import React, { useEffect, useState, useRef } from 'react';

const LessonStream = ({ subtopic, onAskQuestion }) => {

    const [elements, setElements] = useState([]);

    const [isFinished, setIsFinished] = useState(false);

    const socketRef = useRef(null);

    const seenImagesRef = useRef(new Set());

    const seenVideosRef = useRef(new Set());

    const seenWarningRef = useRef(false);

    const textBufferRef = useRef('');

    // Parse incoming chunks into React nodes

    const parseChunk = (chunk) => {

        const parts = chunk.split(/(<<image:[^>]+>>|<<video:[^>]+>>)/g);

        return parts.flatMap((seg) => {

            // Out-of-syllabus warning

            if (seg.includes("This topic seems to be outside the syllabus")) {

                if (seenWarningRef.current) return [];

                seenWarningRef.current = true;

                return [

                    <p key="warning" style={{ color: '#a00', fontWeight: 'bold', textAlign: 'center' }}>

                        {seg.trim()}

                    </p>

                ];

            }

            // Image embedding

            const imgMatch = seg.match(/<<image:\s\*([^\s>]+)\s\*>>/i);

            if (imgMatch) {

                const name = imgMatch[1];

                if (seenImagesRef.current.has(name)) return [];

                seenImagesRef.current.add(name);

                return [

                    <div key={`img-${name}`} style={{ textAlign: 'center', margin: '1em 0' }}>

                        <img

                            src={`http://localhost:8000/images/${name}`}

                            alt={name}

                            style={{ maxWidth: '100%', border: '1px solid #ccc', borderRadius: '4px' }}

                            onError={(e) => console.error(`Error loading image: ${e.target.src}`)}

                        />

                        <div style={{ fontSize: '0.9em', color: '#555', marginTop: '0.3em' }}>{name}</div>

                    </div>

                ];

            }

            // Video embedding

            const vidMatch = seg.match(/<<video:\s\*([^\s>]+)\s\*>>/);

            if (vidMatch) {

                const id = vidMatch[1];

                if (seenVideosRef.current.has(id)) return [];

                seenVideosRef.current.add(id);

                return [

                    <div key={`vid-${id}`} style={{ margin: '1em 0' }}>

                        <iframe

                            width="560"

                            height="315"

                            src={`https://www.youtube.com/embed/${id}`}

                            title="Lesson video"

                            frameBorder="0"

                            allow="accelerometer; autoplay; clipboard-write; encrypted-media; gyroscope; picture-in-picture"

                            allowFullScreen

                            onError={(e) => console.error(`Error loading video: ${e.target.src}`)}

                        />

                    </div>

                ];

            }

            // Plain text

            if (seg && seg.trim()) {

                return [<p key={`txt-${Math.random().toString(36).substr(2, 9)}`}>{seg}</p>];

            }

            return [];

        });

    };

    // WebSocket effect

    useEffect(() => {

        seenImagesRef.current.clear();

        seenVideosRef.current.clear();

        seenWarningRef.current = false;

        textBufferRef.current = '';

        setElements([]);

        setIsFinished(false);

        const socket = new WebSocket('ws://localhost:8000/ws/lesson');

        socketRef.current = socket;

        socket.onopen = () => {

            console.log('WebSocket connected');

            socket.send(JSON.stringify({ subtopic }));

        };

        socket.onmessage = ({ data }) => {

            if (data === '[DONE]') {

                setIsFinished(true);

                socketRef.current?.close();

                return;

            }

            textBufferRef.current += data;

            let buffer = textBufferRef.current;

            const lastImg = buffer.lastIndexOf('<<image:');

            const lastVid = buffer.lastIndexOf('<<video:');

            const lastOpen = Math.max(lastImg, lastVid);

            let processStr = buffer;

            let leftover = '';

            if (lastOpen !== -1) {

                const closingIdx = buffer.indexOf('>>', lastOpen);

                if (closingIdx === -1) {

                    processStr = buffer.substring(0, lastOpen);

                    leftover = buffer.substring(lastOpen);

                }

            }

            textBufferRef.current = leftover;

            const newNodes = parseChunk(processStr);

            setElements(prev => [...prev, ...newNodes]);

        };

        socket.onerror = (err) => console.error('WebSocket error', err);

        socket.onclose = () => {

            console.log('WebSocket closed');

            socketRef.current = null;

        };

        return () => {

            if (socketRef.current && socketRef.current.readyState === WebSocket.OPEN) {

                socketRef.current.close();

            }

        };

