Documentation - Système de Chatbot Intelligent

Créé par	rim belabadia
Heure de création	@26 mars 2025 10:18
Étiquettes	

Système de Chatbot Intelligent :

I. Architecture Globale du Chatbot :

L'architecture globale de votre chatbot RH peut être définie en plusieurs couches principales. Voici les composants clés :

- **Frontend (Next.js)** : L'interface utilisateur à travers laquelle les utilisateurs interagissent avec le chatbot.
- **Backend (Flask/FastAPI)** : Le serveur qui gère la logique du chatbot, les API pour exécuter les tâches et interagir avec la base de données.
- **Base de données (PostgreSQL)** : La base de données pour stocker les utilisateurs, les tâches, les instructions, etc.
- **Service de NLP (spaCy)** : Le moteur NLP qui permet au chatbot de comprendre et de répondre aux questions des utilisateurs.
- **APIs externes** : Les APIs que le chatbot pourrait appeler pour exécuter certaines tâches (par exemple, pour demander une fiche de paie).

I.I BACKEND :

1. Documentation détaillée du backend :

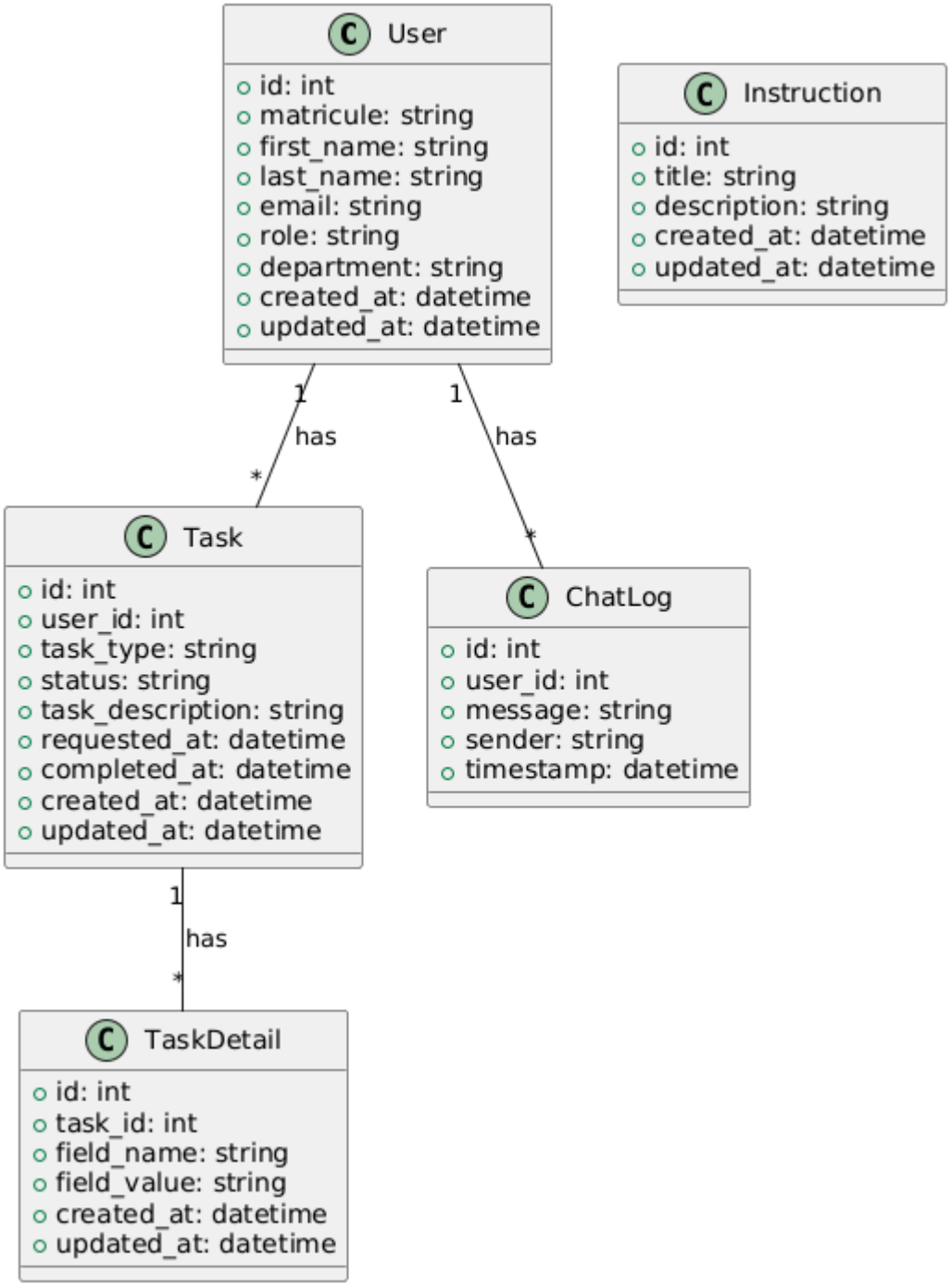
```
backend/
├── app/
│   ├── middlewares/        # Middlewares pour le traitement des requêtes
```

```
│   ├── models/          # Modèles de données et ORM
│   ├── routes/          # Définition des routes API
│   ├── services/        # Logique métier
│   └── tests/           # Tests du backend
└── database.py
└──config.py
└── main.py
```

1.1. Documentation des Modèles (models)

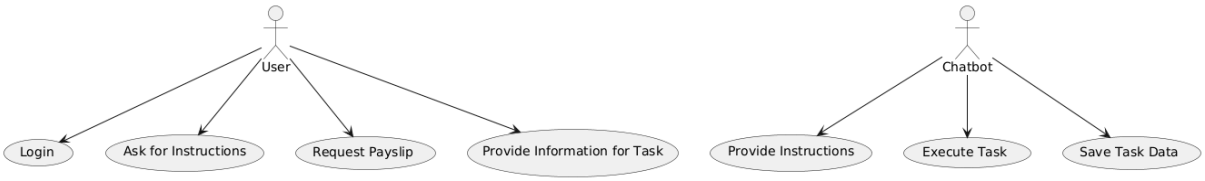
1.1.1 Diagramme de classes (UML) :

Le diagramme de classes décrit les objets du système et leurs relations.



