# To run this code you need to install the following dependencies:

# pip install google-genai

import base64

import mimetypes

import os

import re

import struct

from google import genai

from google.genai import types

def save\_binary\_file(file\_name, data):

    f = open(file\_name, "wb")

    f.write(data)

    f.close()

    print(f"File saved to to: {file\_name}")

def generate():

    client = genai.Client(

        api\_key=os.environ.get("GEMINI\_API\_KEY"),

    )

    model = "gemini-2.5-flash-preview-tts"

    contents = [

        types.Content(

            role="user",

            parts=[

                types.Part.from\_text(text="""พูดคุยแบบคนรักหยอกล้อ

Speaker 1: สวัสดีค่ะที่รัก

Speaker 2: รักที่สุด สุดที่รัก"""),

            ],

        ),

    ]

    generate\_content\_config = types.GenerateContentConfig(

        temperature=1,

        response\_modalities=[

            "audio",

        ],

        speech\_config=types.SpeechConfig(

            multi\_speaker\_voice\_config=types.MultiSpeakerVoiceConfig(

                speaker\_voice\_configs=[

                    types.SpeakerVoiceConfig(

                        speaker="Speaker 1",

                        voice\_config=types.VoiceConfig(

                            prebuilt\_voice\_config=types.PrebuiltVoiceConfig(

                                voice\_name="Zephyr"

                            )

                        ),

                    ),

                    types.SpeakerVoiceConfig(

                        speaker="Speaker 2",

                        voice\_config=types.VoiceConfig(

                            prebuilt\_voice\_config=types.PrebuiltVoiceConfig(

                                voice\_name="Puck"

                            )

                        ),

                    ),

                ]

            ),

        ),

    )

    file\_index = 0

    for chunk in client.models.generate\_content\_stream(

        model=model,

        contents=contents,

        config=generate\_content\_config,

    ):

        if (

            chunk.candidates is None

            or chunk.candidates[0].content is None

            or chunk.candidates[0].content.parts is None

        ):

            continue

        if chunk.candidates[0].content.parts[0].inline\_data and chunk.candidates[0].content.parts[0].inline\_data.data:

            file\_name = f"ENTER\_FILE\_NAME\_{file\_index}"

            file\_index += 1

            inline\_data = chunk.candidates[0].content.parts[0].inline\_data

            data\_buffer = inline\_data.data

            file\_extension = mimetypes.guess\_extension(inline\_data.mime\_type)

            if file\_extension is None:

                file\_extension = ".wav"

                data\_buffer = convert\_to\_wav(inline\_data.data, inline\_data.mime\_type)

            save\_binary\_file(f"{file\_name}{file\_extension}", data\_buffer)

        else:

            print(chunk.text)

def convert\_to\_wav(audio\_data: bytes, mime\_type: str) -> bytes:

    """Generates a WAV file header for the given audio data and parameters.

    Args:

        audio\_data: The raw audio data as a bytes object.

        mime\_type: Mime type of the audio data.

    Returns:

        A bytes object representing the WAV file header.

    """

    parameters = parse\_audio\_mime\_type(mime\_type)

    bits\_per\_sample = parameters["bits\_per\_sample"]

    sample\_rate = parameters["rate"]

    num\_channels = 1

    data\_size = len(audio\_data)

    bytes\_per\_sample = bits\_per\_sample // 8

    block\_align = num\_channels \* bytes\_per\_sample

    byte\_rate = sample\_rate \* block\_align

    chunk\_size = 36 + data\_size  # 36 bytes for header fields before data chunk size

    # http://soundfile.sapp.org/doc/WaveFormat/

    header = struct.pack(

        "<4sI4s4sIHHIIHH4sI",

        b"RIFF",          # ChunkID

        chunk\_size,       # ChunkSize (total file size - 8 bytes)

        b"WAVE",          # Format

        b"fmt ",          # Subchunk1ID

        16,               # Subchunk1Size (16 for PCM)

        1,                # AudioFormat (1 for PCM)

        num\_channels,     # NumChannels

        sample\_rate,      # SampleRate

        byte\_rate,        # ByteRate

        block\_align,      # BlockAlign

        bits\_per\_sample,  # BitsPerSample

        b"data",          # Subchunk2ID

        data\_size         # Subchunk2Size (size of audio data)

    )

    return header + audio\_data

def parse\_audio\_mime\_type(mime\_type: str) -> dict[str, int | None]:

    """Parses bits per sample and rate from an audio MIME type string.

    Assumes bits per sample is encoded like "L16" and rate as "rate=xxxxx".

    Args:

        mime\_type: The audio MIME type string (e.g., "audio/L16;rate=24000").

    Returns:

        A dictionary with "bits\_per\_sample" and "rate" keys. Values will be

        integers if found, otherwise None.

    """

    bits\_per\_sample = 16

    rate = 24000

    # Extract rate from parameters

    parts = mime\_type.split(";")

    for param in parts: # Skip the main type part

        param = param.strip()

        if param.lower().startswith("rate="):

            try:

                rate\_str = param.split("=", 1)[1]

                rate = int(rate\_str)

            except (ValueError, IndexError):

                # Handle cases like "rate=" with no value or non-integer value

                pass # Keep rate as default

        elif param.startswith("audio/L"):

            try:

                bits\_per\_sample = int(param.split("L", 1)[1])

            except (ValueError, IndexError):

                pass # Keep bits\_per\_sample as default if conversion fails

    return {"bits\_per\_sample": bits\_per\_sample, "rate": rate}

if \_\_name\_\_ == "\_\_main\_\_":

    generate()