# A Model-View-Update Framework for Interactive Web Audio Applications

**Andrew Thompson**
andrew.thompson@qmul.ac.uk
Queen Mary University of London
London

**Gyorgy Fazekas**
g.fazekas@qmul.ac.uk
Queen Mary University of London
London

## ABSTRACT

We present the Flow framework[1], a front-end framework for interactive Web applications built on the Web Audio API. It encourages a purely declarative approach to application design by providing a number of abstractions for the creation of HTML, audio processing graphs, and event listeners. In doing so we place the burden of tracking and managing state solely on to the framework rather than the developer.

We introduce the Model-View-Update architecture and how it applies to audio application design. The MVU architecture is built on the unidirectional flow of data through pure functions, pushing side effects onto the framework's runtime. Flow conceptualises the audio graph as another *View* into application state, and uses this conceptualisation to enforce strict separation of the audio and visual output of an application.

Future plans for the framework include a robust plug-in system to add support for third-party audio nodes, a time travelling debugger to replay sequences of actions to the runtime, and a bespoke programming language that better aligns with Flow's functional influences.

## CCS CONCEPTS

• **Applied computing** → **Sound and music computing**;
• **Information systems** → *Web applications.*

## KEYWORDS

Declarative Programming Framework, Functional Programming, Web Audio API,

[1]https://github.com/flow-lang/flow-framework


*AM'19, September 18–20, 2019, Nottingham, United Kingdom*

ACM ISBN 978-1-4503-7297-8/19/09…$15.00
https://doi.org/10.1145/3356590.3356623

**ACM Reference Format:**
Andrew Thompson and Gyorgy Fazekas. 2019. A Model-View-Update Framework for Interactive Web Audio Applications. In *Audio Mostly (AM'19), September 18–20, 2019, Nottingham, United Kingdom.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3356590.3356623

## 1 INTRODUCTION

The Web has long been a desirable platform for interactive audio application development thanks in part to its ubiquity. The ability to target both mobile and desktop users in a single application makes the Web an attractive platform for artists and creative developers [7, 10]. The JavaScript programming language has become increasingly more powerful over time, facilitating the development of rich and complex applications. This complexity has come at a cost, however. How to effectively manage state in a large application has become a hot topic of development in the JavaScript community [5]. Despite this, it is not always clear how such management can be applied specifically to audio applications. To this end we present Flow as an attempt to unify audio and visual state management and focus developers' attention on building engaging applications.

### The Web Audio API

Real-time audio processing has been available for some time through the Web Audio API. The Web Audio API aims to provide a relatively high-level API for real-time audio processing and synthesis in the browser [8, 11]. Audio nodes such as oscillators, delays, and filters can be connected to one another to describe an audio processing graph.

While the Web Audio API can be considered high-level when viewed from a strictly signal processing perspective (developers rarely manipulate sample streams directly), common nodes such as envelopes and many effects are missing from the API. This is due, in part, to the fact the API aims to be a general solution for audio in the browser and not strictly a musical one.

A number of libraries exist to provide an even higher level of abstraction on top of the API, supporting more complex synthesis and sound processing objects for the express purpose of musical applications [3, 4, 6]. The Flow framework should be considered orthogonal to these types of libraries.

Instead, more apt comparisons could be drawn to larger frameworks that facilitate complete application design such as Flocking.js.

Flocking.js, a SuperCollider inspired framework, provides a declarative API for constructing Web Audio nodes. In it, synths and other unit generators (sound producing objects) are constructed with JavaScript objects (or JSON strings) leading to a so-called "document-oriented" approach to signal graph creation [2]. Although the initial API is highly declarative, synth declarations are stored in variables, falling back to a traditional method-based API for parameter updates.

When reviewing these libraries we have identified two main problems in how these frameworks suggest programs should be structured:

- **Audio and UI Coupling**: Often audio nodes are manipulated directly in the event callbacks of various UI elements. This causes a tight coupling between the UI and the audio graph such that changing one often means changing the other. Situations may also arise whereby external changes to an audio node can cause the UI to become out of sync with the node's actual value.
- **Fragmented State**: As audio nodes and UI elements created with JavaScript are stateful objects, the entire application state can be thought of a collection of objects scattered throughout the source code. This becomes most problematic when coupled with the previous problem, making it difficult to track down what state is changed and when.

