ReactJS and ml5.js Instrument Tuner

Lets build a tuner! But why!? For fun of course! Also I'm a musician, so I suppose it makes sense that I'd like to do something related to music as an interesting project. But additionally, I have not been able to find any reliable, fast, and well designed tuners built for the web. The iPhone/Android app stores have some excellent ones, perhaps because phone microphones are more reliable than many built in mics on computers, and also maybe because there is more of a demand for using a tuner on a phone vs a computer. But for me, my practice space is built around my desktop, and I hate having my phone near my when trying to practice - so I've been pining for a good web based tuner. So here it goes!

The design process has not seriously begun yet - the initial build of this is just working out the logic, so stay "tuned" (get it, tuned?) for updates!

GitHub

CodeSandbox

This tutorial requires good JavaScript knowledge, some html and CSS, and some ReactJS knowledge.

First, lets fire up a Create React App environment. Make sure you have NodeJS installed. Either with that installer linked or with nvm if you're on Mac or Linux (it'll save you some serious heartache when you have projects that need different versions of Node..). Consider working in WSL2 if you're on Windows. Its super cool. You could also just use a codesandbox.

So in your terminal run:

```
npx create-react-app tuner
cd tuner
```

You're in!

Setting up AudioContext

First lets set up access to an AudioContext interface. We need this so that the ml5 pitch detection algorithm has actual data to work with. Just asking for access to a users microphone with getUserMedia is not enough. The AudioContext docs explain this well:

> The `AudioContext` interface represents an audio-processing graph built from audio modules linked together, each represented by an `AudioNode`. An audio context controls both the creation of the nodes it contains and the execution of the audio processing, or decoding. You need to create an `AudioContext` before you do anything else, as everything happens inside a context. It's recommended to create one AudioContext and reuse it instead of initializing a new one each time, and it's OK to use a single `AudioContext` for several different audio source and pipeline concurrently.

So, this gets its own custom React Hook which is setting up a new AudioContext (still have to include webkitAudioContext here for Safari....). Notice the use of useEffect and useRef hooks here. useEffect fires after the browser has painted, therefore allowing everything to mount before we fire off a new AudioContext. useRef can hold a mutable value in its .current property as you'll see below. Again see the docs linked for a refresher. All this is necessary because React handles working with the DOM in it's own React like way...

```
1  import { useRef, useEffect } from "react";
2
3  export default function useAudioContext() {
4    const audio = useRef();
5    useEffect(() => {
6      audio.current = new (window.AudioContext ||
   window.webkitAudioContext)();
7    }, []);
8    return audio;
9  }
10
```

getUserMedia

Now lets get access to the microphone - simple enough and relatively well documented in more standard circumstances, but it doesn't quite get implemented the same way in React.

So back in our App.js, we'll first set up another useEffect hook that will eventually not only getUserMedia, but also call our AudioContext hook, and set up the ml5 pitch detection algorithm. We'll use async/await here to get it loaded up, and only ask for the audio stream, not video.

```
  useEffect(() => {
    (async () => {
      const micStream = await
navigator.mediaDevices.getUserMedia({
        audio: true,
        video: false
      });
```

Now we'll initialize some fun stuff to move on to the next step. We'll get the ref from our useAudioContext hook, and set up a 'pitchDetectorRef' with another useRef().

```
const App = () => {
  const pitchDetectorRef = useRef();
  const audioContextRef = useAudioContext();
```

ml5.js

Now lets install ml5.

```
npm i ml5
```

This is a super awesome project that apparently aims to make machine learning approachable - well, count me in! It's based off of TensorFlow.js, giving us access to pretrained models for detecting cool stuff. In our case, pitch detection!

But it's important to note that you cant just install it and expect it to work without getting the pretrained models - the npm installation does not include these as far as I'm aware. So go get the model and put it in your public folder so that React has access to it directly.

Now make sure to import it into our App.js. By the way, here is what we've imported so far (with useEffect and useState thrown in. It'll come in handy in just a sec).

```
import React, { useEffect, useRef, useState } from "react";
import ml5 from "ml5";
import useAudioContext from "./use-audio-context";
```

Here's our useEffect hook finished, now with useRef initializing pitchDetectorRef.current with the ml5 pitchDetection method. This takes the location of the algorithm model, the AudioContext, the getUserMedia micStream, and a callback function to set the model loaded to true.

```
  useEffect(() => {
    (async () => {
      const micStream = await
navigator.mediaDevices.getUserMedia({
        audio: true,
        video: false
      });
      pitchDetectorRef.current = ml5.pitchDetection(
        "/models/pitch-detection/crepe",
        audioContextRef.current,
        micStream,
        () => setModelLoaded(true)
      );
    })();
  }, []);
```

useInterval

After some experimenting, too much data typically gets sent to the pitch detection algorithm, and you end up with way too much information than needed. This is an attempt to smooth that out a bit. Ideally we'd just try something as simple as using good old setInterval to have the function get called every x number of milliseconds, but unfortunately React and setInterval don't play nice, but there is a way to get them to work well together! Dan Abramov has a great post about this and I have dutifully robbed his code. But go check it out. We're just setting up a way to delay the running of our getPitch method. Cool.

