# Article 1 - Basic concepts behind Web Audio API

[**IN THIS ARTICLE**](https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API/Basic_concepts_behind_Web_Audio_API#toc)

1. [Audio graphs](https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API/Basic_concepts_behind_Web_Audio_API#Audio_graphs)
2. [Audio data: what's in a sample](https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API/Basic_concepts_behind_Web_Audio_API#Audio_data_what's_in_a_sample)
3. [Audio buffers: frames, samples and channels](https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API/Basic_concepts_behind_Web_Audio_API#Audio_buffers_frames_samples_and_channels)
   1. [Planar versus interleaved buffers](https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API/Basic_concepts_behind_Web_Audio_API#Planar_versus_interleaved_buffers)
4. [Audio channels](https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API/Basic_concepts_behind_Web_Audio_API#Audio_channels)
   1. [Up-mixing and down-mixing](https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API/Basic_concepts_behind_Web_Audio_API#Up-mixing_and_down-mixing)
5. [Visualizations](https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API/Basic_concepts_behind_Web_Audio_API#Visualizations)
6. [Spatialisations](https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API/Basic_concepts_behind_Web_Audio_API#Spatialisations)
7. [Fan-in and Fan-out](https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API/Basic_concepts_behind_Web_Audio_API#Fan-in_and_Fan-out)

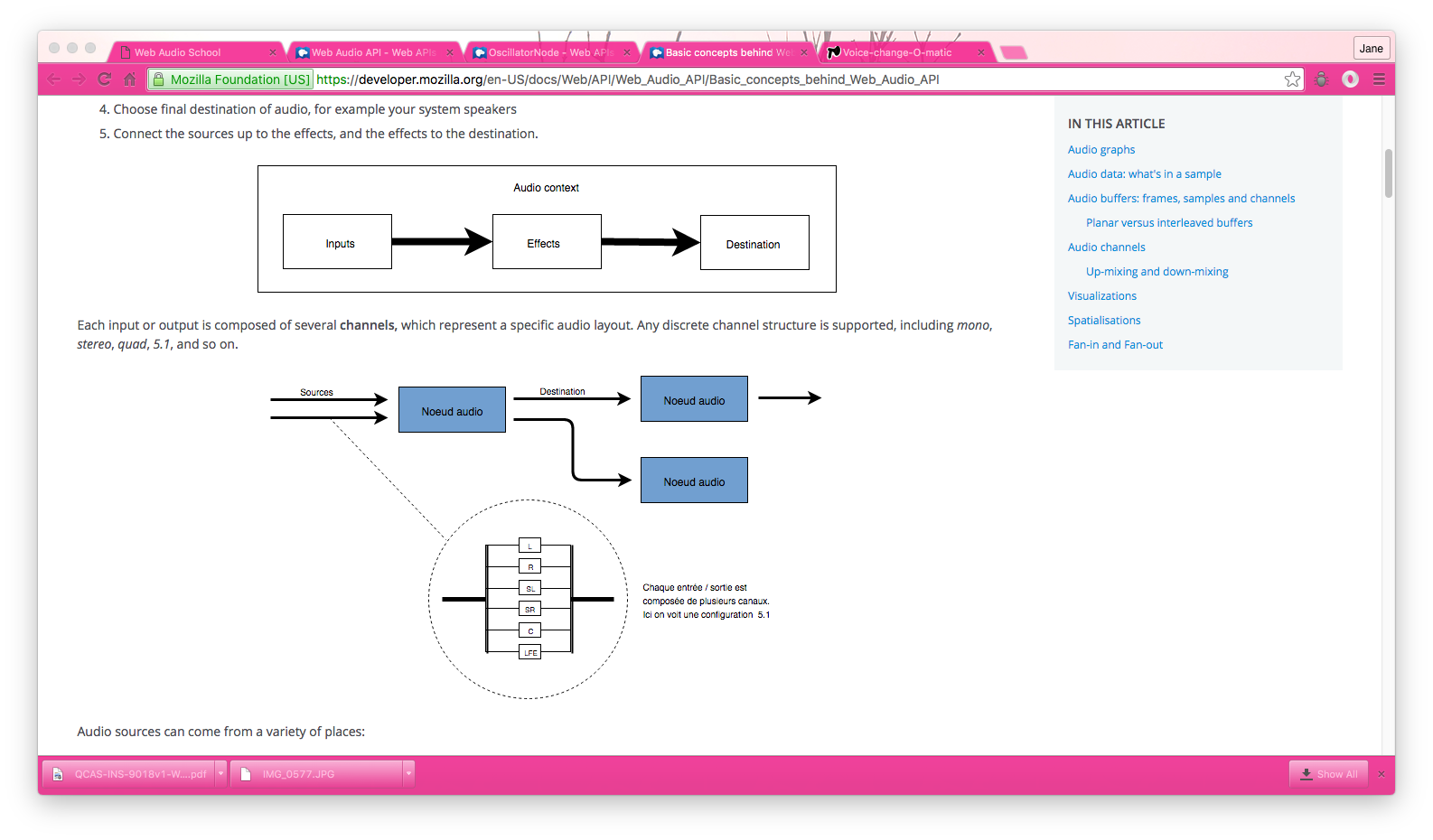
**This article explains some of the audio theory behind how the features of the Web Audio API work. It won't turn you into a master sound engineer, but it will give you enough background to understand why the Web Audio API works like it does, and allow you to make better informed decisions while developing with it.**

# Audio graphs

The Web Audio API involves handling audio operations inside an **audio context**, and has been designed to allow **modular routing**. Basic audio operations are performed with **audio nodes**, which are linked together to form an **audio routing graph**. Several sources — with different types of channel layout — are supported even within a single context. This modular design provides the flexibility to create complex audio functions with dynamic effects.

Audio nodes are linked via their inputs and outputs, forming a chain that starts with one or more sources, goes through one or more nodes, then ends up at a destination (although you don't have to provide a destination if you, say, just want to visualize some audio data.) A simple, typical workflow for web audio would look something like this:

1. Create audio context
2. Inside the context, create sources — such as <audio>, oscillator, stream
3. Create effects nodes, such as reverb, biquad filter, panner, compressor
4. Choose final destination of audio, for example your system speakers
5. Connect the sources up to the effects, and the effects to the destination.



Each input or output is composed of several **channels,**which represent a specific audio layout. Any discrete channel structure is supported, including *mono*,*stereo*, *quad*, *5.1*, and so on.

Audio sources can come from a variety of places:

* Generated directly by JavaScript by an audio node (such as an oscillator.)
* Created from raw PCM data (the audio context has methods to decode supported audio formats.)
* Taken from HTML media elements (such as [<video>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/video) or [<audio>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/audio).)
* Taken directly from a [WebRTC](https://developer.mozilla.org/en-US/docs/WebRTC) [MediaStream](https://developer.mozilla.org/en-US/docs/Web/API/MediaStream) (such as a webcam or microphone.)

