WAP: Ideas for a Web Audio Plug-in Standard

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

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

Guillaume Pellerin, Stéphane Letz
IRCAM, GRAME,
guillaume.pellerin@ircam.fr
letz@grame.fr,

ABSTRACT

Several native audio plug-in formats are popular today including Steinberg's VST, Apple's Audio Units, Avid's AAX and the Linux audio community's LV2. Although the APIs are different, all exist to achieve more or less the same thing - represent an instrument or audio effect and allow it to be loaded by a host application. In the Web Audio API such a high-level audio plug-in entity does not exist. With the emergence of web-based audio software such as digital audio workstations (DAWs), it is desirable to have a standard in order to make Web Audio instruments and effects interoperable. Since there are many ways of developing for Web Audio, such a standard should be flexible enough to support different approaches, including using a variety of programming languages. New functionality that is enabled by the web platform should be available to plug-ins written in different ways. To this end, several groups of developers came together to make their work compatible, and this paper presents the work achieved so far. This includes the development of a draft API specification, a small preliminary SDK, online plug-in validators and a set of examples written in JavaScript. These simple, proof of concept examples show how to discover plug-ins from repositories, how to instantiate a plug-in and how to connect plug-ins together. A more ambitious host has also been developed to validate the WAP standard: a virtual guitar "pedal board" that discovers plug-ins from multiple remote repositories, and allows the musician to chain pedals and control them via MIDI.

1 - INTRODUCTION

The Web Audio API includes a set of unit generators called AudioNodes for graph-based audio DSP algorithms. The standard AudioNodes allow for developing a range of web applications that require audio engines that go beyond simple playback. The recent addition of the AudioWorkletNode provides an efficient way to implement custom low-level processing, significantly increasing the possibilities of this technology. There are many different apps created with the Web Audio API that run independently, however there is no standard way to make them interoperable i.e. take a drum machine developed by X, load it into an application developed by Y and apply audio processing developed by Z. In the native audio world, these interchangeable units are called "audio plug-ins" and applications that can use them are known as "hosts" which are typically DAWs.

The authors of this paper come from different research groups that have all been developing their own solutions for implementing audio plug-in-like entities in the browser. This paper discusses our ideas for a unified "Web Audio plug-in" standard (WAP) and the infrastructure surrounding such a standard. Other researchers' initiatives exist, such as the Web Audio API extension framework (WAAX) [5] and JSAP [1]. Our proposal differs, in that it aims to bring together several approaches already utilized by our groups, allowing Web Audio plug-ins to be developed A) in JavaScript using high-level AudioNodes, B) in JavaScript via AudioWorklet, C) in C++ (via Emscripten/WebAssembly), or D) by using Domain Specific Languages (DSLs) (as illustrated in Fig. 1). We would like to be able to support all approaches with a unifying Web Audio Plug-in standard. The standard should be flexible and consider future possibilities such as use in progressive web apps [1] (PWA) or in native environments.

One of the groups involved created FAUST, a DSL for audio DSP, which supports targeting Web Audio [4][7]. Another group created Web Audio Modules (WAMs) - an API for developing web-based plug-ins using C++ and WebAssembly [2], another group has been creating a variety of Web Audio applications including a virtual pedal board plug-in host and virtual guitar amp simulators [8].

At the time of writing there are relatively few commercial audio-first products based on the Web Audio API, in comparison to the wide range of desktop audio software. These include, for example two web-based DAWs[2], a hearing test app, and two online music notation packages[3]. This is likely to change since the introduction of the AudioWorklet, which will facilitate many more pro-audio use cases. Based on the shared interests of all the authors involved, and our observations about changes in web-based audio software, we believe there is a clear need for a high level audio processing/generating unit, as part of- or to work with the Web Audio API.

2 - CONTEXT

In a previous paper [6] we provided a state of the art of native, desktop audio plug-in formats, and described what makes the web platform different. The authors in [2] and [10] have also presented overviews of the defining characteristics of different native plug-in APIs. For the work in this paper we decided to go back and look in detail at the 2003 Generalized Music Plug-in Interface (GMPI) final draft proposal[4], which, despite its age, provides a thorough overview of desirable qualities in an audio plug-in API. The LV2 plug-in API has been compared to the GMPI document in a categorized table that is published online[5]. We decided to make a similar comparison whilst making our Web Audio Plug-in specification[6] to guide our work. This enabled us both to identify the most important features a plug-in API should provide, but also to discard irrelevant specifications, and to think about the differences afforded by the web platform.

