Getting Started with Web Audio API

By Boris Smus

Published: October 14th, 2011
Updated: October 29th, 2013
Comments: 16

Before the HTML5 `<audio>` element, Flash or another plugin was required to break the silence of the web. While audio on the web no longer requires a plugin, the audio tag brings significant limitations for implementing sophisticated games and interactive applications.

The Web Audio API is a high-level JavaScript API for processing and synthesizing audio in web applications. The goal of this API is to include capabilities found in modern game audio engines and some of the mixing, processing, and filtering tasks that are found in modern desktop audio production applications. What follows is a gentle introduction to using this powerful API.

Getting started with the AudioContext

An AudioContext is for managing and playing all sounds. To produce a sound using the Web Audio API, create one or more sound sources and connect them to the sound destination provided by the `AudioContext` instance. This connection doesn't need to be direct, and can go through any number of intermediate AudioNodes which act as processing modules for the audio signal. This routing is described in greater detail at the Web Audio specification.

A single instance of `AudioContext` can support multiple sound inputs and complex audio graphs, so we will only need one of these for each audio application we create. Many of the interesting Web Audio API functions such as creating AudioNodes and decoding audio file data are methods of `AudioContext`.

The following snippet creates an `AudioContext`:

```
var context;
window.addEventListener('load', init, false);
function init() {
  try {
    // Fix up for prefixing
    window.AudioContext =
window.AudioContext||window.webkitAudioContext;
    context = new AudioContext();
  }
  catch(e) {
    alert('Web Audio API is not supported in this browser');
  }
}
```

For WebKit- and Blink-based browsers, you currently need to use the `webkit` prefix, i.e. `webkitAudioContext`.

Loading sounds

The Web Audio API uses an AudioBuffer for short- to medium-length sounds. The basic approach is to use XMLHttpRequest for fetching sound files.

The API supports loading audio file data in multiple formats, such as WAV, MP3, AAC, OGG and others. Browser support for different audio formats varies.

The following snippet demonstrates loading a sound sample:

```
var dogBarkingBuffer = null;
// Fix up prefixing
window.AudioContext = window.AudioContext ||
window.webkitAudioContext;
var context = new AudioContext();

function loadDogSound(url) {
  var request = new XMLHttpRequest();
  request.open('GET', url, true);
  request.responseType = 'arraybuffer';

  // Decode asynchronously
```

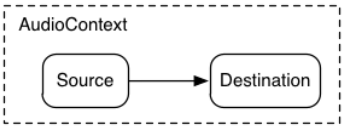
```
  request.onload = function() {
    context.decodeAudioData(request.response, function(buffer) {
      dogBarkingBuffer = buffer;
    }, onError);
  }
  request.send();
}
```

The audio file data is binary (not text), so we set the `responseType` of the request to `'arraybuffer'`. For more information about `ArrayBuffers`, see this article about XHR2.

Once the (undecoded) audio file data has been received, it can be kept around for later decoding, or it can be decoded right away using the AudioContext `decodeAudioData()` method. This method takes the `ArrayBuffer` of audio file data stored in `request.response` and decodes it asynchronously (not blocking the main JavaScript execution thread).

When `decodeAudioData()` is finished, it calls a callback function which provides the decoded PCM audio data as an `AudioBuffer`.

Playing sounds



A simple audio graph

Once one or more `AudioBuffers` are loaded, then we're ready to play sounds. Let's assume we've just loaded an `AudioBuffer` with the sound of a dog barking and that the loading has finished. Then we can play this buffer with the following code.

```
// Fix up prefixing
window.AudioContext = window.AudioContext ||
window.webkitAudioContext;
var context = new AudioContext();

function playSound(buffer) {
  var source = context.createBufferSource(); // creates a sound source
  source.buffer = buffer;                    // tell the source which
sound to play
  source.connect(context.destination);       // connect the source to
the context's destination (the speakers)
  source.start(0);                           // play the source now
                                             // note: on older
systems, may have to use deprecated noteOn(time);
}
```

This `playSound()` function could be called every time somebody presses a key or clicks something with the mouse.