    }, [subtopic]);

    return (

        <div className="lesson-stream" style={{ position: 'relative' }}>

            <button

                onClick={onAskQuestion}

                style={{ position: 'absolute', top: '1em', right: '1em', zIndex: 10 }}

            >

                Ask Question

            </button>

            {elements}

            {isFinished && (

                <div style={{ marginTop: '2em', padding: '1em', borderTop: '1px solid #ccc', textAlign: 'center' }}>

                    <h2>🎉 Lesson Complete!</h2>

                    <p>Key takeaways above!</p>

                </div>

            )}

        </div>

    );

};

export default LessonStream;

lessonstream.js  
  
  
  
  
  
  
  
  
  
  
import React, { useState } from 'react';

import LessonStream from './LessonStream';

import ChatWindow from './ChatWindow';

function App() {

  // Controlled input for typing the topic

  const [topicInput, setTopicInput] = useState("");

  // Active lesson subtopic

  const [subtopic, setSubtopic] = useState("");

  const [lessonStarted, setLessonStarted] = useState(false);

  // Tracks if user is in question (doubt) mode

  const [questionMode, setQuestionMode] = useState(false);

  // Used to reset LessonStream when restarting

  const [streamKey, setStreamKey] = useState(0);

  const [chatHistory, setChatHistory] = useState([]);

  const handleStartLesson = () => {

    const trimmedTopic = topicInput.trim();

    if (!trimmedTopic) return;

    setSubtopic(trimmedTopic);

    setLessonStarted(true);

    setQuestionMode(false);

    setChatHistory([]);

    setStreamKey(k => k + 1);

  };

  const handleAskQuestion = () => {

    // Pause the lesson stream

    setQuestionMode(true);

  };

  const handleChatSend = async (message) => {

    // Confirmation check

    const confirmRes = await fetch('http://localhost:8000/confirm', {

      method: 'POST',

      headers: { 'Content-Type': 'application/json' },

      body: JSON.stringify({ message })

    });

    const { confirm } = await confirmRes.json();

    if (confirm) {

      // Student understood: resume lesson

      setChatHistory([]);

      setQuestionMode(false);

      return;

    }

    // Append student message

    const updatedHistory = [...chatHistory, { role: 'student', text: message }];

    setChatHistory(updatedHistory);

    // Send to chat endpoint

    const res = await fetch('http://localhost:8000/chat', {

      method: 'POST',

      headers: { 'Content-Type': 'application/json' },

      body: JSON.stringify({ subtopic, history: updatedHistory, question: message })

    });

    let botText = '';

    const reader = res.body.getReader();

    const decoder = new TextDecoder();

    while (true) {

      const { value, done } = await reader.read();

      if (done) break;

      botText += decoder.decode(value);

    }

    setChatHistory(h => [...h, { role: 'bot', text: botText.trim() }]);

  };

  return (

    <div className="App" style={{ maxWidth: 800, margin: '0 auto', padding: '1rem' }}>

      <h1 style={{ textAlign: 'center' }}>📚 AI Science Teacher</h1>

      {/\* Topic Input \*/}

      {!lessonStarted && (

        <div style={{ display: 'flex', gap: '0.5rem', marginTop: '1rem' }}>

          <input

            type="text"

            placeholder="Enter a lesson topic (e.g., Photosynthesis)"

            value={topicInput}

            onChange={(e) => setTopicInput(e.target.value)}

            style={{ flex: 1, padding: '0.5rem', fontSize: '1rem' }}

          />

          <button

            onClick={handleStartLesson}

            disabled={!topicInput.trim()}

            style={{ padding: '0.5rem 1rem', fontSize: '1rem' }}

          >

            Start Lesson

          </button>

        </div>

      )}

      {/\* Lesson Stream with multimedia \*/}

      {lessonStarted && !questionMode && (

        <LessonStream

          key={streamKey}

          subtopic={subtopic}

          onAskQuestion={handleAskQuestion}

          questionMode={questionMode} // pass pause flag

        />

      )}

      {/\* Chat Window for questions \*/}

      {lessonStarted && questionMode && (

        <ChatWindow history={chatHistory} onSend={handleChatSend} />

      )}

    </div>

  );

}

export default App;

app.js  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
import os

from fastapi import FastAPI, WebSocket, HTTPException

from fastapi.middleware.cors import CORSMiddleware

from fastapi.staticfiles import StaticFiles

from fastapi.responses import StreamingResponse

from pydantic import BaseModel

from backend.agent import get\_lesson\_prompt, rag\_retriever

from backend.tools.llm\_tools import stream\_grok, summarize\_text,stream\_chat

app = FastAPI()

# Path to the image directory

image\_dir = os.path.join(os.path.dirname(\_\_file\_\_), "tools", "images")

# Mount static files

app.mount("/images", StaticFiles(directory=image\_dir), name="images")

# CORS middleware

app.add\_middleware(

    CORSMiddleware,

    allow\_origins=["http://localhost:3000"],

    allow\_credentials=True,

    allow\_methods=["\*"],

    allow\_headers=["\*"],

)

# Helper: use LLM to classify confirmation

async def classify\_confirmation(reply: str) -> bool:

    """

    Returns True if the LLM classifies the reply as indicating understanding.