# Audio data: what's in a sample

When an audio signal is processed, **sampling** means the conversion of a [continuous signal](http://en.wikipedia.org/wiki/Continuous_signal) to a [discrete signal](http://en.wikipedia.org/wiki/Discrete_signal); put another way, a continuous sound wave such as a band playing live is converted to a sequence of samples (a discrete-time signal) that allow a computer to handle the audio in distinct blocks.

A lot more information can be found at the Wikipedia page on [Sampling (signal processing)](http://en.wikipedia.org/wiki/Sampling_%28signal_processing%29).

# Audio buffers: frames, samples and channels

An [AudioBuffer](https://developer.mozilla.org/en-US/docs/Web/API/AudioBuffer) takes as its parameters:

* a number of channels (1 for mono, 2 for stereo, etc),
* a length, meaning the number of sample frames inside the buffer;
* a sample rate, which is the number of sample frames played per second.

A sample is a single float32 value that represents the value of the audio stream at each specific point in time, in a specific channel (left or right, if in the case of stereo). A frame, or sample frame is the set of all values for all channels that will play at a specific point in time: all the samples of all the channels that play at the same time (two for a stereo sound, six for 5.1, etc.)

The sample rate is the number of those samples (or frames, since all samples of a frame play at the same time) that will play in one second, measured in Hz. The higher the sample rate, the better the sound quality.

Let's look at a Mono and a Stereo audio buffer, each one second long, and playing at 44100Hz:

* The Mono buffer will have 44100 samples, and 44100 frames. The length property will be 44100.
* The Stereo buffer will have 82100 samples, but still 44100 frames. The length property will still be 44100 since it's equal to the number of frames.

When a buffer plays, you will hear the left most sample frame, and then the one right next to it, etc, etc. In the case of stereo, you will hear both channels at the same time. Sample frames are very useful, because they are independent of the number of channels, and represent time, in a useful way for doing precise audio manipulation.

**Note**: To get a time in seconds from a frame count, simply divide the number of frames by the sample rate. To get a number of frames from a number of samples, simply divide by the channel count.

Here's a couple of simple trivial examples:

var context = new AudioContext();

var buffer = context.createBuffer(2, 22050, 44100);

**Note**: In [digital audio](https://en.wikipedia.org/wiki/Digital_audio), **44,100**[**Hz**](https://en.wikipedia.org/wiki/Hertz) (alternately represented as **44.1 kHz**) is a common [sampling frequency](https://en.wikipedia.org/wiki/Sampling_frequency). Why 44.1kHz? Firstly, because the [hearing range](https://en.wikipedia.org/wiki/Hearing_range) of human ears is roughly 20 Hz to 20,000 Hz, and via the [Nyquist–Shannon sampling theorem](https://en.wikipedia.org/wiki/Nyquist%E2%80%93Shannon_sampling_theorem) the sampling frequency must be greater than twice the maximum frequency one wishes to reproduce, the sampling rate therefore had to be greater than 40 kHz. In addition to this, signals must be [low-pass filtered](https://en.wikipedia.org/wiki/Low-pass_filter) before sampling, otherwise[aliasing](https://en.wikipedia.org/wiki/Aliasing) occurs, and, while an ideal low-pass filter would perfectly pass frequencies below 20 kHz (without attenuating them) and perfectly cut off frequencies above 20 kHz, in practice a [transition band](https://en.wikipedia.org/wiki/Transition_band)is necessary, where frequencies are partly attenuated. The wider this transition band is, the easier and more economical it is to make an [anti-aliasing filter](https://en.wikipedia.org/wiki/Anti-aliasing_filter). The 44.1 kHz sampling frequency allows for a 2.05 kHz transition band.

If you use this call, you will get a stereo buffer (two channels), that, when played back on an AudioContext running at 44100Hz (very common, most normal sound cards run at this rate), will last for 0.5 seconds: 22050 frames / 44100Hz = 0.5 seconds.

var context = new AudioContext();

var buffer = context.createBuffer(1, 22050, 22050);

If you use this call, you will get a mono buffer (one channel), that, when played back on an AudioContext running at 44100Hz, will be automatically \*resampled\* to 44100Hz (and therefore yield 44100 frames), and last for 1.0 second: 44100 frames / 44100Hz = 1 second.

**Note**: audio resampling is very similar to image resizing: say you've got a 16 x 16 image, but you want it to fill a 32x32 area: you resize (resample) it. the result has less quality (it can be blurry or edgy, depending on the resizing algorithm), but it works, and the resized image takes up less space. Resampled audio is exactly the same — you save space, but in practice you will be unable to properly reproduce high frequency content (treble sound).

# Planar versus interleaved buffers

The Web Audio API uses a planar buffer format: the left and right channels are stored like this:

LLLLLLLLLLLLLLLLRRRRRRRRRRRRRRRR (for a buffer of 16 frames)

This is very common in audio processing: it makes it easy to process each channel independently.

The alternative is to use an interleaved buffer format:

LRLRLRLRLRLRLRLRLRLRLRLRLRLRLRLR (for a buffer of 16 frames)

This format is very common for storing and playing back audio without much processing, for example a decoded MP3 stream.  
  
The Web Audio API exposes \*only\* planar buffers, because it's made for processing. It works with planar, but converts the audio to interleaved when it is sent to the  
sound card, for playback. Conversely, when an MP3 is decoded, it starts off in interleaved format, but is converted to planar for processing.

# Audio channels

Different audio buffers contain different numbers of channels, from the more basic mono (only one channel) and stereo (left and right channels) to more complex sets like quad and 5.1, which have different sound samples contained in each channel, leading to a richer sound experience. The channels are usually represented by standard abbreviations detaild in the table below:

|  |  |
| --- | --- |
| *Mono* | 0: M: mono |
| *Stereo* | 0: L: left 1: R: right |
| *Quad* | 0: L: left 1: R: right 2: SL: surround left 3: SR: surround right |
| *5.1* | 0: L: left 1: R: right 2: C: center 3: LFE: subwoofer 4: SL: surround left 5: SR: surround right |

