Modes d'Utilisation :**

- **Option 'Basic'** : Dans ce mode, le chatbot répond uniquement à partir des données uploadées par l'utilisateur dans une base de don-

nées vectorielle. Si aucune donnée pertinente n'est trouvée, le chatbot informe l'utilisateur qu'il ne peut pas répondre à la question.

- **Option 'OpenAI' :** Ce mode permet au chatbot d'utiliser le modèle de langage (LLM) ChatGPT-3.5 pour répondre à toute question posée par l'utilisateur, en exploitant la puissance de traitement et de compréhension du langage naturel d'OpenAI.

### 3.1.3 Traitement des Fichiers Uploadés :

Lorsque l'utilisateur upload un fichier, celui-ci est divisé en petits morceaux. Chaque morceau est ensuite converti en une représentation vectorielle numérique. Cette conversion facilite la recherche d'informations pertinentes en utilisant des techniques de recherche vectorielle.

### 3.1.4 Gestion des Questions :

Lorsqu'un utilisateur pose une question, celle-ci est transformée en une "standalone question". Cette question résumée contient les éléments essentiels pour identifier les informations les plus proches dans la base de données vectorielle. La "standalone question" permet de trouver les informations les plus proches dans la base de données vectorielle. Ces informations sont ensuite rassemblées sous forme de texte pour être utilisées par le modèle de langage (LLM).

### 3.1.5 Utilisation du Modèle de Langage (LLM) :

Le modèle de langage ChatGPT-3.5 est utilisé pour répondre aux questions des utilisateurs, en exploitant les informations textuelles rassemblées à partir de la base de données vectorielle. Cela permet de fournir des réponses précises et pertinentes en temps réel.

Cette architecture assure une interaction fluide et efficace entre les utilisateurs, les administrateurs et le système de chatbot, en utilisant des techniques avancées de traitement du langage naturel et de gestion de données vectorielles, tout en s'appuyant sur le framework LangChain pour intégrer le modèle de langage (LLM) dans l'application.



## 3.2 Modélisation UML (Unified Modeling Language)

Pourquoi UML ?

L'utilisation de UML (Unified Modeling Language) dans le cadre de notre projet est justifiée par plusieurs raisons. Tout d'abord, UML est une norme largement acceptée pour la modélisation des systèmes logiciels, offrant une représentation visuelle claire et structurée de l'architecture et des composants du système. En utilisant UML, nous pouvons créer des diagrammes qui facilitent la communication entre les membres de l'équipe, les parties prenantes et les développeurs. Ces diagrammes incluent des cas d'utilisation, des diagrammes de classes, des diagrammes de séquence, et des diagrammes d'activités, chacun apportant une perspective différente mais complémentaire du système. De plus, UML aide à identifier les exigences du système de manière systématique et à détecter les problèmes potentiels dès les phases de conception. En résumé, UML permet de garantir une conception cohérente, de réduire les ambiguïtés et d'améliorer la qualité globale du développement logiciel.

### 3.2.1 Diagramme de cas d'utilisation

Nous avons construit un diagramme de cas d'utilisation (figure 2.1) pour représenter l'interaction entre les utilisateurs et le système.

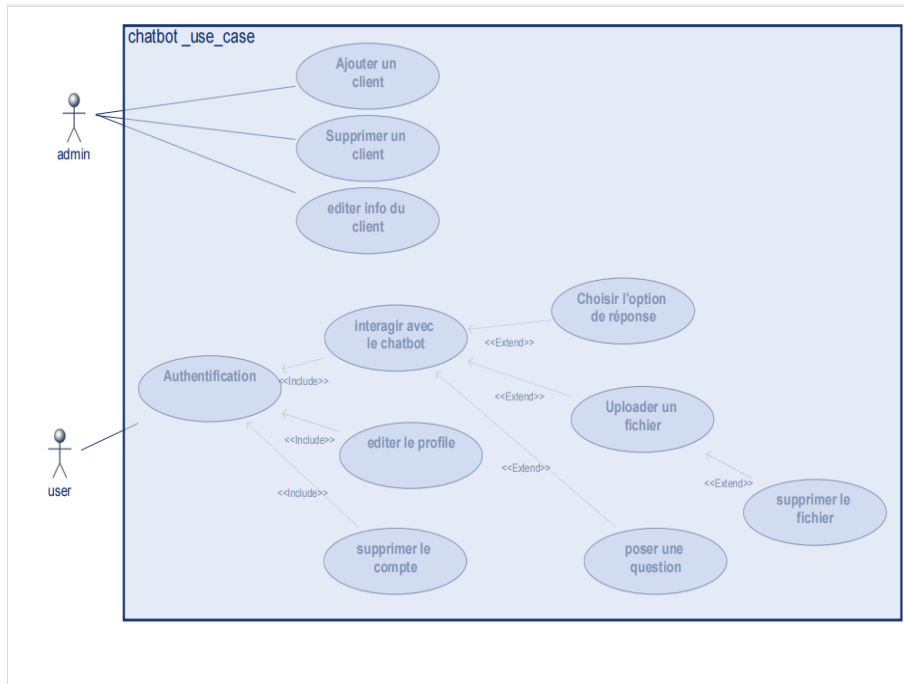


FIGURE 3.2.1 – Diagramme de cas d'utilisation

Comme nous l'avons montré dans le diagramme ci-dessus, il existe deux types d'utilisateurs : administrateur et client. L'administrateur peut ajouter un client pour qu'il devienne administrateur. Il est également capable de modifier les données d'un client ou de le supprimer. Le client peut télécharger un fichier et le supprimer. Il peut aussi éditer son profil ou supprimer son compte, et poser des questions et choisir l'option de réponse.

### 3.2.2 Diagramme de classe

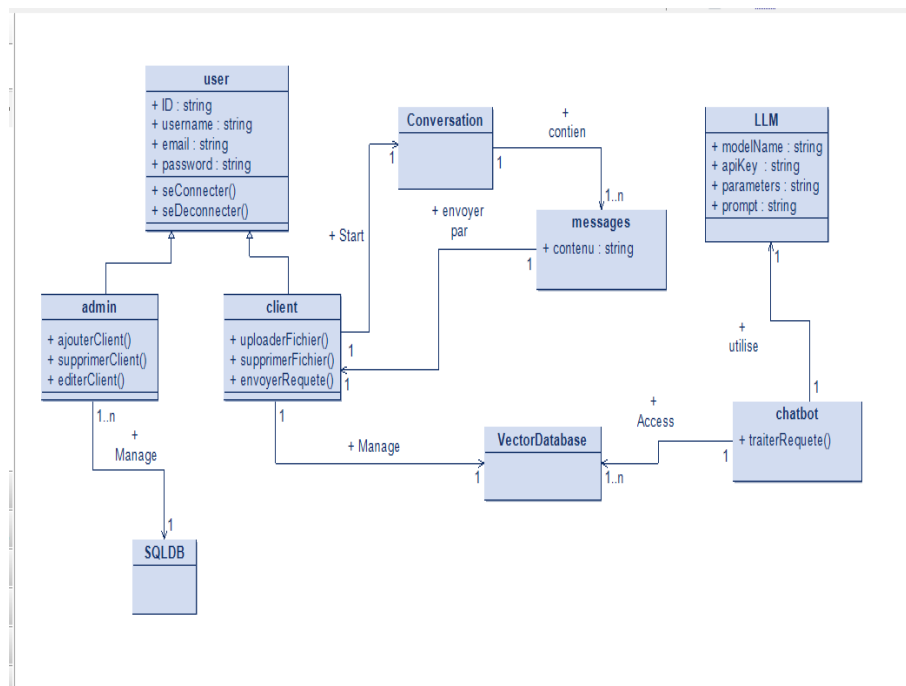


FIGURE 3.2.2 – Diagramme de classe

L'image ci-dessus représente un diagramme de classe modélisant un système de gestion de conversations et de traitement de requêtes par un chatbot utilisant un modèle de langage (LLM). Voici une description détaillée des classes et de leurs relations :

- **Classe ‘user’**

- **Attributs :**

- *ID* : *int* - Identifiant unique de l'utilisateur.
    - *username* : *string* - Nom d'utilisateur.
    - *email* : *string* - Adresse email de l'utilisateur.
    - *password* : *string* - Mot de passe de l'utilisateur.

- **Méthodes :**

- *seConnecter()* - Méthode permettant à l'utilisateur de se connecter.
    - *seDeconnecter()* - Méthode permettant à l'utilisateur de se déconnecter.

- **Relations :**

- *admin* et *client* sont des sous-classes de *user*, héritant de ses attributs et méthodes.

- **Classe ‘admin’**
  - **Méthodes :**
    - *ajouterClient()* - Méthode pour ajouter un client.
    - *supprimerClient()* - Méthode pour supprimer un client.
    - *editerClient()* - Méthode pour éditer les informations d’un client.
  - **Relations :**
    - Un *admin* gère un ou plusieurs (*1.n*) enregistrements dans *SQLDB*.
- **Classe ‘client’**
  - **Méthodes :**
    - *uploaderFichier()* - Méthode pour uploader un fichier.
    - *supprimerFichier()* - Méthode pour supprimer un fichier.
    - *envoyerRequete()* - Méthode pour envoyer une requête.
  - **Relations :**
    - Un *client* commence (*Start*) une ou plusieurs (*1.n*) *Conversation*.
    - Un *client* gère (*Manage*) une ou plusieurs (*1.n*) entrées dans *VectorDatabase*.
- **Classe ‘Conversation’**
  - **Relations :**
    - Une *Conversation* est initiée par (*Start*) un *client*.
    - Une *Conversation* contient (*contenir*) un ou plusieurs (*1.n*) *messages*.
- **Classe ‘messages’**
  - **Attributs :**
    - *contenu : string* - Le contenu du message.
  - **Relations :**
    - Un *message* est envoyé par (*envoyer par*) une *Conversation*.
- **Classe ‘VectorDatabase’**
  - **Relations :**
    - La *VectorDatabase* est gérée (*Manage*) par un *client*.
    - La *VectorDatabase* donne accès (*Access*) au *chatbot*.
- **Classe ‘chatbot’**
  - **Méthodes :**
    - *traiterRequete()* - Méthode pour traiter une requête.
  - **Relations :**
    - Le *chatbot* utilise (*utilise*) un modèle de langage (*LLM*).
    - Le *chatbot* accède (*Access*) à la *VectorDatabase*.

- Classe ‘LLM’
- Attributs :
  - *modelName* : *string* - Le nom du modèle de langage.
  - *apiKey* : *string* - La clé API pour accéder au modèle.
  - *parameters* : *float* - Les paramètres utilisés pour configurer le modèle.
  - *prompt* : *string* - Le prompt utilisé pour interagir avec le modèle.
- Classe ‘SQLDB’
- Relations :
  - La base de données SQL (*SQLDB*) est gérée par (*Manage*) un ou plusieurs (*1.n*) *admin*.

