



TRABAJO FIN DE GRADO
INGENIERÍA EN INGENIERIA INFORMÁTICA

Sintetizador virtual

Autor

Miguel García Tenorio

Directores

Carlos Ureña Almagro



Escuela Técnica Superior de Ingenierías Informática y de
Telecomunicación

—
Granada, Julio de 2021



ugr

Universidad
de Granada

Sintetizador Virtual

Autor

Miguel García Tenorio

Directores

Carlos Ureña Almagro

Sintetizador virtual

Miguel García Tenorio

Palabras clave: Aplicación web, sintetizador, arquitectura en capas, síntesis digital, MIDI, Web Audio API, API REST

Resumen

El desarrollo de este Trabajo de Fin de Grado (TFG) tiene como principal objetivo la creación de una aplicación web capaz que permita generar y reproducir sonidos de manera virtual emulando el comportamiento de un sintetizador analógico. Para ello, este documento contendrá los procesos de análisis, diseño e implantación empleados en la creación de la aplicación, así como la metodología de trabajo empleada durante este proceso.

La aplicación web estará diseñada entorno a arquitectura en capas, en la que tendremos una capa de datos, una de servicios y otra de interfaz, reduciendo el acoplamiento entre los componentes del sistema.

El usuario podrá acceder a una interfaz reactiva desde su navegador web con la que podrá interaccionar para generar y reproducir sonidos de manera totalmente digital, sencilla e intuitiva. El usuario, no solo podrá generar sonidos y reproducirlos, sino que también podrá visualizarlos de manera gráfica, aplicar efectos sobre ellos, grabar su reproducción, almacenarlos, eliminarlos o cargarlos en la aplicación. Además, la aplicación permitirá la comunicación con hardware externo a la computadora del usuario de tipo MIDI, permitiendo al usuario reproducir los sonidos mediante un teclado MIDI.

Para que la aplicación web desarrollada logre la síntesis digital necesitada, se hará uso de la tecnología JavaScript, y en especial de la Web Audio API, la cual será detallada a lo largo de esta memoria. Además, se utilizará NodeJS como entorno de ejecución en el sistema para integrar la tecnología JavaScript y lograr una interfaz reactiva también basada en esta tecnología.

Los datos que maneje la aplicación serán almacenados en una base de datos no relacional, que permitirá dotar de flexibilidad al modelo de datos requerido en la aplicación. Dichos datos podrán ser accedidos y alterados mediante la creación de una API REST que manejará las distintas peticiones en los distintos endpoints. Sin embargo, la manera en la que la aplicación accede y altera el modelo de datos es totalmente ajena a los usuarios que percibirán el sistema como un TODO a través de la interfaz.

Para lograr el desarrollo de la aplicación web se han utilizado todos los conocimientos adquiridos a lo largo del grado, además de una labor de trabajo autónomo e investigación. También, el uso de una metodología de trabajo ágil ha permitido hacer frente a los problemas surgidos durante la implementación de una manera ágil, gracias a una planificación dinámica y adaptada a cada situación.

Sintetizador virtual

Miguel García Tenorio

Keywords: web application, synthesizer, layered architecture, digital synthesis , MIDI, Web Audio API, API REST

Abstract

The main objective of the development of this Final Degree Project (TFG) is to create a capable web application that allows the generation and reproduction of sounds in a virtual way, emulating the behavior of an analog synthesizer. For this, this document will contain the analysis, design and implementation processes used in the creation of the application, as well as the work methodology used during this process.

The web application will be designed around a layered architecture, in which we will have a data layer, a services layer and an interface layer, reducing the coupling between the system components.

Users will be able to access a reactive interface from their web browser with which they can interact to generate and reproduce sounds in a totally digital, simple and intuitive way. The user will not only be able to generate sounds and reproduce them, but will also be able to visualize them graphically, apply effects on them, record their reproduction, store them, delete them or load them into the application. In addition, the application will allow communication with hardware external to the user's computer of MIDI type, allowing the user to reproduce the sounds through a MIDI keyboard.

For the developed web application to achieve the digital synthesis needed, use will be made of JavaScript technology, and especially the Web Audio API, which will be detailed throughout this report. In addition, NodeJS will be used as an execution environment in the system to integrate JavaScript technology and achieve a reactive interface also based on this technology.

The data handled by the application will be stored in a non-relational database, which will provide flexibility to the data model required in the application. Said data can be accessed and altered by creating a REST API that will handle the different requests on the different endpoints. However, the way in which the application accesses and alters the data model is totally alien to the users who will perceive the system as a WHOLE through the interface.

To achieve the development of the web application, all the knowledge acquired throughout the degree has been used, in addition to autonomous work and research. Also, the use of an agile work methodology has made it possible to face the problems that arose during the implementation in an agile way, thanks to a dynamic planning adapted to each situation.

Yo, **Miguel García Tenorio**, alumno de la titulación Grado en Ingeniería Informática de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 75576490P, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Miguel García Tenorio

Granada a 4 de Julio de 2021.

D. **Carlos Ureña Almagro** Profesor del Área Informática
Gráfica y Sistemas con Concurrentes y Distribuidos del
Departamento de Lenguajes y Sistemas Informáticos de la
Universidad de Granada.

Informan:

Que el presente trabajo, titulado ***Sintetizador Virtual***, ha sido realizado
bajo su supervisión por **Miguel García Tenorio**, y autorizamos la
defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a
X de mes de 201 .

Los directores:

Carlos Ureña Almagro

Agradecimientos

A mi familia por estar siempre a mi lado.

Índice

Capítulo 1: Introducción	18
1.1. Sintetizador	18
1.2. Sintetizador digital	19
1.2. Glosario de términos	20
Capítulo 2: Objetivos.....	21
Capítulo 3: Planificación y presupuesto	21
3.1. Planificación Inicial	21
3.2. Presupuesto	24
3.2.1. Justificación del presupuesto	24
Capítulo 4: Análisis	25
4.1. Metodología de desarrollo	25
4.1.1. SCRUM	25
4.1.2. Componentes Scrum	25
4.1.3. Adaptación de la metodología.....	26
4.1.2. Velocidad de Trabajo	26
4.2. Análisis del entorno	27
4.2.1. Análisis competitivo	27
4.2.2. Personas y escenarios.....	28
4.2.3. Aspectos a tratar	32
4.3. Requisitos del sistema.....	33
4.3.1. Requisitos Funcionales	33
4.3.2. Requisitos no funcionales	36
4.4. Product Backlog	37
4.5. Sprints Backlogs.....	38
4.5.1. Sprint 1	38
4.5.2. Sprint 2	40
4.5.3. Sprint 3	42
4.5.4. Sprint 4	46
4.5.5. Sprint 5	48
4.5.6. Sprint 6	50
4.5.7. Sprint 7	52
4.5.8. Sprint 8	54
4.5.9. Sprint 9	56
Capítulo 5: Diseño	57
5.1. Arquitectura.....	57

5.1.1. Arquitectura n-capas	57
5.2. Diseño Capa de Datos	60
5.2.1. State Model.....	61
5.2.2. User Model	62
5.2.3. Notes Model.....	62
5.3. Diseño Capa Back-end.....	63
5.3.1. Protocolo REST [11]	63
5.3.2. Arquitectura de la capa	69
5.4. Diseño Capa Front-end	70
5.4.1. Arquitectura de la capa	70
5.4.2. Patrón de diseño.....	71
5.4.3. Diagramas de actividad.....	72
5.4.3. Diagrama de paquetes.....	76
5.4.4. Diagrama de clases	76
5.4.5. Wireframes	79
Capítulo 6: Implementación	84
6.1. Estudio de las tecnologías.....	84
6.1.1. Análisis de las diferentes tecnologías	84
6.1.2. Conclusión del análisis de tecnologías.....	85
6.2. Tecnologías empleadas	86
6.2.1. Herramientas	86
6.2.2. Lenguajes de programación.....	87
6.2.3. Entorno de Ejecución [15].....	88
6.2.4. Frameworks	88
6.2.5. Web Audio API [19].....	90
6.2.6. Base de datos [9]	91
6.2.7. Control de versiones	91
6.3. Instalación y ejecución	92
6.3.1. Instalación.....	92
6.3.2. Ejecución	93
6.4. Implementación	93
6.4.1. Frontend	93
6.4.2. Backend.....	164
Capítulo 7: Conclusiones y vías futuras	174
Capítulo 8: Bibliografía final	175

Capítulo 1: Introducción

Hoy en día la música rodea nuestras vidas. Con el auge de las nuevas tecnologías y la aparición de plataformas streaming como Spotify o plataformas como Youtube es posible reproducir música de manera muy sencilla e inmediata mediante un solo click. Esta facilidad de acceso está generando un aumento exponencial de la demanda musical.

De acuerdo con un estudio de la *Federación Internacional de la Industria Fonográfica* realizado en el año 2020, solo en España las personas escuchan de media 56 canciones diarias por persona. Esto se traduce en unos beneficios generados por la industria musical de 81.900 millones de € anuales solo en la Unión Europea. [1]

Esta gran demanda va estrechamente unida a la forma en que la música es creada, ya que ha tenido que adaptarse a la época a la que nos encontramos. Al consumirse tanta música, el proceso de creación de esta se ha acelerado para responder a este volumen de consumo. Tradicionalmente el proceso de creación y producción musical comprendía grabar los distintos instrumentos utilizados y la voz del cantante en un estudio de grabación mediante equipos analógicos, cosa que requería un largo periodo de tiempo. Gracias a esta revolución digital en la que nos encontramos, este proceso se ha ido digitalizando y agilizando de manera que es posible crear música mediante un software ejecutándose en un ordenador en cuestión de minutos u horas.

El avance tecnológico que ha habido en las últimas décadas ha afectado también a los instrumentos utilizados en las composiciones musicales. En la década de los 60 surgieron unos nuevos instrumentos, los denominados sintetizadores analógicos. Estos instrumentos con aspecto de un “*piano con botones y clavijas*”, eran capaces de generar sonidos electrónicamente mediante el uso de circuitos analógicos. Estos instrumentos proporcionaron sonidos nunca antes escuchados en la música que revolucionaron el panorama musical e impulsaron la creación de nuevos géneros musicales como el *House* o el *Synthpop*. Sin embargo, estos instrumentos tienen un precio elevado que no está al alcance de cualquier persona. [2]

La evolución de los ordenadores y del software hizo posible la aparición de sintetizadores digitales, que consistían en programas que emulaban el comportamiento de un sintetizador analógico de manera totalmente digital y sin la necesidad de incorporar circuitos electrónicos adicionales, simplemente mediante cálculos realizados en el ordenador. Esta evolución a lo digital ha permitido que estos instrumentos sean más sofisticados y estén al alcance de cualquier persona, generando la aparición de mucha música.

Peros, ¿qué es en realidad un sintetizador? ¿Cómo es capaz de generar sonidos?

1.1. Sintetizador

Un sintetizador es un instrumento que genera, a través de circuitos, señales eléctricas

que son convertidas en señales audibles. Estos instrumentos ofrecen una serie de controles para poder crear y modificar los sonidos, llegando a poder generar sonidos que imiten a otros instrumentos o generar nuevos sonidos nuevos. Los sintetizadores están formados por tres tipos componentes: [3]

- **Fuente:** Elementos que generan una señal de audio, como los osciladores
- **Modificadores:** Elementos que altera la señal de audio que generan las fuentes, como los filtros y los efectos.
- **Controladores:** Elementos que controlan los parámetros de las fuentes y/o los modificadores, estos pueden ser fijos como un Knob de un controlador MIDI o variables como los generadores de la envolvente

En función de como los distintos elementos del sintetizador están conectados entre sí se obtendrá un tipo de síntesis u otra. Podremos encontrarnos con los siguientes tipos de síntesis: [4]

- **Síntesis aditiva:** Consiste en la superposición o mezcla de ondas simples para construir ondas complejas, en la práctica consiste en combinar varios osciladores para conseguir un vario sonido o timbre
- **Síntesis substractiva:** Es el más común. Se genera una señal de audio producida uno o varios osciladores que poseen diferentes tipos de forma de onda. Después, esta señal es filtrada para substraer las frecuencias que no se quieran en la señal de audio final.
- **Síntesis por modulación:** Consiste en alterar algún parámetro de la señal en razón de otra, para producir señales de audio complejas. Este tipo de síntesis a su vez, se divide en tipos
 - **Síntesis FM:** Consiste en variar la frecuencia de una señal en función de la forma de onda de otra señal.
 - **Síntesis AM:** Consiste en variar la amplitud de una señal en función de la forma de onda de otra señal.
- **Síntesis por modelado físico:** La señal de audio se genera a partir de la simulación en una computadora de las propiedades y parámetros de un instrumento físico ya existente.
- **Síntesis granular:** El sonido es visto de una manera cuántica en la que los elementos mas pequeños son los gránulos. Un granulo es un fragmento sonoro de muy corta duración. Al repetir varios de estos, se funden produciendo sonidos continuos.

Un sintetizador podrá usar un tipo de síntesis o combinar varias dando lugar a sonidos sorprendentes.

La base de generación de las señales de audio se encuentra en los **Osciladores**. Un oscilador generará una onda, que se corresponde con una señal de audio, sin una entrada. Los filtros modificarán la frecuencia de las señales generadas por los osciladores y los efectos modificarán múltiples propiedades de estas como la amplitud o el timbre.

1.2. Sintetizador digital

Un sintetizador digital es una emulación de un sintetizador analógico mediante un software. Para ello, los desarrolladores deberán trasladar todos los elementos expuestos en la sección anterior en un código que al compilarlo y ejecutarlo nos permita realizar ese proceso de síntesis. Comúnmente, estos sintetizadores son usados a través de otros programas para la creación musical, denominados DAWs. Estos programas

integran a los sintetizadores digitales como plugins, normalmente implementados en C++.

Sin embargo, estos sintetizadores diseñados como plugins se quedan en el ámbito de escritorio, ya que se utilizan con programas de escritorio. No obstante, el auge de la computación en la nube esta propiciando que el mundo de la síntesis se traslade a la nube, mediante el uso de tecnologías como JavaScript y la Web Audio API.

1.2. Glosario de términos

Antes de entrar en la materia del trabajo, es necesario saber lo que significan los siguientes conceptos:

- **DAW:** Acrónimo en inglés para Digital Audio Workstation, o en español, estación de trabajo de audio digital. Un DAW es un sistema software dedicado a la grabación y edición de audio que utiliza una interfaz de audio digital para realizar la conversión digital-analógica y viceversa, para que un hardware pueda reproducir el audio digital editado.
- **Amplitud:** Es la fluctuación o desplazamiento de una onda desde su valor medio. Aplicado al sonido, es la cantidad de fuerza o energía de un sonido, conocida como volumen. Su medida son los decibelios (dB).
- **Decibelios (DB):** Sistema de medida logarítmico empleado para comparar el nivel de proporción entre dos valores de presión sonora, voltaje de señal o potencia. En el procesamiento de audio digital se trabaja con el intervalo $[-\infty \text{ db}, 0\text{db}]$ que representan la atenuación en el nivel de la salida del amplificador.
- **Envolvente acústica:** Describe la evolución temporal de la amplitud de cualquier sonido a partir de cuatro parámetros: Attack, Decay, Sustain, Release.
- **Attack:** Es el tiempo de entrada. Lo que tarda en un sonido en alcanzar su amplitud máxima después de haber sido ejecutado el instrumento. No tiene porque ser lineal y se mide en mili segundos (ms).
- **Decay:** Es el tiempo que tarda el sonido en caer a una etapa de sostenimiento, después de haber alcanzado la amplitud máxima sin soltar la tecla pulsada o punto de inducción vibratoria que se pulso para ejecutar el instrumento. No tiene porque ser lineal y se mide en mili segundos (ms).
- **Sustain:** Tiempo durante el cual la amplitud del sonido se mantiene constante hasta que se deja de emitir vibración o se suelta la tecla o el punto de inducción vibratoria. Es lineal y se mide en mili segundos (ms).
- **Release:** Tiempo que tarda un sonido en perder toda su amplitud (volumen) después de despear la tecla o punto de inducción vibratoria. Es lineal y se mide en mili segundos (ms).
- **Polifonía:** Es la capacidad de un instrumento de generar acordes, o en otras palabras de generar distintas frecuencias a la vez.
- **Frecuencia de oscilación:** Es la cantidad de oscilaciones por segundo de una onda electromagnética. Su unidad es el Hz y en música cada frecuencia se corresponde con tonalidades.
- **Oscilación de onda:** Variación, perturbación o fluctuación en el tiempo de una onda.
- **Espectro frecuencial:** Es una distribución de amplitudes para cada frecuencia de un fenómeno ondulatorio como puede ser el sonido.
- **Octava:** Se denomina **octava** al intervalo de ocho grados entre dos notas de la escala **musical**. Físicamente hablando Puede tratarse del intervalo que existe entre un par de sonidos que disponen de frecuencias que mantienen un vínculo de 2-1. Si un sonido tiene una frecuencia fundamental de 2640 Hz, se encontrará

una **octava** más alto que aquel cuya frecuencia es de 1320 Hz.

- **Plugin:** Software de un tercero que se integra en otro software.
- **MIDI:** Siglas para *Musical Instrument Digital Interface*. Es un estándar tecnológico que describe un protocolo e interfaz que permite conectar instrumentos musicales electrónicos con ordenadores y otros dispositivos para que se comuniquen entre sí.

Capítulo 2: Objetivos

Se consideran los siguientes, objetivos que se esperan cumplir durante el desarrollo de esta memoria:

1. Generar y reproducir señales de audio digitales.
2. Simular el comportamiento de un sintetizador analógico de manera digital.
3. Estudiar la web Audio Api de JavaScript.
4. Desarrollar una aplicación web.
5. Llevar a cabo comunicaciones con un servidor.
6. Comunicar una aplicación con dispositivos hardware externos (MIDI).
7. Integrar gráficos dinámicos en la aplicación.

Capítulo 3: Planificación y presupuesto

3.1. Planificación Inicial

Inicialmente se realiza una planificación en la que se opta por dividir el desarrollo del proyecto en 6 fases o iteraciones, las cuales se dividen en 2 Sprint de dos semanas, excepto la primera iteración y la última que contienen un Sprint, en la que se planifican 8 bloques principales que conllevarán una serie de tareas a realizar.

Se realiza esta división, debido a que se seguirá una metodología ágil de desarrollo, Scrum, que será expuesta en secciones posteriores ([ver Sección 4.1](#)).

El proyecto comienza la semana del 15/02/2021 y termina la semana del 5/07/2021, por lo que se planifican 20 semanas en total. La planificación consta de 8 bloques principales. Cabe destacar, que esto se corresponde a la planificación inicial del proyecto. Al comienzo de cada Sprint, se ha realizado una planificación de las historias de usuario correspondientes al Sprint, y en caso de incidencias, se hizo una replanificación. La [figura 3.1](#) muestra la temporización de la realización de cada uno de los bloques de acuerdo a esta división

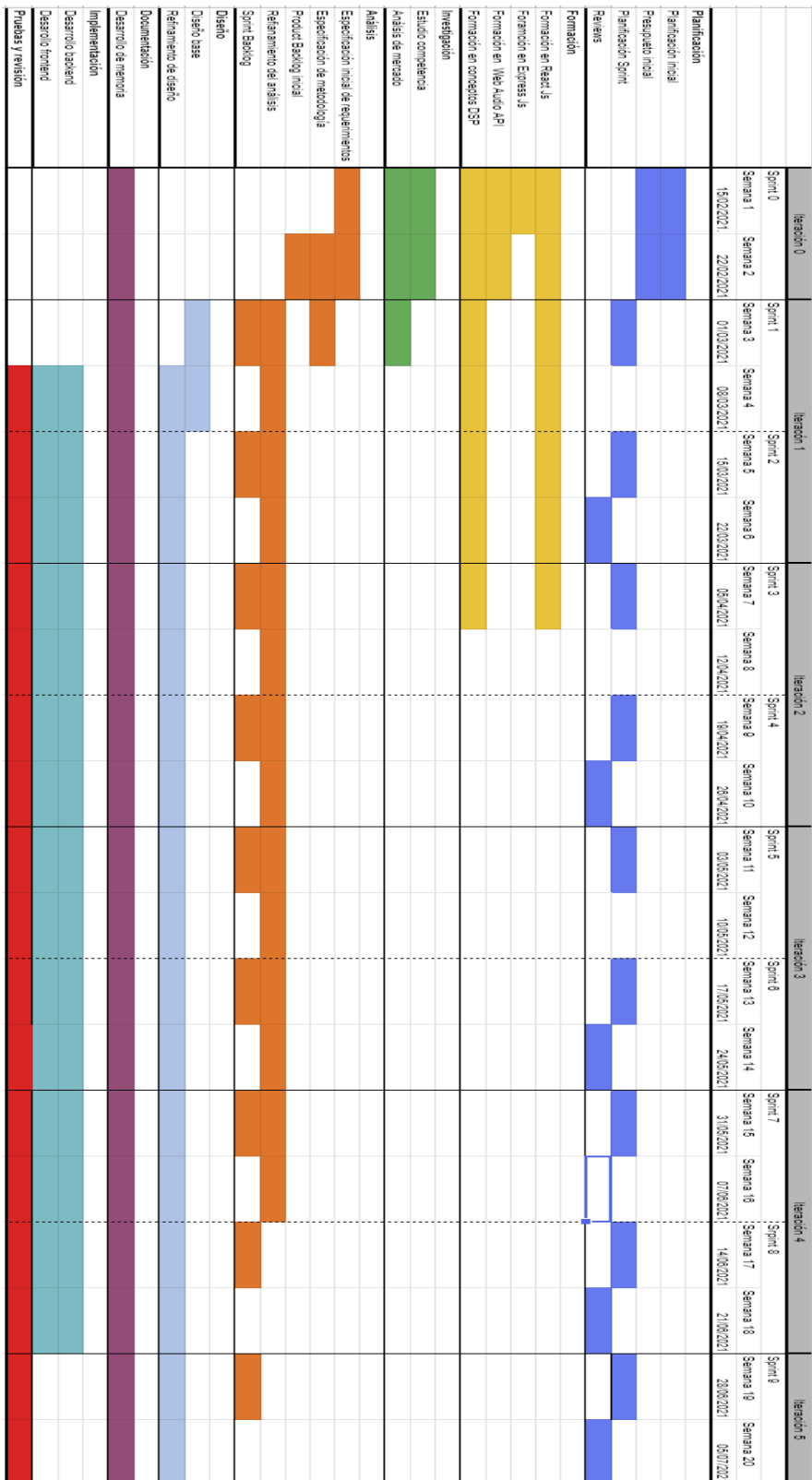


Figura 3.1. Diagrama Gantt planificación inicial

De acuerdo con esta planificación inicial se estiman las horas que será necesarias para la realización de cada uno de los bloques.