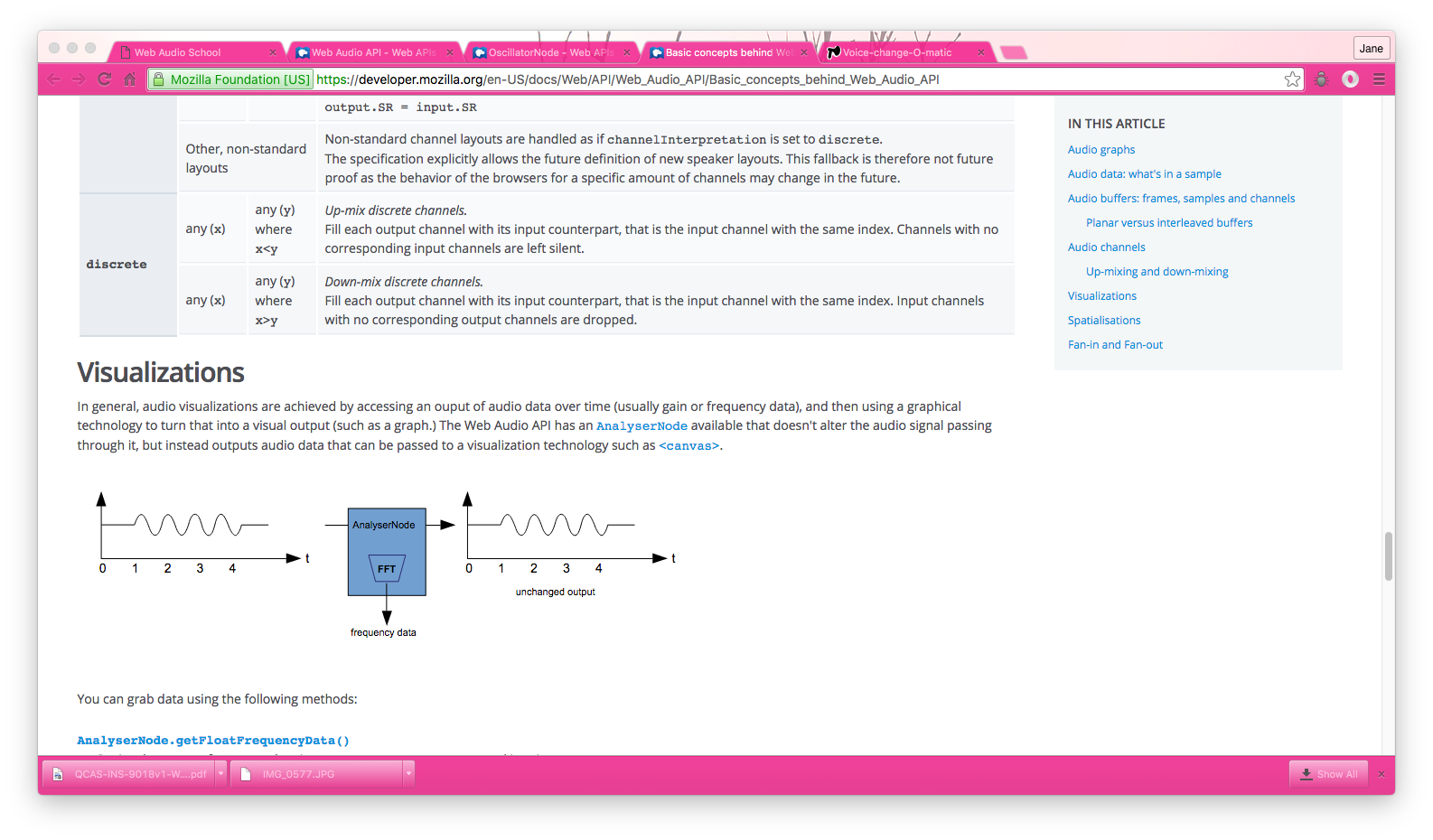
# Up-mixing and down-mixing

When the amount of channels doesn't match between an input and an output, up- or down-mixing happens according the following rules. This can be somewhat controlled by setting the [AudioNode.channelInterpretation](https://developer.mozilla.org/en-US/docs/Web/API/AudioNode/channelInterpretation) property to speakers or discrete.

| **Interpretation** | **Input channels** | **Output channels** | **Mixing rules** |
| --- | --- | --- | --- |
| **speakers** | 1 *(Mono)* | 2 *(Stereo)* | *Up-mix from mono to stereo*. The M input channel is used for both output channels (L and R). output.L = input.M output.R = input.M |
| 1 *(Mono)* | 4 *(Quad)* | *Up-mix from mono to quad.* The M input channel is used for non-surround output channels (L and R). Surround output channels (SL and SR) are silent. output.L = input.M output.R = input.M output.SL = 0 output.SR = 0 |
| 1 *(Mono)* | 6 *(5.1)* | *Up-mix from mono to 5.1.* The M input channel is used for the center output channel (C). All the others (L, R, LFE, SL, and SR) are silent. output.L = 0 output.R = 0 output.C = input.M output.LFE = 0 output.SL = 0 output.SR = 0 |
| 2 *(Stereo)* | 1 *(Mono)* | *Down-mix from stereo to mono*. Both input channels (L and R) are equally combined to produce the unique output channel (M). output.M = 0.5 \* (input.L + input.R) |
| 2 *(Stereo)* | 4 *(Quad)* | *Up-mix from stereo to quad.* The L and R input channels are used for their non-surround respective output channels (L and R). Surround output channels (SL and SR) are silent. output.L = input.L output.R = input.R output.SL = 0 output.SR = 0 |
| 2 *(Stereo)* | 6 *(5.1)* | *Up-mix from stereo to 5.1.* The L and R input channels are used for their non-surround respective output channels (L and R). Surround output channels (SL and SR), as well as the center (C) and subwoofer (LFE) channels, are left silent. output.L = input.L output.R = input.R output.C = 0 output.LFE = 0 output.SL = 0 output.SR = 0 |
| 4 *(Quad)* | 1 *(Mono)* | *Down-mix from quad to mono*. All four input channels (L, R, SL, and SR) are equally combined to produce the unique output channel (M). output.M = 0.25 \* (input.L + input.R + input.SL + input.SR) |
| 4 *(Quad)* | 2 *(Stereo)* | *Down-mix from quad to stereo*. Both left input channels (L and SL) are equally combined to produce the unique left output channel (L). And similarly, both right input channels (R and SR) are equally combined to produce the unique right output channel (R). output.L = 0.5 \* (input.L + input.SL) output.R = 0.5 \* (input.R + input.SR) |
| 4 *(Quad)* | 6 *(5.1)* | *Up-mix from quad to 5.1.* The L, R, SL, and SR input channels are used for their respective output channels (L and R). Center (C) and subwoofer (LFE) channels are left silent. output.L = input.L output.R = input.R output.C = 0 output.LFE = 0 output.SL = input.SL output.SR = input.SR |
| 6 *(5.1)* | 1 *(Mono)* | *Down-mix from 5.1 to mono.* The left (L and SL), right (R and SR) and central channels are all mixed together. The surround channels are slightly attenuated and the regular lateral channels are power-compensated to make them count as a single channel by multiplying by √2/2. The subwoofer (LFE) channel is lost. output.M = 0.7071 \* (input.L + input.R) + input.C + 0.5 \* (input.SL + input.SR) |
| 6 *(5.1)* | 2 *(Stereo)* | *Down-mix from 5.1 to stereo.* The central channel (C) is summed with each lateral surround channel (SL or SR) and mixed to each lateral channel. As it is mixed down to two channels, it is mixed at a lower power: in each case it is multiplied by √2/2. The subwoofer (LFE) channel is lost. output.L = input.L + 0.7071 \* (input.C + input.SL) output.R = input.R + 0.7071 \* (input.C + input.SR) |
| 6 *(5.1)* | 4 *(Quad)* | *Down-mix from 5.1 to quad.* The central (C) is mixed with the lateral non-surround channels (L and R). As it is mixed down to two channels, it is mixed at a lower power: in each case it is multiplied by √2/2. The surround channels are passed unchanged. The subwoofer (LFE) channel is lost. output.L = input.L + 0.7071 \* input.C output.R = input.R + 0.7071 \* input.C output.SL = input.SL output.SR = input.SR |
| Other, non-standard layouts | | Non-standard channel layouts are handled as if channelInterpretation is set to discrete. The specification explicitly allows the future definition of new speaker layouts. This fallback is therefore not future proof as the behavior of the browsers for a specific amount of channels may change in the future. |
| **discrete** | any (x) | any (y) wherex<y | *Up-mix discrete channels.* Fill each output channel with its input counterpart, that is the input channel with the same index. Channels with no corresponding input channels are left silent. |
| any (x) | any (y) wherex>y | *Down-mix discrete channels.* Fill each output channel with its input counterpart, that is the input channel with the same index. Input channels with no corresponding output channels are dropped. |

