Introduction to the Web Audio API

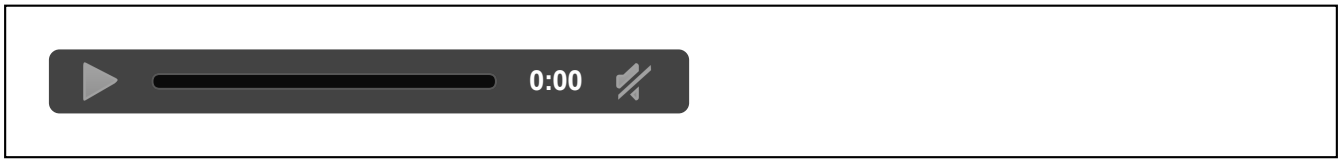
SHORTCOMINGS OF THE STANDARD APIS THAT WE HAVE DISCUSSED SO FAR...

In week 2 of the HTML 5 Part 1 course, you learned how to add an audio or video player to an HTML document, using the `<audio>` and `<video>` elements.

For example:

```
<audiosrc="http://mainline.i3s.unice.fr/mooc/LaSueur.mp3" con
trols>
```

Would render like this in your document:



What this HTML code does under the hood:

1. initiates a network request to stream the content,
2. deals with decoding/streaming/buffering the incoming data,
3. renders audio controls,
4. updates the progress indicator, time, etc.

You also learnt that it's possible to write a custom player: to make your own controls and use the JavaScript API of the `<audio>` and `<video>` elements; to call `play()` `and pause();` to read/write properties such as `currentTime;` to listen to events (`ended`, `error`, `timeupdate`, etc.); and to manage a playlist, etc.

However, there are many things we still cannot do, including:

- Play multiple sounds or music in perfect sync,
- Play non-streamed sounds (this is a requirement for games: sounds must be loaded in memory),
- Output goes straight to the speakers; no special effects, equalizer, stereo balancing, reverb, etc.
- No fancy visualizations that dance with the music (e.g. waveforms and frequencies).

The Web Audio API will provide the missing parts, and much more.

In this course we will not cover the whole Web Audio API specification. Instead, we focus on the parts of the API that can be useful for writing enhanced multimedia players (that work with streamed audio or video), and on parts that will be useful for games (i.e. parts that work with small sound samples loaded in memory). There is a whole part of the API that specializes in music synthesis and scheduling notes, that will not be covered here.

Screenshot from one example you will study: an audio player with animated waveform and volume meters that dance with the music:

WEB AUDIO CONCEPTS

The audio context

The canvas used a graphic context for drawing shapes and handling properties such as colors and line widths.

The Web Audio API takes a similar approach, using an AudioContext for all its operations.

