Informe Técnico: Chatbot de Consulta SQL

1. Introducción

Imagina que quieres preguntarle cosas a una base de datos, como si fuera una gran tabla de Excel, pero en lugar de escribir fórmulas raras, ¡quieres hacerlo con palabras normales! Este conjunto de archivos crea un "chatbot" que hace precisamente eso. Toma tus preguntas en lenguaje natural y las convierte en el lenguaje que entiende la base de datos (SQL).

2. Descripción de los Archivos (Uno por Uno)

- **chatbot.py:** Este es el "cerebro" del chatbot.
    - Piensa en él como el que recibe tu pregunta, la pasa al traductor, obtiene la respuesta de la base de datos y te la muestra de una manera fácil de entender.
    - ChatbotSQL es la clase principal que se encarga de inicializar la conexión a la base de datos y configurar el agente de LangChain.
    - Puede entender tanto preguntas normales como instrucciones directas en el lenguaje de la base de datos (SQL), lo cual es útil para los programadores.
- **config.py:** Este archivo es como el "archivador" de la configuración.
    - Guarda información importante como dónde está la base de datos, el nombre de usuario y la contraseña, y la clave para usar el traductor de lenguaje (Groq).
    - Usa algo llamado dotenv para leer esta información de un archivo externo, ¡así no tienes que escribir contraseñas directamente en el código!.
- **database.py:** Este archivo se encarga de hablar directamente con la base de datos.
    - Tiene las funciones para conectarse, enviar preguntas (consultas) y obtener las respuestas.
    - Usa una herramienta llamada SQLAlchemy para hacer esto de manera organizada.
    - La clase DatabaseManager es la que gestiona estas operaciones.
- **langchain_setup.py:** Este archivo configura el "traductor" de lenguaje.
    - Utiliza LangChain y Groq para convertir el lenguaje natural a SQL.
    - Le da al traductor información sobre cómo está organizada la base de datos para que pueda hacer las preguntas correctas.
    - Define el "prompt", que son las instrucciones que se le dan al modelo de lenguaje para generar las consultas SQL.
- **models.py:** Este archivo define la estructura de la base de datos en el código.
    - Piensa en él como un plano de las tablas de la base de datos, definiendo qué tipo de información contiene cada una (por ejemplo, clientes, productos, pedidos).
    - También usa SQLAlchemy para esto.

    - Define "modelos" que representan las tablas de la base de datos (Clientes, Productos, Pedidos, DetallesPedido).
- **sql_database.txt:** Este archivo es un "script" que se usa para crear la base de datos desde cero.
    - Contiene las instrucciones en lenguaje SQL para crear las tablas y agregar algunos datos de ejemplo.

3. Cómo Trabajan Juntos

Estos archivos no trabajan aislados, ¡colaboran!

- chatbot.py usa config.py para saber dónde está la base de datos y usa langchain_setup.py para configurar el traductor. También usa database.py para enviar y recibir información de la base de datos.
- config.py le da a database.py y a langchain_setup.py la información necesaria para conectarse a la base de datos.
- database.py usa models.py para entender la estructura de la base de datos.
- langchain_setup.py usa config.py para obtener la clave del traductor y database.py para saber cómo está organizada la base de datos.
- models.py también usa config.py para conectarse a la base de datos y define cómo se ven las tablas.
- sql_database.txt es como el "molde" inicial para la base de datos, que los otros archivos usan como referencia.

En Resumen

Este conjunto de archivos crea un sistema donde puedes hacer preguntas sobre una base de datos en tu propio lenguaje. Es como tener un asistente que entiende lo que quieres y te da las respuestas de la base de datos de manera clara.

2. Explicación detallada del código de cada archivo

1. chatbot.py

Este archivo contiene la lógica principal del chatbot.

- **Importaciones:** Importa las bibliotecas y módulos necesarios:
    - typing: Para definir los tipos de datos (Opcional, Dict, Any, List, Tuple).
    - langchain_setup: Para la configuración del agente SQL.
    - database: Para la interacción con la base de datos.
    - langchain_community.utilities.sql_database: Clase de LangChain para conectarse a SQL databases.
    - config: Para obtener la URI de la base de datos.

