Universidad del País Vasco

Sistemas Web

Memoria Técnica

Phonos

Nicolás Aguado, Asier Contreras, Martín Horsfield y Abraham Casas

26 de noviembre de 2023

Índice

1. Introducción

Phonos es una grabadora y reproductora de audio, del estilo de una aplicación de "notas de voz". Con una interfaz sencilla, permite grabar audios desde el propio navegador y contiene una sincronización para guardar las grabaciones en el servidor.

Figura 1: Vista principal de la aplicación

2. Descripción del proyecto

2.1. Framework y Tecnologías Usadas

Este proyecto está enteramente basado en Node.js. Como servidor de archivos (tanto para el back-end como para el front-end) se ha optado por el framework web *express*[2]. Como gestor de paquetes de node-js el elegido es npm[6].

Para el desarrollo del proyecto, como IDE se ha usado IntelliJ IDEA Ultimate[5]. Aunque éste sea ofrecido como un editor para Java, estamos usando los plugins de *javascript* y *node.js* (ya preinstalados en la versión *Ultimate*) para poder trabajar correctamente en la aplicación. Además, el proyecto se encuentra alojado en github para facilitar el desarrollo colaborativo. Este repositorio tiene acceso a la herramienta *Copilot*, así que también nos hemos podido aprovechar de ello.

2.2. Creación del proyecto

Para crear el proyecto se ha utilizado la estructura facilitada por el framework *express*. Para obtener la estructura de carpetas y la base necesaria han bastado los siguientes comandos:

1. Instalar modulos base (en global)

   ```
   npm install -g -production express
   ```

   ```
   npm install -g -production cross-env
   ```

2. Generar carpetas y estructura

   ```
   express --view=ejs phonos
   ```

3. Instalar el resto de dependencias (locales al proyecto)

   ```
   npm install
   ```

2.3. Organización de Archivos

En cuanto a la estructura de carpetas, analizaremos la creada con el framework express en el entorno de trabajo de IntelliJ IDEA.

En el directorio *.idea* se guarda la configuración privada del IDE de cada desarrollador y en la carpeta *.run* están contenidas las distintas configuraciones de ejecución de la aplicación (estas sí serán públicas y compartidas entre todos los desarrolladores).

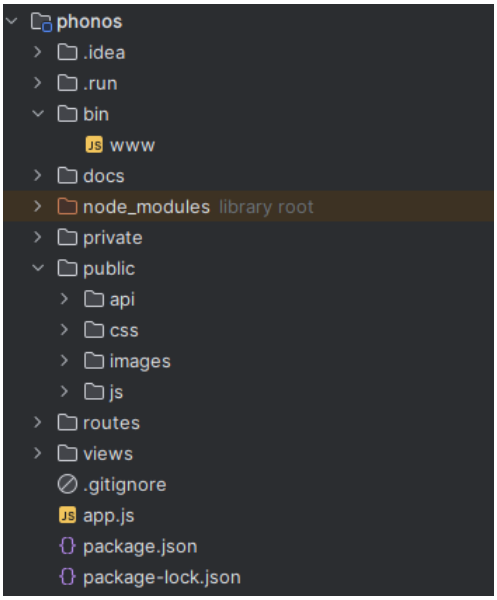

Figura 2: Organización de carpetas

La carpeta *bin* contiene la lógica de ejecución del servidor en sí. Nos interesará separar esto de la parte de front-end y de la de back-end, así que esto simplemente serán asuntos relativos a la ejecución del servidor (puertos, etc...).

En la carpeta *docs* está toda la documentación, y manuales de ayuda y en *node_modules* están instaladas todas las dependencias del lado servidor y del *back-end*.

En la carpeta *private* se ha decidido guardar información útil al equipo de desarrollo pero que no se quiere que transcienda a más. Una especie de archivos compartidos donde por ejemplo en este caso tenemos el archivo juananaudio.js que se trata del ejemplo de grabación de audio facilitado.

En el directorio *public* están las librerías, imágenes y ficheros javascript relativos al *front-end*.

La carpeta *api* está ahí de forma temporal. Será reubicada a la parte del back-end en la siguiente fase del proyecto. En el archivo *js/main.js* es donde está principalmente ubicada la lógica de la aplicación *front-end*.

En el directorio *routes* nos encontramos con las distintas abstracciones y rutas de acceso para posibilitar el tener varios front-ends. Ésta es una carpeta contenido del framework express y por el momento no aprovecharemos su potencial. Contendremos nada más una ruta a nuestra *vista* principal, en la carpeta *views*. Aquí el archivo *index.ejs* será nuestro punto de anclaje desde el que crearemos el HTML.

