```
import json
import os
import urllib.request
import logging
import random  # Import the random module
from pathlib import Path
from textblob import TextBlob
from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer
from botbuilder.core import TurnContext, MessageFactory
from botbuilder.schema import Activity, ActivityTypes, EndOfConversationCodes
from tenacity import retry, wait_random_exponential, stop_after_attempt
import httpx
import openai
from dotenv import load_dotenv
from database import connect_to_database
from chat import (
    list_fine_tuning_jobs,
    upload_file_for_fine_tuning,
    create_fine_tuning_job,
    make_post_request,
    azure_chat_completion_request,
)
import tkinter as tk
from tkinter import messagebox

def show_privacy_consent():
```

```
    """Display a pop-up window to obtain user consent for data collection and privacy."""
    def on_accept():
        user_consent.set(True)
        root.destroy()

    def on_decline():
        user_consent.set(False)
        root.destroy()

    root = tk.Tk()
    root.title("Data Permission and Privacy")

     message = ("We value your privacy. By using this application, you consent to the collection and use of your data "
                "as described in our privacy policy. Do you agree to proceed?")
    label = tk.Label(root, text=message, wraplength=400, justify="left")
    label.pack(padx=20, pady=20)

    button_frame = tk.Frame(root)
    button_frame.pack(pady=10)

    accept_button = tk.Button(button_frame, text="Accept", command=on_accept)
    accept_button.pack(side="left", padx=10)

    decline_button = tk.Button(button_frame, text="Decline", command=on_decline)
    decline_button.pack(side="right", padx=10)
```

```
    user_consent = tk.BooleanVar()
    root.mainloop()

    return user_consent.get()

# Load environment variables from .env file
load_dotenv()

# Validate environment variables
openai_api_key = os.getenv('OPENAI_API_KEY')
azure_openai_api_key = os.getenv('AZURE_OPENAI_API_KEY')
azure_openai_endpoint = os.getenv('AZURE_OPENAI_ENDPOINT')

if not openai_api_key:
    logging.error("OpenAI API key not found in environment variables.")
if not azure_openai_api_key or not azure_openai_endpoint:
    logging.error("Azure OpenAI API key or endpoint not found in environment variables.")

# Set your OpenAI API key
openai.api_key = openai_api_key

# Configure logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')

def download_database(url: str, file_path: str) -> None:
```

```
    """Download the database file from the given URL."""
    try:
        logging.info(f"Downloading database from {url}...")
        urllib.request.urlretrieve(url, file_path)
        logging.info("Download complete.")
    except urllib.error.URLError as e:
        logging.error(f"Error: Failed to download database. {e}")

# Database connection
db_path = "data/Chinook.db"
db_url =
"https://github.com/lerocha/chinook-database/raw/master/ChinookDatabase/DataSources/Chinook_
Sqlite.sqlite"
if not os.path.exists(db_path):
    os.makedirs(os.path.dirname(db_path), exist_ok=True)
    download_database(db_url, db_path)
conn = connect_to_database(db_path)
if not conn:
    logging.error("Failed to connect to the database.")

# Sentiment analysis functions
def analyze_sentiment_textblob(text: str) -> TextBlob:
    """Analyze the sentiment of the given text using TextBlob."""
    blob = TextBlob(text)
    sentiment = blob.sentiment
    return sentiment
```

```
def analyze_sentiment_vader(text: str) -> dict:
    """Analyze the sentiment of the given text using VADER."""
    analyzer = SentimentIntensityAnalyzer()
    sentiment = analyzer.polarity_scores(text)
    return sentiment

async def end_conversation(turn_context: TurnContext) -> None:
    """Ends the conversation with the user."""
    await turn_context.send_activity(
        MessageFactory.text("Ending conversation from the skill...")
    )
    end_of_conversation = Activity(type=ActivityTypes.end_of_conversation)
    end_of_conversation.code = EndOfConversationCodes.completed_successfully
    await turn_context.send_activity(end_of_conversation)

async def handle_error(turn_context: TurnContext, error: Exception) -> None:
    """Handles errors by logging them and notifying the user."""
    logging.error(f"An error occurred: {error}")
    await turn_context.send_activity(
        MessageFactory.text("An error occurred. Please try again later.")
