VoiceChat.tsx

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

import { Button } from './ui/button'

import { Card, CardContent, CardHeader, CardTitle } from './ui/card'

import { Mic, MicOff, Loader2, WifiOff, Waves } from 'lucide-react'

import { BlandWebClient } from 'bland-client-js-sdk'

type Message = {

id: string

text: string

sender: 'user' | 'ai'

}

interface VoiceChatProps {

agentId: string;

}

export default function VoiceChat({ agentId }: VoiceChatProps) {

const [messages, setMessages] = useState<Message[]>([])

const [isRecording, setIsRecording] = useState(false)

const [status, setStatus] = useState<string>('Idle')

const [error, setError] = useState<string | null>(null)

const [audioLevel, setAudioLevel] = useState<number>(0)

const [isLoading, setIsLoading] = useState(false)

const clientRef = useRef<BlandWebClient | null>(null)

const audioLevelIntervalRef = useRef<NodeJS.Timeout>()

const [isConnected, setIsConnected] = useState(false)

const [callId, setCallId] = useState<string>('')

// Utility functions for cleanup

const cleanupWebSocket = async (wsInstance: WebSocket): Promise<void> => {

if (wsInstance && wsInstance.readyState !== WebSocket.CLOSED) {

// Remove listeners

wsInstance.onclose = null;

wsInstance.onerror = null;

wsInstance.onmessage = null;

wsInstance.onopen = null;

// Close connection

wsInstance.close(1000, 'User disconnected');

// Wait for closure

await new Promise<void>((resolve) => {

const checkClosed = setInterval(() => {

if (wsInstance.readyState === WebSocket.CLOSED) {

clearInterval(checkClosed);

resolve();

}

}, 50);

setTimeout(() => {

clearInterval(checkClosed);

resolve();

}, 2000);

});

}

};

const cleanupMediaStream = (mediaStream: MediaStream) => {

if (mediaStream?.getTracks) {

mediaStream.getTracks().forEach((track: MediaStreamTrack) => {

track.enabled = false;

track.stop();

});

}

};

const cleanupAudioContext = async (audioContext: AudioContext) => {

if (audioContext && audioContext.state !== 'closed') {

await audioContext.close();

}

};

const releaseMicrophone = async () => {

try {

const devices = await navigator.mediaDevices.enumerateDevices();

const audioDevices = devices.filter(device => device.kind === 'audioinput');

for (const device of audioDevices) {

const stream = await navigator.mediaDevices.getUserMedia({

audio: { deviceId: device.deviceId }

});

stream.getTracks().forEach(track => {

track.enabled = false;

track.stop();

});

}

} catch (err) {

console.warn('Could not release microphone:', err);

}

};

const cleanup = useCallback(async () => {

setIsLoading(true);

try {

// Clear audio level interval

if (audioLevelIntervalRef.current) {

clearInterval(audioLevelIntervalRef.current);

audioLevelIntervalRef.current = undefined;

}

if (clientRef.current) {

try {

// Stop ongoing processes

if (typeof (clientRef.current as any).stop === 'function') {

await (clientRef.current as any).stop();

}

// Cleanup WebSocket

const wsInstance = (clientRef.current as any).\_ws ||

(clientRef.current as any).ws ||

(clientRef.current as any).webSocket;

await cleanupWebSocket(wsInstance);

// Cleanup client

if (typeof clientRef.current.disconnect === 'function') {

await clientRef.current.disconnect();

}

// Cleanup media stream

cleanupMediaStream((clientRef.current as any).mediaStream);

// Cleanup audio context

await cleanupAudioContext((clientRef.current as any).audioContext);

clientRef.current = null;

} catch (err) {

console.error('Error stopping client:', err);

throw new Error('Failed to disconnect properly');

}

}

// Release microphone as final step

await releaseMicrophone();

} catch (err) {

console.error('Error during cleanup:', err);

setError(err instanceof Error ? err.message : 'Cleanup failed');

} finally {

// Reset states

setIsConnected(false);

setIsRecording(false);

setStatus('Disconnected');

setCallId('');

setAudioLevel(0);

setIsLoading(false);