Arguably the above comes down to a matter of coding style and can be avoided with discipline. The APIs exposed by many libraries make it easy to run into the above problems, however. BRAID [9] embraces the tight coupling between UI elements and audio nodes so much so that each UI widget also has an accompanying code editor that contains that element's callback code. These callbacks rely on global variables to access audio nodes and other parts of the application. The Web Audio eXtension (WAAX) provides both a collection of audio nodes and custom UI elements tailored to music interfaces. These UI elements can be connected directly to audio nodes and their parameters [1].

Flow takes a different approach. Both audio and HTML are created using the document-oriented approach presented by Flocking.js, but expands on this by mandating that both must be created from a single model of application state while also tightly controlling when and how that model can be updated.

## 2 THE FLOW FRAMEWORK

The Flow framework introduces a Model-View-Update architecture (Figure 1) for structuring applications. Central to this is a runtime that enforces unidirectional flow of data in an application and reliance on pure functions to provide guarantees about application state that are otherwise not possible.

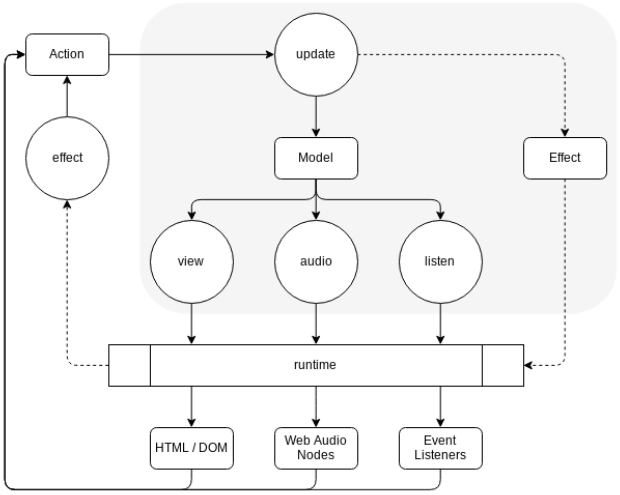


**Figure 1: Unidirectional data flow in the Model-View-Update architecture. Rectangles represent data/objects and circles represent functions. The grey shaded area shows the pure core of the Flow framework.**

The application's **Model** should represent the entire state of an application. This serves as the single source of truth for all other pieces of an application to draw from. This makes it very difficult to get into situations where application state is fragmented across the codebase.

The runtime then passes the Model to a **View**. A typical Flow program has two main views: an HTML tree and an audio graph. While at first it can seem odd to extend the definition of an application View to include the audio graph, conceptualising the audio graph as just another representation of the Model fits naturally into Flow's architecture. By ensuring the HTML and audio graph are both generated from the same Model, Flow can ensure situations where the audio and visual output of an application are out of sync are impossible. Importantly, the audio and UI are now decoupled. UI elements cannot modify an audio element's state directly.

Instead, a single **Update** function has the sole responsibility of updating application state. The runtime dispatches **Actions** in response to events such as user interaction. Actions have the general form

```
{ action: String, payload: any }
```

and are used by the Update function to determine what pieces of the Model need to be changed and how. Actions and the Update function make changes to application state explicit, making it easy for developers understand what an

application does while also making it simple to add new functionality.

Event listeners are needed to tell the runtime what actions to dispatch in response to specific events. Much like the Model and Update function, event listeners are centralised in a single **Listen** function. Here, DOM events such as a click on a specific element as well as audio timing events can be listened to.

```
const listen = model => [
  DOM.Event.click('.play', () => ({
    action: 'play'
  })),
  model.playing
    ? Audio.Event.every(0.25, time => ({
      action: 'next',
      payload: { time }
    }))
    : Audio.Event.none()
]
```

This has the effect of separating event listeners from the event source. The click event handler above uses a CSS selector to listen to click events on any DOM element with the ".play" class but makes no assumptions about the existence of such an element. This is powerful because it clearly separates *what should happen* from *what to show*.

