Web Audio API

Benedictus Jason Reinhart

Contents

Getting Started 3

Introduction 3

Preparation 3

AudioNode Interface & AudioContext 7

AudioNode as the Generic Interface 7

Source Node 7

Audio Effects Node 8

Data Analysis and Visualization 9

Destination Node 11

Combining it All 11

References 17

Documentation & Specification 17

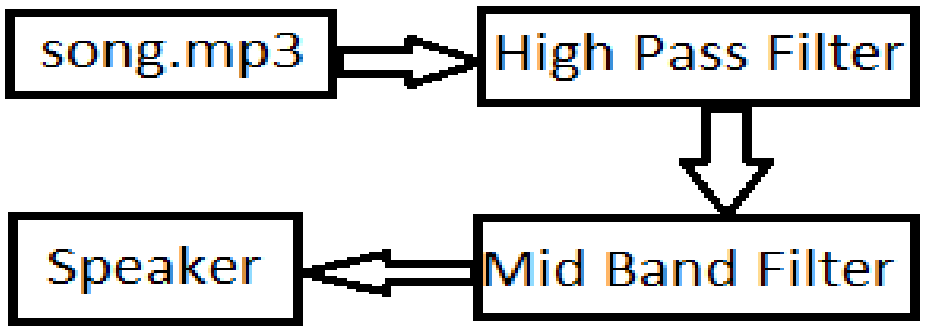
Libraries 17

Getting Started

Introduction

The Web Audio API provides a powerful system for controlling audio, allowing developers to choose audio sources, add effects to audio (such as filters), create audio visualizations, apply spatial effects (such as panning) and much more. The API handles most of the low-level signal processing into modular routing by using nodes. There are various type of node that handle different processes. These nodes has an input and an output, except the source node being the first node of the chain and the destination node being the last node of the chain. These nodes are connected as a chain of nodes, being processed one by one starting from the initial source node. The source node will output an audio, which the next node will use as an input. Then, the next node will process the input from the previous node output, and produces another different audio output. The process repeats until it reaches the final node, the destination node, usually a speaker or headphone connected to the computer and detected by the browser.

Example of nodes:

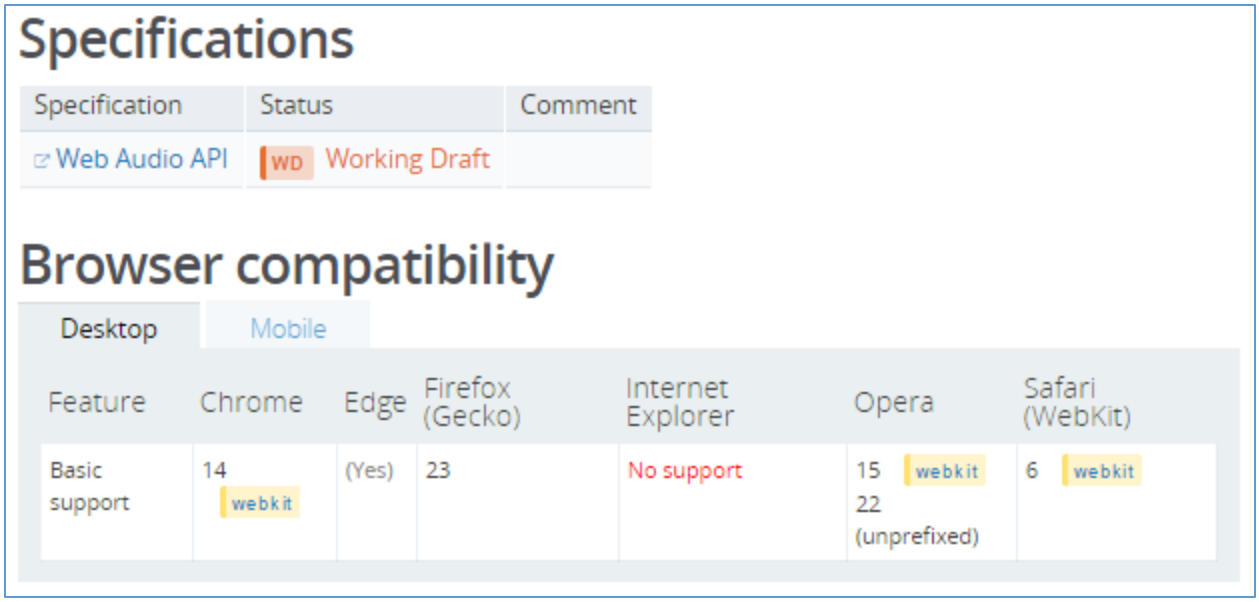


The result would be an audio that the lower frequencies filtered out and mid-range frequencies being louder. This is why the Web Audio API is modular, it separates all processes and concerns into a single node, which later be combined into a chain of processes.