De acuerdo con el Plan de Ordenación Docente del curso 2020/201 de la UGR 1 crédito ECTS se corresponde con 25 horas de trabajo, por lo que 12 créditos ECTS del Trabajo Final de Grado se corresponden con 300 horas de trabajo que hay que planificar. En la [tabla 3.1](#) se muestra la correspondencia en horas para cada uno de los bloques planificados.

	HORAS	PORCENTAJE TOTAL
Planificación	22	7,3
Formación	21	7
Investigación	7	2,3
Análisis	40	13,3
Diseño	41	13,6
Documentación	32,5	10,83
Implementación	109	36,3
Revisión y Pruebas	27,5	9,17
TOTAL	300	100

Tabla 3.1. Horas planificadas

De las cuales finalmente se realizó:

	PORCENTAJE TOTAL REALIZADO
Planificación	100%
Formación	100%
Investigación	80%
Análisis	95%
Diseño	100%
Documentación	120%
Implementación	150%
Revisión y Pruebas	70%

Tabla 3.2. Porcentaje de horas realizadas

3.2. Presupuesto

Se estima que el precio del proyecto ronde los 5490€

Elementos	Coste	Total
Ordenador	780 €	780 €
Teclado MIDI	100 €	100 €
Auriculares estudio	60 €	60 €
Espacio de coworking (4 meses)	100 € / mes	400 €
Trabajo autónomo (4 meses)	13,5 € / hora	4050 €
Bibliografía	100 €	100 €
Total		5490 €

Tabla 3.3. Presupuesto del proyecto

3.2.1. Justificación del presupuesto

A continuación, se justifica cada elemento del presupuesto:

- **Ordenador:** Será necesario un ordenador de gama media/alta ya que se requiere de capacidad computacional para ejecutar el entorno de programación, lanzar el servidor para realizar pruebas y ejecutar el programa.
- **Teclado MIDI:** Se requiere un teclado MIDI para probar alguna de la funcionalidad que ofrece el sistema. Basta con un teclado MIDI de gama media/baja para realizar las pruebas que rondan entorno a los 100 €
- **Auriculares de estudio:** Es preciso comprar unos auriculares de estudio de gama media para tener un sonido claro y limpio de los sonidos producidos por el software que se está desarrollando. Este tipo de auriculares rondan los 60€ en el mercado.
- **Espacio de coworking:** Se supone que no se dispone de espacio de trabajo, por lo que se opta por alquilar una plaza en un espacio coworking en la que se dispondrá de electricidad, agua, calefacción, internet y sala de reuniones entre otros. Aproximadamente la tarifa ronda entre los 100 € mensuales.
- **Trabajo autónomo:** Como se planificaron 300 horas de trabajo (ver [Tabla 3.1](#)), y el salario de un Ingeniero informático Junior ronda los 13,5 € la hora y el proyecto tiene una duración de 4 meses, se estiman unos 4050 € de trabajo autónomo.
- **Bibliografía:** Serán necesarias fuentes que no se encuentran en Internet y no se encuentran en bibliotecas públicas (salvo en la de la UGR) que conllevarán un coste adicional.

Capítulo 4: Análisis

4.1. Metodología de desarrollo

Se ha optado por una metodología ágil para el desarrollo de este proyecto. En particular se ha elegido SCRUM como metodología de desarrollo, aunque se ha adaptado al proyecto. A continuación, se explica detalladamente.

4.1.1. SCRUM

Scrum es una metodología ágil de trabajo identificado y definido por Ikujiro Nonaka y Takeuchi en los años 80 al analizar como desarrollaban productos grandes empresas tecnológicas como Canon. Inicialmente se esta metodología se diseño para el desarrollo de manera ágil del software, aunque el éxito de esta metodología ha provocado que se adapte a otras industrias. [5]

Scrum se basa en el respeto a las personas y en la autoorganización de los equipos para hacer frente a las adversidades no previstas surgidas en un proyecto y resolver problemas inspeccionando y adaptando el proyecto continuamente. Esta metodología esta pensada para equipos de trabajo pequeños de 3 a 9 personas, si se sobrepasa este número la agilidad de los equipos podría llegar a desaparecer. Scrum no es reacia a los cambios en un proyecto, ya que define los mecanismos necesarios para hacerles frente. [5]

4.1.2. Componentes Scrum

Scrum se compone de los siguientes componentes: [6]

- **Eventos:** Son las reuniones que existirán en el equipo de trabajo. Están pensados para la continua comunicación y colaboración del equipo y minimizar la necesidad de realizar reuniones no definidas que pudieran afectar a la planificación del proyecto. Hay varios tipos de eventos:
 - **Sprint:** Es un intervalo de tiempo fijo de entre una y cuatro semanas en durante el cual se crean incrementos de valor sobre el producto que estamos desarrollando, de manera que el cliente al final de cada Sprint podrá tener un producto terminado.
 - **Sprint Planning:** Es una reunión que se hace al comienzo de cada sprint para planificar el trabajo que se va a realizar a lo largo de este.
 - **Dailys:** Son reuniones diarias, de poca duración en las los miembros del equipo comentan el trabajo que se realizo el día anterior y el trabajo que se va a realizar en el día de la reunión.
 - **Sprint Review:** Es una reunión en la que cada miembro del equipo muestra el trabajo que ha realizado a lo largo del Sprint y se dice que se deberá cambiar del producto para el Sprint siguiente:
 - **Sprint Retrospective:** Reunión en la que se analiza el trabajo como equipo y se exponen aspectos a mejorar como equipo para el sprint siguiente.
- **Artefactos:** Se utilizan para fomentar la transparencia en la información, de manera que todos los miembros del equipo puedan conocer que se está llevando a cabo mediante los artefactos. Son los siguientes:
 - **Product Backlog:** Contienen todas las historias de usuario sin estimar del proyecto, su gestión es responsabilidad del Product Owner.
 - **Sprint Backlog:** Contiene las historias de usuario estimadas y planificadas

- que se realizarán en un sprint.
- Incremento: Es el producto que se generará cuando termine el sprint.
- **Roles:** Son las funciones que tendrán los miembros del equipo:
 - Product Owner: Representará al cliente. Su función será definir las historias de usuario y validar el producto.
 - Scrum Master: Su función es garantizar que se realice de manera adecuada scrum en el equipo, además será el encargado de dirigir y concretar los distintos eventos.
 - Desarrolladores: Se encargan de llevar a cabo las historias de usuario.
 - Stakeholders: No son miembros del equipo, ya que no están inmersos en la dinámica de trabajo, pero representan una parte fundamental ya que son partes interesadas en el producto.

4.1.3. Adaptación de la metodología

Scrum es una metodología de trabajo pensada para equipos. Sin embargo, este TFG será realizado por una sola persona. Es por esto, que no se utilizará scrum al 100% sino que se utilizará una adaptación de esta metodología durante el desarrollo del proyecto.

El autor de este trabajo, Miguel García Tenorio, representará todos los roles del equipo. Además, existirá un stakeholders, que será el tutor del proyecto, Carlos Ureña Almagro.

En cuanto a los eventos de scrum, se harán Sprint reviews al final de cada iteración grande, en las que se le mostrará al tutor el estado del proyecto. No se realizan al final de cada Sprint con el objetivo de que se reduzcan estas reuniones con el tutor. En cuanto al resto de eventos se seguirá la misma filosofía, pero a nivel personal, es decir, se dividirá el desarrollo en sprint, habrá planificación del sprint, etc...

Finalmente, se creará un producto Backlog y un sprint Backlog para cada sprint, los cuales podrán ser consultados en esta memoria.

4.1.2. Velocidad de Trabajo

Partimos de un equipo de desarrollo formado por 1 programador que va a dedicar un 100% de su trabajo al proyecto.

La duración de cada uno de los sprints que vamos a realizar en el proyecto va a ser de 2 semanas.

La estimación del esfuerzo de cada una de las historias de usuario se ha expresado en días ideales de programación.

Se estima que un día ideal de programación se va a corresponder con 2 a 3 días reales de trabajo.

La duración de Sprint va a ser: 1 Sprint = 2 semanas = 12 Días reales

La Velocidad del equipo de desarrollo medido en punto de historia es: 1 Programador * 12 = 12 días reales por iteración => de 10 a 12 PH por iteración.

Se ha decidido usar 12 Puntos de historia como la velocidad estimada del equipo.

4.2. Análisis del entorno

Antes de entrar en el análisis de los requerimientos del sistema o las historias de usuario, es necesario analizar el entorno para conocer cómo se comportan los usuarios al utilizar un sintetizador virtual y lograr un software más adaptado a los usuarios.

4.2.1. Análisis competitivo

En primer lugar, se realiza un análisis de la competencia para encontrar puntos fuertes y débiles de estos sistemas.

Hay que destacar que los programas software de síntesis digital más populares están diseñados para ser extensiones de programas DAW, por lo que muchos de estos no se pueden utilizar sin un programa de este tipo, o bien se pueden utilizar, pero con una funcionalidad limitada. A pesar de esto, podemos encontrar otras opciones que no necesitan de un DAW para su funcionamiento.

De esta manera, podemos clasificar los sintetizadores digitales en dos grandes bloques:

- Aquellos que necesitan de un DAW para su utilización.
- Aquellos que no necesitan un DAW para su utilización.

De acuerdo con esta clasificación, para cada grupo, se analiza el sintetizador más popular:

- **Serum:**
 - Esta dentro del primer grupo (necesita un DAW).
 - Diseñado por la empresa Xfer Records.
 - Implementado en C++ usando JUCE ¹ como framework.
 - Puntos fuertes
 - En cuanto a síntesis digital, es de los más sofisticados en el mercado.
 - Tiene dos osciladores.
 - Interfaz bastante sencilla e intuitiva.
 - Posibilidad de elegir distintos tipos de onda.
 - Ofrece la posibilidad de modificar la onda de audio de cada oscilador a nuestro antojo.
 - Sistema de modulación bastante versátil, permite arrastrar el componente de modulación a lo que se quiera modular e incluso cuenta con una matriz de modulación.
 - Permite guardar configuraciones de sonidos.
 - Posee un banco de configuraciones preestablecidas, es decir, de sonido ya diseñados que se pueden buscar y filtrar por categorías u otros criterios.
 - Cuenta con gran variedad de efectos que se pueden aplicar a la vez a un mismo sonido.
 - En la parte inferior de la interfaz hay una simulación de un piano que nos va indicando que nota estamos pulsando encada momento.
 - Posee gráficos que se sincronizan con el sonido producido.
 - Permite insertar muestras de sonido (simples) para su síntesis.
 - Puntos débiles
 - Solo se puede utilizar a través de un DAW, es decir, es un plugin.

¹ JUCE: Framework basado en C++ para el desarrollo de Plugins para DAWS


- Tiene bastantes parámetros que se pueden modificar por lo que se requiere un conocimiento alto en síntesis digital.
 - Debido a que es un plugin, sus funciones se limitan a crear sonidos y enviarlos al DAW.
 - Elevado coste, actualmente \$189 USD.
- **Midi.city:**
 - Esta dentro del segundo grupo (no necesita un DAW).
 - Es un sintetizador en la nube, que puede usarse en cualquier dispositivo que disponga de un navegador web.
 - Constituye una comunidad, en la que el usuario puede crearse una cuenta para participar en foros, votar para nuevas características, acceder al registro de cambios, reportar algún fallo etc....
 - Principalmente se basa en JavaScript usando ToneJS ²
 - Puntos fuertes:
 - Se puede utilizar en cualquier dispositivo con navegador web.
 - Es open source, su utilización no requiere comprar ninguna licencia.
 - Posee todas las octavas.
 - Posee gráficos que se sincronizan con el sonido producido
 - Los sonidos generados se pueden reproducir, con el teclado, pulsando con el ratón en la nota o conectando un teclado MIDI.
 - Cuenta con un metrónomo.
 - Es capaz de secuenciar archivos MIDI.
 - Podemos reproducir una secuencia de percusión a la vez que reproducimos los sonidos sintetizados.
 - Cuenta con un banco de sonidos bien clasificados.
 - Puntos débiles:
 - No se pueden crear nuevos sonidos, los sonidos se limitan a los que vienen por defecto.
 - No se pueden grabar piezas musicales tocadas con los sonidos sintetizados.
 - No se pueden modificar los parámetros de los sonidos, la síntesis es invisible al usuario se realiza en un segundo plano.
 - En ocasiones presenta latencia.

4.2.2. Personas y escenarios

Para identificar las metas y los puntos débiles de nuestro usuario objetivo, se crean dos personas que interaccionarán con Serum y Midi.city.


² Framework para JavaScript que ofrece soluciones para la síntesis digital.

PERSONA 01

Nombre	Javier Pérez	Foto 
Edad	23	
Sexo	M	
Educación	Graduado en comunicación audiovisual. Master en producción musical	
Contexto de uso		
Cuando	En horario laboral	
Dónde	En el estudio	
Dispositivo	Ordenador de sobremesa	
Misión		
Objetivo	Quiere diseñar nuevos sonidos para sus producciones	
Expectativas	Obtener sonidos de calidad y de manera sencilla	
Motivación		
Urgencia	Lo necesita cuanto antes	
Deseo	Quiere diseñar sonidos nuevos por el mismo	
Actitud hacia la tecnología		
Esta siempre a la última, tiene bastante soltura con el uso de las tecnologías		

PERSONA 02

Nombre	Ava Miller	Foto 
Edad	27	
Sexo	10	
Educación	Graduado en Bellas Artes. Master en artes escénicas	
Contexto de uso		
Cuándo	En su tiempo libre	
Dónde	En casa	
Dispositivo	Portátil/ Móvil	
Misión		
Objetivo	Quiere poder tocar el piano con su nuevo teclado MIDI	
Expectativas	No gastar dinero en software adicional y poder grabar sus piezas	
Motivación		
Urgencia	Está aprendiendo de manera autodidacta a tocar el piano y necesita empezar a tocar cuanto antes	
Deseo	Poner en práctica las habilidades aprendidas al piano y experimentar con variedad de sonidos	
Actitud hacia la tecnología		
Se desenvuelve, pero tampoco es una experta		

ESCENARIO 02		
Nombre persona	Ava Miller	Foto 
Objetivo persona	Tocar el piano, probar sonidos diferentes y grabar piezas musicales	
Escenario		
<p>Ava es chica de 27 nacida en Londres, que trabaja para una compañía española de macro-festivales que incluye espectáculos escénicos en sus festivales realizados por todo el mundo. Debido a la situación de pandemia, la empresa no puede realizar su actividad por lo que Ava se encuentra en un ERTE.</p> <p>Ava es una persona inquieta y curiosa y debido a esta situación ha necesitado buscarse un entretenimiento, el piano. Ava ha comenzado a aprender a tocar el piano de manera autodidacta a través de un teclado MIDI que tenía su hermano.</p> <p>Este tipo de teclados requieren de un software para que se escuche “algo”. Existen muchas alternativas de pago, pero Ava no quiere gastarse dinero ya que la situación económica no es favorable y además necesita algo que pueda usarlo en varios ordenadores ya que no siempre usa el mismo.</p> <p>Investigando por la red, encontró Midi.city. Para poner a prueba lo que está aprendiendo, conectó su teclado al ordenador y comenzó a tocar notas. Se dio cuenta que había una biblioteca de sonidos, así que eligió el piano. Para su sorpresa, a la vez que tocaba podía establecer una percusión que le iba acompañando mientras tocaba, además de poder establecer la velocidad a la que iba, había un metrónomo.</p> <p>Por desgracia, su experiencia no fue satisfactoria al completo, ya que no encontró ningún botón donde poder grabar lo que iba tocando.</p>		

4.2.3. Aspectos a tratar

A partir del análisis competitivo y de la creación y propuesta de personas y escenarios, se establecen los siguientes aspectos que deberían tratarse a la hora de definir la funcionalidad de nuestro sistema ya que van a aportar un valor añadido al producto final:

Interesante/relevante	Criticas
<ul style="list-style-type: none"> • La portabilidad del software es una ventaja competitiva • Existencia de gráficos que acompañen al audio • Uno o más osciladores • Uno o más efectos • Existencia de filtros • Posibilidad de usar un teclado MIDI para reproducir los sonidos generados • Biblioteca de sonidos ya diseñados para poder usar alguno 	<ul style="list-style-type: none"> • Demasiados parámetros hacen compleja la interfaz y el diseño de sonido • Las interfaces no son accesibles y no se adaptan al usuario • El coste de estos programas es muy elevado, no existen muchas opciones open source • Muchos requieren de otros programas de pago para funcionar, hay usuarios que no quieren esto.

y/o modificarlos <ul style="list-style-type: none"> • Opción para guardar aquellos sonidos creados o modificados • Posibilidad de importar sonidos para realizar una síntesis a partir de estos • Grabación de las piezas musicales tocadas • Están disponibles todas las octavas 	<ul style="list-style-type: none"> • Problemas frecuentes relacionados con la latencia
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------

4.3. Requisitos del sistema

A partir del análisis del análisis previo, se especifican los requisitos que debe cumplir el sistema:

4.3.1. Requisitos Funcionales

RF-1. Gestión de las “Fuentes”. El sistema será capaz de generar señales de audio mediante la síntesis digital mediante el uso de osciladores.

RF-1.1. Uso de uno o dos osciladores. El usuario podrá encender y apagar cualquiera de los dos osciladores, permitiendo que los dos estén encendidos simultáneamente.

RF- 1.1.1. Elección del tipo de onda. Para cada oscilador, se podrá elegir entre una onda con forma de seno, triangulo, cuadrado, o diente de sierra.

RF- 1.1.2. Los osciladores generarán todas las frecuencias correspondientes a todas las octavas de un piano.

RF-1.2. Polifonía. El sistema será capaz de generar polifonía, es decir, será capaz de generar la señal de audio correspondiente a un acorde o a una nota por separado.

RF-1.3. Suma de señales. El audio generado por las fuentes será la suma de las señales de audio generadas por las fuentes.

RF-2. Gestión de los “Modificadores”. El sistema podrá alterar la señal de audio que las fuentes generan.

RF-2.1. Aplicación de la envolvente. Para cada uno de los osciladores existirá una envolvente de la amplitud de audio que controlará la evolución temporal de la seña generada.

RF-2.2. Uso de reverberación (Reverb). El usuario podrá aplicar un efecto de Reverberación sobre la señal de audio generada por las fuentes.

RF-2.2.1. Encendido y apagado. Se podrá encender y apagar la reverberación en cada momento.

RF-2.3. Uso de Delay. El usuario podrá aplicar un efecto de Delay sobre la señal de audio generada por las fuentes.

RF-2.3.1. Encendido y apagado. Se podrá encender y apagar el Delay en cada momento.

RF-2.4. Uso de distorsión. El usuario podrá aplicar un efecto de Distorsión que modifique el espectro frecuencial de la señal generada

RF-2.4.1. Encendido y apagado. Se podrá encender y apagar el efecto de distorsión en cada momento.

RF-2.5. Uso de filtros. El usuario podrá aplicar filtros de frecuencia sobre la señal de audio generada por las fuentes.

RF-2.5.1. El usurario podrá aplicar distintos tipos de filtros en función de lo que requiera para modificar el sonido.

RF-2.4. Uso de ecualizador de 5 bandas. El usuario podrá aplicar ecualización de 5 bandas sobre la señal de audio generada por las fuentes.

RF-2.4.1. Encendido y apagado. Se podrá encender y apagar el ecualizador en cada momento.

RF-3. Gestión de los “Controladores”. El sistema permitirá al usuario controlar los modificadores y las fuentes

RF-3.1. Control de la envolvente. Para cada envolvente se podrán modificar los valores de Attack, Decay, Sustain, Release.

RF-3.1.1. Modificar el tiempo de ataque (Attack). El sistema permitirá modificar tiempo de ataque de una envolvente, el tiempo que tarda la señal en alcanzar su máximo valor de amplitud. Este valor se podrá modificar entre los 0 y los 2 ms.

