Towards an open Web Audio plug-in standard

Michel Buffa, Jerome Lebrun, Jari Kleimola, Oliver Larkin, Stephane Letz

Towards an open Web Audio plug-in standard

Michel Buffa, Jérôme Lebrun
Université Côte d'Azur
CNRS, INRIA
(buffa, lebrun)@i3s.unice.fr

Jari Kleimola, Oliver Larkin
webaudiomodules.org
(jari, oli)@webaudiomodules.org

Stéphane Letz
GRAME
letz@grame.fr

ABSTRACT

Web Audio is a recent W3C API that brings the world of computer music applications to the browser. Although developers have been actively using it since the first beta implementations in 2012, the number of web apps built using Web Audio API cannot yet compare to the number of commercial and open source audio software tools available on native platforms. Many of the sites using this new technology are of an experimental nature or are very limited in their scope. While JavaScript and Web standards are increasingly flexible and powerful, C and C++ are the languages most often used for real-time audio applications and domain specific languages such as FAUST facilitate rapid development with high performance. Our work aims to create a continuum between native and browser based audio app development and to appeal to programmers from both worlds. This paper presents our proposal including guidelines and proof of concept implementations for an open Web Audio plug-in standard - essentially the infrastructure to support high level audio plug-ins for the browser.

CCS CONCEPTS

• **Software and its engineering → Software organization and properties → Software system structures** → Abstraction, modeling and modularity

KEYWORDS

WebAudio, Audio Effects and Instruments, Plug-in Architecture, Web Standards

1 INTRODUCTION

This paper will introduce work that was originally being done separately by three groups of researchers who had different initial interests. Due to the clear similarities in the work, the authors decided to join forces to develop interoperable Web Audio plug-ins and plug-in hosts, and to synchronize their efforts towards the beginnings of an open standard. One group has been developing the FAUST domain specific language (DSL) since 2002 [7]. FAUST is a DSL for digital signal processing with a focus on audio applications. Hundreds of musical instruments and audio effects are available written using the FAUST language, and the toolchain can compile them to different targets, including Web Audio.

Another group has been developing the Web Audio Module (WAM) API since 2014 [4]. WAMs are high level audio plug-ins for the Web browser and have a C++ API, like native plug-in formats. At the time of writing there are six WAM synths available that have been ported from native code. The last group of researchers have been developing their own Web Audio plug-ins since 2012 in the context of a web-based "pedal board" for guitarists. Each virtual pedal as well as the amp simulator is effectively a plug-in, and the pedal board as a whole can be considered a plug-in host. Authors from this group are also W3C Advisory Committee representatives who participate to the W3C Web Audio working group.

Despite differences in implementation in the existing work, all three groups have had to deal with similar issues regarding the specification of a high-level audio processing "plug-in" for the browser, and so decided to work together. A publicly available demo which showcases the work is actively being developed as a proof of concept[1]. This paper addresses the different aspects of what the definition of an open standard for Web Audio plug-ins should be, including the APIs, hosting and approaches to packaging and distribution.

2 BACKGROUND CONTEXT AND TERMS

2.1 The Web Audio API and the lack of an open plug-in standard

The W3C Web Audio API[2] is now a recommendation candidate. It proposes a set of unit generators called AudioNodes for graph-based realizations of audio algorithms and it is supported in the latest versions of most popular desktop browsers, with some support on mobile. The connection of these nodes in the browser via the JavaScript API allows for a range of different applications involving real time audio processing. Music apps are not the only ones that require complex audio backends and the API is designed to support the needs of games and other use cases. The API comes with a limited set of standard nodes for common operations such as volume control, audio filtering, time delay/echo, reverberation, dynamics processing, spatialization, etc. The recent addition of the AudioWorklet node provides a solution for implementing custom low level audio processing. The AudioWorklet is the last inclusion in the Web Audio API version 1 and replaces the deprecated ScriptProcessorNode (SPN), which offered similar functionality but was unsuitable for real-time audio since audio processing was performed in the main thread

WWW '18 Companion April 23-27, 2018, Lyon, France.
 ACM ISBN 978-1-4503-5640-4/18/04. DOI: https://doi.org/10.1145/3184558.3188737

[1] The pedal board host that works with the plug-ins made by the three different groups can be tried online at the following url: https://wasabi.i3s.unice.fr/pedalboard. A video is available here: https://youtu.be/elbjh6tBK6U