The `start(time)` function makes it easy to schedule precise sound playback for games and other time-critical applications. However, to get this scheduling working properly, ensure that your sound buffers are pre-loaded. (On older systems, you may need to call `noteOn(time)` instead of `start(time)`.)

An important point to note is that **on iOS, Apple currently mutes all sound output until the first time a sound is played during a user interaction event** - for example, calling `playSound()` inside a touch event handler. You may struggle with Web Audio on iOS "not working" unless you circumvent this - in order to avoid problems like this, just play a sound (it can even be muted by connecting to a Gain Node with zero gain) inside an early UI event - e.g. "touch here to play".

Abstracting the Web Audio API

Of course, it would be better to create a more general loading system which isn't hard-coded to loading this specific sound. There are many approaches for dealing with the many short- to medium-length sounds that an audio application or game would use– here's one way using a BufferLoader class.

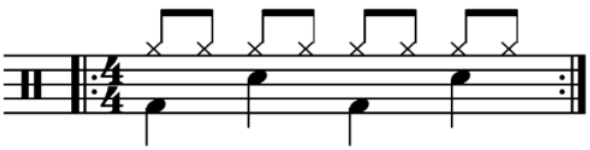
The following is an example of how you can use the `BufferLoader` class. Let's create two `AudioBuffers`; and, as soon as they are loaded, let's play them back at the same

time.

```
window.onload = init;
var context;
var bufferLoader;

function init() {
  // Fix up prefixing
  window.AudioContext = window.AudioContext ||
window.webkitAudioContext;
  context = new AudioContext();

  bufferLoader = new BufferLoader(
    context,
    [
      '../sounds/hyper-reality/br-jam-loop.wav',
      '../sounds/hyper-reality/laughter.wav',
    ],
    finishedLoading
    );

  bufferLoader.load();
}

function finishedLoading(bufferList) {
  // Create two sources and play them both together.
  var source1 = context.createBufferSource();
  var source2 = context.createBufferSource();
  source1.buffer = bufferList[0];
  source2.buffer = bufferList[1];

  source1.connect(context.destination);
  source2.connect(context.destination);
  source1.start(0);
  source2.start(0);
}
```

Dealing with time: playing sounds with rhythm

The Web Audio API lets developers precisely schedule playback. To demonstrate this, let's set up a simple rhythm track. Probably the most widely known drumkit pattern is the following:



A simple rock drum pattern

in which a hihat is played every eighth note, and kick and snare are played alternating every quarter, in 4/4 time.

Supposing we have loaded the `kick`, `snare` and `hihat` buffers, the code to do this is simple:

```
for (var bar = 0; bar < 2; bar++) {
  var time = startTime + bar * 8 * eighthNoteTime;
  // Play the bass (kick) drum on beats 1, 5
  playSound(kick, time);
  playSound(kick, time + 4 * eighthNoteTime);

  // Play the snare drum on beats 3, 7
  playSound(snare, time + 2 * eighthNoteTime);
  playSound(snare, time + 6 * eighthNoteTime);

  // Play the hi-hat every eighth note.
  for (var i = 0; i < 8; ++i) {
    playSound(hihat, time + i * eighthNoteTime);
  }
}
```

Here, we make only one repeat instead of the unlimited loop we see in the sheet music. The function `playSound` is a method that plays a buffer at a specified time, as follows:

```
function playSound(buffer, time) {
  var source = context.createBufferSource();
  source.buffer = buffer;
  source.connect(context.destination);
```

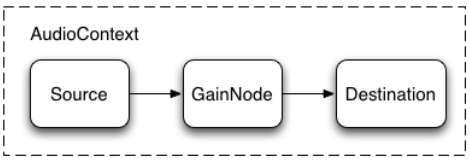
```
  source.start(time);
}
```

Play

full source code

Changing the volume of a sound

One of the most basic operations you might want to do to a sound is change its volume. Using the Web Audio API, we can route our source to its destination through an GainNode in order to manipulate the volume:



Audio graph with a gain node

This connection setup can be achieved as follows:

```
// Create a gain node.
var gainNode = context.createGain();
// Connect the source to the gain node.
source.connect(gainNode);
// Connect the gain node to the destination.
gainNode.connect(context.destination);
```

