Converting Speech to PDF with NextJS and ExpressJS

 **Onuorah Bonaventure Chukwudi** Aug 4, 2022

With speech interfaces becoming more of a thing, it's worth exploring some of the things we can do with speech interactions. Like, what if we could say something and have that transcribed and pumped out as a downloadable PDF?

Well, spoiler alert: we absolutely *can* do that! There are libraries and frameworks we can cobble together to make it happen, and that's what we're going to do together in this article.

SERVER REPO (HTTPS://GITHUB.COM/BONARHYME/AUDIO-TO-PDF-SERVER)

CLIENT REPO (HTTPS://GITHUB.COM/BONARHYME/AUDIO-TO-PDF-WEB-CLIENT)

(#aa-these-are-the-tools-were-using) These are the tools we're using

First off, these are the two big players: Next.js and Express.js.

Next.js (https://nextjs.org) tacks on additional functionalities to React, including key features for building static sites. It's a go-to for many developers because of what it offers right out of the box, like dynamic routing, image optimization, built-in-domain and subdomain routing, fast refreshes, file system routing, and API routes... among many, many other things (https://nextjs.org/docs/getting-started) .

In our case, we definitely need Next.js for its API routes (https://nextjs.org/docs/api-routes/introduction) on our client server. We want a route that takes a text file, converts it to PDF, writes it to our filesystem, then sends a response to the client.

Express.js (https://expressjs.com) allows us to get a little Node.js app going with routing, HTTP helpers, and templating. It's a server for our own API, which is what we'll need as we pass and parse data between things.

We have some other dependencies we'll be putting to use:

1. react-speech-recognition (https://www.npmjs.com/package/react-speech-recognition) : A library for converting speech to text, making it available to React components.
2. regenerator-runtime (https://www.npmjs.com/package/regenerator-runtime) : A library for troubleshooting the "`regeneratorRuntime` is not defined" error that shows up in Next.js when using react-speech-recognition
3. html-pdf-node (https://www.npmjs.com/package/html-pdf-node) : A library for converting an HTML page or public URL into a PDF

4. axios (https://axios-http.com/) : A library for making HTTP requests in both the browser and Node.js
5. cors (https://www.npmjs.com/package/cors) : A library that allows cross-origin resource sharing

(#aa-setting-up) Setting up

The first thing we want to do is create two project folders, one for the client and one for the server. Name them whatever you'd like. I'm naming mine `audio-to-pdf-client` and `audio-to-pdf-server`, respectively.

The fastest way to get started with Next.js on the client side is to bootstrap it with create-next-app (https://nextjs.org/learn/basics/create-nextjs-app/setup) . So, open your terminal and run the following command from your client project folder:

```
Terminal
npx create-next-app client
```

Now we need our Express server. We can get it by `cd`-ing into the server project folder and running the `npm init` command. A `package.json` file will be created in the server project folder once it's done.

We still need to actually install Express, so let's do that now with `npm install express`. Now we can create a new `index.js` file in the server project folder and drop this code in there:

```
JavaScript
const express = require("express")
const app = express()

app.listen(4000, () => console.log("Server is running on port 4000"))
```

Ready to run the server?

```
Terminal
node index.js
```

We're going to need a couple more folders and and another file to move forward:

- Create a `components` folder in the client project folder.
- Create a `SpeechToText.jsx` file in the `components` subfolder.

Before we go any further, we have a little cleanup to do. Specifically, we need to replace the default code in the `pages/index.js` file with this:

```
JavaScript
import Head from "next/head";
import SpeechToText from "../components/SpeechToText";

export default function Home() {
  return (
    <div className="home">
      <Head>
        <title>Audio To PDF</title>
        <meta
          name="description"
          content="An app that converts audio to pdf in the browser"
```

```
        />
        <link rel="icon" href="/favicon.ico" />
      </Head>

      <h1>Convert your speech to pdf</h1>

      <main>
        <SpeechToText />
      </main>
    </div>
  );
}
```

> *Hey!* The imported `SpeechToText` component will eventually be exported from `components/SpeechToText.jsx`.

(#aa-lets-install-the-other-dependencies) Let's install the other dependencies

Alright, we have the initial setup for our app out of the way. Now we can install the libraries that handle the data that's passed around.