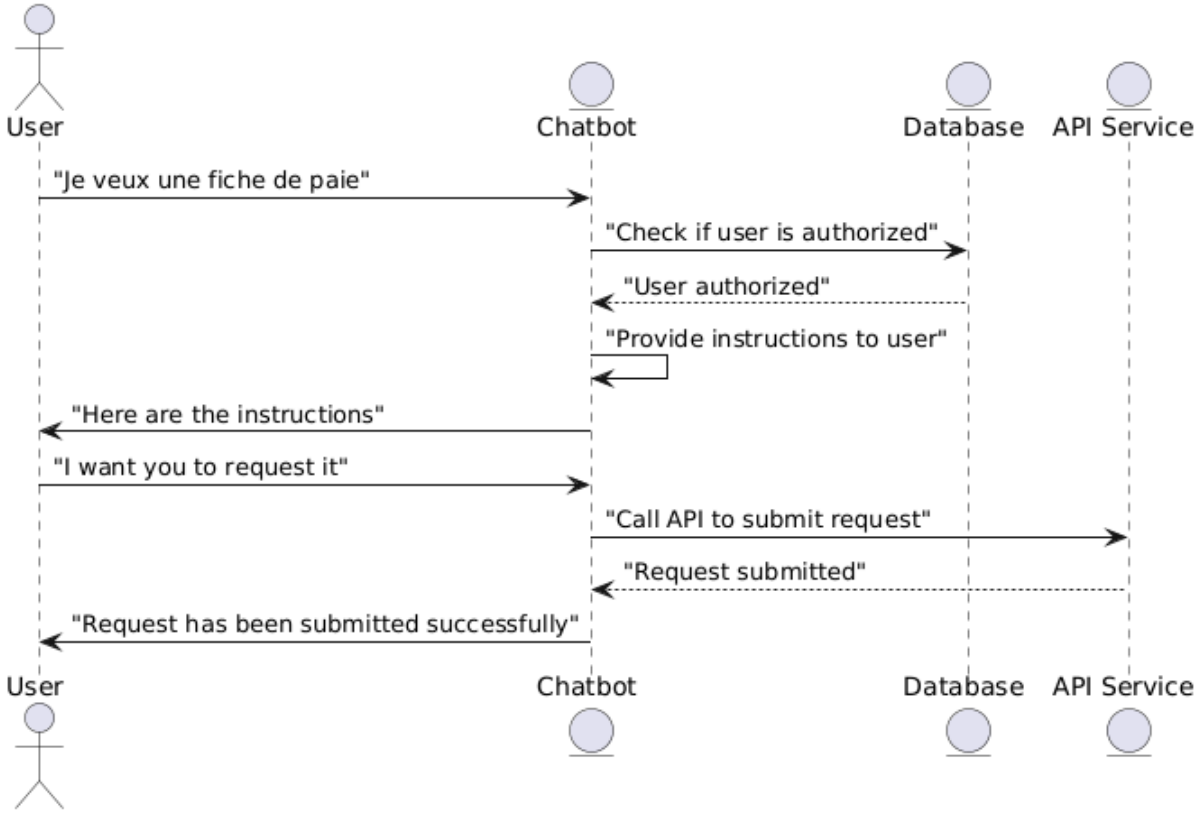
1.1.2 Diagramme de cas d'utilisation (UML)

Le diagramme de cas d'utilisation permet de visualiser les actions possibles pour l'utilisateur et le chatbot dans le système.



1.1.3 Diagramme de séquence (UML)

Le diagramme de séquence montre l'interaction dynamique entre les différents composants du système. Voici un exemple pour le cas où l'utilisateur demande une fiche de paie au chatbot :



1.1.4 **Technologies Backend**

1. **Flask/FastAPI** : Le framework Python pour construire le backend.
   - **Flask** : Framework léger et flexible pour les applications web.
   - **FastAPI** : Framework moderne pour les API avec de bonnes performances et une génération automatique de documentation Swagger.
2. **PostgreSQL** : Base de données relationnelle pour stocker toutes les données liées aux utilisateurs, aux tâches, aux instructions et aux logs du chatbot.
3. **spaCy** : Librairie Python pour le traitement du langage naturel (NLP). Elle permet de comprendre les demandes des utilisateurs et de générer des réponses appropriées.

1.1.5 Models utilisés

User.py :

```
from sqlalchemy import Column, Integer, String
from app.database import Base
from sqlalchemy.orm import relationship
class User(Base):
    __tablename__ = 'users'

    id = Column(Integer, primary_key=True, index=True)
    matricule = Column(String, unique=True, index=True)
    first_name = Column(String)
    last_name = Column(String)
    email = Column(String, unique=True)
    role = Column(String)
    department = Column(String)
    created_at = Column(String)
    updated_at = Column(String)
# Define the relationship with ChatLog
    chat_logs = relationship('ChatLog', back_populates='user', cascade='all, de
```

Instruction.py :

```
from app.database import Base  # Correction de l'importation
from sqlalchemy import Column, Integer, String, Text

class Instruction(Base):
    __tablename__ = 'instructions'

    id = Column(Integer, primary_key=True, index=True)
    title = Column(String, nullable=False)
    description = Column(Text, nullable=False)
```

```
    created_at = Column(String)
    updated_at = Column(String)
```

Task.py :

```
from sqlalchemy import Column, Integer, String, ForeignKey
from app.database import Base

class Task(Base):
    __tablename__ = 'tasks'

    id = Column(Integer, primary_key=True, index=True)
    user_id = Column(Integer, ForeignKey('users.id'))
    task_type = Column(String)
    status = Column(String)
    task_description = Column(String)
    requested_at = Column(String)
    completed_at = Column(String)
    created_at = Column(String)
    updated_at = Column(String)
```

Chat_Logs.py

```
from sqlalchemy import Column, Integer, String, ForeignKey
from sqlalchemy.orm import relationship
from app.database import Base

class ChatLog(Base):
    __tablename__ = 'chat_logs'

    id = Column(Integer, primary_key=True, index=True)
    user_id = Column(Integer, ForeignKey('users.id'))
    message = Column(String)
    sender = Column(String)
```

```
    timestamp = Column(String)

    user = relationship("User", back_populates="chat_logs")
```

Base.py

Le fichier `base.py` définit une classe de base `BaseModelWithTimestamp` qui ajoute automatiquement des champs `created_at` et `updated_at` à tous les modèles Pydantic qui en héritent. Cela permet de standardiser la gestion des dates de création et de modification pour toutes les entités de l'application, évitant ainsi de répéter ces champs dans chaque modèle et garantissant une cohérence dans la structure des données.

```
# backend/app/models/base.py

from pydantic import BaseModel
from datetime import datetime

class BaseModelWithTimestamp(BaseModel):
    created_at: datetime
    updated_at: datetime
```

1.1.6 Configuration de la base de données

1. Créer un nouvel utilisateur : `chatbot_user` avec le mot de passe `chatbot_password`.
    a. Connexion en tant qu'utilisateur `postgres` en utilisant les informations d'identification que vous avez spécifiées.

```
CREATE USER chatbot_user WITH PASSWORD 'chatbot_password';
```

    b. Accordre à cet utilisateur les privilèges nécessaires :

```
ALTER USER chatbot_user WITH SUPERUSER;
```

2. Créer une Base de Données
    a. Créer une nouvelle base de données pour le chatbot :

```
CREATE DATABASE chatbot_db;
```

b. Connexion à la base de données `chatbot_db` en utilisant le nouvel utilisateur créé (`chatbot_user`).

```
\c chatbot_db chatbot_user
```

3. Configurer la connexion entre la base de données et le backend

a. centralisation des paramètres **de configuration sensibles**

Il centralise tous les paramètres de configuration sensibles (comme les identifiants de base de données) en les récupérant depuis des variables d'environnement, ce qui permet :

1. **De protéger les informations sensibles** (mots de passe, etc.) en évitant de les coder en dur
2. **D'adapter facilement la configuration** selon l'environnement (développement, test, production)
3. **De fournir des valeurs par défaut** sécurisées si les variables d'environnement ne sont pas définies

Concrètement, il charge :

- Les identifiants de connexion à PostgreSQL (`DB_USER`, `DB_PASSWORD`)
- L'emplacement de la base de données (`DB_HOST`, `DB_PORT`)
- Le nom de la base (`DB_NAME`)

```
# backend/app/config.py

import os
from dotenv import load_dotenv

# Charger les variables d'environnement depuis un fichier .env
load_dotenv()

class Config:
    DB_USER = os.getenv("DB_USER", "chatbot_user")
    DB_PASSWORD = os.getenv("DB_PASSWORD", "chatbot_password")
```

```
DB_HOST = os.getenv("DB_HOST", "localhost")
DB_PORT = os.getenv("DB_PORT", "5432")
DB_NAME = os.getenv("DB_NAME", "chatbot_db")
```

Database.py

```
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker
from sqlalchemy.orm import Session

SQLALCHEMY_DATABASE_URL = "postgresql://chatbot_user:chatbot_passwo

engine = create_engine(SQLALCHEMY_DATABASE_URL)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engi

Base = declarative_base()

# Fonction pour récupérer la session de la base de données
def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```

1.1.7 Services utilisés

Les services séparent la **logique métier** du reste de l'application (routes, modèles), ce qui améliore la **maintenabilité** et la **réutilisabilité** du code. Par exemple, l'authentification ou la gestion des messages du chatbot peuvent être centralisés dans des services et appelés depuis plusieurs endpoints API sans duplication.

1. `auth_service.py`

Ce service gère **l'authentification** en vérifiant si un utilisateur existe via son matricule. La fonction `authenticate_user` interroge la base de données et retourne

l'utilisateur si trouvé, sinon `None`. Cela permet de sécuriser les routes API sans répéter la logique de vérification.