El archivo *package.json* contendrá información sobre nuestro proyecto (autores, librerías, scripts de ejecución), *.gitignore* prevendrá que se haga *commit* de información no deseada y app.js será el punto de partida de la parte de *back-end* de la aplicación.

2.4. Instrucciones de Desarollo y Ejecución

Para empezar a trabajar, nos hemos de asegurar de tener *node.js* y *npm* instalados. Por si no se encuentran presentes, abrir una terminal y ejecutar:

1. Para instalar *nvm*

   ```
   curl -L cdn.jsdelivr.net/gh/nvm-sh/nvm@0.39.1/install.sh | bash
   ```

2. Para Instalar Node:

   ```
   nvm install node
   ```

El punto de entrada de la aplicación es el archivo *js/main.js*. Las librerías en sí están contenidas en la carpeta *js/utils/*. Para poder ejecutar la aplicación, es necesario tener instalado node.js[7] (junto con npm[6]). Una vez se satisface este requerimiento, se han definido dos *configuraciones de lanzamiento* (Ver figura 3) en IntelliJ para ejecutar la aplicación. Por dentro, éstas simplemente llamarán a los *scripts* que se han definido en el archivo *package.json*. Servirán para lanzar la aplicación en modo *debug* y ya para producción, respectivamente.

package.json

```
1  "scripts": {
2    "rundevel": "cross-env \"DEBUG=nots-voz:*\" node
         ./bin/www",
3    "start": "node ./bin/www"
4  }
5 };
```

3. Front-End

3.1. Estilos y Diseño

En lo que a los estilos se refiere, hemos hecho la página principal utilizando archivos de extensión *.ejs*. En el archivo *index.ejs* se ha realizado un

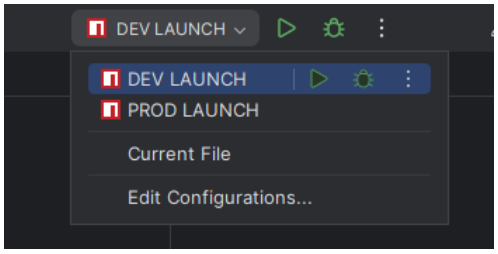

Figura 3: Configuraciones de Lanzamiento en IntelliJ IDEA

diseño base con pocos elementos y las asignaciones de identificadores correspondientes para luego generar el contenido dinámicamente.

Se han definido dos listas (*unordered list*) para poder, por un lado insertar los botones de grabar, reproducir y subir, y por otro lado para poder mostrar la lista de ficheros en el servidor.

En lo que a los estilos se refiere, como se puede apreciar en la figura 2, dentro de la carpeta *public* se encuentra otra carpeta con los archivos de extensión *.css*. En ésta, se han definido diferentes estilos y animaciones para adornar la página principal y conseguir un aspecto más agradable.

3.2. Estructura de Aplicación

Esta aplicación cuenta con una arquitectura principalmente modular y se basa enteramente en la reutilización de código. Se encuentran presentes librerías de utilidades (procedentes de proyectos open-source), librerías de implementación propia para tareas secundarias y librerías *generadoras*. Todas estas trabajarán juntas para que en el archivo principal (*js/main.js*) se puedan llevar acabo las acciones necesarias. Es en éste archivo donde se ha encapsulado toda la lógica relativa a la grabadora en una clase (*App*). Esta clase contendrá una serie de atributos donde se guardará información (estado, audios, etc..) y una serie de métodos para las distintas acciones. Aquí los detallaremos agrupados en 4 grupos para facilitar una mejor comprensión del programa.

3.2.1. Métodos de inicialización

En este grupo encapsularemos a los métodos que se encargan de la inicilización de los diferentes elementos que vayamos a necesitar para gestionar los audios. El método *init()* es el encargado de generar los botones (mediante el uso de métodos que importaremos de otros archivos js) y también de inicializar algunos de los atributos de la clase. Posteriormente, tenemos el método

initAudio() que es el encargado de inicilizar el atributo que contendrá al audio grababo (*audio*) de la clase y agregar *listeners* a sus diferentes eventos. Por último tenemos el *initRecord()*, que es el método que se utiliza para inicializar el mediaRecorder, el objeto js encargado en sí de grabar. Se le añadirán además una serie de listeners para su posterior tratamiento.

3.2.2. Métodos de gestión de los audios

En este grupo tenemos los métodos que gestionan la grabación reproducción, etc.

Para empezar está el método *record()*, que se encarga de poner en marcha el mediaRecorder y de activar un reloj para que se visualicen los segundos. Mientras tanto, gracias a los *listeners* añadidos en la fase de inicialización, se han ido guardando los datos de la voz en *chunks* en el atributo correspondiente.