        - re: Para expresiones regulares (usado para extraer consultas SQL).
        - logging: Para registrar información y errores.
- **Configuración de Logging:** Configura el sistema de registro para guardar mensajes informativos y de error.
- **Clase ChatbotSQL:** Define el comportamiento del chatbot.
    - __init__:
        - Inicializa el DatabaseManager para manejar la conexión a la base de datos.
        - Establece el agente SQL (self.agent) y la conexión de LangChain a la base de datos (self.sql_db).
        - Si hay algún error al conectar o inicializar, se asegura de limpiar los recursos y levanta un error.
    - process_query:
        - Esta es la función principal para procesar las consultas del usuario.
        - Si la consulta comienza con "sql:", la ejecuta directamente en la base de datos usando _execute_direct_query. Esto es útil para ejecutar comandos SQL directamente.
        - Si no es una consulta SQL directa, la pasa al agente de LangChain para que la traduzca y la ejecute.
        - Formatea la respuesta y maneja los errores.
    - _execute_direct_query: Ejecuta una consulta SQL directamente usando el DatabaseManager.
    - _format_results: Toma los resultados de la base de datos y los formatea como una tabla HTML para mostrarlos al usuario.
    - _extract_sql_query: Extrae la consulta SQL generada por el agente de LangChain de la respuesta. Usa expresiones regulares para encontrar el bloque de código SQL.
    - _cleanup_resources: Cierra la conexión a la base de datos para liberar recursos.
    - close: Cierra todos los recursos del chatbot de manera ordenada.
    - __enter__ y __exit__: Permiten usar el chatbot en un bloque with, asegurando que la conexión se cierre automáticamente.

2. config.py

Este archivo maneja la configuración de la aplicación.

- **Importaciones:**
    - os: Para interactuar con el sistema operativo (por ejemplo, para leer variables de entorno).
    - dotenv: Para cargar variables de entorno desde un archivo .env.

    - pathlib.Path: Para manejar las rutas de los directorios.
- **load_dotenv():** Carga las variables de entorno desde un archivo .env (si existe). Esto es importante para no incluir información sensible como contraseñas directamente en el código.
- **DB_CONFIG:** Un diccionario que contiene la configuración para la conexión a la base de datos PostgreSQL (host, database name, user, password, port). Obtiene estos valores de las variables de entorno.
- **GROQ_API_KEY y MODEL_NAME:** Guardan la clave de la API y el nombre del modelo para el servicio Groq, que se utiliza para la traducción de lenguaje natural a SQL.
- **APP_CONFIG:** Configuración general de la aplicación, como las rutas a los directorios de plantillas y archivos estáticos, y el modo de depuración.
- **Validación:** Verifica que la GROQ_API_KEY esté configurada. Si no lo está, levanta un error, ya que la clave de la API es esencial para que el chatbot funcione.
- **get_db_uri():** Una función que construye la "cadena de conexión" (URI) que SQLAlchemy utiliza para conectarse a la base de datos. Combina la información del diccionario DB_CONFIG en el formato correcto.

3. database.py

Este archivo se encarga de la comunicación con la base de datos.

- **Importaciones:**
    - sqlalchemy.orm.sessionmaker: Para crear "sesiones" de base de datos (una forma de interactuar con la base de datos).
    - sqlalchemy.exc.SQLAlchemyError: Para manejar errores específicos de SQLAlchemy.
    - sqlalchemy.inspection.inspect: Para obtener información sobre la base de datos (como el esquema).
    - src.models.engine: Importa el "engine" de SQLAlchemy que ya está configurado en models.py. El "engine" es el componente que SQLAlchemy usa para comunicarse con la base de datos.
    - typing: Para las anotaciones de tipo.
    - logging: Para el registro de eventos.
- **DatabaseManager Class:**
    - __init__: Inicializa el DatabaseManager. Crea self.Session usando sessionmaker, que se utilizará para crear objetos de sesión. self.session se inicializa como None.
    - connect():
        - Establece una conexión a la base de datos.

        - Si no hay una sesión activa, crea una nueva usando self.Session().
        - Maneja las excepciones (SQLAlchemyError) que pueden ocurrir durante la conexión.
    - execute_query(query, params=None):
        - Ejecuta una consulta SQL dada (query).
        - Opcionalmente, puede recibir parámetros (params) para la consulta, lo cual es importante para prevenir ataques de "inyección SQL".
        - Si la consulta devuelve filas (como un SELECT), obtiene los nombres de las columnas y los datos.
        - Si la consulta no devuelve filas (como un INSERT, UPDATE o DELETE), realiza un commit() para guardar los cambios en la base de datos.
        - Maneja los errores y realiza un rollback() en caso de error para deshacer cualquier cambio parcial.
    - get_database_schema():
        - Obtiene el esquema de la base de datos. El esquema describe la estructura de la base de datos: tablas, columnas, tipos de datos, relaciones, etc..
        - Utiliza sqlalchemy.inspection.inspect para obtener esta información.
        - Organiza el esquema en un diccionario, donde las claves son los nombres de las tablas, y los valores son diccionarios que describen las columnas y las claves foráneas de cada tabla.
    - close(): Cierra la sesión de la base de datos. Es importante cerrar las sesiones para liberar recursos.

4. langchain_setup.py

Este archivo configura el agente de LangChain que se utiliza para interactuar con la base de datos.

- **Importaciones:**
    - langchain_community.agent_toolkits.create_sql_agent: Función de LangChain para crear un agente diseñado para interactuar con bases de datos SQL.
    - langchain_groq.ChatGroq: Clase para usar el modelo de lenguaje ChatGroq.
    - langchain_core.prompts.ChatPromptTemplate: Para crear "prompts" (instrucciones) para el modelo de lenguaje.
    - langchain_community.utilities.sql_database.SQLDatabase: Clase de LangChain para conectarse a SQL databases.
    - src.config: Para obtener la clave de la API de Groq y la URI de la base de datos.
    - logging: Para el registro de eventos.