It's going in its own file.

```
import { useEffect, useRef } from "react";

```

```
export default function useInterval(callback, delay) {
  const savedCallback = useRef();

  // Remember the latest callback.
  useEffect(() => {
    savedCallback.current = callback;
  }, [callback]);

  // Set up the interval.
  useEffect(() => {
    function tick() {
      savedCallback.current();
    }
    if (delay !== null) {
      let id = setInterval(tick, delay);
      return () => clearInterval(id);
    }
  }, [delay]);
}
```

Back in our App.js, lets set this up. Firstly we'll just if check there is anything in our pitchDetectorRef in the first place. Grand.

Oh, we also need to initialize a bunch of variables with useState. Here is everything that has been initialized so far -

```
const App = () => {
  const pitchDetectorRef = useRef();
  const audioContextRef = useAudioContext();
  const [tunerStarted, setTunerStarted] = useState(false);
  const [modelLoaded, setModelLoaded] = useState(false);
  const [pitchfreq, setPitchFreq] = useState(0);
  const [diff, setDiff] = useState(0);
  const [note, setNote] = useState([]);

```

Now we'll set up our useInterval hook to run main functions. I'll talk about this more later in the next section.

```
  useInterval(() => {
    if (!tunerStarted) {
      return;
    }
```

```
    if (!pitchDetectorRef.current) {
      return;
    }
    pitchDetectorRef.current.getPitch((err, detectedPitch)
=> {
      setNote(getNoteFromSemitones(pitchfreq,
getNumSemitonesFromA(pitchfreq)));
      setPitchFreq(Math.round(detectedPitch * 10) / 10);
      setDiff(
        getDifferenceInCents(detectedPitch,
getNumSemitonesFromA(detectedPitch))
      );
    });
  }, 1000 / 80);
```

Reference Frequency and Temperament

Unfortunately, all that the ml5 pitch detection algorithm does for us is detect the frequency(Hz) that is being played into the users microphone. There is a lot more work we need to do to get the rest of the information needed. Such as what note that frequency is closest to, and how far away we are from that intended note (in Cents, not Hz).

First we need our reference frequency - in this case, A = 440Hz. Currently I'm hardcoding this in but there is a good argument for allowing a user to enter their preferred reference A. There are many situations where one may want to tune to A = 442Hz, or even the conspiracy theorist attracting "universal pitch" of A = 432Hz! (This little world is quite fascinating, although, just for record, I don't believe any of that stuff).

Next we need to choose our temperament - we'll use equal temperament. This is a less contentious topic, although for practice purposes, an option to choose "just" temperament, along with the reference key could be useful - another possible future addition. But just a note, the difference between these two - just and equal - are actually quite shocking. The just temperament scale is the more "natural" version. All notes here occur naturally as part of the overtone series, and all are related by rational

numbers. Unfortunately though, they all have to relate to the key your are in, necessitating a reference key input if implementing this feature. So our modern idea of music, that has all sorts of notes and chords that are technically out of the key won't work well under this. Equal temperament was originally developed for keyboard instruments so they could be played in any key - although the result is a compromise, making it so that no two notes have that magical ringing 'perfectly in tune' sound. Anyways that's enough of that. Check this out if you are interested.

Also we need an array with all the notes of the scale.

```
const A = 440;
const equalTemperment = 1.059463;
const scale = ['A', 'A#', 'B', 'C', 'C#', 'D', 'D#', 'E',
'F', 'F#', 'G', 'G#'];
```

Detect how many semitones away from A

First step is to work out how many semitones we are from A440. Our wonderful friend Google has helped me find the equation we need here.

$fn = f0 * (a)^n$

> *f0* = the frequency of one fixed note which must be defined. A common choice is setting the A above middle C (A4) at *f0* = 440Hz.
> *n* = the number of half steps away from the fixed note you are. If you are at a higher note, *n* is positive. If you are on a lower note, *n* is negative.
> f_n = the frequency of the note *n* half steps away.
> *a* = (2)1/12 = the twelfth root of 2 = the number which when multiplied by itself 12 times equals 2 = 1.059463094359...

As suggested, we'll be using A 440Hz as our fixed note, and we'll be solving for *n*. The number of semitones(half steps) away we are, given the frequency. And *a* is equal temperament(1.059....).