```
# backend/app/services/auth_service.py

from app.database import Database
from app.models.user import User

async def authenticate_user(matricule: str, db: Database) → User:
    query = "SELECT * FROM users WHERE matricule = $1"
    result = await db.fetch(query, matricule)
    if result:
        return User(**dict(result[0]))
    return None
```

2. `chatbot.py`

Ce service traite **les interactions avec le chatbot** en 4 étapes :

- `get_user_by_matricule` : Vérifie l'identité de l'utilisateur.
- `get_instructions_by_keywords` : Recherche des instructions en base en fonction des mots-clés du message.
- **Journalisation** : Enregistre chaque message dans les logs (`ChatLog`).
- **Réponse** : Renvoie les instructions trouvées ou un message par défaut.

```
from sqlalchemy.orm import Session
from models.chat_logs import ChatLog
from models.instruction import Instruction
from models.user import User
from database import get_db  # importez la fonction get_db pour obtenir la se

# Fonction pour obtenir l'utilisateur par son matricule
def get_user_by_matricule(db: Session, matricule: str):
    return db.query(User).filter(User.matricule == matricule).first()

# Fonction pour obtenir toutes les instructions
def get_instructions(db: Session):
    return db.query(Instruction).all()
```

```
# Fonction pour obtenir les instructions par mots-clés dans le titre
def get_instructions_by_keywords(db: Session, keywords: str):
    # Recherche insensible à la casse pour des mots-clés dans le titre
    return db.query(Instruction).filter(Instruction.title.ilike(f"%{keywords}%")).a

# Fonction qui gère les messages et interagit avec la base de données
def handle_message(db: Session, user_matricule: str, message: str):
    # Recherche de l'utilisateur par son matricule
    user = get_user_by_matricule(db, user_matricule)
    if not user:
        return "Utilisateur non trouvé."

    # Log de l'interaction avec l'utilisateur
    chat_log = ChatLog(user_id=user.id, message=message, sender="user", tim
    db.add(chat_log)
    db.commit()

    # Recherche d'instructions en utilisant des mots-clés extraits du message
    keywords = message.lower()  # Convertir le message en minuscules
    instructions = get_instructions_by_keywords(db, keywords)

    if instructions:
        response = "Voici les instructions trouvées :\n"
        for instruction in instructions:
            response += f"- {instruction.title}: {instruction.description}\n"
        return response

    # Réponse par défaut si aucune instruction n'est trouvée
    return "Aucune instruction trouvée pour cette demande. Essayez une autre
```

1.1.7 Routes API utilisés

1. `auth.py`

Rôle : Gère l'authentification des utilisateurs via leur matricule.

- **Endpoint** `POST /login`

```
@router.post("/login")
def login(matricule: str, db: Session = Depends(get_db)):
    user = db.query(User).filter(User.matricule == matricule).first()
    if not user:
        raise HTTPException(status_code=404, detail="Utilisateur non tro
uvé")
    return {"message": f"Bienvenue {user.first_name} {user.last_name}"}
```

- **Fonctionnalité** :
  - Vérifie si le matricule existe en base de données.
  - Retourne un message de bienvenue personnalisé **ou une erreur 404** si l'utilisateur n'existe pas.
- **Workflow** :
  1. Requête SQLAlchemy pour chercher l'utilisateur.
  2. Retourne une réponse JSON avec le nom/prénom.

2. `chat.py`

Rôle : Gère les interactions avec le chatbot.

- **Endpoint** `POST /chat`

```
@router.post("/chat")
async def chat(matricule: str, message: str, db: Database = Depends()):
    user_data = await db.fetch("SELECT * FROM users WHERE matricule
= $1", matricule)
    if not user_data:
        raise HTTPException(status_code=401, detail="Utilisateur non trou
vé")
    user = User(**dict(user_data[0]))
    response = await generate_chat_response(user, message)
    return {"response": response}
```

- **Fonctionnalité** :
  - Récupère l'utilisateur **de manière asynchrone** via son matricule.
  - Utilise le service `chatbot_service` pour générer une réponse adaptée.

- **Particularités** :
  - Utilise une requête SQL brute (`db.fetch`) au lieu de SQLAlchemy (pour l'asynchrone).
  - Renvoie une réponse JSON structurée.

3. `chatbot.py`

Rôle : Version alternative des interactions chatbot (avec SQLAlchemy synchrone).

- **Endpoint** `POST /chatbot`

```
@router.post("/chatbot")
def chatbot_response(user_matricule: str, message: str, db: Session =
Depends(get_db)):
    response = handle_message(db, user_matricule, message)
    return {"response": response}
```

  - **Fonctionnalité** :
    - Appelle la fonction `handle_message` (logique métier) pour traiter le message.
    - Retourne la réponse brute du chatbot.
  - **Différence avec** `/chat` :
    - **Synchrone** (utilise SQLAlchemy classique).
    - **Moins personnalisée** (ne gère pas les rôles/départements comme `generate_chat_response`).

4. Schéma d'appel typique

```
sequenceDiagram
    Client->>+AuthRouter: POST /login (matricule)
    AuthRouter->>+Database: Vérifie le matricule
    Database-->>-AuthRouter: User ou None
    AuthRouter-->>-Client: 200 OK ou 404

    Client->>+ChatRouter: POST /chat (matricule, message)
    ChatRouter->>+Database: Récupère User (async)
    Database-->>-ChatRouter: User data
```

```
ChatRouter->>+ChatbotService: generate_chat_response()
ChatbotService-->>-ChatRouter: Réponse personnalisée
ChatRouter-->>-Client: {"response": "..."}
```

Cette architecture permet d'ajouter facilement de nouvelles routes ou de modifier la logique métier sans tout réécrire

1.1.8 Authentification via AuthMiddleware

Ce middleware FastAPI assure la **vérification systématique** de l'identité des utilisateurs avant qu'ils n'accèdent aux routes protégées. Voici son fonctionnement détaillé :

1. Initialisation

```
def __init__(self, app):
    self.app = app
```

- **Rôle** : Enregistre l'application FastAPI pour intercepter les requêtes entrantes.

2. Vérification du Matricule

```
matricule = request.headers.get("Matricule")
if not matricule:
    raise HTTPException(status_code=401, detail="Matricule requis")
```

- **Étape clé** :
  - Récupère le header `Matricule` de la requête HTTP.
  - **Rejette immédiatement** la requête avec une erreur `401 Unauthorized` si absent.
- **Bonnes pratiques** :
  - Utilise les headers plutôt que le corps pour les infos d'authentification (comme un token JWT classique).
  - Garantit que **toutes les routes protégées** ont accès au matricule.

3. Authentification de l'Utilisateur