However, the Web Audio API status is still a Working Draft specification, so do not expect it to be free of bugs and changes.

Preparation

To use the Web Audio API, you need a browser that supports the API. Here is a screenshot of the Web Audio API details, taken from Mozilla Developer Documentation (2016-12-05):



Specifications

Specification	Status	Comment
Web Audio API	WD Working Draft	

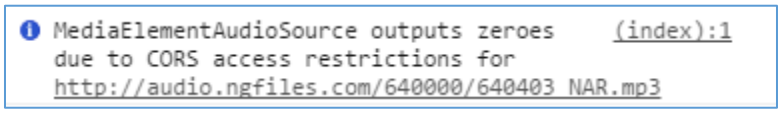
Browser compatibility

Desktop | Mobile

Feature	Chrome	Edge	Firefox (Gecko)	Internet Explorer	Opera	Safari (WebKit)
Basic support	14 webkit	(Yes)	23	No support	15 webkit 22 (unprefixed)	6 webkit

Make sure that your browser version meets the criteria above so you can test your application in your browser.

You also need a server that serves audio files that accepts cross-domain requests, because some browser might give you error when you try to request cross-domain audio files. Most servers deny CORS requests like this:



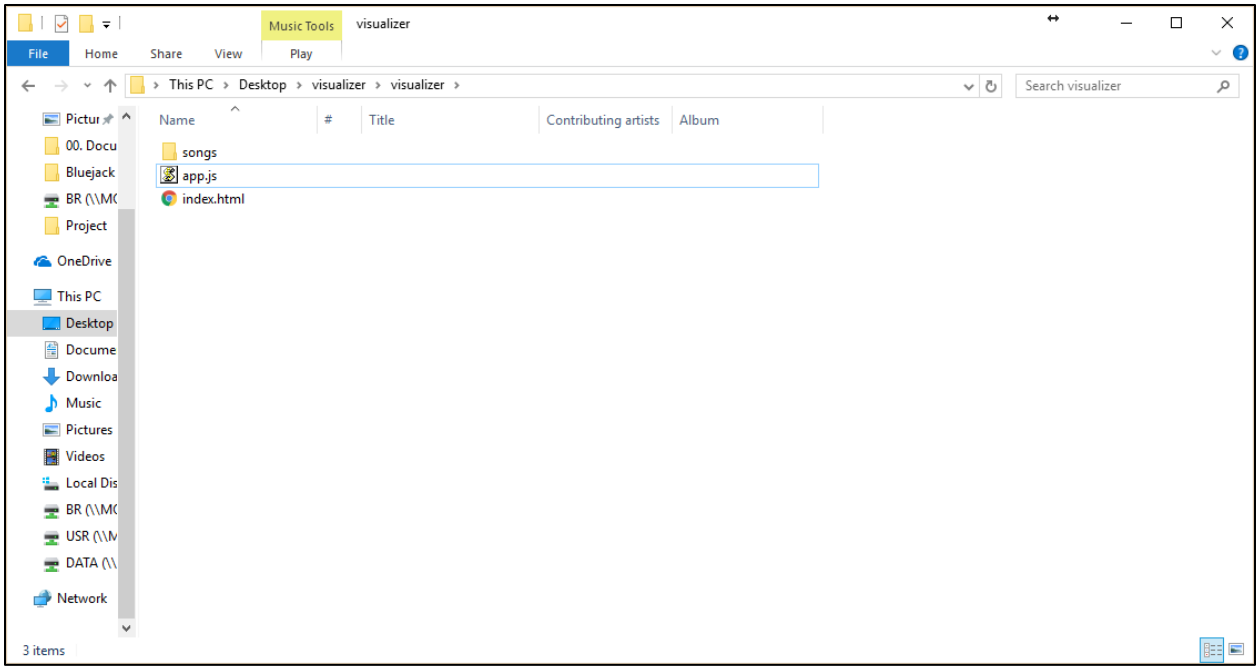
CORS access restrictions example

To solve it, make your own web server that serves your HTML, CSS, JS and mp3 files. When your HTML requests an audio file from your server, it will not deny the requests because of the CORS access restrictions, because the origin request is the same as the response.

We will use Node.js to serve your files for example. You can download Node.js from here and follow the installation instruction. After that, install the Express web application framework by typing:

```
npm install express --save
```

This will save the express package in your disk. After the installation finishes, we can start building our web application project. The project folder structure should look like this:



Node.js needs an entry point to start the application. We create `app.js` file as the mentioned entry point.

