Introduction to Web Audio API

BY GREG HOVANESYAN ON MARCH 6, 2017
WEB AUDIO API

Web Audio API lets us make sound right in the browser. It makes your sites, apps, and games more fun and engaging. You can even build music-specific applications like drum machines and synthesizers. In this article, we'll learn about working with the Web Audio API by building some fun and simple projects.

Getting Started

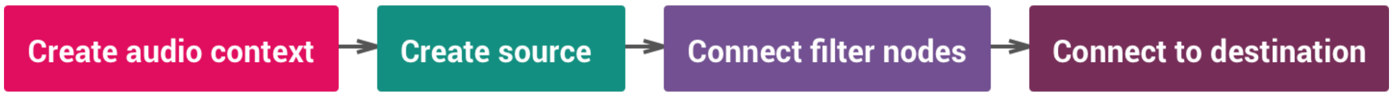
Let's do some terminology. All audio operations in Web Audio API are handled inside an *audio context*. Each basic audio operation is performed with audio *nodes* that are chained together, forming an *audio routing graph*. Before playing any sound, you'll need to create this audio context. It is very similar to how we would create a context to draw inside with the `<canvas>` element. Here's how we create an audio context:

JS

```js
var context = new (window.AudioContext || window.webkitAudioContext)();
```

Safari requires a webkit prefix to support AudioContext, so you should use that line instead of `new AudioContext();`

Normally the Web Audio API workflow looks like this:



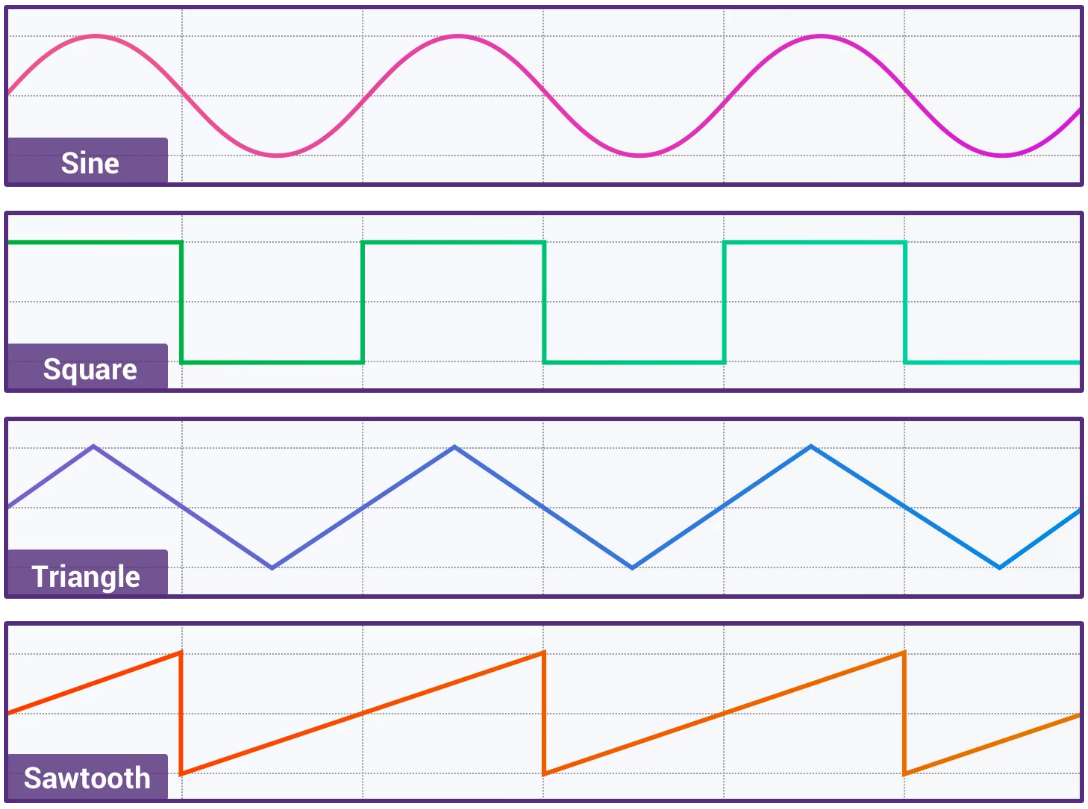
create source -> connect filter nodes -> connect to destination" />

There are three types of sources:

1. Oscillator - mathematically computed sounds
2. Audio Samples - from audio/video files
3. Audio Stream - audio from webcam or microphone

Let's start with the oscillator

An oscillator is a repeating waveform. It has a frequency and peak amplitude. One of the most important features of the oscillator, aside from its frequency and amplitude, is the shape of its waveform. The four most commonly used oscillator waveforms are sine, triangle, square, and sawtooth.