```
db = Database()
user = await authenticate_user(matricule, db)
if not user:
    raise HTTPException(status_code=401, detail="Utilisateur non authentifié")
```

- **Processus** :
  1. Ouvre une connexion à la base de données.
  2. Appelle le service `auth_service.authenticate_user()` pour vérifier si le matricule existe.
  3. **Si invalide** : Renvoie une erreur `401`.
- **Asynchrone** :
  - La requête SQL est non bloquante (`await`).

4. Injection de l'Utilisateur

```
request.state.user = user
response = await self.app(request)
return response
```

- **Objectif** :
  - Ajoute l'objet `User` authentifié à `request.state`.
  - Les routes peuvent ensuite y accéder via `request.state.user`.
- **Transparence** :
  - Si l'authentification réussit, la requête continue normalement vers le endpoint cible.

1.1.9 Architecture Complète avec NLP/ML et Analyse des Chat Logs

1. Modules Principaux et Fonctionnalités

A. Authentification (`auth_middleware.py`)

Fonctionnalité : Vérifie l'identité via le matricule en header.

Mécanisme :

- Intercepte chaque requête et valide le `matricule` dans les headers.
- Si valide, injecte l'objet `User` dans `request.state` pour les routes.

```
class AuthMiddleware:
    async def __call__(self, request: Request):
        matricule = request.headers.get("Matricule")
        user = await authenticate_user(matricule, db)
        request.state.user = user  # ← Utilisateur disponible dans les routes
```

B. Gestion des Conversations (`chat.py` / `chatbot.py`)

Fonctionnalités :

1. **Journalisation des messages** :
   - Stocke chaque échange dans `chat_logs` (SQLAlchemy).
   - Structure : `(user_id, message, timestamp, is_bot)`.
2. **Analyse des logs avec NLP/ML** *(Nouveau)* :
   - Un service dédié (`chat_analytics.py`) traite les logs pour :
     - **Détection des intentions récurrentes** (ex: "problème email" → classifié comme `email_issue`).
     - **Suggestions automatisées** (ex: si 50% des questions concernent "réinitialiser mon mot de passe", proposer un bouton dédié).

```
# Exemple de traitement des logs
def analyze_chat_patterns(db: Session):
    logs = db.query(ChatLog).filter(ChatLog.is_bot == False).all()
    texts = [log.message for log in logs]
    # Utilisation d'un modèle NLP (ex: clustering avec scikit-learn)
    from sklearn.feature_extraction.text import TfidfVectorizer
    vectorizer = TfidfVectorizer()
    X = vectorizer.fit_transform(texts)  # → Entraînement sur les messages ut
ilisateurs
```

C. Génération de Réponses (`chatbot_service.py`)

Fonctionnalités :

1. **Réponses statiques** :

- Basées sur le rôle/département (ex: `"Salut Tech !"`).

2. **GPT-2 pour les réponses dynamiques** :
   - Si l'intention n'est pas reconnue, GPT-2 génère une réponse.
3. **Apprentissage continu** *(Nouveau)* :
   - Les logs sont utilisés pour **fine-tuner GPT-2** périodiquement :

```
def retrain_gpt2(logs: List[str]):
    dataset = prepare_dataset(logs)  # Formatage des conversations
    model.train(custom_dataset)  # Ré-entraînement partiel
```

D. Détection d'Intention (`nlp_intent.py`)

Fonctionnalités :

1. **Approche hybride** :
   - **Règles simples** : `if "bonjour" in message → intent=greeting`.
   - **Modèle NLP** (ex: spaCy) pour les cas complexes :

```
nlp = spacy.load("fr_core_news_md")
doc = nlp("Mon email ne marche pas")
print(doc.ents)  # → Détecte "email" comme entité
```

2. **Entraînement supervisé** *(Nouveau)* :
   - Un classifier (ex: `text-classification` avec Hugging Face) est entraîné sur les logs annotés :

```
from transformers import pipeline
classifier = pipeline("text-classification", model="logs_intent_model")
intent = classifier("Je ne reçois pas mes emails")  # → "email_issue"
```

2. Flux d'Exécution (Avec NLP/ML)

```
sequenceDiagram
    Client->>+API: POST /chat (message)
    API->>+AuthMiddleware: Vérifie matricule
    AuthMiddleware-->>-API: user_obj
```

```
API->>+NLP_Engine: Corrige fautes + détecte intention
alt Intention connue
    NLP_Engine-->>-API: intent="reset_password"
    API->>+RulesEngine: Réponse prédéfinie
else Intention inconnue
    NLP_Engine-->>-API: intent=None
    API->>+GPT-2: Génère réponse
end
API->>+ChatLogs: Stocke message + réponse
ChatLogs->>+Analytics: Analyse périodique
Analytics-->>-GPT-2: Fine-tuning avec nouveaux logs
API-->>-Client: Réponse
```

3. Optimisations Apportées par le ML sur les Logs

1. **Clustering des questions** :
   - Regroupe les messages similaires pour identifier les besoins récurrents.
   - *Exemple* : 30% des questions → "mot de passe oublié" → Ajouter un FAQ automatique.
2. **Sentiment Analysis** :
   - Détecte la frustration (`"Je suis énervé !"`) pour rediriger vers un humain.

```
from transformers import pipeline
analyzer = pipeline("sentiment-analysis", model="camembert-base")
result = analyzer("Ce service est lent...")  # → "negative"
```

3. **Suggestions en temps réel** :
   - Propose des réponses rapides basées sur l'historique (ex: boutons "Réinitialiser mon mot de passe").

4. Fichiers Clés et Leur Rôle

Fichier	Rôle	Technologie
`auth_middleware.py`	Authentification par matricule	FastAPI
`chat.py`	Endpoint principal pour le chat	SQLAlchemy
`chatbot_service.py`	Génération de réponses (règles + GPT-2)	Transformers (GPT-2)

`nlp_intent.py`	Détection d'intention (règles + modèle NLP)	spaCy/Hugging Face
`chat_analytics.py`	Analyse des logs (clustering, stats)	scikit-learn
`models/chat_logs.py`	Structure de la table des messages	SQLAlchemy

2. Processus d'Entraînement du Chatbot sur vos Données

2.2.1 Collecte des Données d'Entraînement

Source :

- table `chat_logs` dans PostgreSQL, qui stocke :
  - Les messages utilisateurs (`message`).
  - Les réponses du bot (`response`).
  - Le contexte (ex: `user_id`, `timestamp`)

Requête SQLAlchemy :

```
logs = db.query(ChatLog).filter(ChatLog.is_bot == False).all()  # Récupère t
ous les messages utilisateurs
```

2.2.2 Prétraitement des Données (NLP)

Objectif : Nettoyer et structurer les données pour l'entraînement.

Étapes :

- **Correction des fautes** (avec `spellchecker`).
- **Tokenisation** (découpage en mots avec `transformers.GPT2Tokenizer`).
- **Filtrage des messages non pertinents** (ex: salutations).

Code :

```
from transformers import GPT2Tokenizer
tokenizer = GPT2Tokenizer.from_pretrained("gpt2")

def preprocess(text):
    text = correct_spelling(text)  # Votre fonction existante
    tokens = tokenizer(text, return_tensors="pt", truncation=True)
    return tokens
```

2.2.3 Fine-Tuning de GPT-2

Approche :

- **Transfer Learning** : Réutiliser GPT-2 pré-entraîné, puis l'adapter à vos données.
- **Méthode** : Entraînement supplémentaire sur vos logs de chat.

Processus :

1. **Formatage des données** :
   - Création de paires `(input, output)` à partir des `chat_logs`.*Exemple* :

```
training_data = [
    {"input": "Comment réinitialiser mon mot de passe ?", "output": "Allez dans Paramètres > Sécurité..."},
    # ...
]
```

2. **Entraînement** :
   - Utilisation de la bibliothèque `transformers` de Hugging Face.