    - typing: Para las anotaciones de tipo.
- **setup_sql_agent(db_manager) Function:**
    - Verifica que la GROQ_API_KEY esté configurada.
    - Crea una instancia de SQLDatabase de LangChain para conectarse a la base de datos.
    - Obtiene el esquema de la base de datos usando el db_manager (instancia de DatabaseManager). El esquema es crucial para que el modelo de lenguaje sepa cómo hacer las consultas SQL correctas.
    - Construye una descripción en texto del esquema de la base de datos. Incluye los nombres de las tablas, los nombres y tipos de datos de las columnas, las claves primarias y las restricciones NOT NULL, y las relaciones de clave externa. Este texto se inserta en el "prompt" para el modelo de lenguaje.
    - Crea una instancia del modelo de lenguaje ChatGroq. Configura parámetros como la temperatura (para controlar la aleatoriedad de las respuestas), el nombre del modelo, la clave de la API y el número máximo de tokens.
    - Define el "prompt" para el modelo de lenguaje. El prompt es una instrucción que le dice al modelo cómo debe comportarse. En este caso, el prompt le indica al modelo que actúe como un experto en bases de datos PostgreSQL y que traduzca preguntas en lenguaje natural a consultas SQL. El prompt también incluye el esquema de la base de datos y reglas específicas para generar consultas SQL correctas y eficientes.
    - Crea el agente SQL utilizando create_sql_agent de LangChain. Le pasa el modelo de lenguaje, la conexión a la base de datos, el prompt y otros parámetros de configuración.
    - Maneja los errores que pueden ocurrir durante la creación del agente.

5. models.py

Este archivo define la estructura de la base de datos utilizando SQLAlchemy.

- **Importaciones:**
    - sqlalchemy: Biblioteca de Python para interactuar con bases de datos SQL. Importa componentes como Column, Integer, String, Date, Numeric, ForeignKey.
    - sqlalchemy.ext.declarative.declarative_base: Función para crear una "base" para las clases que representan tablas de la base de datos.
    - sqlalchemy.orm.relationship: Para definir las relaciones entre las tablas.
    - sqlalchemy.create_engine: Para crear el "engine", que es la conexión a la base de datos.
    - src.config.get_db_uri: Para obtener la URI de la base de datos.

- **Base = declarative_base():** Crea la clase base para los modelos. Todas las clases que representan tablas heredarán de esta clase.
- **engine = create_engine(...):** Crea el "engine" de SQLAlchemy.
    - Le pasa la URI de la base de datos obtenida de src.config.get_db_uri().
    - Configura parámetros como pool_size, max_overflow y pool_pre_ping para manejar eficientemente las conexiones a la base de datos.
- **Clases Modelo:** Define clases que corresponden a las tablas de la base de datos.
    - Cada clase hereda de Base.
    - __tablename__: Especifica el nombre de la tabla en la base de datos.
    - Los atributos de la clase (cliente_id, nombre, email, etc.) representan las columnas de la tabla.
    - Column: Se utiliza para definir cada columna. Se especifica el tipo de datos (Integer, String, Date, Numeric), las restricciones (nullable=False, unique=True, primary_key=True), valores por defecto (server_default='CURRENT_DATE') y claves foráneas (ForeignKey).
    - relationship: Se utiliza para definir las relaciones con otras tablas.
        - back_populates: Crea una relación bidireccional.
        - cascade: Especifica qué debe ocurrir con las filas relacionadas cuando se elimina una fila (por ejemplo, cascade="all, delete-orphan" elimina todos los pedidos de un cliente cuando se elimina el cliente).
    - __repr__: Define cómo se representa un objeto de la clase como una cadena (útil para la depuración).
- **create_tables():** Función para crear todas las tablas en la base de datos. Llama a Base.metadata.create_all(engine) para hacer esto.
- **drop_tables():** Función para eliminar todas las tablas de la base de datos.

6. sql_database.txt

Este archivo contiene el código SQL para crear la base de datos y las tablas, e insertar algunos datos de ejemplo.

3. Detalle del código del archivo models.py.

Este archivo es esencial porque define la estructura de la base de datos utilizando SQLAlchemy, una poderosa biblioteca de Python para interactuar con bases de datos SQL.

1. Estructura General

- **Importaciones:** Se importan las clases y funciones necesarias de SQLAlchemy para definir las tablas y sus relaciones. También se importa la función get_db_uri del módulo src.config para obtener la cadena de conexión a la base de datos.
- **Base = declarative_base():** Se crea una "base" para los modelos.
- **engine = create_engine(...):** Se crea el "motor" de la base de datos.
- **Definición de las Clases Modelo:** Se definen clases de Python que representan las tablas de la base de datos (Cliente, Producto, Pedido, DetallePedido).
- **create_tables() y drop_tables() Functions:** Se definen funciones para crear y eliminar las tablas en la base de datos.

