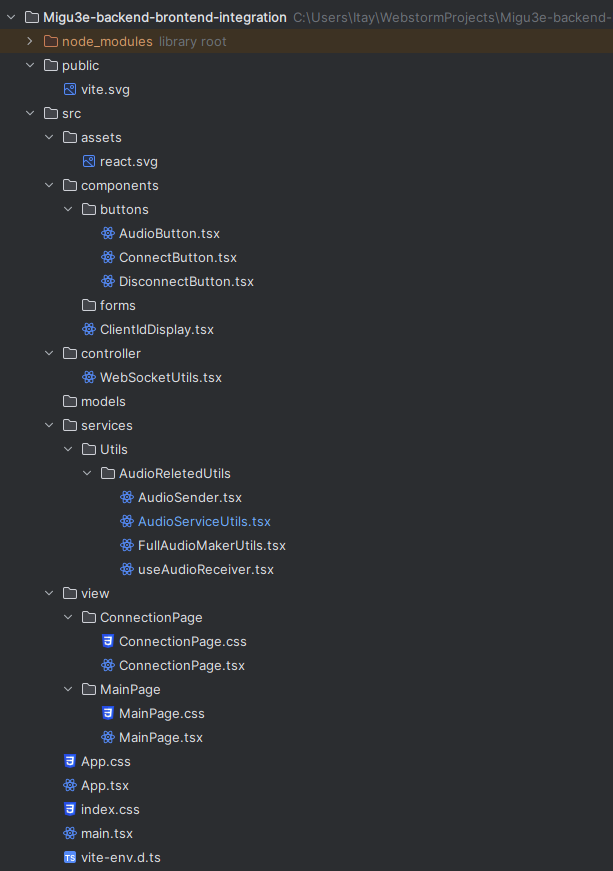
פרוקיט

**מבנה client:**



הקבצים בclient:

**ConnectionPage.tsx**

הדף הזה אחראי על הכניסה וחיבור, כאשר לוחצים על כפתור החיבור הוא מועבר לhandleConnect ומחכה שהפונקציה connect שמיובאת מWebSocketUtils תתחבר לסרבר ומועבר לף הראשי:

await connect();  
navigate('/main');

**MainPage.tsx - הדף הראשי**

הדף הראשי הוא הדף המרכזי בפרויקט, משמש כהמקום בו קורה רוב הדברים. הוא משתמש בWebSocketUtils לפעולות לשלו, והוא אחראי על:

* הצגה של הclientID
* כפתור התחל שידור הפסק שידור
* כפתור התנתקות
* בחירת ערוץ

ההצגה של הclientID מתנהלת על ידי useEffect

useEffect(() => {  
 setClientId(getClientId());  
}, []);

וuseEffect גם מוודא שכאשר יומאים מהpage אז מפסיק לשדר (אם משדר)

useEffect(() => {  
 return () => {  
 if (isTransmitting) {  
 stopTransmission();  
 }  
 };  
}, []);

**WebSocketUtils.tsx – אחראי על החיבור והפעלת הלוגיקה**

* מתחיל חיבור וניתוק על הסרבר
* מנהל את הID של הclient
* מנהל את האודיו (לא באופן אקטיבי רק קורא לפונקציות)

התחלת חיבור עם הclient מתרחש כאשר בConnectionPage לוחצים על חיבור, פה הסרבר מייד מחזיר ClientID (הסרבר מחזיר רק string אחד בכל מהלך התוכנית וזה הID):

if (typeof event.data === 'string') {  
 //the first and only text message is the client ID  
 clientId = event.data;  
 *console*.log('Received client ID:', clientId);

כפתורי ניהול:

1. Transmission – מפעיל את AudioService שמתחיל שידור בערת הפונקציה שקיבל בהתחלה:

AudioService.initAudioService(sendAudioChunk, handleTransmissionStop);

await AudioService.start(channel);