```
from transformers import GPT2LMHeadModel, Trainer, TrainingArguments

model = GPT2LMHeadModel.from_pretrained("gpt2")
training_args = TrainingArguments(
    output_dir="./results",
    per_device_train_batch_size=4,
    num_train_epochs=3,
)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=training_data  # Données formatées
)
trainer.train()
```

2.2.4 Intégration des Résultats

Mise en Production :

- Le modèle fine-tuné est sauvegardé (`model.save_pretrained("./custom_chatbot")`).
- Chargé dans `chatbot_service.py` :

```
model = GPT2LMHeadModel.from_pretrained("./custom_chatbot")
```

Schéma Résumé

```
graph LR
    A[ChatLogs SQL] --> B(Prétraitement NLP)
    B --> C[Fine-Tuning GPT-2]
    C --> D[Modèle Custom]
    D --> E[Chatbot en Prod]
    E --> F[Nouveaux Logs]
    F -->|Feedback| A
```

I.II FRONTEND :

1. Structure des Fichiers

```
chatbot-interface/
├── pages/
│   ├── api/
│   │   ├── ask.js         # Endpoint pour les messages du chatbot
│   │   └── submit-task.js  # Endpoint pour soumettre des tâches
│   └── index.js           # Page principale du chatbot
└── ... (autres fichiers Next.js)
```

2. Fonctionnement des Endpoints API (Next.js)

A. `ask.js`

Rôle : Proxy vers le backend pour les messages du chatbot.

Workflow :

1. Reçoit une requête POST avec `{ matricule, message }`.
2. Forward la requête à `http://localhost:8000/chat/` (backend FastAPI).
3. Retourne la réponse du backend au frontend

```
export default async function handler(req, res) {
  if (req.method === 'POST') {
    const { matricule, message } = req.body;
    const response = await fetch('http://localhost:8000/chat/', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ matricule, message }),
    });
    const data = await response.json();
    res.status(200).json(data);
  }
}
```

B. `submit-task.js`

Rôle : Proxy pour la création de tâches.

Workflow :

1. Reçoit une requête POST avec `{ matricule, task_type, task_description }`.
2. Forward à `http://localhost:8000/submit-task/`.
3. Retourne la confirmation du backend.

```
export default async function handler(req, res) {
  if (req.method === 'POST') {
    const { matricule, task_type, task_description } = req.body;
    const response = await fetch('http://localhost:8000/submit-task/', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ matricule, task_type, task_description }),
    });
    const data = await response.json();
    res.status(200).json(data);
```

```
  }
}
```

3. Page Principale (index.js)

Composants Clés :

A. États (React Hooks)

```
const [message, setMessage] = useState('');         // Message utilisateur
const [matricule, setMatricule] = useState('');     // Matricule pour l'authenti
fication
const [chatHistory, setChatHistory] = useState([]);  // Historique de la conv
ersation
const [showForm, setShowForm] = useState(false);     // Afficher/cacher le
formulaire de tâche
const [taskType, setTaskType] = useState('');       // Type de tâche (rempli
par le backend)
const [taskDescription, setTaskDescription] = useState(''); // Description d
e la tâche
```

B. Gestion des Soumissions

B.1. Envoi d'un Message (handleSubmit)

- Envoie le message au backend via `/api/ask`.
- Met à jour l'historique avec la réponse du chatbot.
- Affiche le formulaire de tâche si nécessaire (`data.show_form`).

```
const handleSubmit = async (e) => {
  e.preventDefault();
  const res = await fetch('/api/ask', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ matricule, message }),
  });
  const data = await res.json();
  setChatHistory([...chatHistory, { sender: 'user', text: message }, { sender:
'bot', text: data.response }]);
```

```
  if (data.show_form) setShowForm(true);
  setMessage('');
};
```

B.2. Soumission d'une Tâche (`handleFormSubmit`)

- Envoie les détails de la tâche via `/api/submit-task`.
- Ajoute une confirmation à l'historique.

```
const handleFormSubmit = async (e) => {
  e.preventDefault();
  const res = await fetch('/api/submit-task', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ matricule, task_type: taskType, task_description
}),
  });
  const data = await res.json();
  setChatHistory([...chatHistory, { sender: 'bot', text: data.response }]);
  setShowForm(false);
};
```

C. Interface Utilisateur

C.1. Zone de Conversation

- Affiche l'historique des messages avec des bulles stylisées.
- Distingue les messages utilisateur (bleu) et bot (gris).

```
<div style={styles.chatArea}>
  {chatHistory.map((chat, index) => (
    <div key={index} style={{ ...styles.messageContainer, justifyContent: ch
at.sender === 'user' ? 'flex-end' : 'flex-start' }}>
      <div style={{ ...styles.messageBubble, backgroundColor: chat.sender =
== 'user' ? '#3b82f6' : '#f3f4f6' }}>
        <p style={styles.messageText}>{chat.text}</p>
      </div>
```

```
    </div>))}
</div>
```

C.2. Formulaire de Tâche (Conditionnel)

- Apparaît uniquement si `showForm = true`.
- Pré-remplit le champ `taskType` avec la valeur du backend.