First, we need the script to start our web application. We start by creating the simple application script. Create a file named `app.js` with the following content:

```
1.  var express = require('express');
2.  var app = express();
3.
4.  app.use(function(req, res, next) {
5.      res.header("Access-Control-Allow-Origin", "*");
6.      res.header("Access-Control-Allow-Headers", "X-Requested-With");
7.      next();
8.  });
9.  app.use(express.static('songs'));
10.
11. app.get('/', function (req, res) {
12.     res.sendFile(__dirname + '/index.html');
13. });
14.
15. app.listen(3000, function () {
16.     console.log('Visualizer server listening on port 3000');
17. });
```

Here we create a web application that accepts cross-site requests. This enables our application (and other applications that requests our resources) to be able to retrieve audio files served by our server. The audio files will be stored in the *songs* folder. We will code all about Web Audio API in `index.html`. Now, whenever we requests audio file, there will not be any errors due to CORS restrictions. For example, we can now request audio files using `<audio>` tag that originates from our server.

```
<audio src="songs/song.mp3">Your browser does not support audio files.</audio>
```

Alternatively, if you have multiple sources of audio files:

```
<audio controls>
  <source src="songs/song.ogg" type="audio/ogg">
  <source src="songs/song.mp3" type="audio/mpeg">
  Your browser does not support the audio tag.
</audio>
```

AudioNode Interface & AudioContext

AudioNode as the Generic Interface

According to Mozilla's documentation, the **AudioNode** interface is a generic interface for representing an audio processing module like an audio source (e.g. an HTML **<audio>** or **<video>** element, an **OscillatorNode**, etc.), the audio destination, intermediate processing module (e.g. a filter like **BiquadFilterNode** or **ConvolverNode**), or volume control (like **GainNode**).

The concept of `AudioNode` is the same as the one explained above. It has input and output. Simply said, all kind of nodes in the API receives input, do some processing on the audio, then generate the process result as its output.

All of `AudioNodes` are contained in an `AudioContext`. All nodes in the context should at least be connected to one other node; otherwise, you should not create the node in the first place. An `AudioContext` should begin with a node that has no input, called source node, and should end on a node that has no output, called destination node.

Source Node

There are several types of audio sources:

1. `OscillatorNode` – generates a sine wave by given frequency
2. `AudioBuffer` – Short audio information in memory which later can be put into an `AudioBufferSourceNode`
3. `AudioBufferSourceNode` – audio source consisting of several `AudioBuffer`
4. `MediaElementAudioSourceNode` – audio source from HTML5 element, <audio> or <video>
5. `MediaStreamAudioSourceNode` – WebRTC MediaStream such as webcam or microphone

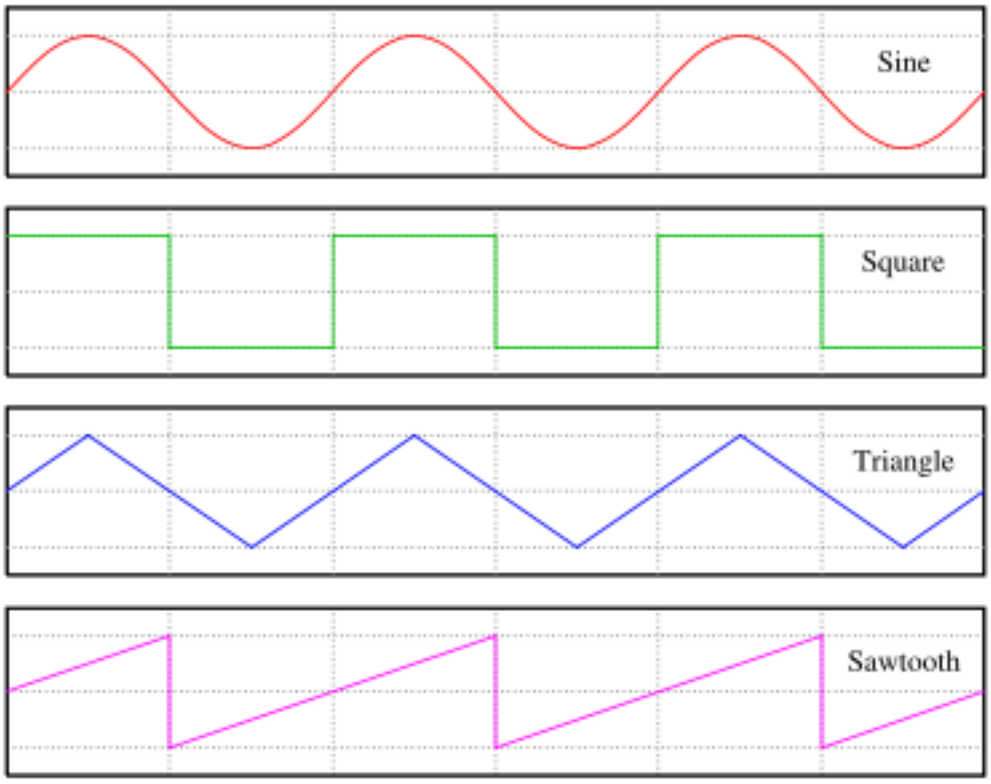
So, if you want to make your own sound, use `OscillatorNode`. If you want to process an existing sound or music, use `AudioBufferSourceNode` or `MediaElementSourceNode`. If you need to process audio from microphone or webcam, use `MediaStreamAudioSourceNode`.