RF-3.1.2. Modificar el tiempo de caída (Decay). El sistema permitirá modificar tiempo de caída de una envolvente, el tiempo que tarda la señal en caer a un valor sostenido después de superar la fase de ataque. Este valor se podrá modificar entre los 0 y los 2 ms.

RF-3.1.3. Modificar el tiempo de sostenimiento (Sustain). El sistema permitirá modificar tiempo de sostenimiento de una envolvente, el tiempo que la señal se mantiene constante después de superar la fase de caída. Este valor se podrá modificar entre los 0 y los 1 ms.

RF-3.1.4. Modificar el tiempo de relajación (Release). El sistema permitirá modificar tiempo de relajación de una envolvente, el tiempo que la señal tarda en perder toda su amplitud. Este valor se podrá modificar entre los 0 y los 5 ms.

RF-3.2. Control del efecto de Reverberación. Para el efecto de reverberación se podrá modificar el Decay, HiCut, LowCut.

RF-3.2.1. Modificar Decay. El sistema permitirá modificar el valor de Decay de la reverberación, es decir el tiempo de duración del efecto.

RF-3.2.2. Modificar HiCut. El sistema permitirá modificar el valor de la frecuencia donde se comienza a filtrar la reverberación. Este valor estará dentro del intervalo de 20k-0hz.

RF-3.2.3. Modificar LowCut. El sistema permitirá modificar el valor de la frecuencia donde se comienza a filtrar la reverberación. Este valor estará dentro del intervalo de 0-20khz.

RF-3.3. Control del efecto de Delay. Para el efecto de Delay se podrá modificar el FeedBack.

RF-3.3.1. Modificar FeedBack. El sistema permitirá modificar la cantidad de FeedBack de efecto Delay, es decir, durante cuánto tiempo tendrá lugar el Delay. Este valor está comprendido entre el 0% y el 100%

RF-3.4. Control del efecto de Distorsión. Para el efecto de Distorsión se podrá modificar la cantidad de distorsión aplicada, es decir, se podrá modificar la curva de distorsión para que el espectro frecuencial de la señal sobre la que se aplica el efecto se modifique más o menos.

RF-3.5. El sistema permitirá modificar la cantidad de efecto que se aplica para cada uno de ellos, es decir, valor de dry/wet. Este valor es proporcional al efecto y está comprendido entre el 0% y el 100%.

RF-3.6. El sistema permitirá alterar el valor de volumen maestro del programa. Este valor oscilará entre los -100 y 0 dB.

RF-3.7. Modificar las bandas del ecualizador. Cada banda podrá sumar o decrementar a la amplitud de dicha banda 25 dB.

RF-3.8. Control oscilador. Para cada oscilador se podrá modificar el Paneo y el volumen.

RF-3.8.1. Modificar el paneo, se podrá establecer cuanto sonido sale por el canal L y cuanto sale por el canal R. Este valor va del -100 a 100 Uds.

RF-3.8.2. Modificar volumen, se podrá se podrá ajustar la cantidad de sonido que se quiera (el volumen del sonido que está produciendo). Este valor está comprendido en el intervalo de -100db-0dB.

RF-4. Gestión de gráficos. El programa pintará de manera gráfica, el espectro frecuencial de la onda que producen los osciladores al tiempo que se reproducen los sonidos

RF-5. Gestión de almacenamiento. El sistema gestionará todo lo relacionado con lo referente a los sonidos y los usuarios de este.

RF-5.1. Guardar un sonido creado o modificado. Se guardarán en base de datos todos los valores que producen la señal de audio final. Además, se podrá asignar un nombre y una categoría, valoración

RF-5.2. Cargar un sonido guardado en base de datos

RF-5.3. Buscar un sonido previamente guardado.

- RF-5.3.1.** Aplicar filtro por categoría o valoración en la búsqueda.
- RF-5.4.** Eliminar un sonido previamente guardado

RF-6. Gestión "MIDI". El sistema será capaz de procesar información MIDI o de control.

RF-6.1. El sistema será capaz de comunicarse con un controlador MIDI.

RF-6.2. Cuando se pulsa una tecla del teclado MIDI, si hay algún oscilador encendido, se produce un sonido correspondiente a una nota, simulando un teclado MIDI. Será posible tocar hasta dos octavas desde el teclado.

RF-7. El sistema permitirá grabar los sonidos producidos y descargarlos en mp3.

RF-8. El sistema permitirá a un usuario registrarse.

RF-9. El sistema permitirá a un usuario hacer login y logout.

RF-10. El sistema permitirá a un usuario registrado modificar su información personal, así como la contraseña.

4.3.2. Requisitos no funcionales

Interfaz

RN-1. Cuando se modifique cualquier valor se debe de mostrar en la interfaz simultáneamente el valor que se es está modificando

RN-2. La interfaz debe ser sencilla, accesible e intuitiva

RN-3. La interfaz debe mostrar un piano

RN-4. Cuando se reproduce un sonido se debe simular que en la interfaz se pulsa la correspondiente tecla del piano

Rendimiento

RN-5. El sistema deberá ocuparse de la sincronización entre gráficos y audio, la latencia no puede ser superior a 1 segundo

RN-6. La latencia entre que se pulsa una tecla del teclado del ordenador, del controlador MIDI o se clica en una tecla de la interfaz, hasta que se reproduce el sonido correspondiente no puede exceder los 700ms.

Disponibilidad

RN-7. El sistema debe poder utilizarse en cualquier dispositivo que disponga de un navegador web

RN-8. El sistema debe estar siempre en ejecución.

Implementación

RN-9. Se deben de usar lenguajes de programación que puedan ser interpretados por

navegadores web

Seguridad

RN-10. Solo podrán acceder al sistema los usuarios dados de alta

RN-11. Cada usuario tendrá un espacio limitado para almacenar sus sonidos

RN-12. Los usuarios logueados deberán estar autorizados para realizar las operaciones

Físicos

RN-13. El sistema requiere de un servidor para almacenar la base de datos y ejecutar tanto el backend como el frontend.

4.4. Product Backlog

A partir de la fase de análisis en la que se especificaron los requerimientos del sistema se realiza un listado priorizado de las historias de usuario que se llevarán a cabo durante el desarrollo del sistema.

La prioridad está expresada entre 1 y 5, siendo 1 las historias más prioritarias y 5 las menos prioritarias. A continuación, se muestra el Product Backlog ordenado por prioridad

ID	Nombre de la historia	Prioridad
1	Como usuario registrado quiero utilizar uno o dos osciladores	1
2	Como usuario registrado quiero elegir el tipo de onda de un oscilador	1
3	Como usuario registrado quiero ajustar el volumen de los osciladores por separado	1
4	Como usuario registrado quiero modificar la envolvente de un oscilador 1	1
5	Como usuario registrado quiero tocar notas clicando en las teclas que aparecen en la pantalla	1
6	Como usuario registrado quiero tocar notas desde el teclado del ordenador	1
7	Como usuario registrado quiero tocar un acorde	1
8	Como usuario registrado quiero aplicar un efecto Reverb	2
9	Como usuario registrado quiero modificar el efecto Reverb	2
10	Como usuario registrado quiero aplicar un efecto Delay	2
11	Como usuario registrado quiero modificar el efecto Delay	2
12	Como usuario registrado quiero controlar el volumen del programa	2
13	Como usuario registrado quiero panear un oscilador	2
14	Como usuario registrado quiero aplicar un filtro	3
15	Como usuario registrado quiero modificar el filtro	3
16	Como usuario registrado quiero aplicar un efecto Distorsión	3
17	Como usuario registrado quiero modificar el efecto de Distorsión	3
18	Como usuario registrado quiero guardar un sonido	3

19	Como usuario registrado quiero buscar un sonido guardado	3
20	Como usuario registrado quiero cargar un sonido	3
21	Como usuario registrado quiero eliminar un sonido guardado	3
22	Como usuario registrado quiero utilizar mi teclado MIDI con el sistema	4
23	Como usuario registrado quiero ver el espectro del sonido que estoy produciendo	4
24	Como usuario registrado quiero utilizar el ecualizador de 5 bandas	4
25	Como usuario registrado quiero grabar una pieza musical	4
26	Como usuario no registrado quiero registrarme	4
27	Como usuario registrado quiero hacer login	4
28	Como usuario registrado quiero hacer logout	4
29	Como usuario registrado quiero modificar mis datos personales	5
30	Como usuario registrado quiero cambiar la contraseña	5

4.5. Sprints Backlogs

A continuación, se dividen las historias de usuario del Product Backlog en los distintos Sprints. Para cada historia se realiza una estimación de los puntos de estas en función del esfuerzo que se estima para la realización de esa historia. Los puntos de historia son valores pertenecientes a la serie de Fibonacci.

4.5.1. Sprint 1

ID	Nombre de la historia	Prioridad	Puntos de historia
1	Como usuario registrado quiero utilizar uno o dos osciladores	1	5
2	Como usuario registrado quiero elegir el tipo de onda de un oscilador	1	1
3	Como usuario registrado quiero ajustar el volumen de los osciladores por separado	1	1
4	Como usuario registrado quiero modificar la envolvente de un oscilador	1	8

Identificador: HU.1	Como usuario registrado quiero utilizar uno o dos osciladores
Descripción: Un usuario registrado en el sistema puede utilizar uno o dos osciladores que generen una señal digital de audio, de manera que pueda encender y apagar los osciladores que quiera en cada momento, así como también pueda utilizar los dos a la vez.	
Estimación: 5	Prioridad: 1
Pruebas de aceptación: <ul style="list-style-type: none"> • Encender un oscilador y obtener un sonido • Encender los dos osciladores a la vez y obtener un sonido como suma de los 	

<p>dos osciladores</p> <ul style="list-style-type: none"> • Apagar los dos osciladores y no obtener ningún sonido
<p>Tareas:</p> <ul style="list-style-type: none"> • Instalar NodeJS, ReactJS y ToneJS • Crear la infraestructura inicial del frontend en React • Crear dos osciladores • Crear una vista para los osciladores • Crear un botón de encendido y apagado en cada oscilador
<p>Observaciones:</p> <p>Se elegirá una onda por defecto para los osciladores y se creará un botón que reproduzca una nota por defecto para realizar las pruebas.</p>

Identificador: HU.2	Como usuario registrado quiero elegir el tipo de onda de un oscilador
<p>Descripción:</p> <p>Un usuario registrado en el sistema puede elegir el tipo de onda de un oscilador entre una onda de tipo seno, seno, triangulo, cuadrado, o diente de sierra.</p>	
Estimación: 1	Prioridad: 1
<p>Pruebas de aceptación:</p> <ul style="list-style-type: none"> • Elegir una onda seno en un oscilador y obtener un sonido senoidal. • Elegir una onda triangular en un oscilador y obtener un sonido triangular. • Elegir una onda cuadrada en un oscilador y obtener un sonido cuadrangular. • Elegir una onda diente de sierra en un oscilador y obtener un sonido de diente de sierra. • Si tengo los dos osciladores encendidos y elijo una onda distinta en cada uno el sonido que obtengo es la mezcla de los dos osciladores. 	
<p>Tareas:</p> <ul style="list-style-type: none"> • Crear en cada oscilador un selector para el tipo de onda • Diseñar algoritmo para la mezcla de dos ondas distintas 	
<p>Observaciones:</p>	

Identificador: HU.3	Como usuario registrado quiero ajustar el volumen de los osciladores por separado
<p>Descripción:</p> <p>Un usuario registrado en el sistema puede ajustar el volumen de salida de cada oscilador entre -100dB y 0, es decir, puede ajustar el nivel de amplitud producido por cada oscilador.</p>	
Estimación: 1	Prioridad: 1

Pruebas de aceptación: <ul style="list-style-type: none"> • Si asigno un valor de volumen a un oscilador superior a 0dB automáticamente se convierte en 0dB. • Si asigno un valor de volumen a un oscilador inferior a -100dB automáticamente se convierte en -100dB. • Si incremento los dB se incrementa el volumen, al contrario, si decremento • Si el volumen está a -100dB no se emite sonido.
Tareas: <ul style="list-style-type: none"> • Crear en la vista de cada oscilador un Knob para el volumen • Diseñar algoritmo para controlar el volumen de los osciladores
Observaciones:

Identificador: HU.3	Como usuario registrado quiero modificar la envolvente de un oscilador
Descripción: Un usuario registrado en el sistema puede modificar la envolvente de un oscilador, es decir, puede modificar los valores Attack, Decay, Sustain, Release.	
Estimación: 8	Prioridad: 1
Pruebas de aceptación: <ul style="list-style-type: none"> • Si asigno un valor de Attack entre 0 y 2ms el inicio del sonido producido por un oscilador se ve modificado • Si asigno un valor de Decay entre 0 y 2ms el tiempo de caída producido por un oscilador se ve modificado. • Si asigno un valor de Decay entre 0 y 1ms el tiempo de sustain producido por un oscilador se ve modificado. • Si asigno un valor de Decay entre 0 y 5ms el tiempo de Release producido por un oscilador se ve modificado. 	
Tareas: <ul style="list-style-type: none"> • Crear una envolvente por defecto. • Crear el algoritmo que para aplicar la envolvente. • Crear en la vista de cada oscilador un Knob para el Attack, Decay, Sustain, Release. 	
Observaciones: La interfaz no permitirá a los usuarios introducir valores inferiores a los establecidos en las pruebas de aceptación.	

4.5.2. Sprint 2

ID	Nombre de la historia	Prioridad	Puntos de historia
5	Como usuario registrado quiero tocar notas clicando en las teclas que aparecen en la pantalla	1	2
6	Como usuario registrado quiero tocar notas desde el teclado del ordenador	1	5
7	Como usuario registrado quiero tocar un acorde	1	5

Identificador: HU.5	Como usuario registrado quiero tocar notas clicando en las teclas que aparecen en la pantalla
Descripción: Un usuario registrado en el sistema puede tocar notas musicales clicando en el piano de la pantalla en sus respectivas notas. Solo podrá tocar por pantalla dos octavas.	
Estimación: 2	Prioridad: 1
Pruebas de aceptación: <ul style="list-style-type: none"> • Si clico en una tecla del piano de la pantalla suena una nota. • Si clico la una tecla del piano suena la nota correspondiente a la del piano. Por ejemplo, si toco Do suena Do. • Si clico en dos notas distintas se aprecia que son notas distintas. • Si cambio de tamaño la ventana el piano se ajusta a esta. 	
Tareas: <ul style="list-style-type: none"> • Implementar método para tocar notas distintas. • Implementar un componente para la vista del piano. • Asociar a cada nota del piano una nota musical. 	
Observaciones: Solo se mostrará por pantalla dos octavas del piano y un botón para subir o bajar de octava.	

Identificador: HU.6	Como usuario registrado quiero tocar notas desde el teclado del ordenador
Descripción: Un usuario registrado en el sistema puede tocar notas musicales pulsando en las teclas del ordenador, de manera que el teclado del ordenador simulara el de un piano. El usuario podrá tocar dos octavas simultáneamente.	
Estimación: 5	Prioridad: 1
Pruebas de aceptación: <ul style="list-style-type: none"> • Cuando toco una tecla suena una nota. • Si toco una tecla que no se corresponde con ninguna nota no se emite ningún sonido. 	

<ul style="list-style-type: none"> • Cuando toco una tecla se indica por pantalla que nota estoy tocando. • Si toco simultáneamente una tecla del teclado y la de la interfaz prevalecerá la del teclado. •
Tareas: <ul style="list-style-type: none"> • Implementar método para hacer coincidir teclas con notas. • Implementar método para que se refleje en pantalla que nota/as estoy tocando al pulsar las teclas. • Implementar manejadores de eventos • Gestionar la simultaneidad de eventos entre el piano de la interfaz y las teclas tocadas. • Crear la infraestructura backend inicial • Crear un modelo en BD para las notas • Crear un endpoint para enviar las notas las notas • Pedir las notas al servidor y realizar un parseo.
Observaciones: Cuando el usuario toca una tecla del teclado y se corresponde con una nota se indica por interfaz que nota/as está tocando.

Identificador: HU.7	Como usuario registrado quiero tocar un acorde
Descripción: Un usuario registrado en el sistema puede tocar un acorde, es decir, podrá tocar un conjunto de notas de manera simultánea y que suene a la vez.	
Estimación: 5	Prioridad: 1
Pruebas de aceptación: <ul style="list-style-type: none"> • Cuando toco una tecla suena una nota. • Cuando toco varias notas suena el acorde correspondiente, es decir, la suma de las notas individuales por separado de manera simultánea. 	
Tareas: <ul style="list-style-type: none"> • Implementar creación de distintas voces • Implementar método para reproducir acordes 	
Observaciones: Cuando el usuario toca una tecla del teclado y se corresponde con una nota se indica por interfaz que nota/as está tocando.	

4.5.3. Sprint 3

ID	Nombre de la historia	Prioridad	Puntos de historia
8	Como usuario registrado quiero aplicar un efecto Reverb	2	3
9	Como usuario registrado quiero modificar el efecto Reverb	2	2
10	Como usuario registrado quiero aplicar un efecto Delay	2	3
11	Como usuario registrado quiero modificar el efecto Delay	2	2
12	Como usuario registrado quiero controlar el volumen del programa	2	2

Identificador: HU.8	Como usuario registrado quiero aplicar un efecto Reverb
Descripción: Un usuario registrado en el sistema puede aplicar un efecto de Reverb o reverberación a los sonidos producidos por los osciladores.	
Estimación: 3	Prioridad: 2
Pruebas de aceptación: <ul style="list-style-type: none"> • Cuando toco una tecla suena el sonido producido por la suma de los osciladores más el Reverb. • Si no tengo encendido el efecto de Reverb el sonido suena limpio. • Se produce una sensación de espacio al aplicarle el efecto 	
Tareas: <ul style="list-style-type: none"> • Crear una clase para gestionar los efectos • Implementar método para aplicar Reverb al conjunto de los dos osciladores. • Implementar un filtro Hicut y Lowpass para aplicarlos al reverb. • Establecer un Reverb por defecto. • Implementar un componente para los efectos en la vista. • Implementar un componente en la vista para alternar entre la vista de efectos y la vista de osciladores. • Implementar un componente dentro del componente de los efectos para el Reverb. • Implementar un botón para encender y apagar el Reverb. 	
Observaciones:	

Identificador: HU.9	Como usuario registrado quiero modificar el efecto Reverb
Descripción: Un usuario registrado en el sistema puede modificar un efecto de Reverb o reverberación. En concreto puede modificar el tiempo de caída o Decay, la frecuencia desde donde empieza a actuar el filtro pasa altos, la frecuencia desde donde empieza a actuar el filtro pasa bajos y la cantidad de efecto que se reproduce.	

Estimación: 2	Prioridad: 2
Pruebas de aceptación: <ul style="list-style-type: none"> • Si incremento el HiPass el Reverb suena menos grave. • Si incremento el LowPass el Reverb suena menos agudo. • Si incremento el Decay el reverb dura más tiempo. 	
Tareas: <ul style="list-style-type: none"> • Gestionar los cambios en los parámetros del efecto • Completar el componente del Reverb para poder modificar sus parámetros desde la vista • Implementar un botón para encender y apagar el Reverb. • Si disminuyo el nivel de wet el efecto se percibe en menor cantidad. 	
Observaciones:	

Identificador: HU.10	Como usuario registrado quiero aplicar un efecto Delay
Descripción: Un usuario registrado en el sistema puede aplicar un efecto de Delay o retardo a los sonidos producidos por los osciladores.	
Estimación: 3	Prioridad: 2
Pruebas de aceptación: <ul style="list-style-type: none"> • Si está encendido el Delay, cuando reproduzco una nota el sonido que produce se repite durante un periodo pequeño de tiempo • Si está apagado, cuando reproduzco una nota esta suena limpia sin repetición alguna 	
Tareas: <ul style="list-style-type: none"> • Implementar un método para aplicar el efecto de Delay • Implementar un componente dentro de los componentes de los efectos para el Delay. • Implementar un botón para encender y apagar el Delay. 	
Observaciones:	

Identificador: HU.11	Como usuario registrado quiero modificar el efecto Delay
Descripción: Un usuario registrado en el sistema puede modificar un efecto Delay. En concreto podrá establecer el tiempo de retardo, el feedback o tiempo que tardará en perder toda la amplitud de manera progresiva el efecto y la cantidad de efecto que se va a	

reproducir	
Estimación: 3	Prioridad: 2
Pruebas de aceptación: <ul style="list-style-type: none"> • Si modifico el tiempo, el tiempo en que tarda en repetirse la señal cambia. • Si establezco un feedback alto el efecto de Delay dura más que si establezco uno bajo. • Si establezco un feedback de 0 no se aplica el efecto. • Si disminuyo el nivel de wet el efecto se percibe en menor cantidad. 	
Tareas: <ul style="list-style-type: none"> • Gestionar los cambios en los parámetros del efecto • Completar el componente del Delay de la interfaz con sus respectivos parámetros. 	
Observaciones:	

Identificador: HU.12	Como usuario registrado quiero controlar el volumen del programa
Descripción: Un usuario registrado en el sistema puede controlar el volumen general de todos los componentes que producen una señal mediante un solo Knob de volumen.	
Estimación: 2	Prioridad: 2
Pruebas de aceptación: <ul style="list-style-type: none"> • Si aumento el volumen general se aprecia un incremento del volumen general • Si disminuyo el volumen general se aprecia una disminución del volumen general. • Si establezco un volumen de 0 no se reproduce nada en los altavoces. 	
Tareas: <ul style="list-style-type: none"> • Conectar todos los nodos de ganancia con un nodo maestro • Establecer métodos que gestionen el control del volumen. • Introducir un Knob de volumen en el componente de la cabecera de la vista 	
Observaciones:	

4.5.4. Sprint 4

ID	Nombre de la historia	Prioridad	Puntos de historia
13	Como usuario registrado quiero panear un oscilador	2	3
14	Como usuario registrado quiero aplicar un filtro	3	3
15	Como usuario registrado quiero modificar el filtro	3	2
16	Como usuario registrado quiero aplicar un efecto Distorsión	3	2
17	Como usuario registrado quiero modificar el efecto de Distorsión	3	2

Identificador: HU.13	Como usuario registrado quiero panear un oscilador
Descripción: Un usuario registrado en el sistema puede panear cada oscilador por separado, es decir, puede establecer cuanta señal desea que salga por un lado u otro de los altavoces	
Estimación: 3	Prioridad: 2
Pruebas de aceptación: <ul style="list-style-type: none"> • A medida que paneo un oscilador a la izquierda este suena en más proporción el lado izquierdo de los altavoces, de igual forma en la derecha. • Si paneo al 100% hacia la izquierda todo sonido producido se reproduce en lado izquierdo del sistema de reproducción, al igual en la derecha • Si no paneo hacia ningún lado la señal se reproduce en ambos lados. 	
Tareas: <ul style="list-style-type: none"> • Establecer un método para aplicar un paneo sobre las ganancias de los osciladores • Introducir un componente en la vista de los osciladores para controlar el paneo. 	
Observaciones: Este paneo es más perceptible en sistemas de reproducción de sonido con dos canales de salida, es decir, que cuentan con más de un altavoz para la reproducción del sonido.	