We can install our client dependencies with:

Terminal
```
npm install react-speech-recognition regenerator-runtime axios
```

Our Express server dependencies are up next, so let's `cd` into the server project folder and install those:

Terminal
```
npm install html-pdf-node cors
```

Probably a good time to pause and make sure the files in our project folders are in tact. Here's what you should have in the client project folder at this point:

File structure
```
/audio-to-pdf-web-client
├─ /components
|  └── SpeechToText.jsx
├─ /pages
|  ├─ _app.js
|  └── index.js
└── /styles
    ├─globals.css
    └── Home.module.css
```

And here's what you should have in the server project folder:

File structure
```
/audio-to-pdf-server
└── index.js
```

(#aa-building-the-ui) Building the UI

Well, our speech-to-PDF wouldn't be all that great if there's no way to interact with it, so let's make a React component for it that we can call `<SpeechToText>`.

You can totally use your own markup. Here's what I've got to give you an idea of the pieces we're putting together:

```JavaScript
import React from "react";

const SpeechToText = () => {
  return (
    <>
      <section>
        <div className="button-container">
          <button type="button" style={{ "--bgColor": "blue" }}>
            Start
          </button>
          <button type="button" style={{ "--bgColor": "orange" }}>
            Stop
          </button>
        </div>
        <div
          className="words"
          contentEditable
          suppressContentEditableWarning={true}
        ></div>
        <div className="button-container">
          <button type="button" style={{ "--bgColor": "red" }}>
            Reset
          </button>
          <button type="button" style={{ "--bgColor": "green" }}>
            Convert to pdf
          </button>
        </div>
      </section>
    </>
  );
};

export default SpeechToText;
```

This component returns a React fragment (https://reactjs.org/docs/fragments.html) that contains an HTML <``section``> element that contains three divs:

- **`.button-container`** contains two buttons that will be used to start and stop speech recognition.
- **`.words`** has `contentEditable` and `suppressContentEditableWarning` attributes to make this element editable and suppress any warnings from React.
- Another **`.button-container`** holds two more buttons that will be used to reset and convert speech to PDF, respectively.

Styling is another thing altogether. I won't go into it here, but you're welcome to use some styles I wrote either as a starting point for your own `styles/global.css` file.

▼ View Full CSS

```CSS
html,
body {
  padding: 0;
  margin: 0;
  font-family: -apple-system, BlinkMacSystemFont, Segoe UI, Roboto, Oxygen,
    Ubuntu, Cantarell, Fira Sans, Droid Sans, Helvetica Neue, sans-serif;
}

a {
  color: inherit;
  text-decoration: none;
}

* {
  box-sizing: border-box;
}
```

