FREE eBook

LEARNING web-audio

Free unaffiliated eBook created from
Stack Overflow contributors.

#web-audio

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: web-audio

It is an unofficial and free web-audio ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official web-audio.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with web-audio

Remarks

The Web Audio API is a W3C standard for lower level access to the audio system than the standard `<audio>`-tag, via a high level API.

Use cases includes games, art, audio synthesis, interactive applications, audio production and any application where fine-grained control of the audio data is required.

The API can accept input from a number of sources, including loading audio files and decoding them via the API or `<audio>`-elements. It also provides facilities to generate sound directly via the API through the use of oscillator nodes.

There is also a number of processing nodes, such as gains, delays and script processors (which eventually will be deprecated and replaced by more efficient nodes). These can in turn be used to build more complex effects and audio graphs.

Examples

Synthesising audio

In this example we show how to generate a simple sine wave, and output it on the user's speakers/headphones.

```
let audioContext = new (window.AudioContext || window.webkitAudioContext)();

let sourceNode = audioContext.createOscillator();
sourceNode.type = 'sine';
sourceNode.frequency.value = 261.6;
sourceNode.detune.value = 0;

//Connect the source to the speakers
sourceNode.connect(audioContext.destination);

//Make the sound audible for 100 ms
sourceNode.start();
window.setTimeout(function() { sourceNode.stop(); }, 100);
```

The `start` and `stop` methods of the `sourceNode` variable above each have an optional parameter `when` that specifies how many **seconds** to wait before starting or stopping.

So, an alternative way to stopping the sound would be:

```
sourceNode.start();
sourceNode.stop(0.1);
```

The `type` parameter of an oscillator node can be set to any of the following values:

- sine (default)
- square
- sawtooth
- traingle
- a custom wave

Custom waves are `PeriodicWaves` and can be created using the `AudioContext.createPeriodicWave` method.

Using effects on audio

Effects can be applied to audio by chaining nodes between the source and the destination node. In this example we use a gain node to mute the source, and only let sound through at specific times. This allows us to create morse code.

```
function morse(gainNode, pattern) {
  let silenceTimeout = 300;
  let noiseTimeout;

  if(pattern === '') {
    //We are done here
    return;
  } else if(pattern.charAt(0) === '.') {
    noiseTimeout = 100;
  } else if(pattern.charAt(0) === '-') {
    noiseTimeout = 400;
  } else {
    console.error(pattern.charAt(0), ': Character not recognized.');
    return;
  }

  //Briefly let sound through this gain node
  gainNode.gain.value = 1;
  window.setTimeout(function() {
    gainNode.gain.value = 0;
    window.setTimeout(morse, silenceTimeout, gainNode, pattern.substring(1));
  }, noiseTimeout);
}
```

```
let audioContext = new (window.AudioContext || window.webkitAudioContext)();

let sourceNode = audioContext.createOscillator();
let gainNode = audioContext.createGain();

sourceNode.type = 'sine';
sourceNode.frequency.value = 261.6;
sourceNode.detune.value = 0;

//Mute sound going through this gain node
gainNode.gain.value = 0;

//SourceNode -> GainNode -> Speakers
sourceNode.connect(gainNode);
gainNode.connect(audioContext.destination);

//The source node starts outputting
```

```
sourceNode.start();

//Output SOS
morse(gainNode, '...---...');
```

Recording audio from a microphone source

In order to record audio from a user's microphone, we must first gain permission from the user to access the device:

```
navigator.mediaDevices.getUserMedia({ audio: true })
    .then(successCallback)
    .catch(failureCallback);
```

On success, our `successCallback` will be called with a `MediaStream` object which we can use to access the microphone:

```
var audioContext = new (window.AudioContext || window.webkitAudioContext)();
// Create a source from our MediaStream
var source = audioContext.createMediaStreamSource(mediaStream);
// Now create a Javascript processing node with the following parameters:
// 4096 = bufferSize (See notes below)
// 2 = numberOfInputChannels (i.e. Stereo input)
// 2 = numberOfOutputChannels (i.e. Stereo output)
var node = audioContext.createScriptProcessor(4096, 2, 2);
node.onaudioprocess = function(data) {
    console.log(data);
}
// Connect the microphone to the script processor
source.connect(node);
node.connect(audioContext.destination);
```

The `onaudioprocess` event gives us access to the Raw PCM data stream from the microphone. We can access the buffered data like so:

```
node.onaudioprocess = function(data) {
    var leftChannel = data.inputBuffer.getChannelData(0).buffer;
    var rightChannel = data.inputBuffer.getChannelData(1).buffer;
}
```

When we're finished recording, we must disconnect from the source and discard the script processor:

```
node.disconnect();
source.disconnect();
```

Important Notes about `bufferSize`

The `bufferSize` determines how frequently the `onaudioprocess` callback is called. Lower values result in lower (better) latency, higher values will reduce audio breakup/glitches. A value of zero will allow the browser implementation to choose an appropriate value. If passed in manually, it must be one of the following values: 256, 512, 1024, 2048, 4096, 8192, 16384.

Playing audio

To play audio using the Web Audio API, we need to get an ArrayBuffer of audio data and pass it to a BufferSource for playback.