Identificador: HU.14	Como usuario registrado quiero aplicar un filtro
Descripción: Un usuario registrado en el sistema puede aplicar un tipo de filtro sobre la señal de audio generada por los osciladores.	
Estimación: 3	Prioridad: 3
Pruebas de aceptación:	

<ul style="list-style-type: none"> • Si enciendo el filtro la señal de audio difiere de la original.
Tareas: <ul style="list-style-type: none"> • Establecer los tipos de filtro • Implementar un método para aplicar los filtros sobre la señal • Implementar un componente en la vista de los efectos para el filtro
Observaciones: Los tipos de filtros serán: "lowpass", "highpass", "lowshelf", "bandpass", "highshelf", "peaking", "notch".

Identificador: HU.15	Como usuario registrado quiero modificar el filtro
Descripción: Un usuario registrado en el sistema puede modificar un filtro. En concreto puede modificar el tipo de filtro que se va aplicar, la frecuencia desde la que este empieza a actuar y la cantidad de efecto aplicada	
Estimación: 2	Prioridad: 3
Pruebas de aceptación: <ul style="list-style-type: none"> • Si cambio el tipo de filtro se aprecia una diferencia en el sonido resultante • Si modifico la frecuencia el sonido resultante se modifica en consecuencia. • Si decremento la cantidad de efecto aplicada, el filtro se aprecia menos, al contrario, si la incremento 	
Tareas: <ul style="list-style-type: none"> • Gestionar el cambio en los parámetros • Completar la vista del filtro para poder controlar todos sus parámetros y alternar entre tipos de filtros. 	
Observaciones:	

Identificador: HU.16	Como usuario registrado quiero aplicar un efecto Distorsión
Descripción: Un usuario registrado en el sistema puede aplicar un efecto de Distorsión que modifique el espectro frecuencial de la señal de audio generada por los osciladores.	
Estimación: 2	Prioridad: 3
Pruebas de aceptación: <ul style="list-style-type: none"> • Si enciendo el efecto se aprecia la distorsión aplicada. 	
Tareas:	

<ul style="list-style-type: none"> • Implementar un método para aplicar un efecto de Distorsión • Implementar un componente en la vista de efectos para la Distorsión
Observaciones:

Identificador: HU.17	Como usuario registrado quiero modificar el efecto de Distorsión
Descripción: Un usuario registrado en el sistema puede modificar el efecto de Distorsión. En concreto, puede modificar la curva de distorsión o nivel de distorsión y la cantidad de efecto aplicada.	
Estimación: 2	Prioridad: 3
Pruebas de aceptación: <ul style="list-style-type: none"> • Si incremento la curva de distorsión, el sonido suena más duro o distorsionado. • Si decremento la curva de distorsión, el sonido suena más suave • Si decremento la cantidad de efecto aplicada, la distorsión se aprecia menos, al contrario, si la incremento. 	
Tareas: <ul style="list-style-type: none"> • Gestionar los cambios en los parámetros • Completar el componente de la vista relativo a la distorsión, con sus respectivos parámetros. 	
Observaciones:	

4.5.5. Sprint 5

ID	Nombre de la historia	Prioridad	Puntos de historia
18	Como usuario registrado quiero guardar un sonido	3	5
19	Como usuario registrado quiero buscar un sonido guardado	3	3
20	Como usuario registrado quiero cargar un sonido	3	5

Identificador: HU.18	Como usuario registrado quiero guardar un sonido
Descripción: Como usuario registrado quiero guardar una configuración correspondiente a un sonido, dándole obligatoriamente un nombre y una categoría, y opcionalmente valoración y descripción. Cada configuración guardada contiene todos los parámetros que intervienen en la generación del sonido final, tanto los parámetros de los osciladores, como el de los efectos	
Estimación: 5	Prioridad: 3
Pruebas de aceptación:	

<ul style="list-style-type: none"> • Si intento guardar un sonido sin nombre me devuelve mensaje de error. • Si intento guardar un sonido con el mismo nombre que otro me devuelve un mensaje de error. • Si intento guardar un sonido con un nombre nuevo me devuelve un mensaje de éxito.
Tareas: <ul style="list-style-type: none"> • Crear los modelos necesarios para la configuración de un sonido en el backend • Crear un end-point en el backend que responda a una petición de guardado • Crear las consultas necesarias para guardar el sonido en base de datos • Solicitar desde el frontend la operación y parsear la respuesta • Implementar una vista para realizar la petición de guardado • Interpretar la respuesta del backend en el frontend
Observaciones: Las categorías serán las siguientes: PAD, BELL, LEAD, PLUCK, KEYS, BASS, PERSCUSION. La valoración será un número del 1 al 5.

Identificador: HU.19	Como usuario registrado quiero buscar un sonido guardado
Descripción: Como usuario registrado quiero buscar un sonido guardado por su nombre. Para ello primero debo consultar los sonidos que tengo almacenados en una tabla y luego realizar una búsqueda sobre esta. Además, en esta búsqueda es posible aplicar filtros sobre la categoría y la valoración.	
Estimación: 3	Prioridad: 3
Pruebas de aceptación: <ul style="list-style-type: none"> • A medida que voy introduciendo caracteres en el buscador, en la tabla solo me aparecen los sonidos cuyo nombre va coincidiendo con los caracteres. • Si aplico los filtros solo me aparecerán los sonidos que coincidan con los valores de los filtros. • Si tengo los filtros activos, A medida que voy introduciendo caracteres en el buscador, en la tabla solo me aparecen los sonidos cuyo nombre va coincidiendo con los caracteres y que se ajustan a los filtros • Si introduzco un nombre que no se encuentra en los sonidos la tabla se pinta sin contenido • Si poseo sonidos guardados y no he realizado ninguna búsqueda ni he aplicado un filtro todavía, me aparecen todos estos sonidos en la tabla 	
Tareas: <ul style="list-style-type: none"> • Crear un end-point en el backend que devuelva los sonidos almacenados de un usuario en concreto • Crear las consultas necesarias para obtener los sonidos de la base de datos • Solicitar desde el frontend la operación y parsear la respuesta • Implementar una vista que se muestre una tabla con los sonidos y un buscador 	

<ul style="list-style-type: none"> • Una vez parseados, cargar los datos traídos desde el backend en la tabla. • Interpretar un método para filtrar la búsqueda por los distintos campos descritos
Observaciones: Cuando hablamos de filtrar por categoría y valoración nos referimos a que el sonido se encuentre en una determinada categoría o que su valoración este en un determinado rango.

Identificador: HU.20	Como usuario registrado quiero cargar un sonido
Descripción: Como usuario registrado quiero cargar la configuración correspondiente a un sonido previamente guardada para poder utilizar dicho sonido y/o realizar modificaciones a partir de este.	
Estimación: 5	Prioridad: 3
Pruebas de aceptación: <ul style="list-style-type: none"> • Si solicito cargar un sonido, finalizada la carga, puedo utilizar dicho sonido correcta • Cuando cargo un sonido por interfaz se muestra que se ha cargado el sonido y se indica el nombre del sonido en la parte superior de la pantalla • Si se produce un error en la carga se muestra un mensaje de error. 	
Tareas: <ul style="list-style-type: none"> • Crear un end-point en el backend que devuelva la configuración en formato JSON del sonido solicitado. • Crear las consultas necesarias para obtener la configuración del sonido de base de datos • Solicitar desde el frontend la operación y parsear la respuesta • Actualizar todos los parámetros correspondientes a la configuración cuando es parseada 	
Observaciones:	

4.5.6. Sprint 6

ID	Nombre de la historia	Prioridad	Puntos de historia
21	Como usuario registrado quiero eliminar un sonido guardado	3	3
22	Como usuario registrado quiero utilizar mi teclado MIDI con el sistema	4	3
23	Como usuario registrado quiero ver el espectro del sonido que estoy produciendo	4	5

Identificador: HU.21	Como usuario registrado quiero eliminar un sonido guardado
Descripción: Como usuario registrado quiero eliminar un sonido que previamente había almacenado.	
Estimación: 3	Prioridad: 3
Pruebas de aceptación: <ul style="list-style-type: none"> • Si solicito eliminar el sonido, se mostrará un mensaje de confirmación para aceptarlo o rechazarlo antes de eliminarlo. • Si se elimina el sonido con éxito se mostrará un mensaje de éxito • Si se produce un error en el borrado se muestra un mensaje de error. 	
Tareas: <ul style="list-style-type: none"> • Crear un end-point en el backend para eliminar el sonido • Crear las consultas necesarias para eliminar el sonido • Implementar una vista para eliminar el sonido • Mandar petición al backend, parsear el resultado e interpretarlo. 	
Observaciones:	

Identificador: HU.22	Como usuario registrado quiero utilizar mi teclado MIDI con el sistema
Descripción: Como usuario registrado quiero conectar mi teclado MIDI al ordenador. Una vez conectado desea tocar los sonidos del oscilador (las notas) utilizando mi teclado MIDI y que estos sonidos vayan acordes con la tecla del teclado que estoy pulsando.	
Estimación: 3	Prioridad: 4
Pruebas de aceptación: <ul style="list-style-type: none"> • Al conectar el teclado MIDI, se indica por pantalla que dispositivo MIDI se ha conectado • Si no tengo ningún dispositivo MIDI conectado se indica por pantalla que no hay ninguno conectado • Si tengo el teclado MIDI conectado si pulso una nota el sonido reproducido se corresponde con dicha nota • Si tengo el teclado MIDI conectado, si pulso una nota de la interfaz o una tecla del teclado se sigue reproduciendo el sonido. 	
Tareas: <ul style="list-style-type: none"> • Implementar un método de comunicación con un Hardware MIDI • Interpretar los datos enviados por el teclado MIDI • Implementar una zona de la interfaz que indique la presencia o no del MIDI • Sincronizar la interfaz con la respuesta MIDI 	

Observaciones:

Identificador: HU.23	Como usuario registrado quiero ver el espectro del sonido que estoy produciendo
-----------------------------	---------------------------------------------------------------------------------

Descripción:
Como usuario registrado quiero poder visualizar en pantalla, mediante unos gráficos de barras, dinámicos, es decir, que se van pintando de manera sincronizada con el sonido reproducido, el espectro de dicho sonido. Además, debo poder visualizar que zonas del espectro están cercanas a valores muy altos de amplitud (volumen) por medio de colores. Por último, cuando pasé el ratón por una de las zonas del gráfico me indicará de que frecuencia se trata.

Estimación: 5	Prioridad: 4
----------------------	---------------------

Pruebas de aceptación:
<ul style="list-style-type: none"> • Cuando reproduzco una nota se visualiza el gráfico acorde con la duración e intensidad del sonido • Si un sonido posee mucha información en graves, la zona del espectro correspondiente a los graves se pinta en rojo, al igual que pasa con el resto de las zonas del espectro • Si pongo el ratón encima de la gráfica me indica la frecuencia en la que me sitúo

Tareas:
<ul style="list-style-type: none"> • Implementar un método de sincronización de los gráficos con el sonido • Implementar un método de pintado de los gráficos • Implementar una vista para los gráficos

Observaciones: Cuanto más alta sea la amplitud de una frecuencia su color estará más cercano al rojo, cuanto más baja al verde, y en un punto cercano al rojo en amarillo.

4.5.7. Sprint 7

ID	Nombre de la historia	Prioridad	Puntos de historia
24	Como usuario registrado quiero utilizar el ecualizador de 5 bandas	4	5
25	Como usuario registrado quiero grabar una pieza musical	4	3
26	Como usuario no registrado quiero registrarme	4	3

Identificador: HU.24	Como usuario registrado quiero utilizar el ecualizador de 5 bandas
Descripción: Como usuario registrado quiero ecualizar los sonidos producidos a partir de un ecualizador de 5 bandas. Habrá una banda para las frecuencias bajas, las frecuencias medias bajas, las medias, medias altas y altas. Estas bandas modificarán la amplitud de las frecuencias en un rango de 25 dB de incremento o decremento, a partir de la aplicación del filtro adecuado para ello.	
Estimación: 5	Prioridad: 4
Pruebas de aceptación: <ul style="list-style-type: none"> • Si incremento la banda grave el sonido suena más grave. Al contrario, si decremento. Lo mismo ocurre con el resto de bandas 	
Tareas: <ul style="list-style-type: none"> • Implementar una vista para las bandas • Implementar la lógica para aplicar los filtros de las bandas 	
Observaciones:	

Identificador: HU.25	Como usuario registrado quiero grabar una pieza musical
Descripción: Como usuario registrado quiero grabar una pieza musical tocada con los sonidos reproducidos por el oscilador, es decir, deseo poder grabar la secuencia de sonidos en un intervalo de tiempo generada por el sintetizador.	
Estimación: 3	Prioridad: 4
Pruebas de aceptación: <ul style="list-style-type: none"> • Si presiono el botón de grabar, toco algunas notas y posteriormente el de pausa, puedo escuchar la grabación hasta el momento de la pausa • Si la grabación estaba pausada y vuelvo a reanudarla la grabación sigue y cuando vuelvo a pausarla puedo escuchar el resultado de la grabación que había antes de la primera pausa seguida de la grabación realizada al reanudarla • Si hay una grabación y pulso en descargar la grabación se descarga en mi ordenador • Si hay una grabación pausada y presiono el botón de reiniciar y realizo otra grabación el resultado de la grabación será esta nueva grabación 	
Tareas: <ul style="list-style-type: none"> • Implementar una vista para la grabación • Implementar la lógica para realizar la grabación 	
Observaciones:	

--

Identificador: HU.26	Como usuario no registrado quiero registrarme
Descripción: Como usuario no registrado deseo crearme una cuenta para posteriormente loguearme en la aplicación	
Estimación: 3	Prioridad: 4
Pruebas de aceptación: <ul style="list-style-type: none"> • Si envió algún campo vacío se muestra un mensaje indicándolo • Si introduzco un nombre de usuario que ya existe se notifica por pantalla • Si introduzco una dirección de correo electrónico asociada a un usuario existente se notifica por pantalla • Si la longitud de la contraseña no es adecuada se muestra un mensaje por pantalla • Si las dos contraseñas introducidas por pantalla no coinciden se muestra un mensaje por pantalla • Si introduzco todos los campos adecuadamente se muestra un mensaje de éxito y mi usuario queda registrado 	
Tareas: <ul style="list-style-type: none"> • Crear los modelos necesarios en la base de datos • Crear un end-point en el backend para registrar un usuario • Crear las consultas necesarias para registra un usuario • Implementar una vista • Crear manejo de rutas en el frontend • Implementar la validación del formulario • Implementar mecanismo de envío y recepción de respuesta • Interpretar las respuestas del servidor 	
Observaciones:	

4.5.8. Sprint 8

ID	Nombre de la historia	Prioridad	Puntos de historia
27	Como usuario registrado quiero hacer login	4	5
28	Como usuario registrado quiero hacer logout	4	1
29	Como usuario registrado quiero modificar mis datos personales	5	3

Identificador: HU.27	Como usuario registrado quiero hacer login
Descripción: Como usuario registrado quiero poder iniciar sesión (login) en la aplicación para poder hacer uso de ella	
Estimación: 5	Prioridad: 4
Pruebas de aceptación: <ul style="list-style-type: none"> • Si introduzco un nombre de usuario que no se corresponde con ninguno registrado se muestra un mensaje de error • Si introduzco una contraseña incorrecta se muestra un mensaje de error • Si introduzco las credenciales correctas me redirige al sintetizador y se me almacena una cookie de usuario y otra de token 	
Tareas: <ul style="list-style-type: none"> • Crear y gestionar el jwt Token en el backend • Crear un end-point en el backend para loguear un usuario • Crear las consultas necesarias para hacer el login • Implementar una vista para el login • Gestión de sesiones • Implementar la validación del formulario • Implementar mecanismo de envío y recepción de respuesta • Interpretar las respuestas del servidor 	
Observaciones:	

Identificador: HU.28	Como usuario registrado quiero hacer logout
Descripción: Como usuario registrado quiero poder cerrar sesión (logout) en la aplicación	
Estimación: 1	Prioridad: 4
Pruebas de aceptación: <ul style="list-style-type: none"> • Si hago logout, se me muestra una alerta, en caso de aceptarla se me redirige al login y se me eliminan las cookies que tenía almacenadas 	
Tareas: <ul style="list-style-type: none"> • Implementar un botón para el logout • Implementar la lógica 	
Observaciones:	

Identificador: HU.29	Como usuario registrado quiero modificar mis datos personales
Descripción:	

Como usuario registrado quiero poder modificar mi nombre de usuario, email, fecha de nacimiento y contraseña. Para ello previamente debo poder consultar los datos de mi perfil almacenados	
Estimación: 3	Prioridad: 5
Pruebas de aceptación: <ul style="list-style-type: none"> • Si intento modificar el nombre de usuario a uno que ya existe se muestra un error • Si intento modificar el email a un que ya existe se muestra un error • Si las contraseñas introducidas no coinciden se muestra un error • Si modifico un campo y lo dejo vacío se muestra un error • Si modifico todos los datos correctamente, se muestra un mensaje de éxito 	
Tareas: <ul style="list-style-type: none"> • Crear un end-point en el backend para consultar el perfil de un usuario • Crear las consultas necesarias para obtener el perfil de un usuario • Crear un end-point en el backend para modificar la contraseña de un usuario • Crear las consultas necesarias para modificar la contraseña de un usuario • Implementar una vista para los datos del perfil • Mandar la petición al backend y gestionar la respuesta • Validar los formularios 	
Observaciones:	

4.5.9. Sprint 9

ID	Nombre de la historia	Prioridad	Puntos de historia
30	Como usuario registrado quiero cambiar la contraseña	5	5

Identificador: HU.30	Como usuario registrado quiero cambiar la contraseña
Descripción: Como usuario registrado quiero poder cambiar mi contraseña	
Estimación: 5	Prioridad: 5
Pruebas de aceptación: <ul style="list-style-type: none"> • Si ambas contraseñas no coinciden el sistema muestra error 	
Tareas: <ul style="list-style-type: none"> • Implementar lógica para modificar contraseña • Implementar vista para modificar contraseña 	

Capítulo 5: Diseño

En este capítulo se va a proceder a detallar como se realizado el diseño del sistema en general, comprendiendo el diseño de la arquitectura de este, el diseño de la interfaz de usuario, así como la estructura de clases que se implementarán.

5.1. Arquitectura

Se busca un modelo software en el que por un lado se encuentre la aplicación del usuario, en la que este interactuará, y por otro lado, un servidor de información que proporcione los datos para el funcionamiento de la aplicación totalmente ajeno al usuario. De esta manera, el usuario percibirá el sistema como un TODO a partir de la aplicación.

A partir de esta diferenciación, podemos observar, que se busca desarrollar un modelo software basado en una arquitectura cliente-servidor ³, aunque por motivos de escalabilidad, la opción más idónea para llevar a cabo este modelo es una arquitectura n-capas.

5.1.1. Arquitectura n-capas

El modelo de arquitectura cliente-servidor es un modelo en el que las tareas son repartidas entre cliente y servidor, de manera, que el cliente realiza peticiones al servidor y este le da respuesta. Es evidente que este modelo realiza una separación, de tipo lógica, aunque también podría ser física, entre el programa cliente y el programa servidor. Las arquitecturas cliente servidor genéricas tienen dos tipos de nodos, un nodo cliente y otro servidor. De acuerdo con esta disposición, se pueden abstraer dos capas, la capa cliente y la capa servidor, cada una con independencia lógica. Es muy común que el servidor, este compuesto por varios programas, es decir, que se divida en varias capas. Esta división del servidor, convierte a la arquitectura cliente-servidor en una arquitectura de n-capas. [7]

La arquitectura n-capas es un modelo software en el que las responsabilidades se separan en forma de capas, es decir, cada representa una abstracción de un conjunto de responsabilidades específicas. Este modelo debe seguir las siguientes características: [8]

- Los niveles tienen que estar separados, físicamente y lógicamente.
- Una capa superior puede utilizar los servicios de una inferior, pero no al revés.