2. Importaciones Detalladas

- **sqlalchemy:**
    - Column, Integer, String, Date, Numeric, ForeignKey: Son clases que se utilizan para definir las columnas de las tablas, especificando sus tipos de datos (entero, cadena, fecha, número decimal) y si son claves foráneas (referencias a otras tablas).
- **sqlalchemy.ext.declarative.declarative_base:**
    - declarative_base: Es una función que crea una "clase base" para las clases de modelo. Todas las clases que representan tablas de la base de datos deben heredar de esta clase base. Esto permite a SQLAlchemy mapear las clases de Python a las tablas de la base de datos.
- **sqlalchemy.orm.relationship:**
    - relationship: Se utiliza para definir las relaciones entre las tablas (por ejemplo, un cliente puede tener muchos pedidos). Esto permite a SQLAlchemy manejar las consultas y las operaciones relacionadas con estas relaciones.
- **sqlalchemy.create_engine:**
    - create_engine: Es una función que crea el "motor" de la base de datos. El motor es el objeto que SQLAlchemy utiliza para comunicarse con la base de datos. Se le pasa la cadena de conexión a la base de datos y otros parámetros de configuración.
- **src.config.get_db_uri:**
    - get_db_uri: Es una función definida en el archivo config.py que construye la cadena de conexión a la base de datos.

3. Base = declarative_base()

- Esta línea crea la clase base para los modelos. Cualquier clase que defina una tabla en la base de datos debe heredar de esta clase. SQLAlchemy utiliza esta clase base para recopilar información sobre las tablas y cómo se relacionan.

4. engine = create_engine(...)

- Esta línea crea el "motor" de la base de datos, que SQLAlchemy utilizará para conectarse a la base de datos.
- get_db_uri(): Obtiene la cadena de conexión a la base de datos desde el archivo config.py. Esta cadena de conexión incluye información como el tipo de base de datos, el nombre de usuario, la contraseña, el host y el nombre de la base de datos.
- pool_size=10: Especifica el número de conexiones que el motor mantendrá abiertas en el "pool" de conexiones. Esto ayuda a mejorar el rendimiento al evitar la necesidad de abrir y cerrar conexiones a la base de datos con frecuencia.
- max_overflow=20: Especifica el número máximo de conexiones que se pueden abrir más allá del pool_size si todas las conexiones del pool están en uso.
- pool_pre_ping=True: Habilita el "pre-ping". SQLAlchemy enviará una señal de "ping" a las conexiones antes de usarlas para asegurarse de que sigan funcionando. Esto ayuda a evitar errores causados por conexiones inactivas.

5. Clases Modelo

A continuación, se definen las clases que representan las tablas de la base de datos. Cada clase hereda de la clase Base y define los atributos que corresponden a las columnas de la tabla.

- **Cliente Class**
    - __tablename__ = 'clientes': Especifica el nombre de la tabla en la base de datos.
    - cliente_id = Column(Integer, primary_key=True, autoincrement=True): Define la columna cliente_id como un entero, clave primaria (identificador único de cada cliente) y con autoincremento (el valor se genera automáticamente al insertar un nuevo cliente).
    - nombre = Column(String(100), nullable=False): Define la columna nombre como una cadena de texto de hasta 100 caracteres, y no puede ser nula (es decir, es obligatorio proporcionar un nombre para cada cliente).
    - email = Column(String(100), unique=True, nullable=False): Define la columna email también como una cadena de texto de hasta 100 caracteres, debe ser única (no puede haber dos clientes con el mismo email) y no puede ser nula.

        - fecha_registro = Column(Date, server_default='CURRENT_DATE'): Define la columna fecha_registro como una fecha, y su valor por defecto es la fecha actual.
        - pedidos = relationship("Pedido", back_populates="cliente", cascade="all, delete-orphan"): Define la relación con la tabla Pedido. Un cliente puede tener muchos pedidos (relación uno a muchos).
            - back_populates="cliente": Define el atributo en la clase Pedido que hace referencia a esta relación.
            - cascade="all, delete-orphan": Configura el comportamiento en cascada. Si se elimina un cliente, todos sus pedidos relacionados también se eliminarán.
        - __repr__(self): Define cómo se representa un objeto Cliente como una cadena de texto (útil para depurar).
- **Producto Class**
    - Similar a Cliente, define las columnas de la tabla productos (producto_id, nombre, precio, categoria) y la relación con DetallePedido.
    - precio = Column(Numeric(10, 2), nullable=False): Utiliza Numeric para almacenar valores decimales con precisión (10 dígitos en total, 2 decimales).
- **Pedido Class**
    - Define las columnas de la tabla pedidos (pedido_id, cliente_id, fecha_pedido, estado).
    - cliente_id = Column(Integer, ForeignKey('clientes.cliente_id'), nullable=False): cliente_id es una clave foránea que referencia la columna cliente_id de la tabla clientes. Esto establece la relación entre clientes y pedidos.
    - Define las relaciones con Cliente y DetallePedido.
- **DetallePedido Class**
    - Define las columnas de la tabla detalles_pedido (detalle_id, pedido_id, producto_id, cantidad, precio_unitario).
    - pedido_id = Column(Integer, ForeignKey('pedidos.pedido_id'), nullable=False) y producto_id = Column(Integer, ForeignKey('productos.producto_id'), nullable=False): pedido_id y producto_id son claves foráneas que referencian las tablas pedidos y productos, respectivamente, estableciendo las relaciones necesarias.
    - Define las relaciones con Pedido y Producto.

