

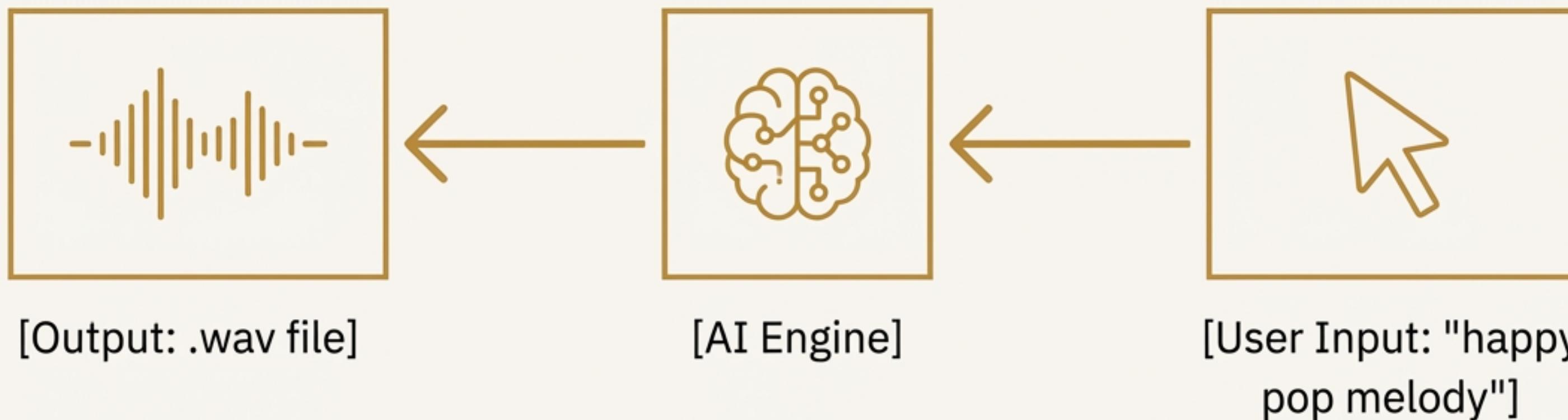
# الملحن الاصطناعي: رحلة فكرة موسيقية

تفكيك بنية تطبيق موسيقي يعتمد على النماذج اللغوية الكبيرة



# من النص إلى النغم: ما الذي نبني هنا؟

هذا المشروع هو عرض عمل لكيفية تنسيق سلسلة من استدعاءات النماذج اللغوية الكبيرة (LLM) لتحويل وصف نصي بسيط من المستخدم إلى مقطوعة موسيقية متكاملة وملف صوتي مسموع. إنه يجسد فكرة تحويل اللغة الطبيعية إلى لغة موسيقية.



# المخطط الهندسي : نظرة عامة على المكونات الرئيسية

يتكون التطبيق من ثلاث وحدات رئيسية تعمل معاً بتناغم: واجهة المستخدم التي تستقبل الأفكار، والمحرك الأساسي الذي يترجمها، ومركب الصوت الذي يحولها إلى حقيقة مسموعة.



# الجزء الأول: المحرك الأساسي في `main.py`

في صميم التطبيق تكمن فئة `MusicLLM`. هذه الفئة ليست مجرد غلاف لواجهة برمجة التطبيقات؛ بل هي المنسق الذي يدير سلسلة من المهام المتخصصة.

تستخدم نموذج `llama-3.1-8b-instant` فائق السرعة عبر Groq لتنفيذ كل خطوة بكفاءة.

اختيار نموذج سريع  
 ومناسب للمهام المتسلسلة

(main.py)

محرك التوليد

المنطق الأساسي باستخدام Groq و LangChain

```
# from main.py
class MusicLLM:
    """High-level helper that wraps Groq's chat API"""

    def __init__(self, temperature: float = 0.7) -> None:
        self.llm = ChatGroq(
            temperature=temperature,
            groq_api_key=api_key,
            model_name="llama-3.1-8b-instant",
        )
```

# هندسة الأوامر: حوار إبداعي مع الذكاء الاصطناعي

بدلاً من أن نطلب من النموذج اللغوي "تأليف موسيقى" في خطوة واحدة، نقوم بتقسيم المهمة إلى أربع خطوات منطقية. كل خطوة لها قالب أمر (Prompt Template) مصمم بدقة لتوليد جزء محدد من المقطوعة الموسيقية.