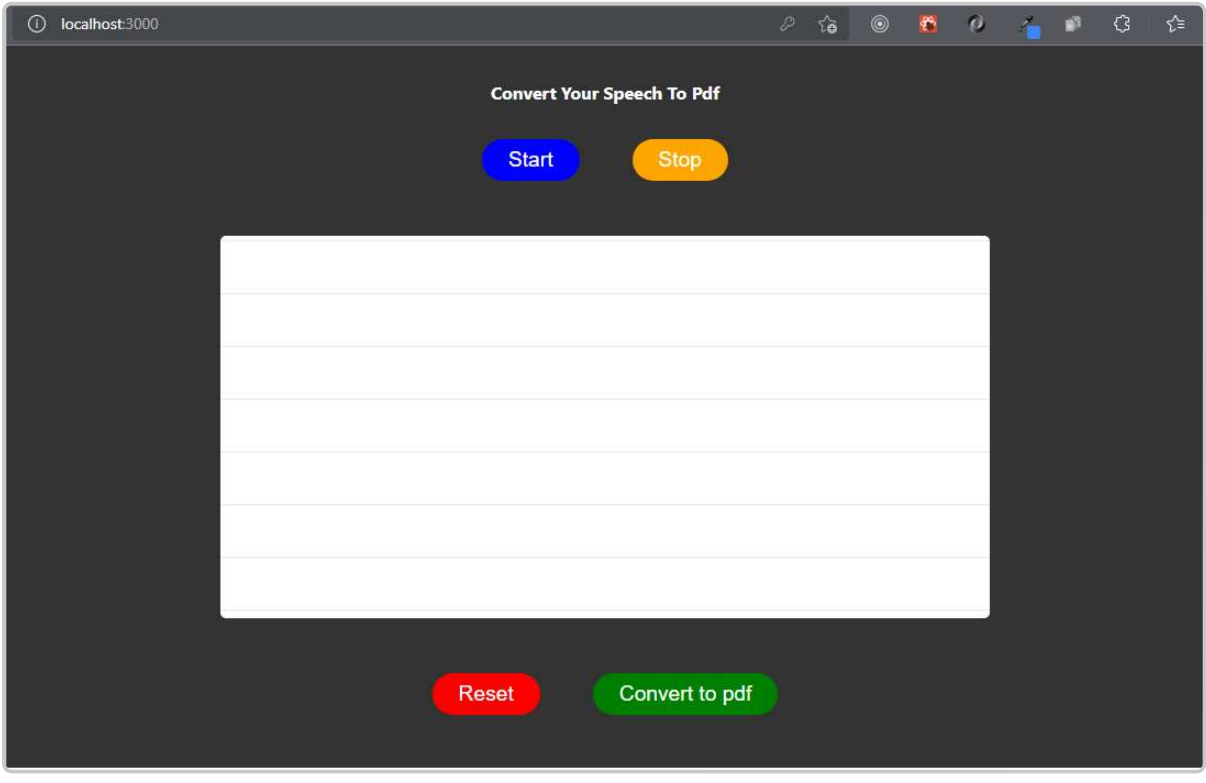
```
.home {
  background-color: #333;
  min-height: 100%;
  padding: 0 1rem;
  padding-bottom: 3rem;
}

h1 {
  width: 100%;
  max-width: 400px;
  margin: auto;
  padding: 2rem 0;
  text-align: center;
  text-transform: capitalize;
  color: white;
  font-size: 1rem;
}

.button-container {
  text-align: center;
  display: flex;
  justify-content: center;
  gap: 3rem;
}

button {
  color: white;
  background-color: var(--bgColor);
  font-size: 1.2rem;
  padding: 0.5rem 1.5rem;
  border: none;
  border-radius: 20px;
  cursor: pointer;
}

button:hover {
  opacity: 0.9;
}

button:active {
  transform: scale(0.99);
}

.words {
  max-width: 700px;
  margin: 50px auto;
  height: 50vh;
  border-radius: 5px;
  padding: 1rem 2rem 1rem 5rem;
  background-image: -webkit-gradient(
    linear,
    0 0,
    0 100%,
    from(#d9eaf3),
    color-stop(4%, #fff)
  ) 0 4px;
  background-size: 100% 3rem;
  background-attachment: scroll;
  position: relative;
  line-height: 3rem;
  overflow-y: auto;
}

.success,
.error {
  background-color: #fff;
  margin: 1rem auto;
  padding: 0.5rem 1rem;
  border-radius: 5px;
  width: max-content;
  text-align: center;
  display: block;
}

.success {
  color: green;
}

.error {
```

```
  color: red;
}
```

Hey! The CSS variables in there are being used to control the background color of the buttons.

Let's see the latest changes! Run `npm run dev` in the terminal and check them out.

You should see this in browser when you visit `http://localhost:3000`:



(#aa-our-first-speech-to-text-conversion) Our first speech to text conversion!

The first action to take is to import the necessary dependencies into our `<SpeechToText>` component:

JavaScript

```
import React, { useRef, useState } from "react";
import SpeechRecognition, {
  useSpeechRecognition,
} from "react-speech-recognition";
import axios from "axios";
```

Then we check if speech recognition is supported by the browser and render a notice if not supported:

JavaScript

```
const speechRecognitionSupported =
  SpeechRecognition.browserSupportsSpeechRecognition();

if (!speechRecognitionSupported) {
```

```
  return <div>Your browser does not support speech recognition.</div>;
}
```

Next up, let's extract `transcript` and `resetTranscript` from the `useSpeechRecognition()` hook:

```
JavaScript
const { transcript, resetTranscript } = useSpeechRecognition();
```

This is what we need for the state that handles `listening`:

```
JavaScript
const [listening, setListening] = useState(false);
```