Example usage:

```
1.      var audioCtx = new AudioContext();
2.
3.      // create Oscillator node
4.      var oscillator = audioCtx.createOscillator();
5.
6.      oscillator.type = 'square'; // there are also 'sine', 'square', 'sawtooth', 'triangle
        ' and 'custom', each with different kind of sound
7.      oscillator.frequency.value = 440; // value in hertz
8.      oscillator.connect(audioCtx.destination); // remember, nodes need to be linked to for
        m a chain of processes
```

```
9.      oscillator.start(); // start producing sound
10.
11.     // audio source from <audio> element
12.     var audioElement = document.querySelector('audio');
13.     var source = audioCtx.createMediaElementSource(audioElement);
14.     source.connect(audioCtx.destination);
15.     /* this does not have any output difference than just putting ordinary <audio> elemen
        t. The only difference is audio playback from the HTML element is re-
        routed to the AudioContext graph. */
```

There are various kinds of sound wave, which our ears interpret it as a different kind of sound. The OscillatorNode can produce these sounds by changing the `type` property. Here is the illustration of the sound wave shape:



Sound wave shapes (source)

Audio Effects Node

The API is surely useless without anything that modify the audio output. Modifying techniques include filters, tone control, equalizer, reverb, volume control, pan control, compression and many more. A node can perform more than one effect simultaneously, although there are effects that need more than one node to achieve.

There are so many effect nodes that you can find on the documentation. For example, if we want to boost bass and lower treble on a music while boosting:

```
1.      var audioCtx = new AudioContext();
```

```
    var audioElement = document.querySelector('audio');
    var source = audioCtx.createMediaElementSource(audio);
    var filter1 = audioCtx.createBiquadFilter();
    var filter2 = audioCtx.createBiquadFilter();
    var gain = audioCtx.createGain();

    // connect nodes
    source.connect(filter1);
    filter1.connect(filter2);
    filter2.connect(gain);
    gain.connect(audioCtx.destination);

    // adjust filter
    filter1.type = 'lowshelf'; // there are various types of filter, lowshelf means frequencies below certain value are boosted by a certain dB
    filter1.frequency.value = 2000; // boost frequencies below 2000
    filter1.gain.value = 20; // by 20dB

    filter2.type = 'highshelf'; // frequencies higher than 15000 will be attenuated by 10dB
    filter2.frequency.value = 15000 // lower frequencies higher than 15000
    filter2.gain.value = -10; // by 10dB

    // adjust volume (via gain node)
    gain.gain.value = 5
```

Data Analysis and Visualization

The `AnalyserNode` is a special node that does not modify audio in any way. The `AnalyserNode` provides the information about frequency and time-domain analysis in real-time, enabling you to create a visualizer from the frequency data, whether it is a frequency bars or a sinewave. To create an `AnalyserNode`, we can create it from `AudioContext` using `createAnalyser()` method.

```
    var audioContext = new AudioContext();
    var analyser = audioContext.createAnalyser();
```

As the `AnalyserNode` is still a node in the context which needs input and produces output (although the output is the same as the input as the node does not alter audio at all), we need to connect a source node to the `AnalyserNode`, and the `AnalyserNode` to a destination node (or another effect node, but the effect result will not be analyzed). For example on how to create a visualizer, check this tutorial out. To extract frequency information and time-domain analysis of the audio, we can use:

```
    var freqArray = new Uint8Array(analyser.frequencyBinCount);
    var tdaArray = new Uint8Array(anaylser.frequencyBinCount);
    analyser.getByteFrequencyData(freqArray);
    analyser.getByteTimeDomainData(tdaArray);
    for (var i = 0; i < analyser.frequencyBinCount; i++) {
        console.log('freqArray['+i+']: ' + freqArray[i]);
        console.log('tdaArray['+i+']: ' + tdaArray[i]);
```

```
8.      }
```