In the next sections we present the main features of our proposal as well as its current status.

---

[1] Progressive Web Apps enable users to experience native-like experiences along with web advantages. By installing a PWA, users' engagement allows to bypass some of the webapps' constraints (i.e. PWA can access hard disk storage, etc.)
[2] SoundTrap.com and BandLab.com
[3] noteflight.com, ultimate-guitar.com
[4] https://tinyurl.com/k2wy5ge, now inactive MMA working group.
[5] LV2 achievement of GMPI requirements: http://lv2plug.in/gmpi.html
[6] WebAudio plug-in vs LV2 vs GMPI: https://tinyurl.com/yd5fedrc

2.1 WAP and the GMPI

The GMPI draft proposal lists 114 requirements grouped into 23 categories. We divided the categories further into three groups based on their relevance to the fundamental requirements of an audio plug-in.

We decided that the first draft Web Audio plug-in specification should look at the following categories:

- **Host/plug-in Model**: We need to define how plug-ins are loaded, instantiated, and connected together. Hosting scenarios require mechanisms for plug-in discovery and host-plug-in interface description. We must also bear in mind that with web applications there might not be a dedicated host - a plug-in could run "standalone" in an embedded browser.
- **Events** and **MIDI**: There should be a way to send and receive note and control events to / from plug-ins and host, especially through the MIDI protocol which is a common way to link software as well as hardware instruments and effects. At the end, it should be compatible with other protocols like OSC.
- **Parameters**, **Persistence**: Plug-ins will need to expose their parameter set and provide getter/setters
- **Plug-in Files**: More generally, a way to make the state persistent so that loading and saving of presets/banks can be implemented.
- **User Interfaces**: Although some plug-ins may operate "headless", we need to support both generic and custom GUIs.

The second category group will be targeted in a subsequent API version. The lower priority categories are: **Host Services, Time, Latency, Copy Protection, Localization, API Issues,** and **Wrappers**. Finally, some categories are irrelevant for Web environments, or already defined in the lower level APIs such as Web Audio and WebRTC. These include **Real Time Threading**, **Sample Rate**, **Audio I/O**, **Control I/O** and **Results**.
In addition, we need to take into account that GMPI is dated, and that several modern native plug-in APIs have been developed since 2003 [10]. The web browser environment also means that there are additional considerations, that are not concerns for native plug-ins.

2.2 WAP Uniform Resource Identifier

A Web Audio plug-in standard should be "Web aware" and use URIs as identifiers for plug-ins and repositories which are first class Web citizens/resources. Host web apps should be able to discover remote plug-ins by querying plug-in repositories. Plug-ins should be usable without the need for manual installation, and the mixture of different JavaScript libraries and frameworks, should not cause any naming conflict or dependency problems.

2.3 Support for different WAP approaches

A Web Audio Plug-in standard should be able to support multiple approaches in terms of programming language and programming environment, including pure JavaScript, C++ (via WebAssembly) and domain specific languages. It should be possible to port existing code bases across to work as a WAP and DSLs should be usable for the audio processing part. For example the authors in [4] have developed the WAM API which allows the porting of native plug-ins to WAPs, and this has been demonstrated by porting several desktop plug-ins [6]. WAM support has recently been added to iPlug (a C++ audio plug-in framework) [10], and could also be added to the JUCE framework, to allow desktop plug-ins built with those frameworks to operate on the web. The FAUST creators have developed a script to compile FAUST .dsp files to WAPs [4, 12]. We hope to support more DSLs in the future (Fig. 1).

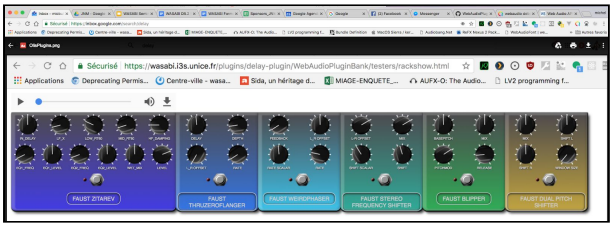

Figure 2. FAUST pedals, packaged as WAP plug-ins.