After the graph has been set up, you can programmatically change the volume by manipulating the `gainNode.gain.value` as follows:

```
// Reduce the volume.
gainNode.gain.value = 0.5;
```

The following is a demo of a volume control implemented with an `<input type="range">` element:

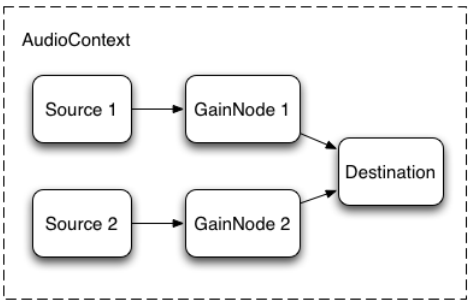
Play/Pause Volume:

full source code

Cross-fading between two sounds

Now, suppose we have a slightly more complex scenario, where we're playing multiple sounds but want to cross fade between them. This is a common case in a DJ-like application, where we have two turntables and want to be able to pan from one sound source to another.

This can be done with the following audio graph:



Audio graph with two sources connected through gain nodes

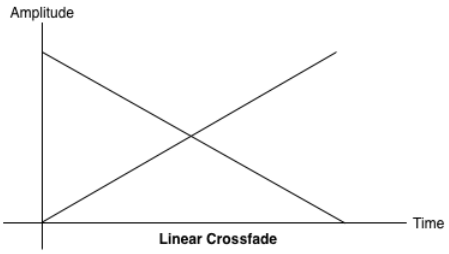
To set this up, we simply create two GainNodes, and connect each source through the nodes, using something like this function:

```
function createSource(buffer) {
  var source = context.createBufferSource();
  // Create a gain node.
```

```
  var gainNode = context.createGain();
  source.buffer = buffer;
  // Turn on looping.
  source.loop = true;
  // Connect source to gain.
  source.connect(gainNode);
  // Connect gain to destination.
  gainNode.connect(context.destination);

  return {
    source: source,
    gainNode: gainNode
  };
}
```

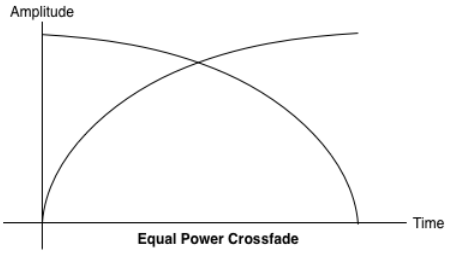
Equal power crossfading

A naive linear crossfade approach exhibits a volume dip as you pan between the samples.



A linear crossfade

To address this issue, we use an equal power curve, in which the corresponding gain curves are non-linear, and intersect at a higher amplitude. This minimizes volume dips between audio regions, resulting in a more even crossfade between regions that might be slightly different in level.



An equal power crossfade

The following demo uses an `<input type="range">` control to crossfade between the two sound sources:

Play/Pause Drums Organ

full source code

Playlist crossfading

Another common crossfader application is for a music player application. When a song changes, we want to fade the current track out, and fade the new one in, to avoid a jarring transition. To do this, schedule a crossfade into the future. While we could use `setTimeout` to do this scheduling, this is not precise. With the Web Audio API, we can use the AudioParam interface to schedule future values for parameters such as the gain value of an `GainNode`.

Thus, given a playlist, we can transition between tracks by scheduling a gain decrease on the currently playing track, and a gain increase on the next one, both slightly before the current track finishes playing:

```
function playHelper(bufferNow, bufferLater) {
  var playNow = createSource(bufferNow);
  var source = playNow.source;
  var gainNode = playNow.gainNode;
  var duration = bufferNow.duration;
  var currTime = context.currentTime;
  // Fade the playNow track in.
  gainNode.gain.linearRampToValueAtTime(0, currTime);
```

```
  gainNode.gain.linearRampToValueAtTime(1, currTime + ctx.FADE_TIME);
  // Play the playNow track.
  source.start(0);
  // At the end of the track, fade it out.
  gainNode.gain.linearRampToValueAtTime(1, currTime + duration-
ctx.FADE_TIME);
  gainNode.gain.linearRampToValueAtTime(0, currTime + duration);
  // Schedule a recursive track change with the tracks swapped.
  var recurse = arguments.callee;
  ctx.timer = setTimeout(function() {
    recurse(bufferLater, bufferNow);
  }, (duration - ctx.FADE_TIME) * 1000);
}
```