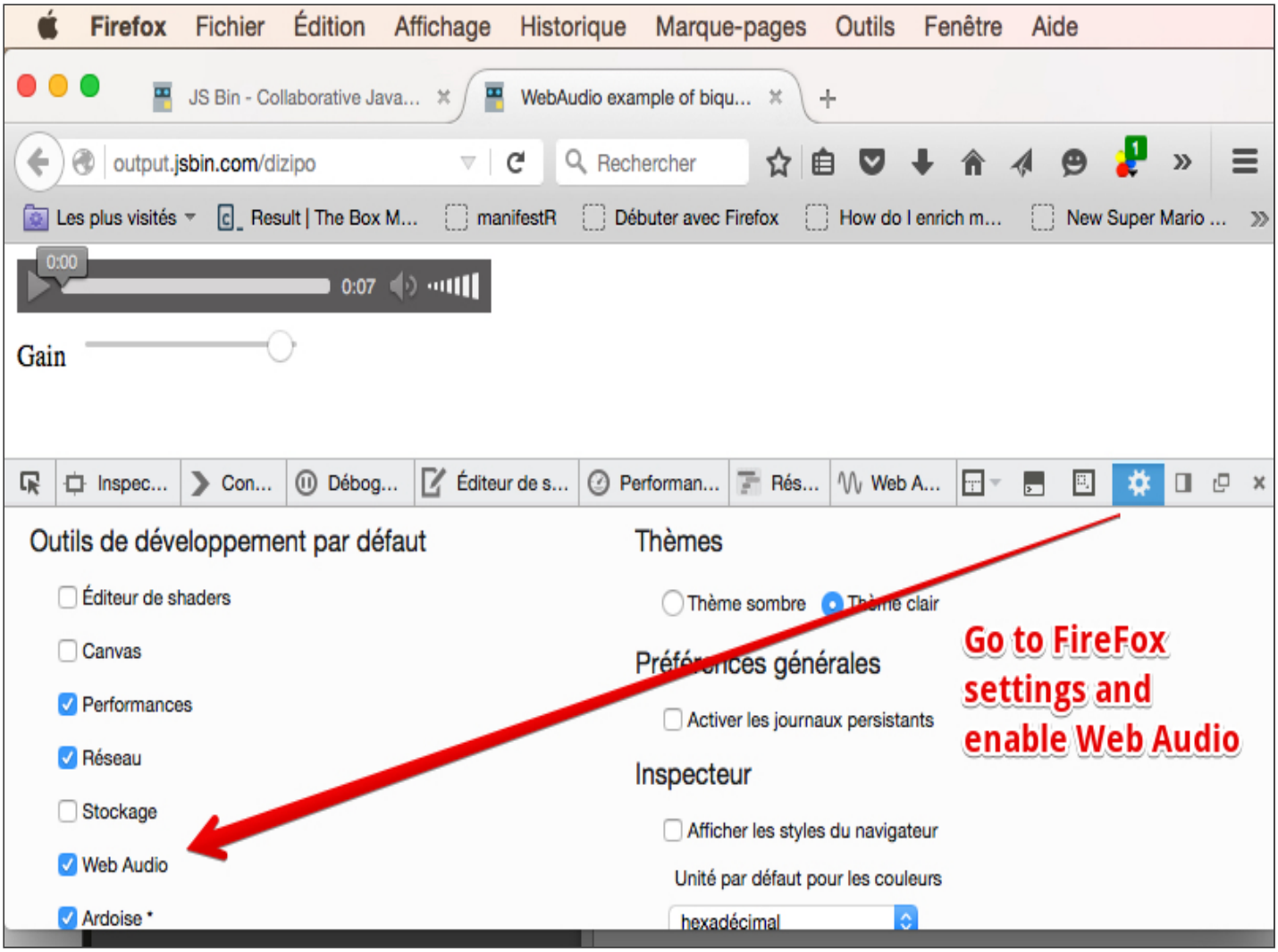
Using this context, the first thing we'll do when using this API is to build an "audio graph" made of "audio nodes" which are linked together. Some node types are for "audio sources", another built-in node is for the speakers, and many other types exist, that correspond to audio effects (delay, reverb, filter, stereo panner, etc.), audio analysis (useful for creating fancy visualizations of the real time signal). Others, which are specialized for music synthesis, will not be covered in this course.

The AudioContext also exposes various properties, such as`sampleRate`, `currentTime` (in seconds, from the start of`AudioContext` creation), `destination`, and the methods for creating all of the various audio nodes.