We also need a `ref` for the `div` with the `contentEditable` attribute, then we need to add the `ref` attribute to it and pass `transcript` as `children`:

```
JavaScript
const textBodyRef = useRef(null);
```

...and:

```
JavaScript
<div
  className="words"
  contentEditable
  ref={textBodyRef}
  suppressContentEditableWarning={true}
  >
  {transcript}
</div>
```

The last thing we need here is a function that triggers speech recognition and to tie that function to the `onClick` event listener of our button. The button sets listening to `true` and makes it run continuously. We'll disable the button while it's in that state to prevent us from firing off additional events.

```
JavaScript
const startListening = () => {
  setListening(true);
  SpeechRecognition.startListening({
    continuous: true,
  });
};
```

...and:

```
JavaScript
<button
  type="button"
  onClick={startListening}
  style={{ "--bgColor": "blue" }}
  disabled={listening}
>
  Start
</button>
```

Clicking on the button should now start up the transcription.

(#aa-more-functions) More functions

OK, so we have a component that can *start* listening. But now we need it to do a few other things as well, like `stopListening`, `resetText` and `handleConversion`. Let's make those functions.

```JavaScript
const stopListening = () => {
  setListening(false);
  SpeechRecognition.stopListening();
};

const resetText = () => {
  stopListening();
  resetTranscript();
  textBodyRef.current.innerText = "";
};

const handleConversion = async () => {}
```

Each of the functions will be added to an `onClick` event listener on the appropriate buttons:

```JavaScript
<button
  type="button"
  onClick={stopListening}
  style={{ "--bgColor": "orange" }}
  disabled={listening === false}
>
  Stop
</button>

<div className="button-container">
  <button
    type="button"
    onClick={resetText}
    style={{ "--bgColor": "red" }}
  >
    Reset
  </button>
  <button
    type="button"
    style={{ "--bgColor": "green" }}
    onClick={handleConversion}
  >
    Convert to pdf
  </button>
</div>
```

Hey!

The `handleConversion` function is asynchronous because we will eventually be making an API request. The "Stop" button has the disabled attribute that would be be triggered when listening is false.

If we restart the server and refresh the browser, we can now start, stop, and reset our speech transcription in the browser.

Now what we need is for the app to *transcribe* that recognized speech by converting it to a PDF file. For that, we need the server-side path from Express.js.

(#aa-setting-up-the-api-route) Setting up the API route

The purpose of this route is to take a text file, convert it to a PDF, write that PDF to our filesystem, then send a response to the client.

To setup, we would open the `server/index.js` file and import the `html-pdf-node` and `fs` dependencies that will be used to write and open our filesystem.

```JavaScript
const HTMLToPDF = require("html-pdf-node");
const fs = require("fs");
const cors = require("cors)
```

Next, we will setup our route:

```JavaScript
app.use(cors())
app.use(express.json())

app.post("/", (req, res) => {
  // etc.
})
```

We then proceed to define our options required in order to use `html-pdf-node` inside the route:

```JavaScript
let options = { format: "A4" };
let file = {
  content: `<html><body><pre style='font-size: 1.2rem'>${req.body.text}</pre></body></html>`,
};
```

The `options` object accepts a value to set the paper size and style. Paper sizes follow a much different system than the sizing units we typically use on the web. For example, A4 is the typical letter size (https://www.papersizes.org/a-paper-sizes.htm) .

The `file` object accepts either the URL of a public website or HTML markup. In order to generate our HTML page, we will use the `html`, `body`, `pre` HTML tags and the text from the `req.body`.

> *Hey!* You can apply any styling of your choice.