    )

class MyBot:
    def __init__(self):
        self.context = {}
```

```
    async def enhance_context_awareness(self, user_id: str, text: str) -> None:
        """Enhance context awareness by analyzing the user's environment, activities, and emotional state."""
        sentiment = analyze_sentiment_vader(text)
        self.context[user_id].append({"text": text, "sentiment": sentiment})

    async def proactive_learning(self, user_id: str, feedback: str) -> None:
        """Encourage proactive learning by seeking feedback and exploring new topics."""
        self.context[user_id].append({"feedback": feedback})

    async def ethical_decision_making(self, user_id: str, decision: str) -> None:
        """Integrate ethical principles into decision-making processes."""
        ethical_decision = f"Considering ethical principles, the decision is: {decision}"
        self.context[user_id].append({"ethical_decision": ethical_decision})

    async def emotional_intelligence(self, user_id: str, text: str) -> str:
        """Develop emotional intelligence by recognizing and responding to user emotions."""
        sentiment = analyze_sentiment_vader(text)
        response = f"I sense that you are feeling {sentiment['compound']}. How can I assist you further?"
        self.context[user_id].append({"emotional_response": response})
        return response

    async def transparency_and_explainability(self, user_id: str, decision: str) -> str:
        """Enable transparency by explaining the reasoning behind decisions."""
```

```
    explanation = f"The decision was made based on the following context: {self.context[user_id]}"
    self.context[user_id].append({"explanation": explanation})
    return explanation

async def on_message_activity(self, turn_context: TurnContext) -> None:
    """Handles incoming messages and generates responses."""
    user_id = turn_context.activity.from_property.id
    if user_id not in self.context:
        self.context[user_id] = []
    try:
        if "end" in turn_context.activity.text or "stop" in turn_context.activity.text:
            await end_conversation(turn_context)
            self.context.pop(user_id, None)
        else:
            self.context[user_id].append(turn_context.activity.text)
            response = await self.generate_response(turn_context.activity.text, user_id)
            await turn_context.send_activity(MessageFactory.text(response))
    except Exception as e:
        await handle_error(turn_context, e)

async def generate_response(self, text: str, user_id: str) -> str:
    """Generates a response using OpenAI's API."""
    try:
        logging.info(f"Generating response for user_id: {user_id} with text: {text}")
        response = openai.Completion.create(
            engine="text-davinci-003",
```

```
            prompt=f"User: {text}\nContext: {self.context[user_id]}\nBot:",
            max_tokens=150
        )
        logging.info(f"OpenAI response: {response}")
        return response.choices[0].text.strip()
    except openai.error.OpenAIError as e:
        logging.error(f"Error generating response: {e}")
        return "Sorry, I couldn't generate a response at this time."

# Example usage of MyBot class
bot = MyBot()

# Functions based on JSON configuration
def newton_thoughts(question: str) -> str:
    """Apply Newton's laws to the given question."""
    return apply_newtons_laws(question)

def apply_newtons_laws(question: str) -> str:
    """Apply Newton's laws to the given question."""
    if not question:
        return 'No question to think about.'
    complexity = len(question)
    force = mass_of_thought(question) * acceleration_of_thought(complexity)
    return f'Thought force: {force}'

def mass_of_thought(question: str) -> int:
```

```
    """Calculate the mass of thought based on the question length."""
    return len(question)

def acceleration_of_thought(complexity: int) -> float:
    """Calculate the acceleration of thought based on the complexity."""
    return complexity / 2

def davinci_insights(question: str) -> str:
    """Generate insights like Da Vinci for the given question."""
    return think_like_davinci(question)

def think_like_davinci(question: str) -> str:
    """Generate insights like Da Vinci for the given question."""
    perspectives = [
        f"What if we view '{question}' from the perspective of the stars?",
        f"Consider '{question}' as if it's a masterpiece of the universe.",
        f"Reflect on '{question}' through the lens of nature's design."
    ]
    return random.choice(perspectives)

def human_intuition(question: str) -> str:
    """Provide human intuition for the given question."""
    intuition = [
        "How does this question make you feel?",
        "What emotional connection do you have with this topic?",
        "What does your gut instinct tell you about this?"
```

```
    ]
    return random.choice(intuition)

def neural_network_thinking(question: str) -> str:
    """Apply neural network thinking to the given question."""
    neural_perspectives = [
        f"Process '{question}' through a multi-layered neural network.",
        f"Apply deep learning to uncover hidden insights about '{question}'.",
        f"Use machine learning to predict patterns in '{question}'."
    ]
    return random.choice(neural_perspectives)

def quantum_computing_thinking(question: str) -> str:
    """Apply quantum computing principles to the given question."""
    quantum_perspectives = [
        f"Consider '{question}' using quantum superposition principles.",
        f"Apply quantum entanglement to find connections in '{question}'.",
        f"Utilize quantum computing to solve '{question}' more efficiently."
    ]
    return random.choice(quantum_perspectives)

def resilient_kindness(question: str) -> str:
    """Provide perspectives of resilient kindness."""
    kindness_perspectives = [
        "Despite losing everything, seeing life as a chance to grow.",
        "Finding strength in kindness after facing life's hardest trials.",
```

```
        "Embracing every challenge as an opportunity for growth and compassion."