De esta manera, se considera que este modelo software de n-capas es el más adecuado para el sistema a desarrollar, ya que nos va a proporcionar escalabilidad e independencia entre las capas facilitando el desarrollo.

A continuación, se muestra una representación del sistema en cuestión siguiendo este modelo software:

³ Arquitectura software en la que un programa (cliente) realiza peticiones a otro programa(servidor), quien le da una respuesta

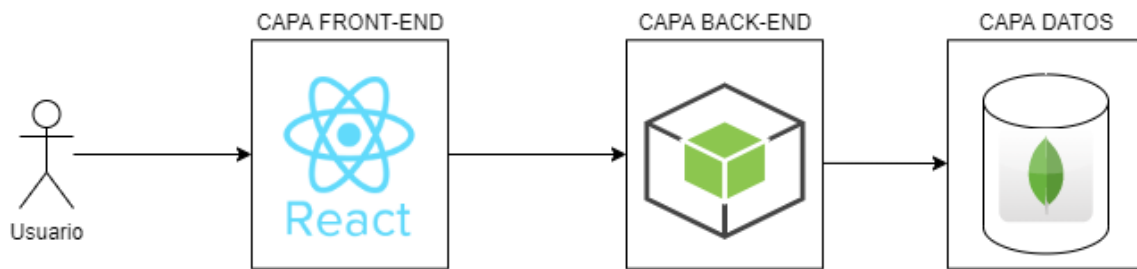


Figura 5.1: Diagrama conceptual de la arquitectura

Como se puede observar en la [Figura 5.1](#), la arquitectura a desarrollar constará de tres capas, por lo que se trata de una **arquitectura 3-capas**. Las capas en las que se va a dividir el sistema son:

- **Capa Font-end:** Es la capa que representa a la vista, es decir, a la parte del sistema con la que el usuario interactuará directamente. Su responsabilidad será abstraer toda la funcionalidad relativa a la interfaz del sistema y para su implementación se utilizará ReactJS principalmente como Framework (ver [sección 6.2.4](#)).
- **Capa Back-end:** Esta capa es completamente invisible al usuario. Su responsabilidad consistirá en realizar peticiones a la base de datos, para consultar, modificar, eliminar o crear información en esta, y proporcionarle esta información de manera procesada al usuario. En otras palabras, esta capa actuará como intermediaria entre la capa Front-end y la capa de datos. Para su implementación se utilizará *NodeJS* y *Express*.
- **Capa de datos:** Invisible al usuario. Se corresponde con una Base de datos MongoDB no relacional. Su responsabilidad será almacenar los datos que el sistema requiere para su adecuado funcionamiento.

Estas capas, como ya se ha mencionado, deben mantener una independencia tanto lógica como física. Cada capa se corresponderá con un programa independiente, logrando una independencia lógica. En cuanto a la independencia física, esta será lograda mediante la ejecución de las distintas capas (programas) en puertos distintos del mismo equipo, ya que por motivos de recursos no se dispone de varias máquinas, aunque el sistema está pensado para que cada capa se ejecute en equipos distintos.

La comunicación entre las capas va a ser bidireccional, pero cumpliéndose siempre que una capa inferior no puede realizar peticiones a algún servicio o método de una capa superior. Las comunicaciones entre capas van a seguir el siguiente esquema:

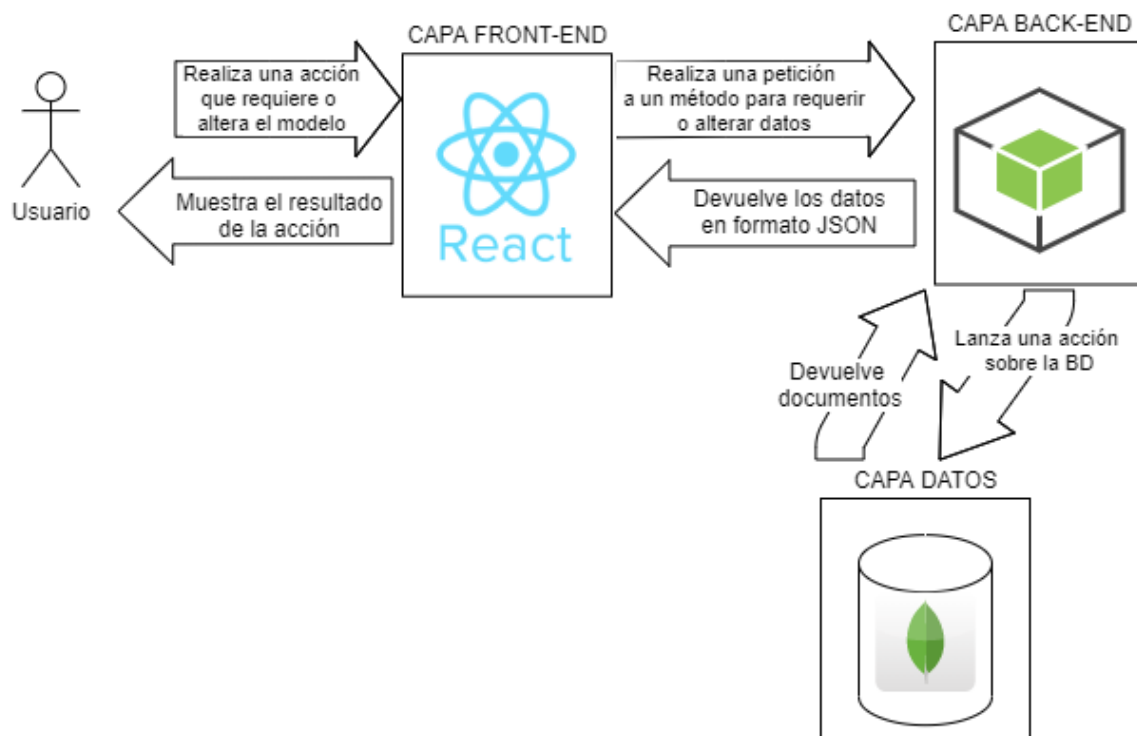


Figura 5.2. Esquema de comunicación entre las capas

De acuerdo con los dos esquemas anteriores, se pueden identificar los diferentes componentes que existirán en el sistema, los cuales se indican en el siguiente diagrama de componentes:

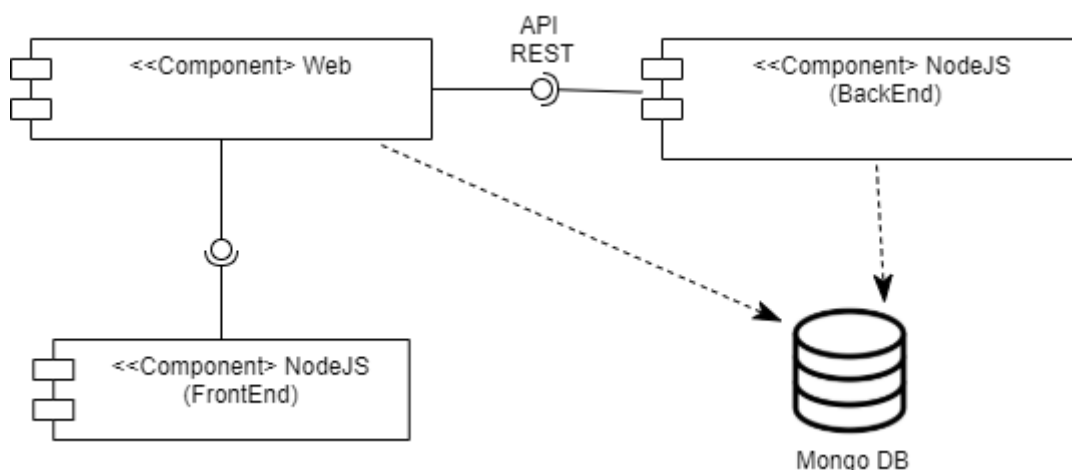


Figura 5.3: Diagrama de componentes del sistema

A continuación, se explican los componentes que aparecen en el diagrama:

- **Web:** Se corresponde con la interfaz. El usuario interactuará continuamente con este componente.
- **NodeJS (frontend):** Este componente, se encarga de traducir todo el código en JavaScript en código para el cliente (HTML). La existencia de este componente se debe a la utilización de React como Framework para el desarrollo de la capa Front-end, por lo que se necesita de este componente para que se realicen las traducciones adecuadas para la visualización de la interfaz de manera correcta.

Este componente es incluso invisible para el desarrollador.

- **NodeJS (backend):** Se corresponde en su totalidad con la capa Backend y proporciona una interfaz al componente web que seguirá un protocolo API-REST, que se tratará en secciones posteriores de esta memoria.

Ya se ha abordado el diseño de la arquitectura del sistema, por lo que se procede al diseño de cada una de las capas por separado.

5.2. Diseño Capa de Datos

La capa de datos consistirá en una base de datos no relacional, en concreto, una base de datos MongoDB, de tipo no relacional, que se comunicará con la capa Back-end (exclusivamente). Este tipo de bases de datos tienen las siguientes características: [9]

- No utilizan el lenguaje SQL para las consultas
- No se requieren estructuras fijas como tablas
- Los datos se almacenan en documentos
- Los conjuntos de datos no tienen por qué estar relacionados

Estas características hacen que este tipo de base de datos se adapte a nuestro sistema, ya que vamos a necesitar datos independientes que no tienen porque estar relacionados entre sí, y nos va a permitir escalar horizontalmente⁴ el sistema de manera más sencilla que si usáramos un modelo de datos relacional.

En esta capa se van a encontrar los datos que necesita el sistema para su funcionamiento. Estos datos se representan en el siguiente esquema:

⁴ Al hablar de escalado horizontal, nos referimos a la capacidad de añadir nuevos nodos a la capa sin que esta se vea afectada.

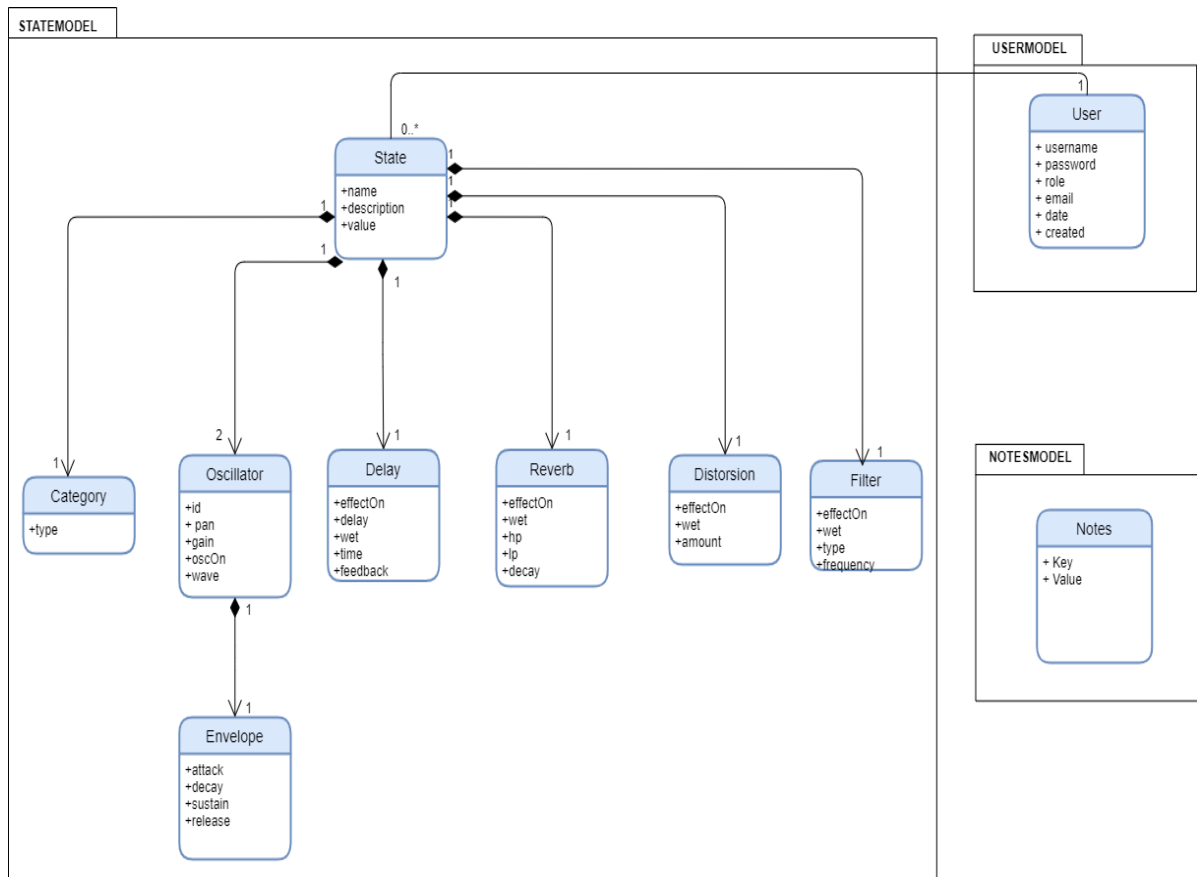


Figura 5.4: Esquema de la base de datos

Del anterior esquema se puede deducir los siguiente:

- La base datos constará con tres paquetes, que constituyen una representación de un conjunto de datos necesarios en el sistema. Estos paquetes se corresponderán cada uno con una colección de documentos. Tendremos:
 - **State Model**
 - **User Model**
 - **Notes Model**
- No todos los elementos están relacionados. Obsérvese que NotesModel es totalmente independiente al resto.

5.2.1. State Model

Se va a corresponder con un estado del sonido, es decir, con todos los datos que intervienen en la generación y reproducción de un sonido digital.

En el esquema de la [figura 5.4](#) se puede observar que se ha seguido un **patrón de diseño composite**. Este patrón consiste en la construcción de un objeto a partir de la construcción de otros más simples, de acuerdo al siguiente esquema, extraído del libro del GoF [10]:

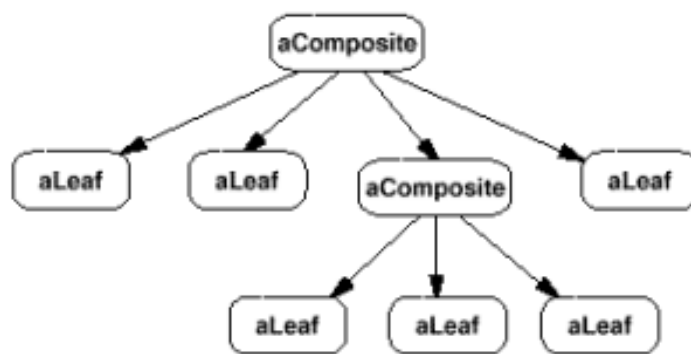


Figura 5.5. Patrón Composite

Siguiendo este patrón tenderemos

- **Objetos Leaf u Hoja:** Van a representar los objetos más simples que no requieren de otros para existir.
- **Objetos Composite o Compuestos:** Objetos que están compuestos de otros más simples, ya sean otros compuestos u objetos hoja.

Todo esto se refleja en el diseño, de la siguiente manera:

- **State:** Va a representar el estado del sonido y es un objeto compuesto de:
 - **Oscillator:** Va a representar los datos referentes a un oscilador y a su vez está compuesto de:
 - **Envelope:** Va a representar a una envolvente y es un objeto hoja.
 - **Delay:** Va a representar al efecto de Delay y es un objeto hoja.
 - **Reverb:** Va a representar al efecto de Reverb y es un objeto hoja.
 - **Filter:** Va a representar al efecto de Filter y es un objeto hoja.
 - **Distorsion:** Va a representar al efecto de Distorsion y es un objeto hoja.
 - **Category:** Va a representar un tipo de sonido y es un objeto hoja.

Por último, un State o Estado estará relacionado con un usuario, a pesar de seguir un modelo no relacional, ya que se necesitará asociar que estados le pertenecen a cada usuario.

5.2.2. User Model

Se corresponde con la representación de un usuario en base de datos. Este paquete contará con único objeto **User**, el cual estará relacionado con 0 o muchos estados ya que un usuario podrá guardar muchos sonidos.

5.2.3. Notes Model

Es una representación de las notas musicales, a partir de parejas clave valor, haciendo coincidir una determinada nota musical con su correspondiente frecuencia. Solo cuenta con el objeto **Notes** y es totalmente independiente al resto de modelos.

5.3. Diseño Capa Back-end

La capa Back-end servirá de intermediaria entre la capa Front-end y la capa de datos. De acuerdo con su responsabilidad, será necesario disponer de una serie de mecanismos de comunicación que hagan posible esta acción de intermediación entre su capa superior e inferior. Dichos mecanismos de comunicación se basarán en una API que seguirá un **protocolo REST**, aunque cabe destacar que dicho protocolo solo se dará entre la comunicación de esta capa con la capa Front-end, ya que la tecnología empleada para el desarrollo del sistema proporciona otros mecanismos para la comunicación con la base de datos, los cuales serán tratados más adelante en esta memoria (ver [sección 6.2](#)).

5.3.1. Protocolo REST [11]

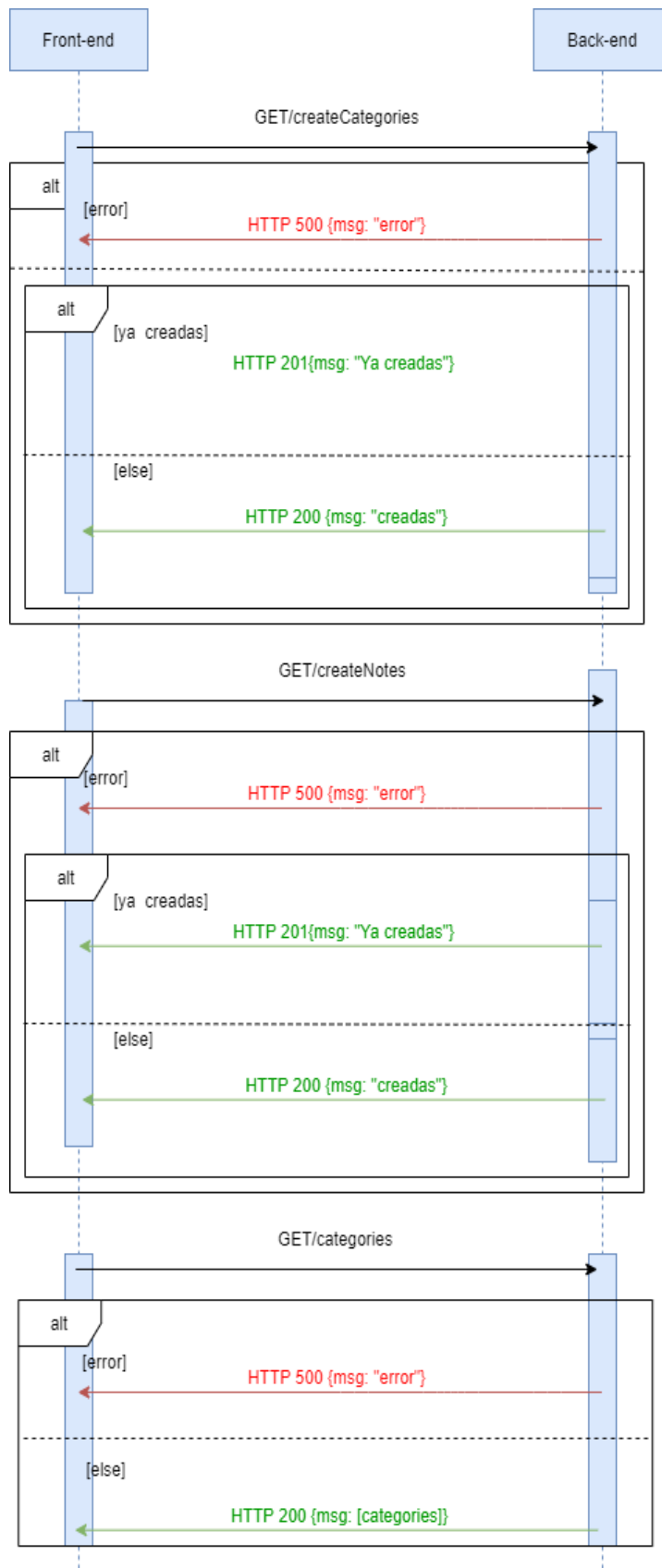
El protocolo REST o protocolo de transferencia de estado representacional, es una interfaz que conecta varios sistemas, en nuestro caso conectará la capa Front-end con la capa Back-end, basándose en el protocolo HTTP ⁵. Este protocolo tiene las siguientes características:

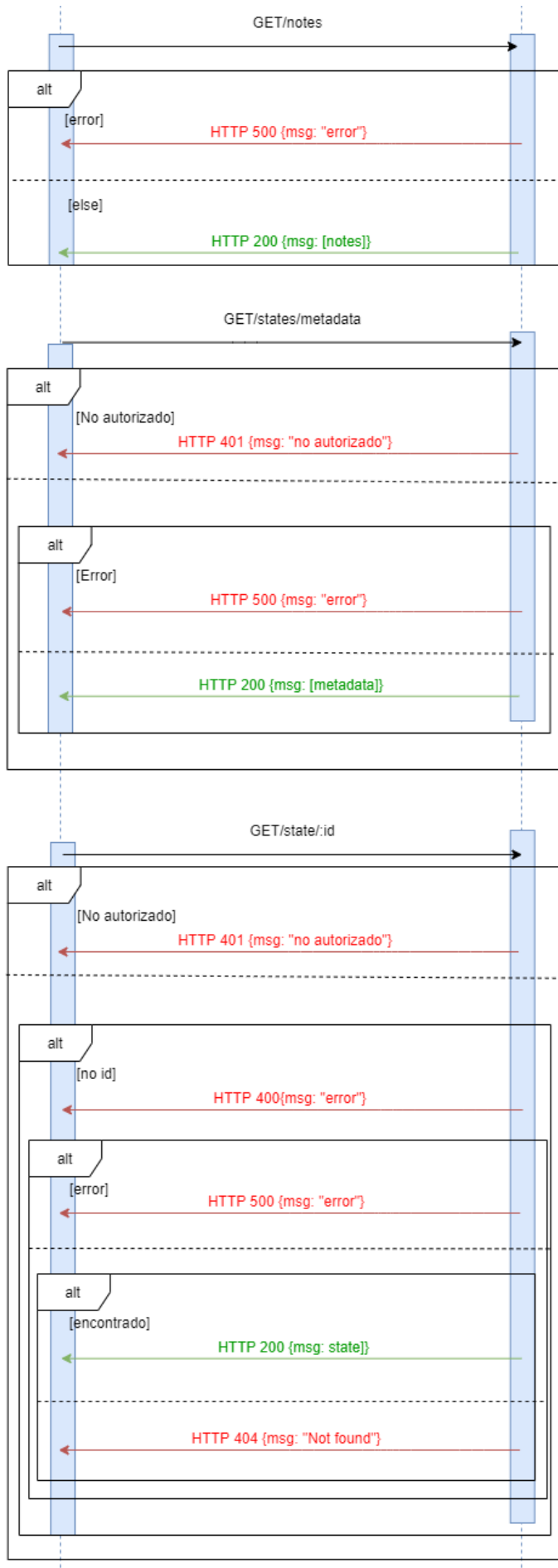
- **No tiene estado:** Toda la información va en la petición por lo que no se tiene que almacenar el estado de las comunicaciones.
- **Las operaciones están definidas:** Operaciones PUT, GET, POST, DELETE, PATCH.
- **Cada recurso se identifica con una URI**

Este protocolo nos va a permitir aislar aún más cada una de las capas, ya que no necesitamos el estado de las comunicaciones favoreciendo que se siga el modelo software que se va diseñar (arquitectura n-capas).