It is also possible to create custom shapes. Different shapes are suitable for different synthesis techniques and they produce different sounds, from smooth to harsh.

The Web Audio API uses `OscillatorNode` to represent the repeating waveform. We can use all of the above shown waveform shapes. To do so, we have to assign the value property like so:

```js
OscillatorNode.type = 'sine'|'square'|'triangle'|'sawtooth';
```

You can create a custom waveform as well. You use the `setPeriodicWave()` method to create the shape for the wave, that will automatically set the type to custom. Let's listen how different waveforms produce different sounds:

See the Pen.

Custom waveforms are created using *Fourier Transforms*. If you want to learn more about custom waveform shapes (like how to make a police siren, for example) you can learn it from this good resource.

Running the oscillator

Let's try to make some noise. Here's what we need for that:

1. We have to create a Web Audio API context
2. Create the oscillator node inside that context
3. Choose waveform type
4. Set frequency
5. Connect oscillator to the destination
6. Start the oscillator

Let's convert those steps into code.

JS

```
var context = new (window.AudioContext || window.webkitAudioContext)();

var oscillator = context.createOscillator();

oscillator.type = 'sine';
oscillator.frequency.value = 440;
oscillator.connect(context.destination);
oscillator.start();
```

Note how we define the audio context. Safari requires the `webkit` prefix, so we make it cross-browser compatible.

Then we create the oscillator and set the type of the waveform. The default value for type is `sine` , so you can skip this line, I just like to add it to make it more clear and easy to update. We set the frequency value to 440, which is the A4 note (which is also the default value). The frequencies of musical notes C0 to B8 are in the range of 16.35 to 7902.13Hz. We will check out an example where we play a lot of different notes later in this article.

Now when we know all of that, let's make the volume adjustable as well. For that we need to create the gain node inside of the context, connect it to the chain, and connect gain to the destination.

JS

```
var gain = context.createGain();
oscillator.connect(gain);
gain.connect(context.destination);

var now = context.currentTime;
gain.gain.setValueAtTime(1, now);
gain.gain.exponentialRampToValueAtTime(0.001, now + 0.5);
```

```
oscillator.start(now);
oscillator.stop(now + 0.5);
```

Now you have some knowledge of working with the oscillator, here's a good exercise. This Pen has the oscillator code setup. Try to make a simple app that changes the volume when you move the cursor up and down your screen, and changes the frequency when you move the cursor left and right.

Timing of Web Audio API

One of the most important things in building audio software is managing *time*. For the precision needed here, using the JavaScript clock is not the best practice, because it's simply not precise enough. However the Web Audio API comes with the `currentTime` property, which is an increasing double hardware timestamp, which can be used for scheduling audio playback. It starts at 0 when the audio context is declared. Try running `console.log(context.currentTime)` to see the timestamp.

For example, if you want the Oscillator to play immediately you should run `oscillator.start(0)` (you can omit the 0, because it's the default value). However you may want it to start in one second from now, play for two seconds, then stop. Here's how to do that:

JS

```
var now = context.currentTime;
oscillator.play(now + 1);
oscillator.stop(now + 3);
```

There are two methods to touch on here.

The `AudioParam.setValueAtTime(value, startTime)` method schedules change of the value at the precise time. For example, you want to change frequency value of the oscillator in one second:

JS

```
oscillator.frequency.setValueAtTime(261.6, context.currentTime + 1);
```

However, you also use it when you want to instantly update the value, like `.setValueAtTime(value, context.currentTime)` . You can set the value by modifying the value property of the `AudioParam` , but any updates to the value are ignored without throwing an exception if they happen at the same moment as the automation events (events scheduled using `AudioParam` methods).