To extract information about the audio frequency data, use `getByteFrequencyData(uintArrayToBeFilled)`. The method will fill the array with the frequency data, ranging from zero to half of the sample rate, which you can obtain by calling the `sampleRate` property in an `AudioContext` object.

```
1.  var audioContext = new AudioContext();
2.  console.log(audioContext.sampleRate);
3.  // Should print 48000
```

The `frequencyBinCount` property of `AnalyserNode` is actually half of the `fftSize` property of the `AnalyserNode`. The `fftSize` determines how the FFT algorithm will transform audio wave into frequency data. Bigger `fftSize` means bigger array size, which then lead to a more detailed information. Here is an example for the explanation:

```
1.      var audioContext = new AudioContext();
2.      var sampleRate = audioContext.sampleRate; // usually 48000
3.      var analyser = audioContext.createAnalyser();
4.
5.      ... // connect nodes here
6.
7.      // determine how detail the frequency information FFT algorithm should provide
8.      analyser.fftSize = 1024; // must be a power of 2, ranging from 32 to 32768
9.
10.     var arrayLength = analyser.frequencyBinCount; // half of the fftSize, should be 512
11.     var frequencyRange = sampleRate / analyser.fftSize; // 48000 / 1024
12.     var array = new Uint8Array(arrayLength);
13.     analyser.getByteFrequencyData(array); // now array is filled with frequency data
14.
15.     // each element represents values of frequency between n * frequencyRange
16.     // and (n+1) * frequencyRange which means array[0] holds the value of 0hz - 46.875hz

17.     // and array[1] 46.876hz - 93.75hz and so on.
```

Note: `getByteFrequencyData()` does not return a value nor an array, it fills the array given in the parameter.

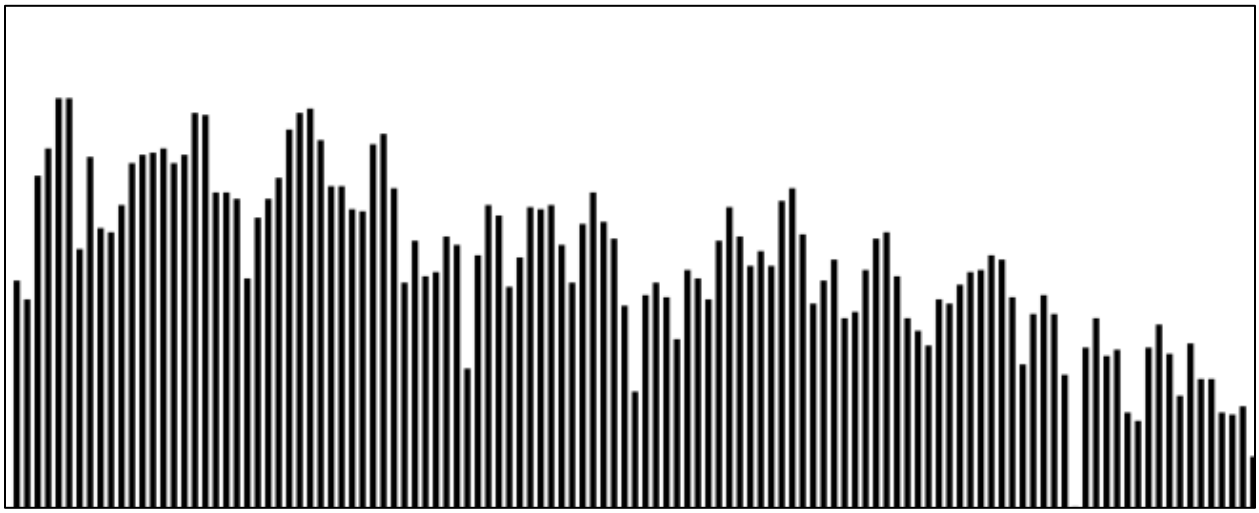
There are also properties that you can change on the `AnalyserNode` to obtain various results. The `smoothingTimeConstant` property for example, can change how the node does value-change smoothing on the frequency value. Higher constant value makes the frequency values move slower and smoother, resulting in a smoother image when you use the data for a visualizer. Try changing the value and see the results yourself:

```
1.      analyser.smoothingTimeConstant = 0.3;  // between 0 and 1
```

Versus:

```
1.      analyser.smoothingTimeConstant = 0.9;
```

You will notice on your visualizer that the higher the constant value, the smoother your visualizer will move. I suggest that you keep the `smoothingTimeConstant` in a high value (more than 0.8) so your visualizer will not move abruptly.