Entonces, al parar de grabar, el método stopRecording() parará la grabación y juntará todos estos trozos en un *blob*. Después (automáticamente), se ejecutará el método *loadBlob()*, que creará una URL para representar el objeto blob y pasará esa dirección al atributo src del audio para la posterior reproducción.

Por último, los métodos *playAudio()* y *stopAudio()* se encargarán de reproducir y parar el audio (ejecutados por el botón)

3.2.3. Métodos de gestión de botones y estados

En este grupo están los métodos *recordBtn()*, *playBtn()*, *uploadBtn()* y *setState()*. Los tres primeros son los métodos que son llamados por los botones. Básicamente comprueban el estado y llaman a su función correspondiente (en el caso de *playBtn()* llama a *playAudio()* o *stopAudio()* dependiendo del estado, con *recordBtn()* básicamente lo mismo, etc..).

Pero claro, ¿cómo gestionaremos el estado del sistema?. Podremos tener varios estados (grabando, reproduciendo) y habrá muchas funciones que tendrán que definirán este estado.

Entonces, tendremos el método *setState()*. Existirá un JSON que describirá el estado de la aplicación en todo momento y este método será el encargado de añadir el atributo correspondiente al JSON.

Así, si se empieza a reproducir una grabación, el atributo *state* tendrá *state.playing* como valor *true*.

Inicialización del atributo App.state

```
1 {
```

```
    "recording":false,
    "uploading":false,
    "audioloaded":false,
    "playing":false,
    "files": [],
    "error":false
}
```

3.2.4. Métodos de gestión de ficheros

En este último grupo agruparemos los métodos *upload()* (función encargada de subir los audios grabados al servidor), *copyToClipboard(filename)* (recibirá como parámetro el nombre del fichero y copiará la URL del mismo en el portapapeles) y el método *deleteFile(id)* (que recibiendo una id por parámetro borrará el audio en cuestión del servidor).

3.3. Flow Típico de Aplicación

Al cargar la página, el usuario se encontrará con dos elementos a destacar. El primero, la reproductora y sus botones; y el segundo, el navegador pidiendo permisos de grabación con el micrófono.

El usuario tiene la opción de rechazar la grabación con su micrófono. En ese caso, quedarán escondidos los botones de grabar y subir. Solo podrá descargarse grabaciones del servidor y reproducirlas.

Si el usuario acepta el uso del micrófono, se encontrará con todos los botones visibles. El usuario puede grabar un audio de una duración máxima de 5 minutos. Tras terminar la grabación, podrá reproducir la grabación y subirla al servidor. Si se ha finalizado la grabación o la reproducción, se podrá descargar audios del servidor y reproducirlos.

Debajo de los botones principales aparecerán todas las grabaciones almacenadas en el servidor, cada una de ellas con dos botones. El botón de la izquierda copiará el enlace al audio y el botón de la izquierda enviará una petición al servidor para borrar el fichero.

3.4. Problemas Encontrados

Realizando el proyecto, nos hemos encontrado con varios problemas que hemos tenido que resolver. Al igual que con cualquier otro trabajo de programación, los *bugs* siempre van a aparecer (código repetido, llamadas en lugares incorrectos, etc...).

Uno de los problemas ha destacar ha sido la gestión de las librerías a importar. Hay muchas maneras de reutilizar código, y cada desarrollador de javascript exporta sus librerías como quiere / puede. Entonces, hay liberías que son módulos, otras que son paquetes, otras que se pueden importar directamente desde javascript y en otras habrá que hacerlo desde el propio html. Librerías que incluyan css y librerías que no sean directamente compatibles.

Por otro lado, muchas veces, los fallos de la aplicación suelen ser generados por los propios programadores y pueden resolverse. Pero, esta vez ha surgido un problema externo que no dependía de nosotros.

Desde 2015 [1], *Chromium* lleva teniendo un problema calculando la duración de una grabación en el objeto *audio*. El bug todavía no se ha resuelto, y desde *stackoverflow* [3], se recomiendan algunas soluciones como sustituir el valor *Infinity* por valores muy grandes. Estas soluciones no funcionan del todo bien.

Puede resultar extraño que este fallo no se encuentre arreglado, ya que son muchas las aplicaciones que usan el micrófono y esta es una funcionalidad clave. Entonces, investigando en la documentación oficial, encotramos que el *audio* (HTMLAudioElement) se encuentra 'mínimamente' implementado en los navegadores. En verdad lo recomendado por la bibliografía es usar un elemento (de más bajo nivel) de tipo *AudioContext*, cuya implementación tiene una base sólida y permite muchas más modificaciones. Entonces, hemos decidido refactorizar la aplicación para usar éste elemento.