The `AudioParam.exponentialRampToValueAtTime(value, endTime)` method schedules gradual change of the value. This code will exponentially decrease the volume of the oscillator in one second, which is a good way to stop sound smoothly:

JS

```
gain.gain.exponentialRampToValueAtTime(0.001, context.currentTime + 1);
```

We can't use 0 as the value because the value needs to be positive, so we use a very small value instead.

Creating the Sound class

Once you stop an oscillator, you cannot start it again. You didn't do anything wrong, it's the feature of the Web Audio API that optimizes the performance. What we can do is to create a sound class that will be responsible from creating oscillator nodes, and play and stop sounds. That way we'll be able to call the sound multiple times. I'm going to use ES6 syntax for this one:

JS

```
class Sound {

  constructor(context) {
    this.context = context;
  }

  init() {
    this.oscillator = this.context.createOscillator();
    this.gainNode = this.context.createGain();

    this.oscillator.connect(this.gainNode);
    this.gainNode.connect(this.context.destination);
    this.oscillator.type = 'sine';
  }

  play(value, time) {
    this.init();

    this.oscillator.frequency.value = value;
    this.gainNode.gain.setValueAtTime(1, this.context.currentTime);

    this.oscillator.start(time);
    this.stop(time);
  }

  stop(time) {
    this.gainNode.gain.exponentialRampToValueAtTime(0.001, time + 1);
    this.oscillator.stop(time + 1);
  }

}
```

We pass the context to the constructor, so we can create all of the instances of the `Sound` class within same context. Then we have the `init` method, that creates the oscillator and all of the necessary filter nodes, connects them, etc. The `Play` method accepts the value (the frequency in hertz of the note it's going to play) and the time when it shall be played. But first, it creates the oscillator, and that happens every time we call the `play` method. The `stop` method exponentially decreases the volume in one second until it stops the oscillator completely. So whenever we need to play the sound again, we create a new instance of the `sound` class and call the play method. Now we can play some notes:

JS

```
let context = new (window.AudioContext || window.webkitAudioContext)();
let note = new Sound(context);
let now = context.currentTime;
note.play(261.63, now);
note.play(293.66, now + 0.5);
note.play(329.63, now + 1);
note.play(349.23, now + 1.5);
note.play(392.00, now + 2);
note.play(440.00, now + 2.5);
note.play(493.88, now + 3);
note.play(523.25, now + 3.5);
```

That will play C D E F G A B C, all within the same context. If you want to know the frequencies of notes in hertz, you can find them here.

Knowing all of this makes us able to build something like a xylophone! It creates a new instance of `Sound` and plays it on `mouseenter` . You can check the example and try make one by yourself as an exercise.

See the Pen Play the Xylophone (Web Audio API) by Greg Hovanesyan (@gregh) on CodePen.

I've created a playground, containing all the required HTML and CSS, and the `Sound` class we've created. Use the `data-frequency` attribute to obtain the note values. Try here.

Working with a recorded sound

Now that you've built something with an oscillator, let's now see how to work with a recorded sound. Some sounds are very hard to reproduce using the oscillator. In order to use realistic sounds in many cases, you'll *have* to use recorded sounds. This can be `.mp3`, `.ogg`, `.wav`, etc. See the full list for more info. I like to use `.mp3` as it's lightweight, widely supported, and has pretty good sound quality.

You can't simply get sound by a URL like you do with images. We have to run an `XMLHttpRequest` to get the files, decode the data, and put into the buffer.

JS

```js
class Buffer {

  constructor(context, urls) {
    this.context = context;
    this.urls = urls;
    this.buffer = [];
  }

  loadSound(url, index) {
    let request = new XMLHttpRequest();
    request.open('get', url, true);
    request.responseType = 'arraybuffer';
    let thisBuffer = this;
    request.onload = function() {
      thisBuffer.context.decodeAudioData(request.response, function(buffe
        thisBuffer.buffer[index] = buffer;
        updateProgress(thisBuffer.urls.length);
        if(index == thisBuffer.urls.length-1) {
          thisBuffer.loaded();
        }
      });
    };
    request.send();
  };
  loadAll() {
    this.urls.forEach((url, index) => {
      this.loadSound(url, index);
    })
  }

  loaded() {
    // what happens when all the files are loaded
  }

  getSoundByIndex(index) {
    return this.buffer[index];
  }

}
```

Let's take a look at the constructor. We receive our context there as we did in the `Sound` class, receive the list of URLa that will be loaded, and an empty array for the buffer.