De acuerdo con el protocolo REST, la comunicación entre la capa Front-end y la capa Back-end seguirá los siguientes y diagramas de interacción:

⁵ “Hypertext Transfer Protocol”, protocolo que permite realizar una petición de datos y recursos





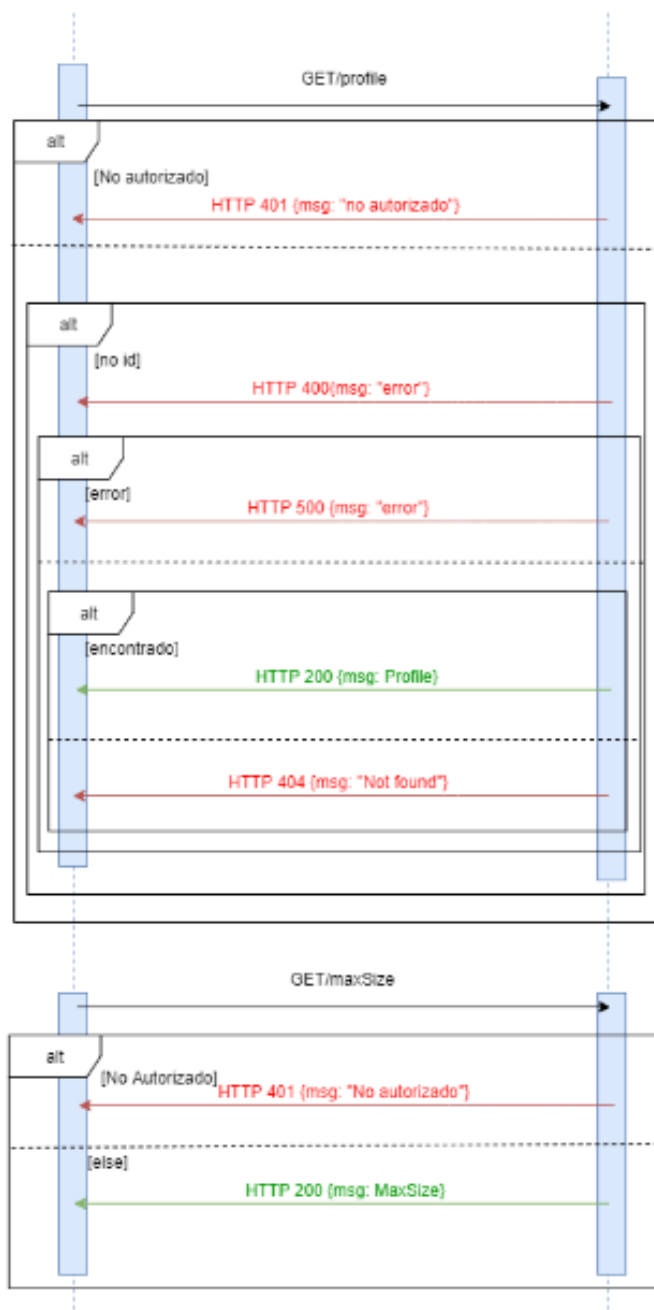
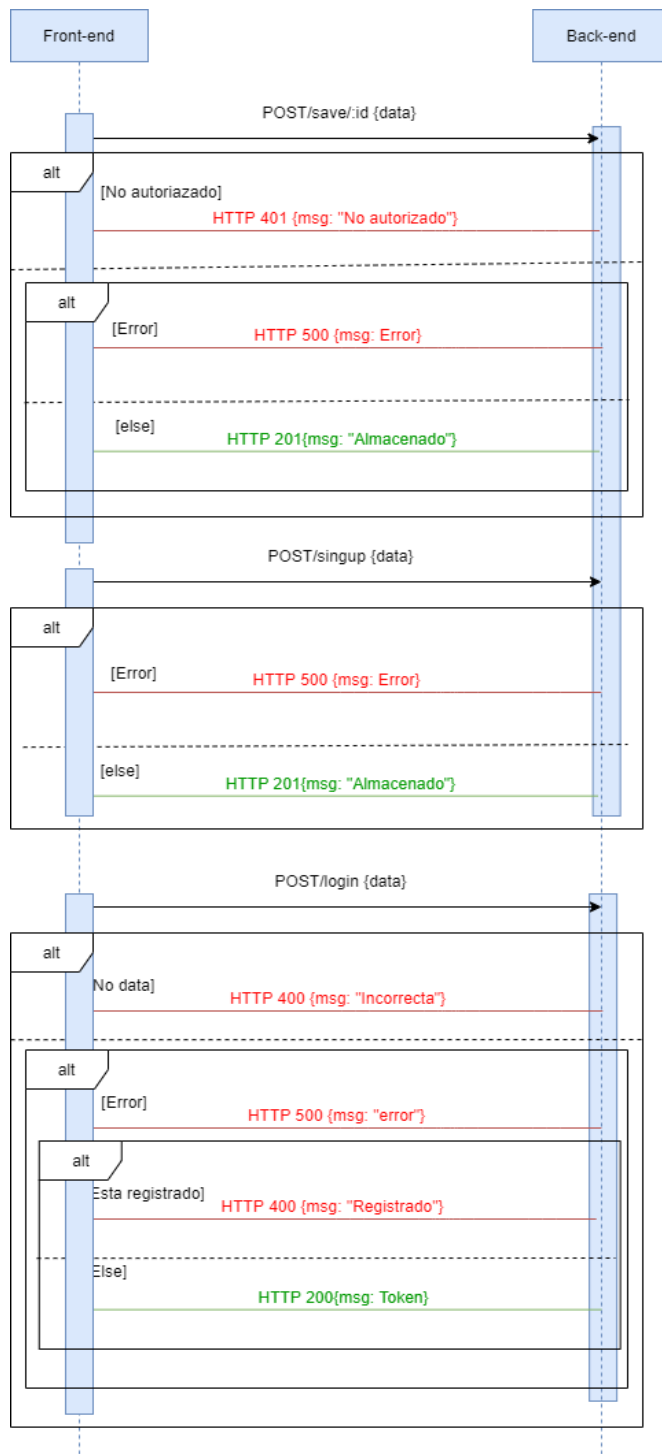


Figura 5.6. Diagrama interacción GET Front-end Back-end



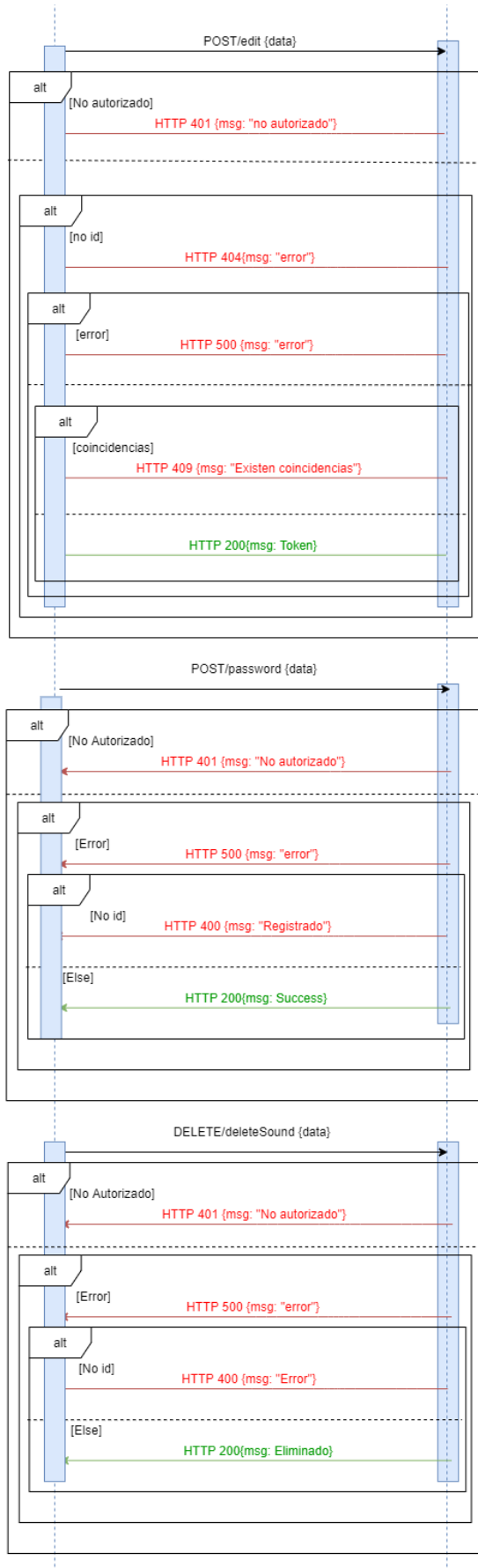


Figura 5.7. Diagrama interacción POST y DELETE Front-end Back-end

5.3.2. Arquitectura de la capa

Siguiendo el modelo arquitectónico elegido para el sistema, esta capa también se diseñará de acuerdo a dicho modelo, por lo que la capa Back-end estará a su vez compuesta por capas. Cabe destacar que se va a adaptar esta arquitectura, ya que no va a existir una independencia física, aunque sí lógica, viéndose modificados algunos de los principios de la arquitectura por capas.

La descomposición por capas de la capa Back-end va a seguir el siguiente esquema:

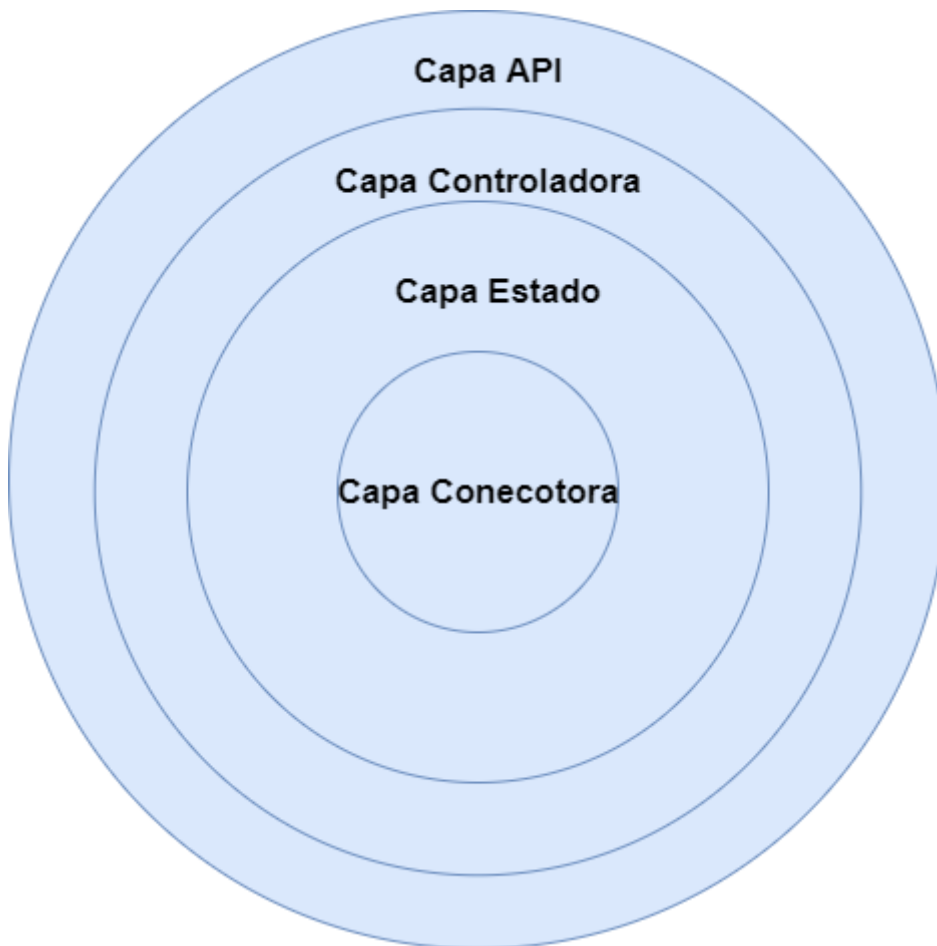


Figura 5.8. Esquema arquitectónico de la capa Back-end

De acuerdo con el anterior esquema, tendremos las siguientes capas:

- **Capa API:** Se va a encargar de escuchar y filtrar las peticiones que llegan por parte de la capa Front-end, así como de delegar la petición a la capa controladora para su interpretación. Constituye una API, que servirá de interfaz siguiendo los diagramas de la [figura 5.6](#) y la [figura 5.7](#).
- **Capa Controladora:** Se va a encargar de interpretar y procesar las peticiones que la capa API delega en ella, así como de enviar instrucciones a los estados de la capa Estado.
- **Capa Estado:** Se va a encargar de crear, modificar y eliminar los distintos objetos de la Base de datos. Constituye una representación directa de las entidades existentes en la base de dato, además es la responsable de crear el modelo de datos.
- **Capa Conectora:** Se va a encargar de realizar la conexión con la Base de datos.

5.4. Diseño Capa Front-end

La capa Front-end será constituirá la capa de presentación del sistema, es decir, su responsabilidad será proporcionar la interfaz con la que el usuario podrá interaccionar con el sistema diseñado en esta memoria, por lo que su diseño estará totalmente centrado en el usuario.

Ya diseñadas la capa Back-end y la capa de datos, a continuación, se procede a diseñar la capa más externa de la arquitectura del sistema.

5.4.1. Arquitectura de la capa

El primer paso será definir la arquitectura de la capa. Siguiendo la arquitectura general del sistema, ya abordada durante el desarrollo de este capítulo, se opta por mantener este diseño en capas e incorporarlo en el diseño de la capa Front-end. De esta manera, al igual que la capa Back-end y el sistema en general, la capa Front-end estará compuesta a su vez por otras “*subcapas*”, lo cual nos va a proporcionar una mayor independencia, menor acoplamiento y mayor cohesión de los objetos de esta capa.

La subdivisión en capas de la capa Front-end seguirá el siguiente esquema:

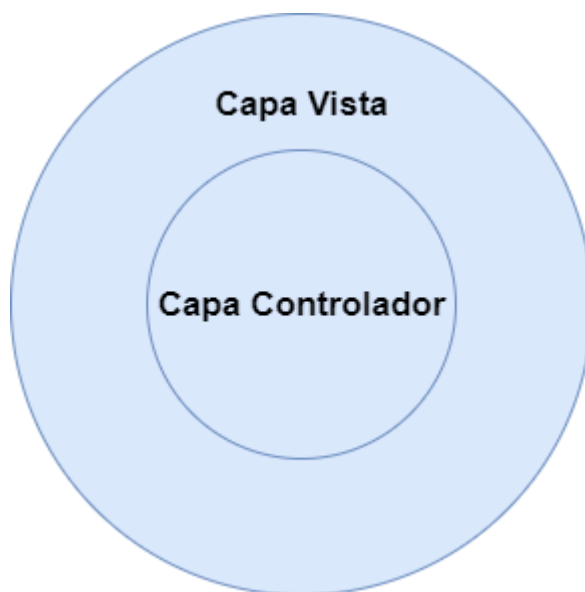


Figura 5.9. Esquema arquitectónico de la capa Front-end

Como se pueda observar, es una arquitectura bastante sencilla en la que solo existirán dos capas, las cuales se describen a continuación:

- **Capa Vista:** Su responsabilidad será renderizar el código HTML y manipular el DOM ⁶ e interpretar las acciones que el usuario realiza sobre la interfaz de la aplicación web.
- **Capa Controlador:** Su responsabilidad será realizar peticiones a la capa Back-end, correspondientes con acciones del usuario en la interfaz, así como hacer uso de la Web Audio API para la generación y reproducción de los sonidos.

⁶ “Document Object Model” representación de la estructura del documento HTML en forma de árbol de etiquetas relacionadas entre sí

5.4.2. Patrón de diseño

Una vez abordada la arquitectura de la capa es necesario establecer un patrón de diseño a seguir. Como se ha mencionado, la capa Front-end proporcionará los componentes necesarios que constituirán la interfaz, así como los mecanismos necesarios para la interacción con la Web Audio API. Es por esto, que se precisarán de distintas “*entidades*” que gestionen todos estos aspectos, ya que habrá responsabilidades independientes dentro de esta capa.

La mejor forma de abordar estas cuestiones, es siguiendo un **paradigma orientado a objetos**, de manera que cada una de estas entidades se corresponderán con objetos que interaccionarán entre si, y estos objetos a su vez estarán representado por clases en el código.

Para dotar de orden y organización a los distintos objetos que compondrán la capa Front-end se opta por seguir un **patrón de diseño Facade o Fachada**.

El libro de *Design Patterns: Elements of Reusable Object-Oriented Software (GoF)* [10] da la siguiente definición sobre este patrón: “Es un patrón de diseño que *proporciona una interfaz unificada a un conjunto de interfaces en un subsistema. La fachada define una interfaz de nivel superior que hace que el subsistema sea más fácil de usar*”.

De acuerdo con la definición, existirá una clase que actuará de interfaz entre subsistemas, en este caso la clase fachada actuará como una interfaz de la subcapa Controlador, es decir comunicará la subcapa de la vista con esta, de acuerdo al siguiente esquema:

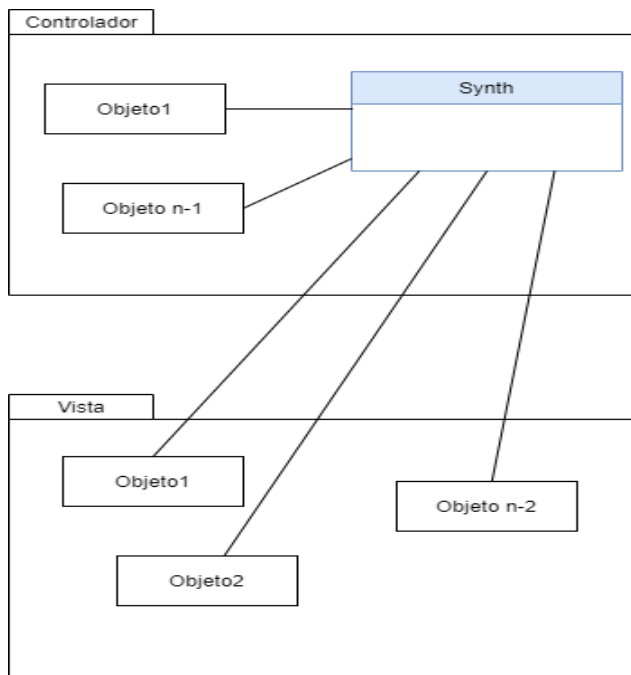


Figura 5.10. Patrón Fachada

Se puede observar lo siguiente en la figura anterior:

- La clase **Synth** actuará de clase fachada. Esta clase atiende a las peticiones de los clientes, que en este caso, son objetos de la subcapa de la vista y conoce perfectamente que clase de la subcapa de controlador puede resolver esta

petición. En otras palabras, la clase fachada conoce a las clases especialistas que contendrán la lógica para resolver las peticiones de las clases clientes y sabe a quién delegar la petición.