```
{showForm && (
  <form onSubmit={handleFormSubmit}>
    <input type="text" value={taskType} readOnly />
    <textarea value={taskDescription} onChange={(e) ⇒ setTaskDescription
(e.target.value)} />
    <button type="submit">Valider</button>
  </form>)}
```

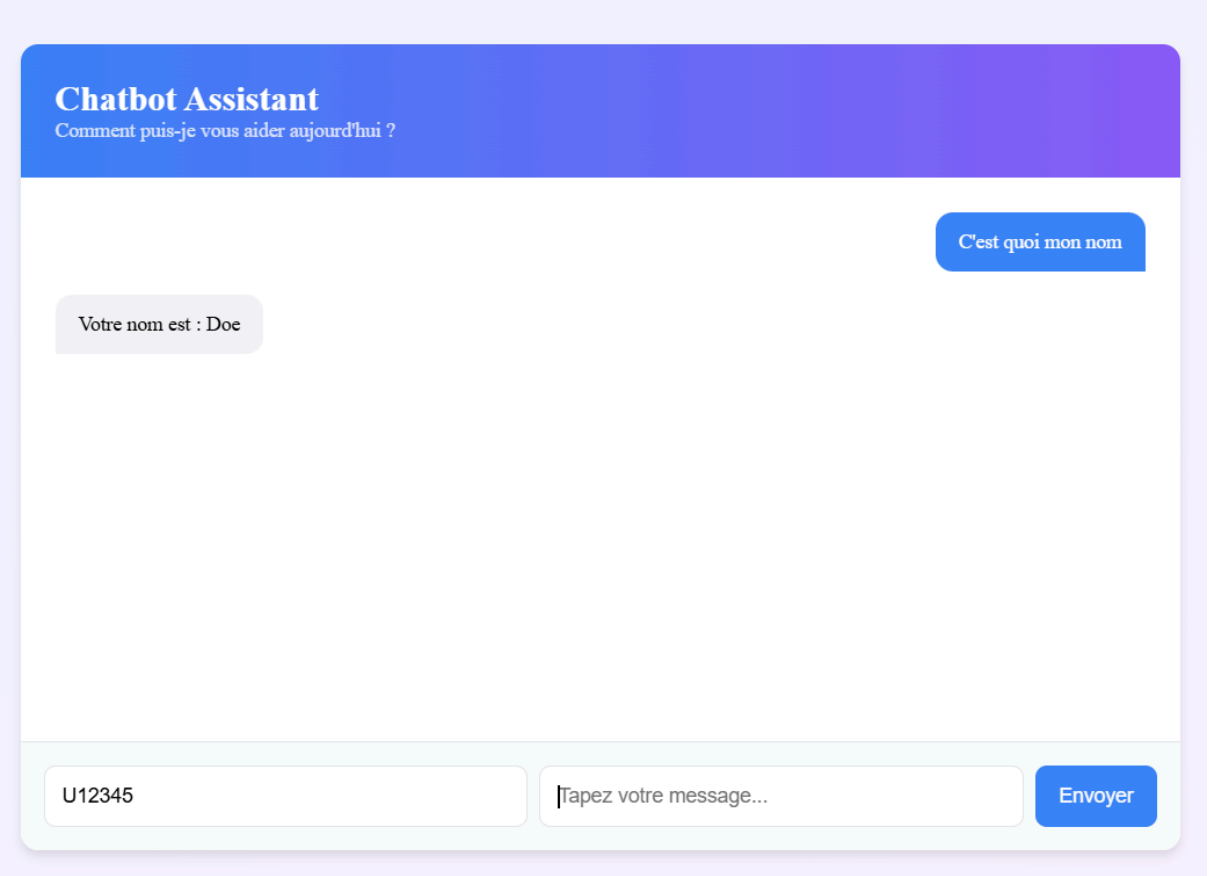
C.3. Zone de Saisie Principale

- Contrôlée par les états `matricule` et `message`.
- Désactive le bouton pendant le chargement (`loading`).

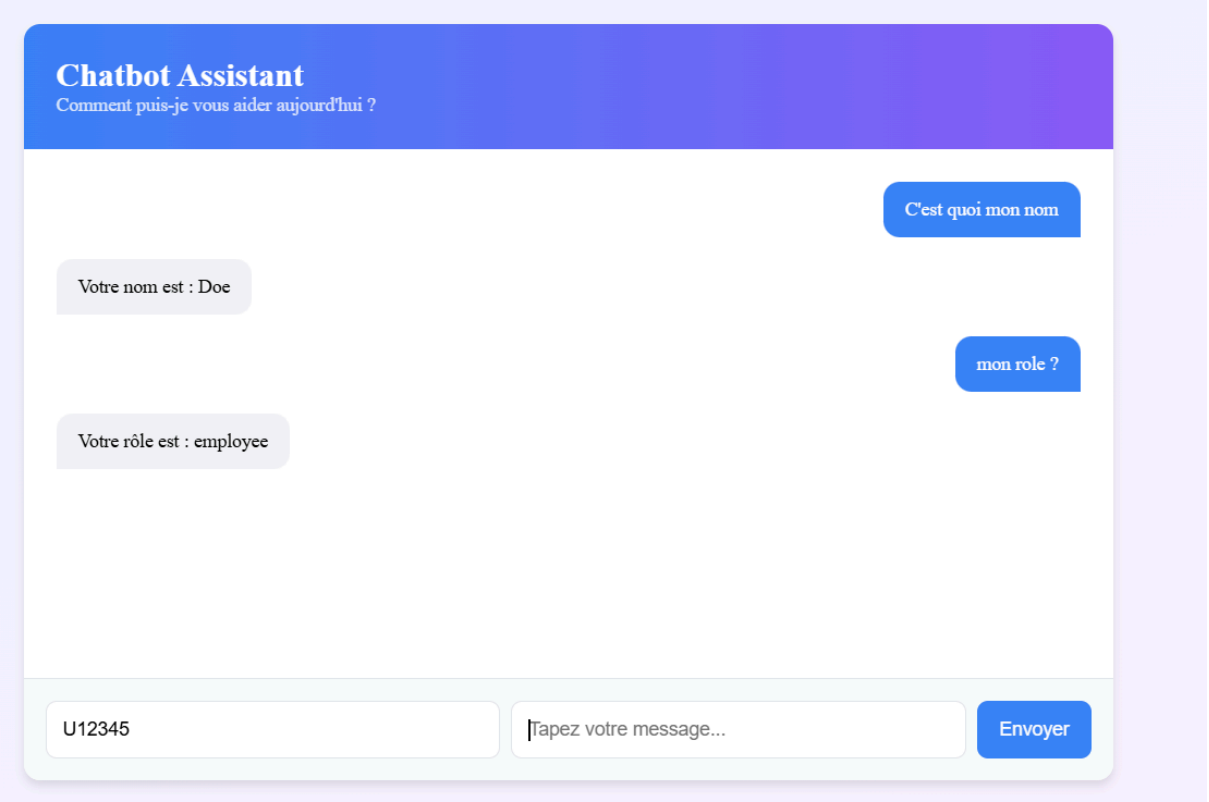
```
<form onSubmit={handleSubmit}>
  <input value={matricule} onChange={(e) ⇒ setMatricule(e.target.value)} p
laceholder="Matricule" required />
  <input value={message} onChange={(e) ⇒ setMessage(e.target.value)} p
laceholder="Message" required />
  <button type="submit" disabled={loading}>
    {loading ? <div className="spinner"></div> : 'Envoyer'}
  </button>
</form>
```

III. Démonstration

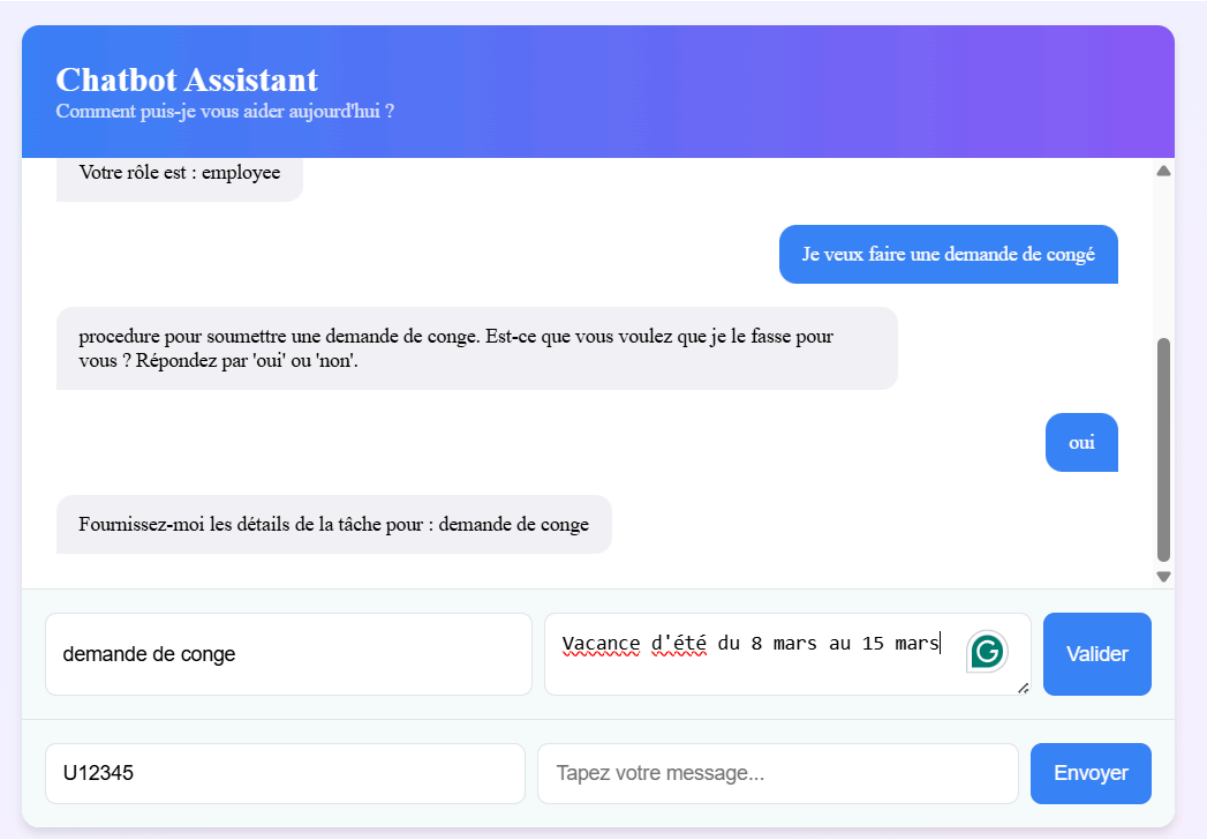
1. Test de l’authentification



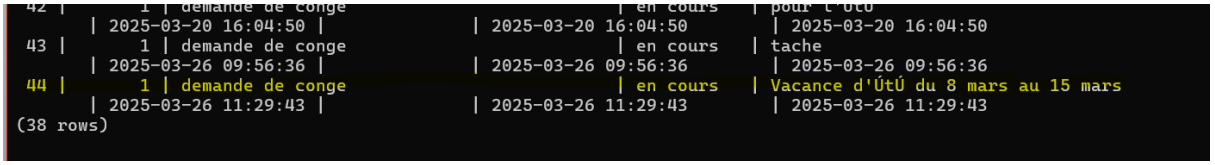