The we have two methods: `loadSound` and `loadAll` . `loadAll` loops through the list of URLs and calls the `loadSound` method. It's important to pass the index, so that we put the buffered sound into the correct element of the array, regardless of which request loads first. This also let's us see which request is the last, which means that on its completion the buffer is loaded.

Then you can call the `loaded()` method, which can do something like hiding the loading indicator. And finally the `getSoundByIndex(index)` method gets the sound from the buffer by index for playback.

The `decodeAudioData` method has a newer Promise-based syntax, but it doesn't work in Safari yet:

JS

```
context.decodeAudioData(audioData).then(function(decodedData) {
  // use the decoded data here
});
```

Then we have to create the class for the sound. Now we have our complete class to work with the recorded sound:

JS

```
class Sound() {

  constructor(context, buffer) {
    this.context = context;
    this.buffer = buffer;
  }

  init() {
    this.gainNode = this.context.createGain();
    this.source = this.context.createBufferSource();
    this.source.buffer = this.buffer;
    this.source.connect(this.gainNode);
    this.gainNode.connect(this.context.destination);
  }

  play() {
    this.setup();
    this.source.start(this.context.currentTime);
  }

  stop() {
    this.gainNode.gain.exponentialRampToValueAtTime(0.001, this.context.c
    this.source.stop(this.context.currentTime + 0.5);
  }

}
```

The constructor accepts the context and the buffer. We create by calling `createBufferSource()` method, instead of `createOscillator` as we did before. The buffer is the note (element from the buffer array) that we get using the `getSoundByIndex()` method. Now instead of the oscillator we create a buffer source, set the buffer, and then connect it to the destination (or gain and other filters).

JS

```
let buffer = new Buffer(context, sounds);
buffer.loadAll();
```

```
sound = new Sound(context, buffer.getSoundByIndex(id));
sound.play();
```

Now we have to create an instance of buffer and call the `loadAll` method, to load all of the sounds into the buffer. We also have the `getSoundById` method to grab the exact sound we need, so we pass the sound to the `Sound` and call `play()` . The `id` can be stored as a data attribute on the button that you click to play the sound.

Here's a project that uses all of that: the buffer, the recorded notes, etc:

See the Pen The Bluesman - You Can Play The Blues (Web Audio API) by Greg Hovanesyan (@gregh) on CodePen.

You can use that example for for reference, but for your own exercise, here's a playground I've created. It has all the necessary HTML and CSS and the URLs to the notes that I have recorded on a real electric guitar. Try writing your own code!

Intro to Filters

The Web Audio API lets you add different filter nodes between your sound source and destination. `BiquadFilterNode` is a simple low-order filter which gives you control over what parts of the frequency parts shall be emphasized and which parts shall be attenuated. This lets you build equalizer apps and other effects. There are 8 types of biquad filters: highpass, lowpass, bandpass, lowshelf, highshelf, peaking, notch, and allpass.

Highpass is a filter that passes higher frequencies well, but attenuates lower frequency components of signals. **Lowpass** passes lower frequencies, but attenuates higher frequencies. They are also called "low cut" and "high cut" filters, because that explains what what happens to the signal.

Highshelf and **Lowshelf** are filters are used to control the bass and treble of the sound. They are used to emphasize or reduce signals above or below the given frequency.

You will find a Q property `BiquadFilterNode` interface, which is a double representing the **Q Factor**. Quality Factor or Q Factor control the bandwidth, the number of frequencies that are affected. The lower the Q factor, the wider the bandwidth, meaning the more frequencies will be affected. The higher the Q factor, that narrower the bandwidth.

You can find more info about filters here, but we can already build a parametric equalizer. It's an equalizer that gives full control for adjusting the frequency, bandwidth and gain.

Let's build a parametric equalizer.

See the Pen.

Let's take a look on how we can apply distortion to the sound. If you wonder what makes an electric guitar sound like one, it is the distortion effect. We use the **WaveShaperNode** interface to represent a non-linear distorter. What we need to do is to create a curve that will shape the signal, distorting and producing the characteristic sound. We don't have to spend a lot of time to create the curve, as it's already done for us. We can adjust the amount of distortion as well:

See the Pen.