[2] https://www.w3.org/TR/webaudio/

(not the high priority audio thread used by the rest of the API).
The Web Audio API's nodes can be assembled into an "audio graph", which developers can use to write more complex audio effects or instruments. Examples of audio effects that could be built in this way might include: tape echo (DelayNode, BiquadFilterNode, GainNode, feedback loop), auto-wah (BiquadFilterNode, OscillatorNode), chorus (multiple DelayNodes and OscillatorNodes for modulation), a distortion (GainNode, WaveShaperNode) etc. Instruments can also be written, either playing back recorded PCM samples (e.g. using AudioBuffer and AudioBufferSourceNode) or direct synthesis of electronic sound using a variety of techniques such as subtractive synthesis (OscillatorNode, BiquadFilterNode, GainNode).
Sometimes, when performing live, you need to chain audio effects together (for example in the guitarist's pedal board) and when composing/producing music multiple effects and instruments are often used. These are use cases where the Web Audio API nodes are too low level, hence the need for a higher-level unit in order to represent the equivalent of a native audio "plug-in". Such a high-level "audio plug-in" standard does not exist yet for Web Audio, but has been discussed for a future version of the API.
In the case of instrument plug-ins or when remote controls are needed the Web MIDI API[3] complements the Web Audio API by offering access to local MIDI devices for control of pitch (via MIDI note messages), and other parameters (via MIDI continuous controller messages).

2.2 State of the art: native plug-in standards

Over the last 20 years there has been a big shift in the technology used in music production and performance. One of the most significant changes is a that there are now far less hardware devices, and many entirely software based studios where a mixing console, effects units and tape machine all have virtual equivalents. One of the landmark moments for this technology was the introduction of VST (Virtual Studio Technology), introduced by Steinberg in 1996. This is a cross-platform native plug-in format, where plug-ins are dynamic libraries that load into a host application which would typically be a Digital Audio Workstation (DAW). A publicly available C++ SDK allowed third parties to develop audio instruments and effects that plug-in to other software. This has become one of the primary means of musicians obtaining new sounds and there are many companies that focus on the development of the software instruments and effects, as well as a lively hobbyist community[4]. Several other manufacturers of DAWs and operating systems developed their own APIs in order to have a better influence on the user experience of their platform (Apple's AudioUnits, Avid's AAX etc.). In addition, the open source community created their own formats[5]. These APIs all share common functionality, namely processing blocks of audio samples, handling parameter changes, handling MIDI and handling state. For this reason, many third-party developers opt to use an intermediate C++ framework such as JUCE[6] or iPlug[7], which allow a single code base to target multiple APIs, saving the developer a lot of time.

2.3 What makes the Web platform different?

Although we aim to introduce the functionality offered by native audio plug-ins to the Web, the differences of the environment require a different approach to API design, and since this is an entirely new platform it provides an opportunity to improve upon some aspects of existing APIs. The web platform also brings with it some new possibilities and unique challenges, when developing real-time audio software:

A**dvantages:** The Web facilitates ease of distribution (no installation required), easy maintenance (resources at URLs can be silently updated), new kinds of collaboration (using technologies such as WebSockets and WebRTC for example), platform/architecture independence, security (the browser has sandboxing that can prevent misbehaving plug-ins from affecting the host or file system). The ability to inspect JavaScript code at runtime can also be useful.

Disadvantages: efficiency (JavaScript is usually slower than native code, with issues that affect real-time audio performance such as a garbage collector), latency (web audio applications do not have access to low-level audio drivers in order to provide the lowest latency possible, with significant differences in latencies across platforms), sandboxing (access to native resources, local hard disk access is forbidden or limited which can be problematic depending on the use case).

Audio plug-ins contain as an "audio processor" part (implementing real time DSP code), and an optional user interface part. Somewhere there needs to be functionality to register the parameters and possibly deal with factory presets and external control. An API is needed for these parts, but also an API or at least guidelines on how to package it for publication on a plug-in repository/server and for its integration in a host. A web based API should be "Web aware" and use URLs as plug-ins identifiers. Plug-ins are just another kind of Web resource, like images, style-sheets or JavaScript files and should be referenced by a relative/local or remote URL. Host web apps should be able to discover local/remote plug-ins by querying plug-in folders/servers. Remote plug-ins should be usable without the need to download them manually, and the mixture of different JavaScript libraries and frameworks, should not raise any naming conflict or dependency problems.

