

**Quick Guides
for Masterminds**

Web Audio API

JavaScript

Take advantage of the Web Audio API to add audio processing to your web applications and video games

J.D Gauchat
www.jdgauchat.com

Quick Guides for Masterminds
Copyright © 2026 John D Gauchat
All Rights Reserved

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law.

Companies, services, or product names used in this guide are for identification purposes only. All trademarks and registered trademarks are the property of their respective owners.

The information in this guide is distributed without warranty. Although every precaution has been taken in the preparation of this work, neither the author nor the publisher shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

Copyright Registration Number: 1140725

1st Edition 2026

Table of Contents

WEB AUDIO API

- Audio Structure
- Audio Context
- Audio Sources
- Connecting Nodes

AUDIO APPLICATION

- Loops and Times
- Audio Nodes
- AudioParam
- GainNode
- DelayNode
- BiquadFilterNode
- DynamicsCompressorNode
- ConvolverNode
- PannerNode and 3D Sound
- AnalyserNode

Web Audio API

Audio Structure

The Web Audio API is a JavaScript API provided by modern browsers to develop audio production applications and game audio engines for the Web. To that end, the API uses a modular organization where everything is a node (called an **AudioNode**). Nodes represent every part of the audio system, from raw sound to the speakers, and they are connected to each other to form the final structure that produces the sound.



Figure 1: Basic node organization

Figure 1 represents the most basic structure possible: a node containing the source of the audio and a node for the destination (speakers, headphones, or whatever output device the system is configured to use). This is the simplest case, a node producing the sound and a node making it audible, but we may have additional nodes for audio sources, filters, volume controls, delays, or effects. The structure may be as complex as our application requires, as shown below.

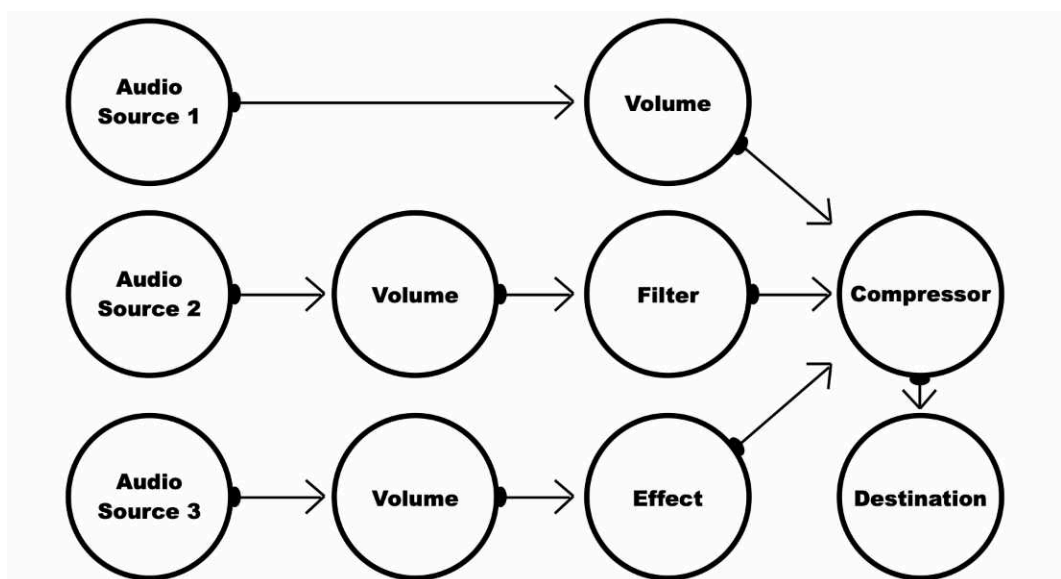


Figure 2: Complex audio system

The first audio source node in Figure 2 is connected to the volume node (called **GainNode**), which in turn is connected to the compressor node (called **DynamicsCompressorNode**), which is finally connected to the destination node (speakers or headphones). The volume node adjusts the volume of this audio source, the compressor node processes it, and then the result is

automatically mixed with the other sounds and sent to the destination node. The sources for audio 2 and 3 in this figure follow a similar path, but they pass through the filter node (called **BiquadFilterNode**) and the effect node (called **ConvolverNode**) before reaching the compressor node and the destination.

Most nodes have an input and output connection to receive sound from a previous node, perform some kind of process, and send the result to the next one. Our work is to create the nodes, provide the information each one requires to work properly, and connect them.

Audio Context

The structure of nodes is set in an audio context. The context is represented by an object of type **AudioContext**. The Web Audio API provides the following constructor to get this object.

AudioContext()

This constructor returns an **AudioContext** object. The object provides methods to create every type of node available, as well as some properties to configure the context and access the destination node that leads to the device's audio output.

The following are the properties provided by the **AudioContext** object to configure the context and the audio system.

destination

This property returns an **AudioDestinationNode** that represents the destination node. This is the last node of an audio system, and its function is to provide access to the device's audio output.

currentTime

This property returns the time in seconds since the context was created.

sampleRate

This property returns the rate (samples per second) at which the audio is being processed.

listener

This property returns an **AudioListener** object with properties and methods to calculate the position and orientation of the listener in a 3D scene.

Audio Sources

The most important nodes are the audio source nodes we have depicted as the starting point of every path in the previous figures. Without these primary sources of audio, we would not have any sound to send to the speakers. These types of nodes may be created from different sources: audio buffers in memory, media streams, or media elements. The API provides the following methods to get the objects to represent each source.

createBufferSource()

This method returns an **AudioBufferSourceNode** object. The object is created from an audio buffer in memory and provides its own properties and methods to play and configure the audio source.

createMediaStreamSource(stream)

This method returns a **MediaStreamAudioSourceNode** object. The object is created from a media stream. The **stream** argument is a stream generated by methods like **getUserMedia()** (the user's web cam, for example).

createMediaElementSource(element)

This method returns a **MediaElementAudioSourceNode** object. The object is created from a media element. The element argument is a reference to an **<audio>** or **<video>** element.