## HARMONY\_PROMPT

Create harmony chords for this melody:

{melody}.

Format:

Format: C4-E4-G4 F4-A4-C5

## MELODY\_PROMPT

Generate a melody based on this input:

{input}.

Represent it as a space seperated notes  
(eg., C4 D4 E4)

## STYLE\_PROMPT

Adapt to {style} style: \n

  Melody: {melody}

  Harmony: {harmony}

  Rhythm: {rhythm}

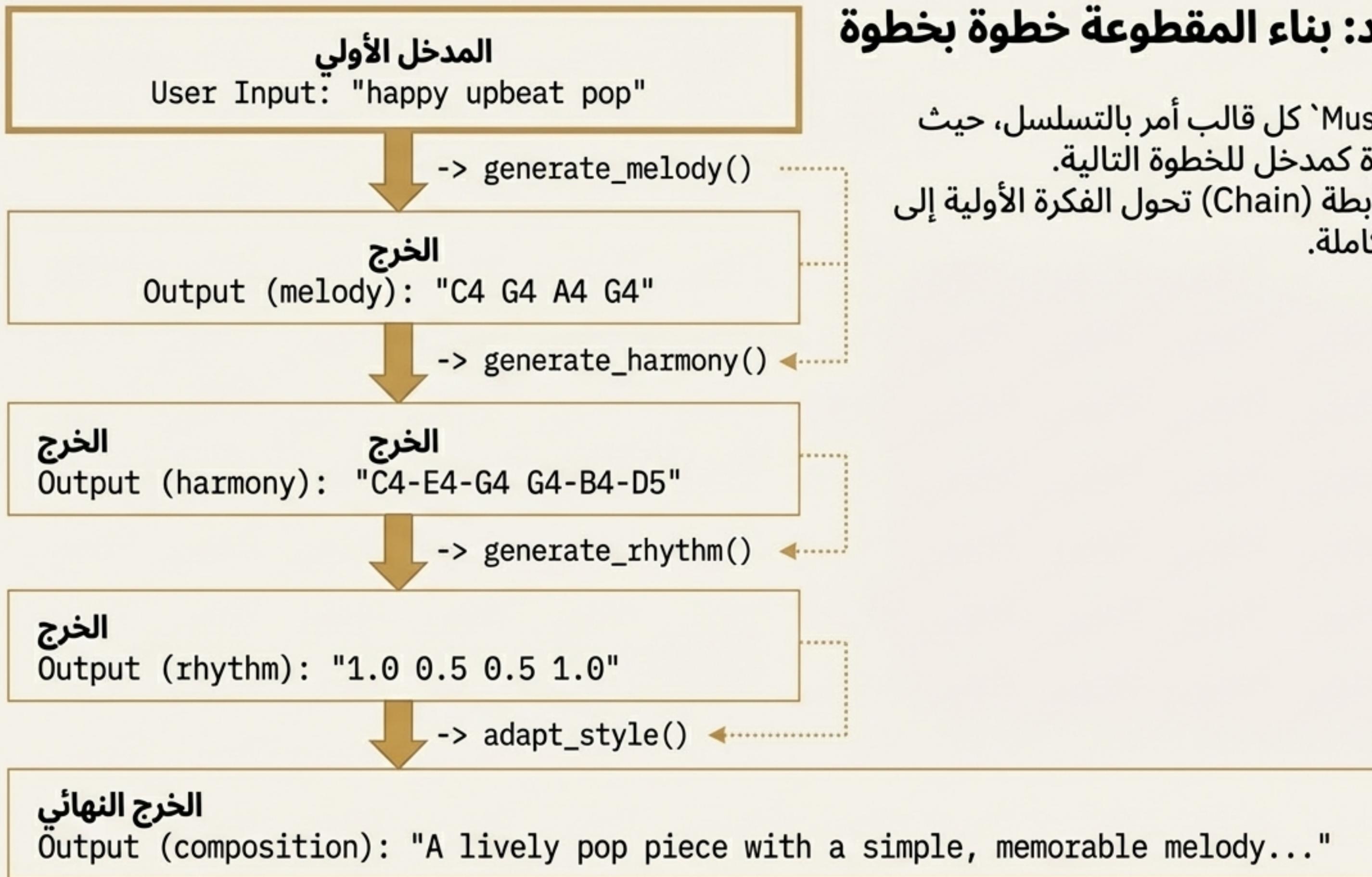
Output single string summary

## RHYTHM\_PROMPT

Suggest rhythm durations (in beats) for  
this melody: {melody}.