3 CURRENT STATE OF OUR THREE PROJECTS

This section presents the work conducted separately by the

[3] http://www.w3.org/TR/webmidi/
[4] http://www.kvraudio.com/
[5] http://lv2plug.in/gmpi.html
[6] http://www.juce.com/
[7] https://github.com/olilarkin/wdl-ol

authors of this paper towards similar goals. We should also mention other researchers' initiatives such as the Web Audio API extension framework[8] (WAAX) [9] that abstracts the Web Audio API node graphs as units, which may be parameterized and interconnected with other Web Audio API nodes, also the work by Jillings et Al. [3] who also proposed a host/plug-in architecture for Web Audio, but that does not address some of our requirements (full encapsulation, remote plug-in support -using URLs etc.).

3.1 Pedalboard plug-in host and pedals

For the WASABI project [6], the WIMMICS team from INRIA/I3S/CNRS developed the first online digital emulation of a tube guitar amplifier: the Marshall JCM 800, a popular amp used by many classic rock artists (AC/DC, Led Zeppelin, Guns and Roses etc.) [1, 2]

Figure 2: A real pedal board used by guitarists. Pedals are chained together before going to the amplifier.

In addition to the sound modification from the amplifier, guitarists often use effects processing. This usually comes in the form of guitar "pedals" (Figure 2), that are plugged between the electric guitar and the amplifier, or connected via the auxiliary "FX loop" of the amplifier. Very often we call a set of pedals a "pedal board". A virtual pedal board web application was thus developed as a host, along with the most common audio effect pedals as audio plug-ins. The amp simulation was also turned into a plug-in (Figure 3) for flexibility (e.g. having multiple amplifiers). The main input of the pedal board is either the signal coming from the guitar (via the computer's analogue to digital converter) or from a sound file, the main output goes to the computer's audio output. In the WASABI pedal board[9], you can drag and drop pedals (or the amp simulator), connect them, change the settings using knobs or sliders, and control the parameters using a MIDI controller device if your browser supports Web MIDI.

In our architecture, each plug-in is a Web Component (*Web Components is a suite of different technologies allowing you to create reusable custom user interface components — with their functionality encapsulated away from the rest of your code — and utilize them in your web apps*[10]). This file embeds the "audio processor" JavaScript and may also define a GUI in its HTML <template> section. The audio processor code inherits an "EffectTemplate" JavaScript class that conforms to an "audio processor" API we defined, that is similar to the interface[11] implemented by all standard Web Audio nodes.

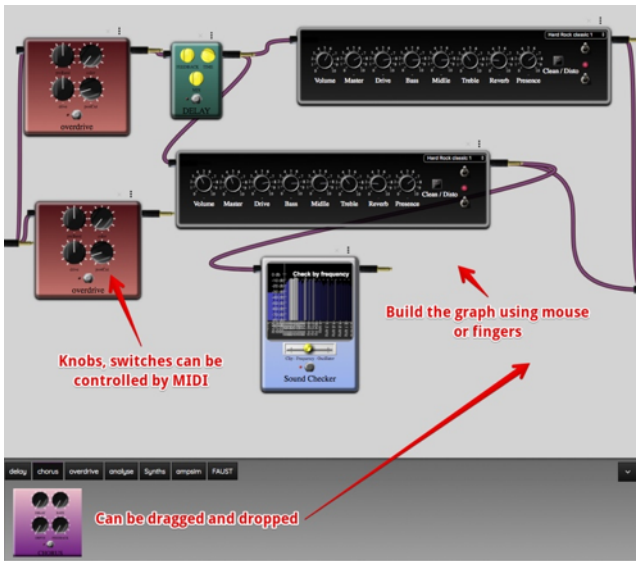

Figure 3: Virtual pedal board. Effects pedals and two instances of the guitar amp simulation form a graph of plug-ins. The signal goes from left to right.

Some methods and properties have been added to this interface for describing the plug-in, for loading/saving its state, and for dealing with its lifecycle (in case it needs to fetch remote resources, for example). The plug-ins we have developed can be assigned to a MIDI device so that they can be modified remotely by any MIDI controller. A "redistributable plug-in", packaged like this, can be published on a repository and easily reused: one just must import the HTML file that defines it as a Web Component.