Next, we will add a `trycatch` to handle any errors that might pop up along the way:

```JavaScript
try {

} catch(error){
  console.log(error);
  res.status(500).send(error);
}
```

Next, we will use the `generatePdf` from the `html-pdf-node` library to generate a `pdfBuffer` (the raw PDF file) from our file and create a unique `pdfName`:

```JavaScript
HTMLToPDF.generatePdf(file, options).then((pdfBuffer) => {
  // console.log("PDF Buffer:-", pdfBuffer);
  const pdfName = "./data/speech" + Date.now() + ".pdf";

  // Next code here
}
```

From there, we use the filesystem module to write, read and (yes, finally!) send a response to the client app:

```JavaScript
fs.writeFile(pdfName, pdfBuffer, function (writeError) {
  if (writeError) {
    return res
      .status(500)
      .json({ message: "Unable to write file. Try again." });
  }

  fs.readFile(pdfName, function (readError, readData) {
    if (!readError && readData) {
      // console.log({ readData });
      res.setHeader("Content-Type", "application/pdf");
      res.setHeader("Content-Disposition", "attachment");
      res.send(readData);
      return;
    }

    return res
      .status(500)
      .json({ message: "Unable to write file. Try again." });
  });
});
```

Let's break that down a bit:

- The `writeFile` filesystem module accepts a file name, data and a callback function that can returns an error message if there's an issue writing to the file. If you're working with a CDN that provides error endpoints, you could use those instead.
- The `readFile` filesystem module accepts a file name and a callback function that is capable or returning a read error as well as the read data. Once we have no read error and the read data is present, we will construct and send a response to the client. Again, this can be replaced with your CDN's endpoints if you have them.
- The `res.setHeader("Content-Type", "application/pdf");` tells the browser that we are sending a PDF file.
- The `res.setHeader("Content-Disposition", "attachment");` tells the browser to make the received data downloadable.

Since the API route ready, we can use it in our app at `http://localhost:4000`. We can the proceed to the client part of our application to complete the `handleConversion` function.

(#aa-handling-the-conversion) Handling the conversion

Before we can start working on a `handleConversion` function, we need to create a state that handles our API requests for loading, error, success, and other messages. We're going use React's `useState (https://reactjs.org/docs/hooks-state.html)` hook to set that up:

```JavaScript
const [response, setResponse] = useState({
  loading: false,
  message: "",
  error: false,
  success: false,
});
```

In the `handleConversion` function, we will check for when the web page has been loaded before running our code and make sure the `div` with the `editable` attribute is not empty:

```JavaScript
if (typeof window !== "undefined") {
const userText = textBodyRef.current.innerText;
  // console.log(textBodyRef.current.innerText);

  if (!userText) {
    alert("Please speak or write some text.");
    return;
  }
}
```

We proceed by wrapping our eventual API request in a `trycatch`, handling any error that may arise, and updating the response state:

```JavaScript
try {

} catch(error){
  setResponse({
    ...response,
    loading: false,
    error: true,
    message:
      "An unexpected error occurred. Text not converted. Please try again",
    success: false,
  });
}
```

Next, we set some values for the response state and also set config for `axios` and make a post request to the server:

```JavaScript
setResponse({
  ...response,
  loading: true,
  message: "",
  error: false,
  success: false,
});
const config = {
  headers: {
    "Content-Type": "application/json",
  },
  responseType: "blob",
};

const res = await axios.post(
  "http://localhost:4000",
  {
    text: textBodyRef.current.innerText,
  },
  config
);
```

Once we have gotten a successful response, we set the response state with the appropriate values and instruct the browser to download the received PDF:

```JavaScript
setResponse({
  ...response,
  loading: false,
  error: false,
  message:
    "Conversion was successful. Your download will start soon...",
  success: true,
});

// convert the received data to a file
const url = window.URL.createObjectURL(new Blob([res.data]));
// create an anchor element
const link = document.createElement("a");
// set the href of the created anchor element
link.href = url;
```

```
// add the download attribute, give the downloaded file a name
link.setAttribute("download", "yourfile.pdf");
// add the created anchor tag to the DOM
document.body.appendChild(link);
// force a click on the link to start a simulated download
link.click();
```

And we can use the following below the contentEditable `div` for displaying messages:

JavaScript

```
<div>
  {response.success && <i className="success">{response.message}</i>}
  {response.error && <i className="error">{response.message}</i>}
</div>
```

(#aa-final-code) Final code

I've packaged everything up on GitHub so you can check out the full source code for both the server and the client.

SERVER REPO (HTTPS://GITHUB.COM/BONARHYME/AUDIO-TO-PDF-SERVER)

CLIENT REPO (HTTPS://GITHUB.COM/BONARHYME/AUDIO-TO-PDF-WEB-CLIENT)