The **createMediaStreamSource()** and **createMediaElementSource()** methods work with streams and media elements that already have their own controls to play, pause, or stop the media, but the object returned by **createBufferSource()** has no such controls of its own. Instead, it provides the following properties and methods to play and configure the audio source.

loop

This property sets or returns a Boolean value that determines if the sound assigned as the source of the node is going to be played in a loop.

buffer

This property sets or returns the audio buffer used as the source of the node.

start(time, offset, duration)

This method starts playing the sound assigned as the source of the node. The **time** argument determines when the action will take place (in seconds). The **offset** and **duration** arguments are optional and determine which part of the buffer must be played, starting from **offset** and as long as determined by **duration** (in seconds).

stop(time)

This method stops playing the sound assigned as the source of the node. The **time** argument determines when the action will take place (in seconds).

The object returned by the **createBufferSource()** method does not work directly with audio files. The files have to be downloaded from the server, stored in memory as audio buffers, and assigned to the audio source node with the **buffer** property. The API provides the following method to convert the audio data from a file into a buffer in memory.

decodeAudioData(arraybuffer)

This asynchronous method returns an **AudioBuffer** object created from binary data. The **arraybuffer** argument is the binary data from the audio file (**ArrayBuffer** type).

Information from audio buffers may be retrieved by the following properties.

duration

This property returns the duration of the buffer in seconds.

length

This property returns the length of the buffer in sample frames.

numberOfChannels

This property returns the number of channels available in the buffer.

sampleRate

This property returns the sample rate of the buffer in samples per second.

Connecting Nodes

There is one more thing we need to study before creating our first audio structure, and that is how to connect the nodes. Audio nodes have input and output connections that allow us to create a path for the sound to follow. The API provides two methods to build and manage this network of nodes.

connect (node)

This method connects the output from one node to the input of another. The syntax is **output.connect(input)**, where **output** is a reference to the node that provides the output, and **input** is a reference to the node that receives the audio from that output.

disconnect()

This method disconnects the node's output. The syntax is **output.disconnect()**, where **output** is a reference to the node to be disconnected.

Audio Application

Usually, sounds are part of complex JavaScript code, like the code necessary to create a 3D video game or an audio production application, but to keep this example simple, we are going to use a basic document that focuses on creating and connecting the audio nodes.

Listing 1: Creating a document to play sounds

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Web Audio API</title>
  <script src="audio.js"></script>
</head>
<body>
  <section>
    <button type="button" id="button" disabled>Play</button>
  </section>
</body>
```

</html>

As we explained before, a basic node structure requires a node for the source and another for the destination. The destination node is returned by the context's **destination** property, but the audio source requires some work. We have to download the audio file from the server, convert the content to an audio buffer, use the buffer as the source of the node, and finally connect both nodes to produce sound through the speakers. For our example, we are going to download a WAV file using the **Fetch API** (for more information about this API, read our book *HTML5 for Masterminds*).

Listing 2: Playing an audio buffer

```
let context;
async function initiate() {
  let mybuffer;
  const button = document.getElementById("button");
  button.addEventListener("click", function() {
    play(mybuffer);
  });
  context = new AudioContext();

  try {
    const response = await fetch("gunshot.wav");
    if (!response.ok) {
      throw new Error("HTTP error: " + response.status);
    }
    const arraybuffer = await response.arrayBuffer();
    mybuffer = await context.decodeAudioData(arraybuffer);
    button.disabled = false;
  } catch (error) {
    console.error(error);
  }
}
function play(mybuffer) {
  const sourceNode = context.createBufferSource();
  sourceNode.buffer = mybuffer;
  sourceNode.connect(context.destination);
  sourceNode.start(0);
}
window.addEventListener("DOMContentLoaded", initiate);
```

This code defines a function called **initiate()** that we execute as soon as the document is loaded to initialize the application (**DOMContentLoaded**). The function starts by getting a reference to the button in the document and adding a listener for the **click** event to execute the **play()** function every time the button is pressed. Immediately after, the audio context is created by the **AudioContext()** constructor and stored in the context variable. The **gunshot.wav** audio file we want to play is downloaded next using the **Fetch API**. We get the file, convert it to an **ArrayBuffer** with the **arrayBuffer()** method, then decode it with the **decodeAudioData()** method, and finally enable the Play button to let the user play the audio.

The whole audio system, including nodes and connections, has to be built every time we want to play a sound. The **initiate()** function takes care of downloading the file and turning its content into an audio buffer in memory, but the rest is done by the **play()** function. Every time the Play button is clicked, the audio system is created by this function. First, the audio source node is created by the **createBufferSource()** method, and then the audio buffer is assigned as the value of the **buffer** property of this node. This process links the node to the audio buffer in memory. After this node is ready, it is connected to the destination node by the **connect()** method, and the sound is finally played with the **start()** method.

Notice that the button is initially disabled, but it is enabled in the **initiate()** function after the data necessary to create the audio system is ready. This is just a design choice to keep the example simple. Once enabled, every time the button is clicked, the audio system is created again, and the sound from the **gunshot.wav** audio file is played.

Do It Yourself

Open your favorite editor (e.g., Visual Studio Code). Create a folder to store this project. Create a new HTML file with the document in Listing 1 and a file called **audio.js** with the code in Listing 2. You can download the **gunshot.wav** file from our website or use your own audio file. Upload all the files to your server or run them through a local server. Open the document in your browser and click the Play button. You should hear the sound playing on your speakers.

IMPORTANT

Some features of the Web Audio API only work when the files are loaded from a server. To test the examples in this guide, you must upload the files to your server or run a local server. There are many applications that can run a server on a computer. The most popular are **MAMP** (www.mamp.info) and **XAMPP** (www.apachefriends.org). For information on how to install and run these servers, visit their websites or read our book *HTML5 for Masterminds*.

Loops and Times

The **start()** method at the end of the **play()** function in the last example was called with the value **0**. When this value is set to **0** or is less than the value of the context's current time, the sound starts playing immediately. Once it is played, the **start()** method cannot be called again to play the same audio source node; we have to recreate the whole audio system. This is the reason why we constructed the audio system in a separate function. Despite this restriction, there are different ways to play a sound several times without having to call the **play()** function over and over again. One alternative is to loop this audio source node by setting its **loop** property to **true**, as shown below.

Listing 3: Playing the audio source node in a loop