For importing and making the above plug-ins moveable with the mouse in the pedal board host (Figure 3) we wrap them into another Web Component that inherits the default behavior that each "plug-in in the pedal board" shares (wires can be connected from outputs to inputs, plug-ins can be moved using the mouse, position and settings can be saved/restored etc.).

The parent class that defines this "host dependent behavior" is

[8] WAAX is no more maintained.
[9] See footnote 1 for URL of the demo, also URL of a demonstration video.
[10] https://developer.mozilla.org/en-US/docs/Web/Web_Components
[11] https://dvcs.w3.org/hg/audio/raw-file/tip/webaudio/specification.html#AudioNode-section

named PBplug-in for "Pedal Board plug-in". For each plug-in, we write a subclass of PBplug-in in which we import the HTML file that defines the "redistributable plug-in" (and this can be a remote URL), and uses the HTML custom element that corresponds to the imported plug-in to generate its GUI and add some JS code to initialize it.

The code of PBplug-in and of its subclass is "host dependent" and is not candidate for a standard. However, the way we defined "redistributable plug-ins": as Web Components (for distribution, solving encapsulation problems and reusability), and separated the "audio processor part" from the "GUI/controller" part, is close to the other approaches presented in the next sections.

3.2 Faust

FAUST [7] is a functional, synchronous, domain specific programming language specifically designed for real-time signal processing and synthesis. A unique feature of Faust, compared to other existing music languages like Max/MSP, PureData, Supercollider, etc., is that programs are not interpreted, but fully compiled. FAUST provides a high-level alternative to C/C++ to implement efficient sample-level DSP algorithms.

Being a specification language, the FAUST code says nothing about the audio drivers or the GUI toolkit to be used. It is the role of the "architecture file" to describe how to relate the DSP code to the external world. Additional generic code is added to connect the DSP computation itself with audio inputs/outputs, and with control parameters, which could be buttons, sliders, numerical entries etc.; all this, in a standard user interface, or any kind of control using a remote protocol like OSC or HTTP.

The FAUST compiler is organized in successive stages, from the DSP block diagram to signals, and finally to the FIR (FAUST Imperative Representation) which is then translated to several target languages. The FIR language describes the computation performed on the samples in a generic manner. It contains primitives to read and write variables and arrays, do arithmetic operations, and it defines the necessary control structures (for and while loops, if statements for branching etc.). Several backends have been developed to translate the FIR to C, C++, Java, asm.js, WebAssembly (WASM) and LLVM IR.

Two FAUST backends have been developed to generate WebAssembly formats [8]. The WAST backend generates human-readable text based code, which is easier to test and debug. The WASM backend generates the equivalent binary format to be directly loaded and executed in browsers.

JavaScript code is used to load the WASM file into a typed array, compile it to a module with **`WebAssembly.compile`**, then instantiate it using **`WebAssembly.Instance`** function, and finally get the callable exported functions.

Monophonic DSPs (generators, processor, analyzers...) are created by wrapping a single DSP Faust object. Polyphonic MIDI controllable instruments can be defined using wrapper code that takes the Faust DSP code to be used as the description of a single voice. Several voices are then automatically created, and a MIDI API to handle incoming events, and to do voices allocation and handling is added. The DSP memory is either allocated inside the WASM module (for monophonic DSPs), or externally in the wrapping JavaScript code (to allocate several voices in the polyphonic instrument case), and is given as parameter when creating the module.

The DSP code is finally wrapped to be usable as a regular AudioNode. Two models have been successfully developed: a specialized ScriptProcessorNode and more recently a specialized AudioWorkletNode. In both cases their standard AudioNode API is extended with some additional methods:

`getNumInputs/getNumOutputs`: returns the number of audio inputs/outputs.

`getParamValue(path,value)/setParamValue(path,value)` to read and write control parameters, and getParams retails an array containing all their paths (as OSC like hierarchical paths).

`getJSON` returns a full JSON description of the DSP control hierarchy, to be used by a GUI manager to create a GUI with buttons, sliders etc.

Additional MIDI control entry points (**`keyOn/keyOff, ctrlChange, pitchWheel…`**) are also added for polyphonic instruments.