6. create_tables() Function

- Base.metadata.create_all(engine): Esta función crea todas las tablas definidas como clases en este archivo en la base de datos. Utiliza el engine que se configuró anteriormente para

conectarse a la base de datos y los metadatos de la Base para obtener información sobre las tablas a crear.

7. drop_tables() Function

- Base.metadata.drop_all(engine): Esta función elimina todas las tablas de la base de datos. Se utiliza principalmente durante el desarrollo para limpiar la base de datos y volver a crearla desde cero.

En resumen, el archivo models.py define la estructura de la base de datos de manera declarativa utilizando las clases de SQLAlchemy. Esto permite que el resto de la aplicación interactúe con la base de datos en términos de objetos de Python, y SQLAlchemy se encarga de traducir esas operaciones a las correspondientes sentencias SQL.

4. Desglosando la lógica de programación del archivo chatbot.py

Paso a paso. Este archivo es el núcleo del chatbot, ya que coordina la interacción entre el usuario, el agente de LangChain (para traducir lenguaje natural a SQL) y la base de datos.

1. Estructura General

- **Importaciones:** Primero, se importan las bibliotecas y módulos necesarios.
- **Configuración de Logging:** Se configura el sistema de registro para guardar información sobre el funcionamiento del programa y los errores que puedan ocurrir.
- **Clase ChatbotSQL:** Aquí se define la lógica principal del chatbot. Esta clase tiene métodos para inicializar el chatbot, procesar las consultas del usuario, ejecutar consultas SQL y formatear las respuestas.

2. Importaciones Detalladas

- typing: Se utiliza para definir los tipos de datos de las variables y funciones. Esto ayuda a que el código sea más claro y fácil de entender, y también permite que las herramientas de desarrollo detecten posibles errores.
    - Optional, Dict, Any, List, Tuple: Son tipos de datos que especifican si una variable puede ser None, diccionarios, cualquier tipo de dato, listas o tuplas, respectivamente.
- src.langchain_setup: Importa el módulo langchain_setup, que contiene la función setup_sql_agent. Esta función se encarga de configurar el agente de LangChain que traduce el lenguaje natural a SQL.
- src.database: Importa el módulo database, que contiene la clase DatabaseManager. Esta clase se encarga de gestionar la conexión a la base de datos y ejecutar las consultas SQL.

- langchain_community.utilities.sql_database.SQLDatabase: Importa la clase SQLDatabase de LangChain. Esta clase se utiliza para establecer la conexión entre LangChain y la base de datos SQL.
- src.config: Importa el módulo config, que contiene la función get_db_uri. Esta función obtiene la cadena de conexión a la base de datos.
- re: Importa el módulo re, que permite utilizar expresiones regulares. Las expresiones regulares son patrones que se utilizan para buscar y manipular texto. En este caso, se utiliza para extraer la consulta SQL generada por el agente de LangChain.
- logging: Importa el módulo logging, que permite registrar mensajes informativos y de error. Esto es útil para depurar el código y entender cómo está funcionando el programa.

3. Configuración de Logging

- logging.basicConfig(level=logging.INFO): Configura el sistema de registro para que guarde todos los mensajes de nivel INFO o superior (es decir, INFO, WARNING, ERROR y CRITICAL).
- logger = logging.getLogger(__name__): Crea un "logger" (registrador) para este módulo. El __name__ es una variable especial que contiene el nombre del módulo actual. Esto permite que los mensajes de registro indiquen de qué parte del código provienen.

4. Clase ChatbotSQL

- __init__(self): Este es el constructor de la clase. Se ejecuta cuando se crea un nuevo objeto ChatbotSQL.
    - self.db_manager = DatabaseManager(): Crea una instancia de la clase DatabaseManager, que se utilizará para gestionar la conexión a la base de datos.
    - self.agent = None: Inicializa la variable self.agent a None. Esta variable almacenará el agente de LangChain que se utiliza para traducir el lenguaje natural a SQL.
    - self.sql_db = None: Inicializa la variable self.sql_db a None. Esta variable almacenará la conexión de LangChain a la base de datos SQL.
    - **Bloque try...except:** Este bloque se utiliza para manejar posibles errores durante la inicialización del chatbot.
        - if not self.db_manager.connect(): raise RuntimeError("No se pudo conectar a PostgreSQL"): Intenta conectar a la base de datos utilizando el método connect() del DatabaseManager. Si la conexión falla, se levanta una excepción RuntimeError.
        - self.sql_db = SQLDatabase.from_uri(get_db_uri()): Crea una instancia de SQLDatabase de LangChain utilizando la URI de conexión obtenida de la función get_db_uri().