2.4 Support for Multiple Web Execution Environments

Although the web is the primary target for WAPs, there are use cases where integration with native applications may be required. For example, even if web DAWs offer very compelling features, experienced users may not be easily persuaded to switch to an online DAW immediately. In this case, supporting Web technologies in native apps could help making projects more portable across web and desktop platforms. Also the ability to test native plug-ins inside a familiar DAW before making a purchase, but without installation, might be an interesting use case. Native game engines and VR frameworks may also benefit from the Web Audio API which has good support for 3D audio. Options for bridging the web and native domains are explored in [9].

Fig. 1 shows WAPs as the pivot standard for all sorts of sources (JS, Faust, C++, etc.) and execution environments (standard browser, or browser embedded in a native plug-in, or as Progressive Web Apps or Chromium-based apps).

3 - CURRENT STATE OF OUR PROPOSAL

Our proposal consists of a draft specification, online tools and set of examples. Our goal is to extend and integrate existing web APIs, and to keep the proposed API as minimal as possible.

3.1 A Draft Specification

The goal of our draft Web Audio plug-in API proposal [6] was to devise a minimal set of mechanisms that allow interoperation between our independently developed frameworks. A high level overview of the proposal is given below.
A WAP extends AudioNode (or AudioWorkletNode) and thus inherits their familiar properties and methods. This ensures interoperation with standard Web Audio API nodes and applications built on top of the Web Audio graph. Integration with Web Midi is provided by MIDIPort members.
WAPs are either *composite* or *custom* audio nodes. Composite nodes[7] encapsulate an audio sub-graph that is built from any number of (elementary) AudioNodes. Custom nodes are

[7] https://tinyurl.com/ybwm3bjv

AudioWorklets, with a ScriptProcessorNode fallback. Although implementation details are outside the scope of the proposed API, a standard (but extensible) communication protocol between AudioWorkletNode and AudioWorkletProcessor was considered beneficial for increased reusability: for example, a generic DSP implementation running in the browser's realtime audio thread may be repurposed just by changing its main thread counterpart.

WAPs are GUI-aware but agnostic about their implementation strategy. This means that WAPs may be headless, or they may expose a visual HTML element (e.g., div, canvas, SVG, or custom element) which can be attached to DOM. *The WAP design will ensure that the GUI code is loaded only if necessary.*

WAP metadata describes implementation specific aspects of the plug-in. Metadata is available as a separate JSON file and also as a runtime object. Metadata describes audio and midi IO configuration, namespace attributes, parameter space, plug-in type, URIs and so on. WAP repositories may collect JSON files into aggregates for discovery purposes.

A WAP REST server/repository is described by a URI, which may point to an online or local filesystem resource. Metadata may describe separate URIs for headless and GUI equipped WAPs. We foresee two embedding strategies: a hosting web page may simply employ one of the URIs in a script/link tag. More complex WAPs, such as those implemented in WASM may however require a dynamic loading mechanism.

```
var ctx = new AudioContext();
 var player = document.getElementById("soundSample");
 var mediaSource = ctx.createMediaElementSource(player);
 var intermediateGain = ctx.createGain();

 var pluginURL = "https://wasabi.i3s.unice.fr/WebAudioPluginBank/WASABI/PingPongDelay3";
 var plugin = new WasabiPingPongDelay(ctx, pluginURL);

 plugin.load().then((node)=>{
   console.log("node",node);

   mediaSource.connect(node);
   node.connect(ctx.destination);
 });
```

Figure 3: Loading a headless plug-in from its URI, and inserting it into the Web Audio graph[8]

3.2 Online Tools, Tutorials and Examples

Along with the online documentation, we propose simple examples/tutorials both for the "host side" and "plug-in side" of our proposal, as well as online tools such as validators/testers. Some are presented in the different figures that follow. Each legend contains a footnote with the link to the runnable web app.