Starting from a given FAUST DSP source file, a set of scripts (faust2wasm, faust2webaudiowasm…) allow to chain successive operations: calling the FAUST compiler to produce the compiled WASM file, to generate the glue JavaScript code starting from the generic architecture file, and even possibly to create a fully-functional self-contained HTML page with a complete GUI to control the DSP processor or instrument.

3.3 WebAudioModules (WAMs)

The Web Audio Module (WAM) API discussed in [4] covered the core functionality of native audio plug-in instruments and effects, and demonstrated two proof-of-concept synthesizer implementations in asmjs, running in ScriptProcessorNode. The authors have since extended the API to target WebAssembly and AudioWorklets.

Figure 4: WAM instruments currently available online

Although WAMs may be implemented entirely in JavaScript, the authors' primary focus is in cross-compiling C++ DSP code into WASM binary format. Given the large amount of existing C++ implementations in native domains, classes have been developed for the iPlug and JUCE frameworks to accelerate porting efforts. Our new work has ported four additional synthesizers which sound and look like their desktop counterparts. The current WAM instrument library is available online[12]. Our community site also contains a draft specification. Source code tracking the latest API draft is available at GitHub[13].

The WAM API is split into two parts. The WAM *processor* implements the DSP part and runs in the high priority audio thread, most often as a WASM module. The WAM processor extends the standard AudioWorkletProcessor class, and provides the glue between JavaScript and WASM. The WAM *controller* is responsible for loading and creating the processor part, handling presets, and exposing the plug-in control interface. The control interface offers scripting, GUI, and MIDI access to the plug-in. The WAM controller is derived from AudioWorkletNode (which runs on the main thread), and is written in JavaScript. The controller and processor communicate via a MessagePort channel with asynchronous events. This bi-directional channel is used for parameter, patch, and MIDI data. Currently the stream flows in one direction only, from the controller to the processor.

To enable interoperability with other Web Audio projects, WAMs need to resolve audio IO and control (scripting, GUI and MIDI) compatibility issues. Since WAMs are essentially extended AudioWorklets, they operate like standard Web Audio API nodes. They integrate directly into the Web Audio API AudioGraph, and thus audio IO and basic scripting support is already compatible with projects conforming to the Web Audio API standard. The WAM controller class also complies with the Web MIDI API.

The specific goal of the WAM framework in this paper is to agree on a common hosting interface, i.e., to devise a standard API for A) extended scripting capabilities for synthesis parameter and patch manipulation, B) MIDI stream connectivity, and C) GUI related issues. In addition, we aim to devise a distribution strategy that streamlines loading and instantiating a WAM in a host environment, and a way to expose WAM collections, distributed via the internet, with proper metadata.

4 CONVERGED API

4.1 How can our projects co-operate?

WAMs describe the audio processor as a single Web Audio node and propose a controller/host to load it and configure its GUI, ports, etc. In the FAUST generated files, when targeting Web Audio, the audio processor is also considered as a single AudioWorklet node, and the JS generated code acts as the controller / host part like the approach used in WAMs. Both FAUST and WAMs have a rather complete API for the audio processor and/or controller part that is similar to that of the Web Audio interface (shared by all Web Audio nodes), with important additions described in previous sections (metadata, MIDI, audio buffer size can be set, etc.)