The Web Audio API provides a convenient set of `RampToValue` methods to gradually change the value of a parameter, such as `linearRampToValueAtTime` and `exponentialRampToValueAtTime`.

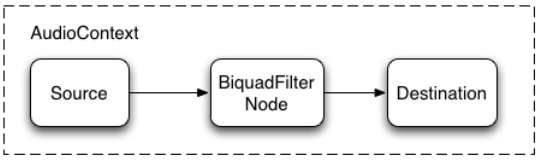
While the transition timing function can be picked from built-in linear and exponential ones (as above), you can also specify your own value curve via an array of values using the `setValueCurveAtTime` function.

The following demo shows an playlist-like auto-crossfade between two tracks using the above approach:

Play/Pause

full source code

Applying a simple filter effect to a sound



An audio graph with a `BiquadFilterNode`

The Web Audio API lets you pipe sound from one audio node into another, creating a potentially complex chain of processors to add complex effects to your soundforms.

One way to do this is to place BiquadFilterNodes between your sound source and destination. This type of audio node can do a variety of low-order filters which can be used to build graphic equalizers and even more complex effects, mostly to do with selecting which parts of the frequency spectrum of a sound to emphasize and which to subdue.

Supported types of filters include:

- Low pass filter
- High pass filter
- Band pass filter
- Low shelf filter
- High shelf filter
- Peaking filter
- Notch filter
- All pass filter

And all of the filters include parameters to specify some amount of gain, the frequency at which to apply the filter, and a quality factor. The low-pass filter keeps the lower frequency range, but discards high frequencies. The break-off point is determined by the frequency value, and the Q factor is unitless, and determines the shape of the graph. The gain only affects certain filters, such as the low-shelf and peaking filters, and not this low-pass filter.

Let's setup a simple low-pass filter to extract only the bases from a sound sample:

```
// Create the filter
var filter = context.createBiquadFilter();
// Create the audio graph.
source.connect(filter);
filter.connect(context.destination);
// Create and specify parameters for the low-pass filter.
filter.type = 'lowpass'; // Low-pass filter. See BiquadFilterNode docs
filter.frequency.value = 440; // Set cutoff to 440 HZ
```

```
// Playback the sound.
source.start(0);
```

The following demo uses a similar technique and lets you enable and disable a lowpass filter via a checkbox, as well as tweak the frequency and quality values with the slider:

Play/Pause Filter on: Frequency: Quality:

full source code

In general, frequency controls need to be tweaked to work on a logarithmic scale since human hearing itself works on the same principle (that is, A4 is 440hz, and A5 is 880hz). For more details, see the `FilterSample.changeFrequency` function in the source code link above.

Lastly, note that the sample code lets you connect and disconnect the filter, dynamically changing the AudioContext graph. We can disconnect AudioNodes from the graph by calling `node.disconnect(outputNumber)`. For example, to re-route the graph from going through a filter, to a direct connection, we can do the following:

```
// Disconnect the source and filter.
source.disconnect(0);
filter.disconnect(0);
// Connect the source directly.
source.connect(context.destination);
```

Further listening

We've covered the basics of the API, including loading and playing audio samples. We've built audio graphs with gain nodes and filters, and scheduled sounds and audio parameter tweaks to enable some common sound effects. At this point, you are ready to go and build some sweet web audio applications!

If you are seeking inspiration, many developers have already created great work using the Web Audio API. Some of my favorite include:

- AudioJedit, an in-browser sound splicing tool that uses SoundCloud permalinks.
- ToneCraft, a sound sequencer where sounds are created by stacking 3D blocks.
- Plink, a collaborative music-making game using Web Audio and Web Sockets.