Host loading a headless plug-in: Fig. 3 shows extracts of a minimal host implementation that loads a headless plug-in and connects it to the Web Audio graph. Behind the scenes, a JSON metadata file is loaded from the plug-in URI. A `<script src="..."></script>` HTML tag is added if needed. Following that, the plug-in is initialized. Since it may load assets such as image files or a WASM module asynchronously, the `load` method returns a JavaScript promise. In this example, the name of the plug-in class is hard-coded but it could have been built dynamically from the content of the plug-in metadata JSON file (further examples show how to do this, such as the online plug-in tester from Fig 5). From a host's point of view, the plug-in might be of any kind: a Web Audio graph in a CompositeNode or a single CustomNode (AudioWorklet) node, written in JavaScript or in WebAssembly, etc.

```
var ctx = new AudioContext();
 var player = document.getElementById("soundSampl
 var mediaSource = ctx.createMediaElementSource(p
 var intermediateGain = ctx.createGain();

 var pluginURL = "https://wasabi.i3s.unice.fr/Web
 var plugin = new WasabiPingPongDelay(ctx, plugin

 plugin.load().then((node)=>{
   console.log("node",node);

   plugin.loadGui().then((elem)=>{
     console.log("elem",elem);
     document.body.appendChild(elem);
   });
   mediaSource.connect(node);
   node.connect(ctx.destination);
```

Figure 4: The same plug-in, with a GUI. GUI code and resources are downloaded only when they are needed[9].

Figure 5. Online plug-in tester[10]: enter the plug-in URI to validate the plug-in before publishing to a repository.

Host loading a plug-in with GUI: Fig. 4 shows the same example but this time, we also load asynchronously the GUI code (HTML, CSS, JS). The `loadGUI` method returns a single HTML element that contains the whole plug-in GUI. Here again, the method is asynchronous and returns a promise as a plug-in can have to load images for knobs, etc.

The `load` and `loadGUI` methods implementations are inherited by default when you extend the WebAudioPluginFactory class from the SDK, but can be overridden by the developer. In our examples, we use Web Components[11] to package the GUI files in a single HTML file, adding encapsulation and avoiding any naming conflicts. Behind the scenes the default `loadGUI` method creates a `<link rel="import" href="main.html">` when needed. If the developer prefers to use a canvas etc. for the GUI, they just need to override the `loadGUI` method.

More detailed examples are available on the documentation pages of the WAP proposal[12]. Some show in particular how to

[8] https://jsbin.com/xevahu/edit?html,js,console,output

[9] https://jsbin.com/jeretab/edit?js,output

[10] https://wasabi.i3s.unice.fr/WebAudioplug-inBank/testers/test2.html

[11] The WebComponents W3C standard (now in the HTML 5.2 specification) defines a way to easily distribute components with encapsulated HTML/CSS/JS/WASM code without namespace conflicts. See https://www.webcomponents.org

[12] http://wasabihome.i3s.unice.fr/webaudio-plug-in-proposal/

do real dynamic discovery, without hard coding any class names in the host code.

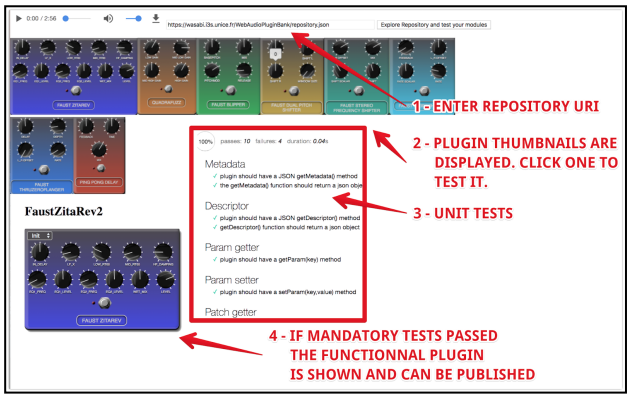

Figure 6: repo and plug-in tester[13].