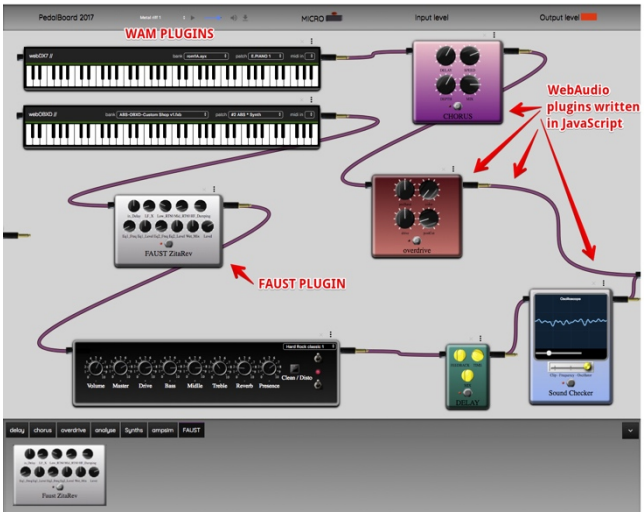

Figure 5: Two WAM synths and a FAUST plug-in have been integrated in the pedal board

```
1  <link rel="import" href="./js/pedal-behavior.html">
2  <link rel="import" href="https://webaudiomodules.org/temp/wam-host.html">
3  <dom-module id="pedal-alike-dx7">
4    <template>
5      <div class="laPedale">
6        <wam-host id="dx7"></wam-host>
7      </div>
8    </template>
9    <script>
10     class PedalAlikeDx7 extends PBPlugin(Polymer.Element) {
11       ready() {
12         // ...
13         host.load("https://webaudiomodules.org/temp/synths/dx7.js")
14         .then( function (wam) {
15           wam.connect(dx.soundNodeOut);
16         });
17       }
18     }
19     // Register the x-custom element with the browser
20     customElements.define(PedalAlikeDx7.is, PedalAlikeDx7);
21   </script>
22 </dom-module>
```

Figure 6: Code for importing the DX7 WAM into the pedal board. The DX7 is imported at line 2. Its GUI is wrapped in the template at line 6.

The pedal board from the WASABI project is mainly a host - an interactive web app for manipulating Web Audio plug-ins, arranging them on the screen, connecting them etc. The plug-ins developed so far are rather simple, with the noticeable exception of the tube guitar amp simulator, but all are made of multiple Web Audio nodes. It's pure "JS audio coded from scratch, not compiled or generated", contrarily to FAUST or WAMs. The "audio processor part" API of each pedal board plug-in is a subset

[12] https://webaudiomodules.org/wamsynths/
[13] https://github.com/webaudiomodules

of FAUST and WAM API methods and properties, with sometimes slightly different names. The definition of plug-ins as Web Components is appealing as components could easily be made available on a remote server and would not conflict with the host code (HTML id attributes, JS names, CSS rules, etc.) As a proof of concept, two WAMs have been repackaged as Web Components and imported in the pedal board: the Yamaha DX7 FM/PM and the OBXD virtual analog synthesizers[14]. This is shown in Figure 5. The code for packaging the remote DX7 synth is shown in Figure 6.

We also managed to import a simple headless FAUST plug-in (the Zita_rev1 reverb[15]) by wrapping it as a Web Component HTML file (reusing the JavaScript code that loads the WASM code and implements its audio processor API). For this last example, we built a simple GUI in the HTML template section of the Web Component.

What did we learn here? First, we need to have an "audio processor" API that can handle both a single node audio processor (e.g. AudioWorkerNode in WAM and FAUST implementations) - as well as multiple nodes (i.e. pedal board plug-in implementation), and expose methods and properties as closely as possible to the those of the standard AudioNode interface. In addition, we should have methods and properties for describing a plug-in (name, version, author etc.), for managing its life cycle, for saving/restoring its state, managing presets/banks and for MIDI. And we must align the parts of our respective APIs that match in terms of functionality/feature.

Finally, we can say that packaging Web Audio plug-ins as Web Components in an HTML file, making them available through a remote URL, is a desirable way of distributing and importing them.

The following sections will detail the proposed API and best practices.

4.2 Headless Plug-ins

P1: class Plugin extends AudioNode {...}

A headless plug-in should inherit from AudioNode (or implement this interface), and extend its properties and methods as follows later in this section. To this end, a single-node plug-in extends AudioWorkletNode. A multi-node plug-in also exposes the AudioNode interface, but instead of matching AudioWorkletProcessor, it contains an internal audio processing graph composed of other AudioNodes.

P2: readonly property JSON metadata

Plug-ins should expose JSON-format metadata containing name, version, category, type (audio / MIDI / both), description, thumbnail image URLs and author information.

P3: readonly property JSON descriptor

A plug-in exposes its parameter space via a descriptor array. Each item in the array holds key, type, range, default, unit, label and normalized flag fields. Type is a hint for generic GUI control type, following the approach used by FAUST.

P4: get/setParam(key, value)

Since plug-ins may have many synthesis / processing parameters, it might not be feasible to implement the entire parameter space as AudioParams. key is an integer or a string and value is a number or object (for parameter group support). The transfer format between controller and processor is `{ type:"param", key:key, value:value }`

P5: get/setPatch(data, index)

data is an opaque block, index is an integer that is optional for set. The transfer format is `{ type:"patch", data:data, index:index }`. Bank is set using P4.