```
function play(mybuffer) {  
  const sourceNode = context.createBufferSource();  
  sourceNode.buffer = mybuffer;
```

```
sourceNode.loop = true;
sourceNode.connect(context.destination);
sourceNode.start(0);
sourceNode.stop(context.currentTime + 3);
}
```

The loop will run indefinitely until the **stop()** method is called. In the code in Listing 3, we use this method to stop the loop after 3 seconds. The time is set using the value returned by the **currentTime** property. By adding the value **3** to the current time, we instruct the browser to stop playing the sound 3 seconds later.

Do It Yourself

Replace the **play()** function in Listing 2 with the new function in Listing 3. Upload the **audio.js** file to your server. Open the document in your browser and click the Play button. The audio should play in a loop for 3 seconds.

Audio Nodes

Every node in the audio system is called an **AudioNode**. The basic nodes we need to produce sound are the audio source node and the destination node created in the previous example, but these are not the only nodes available. The API provides methods to build a variety of audio nodes to process, analyze, and even create sounds.

createAnalyser()

This method creates an **AnalyserNode**. These types of nodes provide access to information of the audio source. The object returned includes several properties and methods for this purpose. It is often used to visualize the audio stream.

createGain()

This method creates a **GainNode**. These types of nodes set the gain of the audio signal (the volume). The object returned provides the **gain** property to control the volume, where **1.0** keeps the original volume, values below **1.0** reduce it, and values above **1.0** amplify it.

createDelay(maxDelayTime)

This method creates a **DelayNode**. These types of nodes set a delay for the audio source. The object returned provides the **delayTime** property to set the delay in seconds. The **maxDelayTime** argument is optional and declares the maximum time for the delay in seconds.

createBiquadFilter()

This method creates a **BiquadFilterNode**. These types of nodes apply filters to the audio signal. The object returned provides properties and methods to set the characteristics of the filter.

createWaveShaper()

This method creates a **WaveShaperNode**. These types of nodes apply a distortion effect based on a shaping curve. The object returned provides the curve property for declaring the shaping curve (**Float32Array** type).

createPanner()

This method creates a **PannerNode**. These types of nodes are used to determine the position and orientation of the sound in a three-dimensional space. The effect produced depends on the current values set for the listener. The object returned provides several methods and properties to configure the parameters of the node.

createConvolver()

This method creates a **ConvolverNode**. These types of nodes apply a convolution effect to the audio signal based on an impulse response. The object returned provides two properties: **buffer** and **normalize**. The **buffer** property is necessary to set the buffer to be used as the impulse response, and the **normalize** property receives a Boolean value to set whether or not the impulse response will be scaled.

createChannelSplitter(numberOfOutputs)

This method creates a **ChannelSplitterNode**. These types of nodes generate different outputs for each channel of the audio stream. The **numberOfOutputs** argument declares the number of outputs to be generated.

createChannelMerger(numberOfInputs)

This method creates a **ChannelMergerNode**. These types of nodes combine channels from multiple audio streams into one. The **numberOfInputs** argument declares the number of inputs to be merged.

createDynamicsCompressor()

This method creates a **DynamicsCompressorNode**. These types of nodes apply a compression effect to the audio signal. The object returned provides several properties to configure the effect.

createOscillator()

This method creates an **OscillatorNode**. These types of nodes generate waveforms for audio synthesis. The object returned provides several properties and methods to configure the waveform.

IMPORTANT

The application of some of these methods requires knowledge of audio engineering. The topic is beyond the scope of this guide, but we will study how to implement the methods needed to build common audio applications, as well as 2D and 3D video games, next.

AudioParam

Besides the common properties and methods provided by audio node objects to configure the node, there is an additional object defined by the API to set specific parameters. The **AudioParam** object is like a control unit linked to the node, ready to adjust its values at any time.

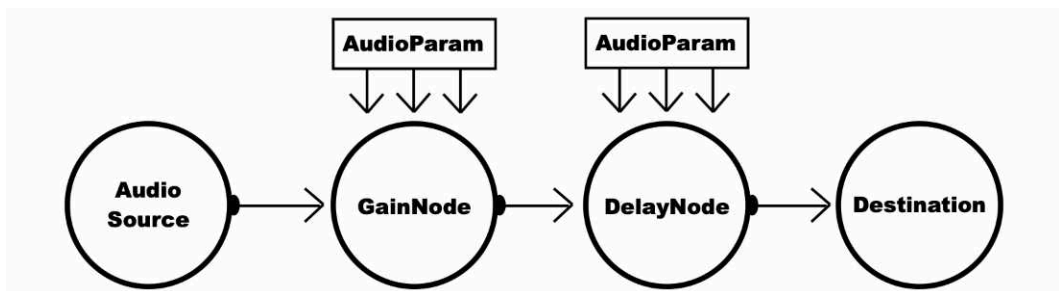


Figure 3: Nodes controlled by AudioParam properties and methods

This object can set values immediately or be configured to do it later following a schedule. Listed below are the properties and methods provided for this purpose.

value

This property sets or returns the current value of the parameter immediately, without following a schedule.

setValueAtTime(value, startTime)

This method sets the value of the parameter at a given time. The **value** and **startTime** arguments declare the new value and the time in seconds, respectively.

linearRampToValueAtTime(value, endTime)

This method changes the value of the parameter gradually from the previous value to the one specified in the arguments. The **value** and **endTime** arguments declare the new value and the end time of the process in seconds, respectively.

exponentialRampToValueAtTime(value, endTime)

This method changes the current value of the parameter exponentially to the value specified by the arguments. The **value** and **endTime** arguments declare the new value and the ending time of the process in seconds, respectively.

setTargetAtTime(target, startTime, timeConstant)

This method changes the value of the parameter exponentially to the value of the **target** argument. The **startTime** argument declares the starting time of the process, and **timeConstant** sets the rate at which the previous value will approach the new one.

setValueCurveAtTime(values, startTime, duration)

This method changes the value of the parameter to follow an arbitrary curve traced by the array declared by the **values** argument (**Float32Array** type), spread evenly across the given duration.

cancelScheduledValues(startTime)

This method cancels any previously scheduled changes at or after the time specified by the **startTime** argument.

GainNode

The first thing we generally want to do when playing a sound is to turn the volume up or down. For this purpose, we are going to introduce a **GainNode** between the audio source node and the destination node created in previous examples.

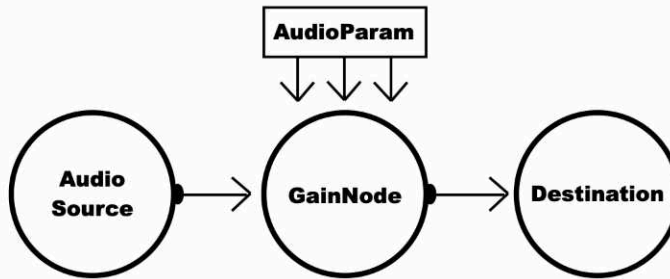


Figure 4: GainNode controlling volume

When we add a new node to an audio system, the basic structure remains the same; we just have to create the new node, provide the configuration values, and connect it to the rest of the structure. In this case, the audio source node has to be connected to the **GainNode** and the **GainNode** to the destination, as shown below.

Listing 4: Turning down the volume using a GainNode