1. התנתק – הכפתור אחראי על הניתוק מהשרת, דואג להפסיק שידוק אם שידור קיים ומנקה את האודיו הקיים משליחת האדיו המלא.

export function disconnect(): void {  
 stopTransmission();  
 cleanupAudioContext();  
 if (socket) { socket.close(); socket = null; setSocket(null); }

1. העברת ערוץ – כאשר בmain מתקבל ערוץ הוא מובא על ידי מתשנה לפונקציה לפה ומשם מועבר לפונקציות שצריכות אותו.

export async function startTransmission(channel: number): Promise<void> {

WebSocketUtils מנהל את הclient ואת החיבור לwebsocket.

**AudioServiceUtils.tsx – אחראי בעיקר על הקלטת אודיו.**

* אחראי על הקלטת אודיו בזמן אמת
* אחראי על ההקלטת האודיו מלא

הקלטת האדיו מתחילה בלחצית על transmission בmain שקורא לwebSocketUtils שקורא לפה:

export const start = async (channel: number): Promise<void> => {

שימוש בpromise מאפשרת להשלים פעולות אסינכרוניות לפני פתרון.

שימוש בMediaDevices API

שימוש בWeb Audio API

let audioContext: AudioContext | null = null;

שימוש בMediaRecorder API

let mediaRecorder: MediaRecorder | null = null;

שליחת אודיו כל 46ms~ (2048 סמפל ב44.1kHz)

const source = audioContext.createMediaStreamSource(stream);  
const processor = audioContext.createScriptProcessor(2048, 1, 1);  
  
source.connect(processor);  
processor.connect(audioContext.destination);  
  
processor.onaudioprocess = (e) => {  
 const inputData = e.inputBuffer.getChannelData(0);  
 const audioData = new *Float32Array*(inputData);  
 onDataAvailable(audioData.buffer, channel);  
};

onDataAvailable היא פונקציה שמתקבלת מהwebSocketUtils והיא אחראית על שליחת מידע לסרבר.

בMediaRecorder יש הקלטה כל 10 מילישניות וזה נועד להקלטה הסופית שנמרת ב audioChunks.

בהוסף יש את הפונקציה שעוצרת הכל:

export const stop = (): void => {  
 if (mediaRecorder && mediaRecorder.state !== 'inactive') {  
 mediaRecorder.stop();  
 }  
 if (audioContext) {  
 audioContext.close();  
 audioContext = null;  
 }  
 onStop();  
};

הפונקציה עוצרת את העברת המידע לסרבר, מפסיקה את ההקלטה ומדיע לשאר התוכנית שזה קרה, ואז onStop מעבירה אתה ההקלטה המלאה (מפונקציה אחרת מFullAudioMakerUtils.tsx לא מאותה אחת כמו מAudioSender).

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise>

**audioSender.tsx – אחראי על שליחת המידע לסרבר.**

* אחראי על שליחת המידע בזמן אמת לסרבר
* אחראי על שליחת ההקלטה המלא

הדרך שבה השליחה עובדת הוא ככה הרכבת message מ:

1. ההדר שמורכב מהדר רגיל והערוץ בסוף (בצד השרת הסרבר מטבל בערוצים)
2. האורך
3. הנתונים

const header = new *Uint8Array*([0xAA, 0xAA, 0xAA, channel]);

const sampleRateBytes = new *Uint8Array*(new *Uint32Array*([sampleRate]).buffer);

const message = new *Uint8Array*(header.length + sampleRateBytes.length + int16Data.buffer.byteLength);

message.set(header);

message.set(sampleRateBytes, header.length);

message.set(new *Uint8Array*(int16Data.buffer), header.length + sampleRateBytes.length);

socket.send(message.buffer);

הקוד מעביר את המידע לint16 כי כעיקרון הוא שוקל פחות.