To get an audio buffer of the sound to play, you need to use the `AudioContext.decodeAudioData` method like so:

```
const audioCtx = new (window.AudioContext || window.webkitAudioContext)();
// Fetch the MP3 file from the server
fetch("sound/track.mp3")
    // Return the data as an ArrayBuffer
    .then(response => response.arrayBuffer())
    // Decode the audio data
    .then(buffer => audioCtx.decodeAudioData(buffer))
    .then(decodedData => {
        // ...
    });
```

When the final promise resolves, you'll be given the audio in the form of an `AudioBuffer`. This can then be attached to an `AudioBufferSourceNode` - and played - like so:

```
const source = context.createBufferSource();
source.buffer = decodedData; // <==
source.start(...);
```

Where `.start()` has three parameters that offset when to play the sample, offset where in the sample to play from, and tell how long to play the sample, respectively.

More information about how to manipulate the buffer source can be found on MDN.

Realtime altering of two audio sources

This example shows how to use two audio sources, and alter one of them based on the other. In this case we create an audio Ducker, that will lower the volume of the primary track if the secondary track produces sound.

The ScriptProcessorNode will send regular events to its audioprocess handler. In this handler, which is linked to the secondary audio source, we calculate the "loudness" of the audio, and use it to alter a dynamic compressor on the primary audio source. Both are then sent to the speakers/headphones of the user. The result is very abrupt volume changes in the primary audio track when sound is detected in the secondary audio track. We could make this smoother by using an average, and using a delay line to change the volume before the secondary audio is detected, but the process should be clear in this example.

```
//The DuckingNode will heavily compress the primary source when the secondary source produces
sound
class DuckingNode {
  constructor(context) {
    let blocksize = 2048;
    let normalThreshold = -50;
```

```
    let duckThreshold = 0.15;

    //Creating nodes
    this.compressor = context.createDynamicsCompressor();
    this.processor = context.createScriptProcessor(blocksize, 2, 2);

    //Configuring nodes
    this.compressor.threshold.value = normalThreshold;
    this.compressor.knee.value = 0;
    this.compressor.ratio.value = 12;
    this.compressor.reduction.value = -20;
    this.compressor.attack.value = 0;
    this.compressor.release.value = 1.5;

    let self = this;
    this.processor.onaudioprocess = function(audioProcessingEvent) {
      let inputBuffer = audioProcessingEvent.inputBuffer;
      let outputBuffer = audioProcessingEvent.outputBuffer;
      let rms;
      let total = 0.0;
      let len = blocksize * outputBuffer.numberOfChannels;

      for (let channel = 0; channel < outputBuffer.numberOfChannels; channel++) {
        let inputData = inputBuffer.getChannelData(channel);
        let outputData = outputBuffer.getChannelData(channel);

        for (let sample = 0; sample < inputBuffer.length; sample++) {
          // make output equal to the same as the input
          outputData[sample] = inputData[sample];
          total += Math.abs(inputData[sample]);
        }
      }

      //Root mean square
      rms = Math.sqrt( total / len );

      //We set the threshold to at least 'normalThreshold'
      self.compressor.threshold.value = normalThreshold + Math.min(rms - duckThreshold, 0) * 5
* normalThreshold;
    }
  }

  get primaryInput () {
    return this.compressor;
  }

  get secondaryInput () {
    return this.processor;
  }

  connectPrimary(node) {
    this.compressor.connect(node);
  }

  connectSecondary(node) {
    this.processor.connect(node);
  }
};
```

```
let audioContext = new (window.AudioContext || window.webkitAudioContext)();
```

```
//Select two <audio> elements on the page. Ideally they have the autoplay attribute
let musicElement = document.getElementById("music");
let voiceoverElement = document.getElementById("voiceover");

//We create source nodes from them
let musicSourceNode = audioContext.createMediaElementSource(musicElement);
let voiceoverSourceNode = audioContext.createMediaElementSource(voiceoverElement);

let duckingNode = new DuckingNode(audioContext);

//Connect everything up
musicSourceNode.connect(duckingNode.primaryInput);
voiceoverSourceNode.connect(duckingNode.secondaryInput);
duckingNode.connectPrimary(audioContext.destination);
duckingNode.connectSecondary(audioContext.destination);
```

Part of this example comes from this answer by Kevin Ennis and the code on mdn.

Setup

We start off by creating an audio context and then create an oscillator which is the easiest way to check that your setup works. (Example fiddle)

```
// We can either use the standard AudioContext or the webkitAudioContext (Safari)
var audioContext = (window.AudioContext || window.webkitAudioContext);

// We create a context from which we will create all web audio nodes
var context = new audioContext();

// Create an oscillator and make some noise
var osc = context.createOscillator();

// set a frequecy, in this case 440Hz which is an A note
osc.frequency.value = 440;

// connect the oscillator to the context destination (which routes to your speakers)
osc.connect(context.destination);

// start the sound right away
osc.start(context.currentTime);

// stop the sound in a second
osc.stop(context.currentTime + 1);
```

Read Getting started with web-audio online: https://riptutorial.com/web-audio/topic/6984/getting-started-with-web-audio

Credits

S. No	Chapters	Contributors
1	Getting started with web-audio	CodingIntrigue, Community, Mike McCaughan, Mikhail, Oskar Eriksson, SirPython, Sumurai8, thanksd