```
function play(mybuffer) {
  const sourceNode = context.createBufferSource();
  sourceNode.buffer = mybuffer;

  const volumeNode = context.createGain();
  volumeNode.gain.value = 0.2;
  sourceNode.connect(volumeNode);
  volumeNode.connect(context.destination);
  sourceNode.start(0);
}
```

The new **play()** function in Listing 4 introduces a **GainNode** to our audio system. The node is created by the **createGain()** method. Its **gain** property is an **AudioParam**, so we set its **value** property to **0.2** to turn the volume down. This property is not limited to a fixed range: **0.0** mutes the sound, **1.0** keeps the original volume, and values above **1.0** amplify it further. By default, the value of gain is **1**, so setting it to **0.2** in this example reduces the volume to 20% of the original.

The connections made at the end of the code include the new **GainNode**. First, the audio source node is connected to the **GainNode** (**sourceNode.connect(volumeNode)**), and then the **GainNode** is connected to the destination node (**volumeNode.connect(context.destination)**).

The value of the **gain** property for this example was set to a fixed value using the **value** property because the audio file contains only a short sound of a gunshot, but we could have used any of the **AudioParam** methods instead to increase or reduce the volume at different times. Methods such as **linearRampToValueAtTime()** may be used in combination with the **duration** property of the buffers to mix songs, fading out one track and fading in the next one.

Do It Yourself

Replace the **play()** function in Listing 2 with the new function in Listing 4. Upload the file to your server, open the document in your browser, and click the Play button. The audio should play again, but at a reduced volume.

DelayNode

The purpose of a **DelayNode** is to delay the sound by the amount of time specified by its **delayTime** property (an **AudioParam** object). In the following example, we are going to introduce this node to play the gunshot sound one second later.

Listing 5: Introducing delay

```
function play(mybuffer) {
  const sourceNode = context.createBufferSource();
  sourceNode.buffer = mybuffer;

  const delayNode = context.createDelay();
  delayNode.delayTime.value = 1;
  sourceNode.connect(delayNode);
  delayNode.connect(context.destination);
  sourceNode.start(0);
}
```

The **play()** function in Listing 5 replaces the **GainNode** in the previous example with a **DelayNode**. The delay is set to 1 second using **delayTime**'s **value** property, and the connections are set following the same path as before: the audio source node is connected to the **DelayNode**, and this node is connected to the destination node.

A **DelayNode** produces better results when it is combined with other nodes, as in the following example.

Listing 6: Getting an echo effect with the DelayNode

```
function play(mybuffer) {
  const sourceNode = context.createBufferSource();
  sourceNode.buffer = mybuffer;

  const delayNode = context.createDelay();
  delayNode.delayTime.value = 0.3;

  const volumeNode = context.createGain();
  volumeNode.gain.value = 0.2;

