ChatGPT

MVP Audio Capture and Playback Web App

1. Microphone Capture

Use the MediaDevices API to prompt the user for microphone access. For example, calling `navigator.mediaDevices.getUserMedia({ audio: true })` will ask permission and, on success, return a **MediaStream** containing the mic's audio track [1]. This API only works in secure contexts (HTTPS) [2]. Once obtained, connect the stream into a Web Audio graph: e.g., create an `AudioContext` and use `audioContext.createMediaStreamSource(stream)` to feed the mic input into your processing pipeline.

- The `getUserMedia()` call returns a promise that resolves to a `MediaStream` whose tracks include the requested audio (the microphone) [1].
- If the user denies access or no mic is available, handle the rejected promise (e.g., show an error).
- Ensure the page is served over HTTPS (or `localhost`) because media capture APIs require secure context [2].

2. Low-Latency Audio Processing

For real-time audio I/O, use an **AudioWorklet**. An `AudioWorklet` runs its processing code on a separate thread for very low latency [3]. In the main thread, load and attach an AudioWorklet processor module after creating the `AudioContext`:

```
const audioContext = new AudioContext();
// Load the AudioWorklet processor (in a separate JS file)
await audioContext.audioWorklet.addModule('processor.js');
// Create the node and connect the mic source to it
const micSource = audioContext.createMediaStreamSource(stream);
const workletNode = new AudioWorkletNode(audioContext, 'my-processor');
micSource.connect(workletNode);
```

Inside the AudioWorklet's `process(inputs, outputs)` method, you can read raw PCM samples from the input. To prepare data for sending, typically:
1. **Convert to mono and resample:** Mix channels if needed and downsample to 16 kHz PCM (the backend's expected rate). Web Audio runs at the hardware sample rate (often 48000 Hz), so you must resample. One approach is to implement a resampler inside the worklet (e.g. using a library like libsamplerate) [4]. The AudioWorklet is ideal for CPU-heavy tasks like resampling [4].
2. **Frame and send:** Bundle the PCM into fixed-size chunks (20–30 ms, i.e. ~320–480 samples at 16 kHz). Post these chunks to the main thread (via `workletNode.port.postMessage`) or directly send them over the network.

For example, in the worklet you might do:

```
class MyProcessor extends AudioWorkletProcessor {
  process(inputs) {
    const input = inputs[0][0]; // Float32Array of incoming samples
    // TODO: downsample to 16 kHz and convert to Int16 if needed
    // Accumulate 20–30ms of samples and post to main thread
    this.port.postMessage({ audioChunk: downsampledChunk });
    return true;
  }
}
registerProcessor('my-processor', MyProcessor);
```

3. Network Transport (WebSocket vs WebRTC)

To send the audio to a backend, start with a simple binary WebSocket. For example:
1. Open a WebSocket to the server (`const socket = new WebSocket(url); socket.binaryType = 'arraybuffer';`).
2. Each time a 20–30ms audio chunk is ready, send it: `socket.send(audioChunkBuffer)`. On the server, process or relay these bytes.
3. The server can then return audio frames (e.g. translated speech) over the same socket for client playback.

For more robust real-time streaming (e.g. across NATs or when low jitter is needed), consider WebRTC. Python's aiortc library provides a full WebRTC implementation [5]. WebRTC handles NAT/firewall traversal using ICE with STUN/TURN servers [6], and includes built-in jitter buffering and adaptive media streaming. In that mode, the browser and the aiortc server form a peer connection to exchange audio RTP streams rather than using WebSockets. Using WebRTC requires extra signaling and setup, but it provides features like latency control and re-negotiation of devices automatically.

4. Audio Playback in Browser

On receiving PCM audio from the server (assumed 16 kHz mono), feed it into the Web Audio API for playback. A common method is:
1. Create an `AudioContext` (you may set `{ sampleRate: 16000 }` to match the data, but browsers may override it).
2. For each incoming audio frame, create or reuse an `AudioBuffer`:
- Use `audioContext.createBuffer(1, frameLength, 16000)` to make a buffer of the correct length and rate.
- Copy the PCM data into the buffer's channel data.
3. Create an `AudioBufferSourceNode`, set its buffer to this `AudioBuffer`, connect it to `audioContext.destination`, and call `start()` to play.

```
const audioCtx = new AudioContext();
function playPCM(pcmArray) {
```

```
  const buffer = audioCtx.createBuffer(1, pcmArray.length, 16000);
  buffer.copyToChannel(pcmArray, 0);
  const src = new AudioBufferSourceNode(audioCtx, { buffer });
  src.connect(audioCtx.destination);
  src.start();
}
```

This will sequentially play each chunk as it arrives. The `AudioBufferSourceNode` approach is recommended by MDN for playing raw audio data [7].