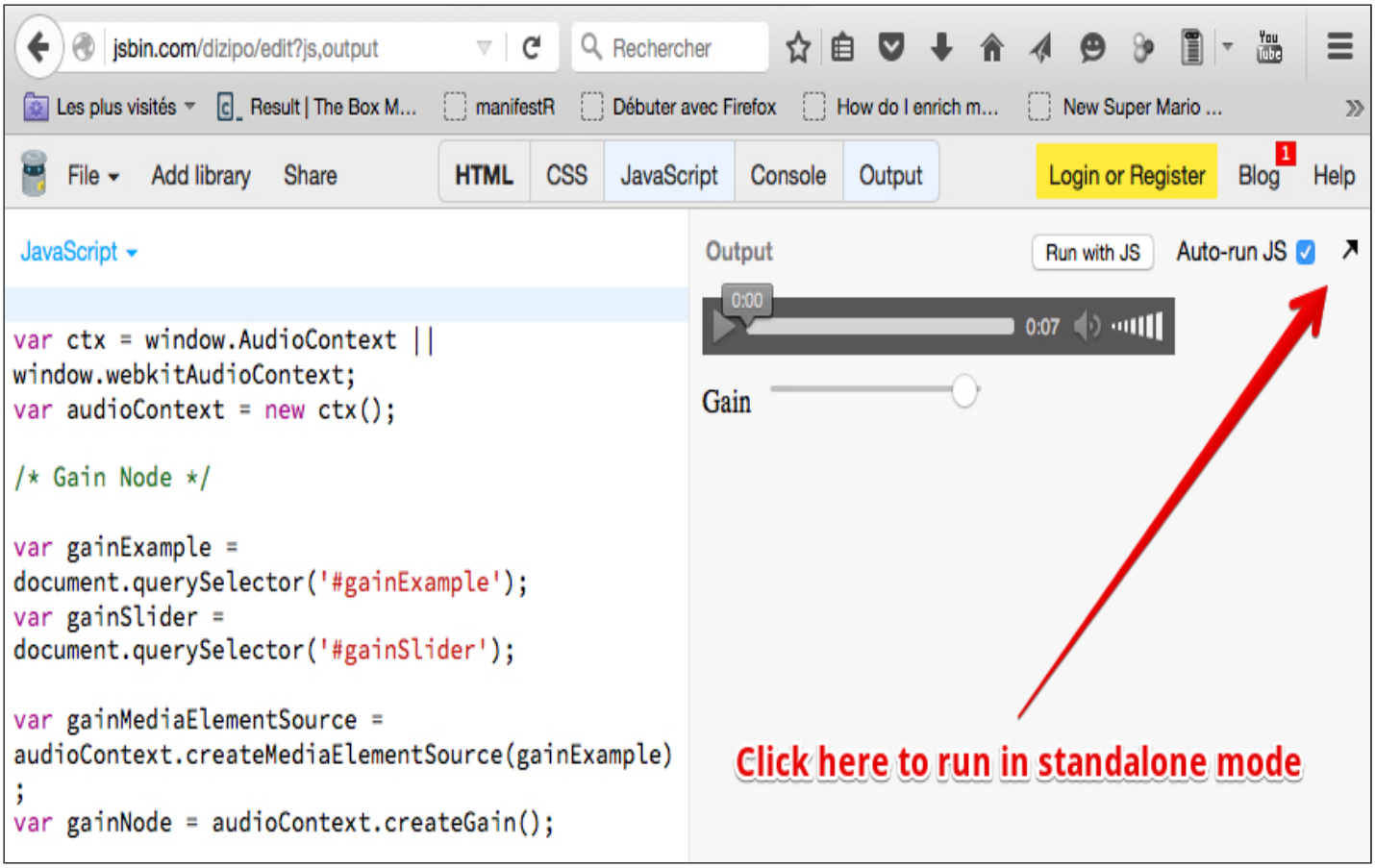
The easiest way to understand this principle is to look at a first example at JSBin.

This example will be detailed in the next lesson. For the moment, all you need to know is that it routes the signal from an `<audio>` element using a special node that bridges the "streamed audio" world to the Web Audio World, called a`MediaElementSourceNode`, then this node is connected to a`GainNode` that will enable volume control. This node is then connected to the speakers. We can look at the audio graph of this application using a recent version of FireFox. This browser is the only one (as at November 2015) to provide a view of the audio graph, which is very useful for debugging:

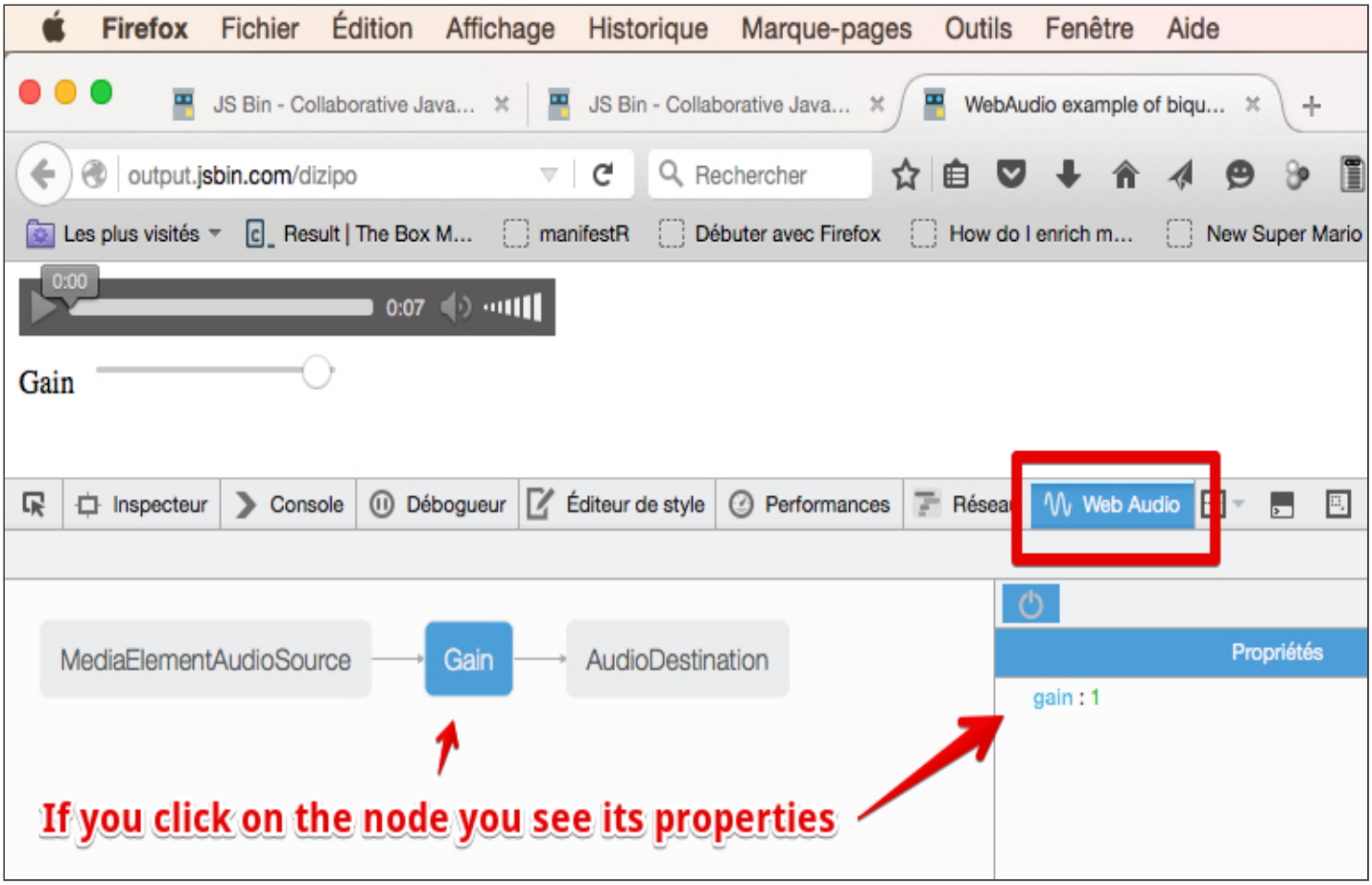
Step 1 - enable Web Audio debug in the devtools



Step 2 : Open the JSBin example in standalone mode (not in editor mode)



Step 3 - open the devtools and go to the Web Audio tab, reload the page if needed



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 just want to visualize some audio data, for example).