- self.agent = setup_sql_agent(self.db_manager): Inicializa el agente SQL llamando a la función setup_sql_agent() y pasándole el DatabaseManager.
- logger.info("Chatbot SQL inicializado correctamente"): Si todo va bien, registra un mensaje informativo.
- except Exception as e: Si ocurre cualquier excepción dentro del bloque try, se ejecuta este bloque.
    - logger.error(f"Error durante la inicialización: {e}"): Registra un mensaje de error con la descripción de la excepción.
    - self._cleanup_resources(): Llama al método _cleanup_resources() para liberar cualquier recurso que se haya podido haber asignado antes del error.
    - raise RuntimeError("No se pudo inicializar el chatbot") from e: Vuelve a levantar la excepción, indicando que no se pudo inicializar el chatbot. El from e encadena la excepción original para facilitar la depuración.

- process_query(self, user_input: str) -> Dict[str, Any]: Este método procesa una consulta del usuario.
    - response = { ... }: Inicializa un diccionario response con los valores por defecto para la respuesta. Este diccionario se devolverá al usuario.
    - **Bloque try...except:** Maneja posibles errores al procesar la consulta.
        - if user_input.strip().lower().startswith("sql:"): Verifica si la consulta del usuario comienza con "sql:". Si es así, se asume que el usuario está introduciendo una consulta SQL directa.
            - query = user_input[4:].strip(): Extrae la consulta SQL de la entrada del usuario, eliminando el prefijo "sql:" y los espacios en blanco al principio y al final.
            - columns, data = self._execute_direct_query(query): Ejecuta la consulta SQL directamente utilizando el método _execute_direct_query().
            - Si la consulta devuelve resultados (if columns and data), actualiza el diccionario response con los resultados y la consulta.
        - else: Si la consulta no comienza con "sql:", se asume que es una consulta en lenguaje natural.

- agent_response = self.agent.invoke({"input": user_input}): Envía la consulta al agente de LangChain para que la traduzca a SQL y la ejecute.
- sql_query = self._extract_sql_query(agent_response): Extrae la consulta SQL generada por el agente de la respuesta.
- Actualiza el diccionario response con la respuesta del agente y la consulta SQL.

- except Exception as e: Si ocurre algún error durante el procesamiento de la consulta, se ejecuta este bloque.
  - Registra el error y actualiza el diccionario response con un mensaje de error.

- Devuelve el diccionario response.

- _execute_direct_query(self, query: str) -> Tuple[Optional[List[str]], Optional[List[Dict]]]: Ejecuta una consulta SQL directamente en la base de datos.
  - Llama al método execute_query() del DatabaseManager para ejecutar la consulta.
  - Maneja posibles excepciones y devuelve los resultados o None en caso de error.
- _format_results(self, columns: List[str], data: List[Dict]) -> str: Formatea los resultados de la consulta como una tabla HTML.
  - Construye una tabla HTML a partir de los nombres de las columnas y los datos de las filas.
  - Si no hay resultados, devuelve un mensaje indicándolo.
- _extract_sql_query(self, agent_response: Dict[str, Any]) -> Optional[str]: Extrae la consulta SQL generada por el agente de LangChain de la respuesta del agente.
  - Utiliza expresiones regulares (re.findall) para buscar el bloque de código SQL dentro de la respuesta del agente.
  - Devuelve la consulta SQL extraída o None si no se encuentra.
- _cleanup_resources(self): Libera los recursos utilizados por el chatbot.
  - Cierra la conexión a la base de datos llamando al método close() del DatabaseManager.
  - Maneja posibles excepciones durante el cierre de la conexión.
- close(self): Cierra limpiamente todos los recursos del chatbot.
  - Llama a _cleanup_resources() para cerrar la conexión a la base de datos.
  - Registra un mensaje indicando que el chatbot se cerró correctamente.

- __enter__(self) y __exit__(self, exc_type, exc_val, exc_tb): Estos métodos permiten utilizar el chatbot en un bloque with. Esto asegura que los recursos se liberen automáticamente cuando se sale del bloque with.

En resumen, la lógica de chatbot.py se centra en inicializar los componentes necesarios para interactuar con la base de datos a través de lenguaje natural o SQL directo, procesar las consultas de los usuarios de manera robusta, formatear los resultados de forma presentable y gestionar adecuadamente los recursos del sistema.

5. Lógica de programación del archivo langchain_setup.py.

Este archivo es crucial porque configura el "agente" de LangChain, que actúa como el intermediario inteligente entre el usuario y la base de datos SQL. En esencia, este agente toma las preguntas del usuario en lenguaje natural y las traduce a consultas SQL que la base de datos puede entender.