  sourceNode.connect(delayNode);
  delayNode.connect(volumeNode);
  volumeNode.connect(context.destination);
  sourceNode.connect(context.destination);
  sourceNode.start(0);
}
```

In the `play()` function in Listing 6, a **GainNode** is added in the middle to reduce the volume of the delayed sound and generate an echo effect. Two paths are set for the audio source. In one path, the audio source node is connected to the **DelayNode**, the **DelayNode** to the **GainNode**, and the **GainNode** to the destination node. The sound on this path will have a low volume and will be played with a delay of 0.3 seconds. The second path for the source is a direct connection to the destination node. The sound on this path will be played immediately at full volume. The final structure is shown below.

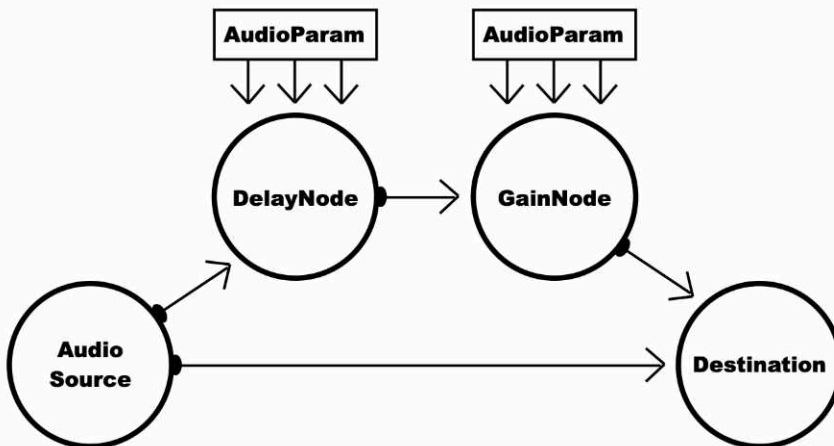


Figure 5: Two paths for the same audio source

Do It Yourself

Replace the `play()` function in Listing 2 with the new function in Listing 5. Upload the `audio.js` file to your server, open the document in your browser, and click the Play button. The sound should start one second later. Replace the function with the one in Listing 6 and repeat the test. This time, you should hear two sounds, one right after the other, simulating an echo effect.

BiquadFilterNode

The **BiquadFilterNode** lets us apply common filters to the audio signal. The node is created by the `createBiquadFilter()` method, and the type of filter is selected by the `type` property. This property takes a string. The possible values are "lowpass", "highpass", "bandpass", "lowshelf", "highshelf", "peaking", "notch", and "allpass". The node also includes the `frequency`, `Q`, and `gain` properties to configure the filter. The effect produced by the values of these properties on the filter depends on the type of filter applied. The following example shows how to implement a highpass effect.

Listing 7: Adding a filter with the BiquadFilterNode

```
function play(mybuffer) {  
  const sourceNode = context.createBufferSource();  
  sourceNode.buffer = mybuffer;
```

```

const filterNode = context.createBiquadFilter();
filterNode.type = "highpass";
filterNode.frequency.value = 1000;

sourceNode.connect(filterNode);
filterNode.connect(context.destination);
sourceNode.start(0);
}

```

In this example, we declare the type of the filter as "highpass" and set the cutoff frequency as 1000. This will cut off some frequencies of the spectrum, making the gunshot sound like a toy gun. Note that the **frequency** property is an **AudioParam**, so its **value** property is used here to set the cutoff frequency directly. This could be replaced with any of the **AudioParam** methods instead, to vary the frequency over time.

IMPORTANT

The effect produced by the values of the **frequency**, **Q**, and **gain** properties depends on the filter selected. Some values will not produce any effect at all. For more information, read the API's specification.

DynamicsCompressorNode

Compressors smooth the audio, reducing the volume of loud sounds and raising the volume of quiet sounds. These types of nodes are usually connected at the end of the audio system with the purpose of coordinating the audio levels of the sources to make them sound unified. The Web Audio API includes the **DynamicsCompressorNode** to produce this effect on the audio signal. The node has several properties to configure the compression: **threshold** (decibels), **knee** (decibels), **ratio** (values from 1 to 20), **attack** (seconds), and **release** (seconds). It also provides a read-only **reduction** property (decibels) that reports how much the signal is currently being attenuated (useful for metering rather than for configuration). The following example shows how to apply a compressor at the end of our audio system.

Listing 8: Adding a dynamic compressor

```

function play(mybuffer) {
  const sourceNode = context.createBufferSource();
  sourceNode.buffer = mybuffer;

  const compressorNode = context.createDynamicsCompressor();
  compressorNode.threshold.value = -60;
  compressorNode.ratio.value = 10;

  sourceNode.connect(compressorNode);
  compressorNode.connect(context.destination);
  sourceNode.start(0);
}

```

This code shows how to create and configure a **DynamicsCompressorNode**, but this node is rarely used this way. This type of node is usually placed at the end of elaborate audio systems that include several audio sources, to control the volume of the combined sound to prevent distortion. In any case, the implementation is the same: we connect all the nodes at the end of the structure to the **DynamicsCompressorNode** and then connect this node to the destination node, as we did in our example.

ConvolverNode

The **ConvolverNode** applies convolution effects to an audio signal. It is commonly used to simulate different acoustic spaces and achieve complex sound distortions, like a voice on a telephone, for example. The effect is achieved by convolving the audio signal with an impulse response, usually a recording of the space we are trying to simulate, loaded from a second audio file. Because of this requirement, to implement the effect, we have to download at least two audio files, one for the source and another for the impulse response, as in the following example.

Listing 9: Applying a convolution effect

```
let context;
async function initiate() {
  let mybuffers;
  const button = document.getElementById("button");
  button.addEventListener("click", function() {
    play(mybuffers);
  });
  context = new AudioContext();

  try {
    const buffer1 = await loadbuffer("gunshot.wav");
    const buffer2 = await loadbuffer("garage.wav");
    mybuffers = [buffer1, buffer2];
    button.disabled = false;
  } catch (error) {
    console.error(error);
  }
}

async function loadbuffer(url) {
  const response = await fetch(url);
  if (!response.ok) {
    throw new Error("HTTP error: " + response.status);
  }
  const arraybuffer = await response.arrayBuffer();
  const decodedata = await context.decodeAudioData(arraybuffer);
  return decodedata;
}

function play(mybuffers) {
  const sourceNode = context.createBufferSource();
  sourceNode.buffer = mybuffers[0];

  const convolverNode = context.createConvolver();
```

```

convolverNode.buffer = mybuffers[1];