### 3.2.3 Diagramme de Séquence

inscription du client :

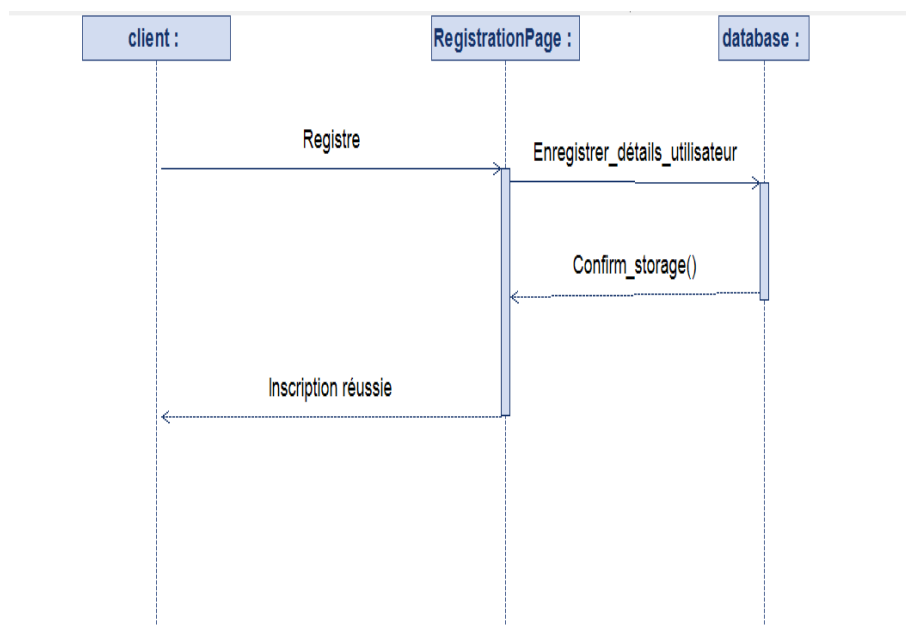


FIGURE 3.2.3 – inscription

login du client :

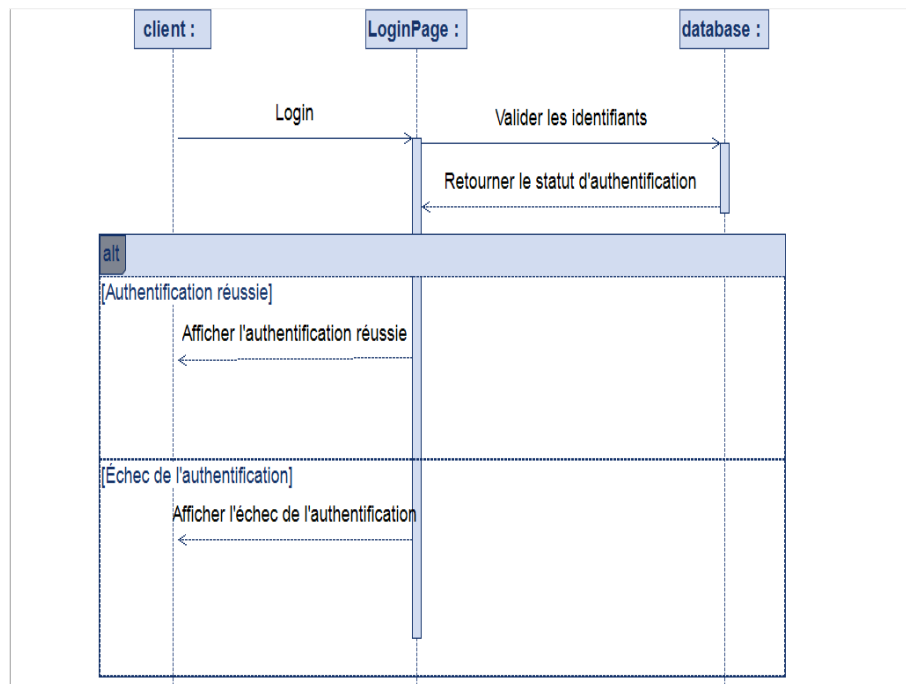


FIGURE 3.2.4 – login

admin session :

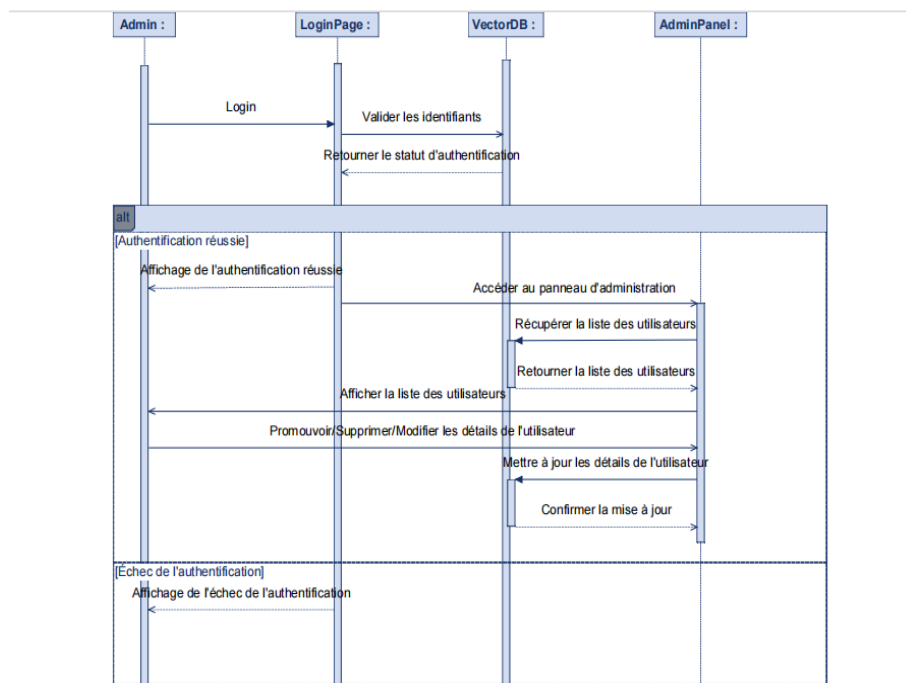


FIGURE 3.2.5 – admin session

upload file :

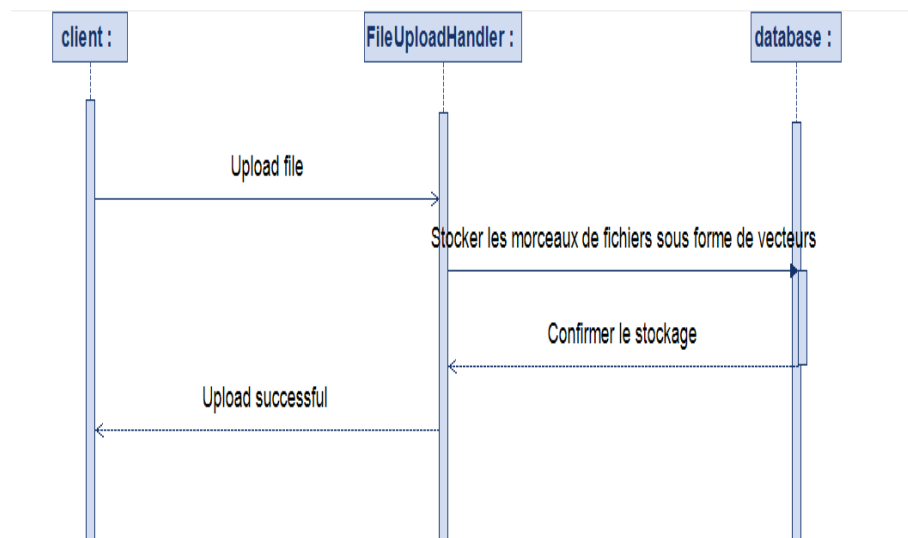


FIGURE 3.2.6 – upload

client session :

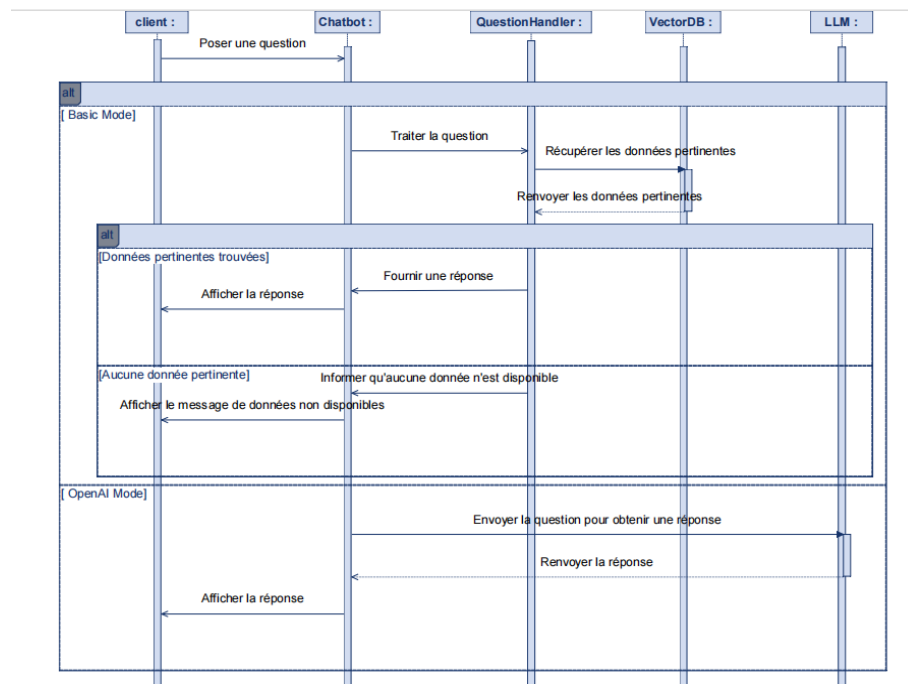


FIGURE 3.2.7 – client session

# Chapitre 4

## implémentation

Dans la phase d'implémentation de notre projet de PFE, nous avons développé notre chatbot en trois étapes distinctes. Dans la première version, nous avons mis en place un simple chatbot permettant à l'utilisateur de télécharger un fichier et de poser des questions à son sujet, ou bien de poser des questions générales auxquelles le chatbot répond en anglais. À ce stade, nous n'avons pas utilisé de base de données vectorielle ; le fichier est simplement téléchargé et interrogé directement.

Dans la deuxième version, nous avons enrichi notre application avec plusieurs améliorations. Nous avons ajouté la langue française, permettant ainsi au chatbot de répondre en français. Deux options ont été introduites : "Basic" et "OpenAI". L'option "Basic" restreint le chatbot à répondre uniquement aux questions en rapport avec les données téléchargées par l'utilisateur, informant ce dernier si une question sort du cadre des données disponibles. Pour améliorer l'efficacité de la recherche, nous avons intégré une base de données vectorielle en utilisant Supabase. Cela nous permet de diviser les données en petits morceaux, de les représenter sous forme de vecteurs et de les stocker dans la base de données vectorielle. À ce stade, l'interface restait simple, sans fonctionnalités d'administration ou de gestion des clients.

Dans la troisième version, nous avons ajouté des fonctionnalités de gestion des utilisateurs, avec des rôles de client et d'administrateur. Les clients peuvent créer un compte et se connecter, chaque client ayant sa propre base de données vectorielle. Ils peuvent télécharger et supprimer des fichiers, poser des questions au chatbot et choisir l'option de réponse. Cette version permet une interaction plus personnalisée et sécurisée, chaque utilisateur gérant ses propres données de manière autonome.



## 4.1 Technologies et Outils Utilisés

### 4.1.1 React

React est une bibliothèque JavaScript populaire utilisée pour construire les interfaces utilisateur. Développée par Facebook, React permet aux développeurs de créer des applications web larges qui peuvent se mettre à jour et se rendre efficacement en réponse aux changements de données. Nous avons utilisé React pour développer l'interface de notre chatbot.

### 4.1.2 Langchain

Langchain est un framework lancé en octobre 2022, utilisé pour simplifier le développement d'applications qui utilisent des modèles de langage large (LLM) comme OpenAI's GPT-4 ou GPT-3.5. Langchain permet de composer de grandes quantités de données qui peuvent facilement être référencées par un LLM avec le moins de puissance de calcul possible. Il fonctionne en prenant une grande source de données, comme un PDF de 50 pages, et en la décomposant en "morceaux" qui sont ensuite intégrés dans une base de données vectorielle.