1. Estructura General

- **Importaciones:** Se importan las bibliotecas y módulos necesarios para trabajar con LangChain, el modelo de lenguaje Groq, la configuración de la aplicación y el registro de eventos.
- **setup_sql_agent(db_manager) Function:** Esta función es el corazón del archivo. Se encarga de crear y configurar el agente SQL de LangChain.

2. Importaciones Detalladas

- langchain_community.agent_toolkits.create_sql_agent: Importa la función create_sql_agent de LangChain. Esta función se utiliza para crear un agente especializado en interactuar con bases de datos SQL. Proporciona las herramientas y la lógica necesarias para que el agente pueda ejecutar consultas y obtener resultados.
- langchain_groq.ChatGroq: Importa la clase ChatGroq. Esta clase permite utilizar el modelo de lenguaje de Groq (como Groq's Large Language Model) dentro de LangChain. Los modelos de lenguaje son los que tienen la capacidad de entender y generar lenguaje natural, y en este caso, se utilizan para traducir las preguntas del usuario a SQL.
- langchain_core.prompts.ChatPromptTemplate: Importa la clase ChatPromptTemplate. En LangChain, un "prompt" es una instrucción o una plantilla que se le da al modelo de lenguaje para indicarle cómo debe comportarse. ChatPromptTemplate se utiliza específicamente para crear prompts para modelos de chat, permitiendo estructurar la interacción con el modelo.
- langchain_community.utilities.sql_database.SQLDatabase: Importa la clase SQLDatabase de LangChain. Esta clase proporciona una interfaz para conectarse a una base de datos SQL y

obtener información sobre su estructura (el "esquema"). El esquema es esencial para que el agente pueda generar consultas SQL válidas.

- src.config: Importa el módulo src.config. Este módulo (que corresponde al archivo config.py) contiene variables de configuración importantes, como la clave de la API para acceder al modelo de Groq (GROQ_API_KEY), el nombre del modelo a utilizar (MODEL_NAME) y la cadena de conexión para la base de datos (get_db_uri).
- logging: Importa el módulo logging. Esto permite registrar mensajes informativos, de advertencia o de error durante la ejecución del código, lo cual es muy útil para depurar y entender el comportamiento del programa.
- typing: Importa el módulo typing. Esto se utiliza para proporcionar "anotaciones de tipo" en el código, lo que ayuda a especificar qué tipo de datos se espera que tengan las variables y las funciones. Mejora la legibilidad y mantenibilidad del código.

3. setup_sql_agent(db_manager) Function

- **Función Principal:** Esta función es responsable de configurar y crear el agente de LangChain que interactúa con la base de datos SQL. Recibe como argumento db_manager, que es una instancia de la clase DatabaseManager (definida en database.py). El db_manager es el objeto que se encarga de la conexión a la base de datos.
- **if not GROQ_API_KEY::** Verifica si la variable GROQ_API_KEY (obtenida del archivo config.py) está configurada. Si no lo está, se lanza una excepción ValueError porque la clave de la API es necesaria para acceder al modelo de lenguaje de Groq.
- **try...except Block (Creación de SQLDatabase)**
    - db = SQLDatabase.from_uri(get_db_uri()): Intenta crear una instancia de la clase SQLDatabase de LangChain. Se le pasa la cadena de conexión a la base de datos, que se obtiene llamando a la función get_db_uri() (del archivo config.py). Esto establece la conexión de LangChain con la base de datos.
    - except Exception as e:: Si ocurre algún error durante la creación de la instancia de SQLDatabase (por ejemplo, si la conexión a la base de datos falla), se captura la excepción, se registra un mensaje de error utilizando el módulo logging y la función devuelve None para indicar que no se pudo crear la conexión.
- **Obtener el Esquema de la Base de Datos**
    - schema = db_manager.get_database_schema(): Llama al método get_database_schema() del objeto db_manager para obtener el esquema de la base de datos. El esquema describe la estructura de la base de datos: qué tablas hay, qué columnas tiene cada tabla, qué tipos de datos tienen las columnas, etc. Esta

información es crucial para que el agente de LangChain pueda generar consultas SQL válidas.

- if not schema:: Verifica si se pudo obtener el esquema. Si schema está vacío (o es None), se registra un mensaje de error y la función devuelve None.