sourceNode.connect(convolverNode);
convolverNode.connect(context.destination);
sourceNode.start(0);
}
window.addEventListener("DOMContentLoaded", initiate);

```

This example rebuilds the previous JavaScript code to include a small loader that downloads the two audio files the effect needs (**gunshot.wav** and **garage.wav**). For this purpose, we move the downloading and decoding logic into its own **loadbuffer()** function, then call it twice, once for each file, awaiting each call before moving on to the next. Because each call is awaited before the next one starts, the two files are downloaded one after the other rather than at the same time, but for two small files, this approach keeps the code simple without any noticeable issue. Once both buffers are ready, they are stored together in the **mybuffers** array for further use, and the Play button is enabled.

The construction of the audio system in the **play()** function is similar to previous examples, except this time two nodes require audio buffers, so we assign the corresponding item of the **mybuffers** array to the **buffer** property of each node, connect them, and start playing.

Do It Yourself

Copy the code in Listing 9 into the **audio.js** file. Upload this file and both audio files (**gunshot.wav** and **garage.wav**) to your server (the files are available in our website). Open the document in your browser and click the Play button. You should hear the gunshot as if it were played in a large room.

PannerNode and 3D Sound

The recreation of real objects on a computer screen has reached a high level of perfection and realism, and because of WebGL, this amazing experience is now available on the Web. But graphics are not enough to recreate the real world; we also need sounds that change along with the position and orientation of the source. **PannerNode** is a node designed specifically to simulate these effects. This node is similar to other nodes, but the sounds must be configured considering the parameters of a 3D world. For this purpose, the object representing the node includes the following properties.

panningModel

This property specifies the algorithm to use to position the audio in the 3D scene. It takes a string. The possible values are "equalpower" and "HRTF".

distanceModel

This property specifies the algorithm to use to reduce the volume of the sound according to the distance between the source and the listener. It takes a string. The possible values are "linear", "inverse", and "exponential".

refDistance

This property specifies a reference value to calculate the distance between the source and the listener. It may be useful to adapt the node to the scale of our 3D scene. The default value is **1**.

maxDistance

This property specifies the maximum distance between the source of the sound and the listener. After this limit, the sound will keep its current values.

rolloffFactor

This property specifies the ratio in which the volume is reduced.

coneInnerAngle

This property specifies the angle for directional audio sources. Inside this angle, the volume is not reduced.

coneOuterAngle

This property specifies the angle for directional audio sources. Outside this angle, the volume is reduced to a value determined by the **coneOuterGain** property.

The **PannerNode** also includes the **positionX**, **positionY**, and **positionZ** properties to specify the position of the audio source in 3D space, and the **orientationX**, **orientationY**, and **orientationZ** properties to specify the direction the audio source is facing. Combined with **coneInnerAngle**, **coneOuterAngle**, and **coneOuterGain**, the orientation properties can determine how the sound's volume changes depending on which way it's pointing relative to the listener.

Because in a 3D scene there is only one listener, its characteristics are defined by the audio context and not by a particular node. The object representing the listener is accessible through the **listener** property of the context and also includes properties for configuration. We have the **positionX**, **positionY**, and **positionZ** properties to specify the position of the listener in 3D space, the **forwardX**, **forwardY**, and **forwardZ** properties to specify the direction the listener is facing, and the **upX**, **upY**, and **upZ** properties to specify the direction pointing to the top of the listener's head (used together with the forward vector to fully determine the listener's orientation).

To show how 3D sound can be implemented, we are going to create a 3D application using the Three.js library (for more information on this library, read our book *HTML5 for Masterminds*). The document we have been using so far in this guide has to be modified to provide the import map so the browser knows where to get the library, load the **audio.js** file with the **type="module"** attribute to be able to import it, and include a **<canvas>** element to display the graphics, as shown below.

Listing 10: Creating a document to test 3D sound

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Web Audio API</title>
  <script type="importmap">
    {
      "imports": {
```

```

    "three": "https://cdn.jsdelivr.net/npm/three@0.185.1/build/
three.module.js"
  }
}
</script>
<script type="module" src="audio.js"></script>
</head>
<body>
  <section>
    <canvas id="canvas" width="500" height="400"></canvas>
  </section>
</body>
</html>

```

The code for this example draws a cube on the screen that we can move backward and forward using the mouse wheel.

Listing 11: Calculating the position of the sound in a 3D scene

```

import * as THREE from 'three';

let context, pannerNode, renderer, scene, camera, mesh, mybuffer,
playing;
async function initiate() {
  const canvas = document.getElementById("canvas");
  canvas.addEventListener("wheel", move);
  canvas.addEventListener("click", function() {
    if (!playing && mybuffer) {
      play(mybuffer);
      playing = true;
    }
  });
  const width = canvas.width;
  const height = canvas.height;
  renderer = new THREE.WebGLRenderer({canvas: canvas});
  renderer.setClearColor(0xFFFFFF);
  scene = new THREE.Scene();
  camera = new THREE.PerspectiveCamera(45, width/height, 0.1, 10000);
  camera.position.set(0, 0, 150);

  const geometry = new THREE.BoxGeometry(50, 50, 50);
  const material = new THREE.MeshPhongMaterial({color: 0xCCCCFF});
  mesh = new THREE.Mesh(geometry, material);
  mesh.rotation.y = 0.5;
  mesh.rotation.x = 0.5;
  scene.add(mesh);

  const light = new THREE.SpotLight(0xFFFFFF, 3);
  light.decay = 0;
  light.position.set(0, 100, 250);
  scene.add(light);
  renderer.render(scene, camera);
}

```

```

context = new AudioContext();
context.listener.positionX.value = 0;
context.listener.positionY.value = 0;
context.listener.positionZ.value = 150;

try {
  const response = await fetch("engine.wav");
  if (!response.ok) {
    throw new Error("HTTP error: " + response.status);
  }
  const arraybuffer = await response.arrayBuffer();
  mybuffer = await context.decodeAudioData(arraybuffer);
} catch (error) {
  console.error(error);
}
}

function play(mybuffer) {
  const sourceNode = context.createBufferSource();
  sourceNode.buffer = mybuffer;
  sourceNode.loop = true;
  pannerNode = context.createPanner();
pannerNode.refDistance = 100;
sourceNode.connect(pannerNode);
pannerNode.connect(context.destination);
  sourceNode.start(0);
}

function move(event) {
  if (playing) {
    mesh.position.z = Math.min(mesh.position.z - event.deltaY / 2,
70);
    pannerNode.positionX.value = mesh.position.x;
pannerNode.positionY.value = mesh.position.y;
pannerNode.positionZ.value = mesh.position.z;
    renderer.render(scene, camera);
  }
}

window.addEventListener("DOMContentLoaded", initiate);

```

The **initiate()** function in Listing 11 initializes all the elements of the 3D scene and downloads the **engine.wav** file with the sound for our cube. Notice that after creating the audio context with the **AudioContext()** constructor, the position of the listener is set using its **positionX**, **positionY**, and **positionZ** properties. This is done in the initialization process because in this example the listener keeps the same position the whole time, but this will probably have to be changed for other applications.