Note how the model is used to conditionally listen to certain events. In this case, audio timing events are only listened to when a sequencer is playing. Just as with the audio and visual output of the application, event listeners are also generated from the model. This makes it simple to permit or restrict certain functionality based on application state as demonstrated above.

Flow operates on the assumption that the functions described above are *pure*. Pure functions are functions whose result depends entirely on its input; guaranteeing the same output is always produced when its input is the same. For Flow this means that for any fixed sequence of Actions, the same audio and visual output should be produced.

### Virtual Audio Graph

In order to help guarantee this purity, Flow models the audio graph (as well as the UI and event listeners) as simple data rather than stateful objects. The so-called "virtual DOM" is a common feature of modern front-end frameworks [2] [3]. Flow takes a similar approach to Flocking.js in this regard, extending this concept to the audio graph. This highly declarative approach allows developers to focus on what the audio graph should be and not the steps necessary to produce it. Where Flow diverges from Flocking.js, however, is in how audio nodes are created. Instead of requiring developers to construct data objects by hand, Flow makes available a library of functions for this purpose. Consider the following:

```
oscillator([ type("sine"), frequency(440) ], [
  gain([], [
    dac()
  ])
])
```

The same chain of audio nodes in the vanilla Web Audio API requires the developer to: (1) create and manage an audio context, (2) remember the API for updating the type and frequency parameters is different, (3) create a "flat" list of calls to each node's connect method, and (4) remember to call the start method on the oscillator to begin producing sound.

Although the resulting data structure is quite similar to those created by Flocking.js, the function-based API allows developers to make use of modern text editor features such as code completion and definition peeking. This goes a long way in reducing developer error, and making the errors that do occur easier to find.

Another, benefit comes from the ability to keep audio nodes "anonymous". Because Flow dictates that the audio graph must ultimately be defined inside a top level function, it is no longer necessary to worry about storing individual nodes in variables and giving them a name. Although perhaps initially trivial, nodes can be thought of as smaller pieces of a larger chunk of signal processing rather than individual objects.

This can be felt when abstracting sub-graphs into reusable pieces. The above oscillator and gain combo can be thought of as a basic synthesiser. To make this reusable in Flow we might write:

```
const synth = (freq, waveform, connections) =>
 oscillator([ type(waveform), frequency(freq) ], [
    gain([], [ ...connections ])
  ])

synth(440, "square", [
  dac()
])
```

The equivalent in vanilla Web Audio involves creating either a closure around a collection of audio nodes and returning an object that mimics methods such as `start` and `connect` or perhaps making use of the more modern class syntax. Both approaches simply involve writing more code to get these custom audio nodes working.

In this way, nodes can be thought of more as sound producing functions instead of sound producing objects. To change the frequency of the synth simply call the function again with a different frequency. To remove a node from the graph

[2] https://reactjs.org/docs/faq-internals.html

[3] https://vuejs.org/v2/guide/render-function.html#The-Virtual-DOM

simply do just that. When nodes do need to be uniquely identified (such as when creating a feedback loop), nodes can be keyed and referred to with a simple API:

```
Keyed.delay("delay", [], [
  gain([], [
    ref("delay")
  ])
])
```

The Flow runtime keeps track of the previously generated virtual audio graph and performs a diffing operation to determine what exactly needs to be created, removed, and updated. Potential errors can also be caught at this stage, preventing situations that might crash an application such as setting the frequency of an oscillator to some string value.

A complete example application can be found in the project's GitHub repository [4].

## 3 CONCLUSIONS AND FUTURE WORK

In these early stages of development, our primary focus has been on the implementation of a runtime suited to the Model-View-Update architecture. In this paper we have discussed why we think this architecture is beneficial to web audio application design: A declarative, consistent API for both audio and UI code, a purely functional core that lends itself to easy testing and is easy to reason about, and strong separation of concerns for more modular code.