ההקלטה המלא קוראת נמצאת ב מFullAudioMakerUtils.tsx ומכינה את הmessage לכן פה צריך רק לשלוח:

export function sendFullAudio(fullAudio: ArrayBuffer): void {  
 if (socket && socket.readyState === *WebSocket*.*OPEN*) {  
 *console*.log(`Sending full audio of length: ${fullAudio.byteLength} bytes`);  
 socket.send(fullAudio);  
 }  
}

**FullAudioMakerUtils.tsx – הקובץ הזה אחראי על הכנת ההודעה שתישלח לסרבר, הוא יכיל את ההקלטה המלאה של השיחה.**

export const handleTransmissionStop = async (): Promise<void> => {  
 *console*.log('Audio transmission stopped');  
  
 const fullAudioBuffer = await AudioService.getFullAudioArrayBuffer();  
 if (fullAudioBuffer.byteLength === 0) {  
 *console*.log('No full audio data to send');  
 return;  
 }  
  
 const header = new *Uint8Array*([0xFF, 0xFF, 0xFF, 0xFF]);  
 const lengthBytes = new *Uint8Array*(new *Uint32Array*([fullAudioBuffer.byteLength]).buffer);  
 const message = new *Uint8Array*(header.length + lengthBytes.length + fullAudioBuffer.byteLength);  
 message.set(header);  
 message.set(lengthBytes, header.length);  
 message.set(new *Uint8Array*(fullAudioBuffer), header.length + lengthBytes.length);  
  
 sendFullAudio(message.buffer);  
  
 AudioService.clearAudioChunks();  
};

בדומה להודעה רגילה בזמן אמת, פה יש הכנת הודעה לשליחה לסרבר, ואז באמצעות פוניקצה שהובאה מ AudioSender(זאת שמקבלת רק message גמור) שולחים את ההודעה.

בנוסף הקוד מנקה את audioChunk לקראת ההקלטה הבאה.

**useAudioReceiver.tsx – הוא custom hook שנועד להשמיע את השמע שמגיע מהסרבר.**

הוא משומש בwebSocketUtils כאשר מתקבל הודעה שאינה string:

handleIncomingMessage(event);

function handleBinaryMessage(buffer: ArrayBuffer): void {  
 const data = new *Uint8Array*(buffer);  
 *console*.log(`Received binary message of length: ${data.length} bytes`);  
  
 if (data.length < 8) {  
 *console*.error('Received message is too short to be valid');  
 return;  
 }  
  
 if (data[0] === 0xAA && data[1] === 0xAA && data[2] === 0xAA) {  
 const receivedChannel = data[3];  
 const sampleRate = new *Uint32Array*(data.buffer, 4, 1)[0];  
 const audioData = data.slice(8);  
 *console*.log(`Received audio chunk: Channel ${receivedChannel}, Sample Rate ${sampleRate}, Length ${audioData.length}`);  
 playAudioData(audioData.buffer);  
 }   
 else   
 {  
 *console*.error('Received message with unknown header');  
 }  
}

פה יש בעיקר debug שלי, אבל העיקרון הוא playAudioData(audioData.buffer);