}

}, []);

// Handle voice toggle

const handleVoiceToggle = async () => {

if (isRecording) {

setStatus('Disconnecting...');

await cleanup();

} else {

initVoiceChat(); // Your existing initVoiceChat function

}

};

// Add connection status effect

useEffect(() => {

return () => {

if (isConnected) {

cleanup();

}

};

}, [isConnected, cleanup]);

const initVoiceChat = async () => {

if (!agentId) {

setError('Agent ID is not set')

return

}

cleanup() // Cleanup any existing connections

setStatus('Initializing...')

setError(null)

setIsLoading(true)

try {

const response = await fetch('/api/getToken', {

method: 'POST',

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

body: JSON.stringify({ agentId })

})

const data = await response.json()

if (!data.token) {

throw new Error('No token received')

}

setStatus('Connecting to Bland AI...')

clientRef.current = new BlandWebClient(agentId, data.token)

const currentCallId = Date.now().toString()

await clientRef.current.initConversation({

sampleRate: 44100,

callId: currentCallId

})

setCallId(currentCallId)

setStatus('Connected! Start speaking...')

setIsRecording(true)

setIsConnected(true)

// Simulate audio levels for visualization

audioLevelIntervalRef.current = setInterval(() => {

setAudioLevel(Math.random())

}, 100)

} catch (err) {

console.error('Voice chat error:', err)

setError(err instanceof Error ? err.message : 'Failed to connect to voice chat')

setStatus('Error connecting')

setIsRecording(false)

cleanup()

} finally {

setIsLoading(false)

}

}

return (

<Card className="w-full max-w-md mx-auto h-[400px] overflow-hidden border-none bg-white/10 backdrop-blur-lg shadow-2xl">

<CardHeader className="pb-4 relative">

<div className="absolute inset-0 bg-gradient-to-b from-white/10 to-transparent"></div>

<CardTitle className="relative flex items-center justify-between text-2xl font-light tracking-tight text-white">

<span className="flex items-center gap-3 w-full justify-center">

Voice Assistant

{isLoading && (

<Loader2 className="h-4 w-4 animate-spin text-white/70" />

)}

</span>

</CardTitle>

</CardHeader>

<CardContent className="flex flex-col items-center justify-between h-[300px] px-8">

<div className="text-center w-full flex flex-col items-center gap-6">

<div className={`text-lg font-light transition-colors duration-300 ${

isRecording ? 'text-white' :

isLoading ? 'text-white/70' : 'text-white/50'

}`}>

{status}

</div>

<Button

onClick={handleVoiceToggle}

size="lg"

className={`

w-24 h-24 rounded-full transition-colors duration-500

${isRecording

? 'bg-red-500/20 hover:bg-red-500/30'

: 'bg-white/10 hover:bg-white/20'

}

border-none shadow-xl hover:shadow-2xl

flex items-center justify-center

group

`}

disabled={isLoading}

>

{isRecording ? (

<MicOff className="h-8 w-8 text-red-500 transition-transform duration-300 group-hover:scale-110" />

) : (

<Mic className="h-8 w-8 text-white transition-transform duration-300 group-hover:scale-110" />

)}

</Button>

<div className="h-12 flex items-center justify-center">

{isRecording ? (

<div className="flex justify-center items-center space-x-1">

{[...Array(12)].map((\_, i) => (

<div

key={i}

className="w-1 bg-white/30 rounded-full animate-pulse"

style={{

height: `${Math.max(12, Math.random() \* 48)}px`,

animationDelay: `${i \* 0.1}s`,

animationDuration: '0.5s'

}}

/>

))}

</div>

) : !isLoading && (

<div className="flex justify-center items-center gap-2 text-white/50">

<WifiOff className="h-4 w-4" />

<span className="text-sm font-light">Ready to start</span>

</div>

)}

</div>

{error && (

<div className="text-red-300/90 text-sm bg-red-500/10 p-4 rounded-2xl backdrop-blur-sm">

{error}

</div>

)}

</div>

</CardContent>

</Card>

)

}