Afterword

Now that you've seen how to work with the Web Audio API, I recommend playing with it on your own and making your own projects!

Here are some libraries for working with web audio:

- Pizzicato.js - Pizzicato aims to simplify the way you create and manipulate sounds via the Web Audio API
- webaudiox.js - webaudiox.js is a bunch of helpers that will make working with the WebAudio API easier
- howler.js - Javascript audio library for the modern web
- WAD - Use the HTML5 Web Audio API for dynamic sound synthesis. It's like jQuery for your ears
- Tone.js - A Web Audio framework for making interactive music in the browser

Related

Play Sound on Hover - Web Audio API Edition

The Web Audio API is a totally different beast than HTML5 audio elements. It provides lower level access to the sound

Recreating Legendary 8-bit Games Music with the Web Audio API

Greg Hovanesyan, who recently posted here an Introduction to the Web Audio API, follows up with another huge post on

Form Validation with Web Audio

I've been thinking about sound on websites for a while now. When we talk about using sound on websites, most of us

Comments

Gregor Adams

# MARCH 6, 2017

I just spent way too much time jamming with that guitar ;)

Sean

# MARCH 10, 2017

I basically wasted my entire lunch break.

Thiago Lagden

# MARCH 6, 2017

I spent hours playing guitar and xylophone!!! Too much good!!
Now, I need do my job :(

Faisal

# MARCH 6, 2017

That distortion is like web version of guitar effect, great!

GlukAlex

# MARCH 6, 2017

I'm with Gregor Adams .
Fantastic tutorial .
Simple to follow, but touches all the deep details
(yes it has a *Math inside*) .

Venerons

# MARCH 7, 2017

I have done a Web Audio synth that features most of the Web Audio nodes, in addition to some artistic visual feedback, I hope you enjoy it! :) https://venerons.github.io/Comet

Adam

# MARCH 8, 2017

Vernons, that is amazing, I had so much fun playing with your Web Audio synth! There should be an option to record!

Ariyo

# MARCH 14, 2017

Very Cool Venerons!

Sebastien Piquemal

# MARCH 9, 2017

There are a few small things to know also when you want to write cross-browser web audio code. I wrote a small blog post on the topic : http://funktion.fm/news/web-audio-in-production

Gustavo

# MARCH 9, 2017

Oh God, this article is so good. Added to markers. Great job!

Cameron

# MARCH 9, 2017

Wow, what an incredible article. Thanks so much for sharing some insight into this. Share more on how to use the Web Audio API and how to program synths/drum machines anytime :) I would definitely read every bit! Cheers.

[illegible]

[illegible]

Fred

# MARCH 11, 2017

Great article. I made a simple synth and implemented a step sequencer, but for some reason the audio slowly gets quiet if I leave it running for a little while, but ONLY if the filter's enabled and hooked up to the gain node. Does anyone know why?

leo

# MARCH 13, 2017

I am new to js and tried to experiment with your code. A few things I found that makes my test working:

not sure about where this is defined: “updateProgress(thisBuffer.urls.length); ” So I commented out.
in the audio file based second “class Sound () {…}” it seems that my firefox doesn’t like this ,so have to get rid of ().
still in the 2nd “class sound”, the “this.setup()” shall be “this.init();”

thanks

Amit Rathod

# MARCH 14, 2017

Too good article, i spent more than an hour on this , played with guitar as well as code …. Thanks For sharing :)

Alexey

# MARCH 14, 2017

Thanks for your great article, Greg!

I am curious, is it possible to “recreate” simple sounds with WebAudio ?

E.g. I want to “recreate” motor-sirene sound, from WIKI `https://upload.wikimedia.org/wikipedia/commons/f/f9/Motorsirene_-_Feuerwehralarm.ogg`

I know that I have to download audio with XHR (array buffer), decode it, play it, and on each frame (`https://developer.mozilla.org/ru/docs/DOM/window.requestAnimationFrame`) take a “snapshot” of this sound.

Lets say I did it, but how to “recreate” this sound only with WebAudio oscillator’s ?
I want to have it like sequence of oscillators that program runs.

Do you have any idea? How to analyse this “snapshots” and select among all of the frequencies “noise” only one or five frequencies that I can use with oscillator’s?

This comment thread is closed. If you have important information to share, please contact us.