Here it is in code, with the addition of it returning null if there is no frequency calculated. And also rounding it up as we need the nearest integer, or full half-steps away value - because remember we have to guess the intended note that the user has played, and frankly the easiest way to guess this is by rounding...

```
function getNumSemitonesFromA (freq) {
  let diffInSemitones = 0;
  if (!freq) {
    return null;
  }
  return (diffInSemitones = Math.round(
    Math.log(freq / A) / Math.log(equalTemperment)
  ));
}
```

Get our current note from the difference in semitones

Here we'll just match up the result of our previous function to the relevant item in our scale array. Remember we have to handle negative numbers here because our equation returns negative numbers when the semitone detected is below A, and positive if it is above.

```
function getNoteFromSemitones (freq, diffInSemitones) {
  let note = scale[0];
  if (diffInSemitones > 0) {
    diffInSemitones = diffInSemitones % 12; // going back over array of scale if semitone above A
    note = scale[diffInSemitones];
  } else if (diffInSemitones < 0) {
    // same but if below A
    diffInSemitones = diffInSemitones % -12;
    note = scale[scale.length + diffInSemitones];
  }
  return note;
}
```

Get the difference in cents

Now we have to figure out how far away we actually are from our desired note. We can currently easily show how far away we are in Hz, but not in Cents. Which is the standard method of presenting this information in music. By the way this was completely new to me. I have been a professional musician for 10 years, and the entire time I thought Cents were the same as Hz!

Sadly, because the universe is evil, it is not as simple as just converting one number to another - the difference in cents will depend on where one is in the audio spectrum - I got the formula for this here.

> A Cent is a logarithmic unit of measure of an interval, and that is a dimensionless "frequency ratio" of *f2/f1*.

c = 1200 × log2 (f2 / f1)

f1 is our "correct" or intended frequency, which we need to solve for in our function. And *f2* is our current frequency.

This function is also solving for what the players' correct frequency should be, using our diffInSemitones value from our first function.

```
  function getDifferenceInCents (freq, diffInSemitones) {
    let centDiff = 0;
    if (!freq) {
      return null;
    }
    //Use the difference in semitones to figure out what the correctFreq should be.
    const correctFreq = A * Math.pow(equalTemperment, diffInSemitones);
    // Below is equation to convert diff in hz to cents. look it up...
    // We're rounding it off. too many decimals..
    centDiff = Math.round(1200 * Math.log2(freq / correctFreq));
    return centDiff;
  }
```

Putting it all together

Now for our central function - remember we'll be using the custom hook, useInterval that we set up earlier. This entire section is fairly straight forward as we've already done all the hard work. We'll be using our useState hooks that we set up to set the state of each of these items - our current note, our current frequency, and the difference in cents (how far we are from the correct pitch).

```
  useInterval(() => {
    if (!tunerStarted) {
      return;
    }
    if (!pitchDetectorRef.current) {
      return;
    }
    pitchDetectorRef.current.getPitch((err, detectedPitch) => {
      setNote(getNoteFromSemitones(pitchfreq, getNumSemitonesFromA(pitchfreq)));
      setPitchFreq(Math.round(detectedPitch * 10) / 10);
      setDiff(
        getDifferenceInCents(detectedPitch, getNumSemitonesFromA(detectedPitch))
      );
    });
  }, 1000 / 80);
```

And lets get it into some jsx! Thanks to the magic of React this is wonderfully easy.

```
return (
    <div>
```

```
      <div className="note-freq">
        {modelLoaded && <h2>Model Loaded</h2>}
        <button
          type="button"
          disabled={tunerStarted}
          onClick={() => setTunerStarted(true)}
        >
          Start
        </button>
        <button type="button" onClick={() =>
setTunerStarted(false)}>
          Stop
        </button>

        <h2>Note: {note}</h2>
        <p>freq: {pitchfreq} </p>
        <p>cents: {diff}</p>
      </div>

```

There we've got some text appearing if the ML model has loaded, a button to start/stop the tuner (which in the final design will definitely be just a toggle), then our note, frequency and difference in cents!

Framer Motion

Here I'll be using a wonderful animation library, framer motion. It is really intuitive, and has also allowed really good flexibility for what I'm currently working on - which is making this thing pretty!

At the moment, as you can see, quite literally all I have is a box with two lines, one that moves up and down. Not exactly very full featured. But it does the job for this first demo.

First get framer motion installed -

```
npm i framer-motion
```

And make sure to import motion as a named import-

```
import { motion } from "framer-motion";
```

I won't spend much time on this framework, but basically you can put "motion." before any html element and it unlocks a nice bag of tricks!

All we'll be doing here is move the y value up and down depending on our difference value. Multiplied by 3 only because I'm attempting to get the animation to be more intuitive. The movements are too small without adjusting this.

```
      <div className="tuner-wrapper">
        <motion.hr
          className="diff-hr"
          animate={{
            y: -diff * 3
          }}
        ></motion.hr>
        <hr className="ref-hr" />
      </div>
```

There are some simple styles that I've applied also, which you can check out if you are curious, but other than that we're done!

Conclusion

This has been a fairly challenging project for me. I started my self directed coding journey not long ago, and choosing this as my first big project was ambitious to say the least. Saying all that, I would really appreciate any feedback at all - even if you just don't like my function names I want to hear it! So please leave a comment, or email me! And remember to stay tuned for the final production version!