Audio Visualizer Example

Destination Node

All nodes you create previously in an `AudioContext` need to be connected. It begins from a source node, and it should end at an `AudioDestinationNode`. The `AudioDestinationNode` does not have any output, *as it is the output itself,* and it only has one input (but many channels, default is 6). You should not create a destination node in an `AudioContext` manually, as the API has already provided it and you can easily retrieve the node using `AudioContext` property:

```
1.      var context = new AudioContext();
2.      ...
3.      ...
4.      var destination = context.destination;
5.      source.connect(destination);
6.      ...
```

The destination node in an `AudioContext` is usually a speaker connected to the computer that runs the browser. Destination node can also be a recorder if `OfflineAudioContext` is used.

Combining it All

Now, let's try to make a real use of the Web Audio API. We will get an audio source from `<audio>` element and make a sound using oscillator, modify the audio by using filters and reverbs, and then combine it and send it to the output node. We will also make a visualizer for the song, a frequency bar visualizer.

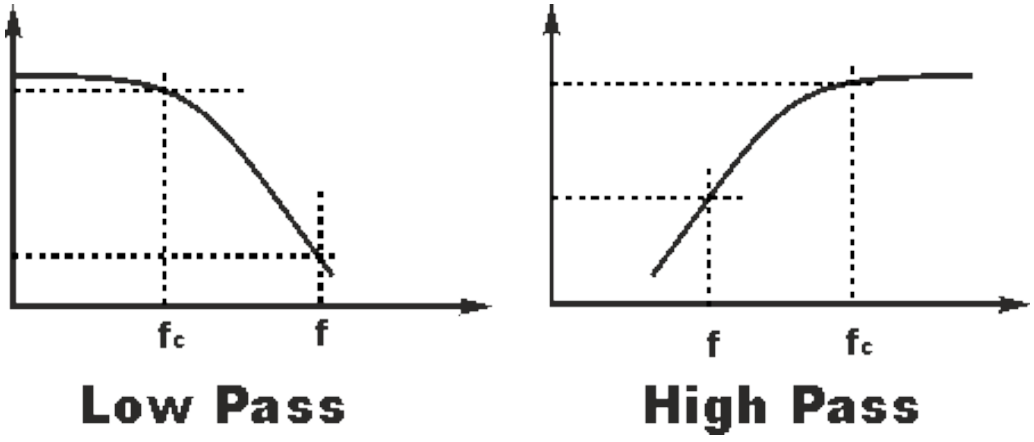
```html
<!DOCTYPE html>
<html>
<head></head>
<body>
    <audio autoplay controls>
        <source src="nar.mp3" type="audio/mpeg"/>
    </audio>
    <canvas></canvas>
    <script>
    // initialize context and nodes
    var audioContext = new AudioContext();
    var oscillatorNode = audioContext.createOscillator();
    var audioSource = document.querySelector('audio');
    var sourceNode = audioContext.createMediaElementSource(audioSource);
    var filterNode = audioContext.createBiquadFilter();
    var gainNode = audioContext.createGain();
    var analyserNode = audioContext.createAnalyser();
    var destinationNode = audioContext.destination;

    // adjust nodes
    oscillatorNode.frequency.value = 440;
    oscillatorNode.type = 'sine';
    oscillatorNode.start();

    filterNode.frequency.value = 8000;
    filterNode.gain.value = -100;
    filterNode.type = 'highpass';

    gainNode.gain.value = 0.17;

    // connect nodes
    sourceNode.connect(filterNode);
    filterNode.connect(analyserNode);

    oscillatorNode.connect(gainNode);
    gainNode.connect(analyserNode);

    analyserNode.connect(destinationNode);
    analyserNode.fftSize = 1024;

    var canvas = document.querySelector('canvas');
    canvas.width = 1200;
    canvas.height = 500;
    var canvasContext = canvas.getContext('2d');
    var bytes = new Uint8Array(analyserNode.frequencyBinCount);
    function animate() {
        requestAnimationFrame(animate);
        canvasContext.clearRect(0, 0, canvas.width, canvas.height);

        analyserNode.getByteFrequencyData(bytes);
        var x = 0;
        for (var i = 0; i < analyserNode.frequencyBinCount; i++) {
            var y = canvas.height - bytes[i];
            canvasContext.fillRect(x, y, 3, bytes[i]);
            x+=5;
        }
    }
    animate();

    </script>
</body>
```

```
62. </html>
```