## Visualizations

In general, audio visualizations are achieved by accessing an ouput of audio data over time (usually gain or frequency data), and then using a graphical technology to turn that into a visual output (such as a graph.)



The Web Audio API has an [AnalyserNode](https://developer.mozilla.org/en-US/docs/Web/API/AnalyserNode) available that doesn't alter the audio signal passing through it, but instead outputs audio data that can be passed to a visualization technology such as [<canvas>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/canvas).

You can grab data using the following methods:

[**AnalyserNode.getFloatFrequencyData()**](https://developer.mozilla.org/en-US/docs/Web/API/AnalyserNode/getFloatFrequencyData)

Copies the current frequency data into a [Float32Array](https://developer.mozilla.org/en-US/docs/Web/API/Float32Array) array passed into it.

[**AnalyserNode.getByteFrequencyData()**](https://developer.mozilla.org/en-US/docs/Web/API/AnalyserNode/getByteFrequencyData)

Copies the current frequency data into a [Uint8Array](https://developer.mozilla.org/en-US/docs/Web/API/Uint8Array) (unsigned byte array) passed into it.

[**AnalyserNode.getFloatTimeDomainData()**](https://developer.mozilla.org/en-US/docs/Web/API/AnalyserNode/getFloatTimeDomainData)

Copies the current waveform, or time-domain, data into a [Float32Array](https://developer.mozilla.org/en-US/docs/Web/API/Float32Array) array passed into it.

[**AnalyserNode.getByteTimeDomainData()**](https://developer.mozilla.org/en-US/docs/Web/API/AnalyserNode/getByteTimeDomainData)

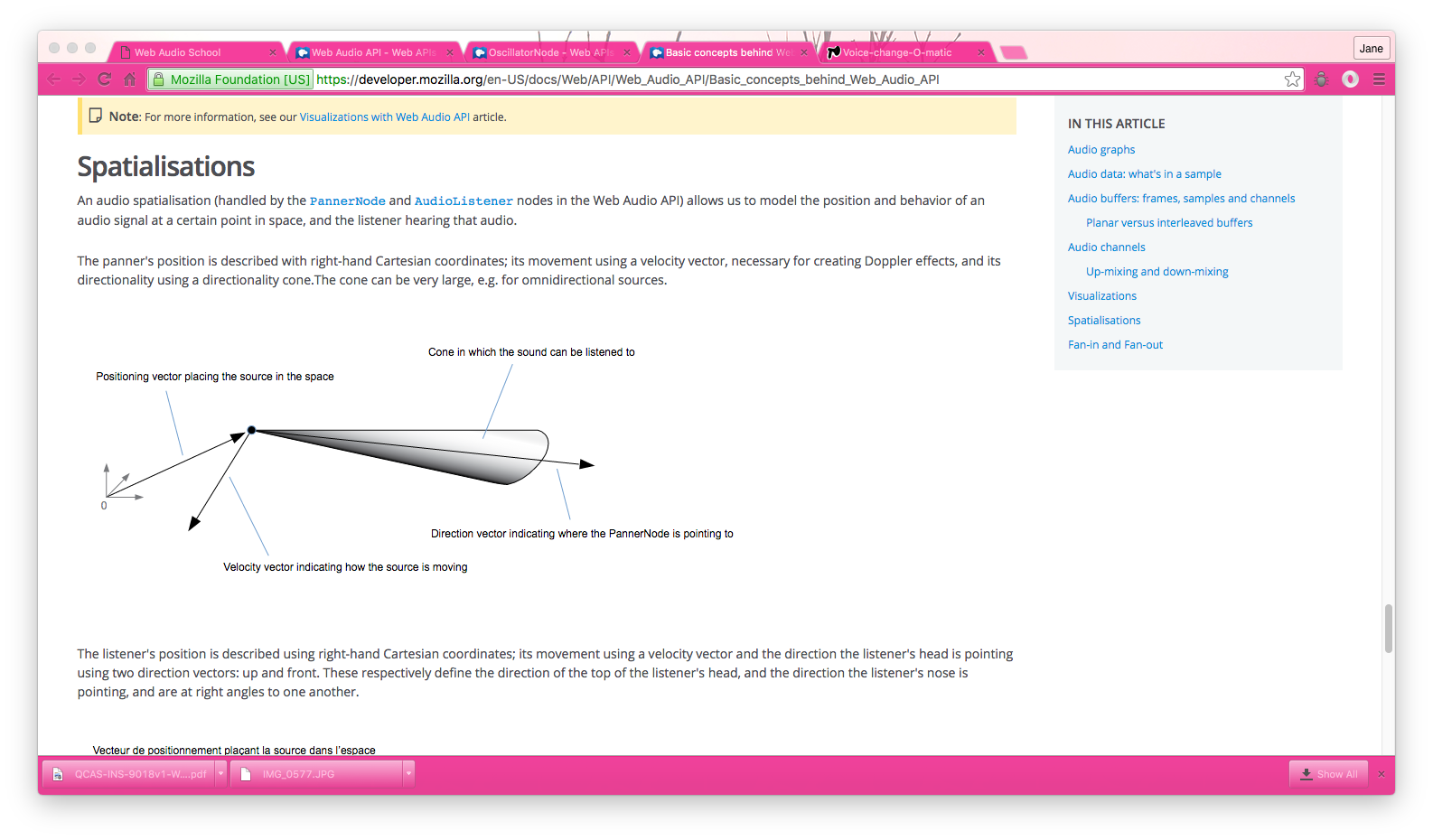
Copies the current waveform, or time-domain, data into a [Uint8Array](https://developer.mozilla.org/en-US/docs/Web/API/Uint8Array) (unsigned byte array) passed into it.

