```
import gc
import os
import threading
from typing import List, Optional

import torch
import uvicorn
import whisperx
from dotenv import load_dotenv
from fastapi import FastAPI
from fastapi.concurrency import asynccontextmanager
from fastapi.middleware.cors import CORSMiddleware
from pydantic import BaseModel

load_dotenv()

hf_token = os.getenv("HF_TOKEN")


class WhisperTranscriber:
    """
    A class for transcribing audio using the Whisper ASR system.

    Args:
        device (str): The device to use for computation (default: "cuda").
        compute_type (str): The compute type to use (default: "float16").
```

```
    batch_size (int): The batch size for transcription (default: 16).

    hf_token (Optional[str]): The Hugging Face authentication token (default: None).

    audio_file (Optional[str]): The path to the audio file to transcribe (default: None).

    audio_files (Optional[List[str]]): A list of paths to audio files to transcribe (default: None).
"""

def __init__(
    self,
    device: str = "cuda",
    compute_type: str = "float16",
    batch_size: int = 16,
    hf_token: Optional[str] = hf_token,
    audio_file: Optional[str] = None,
    audio_files: Optional[List[str]] = None,
):
    self.device = device
    self.compute_type = compute_type
    self.batch_size = batch_size
    self.hf_token = hf_token
    self.lock = threading.Lock()
    self.audio_file = audio_file
    self.audio_files = audio_files

def load_and_transcribe(self, audio_file):
    """
    Load the Whisper ASR model and transcribe the audio file.
```

```
    Args:
        audio_file (str): The path to the audio file.

    Returns:
        dict: The transcription result.
    """
    with self.lock:
        model = whisperx.load_model(
            "large-v2", self.device, compute_type=self.compute_type
        )

    audio = whisperx.load_audio(audio_file)
    result = model.transcribe(audio, batch_size=self.batch_size)
    print(result["segments"])  # Before alignment

    with self.lock:
        del model
        gc.collect()
        torch.cuda.empty_cache()

    return result

def align(self, segments, language_code):
    """
    Align the transcribed segments with the audio using the Whisper alignment model.
```

```
Args:
    segments (list): The transcribed segments.
    language_code (str): The language code.

Returns:
    dict: The alignment result.
"""
with self.lock:
    model_a, metadata = whisperx.load_align_model(
        language_code=language_code, device=self.device
    )

audio = whisperx.load_audio(self.audio_file)
result = whisperx.align(
    segments,
    model_a,
    metadata,
    audio,
    self.device,
    return_char_alignments=False,
)
print(result["segments"])  # After alignment

with self.lock:
    del model_a
```

```
        gc.collect()
        torch.cuda.empty_cache()

    return result

def diarize_and_assign(self, audio_file, segments):
    """
    Diarize the audio and assign speaker IDs to the segments.

    Args:
        audio_file (str): The path to the audio file.
        segments (list): The aligned segments.

    Returns:
        dict: The diarization and assignment result.
    """
    with self.lock:
        diarize_model = whisperx.DiarizationPipeline(
            use_auth_token=self.hf_token, device=self.device
        )

    diarize_segments = diarize_model(audio_file)
    result = whisperx.assign_word_speakers(diarize_segments, segments)
    print(diarize_segments)
    print(result["segments"])  # Segments now assigned speaker IDs
```

```
        return result

    def process_audio(self, audio_file: str):
        """
        Process the audio file by transcribing, aligning, and diarizing it.

        Args:
            audio_file (str): The path to the audio file.

        Returns:
            dict: The final result.
        """
        transcription_result = self.load_and_transcribe(audio_file)
        aligned_result = self.align(
            transcription_result["segments"], transcription_result["language"]
        )
        final_result = self.diarize_and_assign(audio_file, aligned_result)
        return final_result

    def run(self, audio_file: str):
        """
        Run the audio processing pipeline.

        Args:
            audio_file (str): The path to the audio file.
```

```
        Returns:
            dict: The final result.
        """
        return self.process_audio(audio_file)


def worker(audio_file, transcriber):
    print(f"Processing {audio_file}")
    transcriber.process_audio(audio_file)
    print(f"Done with {audio_file}")


@asynccontextmanager
async def lifespan(app: FastAPI):
    """
    An asynchronous context manager for managing the lifecycle of the FastAPI app.
      It ensures that GPU memory is cleared after the app's lifecycle ends, which is essential for efficient resource management in GPU environments.
    """
    yield
    if torch.cuda.is_available():
        torch.cuda.empty_cache()
        torch.cuda.ipc_collect()


def batched_transcribe(audio_files: List[str]):
```

```
    transcriber = WhisperTranscriber(batch_size=16, compute_type="float16")

    threads = []
    for audio_file in audio_files:
        thread = threading.Thread(target=worker, args=(audio_file, transcriber))
        thread.start()
        threads.append(thread)

    for thread in threads:
        thread.join()

    print("All audio files processed.")


class WhisperTranscription(BaseModel):
    file: Optional[str] = None
    model: Optional[str] = "whisperx-large-v2"
    language: Optional[str] = "en"
    prompt: Optional[str] = None
    response_format: Optional[str] = "json"
    temperature: Optional[int] = 0
    timestamp_granularities: Optional[List[str]] = ["sentence"]


# SCHEMA
class WhisperTranscriptionResponse(BaseModel):
```

```
    task: str = "transcription"
    duration: str = "0.0"
    text: str = None
    words: List[str] = []
    segments: List[str] = []


app = FastAPI(debug=True, lifespan=lifespan)

app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)


class ModelList(BaseModel):
    models: List[str] = [
        "whisperx-large-v2",
        "whisperx-small-v2",
    ]


# @app.get("/v1/models", response_model=ModelList)
```

```
# async def list_models():
#     """
#     An endpoint to list available models. It returns a list of model cards.
#     This is useful for clients to query and understand what models are available for use.
#     """
#     model_card = ModelCard(
#         id="cogvlm-chat-17b"
#     )  # can be replaced by your model id like cogagent-chat-18b
#     return ModelList(data=[model_card])


@app.on_event("startup")
async def startup_event():
    global model

    # This is where you can initialize resources that your application needs.
    print("Application startup, initialize resources here.")
    # For example, loading models into memory if necessary.

    # model = WhisperTranscriber(
    #     device="cuda" if torch.cuda.is_available() else "cpu",
    #     compute_type="float16",
    #     hf_token=os.getenv("HF_TOKEN"),
    # )
```

```
@app.on_event("shutdown")
async def shutdown_event():
    print("Application shutdown, cleaning up artifacts")
    if torch.cuda.is_available():
        torch.cuda.empty_cache()
        torch.cuda.ipc_collect()



@app.post("/v1/audio/transcriptions", response_model=WhisperTranscriptionResponse)
async def create_audio_completion(request: WhisperTranscription):
    audio_file: str = request.file

    # Entry
    dict(
        task="transcription",
        audio_file=audio_file,
        model=request.model,
        language=request.language,
        prompt=request.prompt,
        response_format=request.response_format,
        temperature=request.temperature,
        timestamp_granularities=request.timestamp_granularities,
    )

    # Log the entry into supabase
```

```
    transcriber = WhisperTranscriber(
        device="cuda" if torch.cuda.is_available() else "cpu",
    )

    # Run the audio processing pipeline
    out = transcriber.run(audio_file)

    # Response
    return WhisperTranscriptionResponse(task="transcription", text=out["text"]).json()


if __name__ == "__main__":
    uvicorn.run(app, host="0.0.0.0", port=8000, workers=1)
```