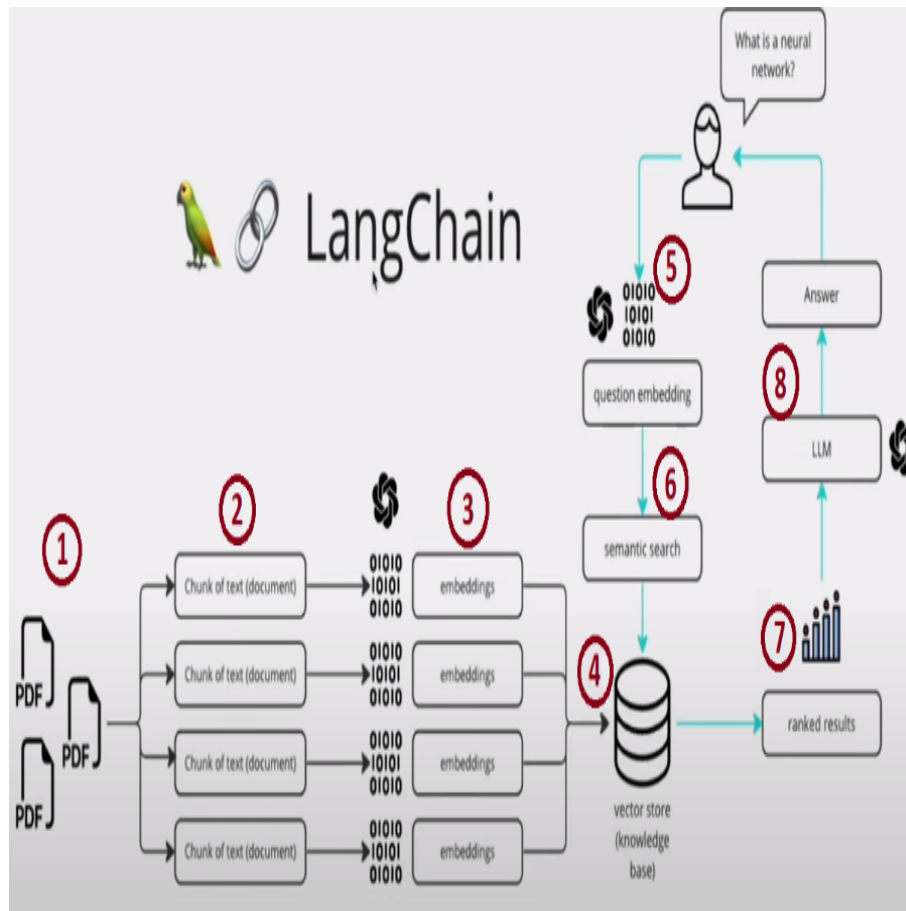


FIGURE 4.1.1 – Schéma de fonctionnement de LangChain

Les vector embeddings, ou plongements vectoriels, représentent sous forme de vecteurs les morceaux que nous avons obtenus à partir des documents que nous avons divisés. Cela nous permet de représenter différents types de données sous forme de points dans un espace multidimensionnel, où des points de données similaires sont regroupés à proximité les uns des autres. Maintenant que nous avons des représentations vectorielles du grand document, nous pouvons utiliser cela en conjonction avec le LLM pour récupérer uniquement les informations dont nous avons besoin pour être référencées lors de la création d'une paire de complétion de prompt.

### 4.1.3 Node.js

Node.js est un environnement d'exécution JavaScript côté serveur construit sur le moteur JavaScript V8 de Google Chrome. Il permet d'exécuter du code JavaScript côté serveur, ce qui était traditionnellement réservé

aux navigateurs web. Nous avons utilisé Node.js pour le développement du backend, en utilisant Express.js, un framework de Node.js. Nous avons également implémenté LangChain en utilisant Node.js.

#### 4.1.4 Supabase

Comme mentionné précédemment, nous devons représenter les documents sous forme de vecteurs et stocker chaque vecteur dans une base de données vectorielle. Il existe plusieurs choix pour la base de données vectorielle. Dans notre cas, nous avons utilisé Supabase. Supabase est une plateforme open source qui offre une base de données relationnelle et des fonctionnalités d'authentification en tant que service. Elle est construite sur PostgreSQL et fournit une API RESTful et des websockets en temps réel pour une intégration facile avec les applications web modernes.

## 4.2 Étape 1 (conception de la première version de notre application) :

### 4.2.1 front-end :

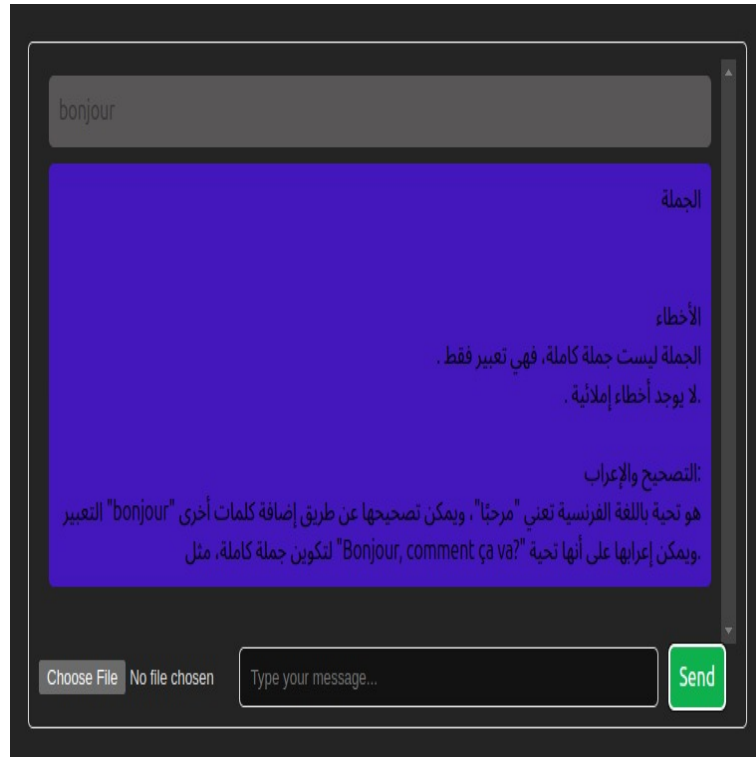


FIGURE 4.2.1 – front-end v1

Comme nous pouvons le voir sur la photo ci-dessous, l'interface est un cadre contenant un champ de saisie pour écrire le message au chatbot, un bouton "Envoyer" pour envoyer le message, et un bouton pour uploader des fichiers à traiter par le chatbot. La conversation contient les messages de l'utilisateur et les réponses du chatbot.

### 4.2.2 back-end :

server.js : Le fichier server.js est un script écrit en JavaScript qui implémente un serveur HTTP en utilisant le framework Express.js. Son rôle principal est de gérer les requêtes HTTP reçues par le serveur, notamment les téléchargements de fichiers et le traitement des données entrées par l'utilisateur.

```

import express from 'express';
import multer from 'multer';
import path from 'path';
import fs from 'fs';
import cors from 'cors';
import { PDFLoader } from "langchain/document_loaders/fs/pdf";
import { TextLoader } from "langchain/document_loaders/fs/text";
import { RecursiveCharacterTextSplitter } from "langchain/text_splitter";
import { ChatOpenAI } from "@langchain/openai";
import { createStuffDocumentsChain } from "langchain/chains/combine_documents";
import { ChatPromptTemplate } from "@langchain/core/prompts";
import { Document } from "@langchain/core/documents";

const app = express();
app.use(cors());
app.use(express.json()); // Parse JSON request body

const upload = multer({ dest: 'uploads/' });

app.post('/upload', upload.single('file'), async (req, res) => {
  if (!req.file) {
    return res.status(400).send('No file uploaded.');
```

```

  }

  try {
    const fileName = req.body.fileName || req.file.originalname;
    const filePath = path.join('uploads', fileName);
    await fs.promises.rename(req.file.path, filePath);
    res.status(200).send('File uploaded successfully');
    console.log(filePath);
  } catch (error) {
    console.error('Error uploading file:', error);
    res.status(500).send('Error uploading file');
  }
});

app.post('/process', async (req, res) => {
```

```

const { filePath, userInput } = req.body;

try {
  const chatModel = new ChatOpenAI({ apiKey: "sk-proj-5paUoRLCWJaigWZA",
    model: "gpt-3.5-turbo", temperature: 0.1 });
  const splitter = new RecursiveCharacterTextSplitter();
  const loader = filePath.endsWith(".pdf") ? new PDFLoader(filePath, "
  const splitDocs = await loader.loadAndSplit(splitter);

  const prompt = ChatPromptTemplate.fromTemplate('**Hey there!** I'm C
  const documentChain = await createStuffDocumentsChain({ llm: chatMod

  const context = [new Document({ pageContent: splitDocs.map(doc => do
  console.log(userInput)
  const output = await documentChain.invoke({ input: userInput, contex

  res.status(200).json({ output });
} catch (error) {
  console.error('Error processing file:', error);
  res.status(500).send('Error processing file');
}
});

const PORT = 3000;
app.listen(PORT, () => {
  console.log('Server is running on port ${PORT}');
});

```

1. **Import des modules :** Le code commence par importer les modules nécessaires à partir de différentes bibliothèques comme Express.js, multer pour la gestion des fichiers téléchargés, path et fs pour la manipulation des chemins de fichiers et des opérations de fichiers, et d'autres modules liés à la manipulation des documents et des modèles AI.

```

import { PDFLoader } from "langchain/document_loaders/fs/pdf";
import { TextLoader } from "langchain/document_loaders/fs/text";

```

Ces deux modules 'PDFLoader' et 'TextLoader' nous permet de charger les documents .PDF et .txt pour les traiter ensuite.

```
import { RecursiveCharacterTextSplitter } from "langchain/text_spli
```

Ce module permet d'instancier un objet de type RecursiveCharacterTextSplitter qui divise les fichiers téléchargés en petits morceaux de taille égale.

```
import { ChatOpenAI } from "@langchain/openai";
```

Ces modules permettent d'utiliser l'API d'une LLM comme celui de ChatGPT. En instanciant un objet de type ChatOpenAI, nous pouvons configurer divers paramètres tels que l'API key, le nom du modèle (optionnel) et la température (optionnelle). Par exemple :

```
const chatModel = new ChatOpenAI({
  apiKey: "sk-proj-5paUoRLCWJaigWZAVBTVT3BlbkFJrqf0v0ec1Xg4PTqzKX
  model: "gpt-3.5-turbo",
  temperature: 0.1
});
```

Ici, nous instancions l'objet ChatOpenAI en fournissant la clé d'API fournie par OpenAI, le nom du modèle(optionnelle) (dans ce cas, 'gpt-3.5-turbo'), et une température de 0.1 (optionnelle). Notez que tous les modules précédemment décrits. sont des modules du framework LangChain.

```
import express from 'express';
import multer from 'multer';
import path from 'path';
import fs from 'fs';
import cors from 'cors';
```

**express** : Express.js est un framework web pour Node.js qui simplifie la création d'applications web en fournissant une API robuste pour le routage des requêtes HTTP, la gestion des middlewares et la création de routes. Il est souvent utilisé comme couche supérieure à Node.js pour simplifier le processus de création d'applications web.

**multer** : Multer est un middleware Node.js utilisé pour gérer les téléchargements de fichiers dans les applications web. Il permet de gérer les fichiers envoyés via des formulaires HTML en les stockant sur le serveur et en fournissant des informations sur les fichiers téléchargés, telles que leur nom et leur taille.

**path** : Le module path fournit des utilitaires pour travailler avec les chemins de fichiers et de répertoires dans Node.js. Il est utilisé pour manipuler les chemins de fichiers de manière portable et sécurisée, indépendamment du système d'exploitation.

**fs** : Le module fs (système de fichiers) est un module intégré à Node.js qui fournit des fonctionnalités pour interagir avec le système de fichiers local. Il permet de lire, d'écrire, de créer et de supprimer des fichiers et des répertoires sur le disque dur.

**cors** : CORS (Cross-Origin Resource Sharing) est un mécanisme de sécurité côté serveur qui permet à des ressources restreintes d'une page web d'être demandées depuis un domaine différent de celui où la ressource originale a été servie. Le module cors est un middleware Express.js qui permet de configurer les en-têtes CORS dans les réponses HTTP pour autoriser les demandes cross-origin.

2. **Configuration de l'application Express** : Ensuite, une instance de l'application Express est créée et configurée. Les middlewares **cors** et **express.json()** sont utilisés pour gérer les requêtes CORS et parser le corps des requêtes au format JSON.
3. **Gestion des téléchargements de fichiers** : Une route POST **/upload** est définie pour gérer les téléchargements de fichiers. Elle utilise le middleware **multer** pour stocker les fichiers téléchargés dans le répertoire **uploads**. Si aucun fichier n'est téléchargé, le serveur renvoie une réponse d'erreur. voici l'explication de quelque module :
4. **Traitement des données** : Une route POST **/process** est définie pour traiter les données entrées par l'utilisateur. Cette route attend deux paramètres dans le corps de la requête : **filePath** (chemin du



fichier téléchargé) et `userInput` (données saisies par l'utilisateur). Le serveur charge le contenu du fichier spécifié, le divise en documents individuels, puis utilise un modèle OpenAI pour générer une réponse basée sur les données saisies par l'utilisateur et le contexte du fichier.