**Note**: For more information, see our [Visualizations with Web Audio API](https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API/Visualizations_with_Web_Audio_API) article.

# Spatialisations

An audio spatialisation (handled by the [PannerNode](https://developer.mozilla.org/en-US/docs/Web/API/PannerNode) and [AudioListener](https://developer.mozilla.org/en-US/docs/Web/API/AudioListener) nodes in the Web Audio API) allows us to model the position and behavior of an audio signal at a certain point in space, and the listener hearing that audio.

The panner's position is described with right-hand Cartesian coordinates; its movement using a velocity vector, necessary for creating Doppler effects, and its directionality using a directionality cone.The cone can be very large, e.g. for omnidirectional sources.



The listener's position is described using right-hand Cartesian coordinates; its movement using a velocity vector and the direction the listener's head is pointing using two direction vectors: up and front. These respectively define the direction of the top of the listener's head, and the direction the listener's nose is pointing, and are at right angles to one another.

**Note**: For more information, see our [Web audio spatialization basics](https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API/Web_audio_spatialization_basics) article.

# Fan-in and Fan-out

In audio terms, **fan-in** describes the process by which a [ChannelMergerNode](https://developer.mozilla.org/en-US/docs/Web/API/ChannelMergerNode) takes a series of mono input sources and outputs a single multi-channel signal:

**Fan-out** describes the opposite process, whereby a [ChannelSplitterNode](https://developer.mozilla.org/en-US/docs/Web/API/ChannelSplitterNode) takes a multi-channel input source and outputs multiple mono output signals:

# Article 2 Web Audio API

The Web Audio API provides a powerful and versatile system for controlling audio on the Web, allowing developers to choose audio sources, add effects to audio, create audio visualizations, apply spatial effects (such as panning)  and much more.

# Web audio concepts and usage

The Web Audio API involves handling audio operations inside an **audio context**, and has been designed to allow **modular routing**. Basic audio operations are performed with **audio nodes**, which are linked together to form an **audio routing graph**. Several sources — with different types of channel layout — are supported even within a single context. This modular design provides the flexibility to create complex audio functions with dynamic effects.

Audio nodes are linked into chains and simple webs by their inputs and outputs. They typically start with one or more sources. Sources provide arrays of sound intensities (samples) at very small timeslices, often tens of thousands of them per second. These could be either computed mathematically (see: [OscillatorNode](https://developer.mozilla.org/en-US/docs/Web/API/OscillatorNode)), or they can be recordings from sound/video files (see:[AudioBufferSourceNode](https://developer.mozilla.org/en-US/docs/Web/API/AudioBufferSourceNode) and [MediaElementAudioSourceNode](https://developer.mozilla.org/en-US/docs/Web/API/MediaElementAudioSourceNode)) and audio streams (see:[MediaStreamAudioSourceNode](https://developer.mozilla.org/en-US/docs/Web/API/MediaStreamAudioSourceNode)). In fact, sound files are just recordings of sound intensities themselves, which come in from microphones or electric instruments, and get mixed down into a single, complicated wave. Outputs of these nodes could be linked to inputs of others, which mix or modify these streams of sound samples into different streams. A common modification is DYNAMICS ltiplying the samples by a value to make them louder or quieter (see: [GainNode](https://developer.mozilla.org/en-US/docs/Web/API/GainNode)). Once the sound has been sufficiently processed for the intended effect, it can be linked to the input of a destination (see:[AudioContext.destination](https://developer.mozilla.org/en-US/docs/Web/API/AudioContext/destination)) that sends it to the speakers or headphones. This last connection is only necessary if the user is supposed to hear the audio.

A simple, typical workflow for web audio would look something like this:

1. Create audio context
2. Inside the context, create sources — such as <audio>, oscillator, stream
3. Create effects nodes, such as reverb, biquad filter, panner, compressor
4. Choose final destination of audio, for example your system speakers
5. Connect the sources up to the effects, and the effects to the destination.

Timing is controlled with high precision and low latency, allowing developers to write code that responds accurately to events and is able to target specific samples, even at a high sample rate. So applications such as drum machines and sequencers are well within reach.

The Web Audio API also allows us to control how audio is *spatialized*. Using a system based on a *source-listener model*, it allows control of the *panning model* and deals with *distance-induced attenuation* or*doppler shift* induced by a moving source (or moving listener).

**Note**: You can read about the theory of the Web Audio API in a lot more detail in our article [Basic concepts behind Web Audio API](https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API/Basic_concepts_behind_Web_Audio_API).

# Web Audio API Interfaces

The Web Audio API has a total of 28 interfaces and associated events, which we have split up into nine categories of functionality.

## General audio graph definition - THIS IS CRITICAL – SETS UP THE APPLICATION

General containers and definitions that shape audio graphs in Web Audio API usage.

[**AudioContext**](https://developer.mozilla.org/en-US/docs/Web/API/AudioContext)

The **AudioContext** interface represents an audio-processing graph built from audio modules linked together, each represented by an [AudioNode](https://developer.mozilla.org/en-US/docs/Web/API/AudioNode). An audio context controls the creation of the nodes it contains and the execution of the audio processing, or decoding. You need to create anAudioContext before you do anything else, as everything happens inside a context. THIS IS REALLY IMPORTANT

[**AudioNode**](https://developer.mozilla.org/en-US/docs/Web/API/AudioNode)

The **AudioNode**interface represents an audio-processing module like an *audio source* (e.g. an HTML[<audio>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/audio) or [<video>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/video) element), *audio destination*, *intermediate processing module* (e.g. a filter like[BiquadFilterNode](https://developer.mozilla.org/en-US/docs/Web/API/BiquadFilterNode), or volume control like [GainNode](https://developer.mozilla.org/en-US/docs/Web/API/GainNode)).

[**AudioParam**](https://developer.mozilla.org/en-US/docs/Web/API/AudioParam)

The **AudioParam**interface represents an audio-related parameter, like one of an [AudioNode](https://developer.mozilla.org/en-US/docs/Web/API/AudioNode). It can be set to a specific value or a change in value, and can be scheduled to happen at a specific time and following a specific pattern.

[**ended**](https://developer.mozilla.org/en-US/docs/Web/Events/ended_(Web_Audio))**(event)**

The ended event is fired when playback has stopped because the end of the media was reached.