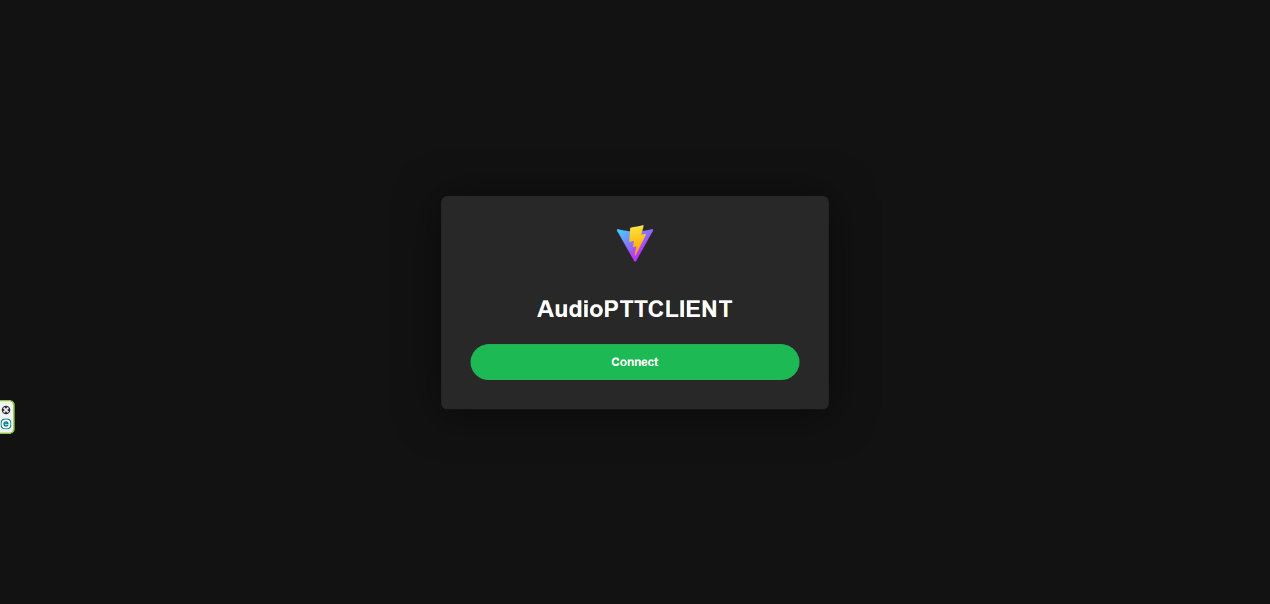
async function playAudioData(audioData: ArrayBuffer): Promise<void> {

הפונקציה מקבלת ArrayBuffer ומעבירה את האודיו מInt16 לFloat32 (כי web audio api משתמש בFloat32) ומשתמשת ב AudioBufferSourceNode בשביל לנגן את המידע.

cleanupAudioContext

מנהקה את ההקלטה לפעם הבאה.

Client connection:



התמונה אני אשנה מתישו

Client main

תמונה שמכילה צילום מסך, טקסט, תכונות מולטימדיה, תוכנה

התיאור נוצר באופן אוטומטי

שאר הקבצים שלא עברתי זה בעיקר כפתורים וכאלה.

**מבנה server:**

תמונה שמכילה טקסט, צילום מסך, גופן

התיאור נוצר באופן אוטומטי

Program – התפקדי רק להתחיל את הסרבר:

var baseDirectory = AppDomain.CurrentDomain.BaseDirectory;  
var clientManager = new ClientManager();  
var gridFsManager = new GridFsManager(baseDirectory);  
var transmitAudio = new TransmitAudio(clientManager);  
var receiveAudio = new ReceiveAudio(transmitAudio, gridFsManager);  
var webSocketServer = new WebSocketServer($"http://localhost:{Constants.**WebSocketServerPort**}/", clientManager, transmitAudio, receiveAudio);  
var serverOptions = new ServerOptions(clientManager, webSocketServer);  
var webSocketServerTask = webSocketServer.StartAsync();  
  
Console.WriteLine(Constants.**StartedConnection**);  
  
bool isRunning = true;  
while (isRunning)  
{  
 isRunning = await serverOptions.HandleInput();  
 await Task.Delay(100);  
}

הסבר מתחיל בprogram ואז מפעיל את ServerOptions שאחראי על הבחירות של הסברבר (מבחינת לחצנים על המקלדת).