Because browsers only allow audio to start playing after a genuine user gesture, the sound cannot simply start on its own when the page loads. To work around this limitation, a listener for the **click** event is added to the **<canvas>** element at the very beginning of **initiate()** to call the **play()** function to start the process. (The **playing** variable keeps track of whether the sound has started, so later clicks are ignored.)

To control the position of the cube, we also add a listener to the **<canvas>** element for the **wheel** event, so every rotation of the mouse wheel (or drag of the fingers in the trackpad) triggers the **move()** function. In this function, we update the position of the cube on the **z** axis according to the direction in which the wheel was moved. Note that we cap it at **70** so the cube cannot move past that point toward the camera. Once this value is ready, we set the new coordinates for the audio source using the panner's **positionX**, **positionY**, and **positionZ** properties to make the cube feel like the real source of the sound.

Do It Yourself

Update your HTML file with the code in Listing 10 and the **audio.js** file with the code in Listing 11. Download the **engine.wav** file from our website and move it to your project. Upload all the files to your server. Open the document in your browser, click the cube to start the sound, and rotate the wheel on your mouse (or drag two fingers on the trackpad) to move it. You should hear the sound changing as the cube gets farther or closer to you.

AnalyserNode

It would be hard to use all the tools provided by the Web Audio API in a professional environment if we were not able to visualize the results on the screen. To do this, the API includes the **AnalyserNode**. This node implements an algorithm called FFT (fast Fourier transform) to turn the waveform of the audio signal into an array of values representing magnitude versus frequency for a given period of time. The values returned may be used to analyze the signal or create charts to show its values on the screen. The **AnalyserNode** provides the following properties and methods for retrieving and processing the information.

fftSize

This property specifies the size of the FFT (the size of the block of data to be analyzed). The value must be a power of 2 (e.g., 128, 256, 512, 1024).

frequencyBinCount

This property returns the number of frequency values provided by the FFT.

minDecibels

This property specifies the minimum value of decibels for the FFT.

maxDecibels

This property specifies the maximum value of decibels for the FFT.

smoothingTimeConstant

This property specifies the averaging constant used to smooth the transition between one analysis and the next. It takes a value from **0** to **1** (0 disables the averaging entirely, and values closer to 1 blend in more of the previous result, producing smoother but slower-responding data).

getFloatFrequencyData(array)

This method retrieves the current frequency data from the audio signal and stores it in the **array** argument as floating-point values. The argument is a reference to a floating-point array already created.

getBytesFrequencyData(array)

This method retrieves the current frequency data from the audio signal and stores it in the **array** argument as unsigned byte values. The argument is a reference to an unsigned byte array already created.

getFloatTimeDomainData(array)

This method retrieves the current waveform data from the audio signal and stores it in the **array** argument as floating-point values. The argument is a reference to a floating-point array already created.

getBytesTimeDomainData(array)

This method retrieves the current waveform data from the audio signal and stores it in the **array** argument as unsigned byte values. The argument is a reference to an unsigned byte array already created.

The data produced by these properties and methods may be used to update a chart on the screen and show the evolution of the audio signal. To see how to implement this kind of application, we are going to use a **<canvas>** element and draw the sound coming from a **<video>** element. The following is the document we need for this example.

Listing 12: Creating a document to experiment with the AnalyserNode

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Web Audio API</title>
  <style>
    section {
      float: left;
    }
  </style>
  <script src="audio.js"></script>
</head>
<body>
  <section>
    <video id="media" width="483" height="272" controls>
      <source src="trailer.mp4" type="video/mp4">
      <source src="trailer.webm" type="video/webm">
    </video>
  </section>
  <section>
    <canvas id="canvas" width="500" height="272"></canvas>
  </section>
</body>
</html>
```

As we explained before, when the source is a media element, the audio source node has to be created with the **createMediaElementSource()** method. In the following code, we apply this and the **createAnalyser()** method to get the nodes we need for our audio system.

Listing 13: Drawing the chart on a <canvas> element

```
let canvasContext, context, analyserNode;
function initiate() {
  const video = document.getElementById("media");
  const canvas = document.getElementById("canvas");
  canvasContext = canvas.getContext("2d");
  context = new AudioContext();

  video.addEventListener("play", function() {
    if (context.state === "suspended") {
      context.resume();
    }
  });
  const sourceNode = context.createMediaElementSource(video);
  analyserNode = context.createAnalyser();
  analyserNode.fftSize = 512;
  analyserNode.smoothingTimeConstant = 0.9;

  sourceNode.connect(analyserNode);
  analyserNode.connect(context.destination);

  showgraphic();
}
function showgraphic() {
  const data = new Uint8Array(analyserNode.frequencyBinCount);
  analyserNode.getBytesFrequencyData(data);