Here we create an audio visualizer using HTML5 canvas and the `AnalyserNode`, an oscillator to demonstrate how the `OscillatorNode` makes sound, `MediaElementSourceNode` to play a song from the `<audio>` element, `GainNode` to control the oscillator volume, and `BiquadFilterNode` to filter out the low frequencies of the song. The `sourceNode` represents the song and we connect the song to the `filterNode` to remove low frequency sound, because our oscillator is set to play at 440 Hz, which means we need to make "room" for the oscillator sound to be heard. We connect the oscillator to the gain node to control its volume, as it is uncontrollable without the gain node. To make the oscillator produce sound, we need to start playing it, using `oscillatorNode.start()`, which makes the oscillator produce sound with a given frequency and wave form.

```
1.  var oscillatorNode = audioContext.createOscillator();
2.  var gainNode = audioContext.createGain();
3.  oscillatorNode.frequency.value = 440;
4.  oscillatorNode.type = 'sine';
5.  oscillatorNode.start();
6.
7.  gainNode.gain.value = 0.17;
8.
9.  oscillatorNode.connect(gainNode);
10. gainNode.connect(analyserNode);
11. analyserNode.connect(destinationNode);
```

The filter we use to reduce the song low frequency is the `BiquadFilterNode`. We set the filter node to be a "High Pass" filter, a kind of filter that reduces all frequency lower than a given frequency value by a given decibel value. There is also the "Low Pass" filter which reduces all frequency higher than a given frequency value by a given decibel value.



Low Pass vs. High Pass (source)

Here, we use it to reduce all frequency volume below 8000 Hz by 100dB, which should be more than enough. The filter should not let you hear anything below 8000 Hz.

```
var filterNode = audioContext.createBiquadFilter();

filterNode.frequency.value = 8000;
filterNode.gain.value = -100;
filterNode.type = 'highpass';
```

Before we connect anything to the destination node, we want everything to be analyzed and visualized; hence, we need to connect it to the `AnalyserNode` before going to the destination node.

```
sourceNode.connect(filterNode);
filterNode.connect(analyserNode);

oscillatorNode.connect(gainNode);
gainNode.connect(analyserNode);

analyserNode.connect(destinationNode);
```

This ensures that the `AnalyserNode` analyzes every sound we produce, so the visualizer we make can actually visualize whatever sound is playing.

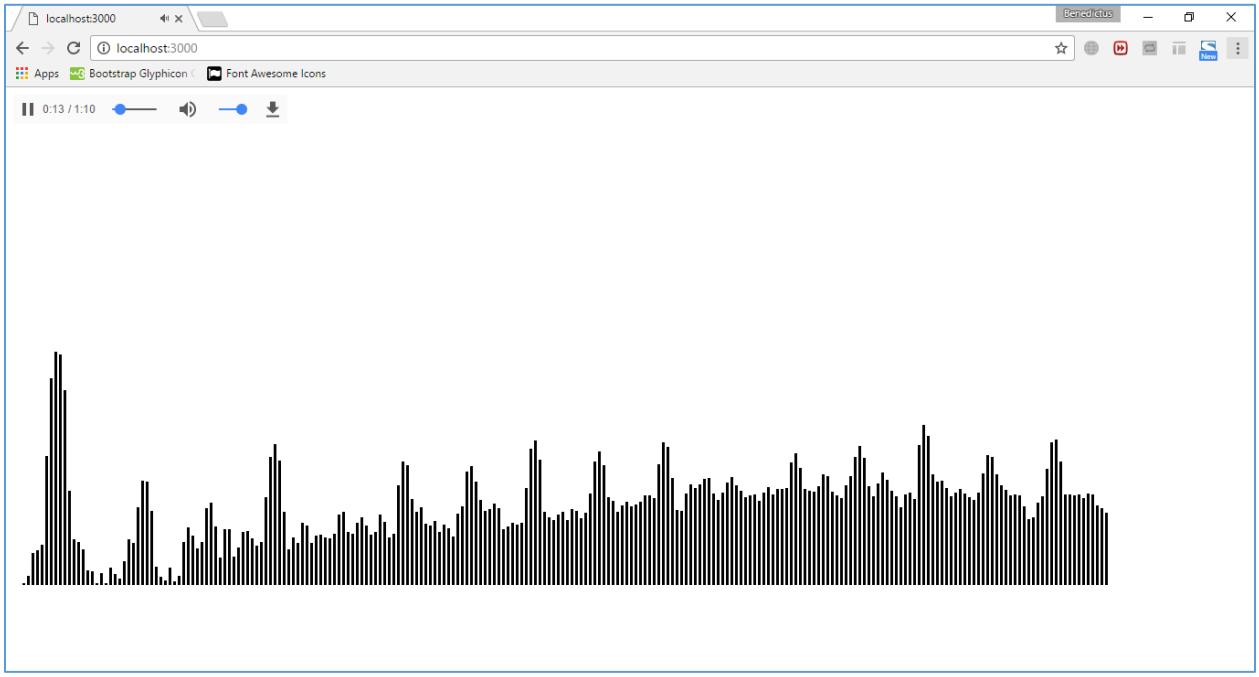
Lastly, we use the `AnalyserNode` to retrieve information about the current frequency, using `analyserNode.getByteFrequencyData(bytes)`. Remember that the `getByteFrequencyData(bytes)` method does not return anything, it fills the array of unsigned integers given in the parameter. We use the `bytes` array to create a visualizer based on its values. Now, having `fftSize` of 1024, we have the `frequencyBinCount` or `bytes[]` array length of 512 (half of 1024). Having a sample rate of (most likely) 48.000Hz, it means our maximum frequency is 24.000Hz (half of 48.000Hz). From here, we know that:

- `bytes[]` array length: 512
- Maximum Frequency: 24.000Hz
- 24000 / 512 = 46.875, each `bytes[]` array element represents a range of 47Hz (rounded to be more readable)
- `bytes[0]` represents 0-47Hz
- `bytes[1]` represents 48-95Hz
- `bytes[2]` represents 96-143Hz
- …and so on until `bytes[511]`

With the frequency information, we can draw a bar to represent the value of each `bytes[]` element. We loop from the first element of `bytes[]` until the last to draw a bar with a height of `bytes[i]`.

```
analyserNode.getByteFrequencyData(bytes);
var x = 0;
for (var i = 0; i < analyserNode.frequencyBinCount; i++) {
    var y = canvas.height - bytes[i];
    canvasContext.fillRect(x, y, 3, bytes[i]);
    x+=5;
}
```

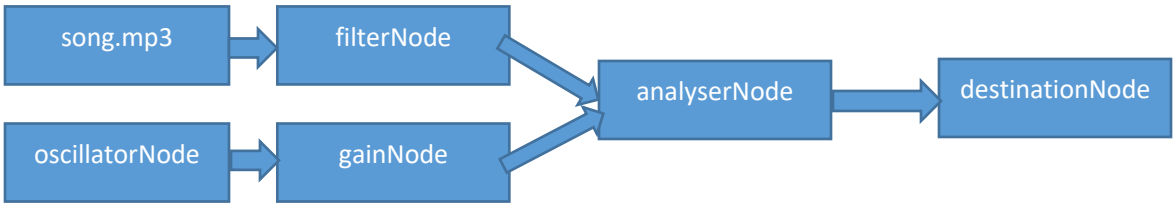
The result should look like this:



The left side bars' height of the visualizer are not really tall, because we suppressed the frequency by using the `BiquadFilterNode`'s high pass filter, so lower frequencies volumes are now lowered. However, few bars outstandingly spike on the left even though we have a high pass filter doing its job. Those bars represents the `OscillatorNode`'s sound wave, which we do not filter the sound using our `BiquadFilterNode`. If you look again on the nodes connection, we do not connect our `OscillatorNode` to the `BiquadFilterNode`, hence why it can still sound so loud even though our song is already filtered.

```
sourceNode.connect(filterNode);
filterNode.connect(analyserNode);

oscillatorNode.connect(gainNode);
gainNode.connect(analyserNode);

```

```
7.  analyserNode.connect(destinationNode);
```



The rest of the bars represent the frequencies of the song. Results may vary between songs. For example, drum and bass songs may have lower treble (mid to high frequencies) and loud bass, resulting in tall low frequency bars and short high frequency bars.

References

Documentation & Specification

The Mozilla Developer Network site has an almost complete documentation on the most implementation of the Web Audio API. For further references, you can read more about what this module does not cover (advanced materials), as this module only cover the basics and try to be newbie-friendly. The documentation should follow the specification.

https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API

As of the publishing of this module (2016-16-12), the Web Audio API is still in its Working Draft, meaning it is not final yet, but the developer is committed to improve the API. You might see some differences in this module and the documentation.

More about Working Draft document: http://whatis.techtarget.com/definition/working-draft-draft-document

Libraries

To simplify things, some developers have made libraries to help other developers to code applications that rely on the Web Audio API. There are libraries that help developers to make sound using the API, and there are some that help developers give effect on audio.

- tones: small library to help developers create sounds by a given specific frequency, a named note in the default 4th octave, and a named note in a specific octave. Great for building simple application that generates simple sound, it also provides tuning and adjustment for the sound it generates.
- Howler: bigger library than tones, Howler gives more features and powerful yet simple API that falls back into HTML5 audio when the browser does not support Web Audio API.
- WAD: library that focuses on being a Digital Audio Workstation (DAW), applications that help user to make music.