P6: void onMidi(msg)

msg is an array-like object identical to the Web MIDI API's onmidimessage event.data. The transfer format is `{ type:"midi", data:msg }`. Sysex is set using `setParam("sysex", data)`, as discussed in P4.

P7: get/setState(data)

fetches or restores plug-in state. data is opaque, and the transfer format is `{ type:"state", data:data }`

P8: readonly property String[] patchNames

Since patch data is treated as an opaque block, the plug-in needs to parse bank patch names internally.

P9: inputChannelCount extension

Audio IO configuration follows AudioWorkletNodeOptions specification, extended with inputChannelCount array.

P10: Plug-ins should preferably conform to two-part namespacing

...in form of vendor.model. This can be used in AudioWorkletProcessor/Node class names, as well as in element names (e.g., `<vendor-name>`).

4.3 Plug-ins with a graphical user interface

The Web provides many ways of implementing user interfaces with HTML/CSS/JS code. Plug-ins that come with a graphical user interface (GUI) should avoid any possible namespace conflicts (HTML id attributes, CSS rules, JavaScript variables), and should come on encapsulated form, easy to import in a host.

One standard solution consists in wrapping them as Web Components. A Web Component follows the W3C standard APIs: HTML templates (inert HTML code meant to be instantiated), shadow DOM (encapsulation), HTML custom element (associate a plug-in with a declarative HTML tag, ex `<wam-obxd></wam-obxd>`), and HTML imports (use `<link rel="import" href="wam-obxd.html">` and you'll import all the HTML/CSS/JS code of the plug-in, avoiding conflicts as everything is encapsulated). Then it becomes possible

[14] http://www.webaudiomodules.org/wamsynths/
[15] https://faust.grame.fr/libraries.html#dm.zita_rev1

in any host, to wrap this plug-in into any existing shell that will make it manageable by the host. In the pedal board host, this shell is also a Web Component that defines a standard HTML template for any plug-in so that it can be positioned using the mouse and connected to other plug-ins. It can be generated after interrogating the imported "standard plug-in": size of its GUI, number of i/o, name, thumbnail icon, etc.

4.4 Host API

The host is responsible for accessing plug-in metadata and standard methods and properties defined above. These include loading and instantiating a plug-in, connecting it to a Web Audio API audio graph, providing MIDI and other control streams, presenting the GUI, providing patch data, and saving and restoring plug-in state. It may also aggregate cloud plug-in folders as discussed in the next section.

Host-plug-in communication, if required, may be handled using key-value pairs as described in P4 above. The property set is still under discussion. A host may also be packaged as a plug-in conforming to a subset of the APIs described in Sections 4.2 and 4.3. A simple single plug-in wrapper, such as `<wam-host src="url">` offers only plug-in loading API. The URL is exposed declaratively as a custom element src attribute, and imperatively as src property and its accompanying load() method. More advanced hosts, such as the pedal board, may offer more elaborate "host dependent" behavior to the plug-in standard we propose, by wrapping plug-ins into a richer container (as explained in section 3.1)

4.5 Guidelines: distributing and reusing plug-ins

The Web Components W3C standard[16] defines a way to easily distribute components with encapsulated HTML/CSS/JS/WASM code without namespace conflicts. Where native plug-ins needed to be downloaded and installed, URLs make no difference between a local or a distant plug-in, a Web Component plug-in could be used remotely just by its (possibly RESTful) URL reference. This makes writing a plug-in remote server easy.

URLs also enable an intuitive distribution strategy for both headless and GUI-equipped versions of a plug-in. A headless version is distributed as a (minified) JavaScript package (plug-in.js), whereas its complete GUI version is encapsulated into an HTML import (plug-in.html).

Repositories and aggregator sites can collect plug-in URLs and expose their collections as "cloud plug-in folders". This folder is also exposed as a URL, pointing to a JSON structure. JSON includes plug-in metadata such as headless/GUI plug-in and thumbnail URLs, name, version, category, type: audio/MIDI/both, description, author info, and freeform tags. A plug-in host may then point to several aggregator site URLs in a similar manner that DAWs handle native plug-in folders.

Repositories may naturally host the plug-ins themselves. Since the plug-in is a Web Component, a repository Web page can embed it as easily as it would embed a video. Prospective users may then explore the plug-in before adding its URL into their own link collection, or DAW. Native plug-in vendors may also demonstrate their offerings online, without requiring manual installation.