  canvasContext.clearRect(0, 0, 500, 272);
  for (let f = 0; f < analyserNode.frequencyBinCount; f++) {
    canvasContext.fillRect(f * 5, 272 - data[f], 3, data[f]);
  }
  requestAnimationFrame(showgraphic);
}
window.addEventListener("DOMContentLoaded", initiate);
```

In this example, the size of the FFT is set to **512**, which determines the amount of frequency data available, and the **smoothingTimeConstant** property is set to **0.9** to smooth the chart's transitions from one frame to the next. After the nodes are created, configured, and connected, the audio system is ready to provide information from the audio signal. The **showgraphic()** function is in charge of processing this data. This function creates an empty **Uint8Array** with a size determined by the **frequencyBinCount** property, and then calls the **getBytesFrequencyData()** method to populate it with values from the audio signal. Because of the type of array, the values stored will be integers from 0 to 255 (8-bit integers). Next, in the **for** loop, we use these values to calculate the size of the bars representing each frequency, and draw them on the canvas. The result is shown below.

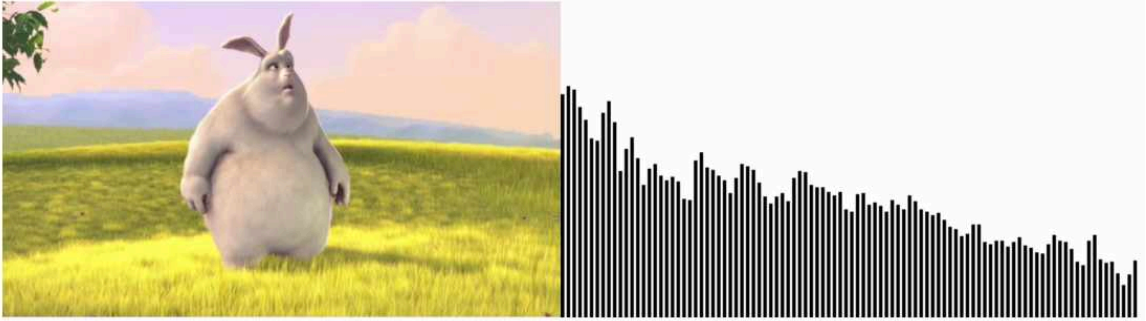


Figure 6: Audio chart for the video created with the `AnalyserNode`

Do It Yourself

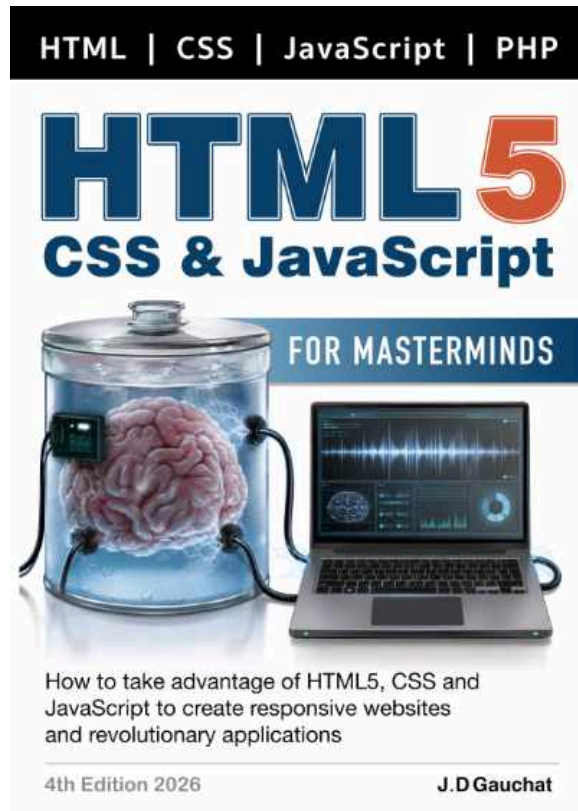
Update your HTML file with the code in Listing 12 and the **audio.js** file with the code in Listing 13. Download the **trailer.mp4** and **trailer.webm** videos from our website. Open the document in your browser and play the video. You will see something similar to Figure 6.

IMPORTANT

JavaScript variables are usually defined without taking into consideration the type of data they are going to work with, but some methods take only values provided in a specific type. The **`Uint8Array()`** constructor, used in Listing 13, is just one of the constructors introduced by the language to provide new array types to satisfy the requirements of some JavaScript APIs.

Where To Go From Here

The Web Audio API is one of many APIs that browsers provide to add features to our websites and create fully functional web applications. For more information on how these APIs work and everything you need to develop your own applications, read our book **HTML5 for Masterminds**. If you are also interested in creating native apps, we have a series of books that teach you how to develop apps for Apple devices and how to get into this lucrative and fascinating market, including **Swift for Masterminds** and **SwiftUI for Masterminds**. And if you are interested in learning how artificial intelligence works and how to incorporate it into your apps, read our book **Apple Intelligence for Masterminds**.



HTML5 for Masterminds

How to take advantage of HTML5, CSS, and JavaScript to create responsive websites and revolutionary applications

Learning web development can feel overwhelming. There are too many languages, too many tools, and too many tutorials that assume you already know the basics. HTML5 for Masterminds takes you by the hand through every feature and every tool, until you can build interactive websites and web applications on your own.

iOS 27 | Swift 6.4 | Xcode 27

SwiftUI

Declarative Framework



How to take advantage of SwiftUI
and AI Agents to develop insanely great
applications for iPhones, iPads, and Macs

7th Edition

J.D Gauchat

SwiftUI for Masterminds

How to take advantage of SwiftUI and AI Agents to develop
insanely great apps for iPhones, iPads, and Macs

SwiftUI for Masterminds is a comprehensive course designed to take you from the fundamentals of SwiftUI to developing professional applications for Apple devices. Learn Apple development step by step, from simple examples to fully functional apps. By the end of this book, you will know how to design user interfaces for iPhones, iPads, and Macs, store and manage data, create animations, access the camera, capture and store images, display interactive maps, work with web content, share information across multiple devices, and accelerate your workflow with the help of AI Agents.

iOS 27 | Swift 6.4 | Xcode 27

APPLE Intelligence



How to create intelligent applications
for iPhones, iPads and Macs

2nd Edition

J.D Gauchat

Apple Intelligence for Masterminds

How to take advantage of Apple Intelligence to create
innovative and powerful apps for iPhones, iPads and Macs

Discover how AI models work and harness the power of Apple Intelligence to create chatbots, translate text, generate images, transcribe spoken audio, recognize text in documents, enhance user writing, process images, identify human poses and gestures, access external models from top companies like OpenAI, and use coding assistants and agents to take your applications to the next level.

www.formasterminds.com