5. **Création du serveur :** Enfin, le serveur est configuré pour écouter les connexions sur le port 3000. Une fois le serveur démarré, un message est affiché dans la console pour indiquer que le serveur est en cours d'exécution.

### 4.3 Étape 2 (conception de la 2eme version de notre application) :

### 4.4 Démonstration

Dans la deuxième version de notre application, nous avons ajouté la langue française. Le chatbot peut maintenant répondre aux questions en français à la place de l'anglais. Nous avons également ajouté deux options que le client peut choisir : l'option 'Basic' et l'option 'OpenAI' mentionnée précédemment. La chose la plus importante ajoutée récemment est la base de données vectorielle, que nous avons créée en utilisant Supabase. De plus, nous avons ajouté de nouvelles techniques pour générer les réponses.

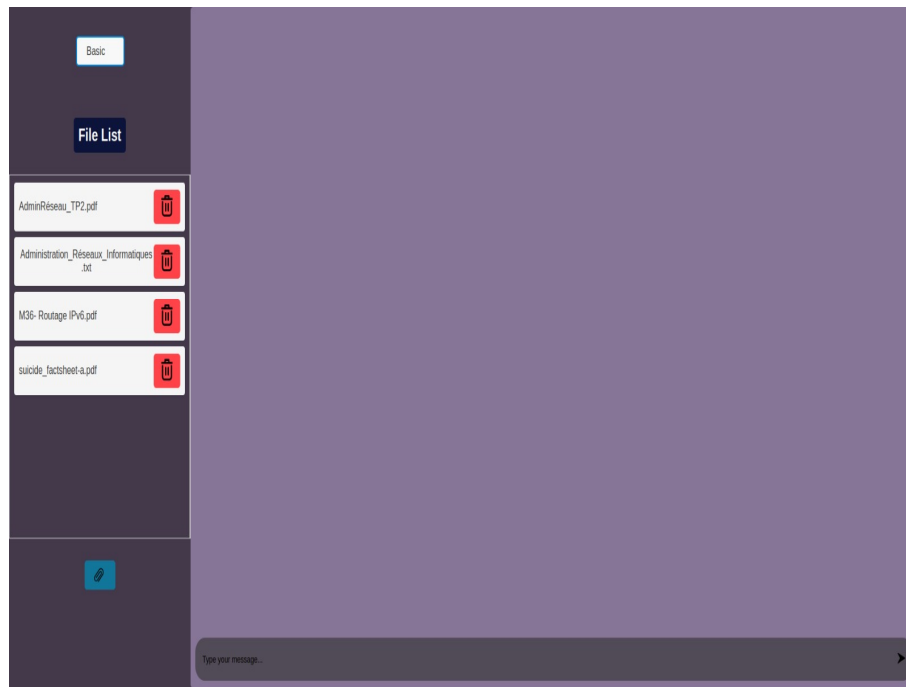


FIGURE 4.4.1 – fornt-endv2

Comme nous pouvons le voir sur la figure ci-dessous, nous avons ajouté une barre latérale pour afficher le nom des fichiers uploadés avec l'option de les supprimer. Nous avons également ajouté une liste déroulante avec deux choix : "Basic" et "OpenAI".

## 4.5 front-end :

À cette étape, nous avons conservé la même structure de code et de fichiers, mais avec de petites modifications (les nouveaux composants ajoutés). Pour plonger dans les détails du code, veuillez consulter la section 1.2 de l'annexe.

## 4.6 back-end :

-server.js :

```
import express from 'express';
import multer from 'multer';
import path from 'path';
import fs from 'fs';
```

```

import fsp from 'fs/promises';
import cors from 'cors';
import { dirname } from 'path';

import { PDFLoader } from "langchain/document_loaders/fs/pdf";
import { TextLoader } from "langchain/document_loaders/fs/text";
import { RecursiveCharacterTextSplitter } from "langchain/text_splitter";
import { ChatOpenAI } from "@langchain/openai";
import { createStuffDocumentsChain } from "langchain/chains/combine_documents";
import { ChatPromptTemplate } from "@langchain/core/prompts";
import { Document } from "@langchain/core/documents";

//Supabase :
import { SupabaseVectorStore } from "@langchain/community/vectorstores/supabase";
import { OpenAIEmbeddings } from "@langchain/openai";
import { createClient } from "@supabase/supabase-js";
import { StringOutputParser } from "@langchain/core/output_parsers";
import { RunnableSequence, RunnablePassthrough } from "@langchain/core/runnables";

import * as dotenv from "dotenv";
dotenv.config();
//SUPABASE_API_KEY :
const supabaseApiKey = process.env.SUPABASE_API_KEY;
if (!supabaseApiKey) throw new Error('Expected env var SUPABASE_PRIVATE_API_KEY');
//SUPABASE_URL :
const sbUrl=process.env.SUPABASE_URL;
if (!sbUrl) throw new Error('Expected env var SUPABASE_URL');
//OPENAI_API_KEY :
const openAIApiKey=process.env.OPENAI_API_KEY;
if (!openAIApiKey) throw new Error('Expected env var OPENAI_API_KEY');

// Template
const questionAutonomeTemplate = 'Étant donné une question, convertissez-la en une question autonome';
const modeleReponseTemplateBasic = ' **Enchanté !** Vous êtes un assistant autonome';
const modeleReponseTemplateOpenAi = ' **Enchanté !** Vous êtes un assistant OpenAI';

```

```

//Use express :
const app = express();
app.use(cors());
app.use(express.json()); // Parse JSON request body

//OpenAi :
const llm = new ChatOpenAI({
  apiKey: openAIApiKey,
  model: "gpt-3.5-turbo",
  temperature: 0.7 ,
  cache: true , });

//Splitter :
const splitter=new RecursiveCharacterTextSplitter({
  chunkSize:1000,
});

//Variable : (Select):
let selectedOption = 'basic';

//SelectOption
app.post('/my-react-app/select', (req, res) => {
  selectedOption = req.body.selectedOption;
  //console.log('Received selected option from client:', selectedOption)
  res.status(200).json({ message: 'Data received successfully!' });
});

let fileNames = [];

//Fetch

app.get('/my-react-app/get-files', async (req, res) => {
  try {
    const uploadDir = 'uploads';
    const files = await fs.promises.readdir(uploadDir);
  }

```

```

    fileName = files;
    res.json(files);

  } catch (error) {
    console.error('Error fetching files:', error);
    res.status(500).send('Failed to get files');
  }
});

//Client :
const client = createClient(sbUrl, supabaseApiKey);

// Uploads file : To (Vectore Databse + Directory) ;
const upload = multer({ dest: 'uploads/' });

app.post('/my-react-app/upload', upload.single('file'), async (req, res) {
  if (!req.file) {
    return res.status(400).send('No file uploaded.');
```

```

  }
  try {
    //Uploading file to server {/$uploads}
    const fileName = req.body.fileName || req.file.originalname;
    const filePath = path.join('uploads', fileName);
    await fs.promises.rename(req.file.path, filePath);
    const loader = filePath.endsWith(".pdf") ? new PDFLoader(filePath, "
    const splitDocU = await loader.loadAndSplit(splitter);
    //Uploading file to Vectore database :
    try {
      await SupabaseVectorStore.fromDocuments(
        splitDocU,
        new OpenAIEmbeddings(),
        {
          client,
          tableName: 'documents',
        }
      );
    }
  }
});
```

```

        console.log('Documents successfully stored with vector embeddings!
    } catch (error) {
        console.error('Error storing documents:', error);
        // Handle errors : file not uploading to vector database
    }
    res.status(200).send('File uploaded successfully');
    console.log(filePath);
} catch (error) {
    console.error('Error uploading file:', error);
    // Handle errors : file not uploading
    res.status(500).send('Error uploading file');
}
});

//Delete file :
//2 Steps : 1 - from vector database : / 2 - from uploads :done

//Embedding :
const embeddings = new OpenAIEmbeddings();

//the deVec:
const vectorStores = new SupabaseVectorStore(embeddings,{
    client,
    tableName:'documents',
    queryName:'match_documents'
});

app.delete('/my-react-app/delete/:index' , async (req, res) => {
    const index = parseInt(req.params.index);
    if (isNaN(index) || index < 0 || index >= fileNames.length) {
        return res.status(400).json({ error: 'Invalid file index' });
    }
    const fileName = fileNames[index];
    const filePath = path.join('uploads',fileName);
    const loader = filePath.endsWith(".pdf") ? new PDFLoader(filePath, "
    const output = await loader.loadAndSplit(splitter);

```

```

    try {
//Deleting file from Vectore database :
try {
  for (let i = 0; i < output.length; i++) {
    const { data: queryData, error: queryError } = await client
      .from('documents')
      .select('id')
      .eq('content', output[i].pageContent);

    if (queryError) {
      // Handle the error
      console.error("Error querying database:", queryError.message);
    } else {
      // Check if queryData is not empty
      if (queryData && queryData.length > 0) {
        // Store the id in a variable
        const id = queryData[0].id;

        const { data: deleteData, error: deleteError } = await client
          .from('documents')
          .delete()
          .eq('id', id);

        if (deleteError) {
          // Handle the deletion error
          console.error("Error deleting row:", deleteError.message);
        } else {
          console.log("Row deleted successfully.");
        }
      } else {
        console.log("No matching id found.");
      }
    }
  }
}
console.log('Documents successfully Deleted from vector store!');
} catch (error) {
  console.error('Error storing documents:', error);
}

```

```

    // Handle errors : file not uploading to vector database
  }

    fs.unlinkSync(filePath);
    fileNames.splice(index, 1);
    res.status(200).json({ message: 'File deleted successfully' });
  } catch (error) {
    console.error('Error deleting file:', error);
    res.status(500).json({ error: 'Error deleting file' });
  }

});

//Retreiver :
const retriever = vectorStores.asRetriever();

function combinerDocuments(docs) {
  return docs.map((doc) => doc.pageContent).join('\n\n');
}

//we use the standalone question batch njbdo chunks li 9rab l question

//Input -> output;

app.post('/my-react-app/process', async (req, res) => {
  const { userInput } = req.body;
  console.log('Received selected option from client:', selectedOption)
  let answerPrompt = '';
  //Prompt
  const standaloneQuestionPrompt =
    ChatPromptTemplate.fromTemplate(questionAutonomeTemplate);

  if(selectedOption == 'basic'){
    console.log(selectedOption);
    answerPrompt = ChatPromptTemplate.fromTemplate(modeleReponseTempla

```



```

}else if(selectedOption == 'openai'){
    console.log(selectedOption);
    answerPrompt =
ChatPromptTemplate.fromTemplate(modeleReponseTemplateOpenAi);
}

//QstChain :
const standaloneQuestionChain = standaloneQuestionPrompt
    .pipe(llm)
    .pipe(new StringOutputParser())
//RetreivalChain :
const retrieverChain = RunnableSequence.from([
    prevResult => prevResult.standalone_question,
    retriever,
    combinerDocuments
])
//AnswerChain :
const answerChain = answerPrompt
    .pipe(llm)
    .pipe(new StringOutputParser())
//Runnable :
const chainPrincipale = RunnableSequence.from([
    {
        standalone_question: standaloneQuestionChain,
        original_input: new RunnablePassthrough()
    },
    {
        context: retrieverChain,
        question: ({ original_input }) => original_input.question
    },
    answerChain
])
try {
    const reponse = await chainPrincipale.invoke({
        question: userInput
    });
}

```

```

        res.status(200).json({ output: reponse });
    } catch (error) {
        console.error('Erreur lors du traitement de la question :', error);
        res.status(500).send('Erreur lors du traitement de la question');
    }
});

const PORT = 3000;
app.listen(PORT, () => {
    console.log('Server is running on port ${PORT}');
});

```

Comme nous le remarquons ici dans le code, nous avons importé de nouveaux modules et fait quelques modifications que nous allons expliquer maintenant.

```
import { SupabaseVectorStore } from "@langchain/community/vectorstores/s
```

Nous avons importé le module `SupabaseVectorStore` depuis le package `@langchain/community/vectorstores/supabase` pour utiliser la base de données vectorielle fournie par Supabase.

```
import { OpenAIEmbeddings } from "@langchain/openai";
```

Nous avons importé le module `OpenAIEmbeddings` depuis le package `@langchain/openai`, qui nous permet d'instancier un objet. Cet objet permet de prendre les morceaux obtenus après avoir divisé les fichiers téléchargés par l'utilisateur et de les représenter sous forme de vecteurs numériques.

```
import { StringOutputParser } from "@langchain/core/output_parsers";
```