Plug-in online validator: this online tool uses this dynamic behavior and is provided to plug-in developers to test their work. Fig. 5 shows an individual online plug-in tester. Copy and paste a plug-in URI and the code will be downloaded, the plug-in tested, and if a minimal set of tests passed, the plug-in will be runnable on the page and its GUI displayed, etc. You can then publish it on a repository. Notice that not all tests are mandatory to make the plug-in usable. For example, if a plug-in does not implement the load/save of its parameter state, it is still usable.

```
Promise.all([
        host.load("./../WASABI/QuadraFuzz/QuadraFuzz.html"),
        host.load("./../Faust/ZitaRev/FaustZitaRev.html"),
        host.load("./../Oliver-Larkin/ThruZeroFlanger/FaustThruZeroFlanger.html"),
        host.load("./../Oliver-Larkin/WeirdPhaser/FaustWeirdPhaser.html"),
        host.load("./../Oliver-Larkin/StereoFrequencyShifter/FaustStereoFreqShifter.html"),
        host.load("./../Oliver-Larkin/Blipper/FaustBlipper.html"),
        host.load("./../Oliver-Larkin/DualPitchShifter/FaustDualPitchShifter.html")
    ])
    .then(function (nodes) {
        nodes[0].connect(nodes[1]);
        nodes[1].connect(nodes[2]);
        nodes[2].connect(nodes[3]);
        nodes[3].connect(nodes[4]);
        nodes[4].connect(nodes[5]);
        nodes[5].connect(nodes[6]);
        nodes[6].connect(ctx.destination);
    });
```

Figure 7: Loading and chaining multiple plug-ins[14]

Plug-in repository online validator: Fig. 6 shows a remote repository tester: enter the URI of a REST endpoint and the list of plug-ins (with associated URIs) is fetched, followed by each plug-in's metadata file. Each plug-in's thumbnail image is displayed on the page and can be clicked to test the corresponding plug-in. If the mandatory tests passed, then you'll be able to try the plug-in online and get a full unit test report.

A plug-in "loader" utility object: in some cases a developer might want to be sure that several plug-ins have been loaded before chaining them. The WAP SDK provides a "plug-in loader" utility object that can be used with the `Promise.all` method from ES6, as shown in Fig. 7.

Writing a plug-in, minimal steps: now that we looked at how a host can discover and use a plug-in, it is easier to illustrate the different steps necessary to make a plug-in. the SDK provides multiple classes that can be subclassed, utility classes and objects. When writing a plug-in, different parts should be considered:

Part 1 - The `main.json` metadata descriptor that will contain a set of key/value pairs. Mandatory: plug-in name, vendor, version, thumbnail file (Fig. 8). The namespaced main class name of the plug-in and its relative URI can be inferred from the vendor and name values. Optionally, the type of current plugin state can be specified (default is JSON but some plugins may prefer binary blobs as get/setState payload).

```
{
  "documentation": "A classic Stereo Delay",
  "name": "PingPongDelay",
  "thumbnail": "WasabiPingPongDelay.png",
  "vendor": "Wasabi",
  "category":"Delay",
  "version": "1.0"
}
```

Figure 8: minimal json file for describing a plug-in

Part 2 - The plug-in "main class" that will implement the Web Audio plug-in API. It should extend either the WebAudioCustomNode class (i.e if the plug-in is an AudioWorklet) or the WebAudioPluginCompositeNode class (if it is made of a set of Web Audio nodes). We opted for a "convention over configuration approach"[15]: minimal steps are needed to have a runnable plug-in as many default values will be inherited. For example, the default main file will be `main.js` except if indicated in the metadata json file, `getParam/setParam` methods will be inherited, etc. Fig. 9 and 10 show source code extracts from the SDK classes that are provided for creating a CompositeNode plug-in, Fig. 11 shows a skeleton of what the main class of a plug-in would look like, and finally, Fig. 12 shows how this plug-in can be used as a regular Web Audio node. Full example is available online.

```
//----- 1 - CompositeAudioNode ----
// has connect/disconnect methods
// A custom composite node can be derived from this prototype.
class CompositeAudioNode {
  get _isCompositeAudioNode () {
    return true;
  }

  constructor (context, options) {
    this.context = context;
    this._input = this.context.createGain();
    this._output = this.context.createGain();
  }

  connect () {
    this._output.connect.apply(this._output, arguments);
  }

  disconnect () {
    this._output.disconnect.apply(this._output, arguments);
  }
}
```

