Documentación Chat-anne

Instalación del proyecto

Requisitos

Instalación con Docker

- Docker: Instalación de Docker
- Docker Compose: Instalación de Docker Compose

Docker se encargará de instalar todas las dependencias necesarias para el proyecto, facilitando su configuración y ejecución.

Instalación local

Si prefieres correr la aplicación de manera local, se requerirán ciertas dependencias y bibliotecas tanto para el frontend como para el backend. Estas incluyen:

- Frontend: Vite, Tailwind y React
- Backend: Flask, Langchain, MongoDB y ChromaDB

Para simplificar la instalación de estas dependencias, se ha desarrollado un script setup.sh que automatiza este proceso.

Instalación del proyecto

1. Clona este repositorio en tu máquina local:
   git clone https://github.com/gdicosimo/Chat-anne

2. Cambia al directorio del repositorio:
   cd Chat-anne

Configuración

Para la instalación automática de las dependencias del proyecto, primero que nada deberá ejecutarse el script setup.sh, proporcionado en el directorio raíz del proyecto:
./setup.sh

Este script abrirá una nueva terminal de CMD y realizará las siguientes tareas:

- Consultará si se desean instalar las dependencias del frontend y del backend de manera local (recomendado si se desea levantar de forma local e individual cada stack). Esto puede rechazarse, mientras que si se elige hacerlo:

    - Instalará las dependencias del backend utilizando un entorno virtual de Python ("venv") en el directorio backend .
    - Instalará las dependencias del frontend utilizando npm en el directorio frontend .
- Seguidamente, solicitará la API key de Gemini obtenida según los pasos descriptos en el apartado de Requisitos . Esta última deberá copiarse y pegarse, y a continuación el script creará un archivo .env en el subdirectorio de backend\src\rag\model. De este modo, la aplicación podrá comunicarse con el modelo provisto por Gemini.
- Por último, consultará si se quiere o no correr la aplicación. Si se elige que sí, dará la posibilidad de hacerlo en modo de desarrollo o producción (preferentemente elegir el de producción). Entonces, dependiendo el caso ejecutará automáticamente el comando de docker compose -f compose.yml up --build o docker compose -f compose-dev.yml up --build, que automáticamente creará los contenedores de Docker, y también los levantará, comenzando a correr así tanto el frontend como el backend. Cuando esto último ocurra, se informará al respecto sobre la misma terminal abierta. No intentar acceder antes a la aplicación puesto que aún no estará ejecutándose completamente, por lo que será inaccesible mientras tanto.

Es importante aclarar que una vez que se desee dejar de ejecutar la aplicación, será necesario dar de baja los contenedores. Para ello, ejecutar el paso número 3 de la sección de Desarrollo o de Producción (según corresponda) en el apartado de Uso .

Tener en cuenta que si se está trabajando desde Windows, será necesario primero iniciar el demonio de Docker Engine para poder levantar los contenedores mientras este se ejecuta en segundo plano. Por el contrario, desde Linux esto no será necesario.

Uso

Si bien el script anteriormente mencionado permite correr la aplicación directamente en modo de producción, también se creó un modo de desarrollo para trabajar sobre la misma.

A continuación se explica brevemente cómo se puede levantar la aplicación de ambas maneras sin necesidad de utilizar el script (tener en cuenta que de todos modos al menos una vez será necesario correrlo dado que los siguientes comandos no incluyen la instalación de dependencias, bibliotecas ni creación del archivo asociado a la API key).

Modo de Desarrollo

Para ejecutar la aplicación en modo de desarrollo, sigue estos pasos:

1. Ejecuta el siguiente comando para construir y levantar los contenedores de Docker en el entorno de desarrollo:

    docker compose -f compose-dev.yml up --build

    No es necesario realizar un --build cada vez que se desea iniciar el proyecto en desarrollo. Este proceso se lleva a cabo solo una vez o cuando se modifica el archivo compose-dev.yml

2. Una vez que todos los contenedores estén levantados, podrás acceder a la aplicación en tu navegador web:
    - Backend (Flask): http://localhost:5000
    - Frontend (React): http://localhost:4200

3. Realiza los cambios necesarios en el código fuente. Los cambios realizados en el código se reflejarán automáticamente en la aplicación sin necesidad de reiniciar los contenedores.