Nous avons importé le module `StringOutputParser` depuis le package `@langchain/core/output...` pour convertir les sorties du modèle en chaînes de caractères simples. Cela permet de manipuler plus facilement les réponses du modèle

```
import { RunnableSequence,RunnablePassthrough } from "@langchain/core/runnables";
```

De plus, nous avons importé les modules `RunnableSequence` et `RunnablePassthrough` depuis le package `@langchain/core/runnables`. `RunnableSequence` permet de chaîner plusieurs runnables ensemble, en passant la sortie d'un runnable comme entrée au suivant, facilitant ainsi la création de séquences de traitement complexes. `RunnablePassthrough`, quant à lui, permet de transmettre les entrées directement à travers un runnable sans les modifier, souvent utilisé pour combiner des données d'entrée avec des sorties de traitement précédentes.

```
import * as dotenv from "dotenv";
dotenv.config();
//SUPABASE_API_KEY :
const supabaseApiKey = process.env.SUPABASE_API_KEY;
if (!supabaseApiKey) throw new Error('Expected env var SUPABASE_PRIVATE_KEY');
//SUPABASE_URL :
const sbUrl=process.env.SUPABASE_URL;
if (!sbUrl) throw new Error('Expected env var SUPABASE_URL');
//OPENAI_API_KEY :
const openAIApiKey=process.env.OPENAI_API_KEY;
if (!openAIApiKey) throw new Error('Expected env var OPENAI_API_KEY');
```

Ce code nous permet d'importer dans notre programme la clé API d'OpenAI ainsi que la clé de l'API de Supabase. Il est nécessaire d'importer ces éléments pour pouvoir travailler avec les API d'OpenAI et de Supabase.

```
const questionAutonomeTemplate = 'Étant donné une question, convertissez-la en une question autonome';
const modeleReponseTemplateBasic = ' **Enchanté !** Vous êtes un assistant';
const modeleReponseTemplateOpenAi = ' **Enchanté !** Vous êtes un assistant';
```

'questionAutonomeTemplate' est un modèle qui demande aux LLM de ChatGPT de convertir la question de l'utilisateur en une question autonome, c'est-à-dire de construire une autre question qui contient juste les éléments importants. Cela permet ensuite de chercher les informations les plus proches dans la base de données vectorielle.

'modeleReponseTemplateBasic' ce template nous permet de guider le chatbot pour répondre seulement avec les données téléchargées par l'utilisateur. En revanche, avec 'modeleReponseTemplateOpenAi', le chatbot peut répondre à la question même s'il n'a pas de relation avec les données téléchargées dans la base de données. Nous avons créé tous les templates mentionnés précédemment en français pour contraindre le chatbot à parler en français.

```
const client = createClient(sbUrl, supabaseApiKey);
```

Cette ligne nous permet de nous connecter à Supabase, où nous avons créé notre base de données vectorielle.

```
const upload = multer({ dest: 'uploads/' });
```

```
app.post('/my-react-app/upload', upload.single('file'), async (req, res) => {
  if (!req.file) {
    return res.status(400).send('No file uploaded.');
```

```
  }
  try {
    //Uploading file to server {/$uploads}
    const fileName = req.body.fileName || req.file.originalname;
    const filePath = path.join('uploads', fileName);
    await fs.promises.rename(req.file.path, filePath);
    const loader = filePath.endsWith(".pdf") ? new PDFLoader(filePath, "
    const splitDocU = await loader.loadAndSplit(splitter);
    //Uploading file to Vectore database :
    try {
      await SupabaseVectorStore.fromDocuments(
        splitDocU,
        new OpenAIEmbeddings(),
        {
          client,
          tableName: 'documents',
        }
      );
    }
  }
});
```

```

        console.log('Documents successfully stored with vector embeddings!');
    } catch (error) {
        console.error('Error storing documents:', error);
        // Handle errors : file not uploading to vector database
    }
    res.status(200).send('File uploaded successfully');
    console.log(filePath);
} catch (error) {
    console.error('Error uploading file:', error);
    // Handle errors : file not uploading
    res.status(500).send('Error uploading file');
}
});

```

ce code utilise Multer, un middleware pour gérer les fichiers envoyés via les formulaires HTML, pour définir un dossier de destination ('uploads/') où les fichiers seront temporairement stockés avant d'être traités.

Il définit une route POST à l'URL '/my-react-app/upload' avec Express. Lorsque cette route est appelée, elle attend un fichier envoyé en utilisant le champ de formulaire nommé 'file'.

Lorsqu'un fichier est envoyé, il vérifie s'il existe. S'il n'y a pas de fichier, il renvoie une réponse d'erreur avec un statut 400 (Bad Request) et un message indiquant "Aucun fichier téléchargé".

Si un fichier est envoyé, il récupère le nom du fichier à partir du corps de la requête (req.body.fileName) ou utilise le nom original du fichier (req.file.originalname).

Ensuite, il déplace le fichier temporaire vers un chemin spécifié ('uploads/' + nomDuFichier) en utilisant les fonctions de système de fichiers de Node.js.

En fonction de l'extension du fichier (.pdf ou autre), il charge le contenu du fichier dans un format approprié à l'aide de différentes classes de chargement (PDFLoader ou TextLoader).

Il divise ensuite le contenu du document en unités (par exemple, des paragraphes) en utilisant une méthode de fractionnement spécifiée (splitter) et stocke ces unités dans un tableau (splitDocU).

Enfin, il utilise une méthode asynchrone pour stocker les documents avec leurs embeddings vectoriels associés dans la base de données, en utilisant

la classe `SupabaseVectorStore.fromDocuments()`. Les embeddings vectoriels sont générés à l'aide d'une classe `OpenAIEmbeddings`.

Si le stockage est réussi, il affiche un message de confirmation dans la console. Sinon, il affiche un message d'erreur.

Il renvoie une réponse avec un statut 200 (OK) indiquant que le fichier a été téléchargé avec succès.

Il gère les erreurs potentielles en affichant un message d'erreur dans la console et en renvoyant une réponse avec un statut 500 (Internal Server Error).

```
const embeddings = new OpenAIEmbeddings();

//the deVec:
const vectorStores = new SupabaseVectorStore(embeddings,{
  client,
  tableName:'documents',
  queryName:'match_documents'
});

app.delete('/my-react-app/delete/:index' , async (req, res) => {
  const index = parseInt(req.params.index);
  if (isNaN(index) || index < 0 || index >= fileNames.length) {
    return res.status(400).json({ error: 'Invalid file index' });
  }
  const fileName = fileNames[index];
  const filePath = path.join('uploads',fileName);
  const loader = filePath.endsWith(".pdf") ? new PDFLoader(filePath, "
  const output = await loader.loadAndSplit(splitter);
  try {
//Deleting file from Vectore database :
try {
  for (let i = 0; i < output.length; i++) {
    const { data: queryData, error: queryError } = await client
      .from('documents')
      .select('id')
      .eq('content', output[i].pageContent);

    if (queryError) {
```

```

        // Handle the error
        console.error("Error querying database:", queryError.message);
    } else {
        // Check if queryData is not empty
        if (queryData && queryData.length > 0) {
            // Store the id in a variable
            const id = queryData[0].id;

            const { data: deleteData, error: deleteError } = await client
                .from('documents')
                .delete()
                .eq('id', id);

            if (deleteError) {
                // Handle the deletion error
                console.error("Error deleting row:", deleteError.message);
            } else {
                console.log("Row deleted successfully.");
            }
        } else {
            console.log("No matching id found.");
        }
    }
}

console.log('Documents successfully Deleted from vector store!');
} catch (error) {
    console.error('Error storing documents:', error);
    // Handle errors : file not uploading to vector database
}

fs.unlinkSync(filePath);
fileNames.splice(index, 1);
res.status(200).json({ message: 'File deleted successfully' });
} catch (error) {
    console.error('Error deleting file:', error);
    res.status(500).json({ error: 'Error deleting file' });
}

```

```
});
```

cet code et pour la suppression de fichiers fonctionne de la manière suivante :

Une route DELETE est définie à  `'/my-react-app/delete/ '`. Lorsqu'elle est appelée avec un index spécifique, elle supprime le fichier associé et ses données correspondantes de la base de données.

Elle vérifie d'abord si l'index fourni est valide. Si ce n'est pas le cas, elle renvoie une réponse d'erreur.

Ensuite, elle récupère le nom du fichier à partir de la liste des noms de fichiers enregistrés.

Elle charge le contenu du fichier pour obtenir des informations pertinentes.

Elle recherche ensuite et supprime les données associées à ce contenu de la base de données vectorielle.

Enfin, elle supprime physiquement le fichier du système de fichiers et renvoie une réponse pour indiquer si la suppression a réussi ou échoué.

```
app.post('/my-react-app/process', async (req, res) => {
  const { userInput } = req.body;
  console.log('Received selected option from client:', selectedOption)
  let answerPrompt = '';
  //Prompt
  const standaloneQuestionPrompt =
    ChatPromptTemplate.fromTemplate(questionAutonomeTemplate);

  if(selectedOption == 'basic'){
    console.log(selectedOption);
    answerPrompt = ChatPromptTemplate.fromTemplate(modeleReponseTempla
  }else if(selectedOption == 'openai'){
    console.log(selectedOption);
    answerPrompt =
    ChatPromptTemplate.fromTemplate(modeleReponseTemplateOpenAi);
  }

  //QstChain :
  const standaloneQuestionChain = standaloneQuestionPrompt
    .pipe(llm)
```



```

        .pipe(new StringOutputParser())
        //RetreivalChain :
const retrieverChain = RunnableSequence.from([
    prevResult => prevResult.standalone_question,
    retriever,
    combinerDocuments
])
//AnswerChain :
const answerChain = answerPrompt
    .pipe(llm)
    .pipe(new StringOutputParser())
//Runnable :
const chainPrincipale = RunnableSequence.from([
    {
        standalone_question: standaloneQuestionChain,
        original_input: new RunnablePassthrough()
    },
    {
        context: retrieverChain,
        question: ({ original_input }) => original_input.question
    },
    answerChain
])
try {
    const reponse = await chainPrincipale.invoke({
        question: userInput
    });

    res.status(200).json({ output: reponse });
} catch (error) {
    console.error('Erreur lors du traitement de la question :', error);
    res.status(500).send('Erreur lors du traitement de la question');
}
});

```

Ce code utilise pour générer une réponse basée sur la question de l'utilisa-

teur il implémente une route POST dans une application Express.js, utilisée pour générer une réponse lorsque l'utilisateur fournit une question. Voici une explication détaillée :

Il définit une fonction `combinerDocuments(docs)` qui prend une liste de documents et retourne une chaîne de caractères combinant le contenu de chaque document avec des sauts de ligne entre eux.

Il instancie un objet `retrieveur` à partir de l'objet `vectorStores`, ce qui permet de récupérer des documents à partir de la base de données vectorielle.

Lorsqu'une requête POST est effectuée à l'URL `'/my-react-app/process'`, elle attend des données JSON contenant une propriété `userInput`.

Selon la valeur de la propriété `selectedOption` reçue dans la requête, il sélectionne un modèle de prompt approprié pour la réponse. Si `selectedOption` est `'basic'`, il utilise `modeleReponseTemplateBasic` ; sinon, s'il est `'openai'`, il utilise `modeleReponseTemplateOpenAi`.

Il construit une chaîne de traitement pour le prompt de la question autonome (`standaloneQuestionPrompt`), le `retrieveur` de documents, et la réponse.

Il exécute une chaîne principale (`chainPrincipale`) qui consiste en trois étapes :

La première étape extrait la question autonome à partir de la chaîne de traitement du prompt de la question autonome. La deuxième étape utilise le `retrieveur` pour récupérer les documents pertinents à partir de la base de données vectorielle. La troisième étape utilise la chaîne de traitement de la réponse pour générer une réponse basée sur la question de l'utilisateur. Il invoque la chaîne principale avec la question de l'utilisateur fournie dans la requête POST.

En cas de succès, il renvoie la réponse générée au format JSON avec un statut 200 (OK).