5 CONVERGED API IMPLEMENTATIONS

The converged API allows different approaches to be used to implement a high-level audio plug-in for the browser and we have tested this with several proofs of concept.

For example, we've got an online host[17] (the pedal board) that is actively being developed to take advantage of the converged API we designed, and that is the main testbed for plug-ins being developed. We are already importing plug-ins located on other domains/

servers (via URLs). WAMs, FAUST and pedal board toolchains and SDKs are being updated to conform to the proposed API / packaging. The packaging process (make the plug-in a Web Component for easy redistribution) can be automated, as well as importing a plug-in into the host without any manual operation. The tools/generators for generating Web Component wrappers are still under development.

6 CONCLUSION / DISCUSSION

The pedal board team's short term plans are to increase the number of pedals available, publish them on a remote repository and hopefully convince other developers to contribute and/or adapt their host/plug-ins to this proposed API. The pedal board still needs development: pure MIDI plug-ins need to be tested, and MIDI I/O and routing should also be implemented (i.e. connect a MIDI event generator plug-in to a synth within the pedal board GUI). The WAM team is working on a DAW that can host plug-ins that follow this API as well as updating the iPlug C++ plug-in framework to support the WAM API out of the box. The FAUST team is investigating how its own pure FAUST based host (using the embedded dynamical compilation model[18]) could be extended to support the defined host API.

The API is still a draft version and needs contribution/critics from audio developers. We should also conduct a comparison with existing native plug-in standards: i.e. identify lacking features (like timestamped control events, a proper DAW style "transport" API with multiple time bases [tempo and musical time, global sample counting clock, etc.]), but also learn from their shortcomings.

This paper is the first initiative that involves synchronizing the efforts of three groups of developers who have been working on various approaches for implementing high level audio "plug-ins" in the browser. We made our plug-ins inter-operable by conforming to a converged API and architecture, and propose a solution to distribute them using URLs and by packaging them as Web Components. Our proposal and proof-of-concept demos will

[16] https://www.webcomponents.org

[17] See footnote 1 for URL of the demo, also URL of a demonstration video.

[18] https://faust.grame.fr/faustplayground/

be submitted to the W3C Web Audio working group for evaluation. We hope that this work may become a starting point for a new addition to the Web Audio v2 specification, to provide needed functionality and infrastructure for musical applications of Web Audio.

ACKNOWLEDGMENTS

This work was supported by the French Research National Agency (ANR) and the WASABI team (contract ANR-16-CE23-0017-01). ElMahdi Korfed and Guillaume Etevenard helped developing these tools.

REFERENCES

[1] M. Buffa and J. Lebrun. 2017. Real time tube guitar amplifier simulation using WebAudio. In *Proc. 3rd Web Audio Conference* (WAC 2017). London, UK.

[2] M. Buffa and J. Lebrun. 2017. Web Audio Guitar Tube Amplifier vs Native Simulations. In *Proc. 3rd Web Audio Conference* (WAC 2017). London, UK.

[3] N. Jillings and al. 2017. Intelligent audio plugin framework for the Web Audio API. In *Proc. 3rd Web Audio Conference* (WAC 2017). London, UK.

[4] J. Kleimola and O. Larkin. 2015. Web audio modules. In *Proc. 12th Sound and Music Computing Conference* (SMC15). Maynooth, Ireland.

[5] M. Buffa, M. Demetrio, and N. Azria. 2016. Guitar pedal board using WebAudio. In *Proc. 2th Web Audio Conference* (WAC 2016). Atlanta, USA.

[6] M. Buffa and al. 2017. WASABI: a Two Million Song Database Project with Audio and Cultural Metadata plus WebAudio enhanced Client Applications. In *Proc. 3rd Web Audio Conference* (WAC 2017). London, UK.

[7] Orlarey, Y., Fober, D. & Letz, S. (2004). Syntactical and Semantical aspects of Faust. *Soft Computing, 8(9):623–632, 2004.*

[8] S. Letz, Y. Orlarey, and D. Fober. 2017. Compiling Faust Audio DSP Code to WebAssembly. In *Proc. 3rd Web Audio Conference* (WAC 2017). London, UK.

[9] Choi, H. & Berger, J. (2013). WAAX: Web Audio API eXtension. In *Proc. Int. Conf. New Interfaces for Musical Expression (NIME'13), Daejeon, Korea.*