The `AudioDestination` node above corresponds to the speakers. In this example the signal goes from left to right: from the `MediaElementSourceNode` (we will see in the code that it's the audio stream from an `<audio>` element), to a `Gain`node (and by adjusting the `gain` property we can set the volume of the sound that outputs from this node), then to the speakers.

Typical code to build an audio graph (the one used in the above example)

HTML code extract:

```
<audiosrc="http://mainline.i3s.unice.fr/mooc/drums.mp3"
```

```
        id="gainExample"
        controls loop
        crossorigin="anonymous">
</audio>
<br>
<label for="gainSlider">Gain</label>
<input type="range" min="0" max="1"step="0.01" value="1" id="
gainSlider" />
```

JavaScript source code:

```
// This line is a trick to initialize the AudioContext
// that will work on all recent browsers
var ctx = window.AudioContext ||window.webkitAudioContext;
var audioContext;
var gainExemple, gainSlider, gainNode;

window.onload = function() {
  // get the AudioContext
  audioContext = new ctx();

  // the audio element
  gainExample =document.querySelector('#gainExample');
  gainSlider =document.querySelector('#gainSlider');

  buildAudioGraph();
  // input listener on the gain slider
  gainSlider.oninput = function(evt){
    gainNode.gain.value =evt.target.value;
  };
};

function buildAudioGraph() {
  // create source and gain node
  var gainMediaElementSource =audioContext.createMediaElement
Source(gainExample);
  gainNode = audioContext.createGain();
  // connect nodes together
```

```
31.     gainMediaElementSource.connect(gainNode);
        gainNode.connect(audioContext.destination);
    }
```

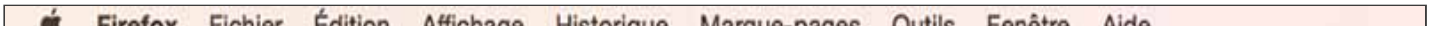
Explanations:

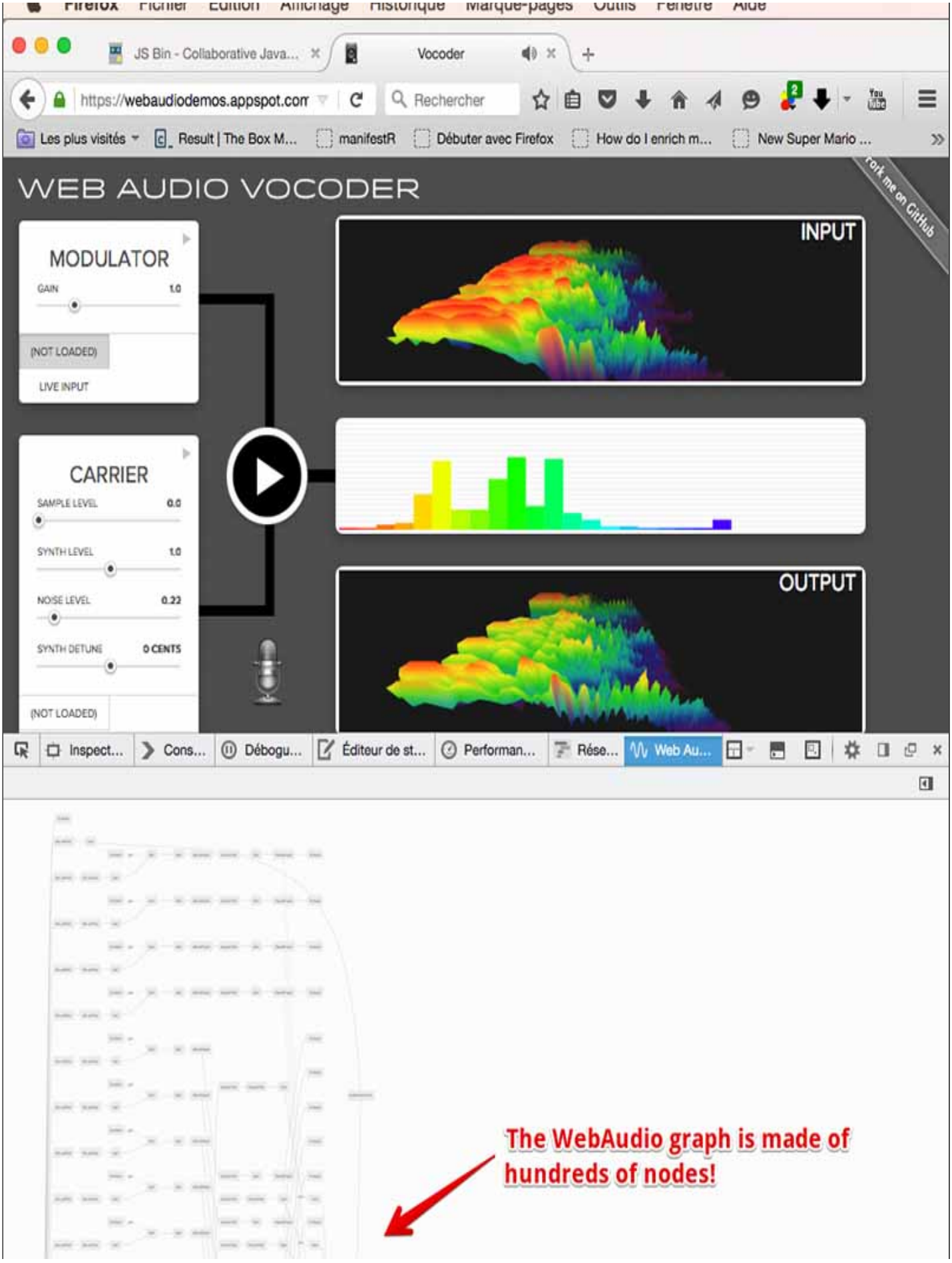
Here we applied a commonly used technique:

- As soon as the page is loaded: initialize the audio context (*line 11*). Here we use a trick so that the code works on all browsers: Chrome, FF, Opera, Safari, Edge. The trick at *line 3* is required for Safari, as it still needs the WebKit prefixed version of the AudioContext constructor.
- Then we build a graph (*line 17*).
- The build graph function first builds the nodes, then connects them to build the audio graph. Notice the use of`audioContext.destination` for the speakers (*line 32*). This is a built-in node.

EXAMPLE OF BIGGER GRAPHS

Web Audio nodes are implemented natively in the browser. The Web Audio framework has been designed to handle a very large number of nodes. It's common to encounter applications with several dozens of nodes: some, such as this vocoder app, use hundreds of nodes:




Firefox Fichier Édition Affichage Historique Marque-pages Outils Fenêtre Aide
JS Bin - Collaborative Java...
Vocoder
https://webaudiodemos.appspot.com/
Rechercher
Les plus visités
Result | The Box M...
manifestR
Débuter avec Firefox
How do I enrich m...
New Super Mario ...
WEB AUDIO VOCODER
Fork me on GitHub
MODULATOR
GAIN 1.0
(NOT LOADED)
LIVE INPUT
CARRIER
SAMPLE LEVEL 0.0
SYNTH LEVEL 1.0
NOISE LEVEL 0.22
SYNTH DETUNE 0 CENTS
(NOT LOADED)
INPUT
OUTPUT
Inspect...
Cons...
Débogu...
Éditeur de st...
Performan...
Rése...
Web Au...
The WebAudio graph is made of hundreds of nodes!

KNOWLEDGE CHECK 1.5.1

The WebAudio API has a modular approach: you build an audio graph made of diffent nodes connected together, each node being a source or corresponding to a sound effect. The signal follows all the routes in this graph, from the sources to the final destination (which is usually the speakers). Can WebAudio handle big graphs made of hundred of nodes?

- ○ It depends on the power of the device, but on modern computers or mobile devices it should be able to handle hundreds of nodes. It has been designed with support for large graphs in mind.
- ○ No, you shouldn't create audio graphs with more than a few nodes.