En cas d'erreur lors du traitement de la question, il affiche un message d'erreur dans la console et renvoie une réponse avec un statut 500 (Internal Server Error).

```
const PORT = 3000;
app.listen(PORT, () => {
  console.log('Server is running on port ${PORT}');
});
```

Ce code est utilisé pour démarrer le serveur Express.js sur un port spécifié. Voici une explication détaillée :

`const PORT = 3000;` : Cette ligne définit une constante `PORT` avec la valeur 3000, qui est le numéro de port sur lequel le serveur va écouter les requêtes entrantes. Vous pouvez choisir un autre numéro de port si nécessaire.

`app.listen(PORT, () => ... );` : Cette ligne utilise la méthode `listen()` de l'objet `app` (instance d'Express.js) pour démarrer le serveur. Elle prend deux arguments : le premier est le numéro de port sur lequel le serveur doit écouter les requêtes entrantes, et le deuxième est une fonction de rappel (callback) qui est exécutée une fois que le serveur est démarré et écoute les connexions entrantes.

Lorsque le serveur démarre avec succès, la fonction de rappel est exécutée, et elle affiche un message dans la console indiquant que le serveur est en cours d'exécution, en incluant le numéro de port sur lequel il écoute.

## 4.7 Étape 3 (conception de la 3eme version de notre application) :

### 4.7.1 front-end :

Homepage :

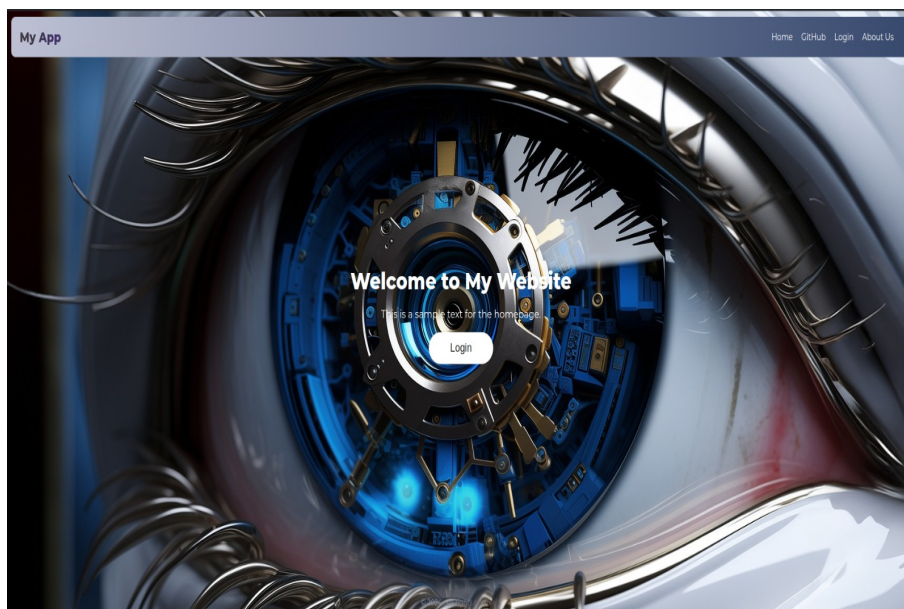


FIGURE 4.7.1 – home-page

La page d'accueil(Home page) contient un menu ainsi qu'un bouton de connexion(Login). Lorsque l'on clique sur ce bouton, la page d'authentifica-

tion apparaît.

formulaire d'authentification :

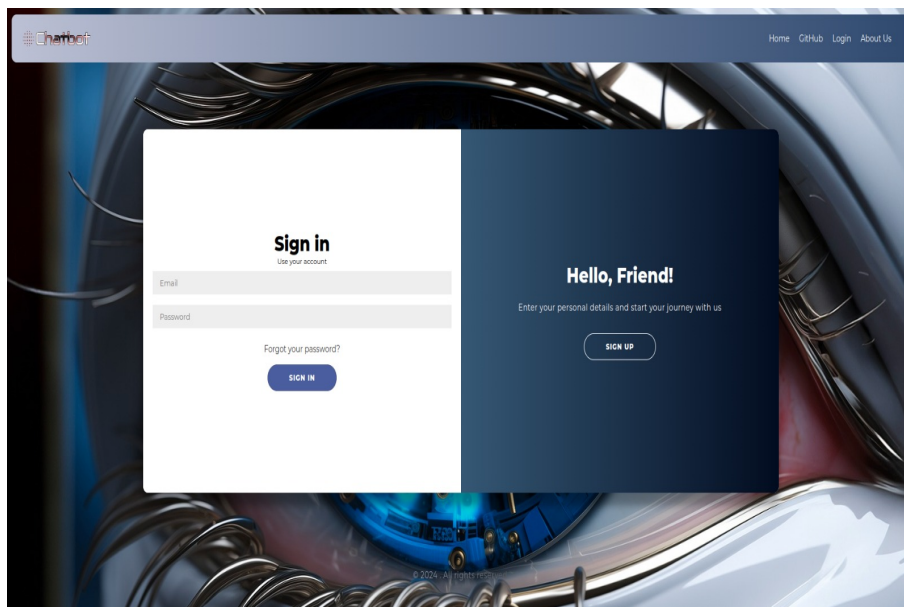
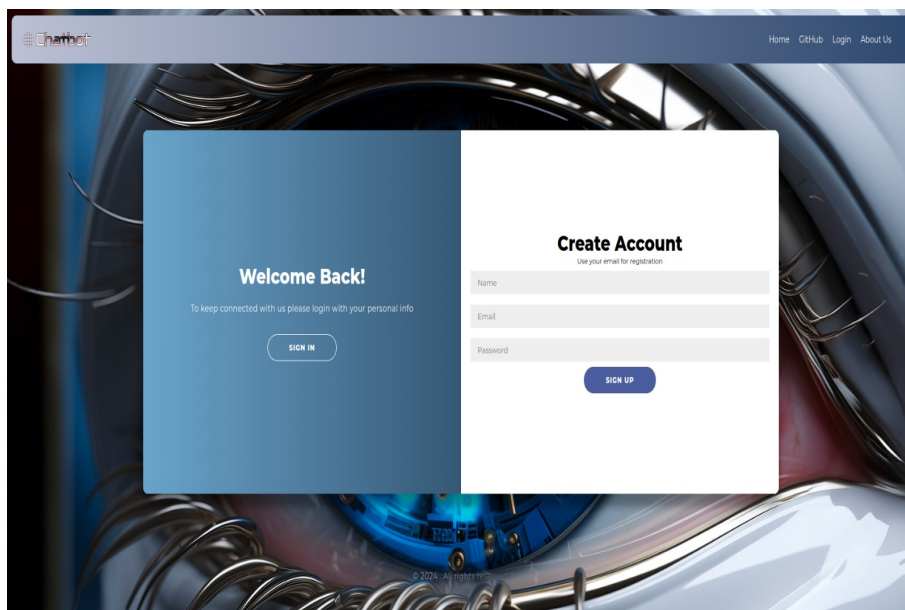


FIGURE 4.7.2 – Sign-in

Cette page contient deux champs, un pour l'email et l'autre pour le mot de passe, ainsi qu'un bouton 'Sign In' pour confirmer la connexion. Si le client n'a pas encore de compte, il peut appuyer sur le bouton 'Sign Up' et le formulaire d'inscription apparaîtra.

## formulaire d'inscription



The image shows a web browser displaying the Charbot website. The header features the Charbot logo on the left and navigation links (Home, GitHub, Login, About Us) on the right. The main content area is split into two panels. The left panel, with a blue background, says "Welcome Back!" and "To keep connected with us please login with your personal info" with a "SIGN IN" button. The right panel, with a white background, is titled "Create Account" and includes the instruction "Use your email for registration". It contains three input fields labeled "Name", "Email", and "Password", followed by a "SIGN UP" button. The background of the entire page is a futuristic, metallic, blue and silver image.

FIGURE 4.7.3 – Sign-up

La page d'inscription contient trois champs : 'Name', 'Email' et 'Password', ainsi qu'un bouton 'SIGN UP' pour confirmer l'inscription.

## Description des Composants de la Page de Profil Utilisateur

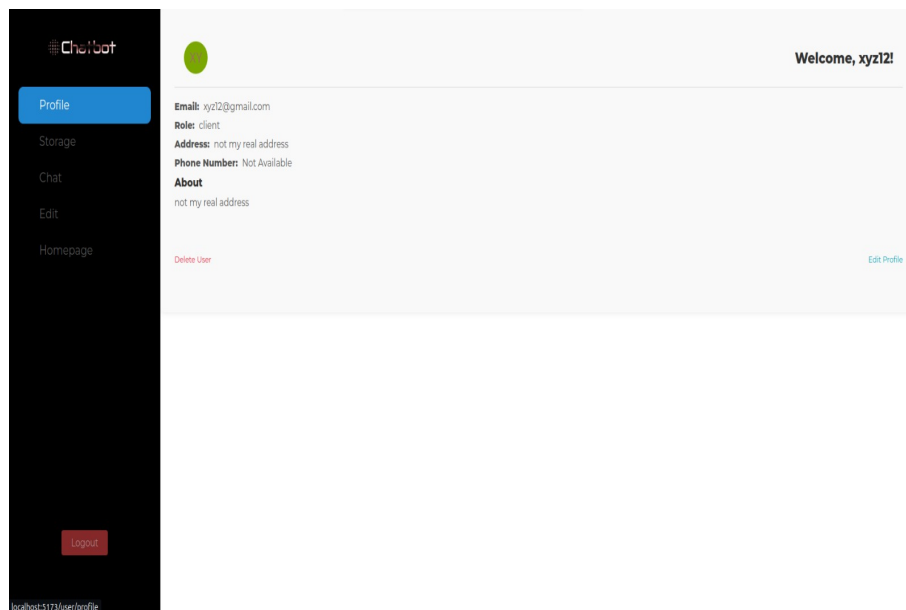


FIGURE 4.7.4 – profileclient

- **Menu latéral gauche :**
  - **Profile :** Affiche les informations du profil de l'utilisateur.
  - **Storage :** une section pour gérer le stockage des fichiers .
  - **Chat :** Accède à la fonctionnalité de chat.
  - **Edit :** Permet de modifier les informations du profil .
  - **Homepage :** Redirige vers la page d'accueil de l'application.
  - **Logout :** Déconnecte l'utilisateur de l'application.
- **Section principale (profil utilisateur) :**
  - **Email :** Affiche l'adresse email de l'utilisateur (xyz12@gmail.com).
  - **Role :** Indique le rôle de l'utilisateur (client).
  - **Address :** Affiche l'adresse de l'utilisateur (not my real address).
  - **Phone Number :** le numéro de téléphone .
  - **About :** Affiche des informations supplémentaires sur l'utilisateur.
- **Options de profil :**
  - **Edit Profile :** Lien permettant de modifier les informations du profil utilisateur.
  - **Delete User :** Bouton pour supprimer le compte utilisateur.

Chacun de ces composants permet à l'utilisateur de naviguer dans l'application, de consulter et de modifier ses informations personnelles, ainsi que

de gérer ses préférences et paramètres de compte.