Figure 9: The `CompositeAudioNode` prototype from the WAP SDK

[13] https://wasabi.i3s.unice.fr/WebAudioplug-inBank/testers/explorandtest.html
[14] https://tinyurl.com/ydg5yt32
[15] https://en.wikipedia.org/wiki/Convention_over_configuration

```
// -----------------------
// CREATE THE PLUGIN CLASS
// -----------------------
class WebAudioPluginCompositeNode extends CompositeAudioNode {
  constructor (context, options) {
    super(context, options);
    // ...
  }
  // P2 from WAP specification...
  set descriptor(descriptor) {
    this._descriptor = descriptor;
  }
  get descriptor() {
    return this._descriptor;
  }
  // ... other default properties and methods
  // that are indicated in the WAP spec
}
```

Figure 10: The class from the SDK that implements parts of the WAP API (default values). Meant to be subclassed.

```
class WasabiStereoDelay extends WebAudioPluginCompositeNode {

  constructor(ctx, options) {
    super(ctx, options)

    this.addParam({name:'feedback', defaultValue: 0.5, minValue: 0, maxValue: 1 });
    this.addParam({name: 'time',defaultValue: 0.5, minValue: 0, maxValue: 1 });
    this.addParam({name: 'mix',defaultValue: 0.5, minValue: 0, maxValue: 1 });

    /*    ################     API PROPERTIES     ###############   */
    this.params = {
      "feedback": this._descriptor.feedback.defaultValue,
      "mix": this._descriptor.mix.defaultValue,
      "time": this._descriptor.time.defaultValue,
      "status": "disable"
    }

    this.buildAudioGraph();
  }
  /*    ################     API METHODS     ###############   */
  get numberOfInputs(){
    return this.inputs.length;
  }

  get numberOfOutputs(){
    return this.outputs.length;
  }
}
```

Figure 11: A composite plug-in should extend the `WebAudioPluginCompositeNode` class.

```
var context = new AudioContext();
var myDelayPluginCompNode = context.createWasabiStereoDelayCompositeNode();
var oscNode = context.createOscillator();

oscNode.connect(myDelayPluginCompNode).connect(context.destination);

oscNode.start();
oscNode.stop(1.0);
```

Figure 12: Finally, a composite WAP can be used like a regular Web Audio node[16].

Part 3 - The plug-in "factory" class that will implement (or inherit) the `load` and `loadGUI` methods. Fig. 13 shows an example of such a factory method.

```
var WAPlugin = WAPlugin || {};

WAPlugin.WasabiStereoDelay = class WasabiStereoDelayFactory extends WebAudioPluginFactory {
    constructor(context, baseURI) {
        super(baseURI)
    }

    // load and loadGUI methods are inherited but
    // can be overriden here...
}
```

Figure 13: Plug-in factory class. This is the class used by the host to instantiate the plug-in.

3.3 The Pedal Board Host Application

Along with our "10-lines of code long example hosts", a "virtual pedal board" web app was developed [9] as a more ambitious host that gives the possibility to interactively chain plug-ins in order to create more complex sounds and configurations (Fig. 14). This pedal board host handles the discovery of plug-ins from multiple repositories (local or distant), global sound card I/O and gain adjustments, plug-ins' life cycles and interconnections as well as saving and restoring states/banks/presets.

The user can create instances of plug-ins by dragging and dropping their thumbnails into the main area. They can then position them, connect them together, etc. Finally, using the GUI (knobs, sliders, switches), the user can adjust each plug-in individually. By assembling an amplifier simulator, a reverb, a fuzz and a stereo delay, for example, one is able to create a rich psychedelic sound. We also contributed to the *webaudiocontrols library*[17] by adding MIDI control to all the GUI elements it offers (knobs, switches, etc.). Hence, the plug-ins can have their GUI controlled remotely via any MIDI controller. Fig. 14 shows a typical screenshot of this host application, combining plug-ins written in FAUST, in C++ (WAMs) and in JavaScript (guitar amp simulator, FX pedals). As explained in [9], this app has been also successfully run in custom browser builds packaged as VST plug-ins, bringing WAPs into native DAWs.