- Las clases de la subcapa controlador, son las clases “*especialistas*” en las que la clase fachada delega responsabilidad. No conocen la existencia de la clase fachada.
- Las clases de la subcapa vista, son las clases “clientes” que llamarán a algún método de la fachada. Conocen la existencia de la clase fachada.

5.4.3. Diagramas de actividad

Llegados a este punto, ya hemos definido el patrón que seguirá el diseño de la capa, es decir, ya hemos definido la estructura de clases a seguir. Para seguir completando dicha estructura con clases, es necesario, poner un foco en las acciones que se espera que el usuario vaya a realizar en la interfaz.

De acuerdo con el con el proceso de análisis que se hizo en el [capítulo 4](#) , donde se especificaron los requisitos del sistema y las historias de usuario, podemos identificar los siguientes cursos de acción que serán fundamentales para el diseño de la capa:

- **Reproducción de un sonido**
- **Guardado de un sonido**
- **Carga de un sonido**
- **Inicio de sesión**

Estos cursos de acción se representan en los diagramas de actividad que vienen a continuación:

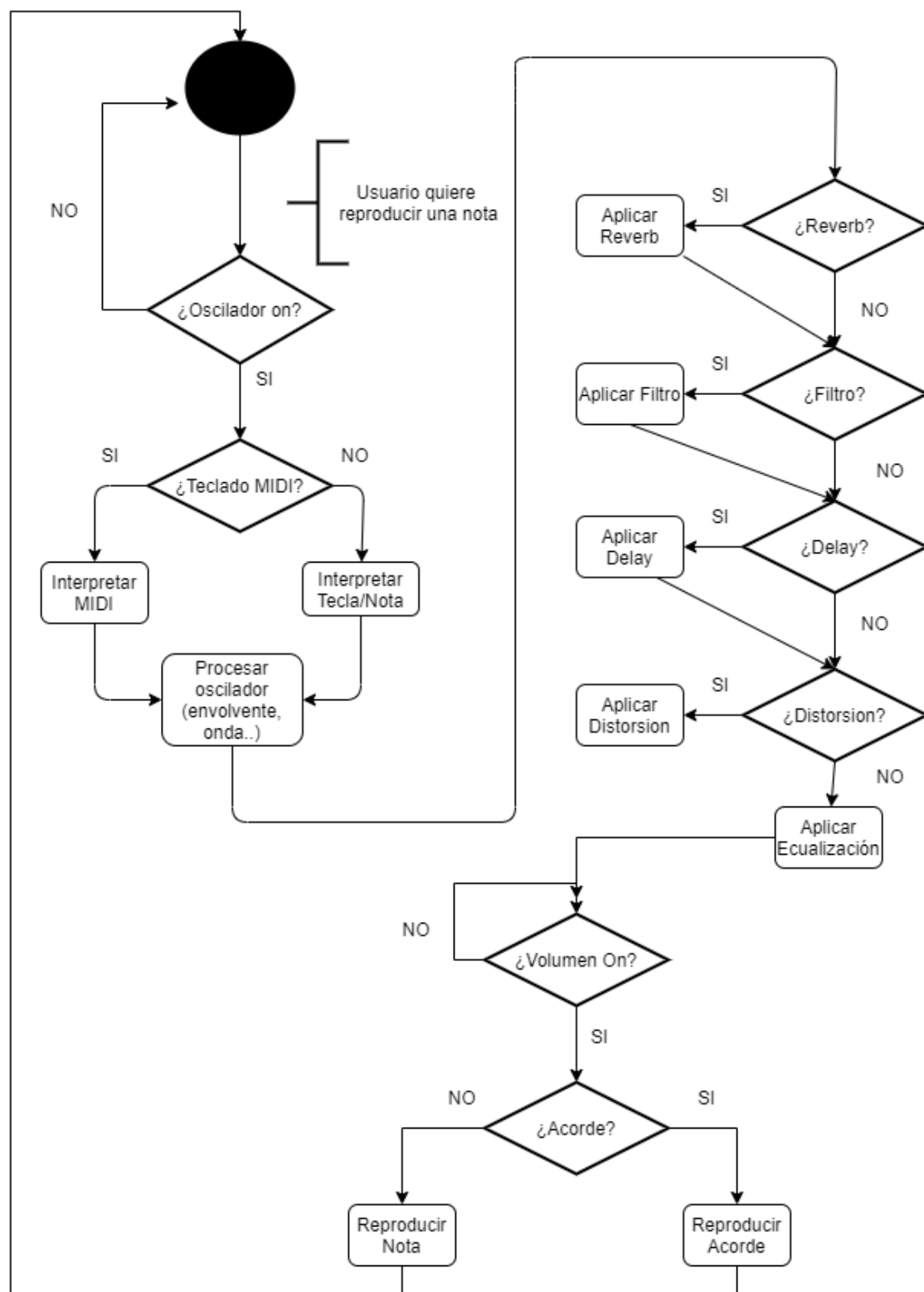


Figura 5.11. Diagrama actividad reproducción sonidos

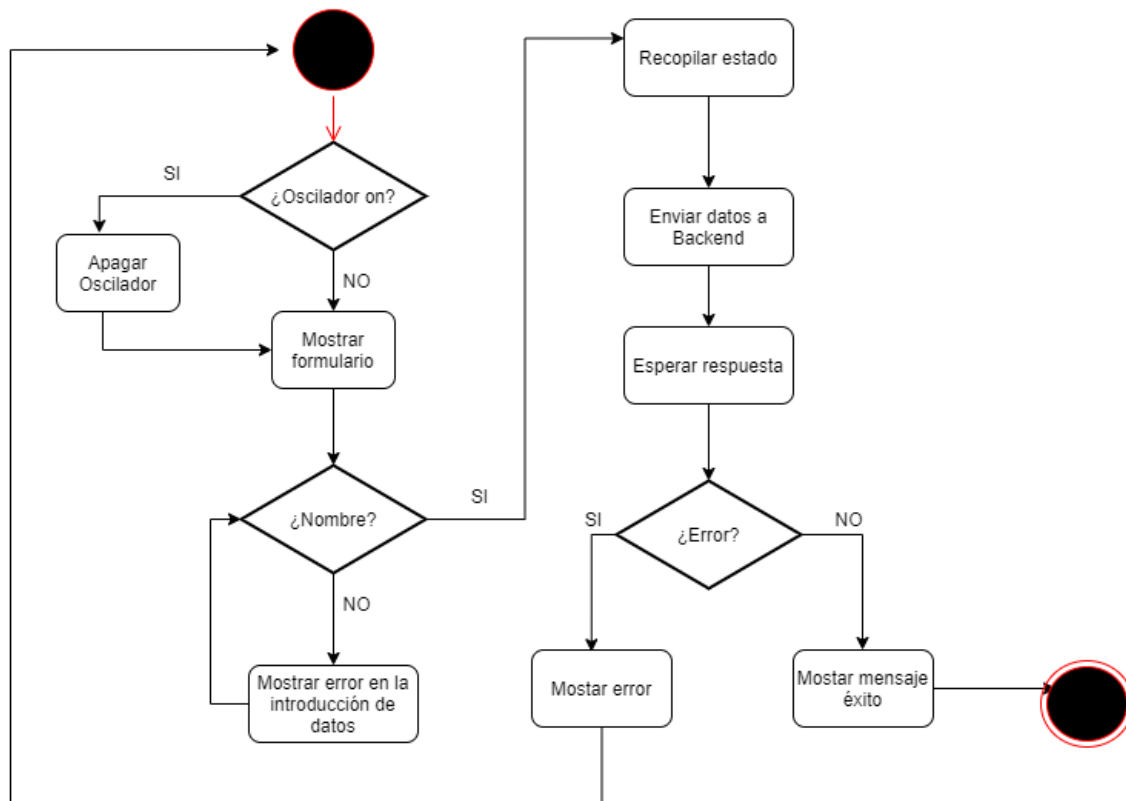


Figura 5.12. Diagrama actividad guardado sonidos

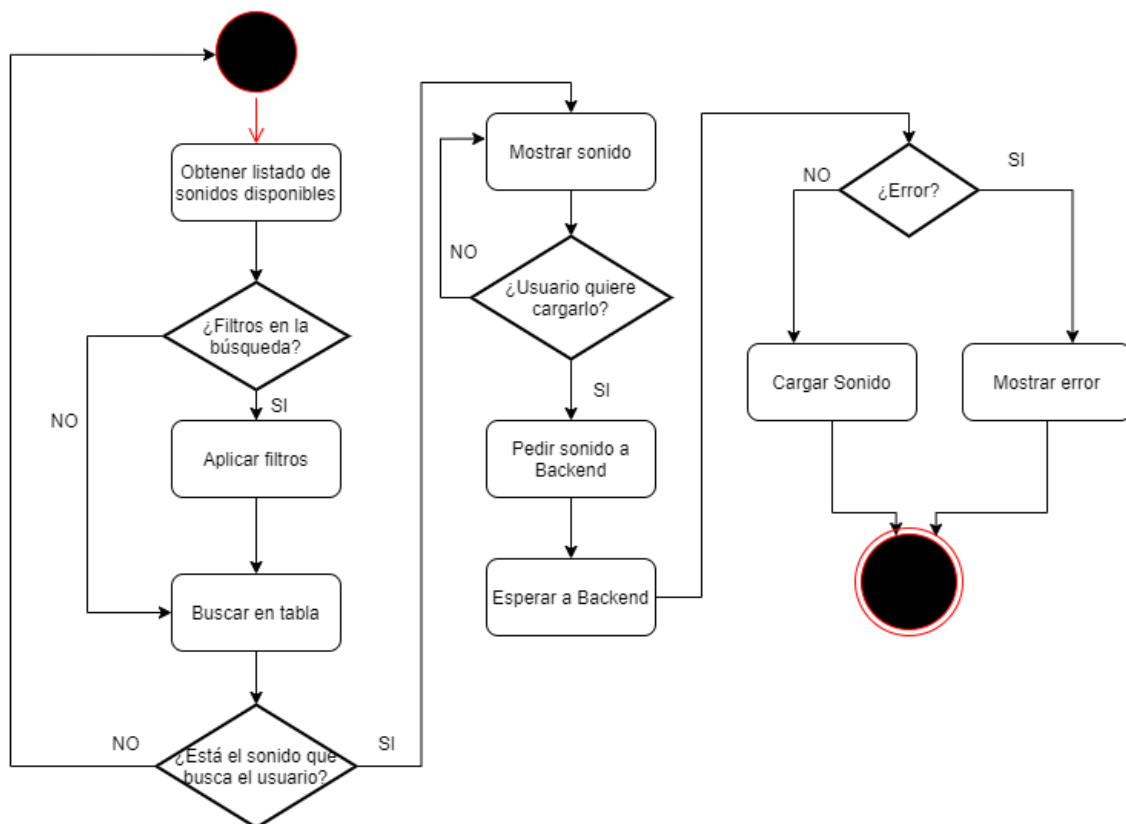


Figura 5.13. Diagrama actividad guardado sonidos

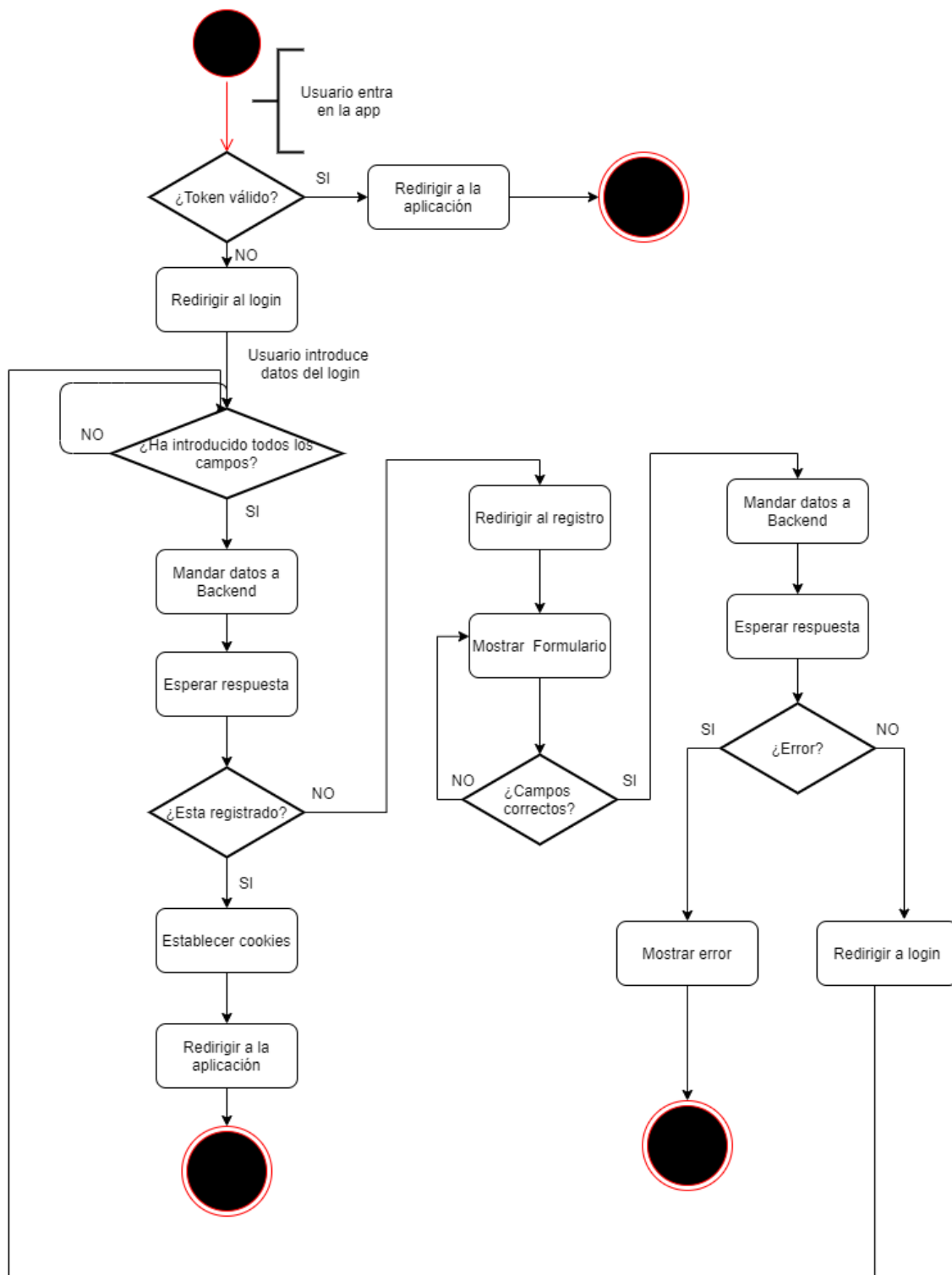


Figura 5.14. Diagrama actividad inicio sesión

5.4.3. Diagrama de paquetes

Los diagramas de actividad expuestos en la sección anterior, nos van a servir de apoyo para identificar, objetos o entidades que van intervenir en la funcionalidad de la capa, es decir, vamos a poder identificar las clases que va a tener la capa. Previamente a especificar el diagrama de clases de la clase Front-end, es necesario concretar como se agruparán las clases de dicha capa, y con qué paquetes o módulos externos a la capa interactuarán las clases de estas. Todo esto se expone en el siguiente diagrama de paquetes:

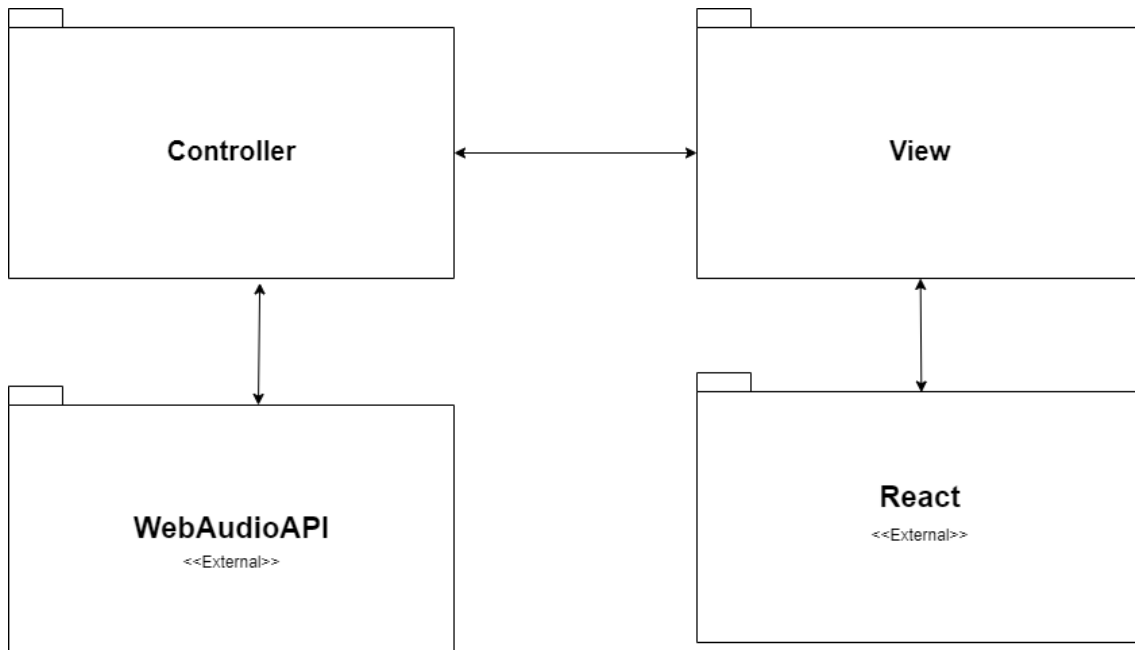


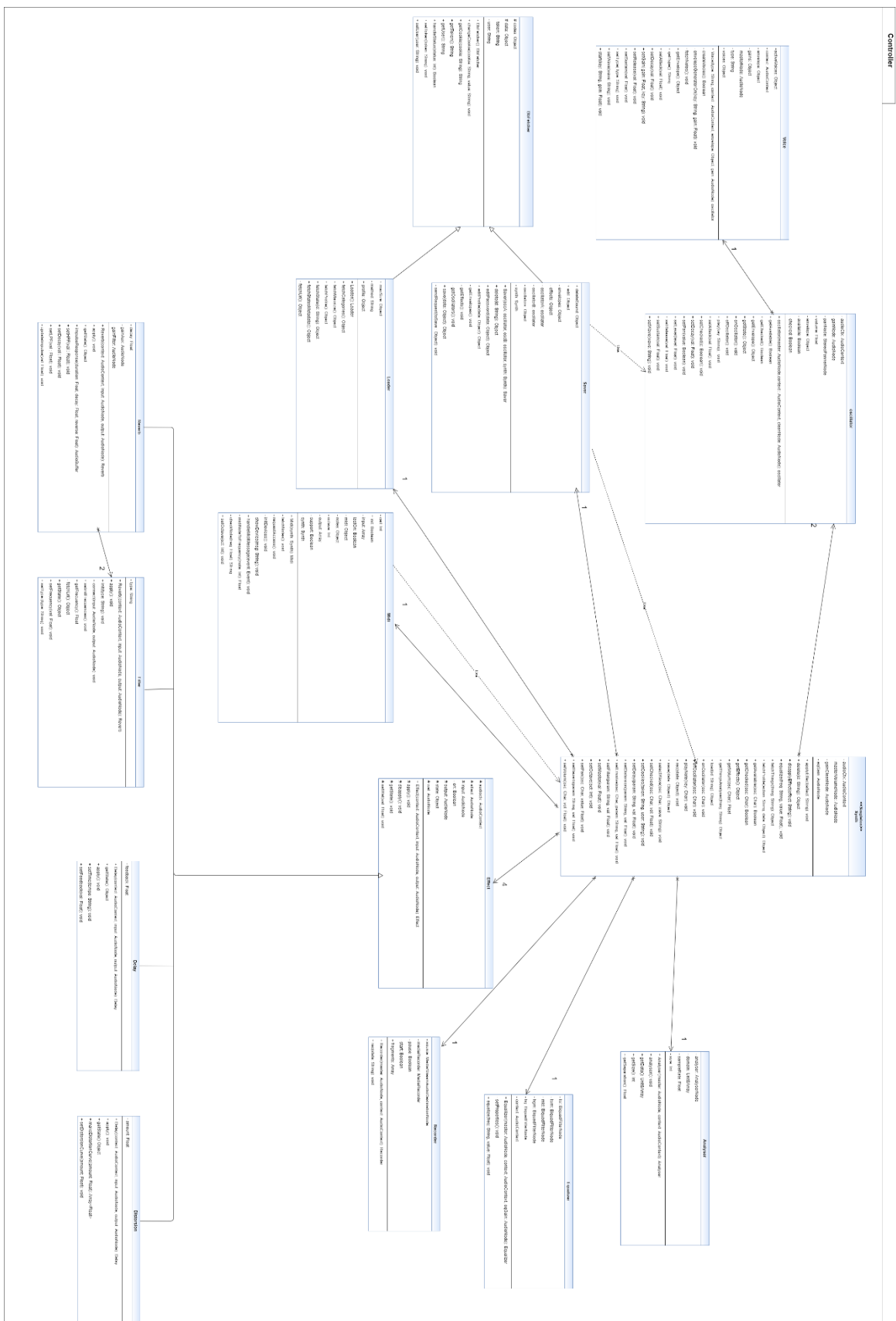
Figura 5.15. Diagrama paquetes capa Front-end

De acuerdo con la figura, se tienen los siguientes paquetes:

- **Controller:** Se corresponde con la subcapa controladora, la cual contiene la clase fachada Synth, este paquete se comunicará tanto con la vista como con la Web Audio API, paquete externo al sistema.
- **View:** Se corresponde con la subcapa de la vista. Contiene la interfaz y se comunicará tanto con el paquete Controller como con las funcionalidades del framework ReactJS.
- **WebAudioApi:** Es externo al sistema. Representa el conjunto la API que ofrece JavaScript para la generación y procesamiento de audio digital.
- **React:** Es externo al sistema. Representa al conjunto de funcionalidades y clases que ofrece el framework ReactJS, el cual servirá de apoyo para el desarrollo de la interfaz.