- **Construir la Descripción del Esquema**
  - schema_lines = ["Esquema de la base de datos:"]: Inicializa una lista llamada schema_lines. Esta lista se utilizará para construir una representación en texto del esquema de la base de datos.
  - **Bucle for table_name, table_info in schema.items()::** Itera sobre las tablas de la base de datos. schema es un diccionario donde las claves son los nombres de las tablas y los valores son diccionarios que contienen información sobre cada tabla.
    - schema_lines.append(f"\nTabla: {table_name}\nColumnas:"): Agrega el nombre de la tabla a la lista schema_lines.
    - **Bucle for column in table_info['columns']::** Itera sobre las columnas de la tabla actual.
      - desc = f"- {column['name']}: {column['type']}": Crea una descripción de la columna, incluyendo su nombre y tipo de datos.
      - if column['primary_key']:: Verifica si la columna es una clave primaria. Si lo es, agrega "(PRIMARY KEY)" a la descripción.
      - if not column['nullable']:: Verifica si la columna no permite valores nulos. Si es así, agrega "NOT NULL" a la descripción.
      - schema_lines.append(desc): Agrega la descripción de la columna a la lista schema_lines.
    - **Bucle for fk in table_info['foreign_keys']::** Itera sobre las claves foráneas de la tabla actual (relaciones con otras tablas).
      - schema_lines.append(f" Claves Foráneas: {fk['constrained_columns']} -> {fk['referred_table']}"): Agrega información sobre las claves foráneas a la lista schema_lines, indicando qué columnas están relacionadas con qué tablas.
  - schema_description = "\n".join(schema_lines): Combina todas las líneas de la lista schema_lines en una sola cadena de texto, separándolas por saltos de línea. Esta cadena schema_description contiene una descripción completa y legible del esquema de la base de datos.

- **Crear el Modelo de Lenguaje (LLM)**
    - llm = ChatGroq(temperature=0, model_name=MODEL_NAME, groq_api_key=GROQ_API_KEY, max_tokens=1024): Crea una instancia del modelo de lenguaje ChatGroq.
        - temperature=0: Configura la "temperatura" del modelo a 0. La temperatura controla la aleatoriedad de las respuestas del modelo. Un valor de 0 hace que el modelo sea muy determinista, es decir, que siempre dé la misma respuesta a la misma pregunta. Esto es importante para las consultas SQL, donde se necesita precisión.
        - model_name=MODEL_NAME: Especifica el nombre del modelo de lenguaje que se va a utilizar (obtenido de config.py).
        - groq_api_key=GROQ_API_KEY: Proporciona la clave de la API para acceder al servicio Groq (obtenida de config.py).
        - max_tokens=1024: Establece el número máximo de "tokens" (palabras o partes de palabras) que el modelo puede generar en una sola respuesta.
- **Crear el Prompt para el LLM**
    - prompt = ChatPromptTemplate.from_messages(...): Crea un "prompt" (instrucción) para el modelo de lenguaje. Este prompt le indica al modelo cómo debe comportarse y qué tipo de respuestas debe generar.
        - El prompt está formado por una lista de "mensajes", donde cada mensaje tiene un tipo ("system", "human" o "placeholder") y un contenido.
        - ("system", ...): El mensaje de tipo "system" proporciona instrucciones generales al modelo. En este caso, le indica que actúe como un experto en bases de datos PostgreSQL y que traduzca preguntas en lenguaje natural a consultas SQL. También le proporciona el esquema de la base de datos (schema_description) y una serie de reglas para generar consultas SQL correctas y eficientes.
        - ("human", "{input}"): El mensaje de tipo "human" representa la pregunta del usuario. {input} es una variable que se reemplazará por la pregunta real del usuario.
        - ("placeholder", "{agent_scratchpad}"): El mensaje de tipo "placeholder" se utiliza para la comunicación interna del agente de LangChain. {agent_scratchpad} es una variable donde el agente puede guardar información temporal durante su proceso de razonamiento.

- **Crear el Agente SQL**
    - try...except Block: Intenta crear el agente SQL y maneja posibles errores.
        - agent = create_sql_agent(...): Crea el agente SQL utilizando la función create_sql_agent de LangChain.
            - llm=llm: Le pasa el modelo de lenguaje que se creó anteriormente.
            - db=db: Le pasa la conexión a la base de datos (SQLDatabase).
            - prompt=prompt: Le pasa el prompt que se creó anteriormente.
            - agent_type="tool-calling": Especifica el tipo de agente que se va a utilizar. "tool-calling" es un tipo de agente que puede utilizar "herramientas" (en este caso, la herramienta para ejecutar consultas SQL).
            - verbose=True: Activa el modo "verbose", lo que hace que el agente imprima información detallada sobre su proceso de razonamiento (útil para la depuración).
            - handle_parsing_errors=True: Indica al agente que debe intentar manejar los errores de análisis (parsing) que puedan ocurrir al ejecutar las consultas SQL.
            - max_iterations=5: Establece el número máximo de "iteraciones" que el agente puede realizar en una sola ejecución. Esto evita que el agente se quede en un bucle infinito.
        - return agent: Si el agente se crea correctamente, la función lo devuelve.
        - except Exception as e:: Si ocurre algún error durante la creación del agente, se captura la excepción, se registra un mensaje de error y la función devuelve None.

En resumen, la lógica de langchain_setup.py es orquestar la creación de un agente de LangChain capaz de interactuar con una base de datos SQL. Esto implica configurar la conexión a la base de datos, proporcionar al modelo de lenguaje la información necesaria sobre la estructura de la base de datos y darle instrucciones claras sobre cómo traducir el lenguaje natural a consultas SQL.