4. Cuando hayas terminado de trabajar, puedes detener los contenedores presionando Ctrl + C en la terminal donde se están ejecutando, y luego ejecutar el siguiente comando para detenerlos y eliminarlos:
    docker compose -f compose-dev.yml up down

Modo de Producción

Para desplegar la aplicación en un entorno de producción, puedes utilizar los mismos comandos de Docker Compose.

1. Ejecuta el siguiente comando para construir y levantar los contenedores de Docker en el entorno de producción:
    docker compose -f compose.yml up --build
2. En este caso ya no es necesario especificar el puerto, dado que Nginx actúa como un proxy inverso, haciendo que ahora el backend ya no sea accesible (corre en el puerto 8080 del contenedor pero no se mapea con ninguno del host, solo puede ser accedido por Nginx). Por otro lado, el frontend corre en el puerto 80 del host. De esta manera, puede accederse a la aplicación a través del siguiente link:
    - Chat-anne : http://localhost/
3. Cuando hayas terminado de trabajar, puedes detener los contenedores presionando Ctrl + C en la terminal donde se están ejecutando, y luego ejecutar el siguiente comando para detenerlos y eliminarlos:
    docker compose -f compose.yml up down

Entradas y salidas para los endpoints

Endpoints de autenticación

Los siguientes endpoints están asociados a las funcionalidades de registro y logueo de los usuarios, trabajando sobre la base de datos de Mongo y el modelo de usuarios.

POST "auth/register" : Relacionado a la funcionalidad de registro del usuario. El cuerpo de la solicitud recibe un JSON con el nombre de usuario (en este caso el mail) y la contraseña del mismo. Puede obtenerse un código de respuesta 200 y un mensaje indicando que el usuario se registró efectivamente, o de lo contrario un código de error 400 con el mensaje de que el mail ya está registrado. Si el registro fue exitoso, el nuevo usuario se almacena en la base de datos de Mongo.

Endpoint `localhost:5000/auth/register`
Request `{"username" : "usr", "pwd" : "pwd"}`
Response `{"message" : "usr registrado correctamente"}`

POST "auth/login" : Relacionado a la funcionalidad de logueo del usuario. El cuerpo de la solicitud recibe un JSON con el nombre de usuario y la contraseña del mismo. Puede obtenerse un mensaje indicando que el logueo se dio exitosamente, además de setear el "access_token_cookie" que es lo que efectivamente loguea al usuario (todo esto si al consultar a la base de Mongo, se encuentra al usuario). De lo contrario puede obtenerse un código de error 401 y un mensaje de logueo fallido si el usuario no estaba registrado.

Endpoint `localhost:5000/auth/login`
Request `{"username" : "usr", "pwd" : "pwd"}`
Response `{"message" : "login Success"}`

Endpoints asociados a los chats

En este caso, los endpoints están asociados a los chats y conversaciones del usuario, para lo cual se trabaja sobre la base de datos de Mongo y el modelo de chats, como también sobre la base de datos vectorial de Chroma, que contiene una colección por cada chat, donde se guardan los vectores con la información de cada PDF adjunto.

POST "chats/" : Relacionado a la funcionalidad de creación de un nuevo chat. Recibe como parámetro el id del chat, y lo almacena en la base de datos de Mongo indicando el dueño del mismo (a partir del token de autenticación obtenido a través de la cookie), su nombre, y el momento de creación. También se crea una colección en la base de datos de Chroma, correspondiente al chat en cuestión (en dicha colección se almacenan los vectores asociados a los PDF del mismo). Si no hubo ningún error, se retorna el código 200 y el id del chat, o sea el que genera automáticamente Mongo.

Endpoint `localhost:5000/chats/`

Request `{"id_chat" : "chat1" }`
Response `{"id_chat" : "" }`

PUT "chats/rename-chat" : Relacionado al renombramiento de un chat. Recibe como parámetro el id del chat en la base de Mongo, y el nuevo nombre. Si no había ningún chat con ese id, retorna un error con código 400, de lo contrario, informa que se modificó exitosamente el nombre y un código 200.