## Description des Composants de la Page de Stockage Utilisateur

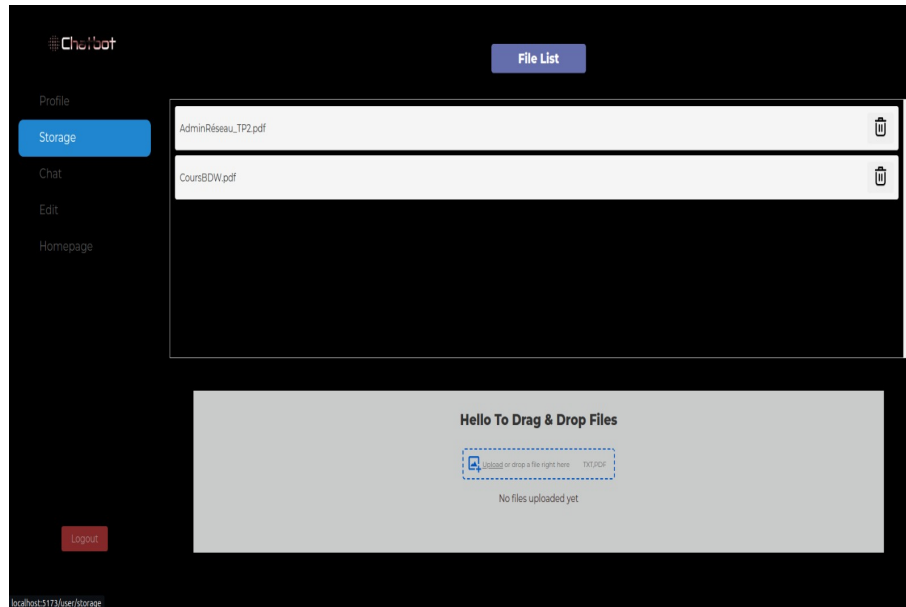


FIGURE 4.7.5 – storage

- **Menu latéral gauche :**
  - **Profile :** Affiche les informations du profil de l'utilisateur.
  - **Storage :** Accède à la section de gestion du stockage des fichiers.
  - **Chat :** Accède à la fonctionnalité de chat.
  - **Edit :** Permet de modifier les informations du profil ou d'autres paramètres.
  - **Homepage :** Redirige vers la page d'accueil de l'application.
  - **Logout :** Déconnecte l'utilisateur de l'application.
- **Section principale (gestion des fichiers) :**
  - **File List :** Affiche la liste des fichiers stockés par l'utilisateur.
    - **AdminRéseau\_TP2.pdf :** Nom d'un fichier PDF stocké.
    - **CoursBDW.pdf :** Nom d'un autre fichier PDF stocké.
    - **Icône de suppression :** Icône de poubelle permettant de supprimer le fichier correspondant.
- **Zone de téléchargement :**
  - **Hello To Drag & Drop Files :** Indication pour glisser-déposer des fichiers dans la zone de téléchargement.

- **Upload or drop a file right here** : Zone où l'utilisateur peut uploader ou déposer un fichier (formats acceptés : TXT, PDF).
- **No files uploaded yet** : Message indiquant qu'aucun fichier n'a encore été uploadé.

Chacun de ces composants permet à l'utilisateur de gérer ses fichiers stockés, d'uploader de nouveaux fichiers .

### Description des Composants de la Page de Chat Utilisateur

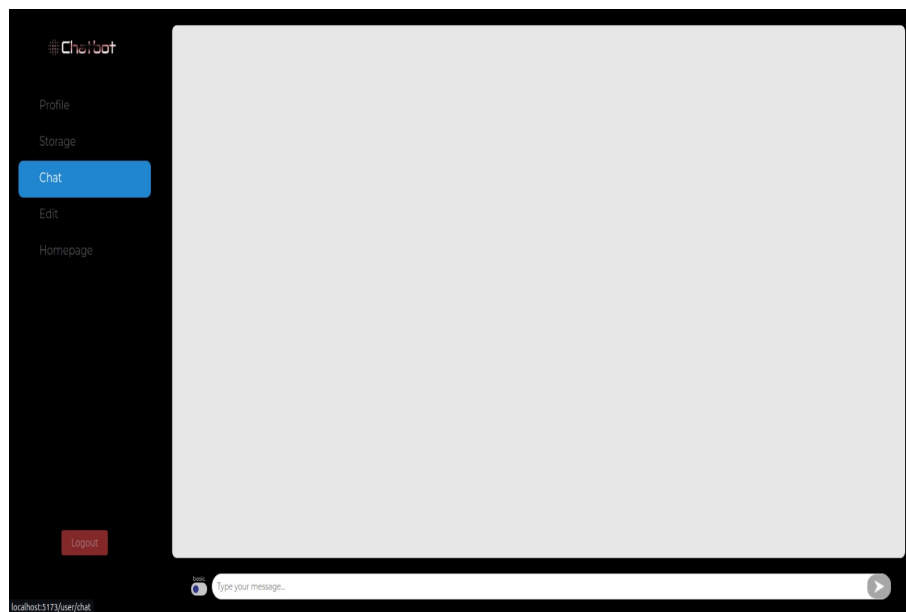


FIGURE 4.7.6 – chat

- **Menu latéral gauche** :
  - **Profile** : Affiche les informations du profil de l'utilisateur.
  - **Storage** : Accède à la section de gestion du stockage des fichiers.
  - **Chat** : Accède à la fonctionnalité de chat.
  - **Edit** : Permet de modifier les informations du profil ou d'autres paramètres.
  - **Homepage** : Redirige vers la page d'accueil de l'application.
  - **Logout** : Déconnecte l'utilisateur de l'application.
- **Section principale (zone de chat)** :
  - **Zone de chat** : Grand espace gris où les messages de chat sont affichés.
  - **Barre de message** : Barre en bas où l'utilisateur peut taper son message.



- **Bouton d'envoi** : Bouton en forme de flèche pour envoyer le message tapé.
- **Bouton select** : bouton pour sélectionner l'option de réponse

## Description des Composants de la Page de Mise à Jour de l'Utilisateur

The screenshot shows a web application interface for updating a user profile. On the left is a dark sidebar with a 'Chatbot' icon and navigation links: Profile, Storage, Chat, Edit (highlighted in blue), and Homepage. At the bottom of the sidebar is a red 'Logout' button. The main content area is titled 'Users' and 'Update User'. It contains several input fields: Name (with placeholder 'xyz'), Email (with placeholder 'xyz@gmail.com'), Password (with placeholder 'xxxxx'), Confirm Password (with placeholder 'xxxxx'), About (with placeholder 'not my real address'), Address (with placeholder 'not my real address'), and Phone Number (with placeholder 'Phone Number'). A blue 'Update' button is located at the bottom of the form.

FIGURE 4.7.7 – update-user

- **Titre de la page** :
  - **Users** : Indique que la page concerne les clients.
- **Sous-titre** :
  - **Update User** : Indique que cette section est pour la mise à jour des informations de client.
- **Formulaire de mise à jour** :
  - **Name** : Champ de saisie pour le nom de l'utilisateur.
  - **Email** : Champ de saisie pour l'adresse email de l'utilisateur.
  - **Password** : Champ de saisie pour le mot de passe de l'utilisateur.
  - **Confirm Password** : Champ de saisie pour confirmer le mot de passe de l'utilisateur.
  - **About** : Champ de saisie pour des informations supplémentaires sur l'utilisateur.
  - **Address** : Champ de saisie pour l'adresse de l'utilisateur.
  - **Phone Number** : Champ de saisie pour le numéro de téléphone de l'utilisateur.

— **Bouton de mise à jour :**

- **Update :** Bouton pour confirmer et enregistrer les modifications apportées aux informations de l'utilisateur.

Chacun de ces composants permet à l'utilisateur de mettre à jour ses informations personnelles de manière structurée et organisée.

#### 4.7.2 back-end

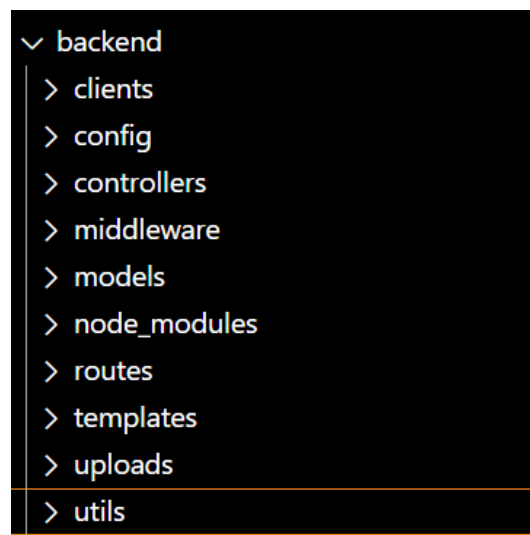


FIGURE 4.7.8 – structure de fichier

Comme mentionné précédemment dans la troisième version, les clients peuvent créer un compte et se connecter, chaque client ayant sa propre base de données vectorielle. Ils peuvent télécharger et supprimer des fichiers, poser des questions au chatbot et choisir l'option de réponse. Pour cela, nous avons travaillé sur plusieurs fonctions dans le back-end qui nous permettent de réaliser ces actions.

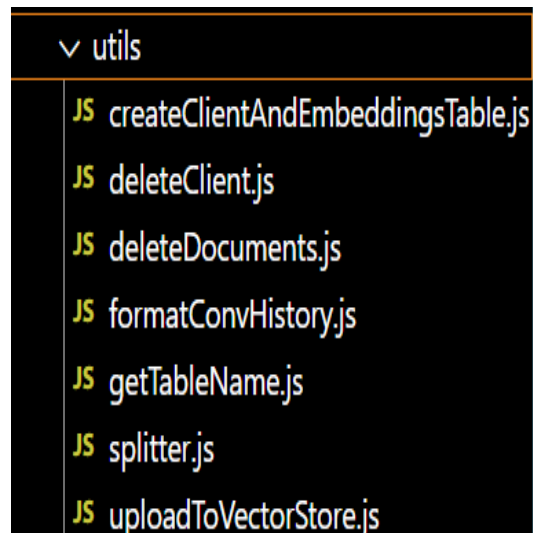


FIGURE 4.7.9 – user-UI

createClientAndEmbeddingsTable.js :

```
import { client } from '../config/openai.js';
import path from 'path';
import fs from 'fs';

export default async function createClientAndEmbeddingsTable(username, e
  try {
    // Insertion des informations du client dans la table client
    console.log('Inserting client information...');
    await client.from('client').insert([{ username, email, password }]);
    console.log('Client information inserted.');
```

```
    // Récupération de l'ID du client
    console.log('Fetching client ID...');
    const { data: clientData, error: fetchError } = await client
      .from('client')
      .select('id')
      .eq('username', username)
      .eq('email', email)
      .eq('password', password)
      .single();

    if (fetchError) {
```

```

        throw new Error('Error fetching client ID: ${fetchError.message}')
    }
    console.log('Client ID fetched:', clientData.id);

    const clientId = clientData.id;

    // Création dynamique d'une table pour les embeddings pour le nouveau client
    const embeddingsTableName = `embeddings${clientId}`;
    const createTableQuery = `
        CREATE TABLE IF NOT EXISTS ${embeddingsTableName} (
            id bigserial PRIMARY KEY,
            content text,
            metadata jsonb,
            embedding vector(1536)
        );
    `;

    console.log('Creating embeddings table "${embeddingsTableName}"...');
    await client.rpc('execute_sql', { sql: createTableQuery });
    console.log('Embeddings table "${embeddingsTableName}" created successfully');

    } catch (error) {
        console.error('Error in createClientAndEmbeddingsTable: ${error.message}');
        throw error;
    }
}

```

Ce code permet d'insérer un nouveau client dans la base de données (tableau client) et de créer son propre tableau pour stocker les embeddings vectoriels. Chaque client a un tableau unique pour stocker les embeddings vectoriels.

```

export default async function createClientAndEmbeddingsTable(username, email, password) {
    try {
        // Insertion des informations du client dans la table client
        console.log('Inserting client information...');
        await client.from('client').insert([{ username, email, password }]);
        console.log('Client information inserted.');
    }
}

```

Cette fonction prend trois paramètres : username, email et password. Elle insère ces trois paramètres dans la table client (id, username, email, password, tablename). L'id est une clé primaire auto-incrémentée. Le nom de la table (tablename) sera inséré après.

```
// Récupération de l'ID du client
console.log('Fetching client ID...');
const { data: clientData, error: fetchError } = await client
  .from('client')
  .select('id')
  .eq('username', username)
  .eq('email', email)
  .eq('password', password)
  .single();
```

Après ça, on cherche l'ID du client nouvellement inséré pour l'utiliser ensuite dans la création du tableau d'embeddings.

```
const clientId = clientData.id;

// Création dynamique d'une table pour les embeddings pour le nouveau client
const embeddingsTableName = `embeddings${clientId}`;
const createTableQuery = `
  CREATE TABLE IF NOT EXISTS ${embeddingsTableName} (
    id bigserial PRIMARY KEY,
    content text,
    metadata jsonb,
    embedding vector(1536)
  );
`;

console.log(`Creating embeddings table "${embeddingsTableName}"...`);
await client.rpc('execute_sql', { sql: createTableQuery });
console.log(`Embeddings table "${embeddingsTableName}" created successfully`);

} catch (error) {
  console.error(`Error in createClientAndEmbeddingsTable: ${error.message}`);
}
```

```

        throw error;
    }
}