ServerOptions – המחלקה הזאת היא המחלקה האחראית על האופציות בסרבר:

1. בלחיצה על Q – הסבר יפסיק
2. בליחצה על L – הסרבר ידפיס את רשימת המחוברים (לא רציך בתכלס זה סתם לעצמי לוודא שהכל בסדר)

switch (key.ToString().ToUpper())  
{  
 case Constants.**QuitCommand**:  
 await \_webSocketServer.StopAsync();  
 return false;  
 case Constants.**ListClientsCommand**:  
 \_clientManager.ListConnectedClients();  
 break;  
 default:  
 Console.WriteLine(Constants.**UnrecognizedKeyMessage**, key, Constants.**QuitCommand**, Constants.**ListClientsCommand**);  
 break;  
}

אין הרבה מה להסביר.

מחלקת Client

אחראית על שמירת הנתונים של כל client: שומרת את ה

1. Id – לזיהיו על ידי פונקציות סויימות
2. WebSocket – לתקשורת
3. ערוץ – הערוץ בוא נמצר הclient

אובייט פשוט ונחמד

public class Client  
{  
 public string Id { get; }  
 public System.Net.WebSockets.WebSocket WebSocket { get; }  
 public int Channel { get; set; }  
  
 public Client(string id, System.Net.WebSockets.WebSocket webSocket, int channel = 1)  
 {  
 Id = id;  
 WebSocket = webSocket;  
 Channel = channel;  
 }  
  
 }

WebSocketServer – אחראי על ההקשבה לחיבורים לסרבר, מקבל את המחלקות האחראיות על השמע:

private readonly HttpListener \_listener;  
 private readonly string \_url;  
 private readonly IClientManager \_clientManager;  
 private bool \_isRunning;  
 private readonly IReceiveAudio \_receiveAudio;  
 private readonly ITransmitAudio \_transmitAudio;

ReceiveAudio וTransmitAudio.

בנוסף את מנהל הclients הurl וגם מגדיר את listener שמקשיב להאזנות וכניסות לשרת.

הפונקציות:

* StartAsync() – אחראי על הקשבה לסרבר, מפעיל את listner לHTTP requests.

בנוסף הוא חאראי על הלופ המרכזי של התוכנית, תמיד מחכה לבקשות התקבלות, אם מתקבל בקשה הוא ממשיך לProcessWebSocketRequest, אם יש בעיה מדפיס בעיה

* StopAsync() – הופך את \_isRunning לfalse שאומר לסרבר להפסיק לקבל בקשות.
* ProcessWebSocketRequest – מנהל את הבקשה המתקבלת באמצעות WebSocketHandler

המחלקה הזאת היא בעצם הסרבר.

WebSocketHandler – המחלקה שאחראית על חיבור אישי של client

זו בעצם מחלקה שנוצרת כל פעם שclient חדש מתחבר, ככה שכאשר מחברים אותו הוא עושה כמו דברים:

1. יוצר Client חדש
2. שולח את הId שלו
3. מקבל ללא הפסקה הודעות מהClient
4. מעבד את המידע המגיע

הפונקציות:

1. HandleConnection – אחראית על יצירת אובייקט Client חדש ויצרת Id יחודי.
2. SendClientId – אחראי על שליחת הId של הclient לסרבר.
3. ProcessMessages – מפעיל לולאה איןסופית לקבלת הודעות מהclient
4. ProcessAudioMessage – בודק האם הודעה תקינה, בנוסף בודק את סוגה (הודעה של שידור ישיר או הקלטה מלא)

אם מדובר בשידור אז:

await \_transmitAudio.BroadcastAudioAsync(client, audioData, audioLength);

אם מדובר בהקלטה

await \_receiveAudio.HandleFullAudioTransmissionAsyncWebSockets(client, audioData);

המחלקה הזאת אחראית על הקבלה של הclient החדש, יוצרת אותו, ומוודאת שאפשר לקבל ממנו אודיו.