## Defining audio sources

*Interfaces that define audio sources for use in the Web Audio API.*

[**OscillatorNode**](https://developer.mozilla.org/en-US/docs/Web/API/OscillatorNode)

The **OscillatorNode**interface represents a sine wave. It is an [AudioNode](https://developer.mozilla.org/en-US/docs/Web/API/AudioNode) audio-processing module that causes a given *frequency* of sine wave to be created.

[**AudioBuffer**](https://developer.mozilla.org/en-US/docs/Web/API/AudioBuffer)

The **AudioBuffer** interface represents a short audio asset residing in memory, created from an audio file using the [AudioContext.decodeAudioData()](https://developer.mozilla.org/en-US/docs/Web/API/AudioContext/decodeAudioData) method, or created with raw data using[AudioContext.createBuffer()](https://developer.mozilla.org/en-US/docs/Web/API/AudioContext/createBuffer). Once decoded into this form, the audio can then be put into an[AudioBufferSourceNode](https://developer.mozilla.org/en-US/docs/Web/API/AudioBufferSourceNode).

[**AudioBufferSourceNode**](https://developer.mozilla.org/en-US/docs/Web/API/AudioBufferSourceNode)

The **AudioBufferSourceNode** interface represents an audio source consisting of in-memory audio data, stored in an [AudioBuffer](https://developer.mozilla.org/en-US/docs/Web/API/AudioBuffer). It is an [AudioNode](https://developer.mozilla.org/en-US/docs/Web/API/AudioNode) that acts as an audio source.

[**MediaElementAudioSourceNode**](https://developer.mozilla.org/en-US/docs/Web/API/MediaElementAudioSourceNode)

The **MediaElementAudioSourceNode** interface represents an audio source consisting of an HTML5[<audio>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/audio) or [<video>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/video) element. It is an [AudioNode](https://developer.mozilla.org/en-US/docs/Web/API/AudioNode) that acts as an audio source.

[**MediaStreamAudioSourceNode**](https://developer.mozilla.org/en-US/docs/Web/API/MediaStreamAudioSourceNode)

The **MediaStreamAudioSourceNode** interface represents an audio source consisting of a [WebRTC](https://developer.mozilla.org/en-US/docs/WebRTC)[MediaStream](https://developer.mozilla.org/en-US/docs/Web/API/MediaStream) (such as a webcam or microphone). It is an [AudioNode](https://developer.mozilla.org/en-US/docs/Web/API/AudioNode) that acts as an audio source.

## Defining audio effects filters

Interfaces for defining effects that you want to apply to your audio sources.

[**BiquadFilterNode**](https://developer.mozilla.org/en-US/docs/Web/API/BiquadFilterNode)

The **BiquadFilterNode**interface represents a simple low-order filter. It is an [AudioNode](https://developer.mozilla.org/en-US/docs/Web/API/AudioNode) that can represent different kinds of filters, tone control devices or graphic equalizers. A BiquadFilterNodealways has exactly one input and one output.

[**ConvolverNode**](https://developer.mozilla.org/en-US/docs/Web/API/ConvolverNode)

The **ConvolverNode**interface is an [AudioNode](https://developer.mozilla.org/en-US/docs/Web/API/AudioNode) that performs a Linear Convolution on a given AudioBuffer, often used to achieve a reverb effect.

[**DelayNode**](https://developer.mozilla.org/en-US/docs/Web/API/DelayNode)

The **DelayNode**interface represents a [delay-line](http://en.wikipedia.org/wiki/Digital_delay_line); an [AudioNode](https://developer.mozilla.org/en-US/docs/Web/API/AudioNode) audio-processing module that causes a delay between the arrival of an input data and its propagation to the output.

[**DynamicsCompressorNode**](https://developer.mozilla.org/en-US/docs/Web/API/DynamicsCompressorNode)

The **DynamicsCompressorNode** interface provides a compression effect, which lowers the volume of the loudest parts of the signal in order to help prevent clipping and distortion that can occur when multiple sounds are played and multiplexed together at once.

[**GainNode**](https://developer.mozilla.org/en-US/docs/Web/API/GainNode)

The **GainNode**interface represents a change in volume. It is an [AudioNode](https://developer.mozilla.org/en-US/docs/Web/API/AudioNode) audio-processing module that causes a given *gain* to be applied to the input data before its propagation to the output.

[**StereoPannerNode**](https://developer.mozilla.org/en-US/docs/Web/API/StereoPannerNode)

The **StereoPannerNode** interface represents a simple stereo panner node  that can be used to pan an audio stream left or right.

[**WaveShaperNode**](https://developer.mozilla.org/en-US/docs/Web/API/WaveShaperNode)

The **WaveShaperNode**interface represents a non-linear distorter. It is an [AudioNode](https://developer.mozilla.org/en-US/docs/Web/API/AudioNode) that use a curve to apply a waveshaping distortion to the signal. Beside obvious distortion effects, it is often used to add a warm feeling to the signal.

[**PeriodicWave**](https://developer.mozilla.org/en-US/docs/Web/API/PeriodicWave)

Used to define a periodic waveform that can be used to shape the output of an [OscillatorNode](https://developer.mozilla.org/en-US/docs/Web/API/OscillatorNode).

## Defining audio destinations - THIS IS REALLY IMPORTANT

Once you are done processing your audio, these interfaces define where to output it.

[**AudioDestinationNode**](https://developer.mozilla.org/en-US/docs/Web/API/AudioDestinationNode)

The **AudioDestinationNode** interface represents the end destination of an audio source in a given context — usually the speakers of your device.

[**MediaStreamAudioDestinationNode**](https://developer.mozilla.org/en-US/docs/Web/API/MediaStreamAudioDestinationNode)

The **MediaStreamAudioDestinationNode** interface represents an audio destination consisting of a[WebRTC](https://developer.mozilla.org/en-US/docs/WebRTC) [MediaStream](https://developer.mozilla.org/en-US/docs/Web/API/MediaStream) with a single AudioMediaStreamTrack, which can be used in a similar way to a MediaStream obtained from [Navigator.getUserMedia](https://developer.mozilla.org/en-US/docs/Web/API/Navigator/getUserMedia). It is an [AudioNode](https://developer.mozilla.org/en-US/docs/Web/API/AudioNode) that acts as an audio destination.

## Data analysis and visualisation

If you want to extract time, frequency and other data from your audio, the AnalyserNode is what you need.

[**AnalyserNode**](https://developer.mozilla.org/en-US/docs/Web/API/AnalyserNode)

The **AnalyserNode** interface represents a node able to provide real-time frequency and time-domain analysis information, for the purposes of data analysis and visualization.

## Splitting and merging audio channels

To split and merge audio channels, you'll use these interfaces.

[**ChannelSplitterNode**](https://developer.mozilla.org/en-US/docs/Web/API/ChannelSplitterNode)

The **ChannelSplitterNode** interface separates the different channels of an audio source out into a set of *mono* outputs.