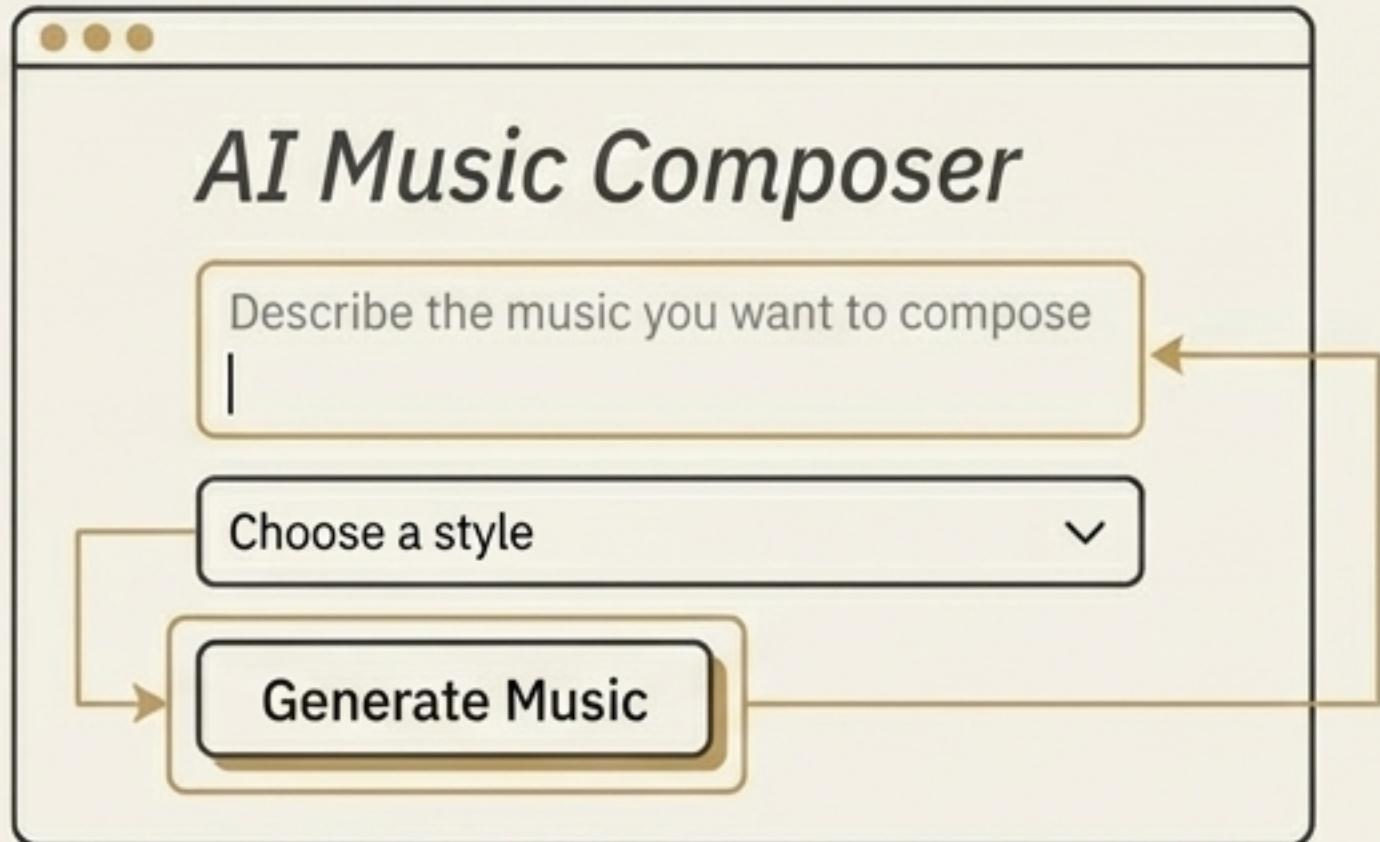
Format: 1.0 0.5 0.5 2.0

# تسلسل التوليد: بناء المقطوعة خطوة بخطوة



تستدعي فئة `MusicLLM` كل قالب أمر بالتسلسل، حيث يعتمد خرج كل خطوة كمدخل للخطوة التالية. هذه السلسلة المتراابطة (Chain) تحول الفكرة الأولية إلى بيانات موسيقية متكاملة.

## الجزء الثاني: الواجهة التفاعلية في `app.py`



لتسهيل التفاعل، نستخدم Streamlit لإنشاء واجهة مستخدم بسيطة.

تجمع هذه الواجهة مدخلات المستخدم - وصف الموسيقى والأسلوب المطلوب - وتعمل كنقطة انطلاق لعملية التوليد بأكملها.

```
# from app.py
st.title(" AI Music Composer")

music_input = st.text_input("Describe the music you want to compose")
style = st.selectbox(
    "Choose a style",
    ["Sad", "Happy", "Jazz", "Romantic", "Extreme"]
)

if st.button("Generate Music") and music_input:
    # ... generation logic ...
```

# ربط الواجهة بالمحرك: لحظة تفعيل الإبداع

عندما يضغط المستخدم على الزر، يتم تفعيل الكتلة البرمجية الشرطية `if st.button("...")`. هنا، يتم إنشاء كائن من `MusicLLM` وتمرير المدخلات التي تم جمعها من الواجهة بدء سلسلة التوليد التي استعرضناها سابقاً.

```
# from app.py
if st.button("Generate Music") and music_input:
    generator = MusicLLM()
    with st.spinner("Generating music"):
        melody = generator.generate_melody(music_input)
        harmony = generator.generate_harmony(melody)
        rhythm = generator.generate_rhythm(melody)
        composition = generator.adapt_style(style, ...)
```

The diagram illustrates the flow of data in the code. It starts with a 'Trigger' arrow pointing to the 'st.button("Generate Music")' call. This triggers the execution of the code block. An 'Instantiate Engine' arrow points to the 'generator = MusicLLM()' line, where the engine is created. A 'Pass input' arrow points to the 'music\_input' parameter being passed to the engine's methods. Finally, a 'Pass style' arrow points to the 'style' parameter being passed to the 'adapt\_style' method.

# الجزء الثالث: من البيانات إلى الصوت في `utils.py`

النموذج اللغوي يمنحك نotas موسيقية كنص (مثل 'C4'). لإنتاج صوت، يجب تحويل هذه الرموز إلى ترددات رقمية (بالهرتز). تتولى الدالة `note\_to\_frequencies` بمكتبة `music21` المتخصصة في التحليل الموسيقي.