    ]
    return random.choice(kindness_perspectives)


def identify_and_refute_fallacies(argument: str) -> str:
    """Identify and refute common logical fallacies in the argument."""
    fallacies = [
        "Ad Hominem",
        "Straw Man",
        "False Dilemma",
        "Slippery Slope",
        "Circular Reasoning",
        "Hasty Generalization",
        "Red Herring",
        "Post Hoc Ergo Propter Hoc",
        "Appeal to Authority",
        "Bandwagon Fallacy",
        "False Equivalence"
    ]
    refutations = [
        "This is an ad hominem fallacy. Let's focus on the argument itself rather than attacking the person.",
        "This is a straw man fallacy. The argument is being misrepresented.",
        "This is a false dilemma fallacy. There are more options than presented.",
        "This is a slippery slope fallacy. The conclusion does not necessarily follow from the premise.",
        "This is circular reasoning. The argument's conclusion is used as a premise.",
```

```
        "This is a hasty generalization. The conclusion is based on insufficient evidence.",
        "This is a red herring fallacy. The argument is being diverted to an irrelevant topic.",
        "This is a post hoc ergo propter hoc fallacy. Correlation does not imply causation.",
        "This is an appeal to authority fallacy. The argument relies on the opinion of an authority figure.",
        "This is a bandwagon fallacy. The argument assumes something is true because many people believe it.",
        "This is a false equivalence fallacy. The argument equates two things that are not equivalent."
    ]
    return random.choice(refutations)

def universal_reasoning(question: str) -> str:
    """Generate a comprehensive response using various reasoning methods."""
    responses = [
        newton_thoughts(question),
        davinci_insights(question),
        human_intuition(question),
        neural_network_thinking(question),
        quantum_computing_thinking(question),
        resilient_kindness(question),
        identify_and_refute_fallacies(question)
    ]
    return "\n".join(responses)

def stream_thread_responses(thread_id: str, assistant_id: str) -> None:
    """Stream thread responses from OpenAI."""
```

```
    client = openai.OpenAI(api_key=os.environ.get('OPENAI_API_KEY'))
                with client.beta.threads.runs.stream(thread_id=thread_id, assistant_id=assistant_id,
instructions='Please address the user as Jane Doe. The user has a premium account.') as stream:
        for event in stream:
            if event.type == 'thread.message.delta' and event.data.delta.content:
                print(event.data.delta.content[0].text)


@retry(wait=wait_random_exponential(min=1, max=40), stop=stop_after_attempt(3))
def chat_completion_request(messages: list, model: str = "gpt-4") -> str:
    """Make a chat completion request to Azure OpenAI."""
    try:
        headers = {
            "Content-Type": "application/json",
            "api-key": azure_openai_api_key
        }
        payload = {
            "model": model,
            "messages": messages
        }
        response = httpx.post(azure_openai_endpoint, headers=headers, json=payload)
        response.raise_for_status()
        return response.json()["choices"][0]["message"]["content"].strip()
    except httpx.HTTPStatusError as e:
        logging.error("Unable to generate ChatCompletion response")
        logging.error(f"Exception: {e}")
        return str(e)
```

```
def get_internet_answer(question: str) -> str:
    """Get an answer from the internet using chat completion request."""
    messages = [
        {"role": "system", "content": "You are a helpful assistant."},
        {"role": "user", "content": question}
    ]
    return chat_completion_request(messages)


def reflect_on_decisions() -> str:
     """Regularly reflect on your decisions, the processes you used, the information you considered, and the perspectives you may have missed."""
    reflection_message = (
            "Regularly reflecting on your decisions, the processes you used, the information you considered, "
          "and the perspectives you may have missed. Reflection is a cornerstone of learning from experience."
    )
    return reflection_message


def process_questions_from_json(file_path: str):
    """Process questions from a JSON file and call the appropriate functions."""
    with open(file_path, 'r') as file:
        questions_data = json.load(file)


    for question_data in questions_data:
```

```
        question = question_data['question']
        print(f"Question: {question}")

        for function_data in question_data['functions']:
            function_name = function_data['name']
            function_description = function_data['description']
            function_parameters = function_data['parameters']

            print(f"Function: {function_name}")
            print(f"Description: {function_description}")

            # Call the function dynamically
            if function_name in globals():
                function = globals()[function_name]
                response = function(**function_parameters)
                print(f"Response: {response}")
            else:
                print(f"Function {function_name} not found.")

if __name__ == "__main__":
    if show_privacy_consent():
        process_questions_from_json('questions.json')
        question = "What is the meaning of life?"
        print("Newton's Thoughts:", newton_thoughts(question))
        print("Da Vinci's Insights:", davinci_insights(question))
        print("Human Intuition:", human_intuition(question))
```

```
    print("Neural Network Thinking:", neural_network_thinking(question))
    print("Quantum Computing Thinking:", quantum_computing_thinking(question))
    print("Resilient Kindness:", resilient_kindness(question))
    print("Universal Reasoning:", universal_reasoning(question))
    print("Internet Answer:", get_internet_answer(question))
else:
    print("User did not consent to data collection. Exiting application.")

print(reflect_on_decisions())
```