    """

    classifier\_prompt = (

        f"You are a helpful assistant. The student just replied:\n\n"

        f"\"{reply}\"\n\n"

        "Does this message indicate that the student understood the previous explanation?"

        " Answer with just 'Yes' or 'No'."

    )

    response = ""

    async for chunk in stream\_grok(classifier\_prompt):

        response += chunk

    first = response.strip().split()[0].lower()

    return first.startswith("y")

# --------- Confirmation Endpoint ---------

class ConfirmRequest(BaseModel):

    message: str

@app.post("/confirm")

async def confirm\_understanding(req: ConfirmRequest):

    is\_confirm = await classify\_confirmation(req.message)

    return {"confirm": is\_confirm}

# --------- Chat Endpoint ---------

class ChatRequest(BaseModel):

    subtopic: str

    history: list[dict]

    question: str

@app.post("/chat")

async def chat(req: ChatRequest):

    # 1. Retrieve lesson context

    context = "\n".join(rag\_retriever.retrieve(req.subtopic, k=5))

    out\_of\_syllabus = not context.strip()

    # 2. Build system prompt

    if out\_of\_syllabus:

        system\_prompt = (

            "⚠️ This question is outside the syllabus. Provide a general answer and warn the student: 'This is outside the syllabus.'\n\n"

        )

    else:

        system\_prompt = "You are an 8th-grade science teacher. Answer student questions using the lesson content. "

    system\_prompt += (

        "By default, give short, doubt-clearing answers. "

        "If the student requests a specific length (e.g., '10 marks', '200 words'), provide a detailed response. "

        "Always end with a follow-up: 'Did that help?', or 'Which part needs more detail?'."

    )

    # 3. Assemble message sequence

    messages = [{"role": "system", "content": system\_prompt}]

    if not out\_of\_syllabus:

        messages.append({"role": "system", "content": f"Lesson content:\n{context}"})

    # Inject previous chat history

    for turn in req.history:

        role = "user" if turn.get("role") == "student" else "assistant"

        messages.append({"role": role, "content": turn.get("text", "")})

    # Add the new student question

    messages.append({"role": "user", "content": req.question})

    # 4. Stream the model's response

        # 4. Flatten messages and stream via generate\_content

    async def event\_stream():

        # Build one text prompt from your message history

        prompt = "\n".join(

            f"{m['role'].upper()}: {m['content']}"

            for m in messages

        )

        # Stream it through your existing generate\_content helper

        async for chunk in stream\_grok(prompt):

            yield chunk

    return StreamingResponse(event\_stream(), media\_type="text/plain")

# --------- WebSocket /ws/lesson ---------

@app.websocket("/ws/lesson")

async def lesson\_stream(websocket: WebSocket):

    await websocket.accept()

    data = await websocket.receive\_json()

    prompt = get\_lesson\_prompt(data["subtopic"])

    if "\*\*[OUT\_OF\_SYLLABUS]\*\*" in prompt:

        await websocket.send\_text(

            "\n\*\*This topic seems to be outside the syllabus. Here's a general overview anyway...\*\*\n"

        )

        prompt = (

            f"You are a science teacher. Give a general, engaging explanation of the topic: {data['subtopic']}"

        )

    full\_text = ""

    buffer = ""

    async for chunk in stream\_grok(prompt):

        buffer += chunk

        has\_open = "<<" in buffer

        has\_close = ">>" in buffer

        if (len(buffer) > 300 and not (has\_open and not has\_close)) or buffer.endswith((".", "!", "?")):

            await websocket.send\_text(buffer)

            full\_text += buffer

            buffer = ""

    if buffer.strip():

        await websocket.send\_text(buffer)

        full\_text += buffer

    await websocket.send\_text("\n\n\n\*\*Lesson Complete! Generating summary...\*\*\n")

    summary = await summarize\_text(full\_text)

    await websocket.send\_text(f"\*\*Summary:\*\* {summary}")

    await websocket.send\_text("[[DONE]]")

    await websocket.close()

main.py