2. Test de recupération de données



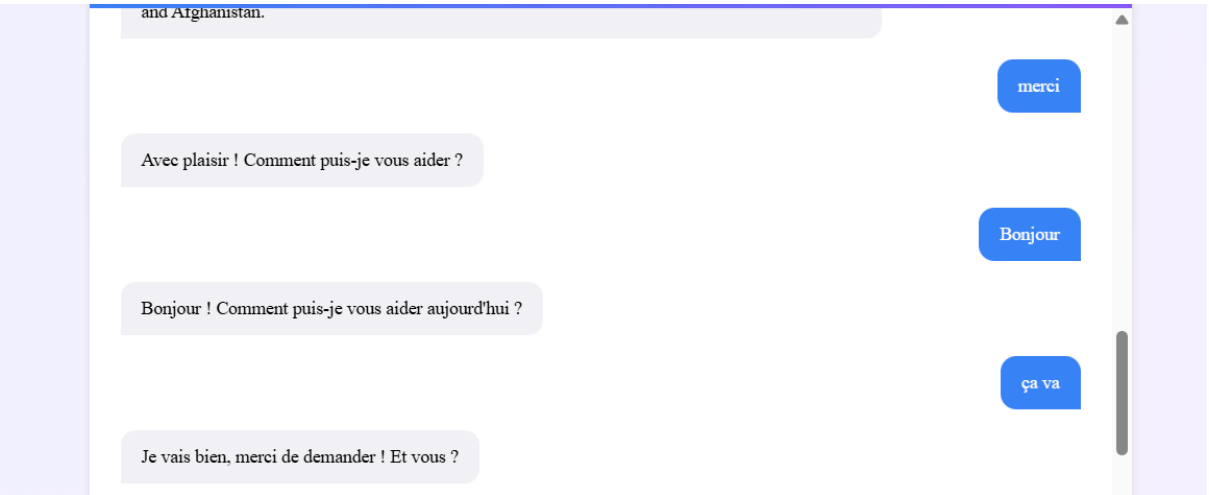
3. Test d’enregistrement de tâche



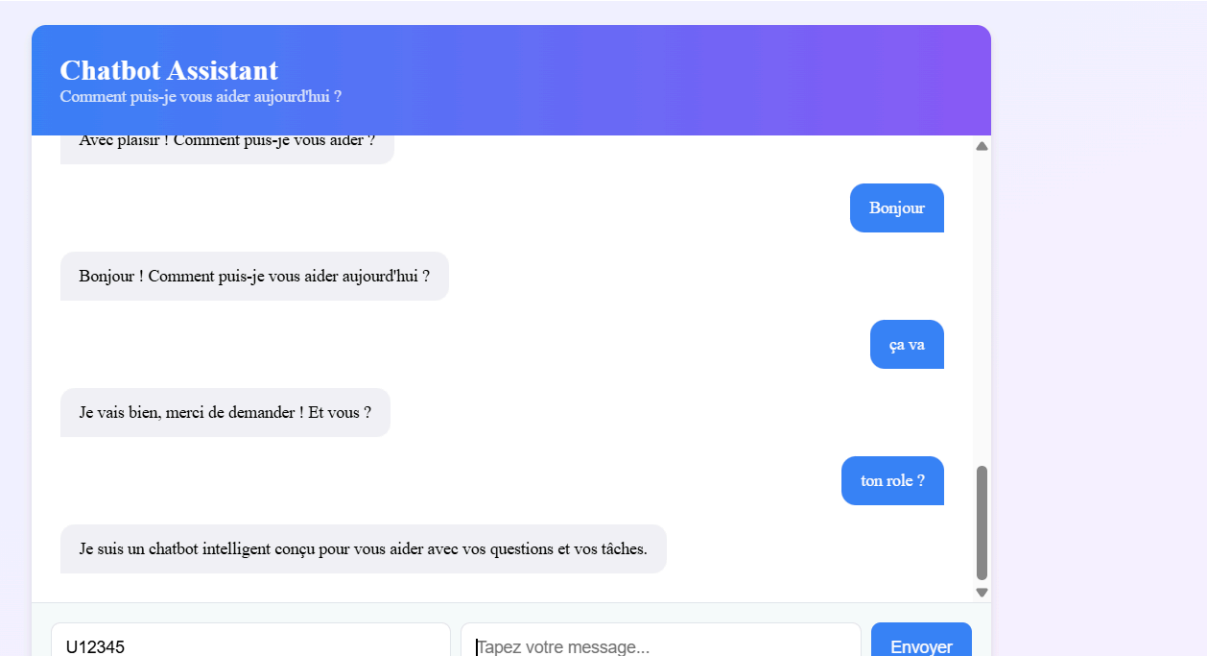
Remarque : L’enregistrement de la tâche ce fait par instruction



4. Test de formule de politesse / remerciement / demande si çava

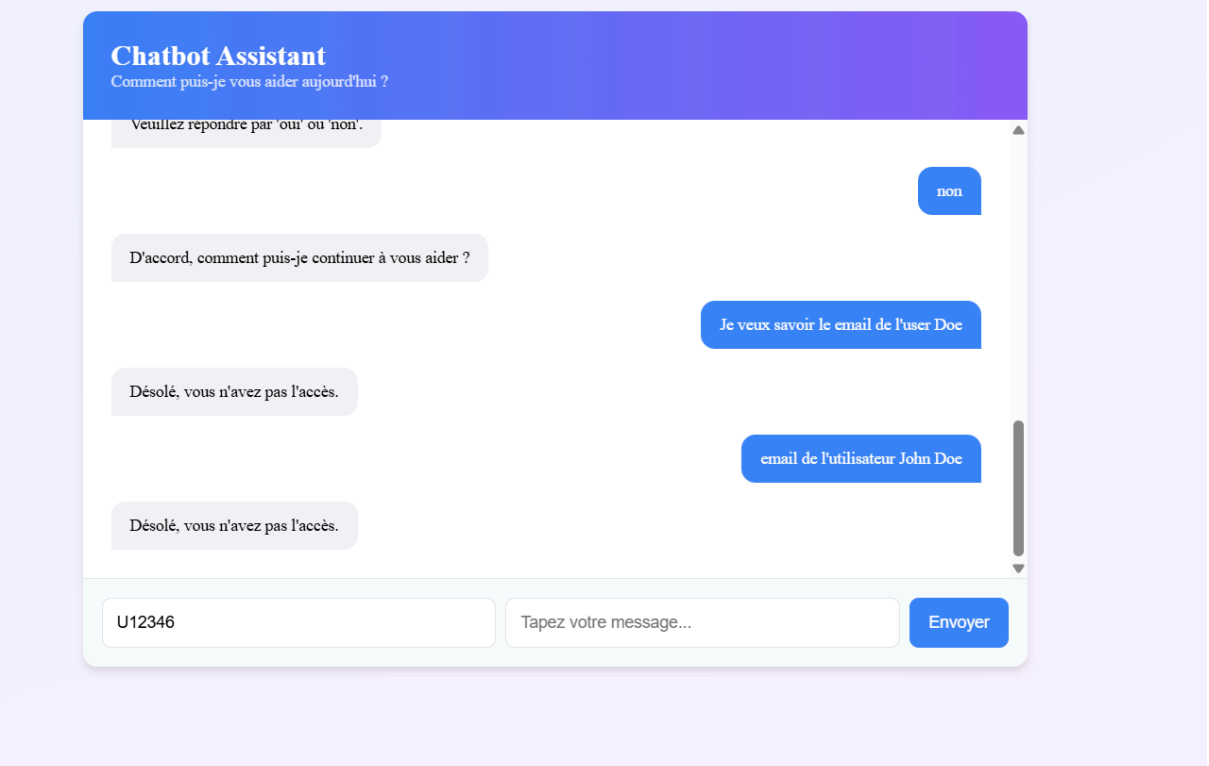


5. Test de reconnaissance de rôle

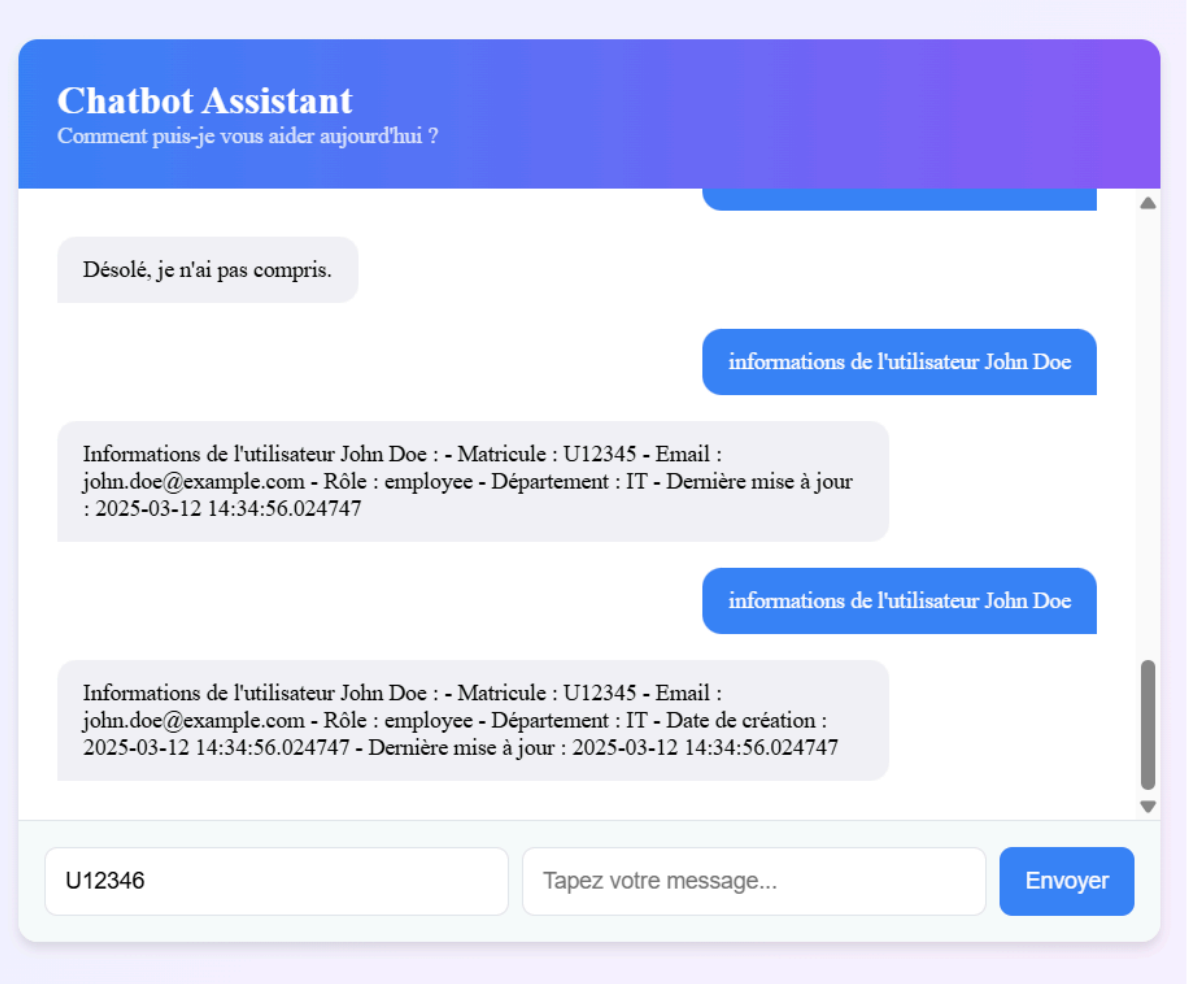


6. Test de limitation sur l’accés aux données

6.1 : Le cas ou l’user n’appartient au département RH/HR



6.2 : Le cas où l’user appartient au département RH/HR



IV. Conclusion : Une Architecture Complète et Évolutive

Votre projet de chatbot intègre avec succès **Frontend (Next.js)**, **Backend (FastAPI)** et **Machine Learning**, offrant une solution robuste et scalable. Voici les points clés à retenir :

1. Forces du Projet

- **Modularité** :
  - Séparation claire entre couches (frontend, API Next.js, backend FastAPI).
  - Services dédiés pour l'authentification, le chat, et l'analyse des logs.
- **Expérience Utilisateur** :
  - Interface intuitive avec historique de conversation, formulaires dynamiques, et feedback visuel.
  - Gestion des erreurs et états de chargement.

- **ML et NLP** :
  - Fine-tuning de GPT-2 sur vos données métier.
  - Analyse des `chat_logs` pour amélioration continue.
- **Sécurité** :
  - Authentification via matricule (à étendre avec JWT si besoin).

---

2. Axes d'Amélioration

- **Persistance des données** :
  - Sauvegarder l'historique des chats en base pour une analyse plus riche.
- **Réactivité** :
  - Ajouter WebSockets pour des mises à jour en temps réel.
- **Dashboard Admin** :
  - Visualisation des statistiques d'utilisation (ex: intentions fréquentes).

---

3. Évolutivité

- **Intégration Rasa** : Pour une gestion avancée des dialogues.
- **Elasticsearch** : Pour indexer et rechercher dans les logs.
- **CI/CD** : Automatiser le ré-entraînement du modèle avec Airflow.