[**ChannelMergerNode**](https://developer.mozilla.org/en-US/docs/Web/API/ChannelMergerNode)

The **ChannelMergerNode** interface reunites different mono inputs into a single output. Each input will be used to fill a channel of the output.

## Audio spatialization

These interfaces allow you to add audio spatialization panning effects to your audio sources.

[**AudioListener**](https://developer.mozilla.org/en-US/docs/Web/API/AudioListener)

The **AudioListener**interface represents the position and orientation of the unique person listening to the audio scene used in audio spatialization.

[**PannerNode**](https://developer.mozilla.org/en-US/docs/Web/API/PannerNode)

The **PannerNode**interface represents the behavior of a signal in space. It is an [AudioNode](https://developer.mozilla.org/en-US/docs/Web/API/AudioNode) audio-processing module describing its position with right-hand Cartesian coordinates, its movement using a velocity vector and its directionality using a directionality cone.

## Audio processing via JavaScript – this looks kind of important! STATUS - SEE AUDIO WORKERS BELOW

If you want to use an external script to process your audio source, the below Node and events make it possible.

**Note**: As of the **August 29 2014** Web Audio API spec publication, these features have been marked as deprecated, and are soon to be replaced by [Audio\_Workers](https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API#Audio_Workers).

[**ScriptProcessorNode**](https://developer.mozilla.org/en-US/docs/Web/API/ScriptProcessorNode)

The **ScriptProcessorNode**interface allows the generation, processing, or analyzing of audio using JavaScript. It is an [AudioNode](https://developer.mozilla.org/en-US/docs/Web/API/AudioNode) audio-processing module that is linked to two buffers, one containing the current input, one containing the output. An event, implementing the [AudioProcessingEvent](https://developer.mozilla.org/en-US/docs/Web/API/AudioProcessingEvent)interface, is sent to the object each time the input buffer contains new data, and the event handler terminates when it has filled the output buffer with data.

[**audioprocess**](https://developer.mozilla.org/en-US/docs/Web/Events/audioprocess)**(event)**

The audioprocess event is fired when an input buffer of a Web Audio API [ScriptProcessorNode](https://developer.mozilla.org/en-US/docs/Web/API/ScriptProcessorNode)is ready to be processed.

[**AudioProcessingEvent**](https://developer.mozilla.org/en-US/docs/Web/API/AudioProcessingEvent)

The [Web Audio API](https://developer.mozilla.org/en-US/docs/Web_Audio_API) AudioProcessingEvent represents events that occur when a[ScriptProcessorNode](https://developer.mozilla.org/en-US/docs/Web/API/ScriptProcessorNode) input buffer is ready to be processed.

## Offline/background audio processing

It is possible to process/render an audio graph very quickly in the background — rendering it to an[AudioBuffer](https://developer.mozilla.org/en-US/docs/Web/API/AudioBuffer) rather than to the device's speakers — with the following.

[**OfflineAudioContext**](https://developer.mozilla.org/en-US/docs/Web/API/OfflineAudioContext)

The **OfflineAudioContext** interface is an [AudioContext](https://developer.mozilla.org/en-US/docs/Web/API/AudioContext) interface representing an audio-processing graph built from linked together [AudioNode](https://developer.mozilla.org/en-US/docs/Web/API/AudioNode)s. In contrast with a standardAudioContext, an OfflineAudioContext doesn't really render the audio but rather generates it,*as fast as it can*, in a buffer.

[**complete**](https://developer.mozilla.org/en-US/docs/Web/Events/complete)**(event)**

The complete event is fired when the rendering of an [OfflineAudioContext](https://developer.mozilla.org/en-US/docs/Web/API/OfflineAudioContext) is terminated.

[**OfflineAudioCompletionEvent**](https://developer.mozilla.org/en-US/docs/Web/API/OfflineAudioCompletionEvent)

The OfflineAudioCompletionEvent represents events that occur when the processing of an[OfflineAudioContext](https://developer.mozilla.org/en-US/docs/Web/API/OfflineAudioContext) is terminated. The [complete](https://developer.mozilla.org/en-US/docs/Web/Events/complete) event implements this interface.

## Audio Workers - IMPORTANT

Audio workers provide the ability for direct scripted audio processing to be done inside a [web worker](https://developer.mozilla.org/en-US/docs/Web/Guide/Performance/Using_web_workers) context, and are defined by a couple of interfaces (new as of 29th August 2014.) These are not implemented in any browsers yet. When implemented, they will replace [ScriptProcessorNode](https://developer.mozilla.org/en-US/docs/Web/API/ScriptProcessorNode), and the other features discussed in the [Audio processing via JavaScript](https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API#Audio_processing_via_JavaScript) section above.

[**AudioWorkerNode**](https://developer.mozilla.org/en-US/docs/Web/API/AudioWorkerNode)

The AudioWorkerNode interface represents an [AudioNode](https://developer.mozilla.org/en-US/docs/Web/API/AudioNode) that interacts with a worker thread to generate, process, or analyse audio directly.

[**AudioWorkerGlobalScope**](https://developer.mozilla.org/en-US/docs/Web/API/AudioWorkerGlobalScope)

The AudioWorkerGlobalScope interface is a DedicatedWorkerGlobalScope-derived object representing a worker context in which an audio processing script is run; it is designed to enable the generation, processing, and analysis of audio data directly using JavaScript in a worker thread.

[**AudioProcessEvent**](https://developer.mozilla.org/en-US/docs/Web/API/AudioProcessEvent)

This is an Event object that is dispatched to [AudioWorkerGlobalScope](https://developer.mozilla.org/en-US/docs/Web/API/AudioWorkerGlobalScope) objects to perform processing.

## Obsolete interfaces

The following interfaces were defined in old versions of the Web Audio API spec, but are now obsolete and have been replaced by other interfaces.

[**JavaScriptNode**](https://developer.mozilla.org/en-US/docs/Web/API/JavaScriptNode)

Used for direct audio processing via JavaScript. This interface is obsolete, and has been replaced by[ScriptProcessorNode](https://developer.mozilla.org/en-US/docs/Web/API/ScriptProcessorNode).

[**WaveTableNode**](https://developer.mozilla.org/en-US/docs/Web/API/WaveTableNode)

Used to define a periodic waveform. This interface is obsolete, and has been replaced by[PeriodicWave](https://developer.mozilla.org/en-US/docs/Web/API/PeriodicWave).

# Example

This example shows a wide variety of Web Audio API functions being used. You can see this code in action on the [Voice-change-o-matic](http://mdn.github.io/voice-change-o-matic/) demo (also check out the [full source code at Github](https://github.com/mdn/voice-change-o-matic)) — this is an experimental voice changer toy demo; keep your speakers turned down low when you use it, at least to start!