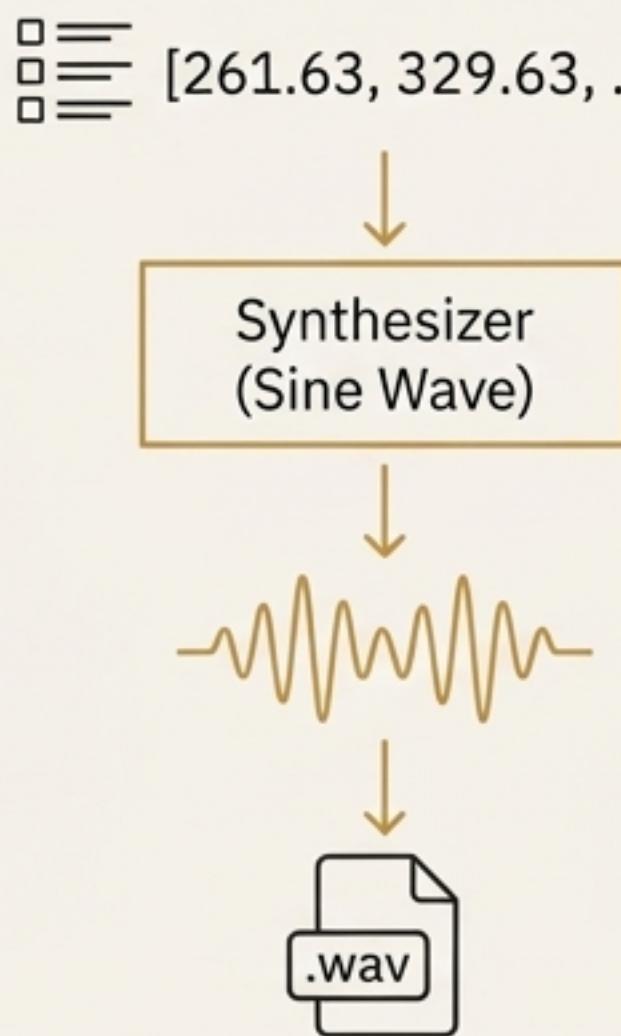
[ "C4", "E4", "G4"] → [261.63, 329.63, 392.00]

```
# from utils.py
import music21

def note_to_frequencies(note_list):
    freqs = []
    for note_str in note_list:
        try:
            note = music21.note.Note(note_str)
            freqs.append(note.pitch.frequency)
        except Exception as e:
            logger.warning("Invalid note ignored...")
    return freqs
```

استخدام مكتبة متخصصة  
لضمان دقة الترجمة الموسيقية

# توليد الموجة الصوتية: إحياء الموسيقى



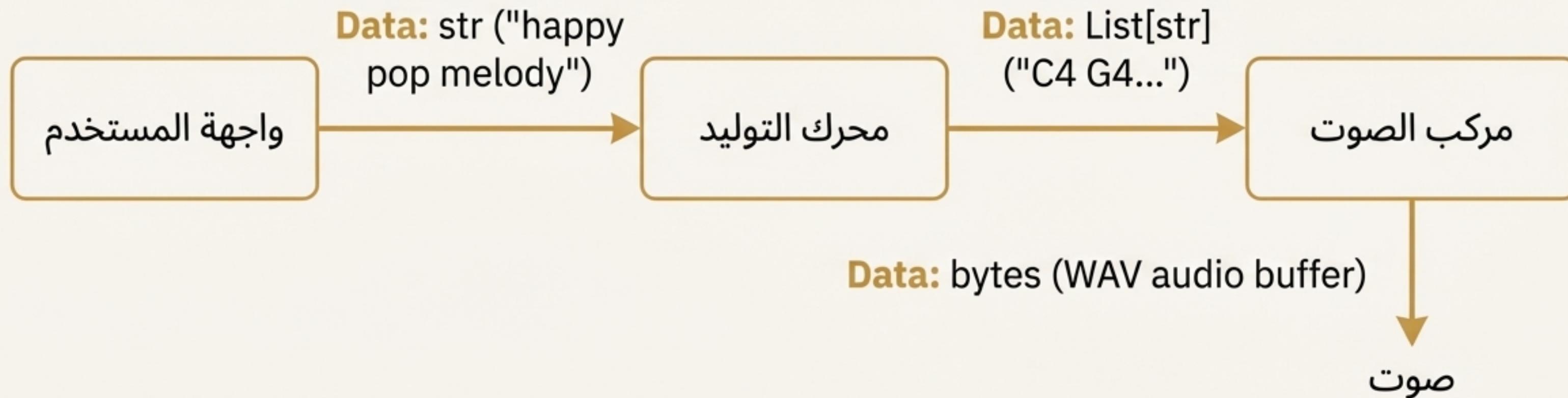
```
# from utils.py
from synthesizer import Synthesizer
import numpy as np
from scipy.io.wavfile import write as wav_write

def generate_wav_bytes_from_notes_freq(notes_freq):
    synth = Synthesizer(...)
    chunks = [synth.generate_constant_wave(freq, 0.5)]
    audio = np.concatenate(chunks)
    buffer = io.BytesIO()
    wav_write(buffer, 44100, audio.astype(np.float32))
    return buffer.getvalue()
```