Figure 14: plug-ins inside a virtual pedal board[18].

4 - FUTURE WORK

For the first draft of the WAP API, we isolated a minimum set of features for implementation from the GMPI spec. Future work for the short term concerns enhanced MIDI functionalities and to look at how we can provide tempo and timeline synchronisation to plug-ins in a similar way to native plug-ins. Multiple input and output buses for common features such as side-chains are also something that must be considered, and how to provide flexible parameter modulation, when the plug-ins are integrated into a host with a timeline. Having the flexibility to support the "intelligent music production" capabilities offered by JSAP [1] is something we should investigate. We also plan to further evaluate the performance of the different approaches, with the latest browsers more thoroughly in the future, [2][7]. The pedalboard already hosts more than 20 WAPs, including some which are resource intensive, however, we need more plugins from independent vendors to make a solid analysis. This would also serve as an open invitation for the community to experiment with WAPs.

One more significant area for future work is to investigate how use in Progressive Web Applications would impact the API. Offline web apps, local filesystem access and low level access to hardware are possible developments of this technology. W3C

[16] https://jsbin.com/wadoqal/edit

[17] https://github.com/g200kg/webaudio-controls, a WebAudio UI widget lib.

[18] https://wasabi.i3s.unice.fr/dynamicPedalboard/

standards such as the Service Worker and File and FileSystem APIs enable web apps to run offline, and can give them privileges such as access to the hard disk when the user grants permission (e.g. by installing the app). Future relaxed constraints could include getting exclusive access to the sound card, for example - improving latency. We will follow the evolution of these W3C initiatives and will update WAPs accordingly.

5 - CONCLUSION

We presented a proposal for an open Web Audio Plug-in standard (WAP), that consists of a draft specification and a small, preliminary SDK including a tutorial and examples written in JavaScript. We have also made online plug-in and repository validators. As of today, FAUST, as well as the Web Audio Module (WAM) API are compatible with WAPs and we hope that more languages/environments will be supported soon. WAPs also use "composite nodes" to unify JavaScript plug-ins made of multiple AudioNodes and allow plug-ins that are a single AudioWorkletNode. In addition, we developed a more ambitious host application (the pedal board) that scans plug-ins repositories (local or remote) and can be used to assemble multiple plug-ins into a visual graph. This app has been evaluated with professional musicians [8] and shows that the host / plugin model we propose is suitable for use in more sophisticated applications. So far we have integrated more than 20 web audio plug-ins, including synthesizers and effects that have been developed using different technologies but are all complying with our API.

The preliminary WAP SDK is available at:
https://github.com/micbuffa/WebAudioPlugins

6 - ACKNOWLEDGMENTS

French Research National Agency (ANR) and the WASABI project team (contract ANR-16-CE23-0017-01).

7 - REFERENCES

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

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

[3] Buffa, M. & 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.

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

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

[6] Buffa, M., Lebrun, J., Kleimola, J., Larkin, O., and Letz, S. 2018. Towards an open Web Audio plug-in standard. In *Companion Proc. The Web Conference 2018* (WWW '18). Lyon, France. (April 23--27, 2018). 759-766. DOI= https://doi.org/10.1145/3184558.3188737

[7] Letz, S., Orlarey, Y., and Fober, D. 2018. FAUST Domain Specific Audio DSP Language Compiled to WebAssembly. In *Companion Proc. The Web Conference 2018* (WWW '18). Lyon, France. (April 23--27, 2018). DOI=https://doi.org/10.1145/3184558.3185970

[8] Buffa, M. and Lebrun, J. 2018. WebAudio Virtual Tube Guitar Amps and Pedal Board Design. Accepted to the *4th Web Audio Conference* (WAC 2018), Berlin, Germany.

[9] Kleimola J. and Campbell O. 2018. Native Web Audio API plugins. Accepted to the *4th Web Audio Conference* (WAC 2018), Berlin, Germany.

[10] Larkin, O., Harker, A., and Kleimola J. 2018. iPlug 2: Desktop Audio Plug-in Framework Meets Web Audio Modules. Accepted to the *4th Web Audio Conference* (WAC 2018), Berlin, Germany.