ClientManager – המחלקה האחראית על ניהול הclients המחוברים לשרת

המחלקה אחראית על החזקת הclients המחוברים וניהולם באמצעות רשימה של clients:

private readonly List<Client> clients = new List<Client>();

הפונקציות:

1. AddClient – מקבל client ומכניס אותו לרשימה, את הclient מקבל מהמחלקה הקודמת WebSocketHandler:

var client = new Client(Guid.NewGuid().ToString(), webSocket);  
 \_clientManager.AddClient(client);

1. RemoveClient – כאשר client מתנתק הוא מוציא אותו מהרשימה וסוגר קישור עם הWebSocket ומדפיס על ההתנתקות:

clients.Remove(client);  
 Console.WriteLine(Constants.**ClientDisconnectedMessage**, client.Id);  
   
 client.WebSocket?.CloseAsync(WebSocketCloseStatus.**NormalClosure**, Constants.**ClientDisconnectReason**, CancellationToken.None).Wait();

1. GetAllClients – מחזיר את רשימת הclients
2. ListConnectedClients – מדפיס את כל הclients המחוברים.

Console.WriteLine(Constants.**ClientInfoMessage**, client.Id, client.Channel);

בקיצור, אחראית על הclients.

TransmitAudio – המחלקה האחראית על שידור האודיו ללקוחות

המחלקה מקבלת את ClientManager בשביל גישה לclients.

פונקציות:

1. BroadcastAudioAsync – אחראי על מצאית הלקוחות שבואותו ערוץ והפעלת SendAudioToClientAsync עלייהם:

var clientsOnSameChannel = \_clientManager.GetAllClients()  
 .Where(client => client.Id != sender.Id && client.Channel == sender.Channel);  
  
var broadcastTasks = clientsOnSameChannel  
 .Select(client => SendAudioToClientAsync(client, audioData, length, (byte)sender.Channel));

1. SendAudioToClientAsync – מקבל את ההודעה שצריך לשלוח ושולח ללקוח.

await client.WebSocket.SendAsync(new ArraySegment<byte>(messageToSend), WebSocketMessageType.**Binary**, true, CancellationToken.None);

בקיצור אחראית על שליחת ההודעות, בדומה לclient בreact גם פה הוא שולח message שמורכב מהדר, אורך, ונתונים

ReceiveAudio – המחלקה האחראית על קבלת האודיו מהלקוחות

המחלקה מקבלת את ITransmitAudio כי אם התקבל הודעה רגילה צריך לשלוח רגיל בהתאם לערוץ, ו IGridFsManagerאם התקבלה ההקלטה המלאה צריך לשמור אותה בממסד נתונים.

פונקציות:

1. HandleRealtimeAudioAsyncWebSockets – משתשמ ב \_transmitAudio

בשביל לשלוח את ההודעה הקלטה אנאערף שהתקבלה.

await \_transmitAudio.BroadcastAudioAsync(sender, audioData, audioData.Length);

1. HandleFullAudioTransmissionAsyncWebSockets – משתמש ב\_gridFsManager בשביל לשמור את ההקלטה

string filename = $"full\_audio\_{DateTime.UtcNow:yyyyMMddHHmmss}\_{client.Id}.wav";  
await \_gridFsManager.SaveAudioAsync(filename, audioData);

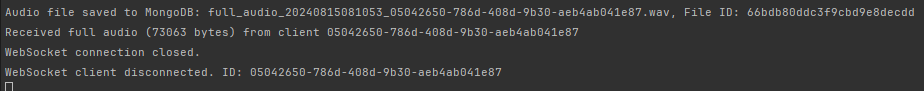
הקובץ ישמר כ.wav עם התאריך שהתקבל, בנוסף לclient שממנו הגיעה ההקלטה הזאת.

GridFsManager – המחלקה האחראית על שמירת קבצי אודיו

המחלקה הזאת פשוטה, היא מקבלת קובץ אודיו ושומר בדאטה בייס.

פונקציות:

1. SaveAudioAsync – מקבלת שם קובץ ושומרת במונגו ובלוקל פיילס.



תמונה שמכילה טקסט, תוכנה, צילום מסך

התיאור נוצר באופן אוטומטי