```

On prend l’ID du client et on crée son propre tableau d’embeddings. Le nom du tableau est une concaténation du mot ‘embeddings’ et de l’ID du client. On insère aussi le nom du tableau d’embeddings dans la colonne ‘tablename’ de la table client.

getTableName.js :

```
// filename: utils/getTableName.js
```

```
import { Embeddings } from '@langchain/core/embeddings';
import { client } from '../config/openai.js';
```

```
export async function getTableName(email) {
  try {
    const { data: clientData } = await client
      .from('client')
      .select('id')
      .eq('email',email);

    if (!clientData || clientData.length === 0) {
      console.error('No client found with the provided email and p
      return null;
    }
    const clientId = clientData[0].id;
    console.log(clientId)
    return String('embeddings'+clientId);
  } catch (err) {
    console.error('Error fetching tablename:', err.message);
    return null;
  }
}
```

Cette fonction nous permet de chercher le nom du tableau d’embeddings d’un client à partir de son email, car nous avons besoin de ce nom pour effectuer

des traitements ultérieurs tels que l'upload de fichier, la suppression de fichier et également la suppression du client.

uploadToVectorStore.js :

```
import { getTableName } from "../getTableName.js";
import { SupabaseVectorStore } from "@langchain/community/vectorstores/supabase";
import { client } from "../config/openai.js";
import { OpenAIEmbeddings } from "@langchain/openai";

async function uploadToVectorStore(email, output) {
  try {
    console.log(email);
    const tablename = await getTableName(String(email));

    if (!tablename) {
      console.error('Error fetching tablename.');
```

return;

```
    }

    await SupabaseVectorStore.fromDocuments(output, new OpenAIEmbeddings(client, {
      tableName: tablename
    }));

    console.log('Documents uploaded to vector store successfully.');
```

} catch (err) {

```
  console.error('Unexpected error in uploadToVectorStore:', err.message);
  console.error(err.stack);
}
```

}

export default uploadToVectorStore;

importation : Importe la fonction getTableName depuis le fichier getTableName.js.

Importe SupabaseVectorStore depuis le package @langchain/community/vectorstores/supabase.

Importe l'objet client depuis le fichier openai.js situé dans le répertoire

config.

Importe OpenAIEmbeddings depuis le package @langchain/openai.

```
async function uploadToVectorStore(email, output) {
  try {
    console.log(email);
    const tablename = await getTableName(String(email));

    if (!tablename) {
      console.error('Error fetching tablename.');
```

return;

```
    }

    await SupabaseVectorStore.fromDocuments(output, new OpenAIEmbeddings(
      client,
      tableName: tablename
    ));

    console.log('Documents uploaded to vector store successfully.');
```

} catch (err) {

```
  console.error('Unexpected error in uploadToVectorStore:', err.message);
  console.error(err.stack);
}
```

}

Déclare la fonction `uploadToVectorStore` comme une fonction asynchrone. Dans le corps de la fonction, un `console.log` affiche l'email passé en paramètre.

Utilise la fonction `getTableName` pour obtenir le nom du tableau d'embeddings associé à l'email. Cette fonction est appelée avec l'email converti en chaîne de caractères (`String(email)`), puis le résultat est stocké dans la variable `tablename`.

Si aucun nom de tableau n'est obtenu (c'est-à-dire si `!tablename` est évalué



à true), un message d'erreur est affiché dans la console et la fonction se termine avec un retour anticipé (return;).

Si un nom de tableau est obtenu avec succès, la méthode fromDocuments de SupabaseVectorStore est appelée pour charger les données output dans le store de vecteurs Supabase. Cette méthode prend plusieurs paramètres : les données à charger, un objet OpenAIEmbeddings, l'objet client et le nom du tableau (tableName).

Si le chargement est effectué avec succès, un message de succès est affiché dans la console.

Si une erreur se produit lors de l'exécution du code dans le bloc try, elle est capturée et gérée dans le bloc catch, où un message d'erreur est affiché dans la console, ainsi que la pile d'appels de l'erreur (err.stack).

deleteDocuments.js :

```
import { client } from '../config/openai.js';
import { getTableName } from './getTableName.js';

export async function deleteDocuments(output, email) {
  const tablename = await getTableName(email);
  for (let i = 0; i < output.length; i++) {
    try {
      const { data: queryData, error: queryError } = await client
        .from(tablename)
        .select('id')
        .eq('content', output[i].pageContent);

      if (queryError) {
        // Handle the error
        console.error("Error querying database:", queryError.message);
        continue;
      }

      // Check if queryData is not empty
      if (queryData && queryData.length > 0) {
        // Store the id in a variable
        const id = queryData[0].id;
```

```

        const { data: deleteData, error: deleteError } = await c
            .from(tablename)
            .delete()
            .eq('id', id);

        if (deleteError) {
            // Handle the deletion error
            console.error("Error deleting row:", deleteError.mes
        } else {
            console.log("Row deleted successfully.");
        }
    } else {
        console.log("No matching id found.");
    }
} catch (err) {
    console.error("Unexpected error:", err.message);
}
}
}

```

Cette fonction nous permet de supprimer un document déjà téléversé dans la base de données par un utilisateur. Elle prend comme arguments l'email de l'utilisateur et un 'output'. L'output est le contenu que nous voulons supprimer.

deleteClient.js :

```

import {getTableName} from './getTableName.js'
import { client } from '../config/openai.js';

async function deleteClient(email) {
    try {
        const tableName = await getTableName(email);
        console.log(tableName);

        if (!tableName || tableName == null) {

```

```

        console.error('Table name not found, cannot proceed with del
        return;
    }

    const { data: clientData, error: clientError } = await client
        .from('client')
        .select('id')
        .eq('email', email);

    if (clientError) throw clientError;

    console.log("Client ID data retrieved:", clientData);

    if (!clientData || clientData.length === 0) {
        console.error('No client found with the provided email.');
```

return;

```

    }

    const clientId = clientData[0].id;
    console.log(clientId)

    const deleteClientSql = `DELETE FROM client WHERE id = ${clientId}`;
    const deleteTableSql = `DROP TABLE ${tableName}`;

    console.log("Executing SQL:", deleteClientSql);
    const { error: deleteClientError } = await client.rpc('execute_sql');
    if (deleteClientError) throw deleteClientError;
    console.log("Client deleted");

    console.log("Executing SQL:", deleteTableSql);
    const { error: deleteTableError } = await client.rpc('execute_sql');
    if (deleteTableError) throw deleteTableError;
    console.log("Table deleted");
} catch (err) {
    console.error('Error during deletion:', err.message);
}
}

```

```
export default deleteClient;
```

La fonction `deleteClient` sert à supprimer un client de la base de données. Elle prend comme arguments l'email du client, puis elle cherche l'ID du client et le nom du tableau des embeddings (`tableName`). Elle supprime la ligne correspondante à l'ID avec la requête SQL suivante :

```
const deleteClientSql = `DELETE FROM client WHERE id = ${clientId}`;
```

Ensuite, elle supprime également la table d'embedding de ce client avec la requête suivante :

```
const deleteTableSql = `DROP TABLE ${tableName}`;
```

splitter.js :

```
import { RecursiveCharacterTextSplitter } from "langchain/text_splitter"
```

```
export const splitter = new RecursiveCharacterTextSplitter({  
  chunkSize: 350,  
  chunkOverlap: 50  
});
```

Dans ce fichier, nous avons défini un text splitter qui va nous permettre de diviser les documents et les fichiers en petits morceaux de taille égale.

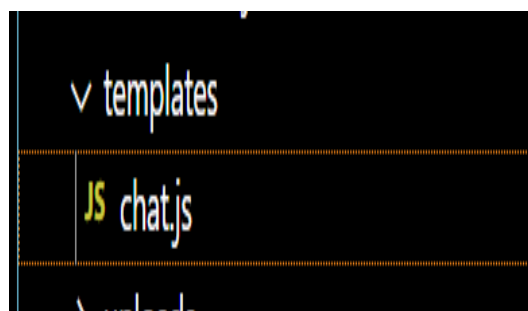


FIGURE 4.7.10 – prompt-template

Dans ce fichier, nous avons défini des "templates de prompt". Un prompt dans Langchain est une séquence d'instructions que nous donnons aux modèles de langage pour guider leur réponse (de quelle manière ils doivent

répondre aux questions de l'utilisateur et dans quelle langue) et aussi pour les aider à comprendre le contexte.

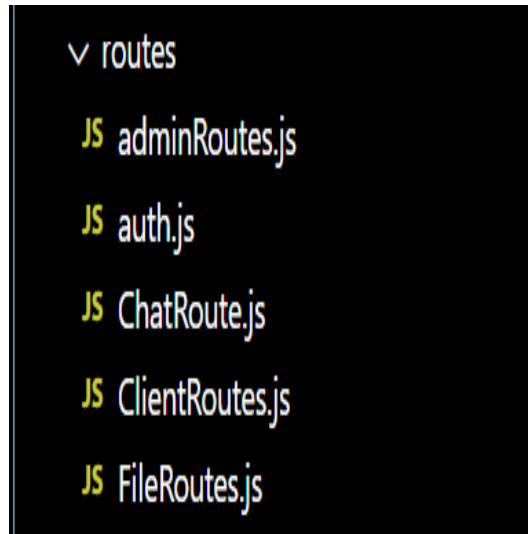


FIGURE 4.7.11 – routes

Voici une explication de chaque fichier de route basé sur le code fourni. Chaque fichier configure des routes spécifiques pour différentes fonctionnalités de l'application Express.js.

```
[caption=adminRoutes.js] import express from 'express'; import updateUser, deleteUser from '../controllers/Users.js';
const router = express.Router();
router.put('/user/update', updateUser); router.delete('/user/delete', deleteUser);
export default router;
```

Ce fichier gère les routes pour mettre à jour et supprimer des utilisateurs. Il expose les points de terminaison suivants :

- PUT `/user/update` pour mettre à jour les informations d'un utilisateur.
- DELETE `/user/delete` pour supprimer un utilisateur.

```
[caption=auth.js] import express from 'express'; import register, login from '../controllers/authController.js'; import verifyToken from '../middleware/authMiddleware.js';
const router = express.Router();
router.post('/register', register); router.post('/login', login);
export default router;
```

Ce fichier gère les routes liées à l'authentification. Il expose les points de terminaison suivants :

- `POST /register` pour enregistrer un nouvel utilisateur.
- `POST /login` pour connecter un utilisateur existant.

```
[caption=ChatRoute.js] import express from 'express'; import processInput from '../controllers/Chat.js';
const router = express.Router();
router.post('/process', processInput);
export default router;
```

Ce fichier gère les routes liées au chat. Il expose le point de terminaison suivant :

- `POST /process` pour traiter l'entrée de l'utilisateur dans le chat.

```
[caption=FileRoutes.js] import express from 'express'; import multer from 'multer'; import uploadFile, getFiles, deleteFile from '../controllers/Files.js';
```

```
const router = express.Router(); const upload = multer( dest : 'uploads/' );
```

```
router.post('/upload', upload.single('file'), uploadFile); router.get('/get-files', getFiles); router.delete('/delete/:index', deleteFile);
export default router;
```

Ce fichier gère les routes liées aux fichiers. Il expose les points de terminaison suivants :

- `POST /upload` pour télécharger un fichier. Utilise `multer` pour gérer les téléchargements.
- `GET /get-files` pour récupérer une liste de fichiers.
- `DELETE /delete/:index` pour supprimer un fichier spécifique en fonction de son index.
- `adminRoutes.js` : Routes pour la mise à jour et la suppression des utilisateurs.
- `auth.js` : Routes pour l'enregistrement et la connexion des utilisateurs.
- `ChatRoute.js` : Route pour traiter les entrées de chat.
- `FileRoutes.js` : Routes pour télécharger, récupérer et supprimer des fichiers.

Chacun de ces fichiers est responsable de gérer une partie spécifique de la fonctionnalité de l'application, et ils sont tous intégrés dans le serveur principal via le `app.use()` d'Express.

# Conclusion

Ce dernier chapitre concerne la conclusion générale de votre projet.

# Bibliographie

- [1] Andrew Tanenbaum (Université libre d'Amsterdam), « Systèmes d'exploitation », 3ème édition, Nouveaux Horizons, 2008.
- [2] [http ://www.linux.org/](http://www.linux.org/)