The Web Audio API lines are highlighted; if you want to find more out about what the different methods, etc. do, have a search around the reference pages.

var audioCtx = new (window.AudioContext || window.webkitAudioContext)(); // define audio context

// Webkit/blink browsers need prefix, Safari won't work without window.

var voiceSelect = document.getElementById("voice"); // select box for selecting voice effect options

var visualSelect = document.getElementById("visual"); // select box for selecting audio visualization options

var mute = document.querySelector('.mute'); // mute button

var drawVisual; // requestAnimationFrame

var analyser = audioCtx.createAnalyser();

var distortion = audioCtx.createWaveShaper();

var gainNode = audioCtx.createGain();

var biquadFilter = audioCtx.createBiquadFilter();

function makeDistortionCurve(amount) { // function to make curve shape for distortion/wave shaper node to use

var k = typeof amount === 'number' ? amount : 50,

n\_samples = 44100,

curve = new Float32Array(n\_samples),

deg = Math.PI / 180,

i = 0,

x;

for ( ; i < n\_samples; ++i ) {

x = i \* 2 / n\_samples - 1;

curve[i] = ( 3 + k ) \* x \* 20 \* deg / ( Math.PI + k \* Math.abs(x) );

}

return curve;

};

navigator.getUserMedia (

// constraints - only audio needed for this app

{

audio: true

},

// Success callback

function(stream) {

source = audioCtx.createMediaStreamSource(stream);

source.connect(analyser);

analyser.connect(distortion);

distortion.connect(biquadFilter);

biquadFilter.connect(gainNode);

gainNode.connect(audioCtx.destination); // connecting the different audio graph nodes together

visualize(stream);

voiceChange();

},

// Error callback

function(err) {

console.log('The following gUM error occured: ' + err);

}

);

function visualize(stream) {

WIDTH = canvas.width;

HEIGHT = canvas.height;

var visualSetting = visualSelect.value;

console.log(visualSetting);

if(visualSetting == "sinewave") {

analyser.fftSize = 2048;

var bufferLength = analyser.frequencyBinCount; // half the FFT value

var dataArray = new Uint8Array(bufferLength); // create an array to store the data

canvasCtx.clearRect(0, 0, WIDTH, HEIGHT);

function draw() {

drawVisual = requestAnimationFrame(draw);

analyser.getByteTimeDomainData(dataArray); // get waveform data and put it into the array created above

canvasCtx.fillStyle = 'rgb(200, 200, 200)'; // draw wave with canvas

canvasCtx.fillRect(0, 0, WIDTH, HEIGHT);

canvasCtx.lineWidth = 2;

canvasCtx.strokeStyle = 'rgb(0, 0, 0)';

canvasCtx.beginPath();

var sliceWidth = WIDTH \* 1.0 / bufferLength;

var x = 0;

for(var i = 0; i < bufferLength; i++) {

var v = dataArray[i] / 128.0;

var y = v \* HEIGHT/2;

if(i === 0) {

canvasCtx.moveTo(x, y);

} else {

canvasCtx.lineTo(x, y);

}

x += sliceWidth;

}

canvasCtx.lineTo(canvas.width, canvas.height/2);

canvasCtx.stroke();

};

draw();

} else if(visualSetting == "off") {

canvasCtx.clearRect(0, 0, WIDTH, HEIGHT);

canvasCtx.fillStyle = "red";

canvasCtx.fillRect(0, 0, WIDTH, HEIGHT);

}

}

function voiceChange() {

distortion.curve = new Float32Array;

biquadFilter.gain.value = 0; // reset the effects each time the voiceChange function is run

var voiceSetting = voiceSelect.value;

console.log(voiceSetting);

if(voiceSetting == "distortion") {

distortion.curve = makeDistortionCurve(400); // apply distortion to sound using waveshaper node

} else if(voiceSetting == "biquad") {

biquadFilter.type = "lowshelf";

biquadFilter.frequency.value = 1000;

biquadFilter.gain.value = 25; // apply lowshelf filter to sounds using biquad

} else if(voiceSetting == "off") {

console.log("Voice settings turned off"); // do nothing, as off option was chosen

}

}

// event listeners to change visualize and voice settings

visualSelect.onchange = function() {

window.cancelAnimationFrame(drawVisual);

visualize(stream);

}

voiceSelect.onchange = function() {

voiceChange();

}

mute.onclick = voiceMute;

function voiceMute() { // toggle to mute and unmute sound

if(mute.id == "") {

gainNode.gain.value = 0; // gain set to 0 to mute sound

mute.id = "activated";

mute.innerHTML = "Unmute";

} else {

gainNode.gain.value = 1; // gain set to 1 to unmute sound

mute.id = "";

mute.innerHTML = "Mute";

}

}

Specifications

|  |  |  |
| --- | --- | --- |
| **Specification** | **Status** | **Comment** |
| [Web Audio API](https://webaudio.github.io/web-audio-api/) | Working Draft |  |

Browser compatibility

* **Desktop**

* Mobile

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Feature | Chrome | Edge | Firefox (Gecko) | Internet Explorer | Opera | Safari (WebKit) |
| Basic support | 14 [webkit](https://developer.mozilla.org/en-US/docs/Web/Guide/Prefixes) | (Yes) | 23 | No support | 15 [webkit](https://developer.mozilla.org/en-US/docs/Web/Guide/Prefixes) 22 (unprefixed) | 6 [webkit](https://developer.mozilla.org/en-US/docs/Web/Guide/Prefixes) |

See also

* [Using the Web Audio API](https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API/Using_Web_Audio_API)
* [Visualizations with Web Audio API](https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API/Visualizations_with_Web_Audio_API)
* [Voice-change-O-matic example](http://mdn.github.io/voice-change-o-matic/)
* [Violent Theremin example](http://mdn.github.io/violent-theremin/)
* [Web audio spatialisation basics](https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API/Web_audio_spatialisation_basics)
* [Mixing Positional Audio and WebGL](http://www.html5rocks.com/tutorials/webaudio/positional_audio/)
* [Developing Game Audio with the Web Audio API](http://www.html5rocks.com/tutorials/webaudio/games/)
* [Porting webkitAudioContext code to standards based AudioContext](https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API/Porting_webkitAudioContext_code_to_standards_based_AudioContext)
* [Tones](https://github.com/bit101/tones): a simple library for playing specific tones/notes using the Web Audio API.
* [howler.js](https://github.com/goldfire/howler.js/): a JS audio library that defaults to [Web Audio API](https://dvcs.w3.org/hg/audio/raw-file/tip/webaudio/specification.html) and falls back to [HTML5 Audio](http://www.whatwg.org/specs/web-apps/current-work/#the-audio-element), as well as providing other useful features.
* [Mooog](https://github.com/mattlima/mooog): jQuery-style chaining of AudioNodes, mixer-style sends/returns, and more.