3.5. Bonus Points

3.5.1. Refactorización de HTMLAudioElement a AudioContext

Para usar el elemento *AudioContext* en lugar de *HTMLAudioElement* como reproductor de audio hay que realizar unas mínimas modificaciones y tener claro el contexto en el que se está operando.

En lugar de tener un único atributo `audio` del que se podrán utilizar todas las acciones de manera intuitiva, se tendrán 3 atributos:

1. `audioContext`: Será el encargado de gestionar el contexto del reproductor de audio. Se creará una vez y los otros dos atributos se generarán a partir de éste. Proporcionara la interfaz por la que se *escucha* en sí. Este atributo reproducirá todo lo que le llegue a *audioContext.destination*

2. `audioBuffer`: Éste tendrá los datos de audio a reproducir. Cuando el blob esté disponible, éste es el atributo que se modificará para establecer el audio base.

3. `audioSource`: Nodo inicio en el que se cargará el buffer de audio a reproducir (`audioBuffer`). Como será el nodo de inicio, será el encargado de reproducir y parar el audio. Tendrá que ser creado *cada vez* que se desee reproducir el audio. A este nodo habrá que conectarlo con el contexto destino de audio (*audioContext.destination)* mediante un grafo para que el usuario lo escuche.

3.5.2. Efectos de Audio

Ya que el elemento `audioSource` utiliza un grafo para calcular el audio destino, se nos ha ocurrido añadir la posibilidad de añadir *efectos* al reproductor de audio.

Esto es un mundo aparte, pero gracias a una página web [8] creada por un ingeniero de Google hemos podido conseguir las *respuestas impulse* necesarias para añadir un nodo que haga de filtro al grafo que calcula el audio destino.

Además, hay proyectos 'puristas' de audio que ofrecen código javascript para crear nodos de filtros usando la interfaz de audio web modificando las tonalidades y demás atributos de audio directamete. Entonces, nos ha parecido bien incluir algun filtro de este estilo [4] en el proyecto.

Como resumen, después de aplicar un filtro al audio base (`audioBuffer`), el grafo que calcula el audio que escucha el usuario final sería algo parecido a la figura 4. Si no se le aplica ningún filtro sería un grafo sin modificadores, sencillamente con el elemento fuente apuntando al elemento destino.

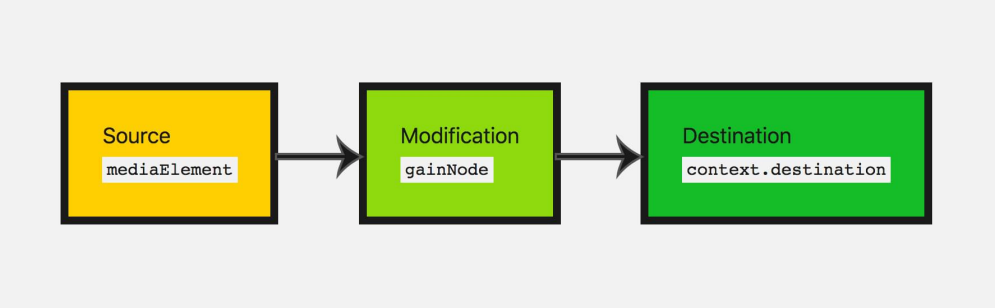

Figura 4: Un grafo destino de ejemplo, usando `gainNode` como modificador y `mediaElement` como fuente inicial

Referencias

[1] Cannot seek when video.duration set to Infinity. https://bugs.chromium.org/p/chromium/issues/detail?id=461733, [Online; accessed 22-November-2023]

[2] Express Web Application Framework. https://expressjs.com/, [Online; accessed 21-November-2023]

[3] Stack Overflow - JavaScript - Audio duration always set to infinity. https://stackoverflow.com/a/52375280, [Online; accessed 23-November-2023]

[4] Denton, Z.: Custom Audio Effects in JavaScript with the Web Audio API. https://noisehack.com/custom-audio-effects-javascript-web-audio-api/, [Online; accessed 26-November-2023]

[5] JetBrains: IntelliJ IDEA. https://www.jetbrains.com/idea/, [Online; accessed 21-November-2023]

[6] npmInc: node package manager. https://www.npmjs.com/, [Online; accessed 21-November-2023]

[7] OpenJSFoundation: Node-JS - Cross Platform Javascript Runtime. https://nodejs.org/en, [Online; accessed 21-November-2023]

[8] Smush, B.: Room Effects - Web Audio API. https://webaudioapi.com/samples/room-effects/, [Online; accessed 26-November-2023]