5. Output Device Selection

Allowing the user to choose the audio output (e.g. speakers vs headphones) improves flexibility. The MediaDevices API provides an experimental `selectAudioOutput()` method to prompt the user to pick a device [8]. After the user selects a device, you receive its `deviceId` and can route audio there using `HTMLMediaElement.setSinkId()` [9]. For example:

```
// Prompt user to pick an output
const deviceInfo = await navigator.mediaDevices.selectAudioOutput();
// Create an audio element or reuse one
const audio = new Audio();
await audio.setSinkId(deviceInfo.deviceId);
audio.srcObject = myOutputStream;
audio.play();
```

According to MDN, `selectAudioOutput()` "prompts the user to select an audio output device" [8] and `setSinkId()` "sets the ID of the audio device to use for output" [9]. Note these require HTTPS and user interaction, and support varies across browsers (they are still marked experimental). As a fallback, simply using the default output is fine on unsupported platforms.

6. Screen/Tab Audio Capture

To translate audio from another app (e.g. a Zoom call or a web video), use `getDisplayMedia`. Calling `navigator.mediaDevices.getDisplayMedia({ audio: true, video: true })` lets the user share a screen, window, or tab and capture its audio track [10]. However, **OS support varies**: on Windows and ChromeOS the browser can capture full system audio, but on macOS and Linux only the shared tab's (or application's) audio is captured [11]. In practice this means instruct users on Mac/Linux to share the specific browser tab containing the audio.

- Example usage:

```
const stream = await navigator.mediaDevices.getDisplayMedia({ audio: true,
video: false });
```

```
// The returned stream.getAudioTracks() may include system or tab audio
depending on OS.
```

- Keep in mind user permissions: the browser will show a “Share audio” checkbox on supported platforms. If system audio isn’t available, you may only get the tab’s sound.

7. Compatibility and Requirements

Most of the above APIs work in modern browsers: `getUserMedia()` and `AudioWorklet` are widely supported [1] [3]. All media capture and sink-selection APIs require HTTPS (or localhost) [2] [12]. For browser support details, consult MDN or CanIUse (e.g. the audio-capture with `getDisplayMedia` page notes Windows/ChromeOS vs macOS differences [11]). Testing on target platforms (Windows, ChromeOS, macOS, Linux) is essential, especially for features like output-device selection and system audio capture that are still experimental.

Summary: The MVP pipeline is: (1) capture mic (or screen) audio via `getUserMedia()`/ `getDisplayMedia()`, (2) process it with an `AudioWorklet` (resampling to 16 kHz and framing), (3) send over WebSocket (or WebRTC/aiortc) to the server, (4) receive processed PCM, (5) play it in-browser with the Web Audio API, and (6) let the user choose the output device via `selectAudioOutput()`/`setSinkId()`. This covers core functionality: microphone capture, low-latency streaming, translation (on the server), and flexible playback [3] [8] [11].

Sources: Browser APIs and specs from MDN and StackOverflow as cited above, e.g. MDN `getUserMedia()` [1], AudioWorklet docs [3], output device APIs [8] [9], and notes on screen-audio capture [11].

---

[1] [2] MediaDevices: getUserMedia() method
https://developer.mozilla.org/en-US/docs/Web/API/MediaDevices/getUserMedia

[3] AudioWorklet
https://developer.mozilla.org/en-US/docs/Web/API/AudioWorklet

[4] Resampling audio via a Web Audio API Audio Worklet - 0110.be
https://0110.be/posts/Resampling_audio_via_a_Web_Audio_API_Audio_Worklet

[5] GitHub - aiortc/aiortc: WebRTC and ORTC implementation for Python using asyncio
https://github.com/aiortc/aiortc

[6] Introduction to WebRTC protocols
https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Protocols

[7] AudioBuffer - Web APIs | MDN
https://developer.mozilla.org/en-US/docs/Web/API/AudioBuffer

[8] [12] MediaDevices: selectAudioOutput() method - Web APIs | MDN
https://developer.mozilla.org/en-US/docs/Web/API/MediaDevices/selectAudioOutput

[9] HTMLMediaElement: setSinkId() method - Web APIs | MDN
https://developer.mozilla.org/en-US/docs/Web/API/HTMLMediaElement/setSinkId

10 MediaDevices: getDisplayMedia() method
https://developer.mozilla.org/en-US/docs/Web/API/MediaDevices/getDisplayMedia

11 javascript - Audio capture with getDisplayMedia is not worked with Chrome in my Macbook - Stack Overflow
https://stackoverflow.com/questions/71766536/audio-capture-with-getdisplaymedia-is-not-worked-with-chrome-in-my-macbook