There are some drawbacks to the framework, however. Particularly in small scale applications; the amount of boilerplate code necessary is far greater than that required by other frameworks. Additionally, interoperability with libraries that add new Web Audio nodes is limited. There is a rudimentary plugin system for Flow that could be utilised to support these libraries with a small amount of effort.

Beyond simply expanding the framework's collection of audio nodes, there are a number of meaningful advancements currently in development or planned for the future. With Flow's focus on "same input, same output" we can ensure that for any fixed sequence of Actions the same audio and visual output will be produced. It is currently possible to send Actions to the runtime directly that allows for a primitive form of debugging. We have planned a "time-travel debugger" that will allow developer to pause the runtime and rewind/replay a sequence of Actions. This combined with the ability to import and export an Action sequence would enable a powerful method of debugging difficult to achieve in frameworks that do not have the same guarantees as Flow.

Looking further into the future, a bespoke domain specific language (DSL) is desirable. While we have described the MVU core as purely functional, this can only be a recommendation and developers are free to mutate state and cause side effects wherever they please. It would be possible to leverage popular libraries such as immutable.js[5] to prevent mutation but we believe a new language entirely could bring additional benefits. In recent years there has been great interest in bringing static type checking to JavaScript, with Typescript[6] being a notable and popular example. Bringing strict typing to Web Audio development may help reduce bugs as the domain brings with it a tighter set of restrictions on values that other JavaScript APIs do not. Additionally, powerful functional programming concepts such as pattern matching and algebraic data types (ADTs) could be leveraged for more expressive code.

[4] https://github.com/flow-lang/flow-framework

[5] https://github.com/immutable-js/immutable-js

[6] https://www.typescriptlang.org/

## ACKNOWLEDGMENTS

This work was supported by the Engineering and Physical Sciences Research Council and the Arts and Humanities Research Council (grant EP/L01632X/1), ESPRC and AHRC Centre for Doctoral Training in Media and Arts Technology at Queen Mary University of London, School of Electronic Engineering and Computer Science.

## REFERENCES

[1] Hongchan Choi and Jonathan Berger. 2013. WAAX: Web Audio API eXtension.. In *NIME*. 499–502.

[2] Colin BD Clark and Adam Tindale. 2014. Flocking: a framework for declarative music-making on the Web. In *SMC Conference and Summer School*. 1550–1557.

[3] Oskar Eriksson. 2019. tuna. https://github.com/Theodeus/tuna/tree/f293fcb8ba3990aea00cfe50046bebc76b539dc6

[4] Yotam Mann. 2015. Interactive music with tone.js. In *Proceedings of the 1st annual Web Audio Conference*.

[5] Luca Mezzalira. 2018. *Front-End Reactive Architectures: Explore the Future of the Front-End using Reactive JavaScript Frameworks and Libraries*. Apress.

[6] Charles Roberts, Graham Wakefield, and Matthew Wright. 2013. The Web Browser As Synthesizer And Interface.. In *NIME*. 313–318.

[7] Charles Roberts, Graham Wakefield, Matthew Wright, and JoAnn Kuchera-Morin. 2015. Designing musical instruments for the browser. *Computer Music Journal* 39, 1 (2015), 27–40.

[8] Chris Rogers and Robert O'Callahan. 2011. *Audio Processing API*. W3C Working Draft. W3C. http://www.w3.org/TR/2011/WD-audioproc-20111215/.

[9] Ben Taylor and Jesse Allison. 2015. BRAID: A web audio instrument builder with embedded code blocks. In *Proceedings of the 1st international Web Audio Conference*.

[10] Florian Thalmann, A Perez Carillo, György Fazekas, and M Sandler. 2016. The semantic music player: A smart mobile player based on ontological structures and analytical feature metadata. (2016).

[11] Raymond Toy and Paul Adenot. 2018. *Web Audio API*. Candidate Recommendation. W3C. https://www.w3.org/TR/2018/CR-webaudio-20180918/.