أخيراً، يتم تمرير قائمة الترددات إلى مركب صوتي (Synthesizer) بسيط. يقوم هذا المركب بتوليد موجات جيبية (Sine Waves) لكل تردد لمدة محددة، ثم يدمجها معًا في مصفوفة رقمية. يتم بعد ذلك ترميز هذه المصفوفة كملف صوتي .scipy و numpy بصيغة WAV باستخدام

# الرحلة الكاملة: من الفكرة إلى الأذن

لقد تبعنا فكرة المستخدم منذ كانت مجرد نص، مروراً بتحولها إلى سلاسل نصية موسيقية، ثم إلى ترددات رقمية، وأخيراً إلى بايتات صوتية مسموعة. كل مكون في بنية المعمارية يؤدي دوراً محدداً وحاسماً في هذه الرحلة.



# المكونات الداعمة: ضمان الجودة والموثوقية

خلف الكواليس، تضمن مكونات مثل `logging\_setup.py` و `dotenv` أن التطبيق ليس فقط فعالاً، بل أيضاً قوياً وقابلأً للصيانة. يوفر التسجيل (Logging) رؤية واضحة لسلوك التطبيق، بينما تدير `dotenv` متغيرات البيئة بشكل آمن.

## Configuration

```
# from main.py
from dotenv import load_dotenv
load_dotenv()
api_key = os.getenv("GROQ_API_KEY", ...)
```

## Logging

```
# from logging_setup.py
from loguru import logger
LOG_FILE = Path(...) / "musicllm.log"
logger.add(
    LOG_FILE,
    rotation="5 MB",
    retention="7 days",
)
```

# الدروس المستفادة والأفكار الرئيسية

يمكن تلخيص فلسفة تصميم هذا التطبيق في ثلاثة مبادئ أساسية:

## فرق تسد (Divide and Conquer)

تحويل مشكلة معقدة (تأليف الموسيقى) إلى سلسلة من المهام البسيطة التي يمكن للموذج اللغوي التعامل معها بفعالية من خلال أوامر موجهة.



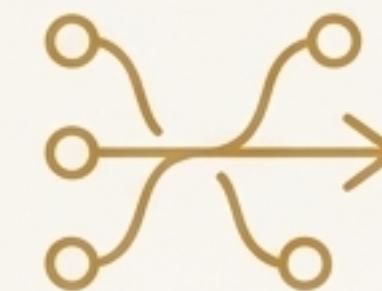
## فصل الاهتمامات (Separation of Concerns)

تصميم وحدات برمجية مستقلة لكل وظيفة: `main.py` للمنطق، `app.py` للواجهة، و `utils.py` للأدوات المساعدة. هذا يسهل الصيانة والتطوير.



## البيانات هي القصة (Data as the Story)

التركيز على رحلة تحول البيانات - من نص إلى سلاسل نصية إلى ترددات إلى صوت - إلى صوت - باعتبارها السرد المركزي الذي يحرك بنية التطبيق بأكملها.





شکرہِ کم

[github.com/example/ai-composer](https://github.com/example/ai-composer)