Endpoint `localhost:5000/chats/rename-chat`
Request `{"id_chat" : "chat1", "new_value" : "chat2"}`
Response `{"message": "El chat chat1 se cambio a .. correctamente!"}`

DELETE "chats/remove-chat" : Relacionado con la eliminación de un determinado chat de un usuario. Recibe como parámetro el id del chat, y lo elimina de la base de Mongo, además de eliminar la colección asociada en la base de Chroma. Luego devuelve un mensaje indicando esto junto con el código 200.

Endpoint `localhost:5000/chats/remove-chat`
Request `{"id_chat" : ""}`
Response `{"message": "El chat .. se eliminó correctamente!"}`

PUT "chats/append-pdf" : Relacionado con el agregado de un pdf a un chat, para así utilizarlo en las respuestas al usuario. Recibe como parámetros el id del chat y el pdf en cuestión. Si no existe un chat con el id indicado y el usuario que se encuentra logueado (determinado a partir del jwt), se envía un mensaje con el código de error 400, de lo contrario se informa que se agregó correctamente con el código 200. En este último caso, se asocia el pdf al chat correspondiente, en la base de Mongo sobre el modelo de chats. También se hace el procesamiento del pdf y su división en documentos (chunks), que se guardan en la colección del chat en la base de Chroma.

Endpoint `localhost:5000/chats/append-pdf`
Request `{"id_chat" : "", "pdf_file": ""}`
Response `{"message": "El pdf .. se agregó al chat .. correctamente!"}`

PUT "chats/pop-pdf" : Relacionado con la eliminación de un pdf de un chat del usuario, para que ya no se conteste más en base a este. Del mismo modo que el endpoint anterior, recibe como parámetro el id del chat y el nombre del pdf. Se elimina tanto del conjunto de PDF´s del chat en la base de Mongo, como de la base de la colección correspondiente en la base de Chroma (para lo cual se eliminan los vectores asociados). Posteriormente se informa esto con un mensaje y el código 200.

Endpoint `localhost:5000/chats/pop-pdf`
Request `{"id_chat" : "", "pdf_name": ""}`
Response `{"message" : "El pdf .. se eliminó del chat .. correctamente!"}`

POST "chats/message" : Relacionado al envío por parte del usuario de un mensaje en un chat determinado, para su consecuente respuesta. Para ello, recibe los parámetros de id de chat y query. Si no existe ese id de chat asociado al usuario logueado actualmente, se informa de ello. Pero si no es el caso, primero se obtiene la respuesta a partir del modelo, usando la información de los PDF cargados hasta el momento en la colección de Chroma para ese chat (si no hay ninguno cargado, la respuesta indicará eso). Luego se guarda el mensaje en el conjunto de mensajes asociados a ese chat en la base de datos de Mongo, almacenando tanto la query, como la respuesta y el momento de creación (este último es esencial para poder mostrar en orden la conversación al volver a abrir ese chat). También se retorna el código 200.

Endpoint `localhost:5000/chats/message`
Request `{"id_chat" : "", "query" : ""}`
Response `{""}`

GET *"chats/" :* Relacionado a la obtención de todos los id y nombres de los chats de un usuario determinado. No recibe parámetros, sino que directamente utiliza el token proporcionado por la cookie (jwt), y devuelve el código de respuesta 200.

Endpoint `localhost:5000/chats/`
Request `{-}`
Response `{"chats" : ""}`

GET *"chats/messages" :* Relacionado a la obtención de todos los mensajes correspondientes a un chat determinado de un usuario. Esto permite listarlos cada vez que se abre ese chat, para así poder ver el historial de la conversación. Recibe como parámetro el id del chat. Si en la base de Mongo no existiera ningún chat con ese id asociado al usuario logueado, se envía esto como respuesta, de lo contrario se retornan todos los mensajes (tanto del usuario como de respuesta del modelo) en orden cronológico y con el código de respuesta 200.

Endpoint `localhost:5000/chats/messages`
Request `{"id_chat" : ""}`
Response `{"chat" : ""}`