5.4.4. Diagrama de clases

A partir de los diagramas vistos durante el diseño de la capa Front-end y de la estructura definida en la [sección 5.4.2](#), se definen los siguientes diagramas de clases que representarán la estructura en clases de dicha capa:



De acuerdo con los diagramas de clases, se tendrán en cuenta las siguientes consideraciones a la hora de implementar la capa:

1. Las clases del paquete View solo se comunicarán con la clase Synth (Fachada) del paquete View.
2. Las clases del paquete View utilizarán elementos del paquete React.
3. Solo las clases del paquete Controller se comunicarán con la WebAudioApi.
4. Las clases del paquete View en su mayoría representarán componentes reactivos de la interfaz.

5.4.5. Wireframes

El último paso en el diseño de la capa Front-end será definir los wireframes o bocetos de la interfaz del usuario, los cuales servirán como guía en la implementación de esta. A continuación, se muestran los bocetos de las pantallas disponibles en la interfaz:

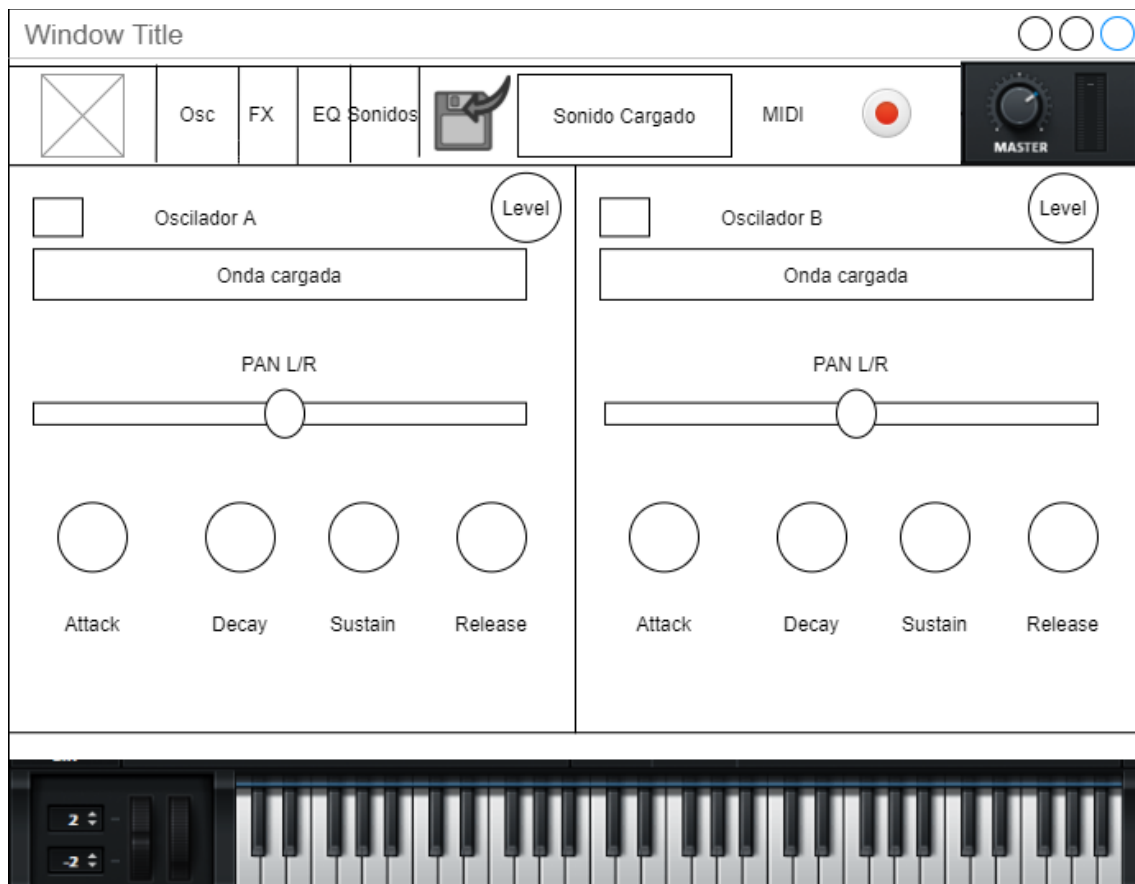


Figura 5.18. Boceto vista Osciladores

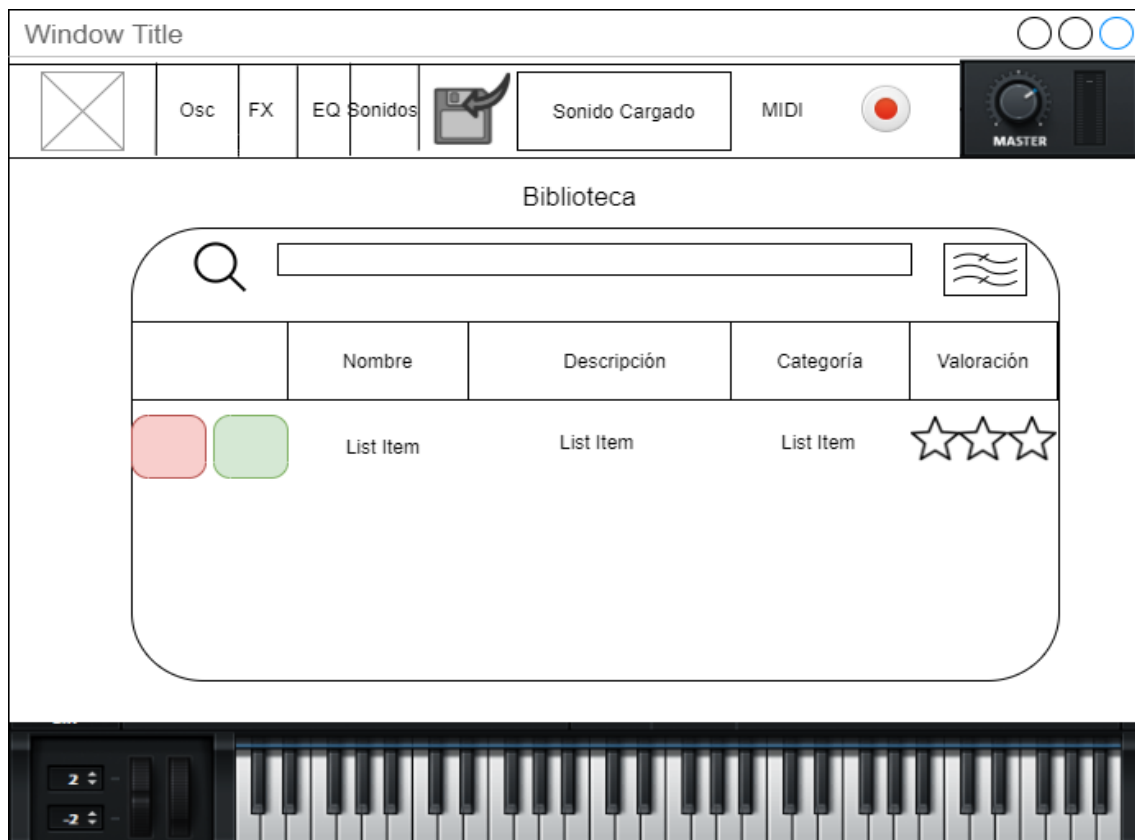


Figura 5.19. Boceto vista Sonidos

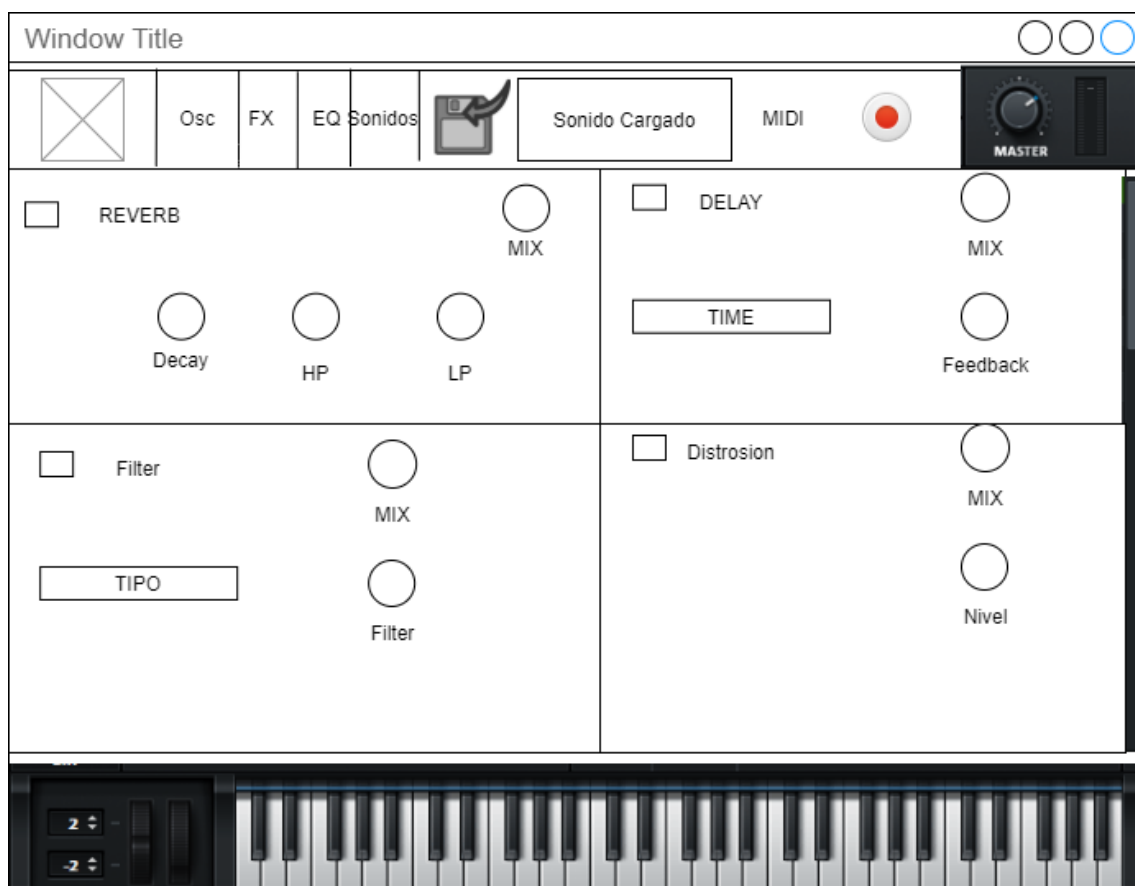


Figura 5.20. Boceto vista Efectos

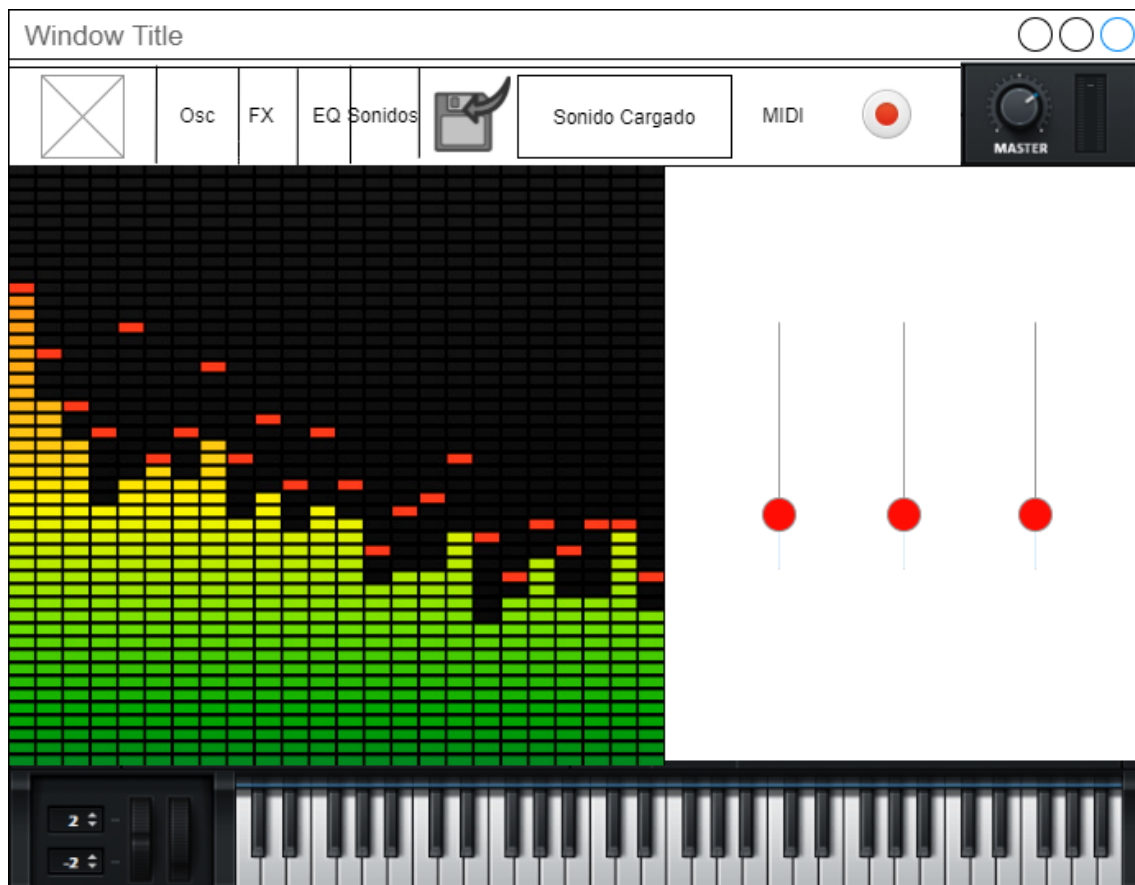


Figura 5.21. Boceto vista EQ

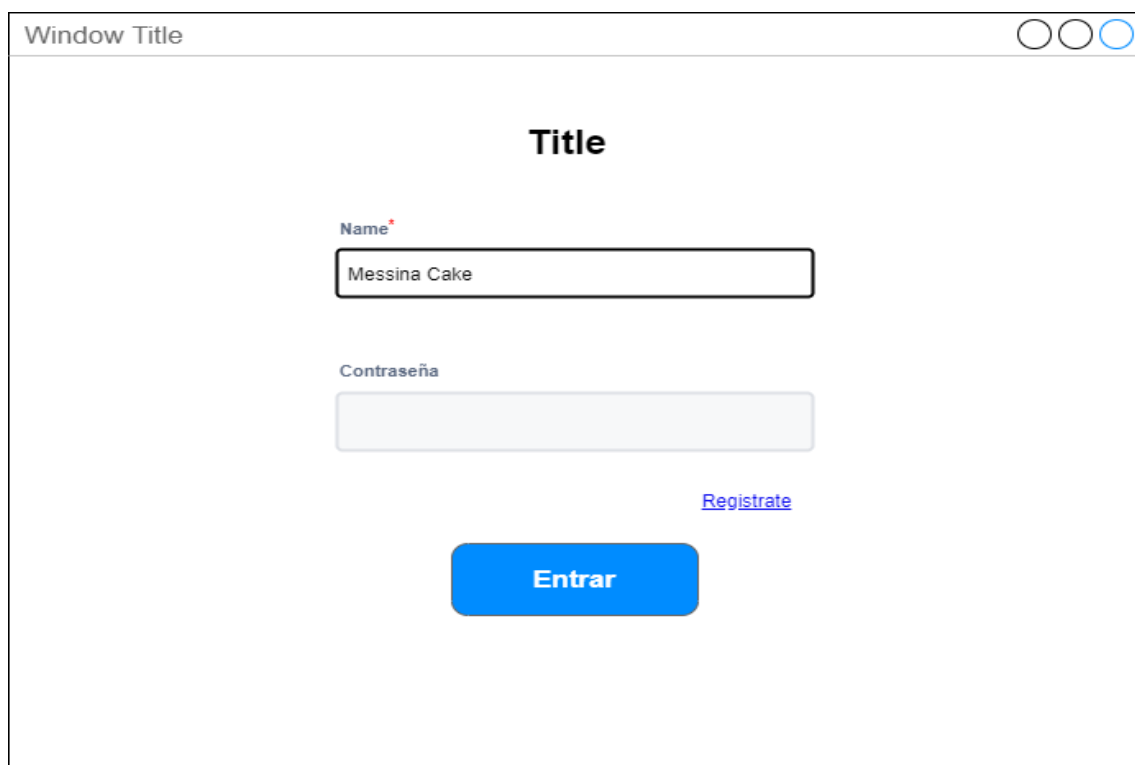


Figura 5.21. Boceto vista Index (Login)

Window Title

Registro

Name*

Email

Fecha


Contraseña

Entrar

Figura 5.22. Boceto vista Registro

Window Title

Perfil



Password

Name*

Email

Fecha

Espacio

Figura 5.23. Boceto vista Perfil

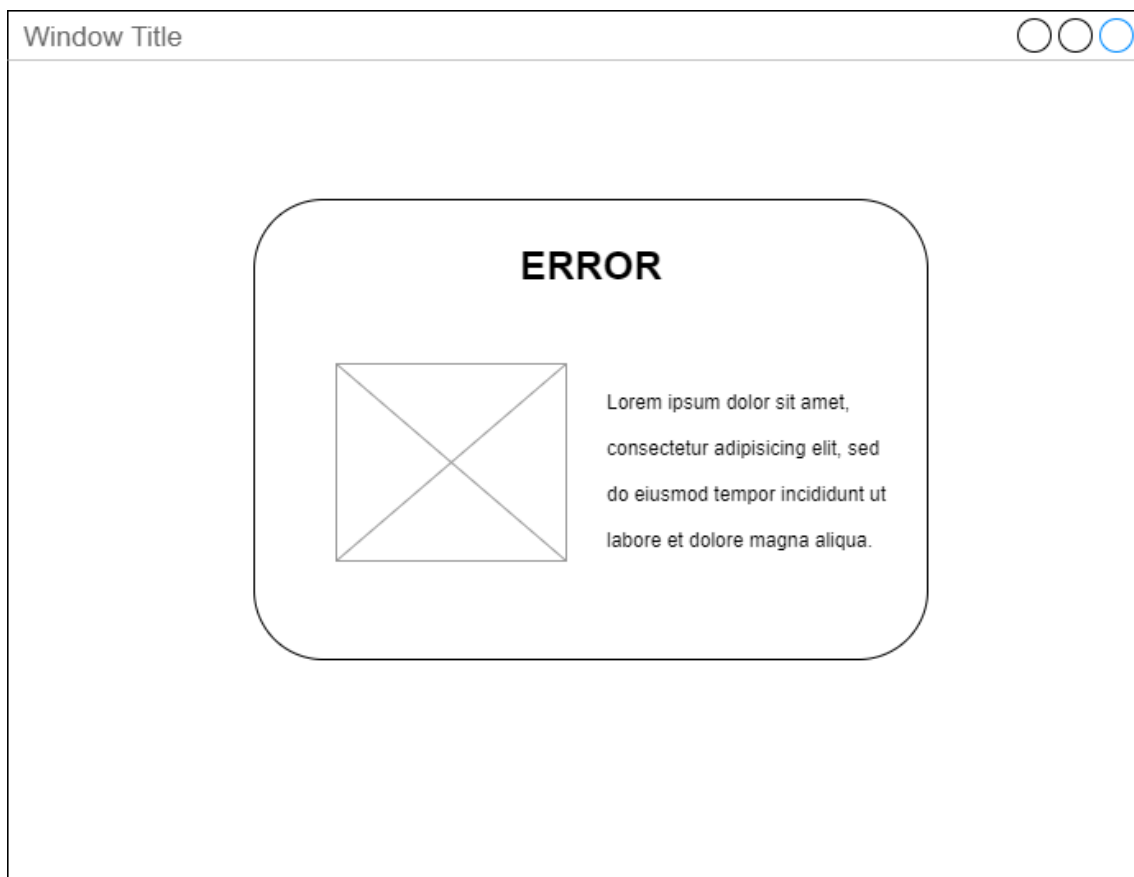


Figura 5.24. Boceto vista Error

Se podrá navegar por las vistas de acuerdo al siguiente esquema de navegación:

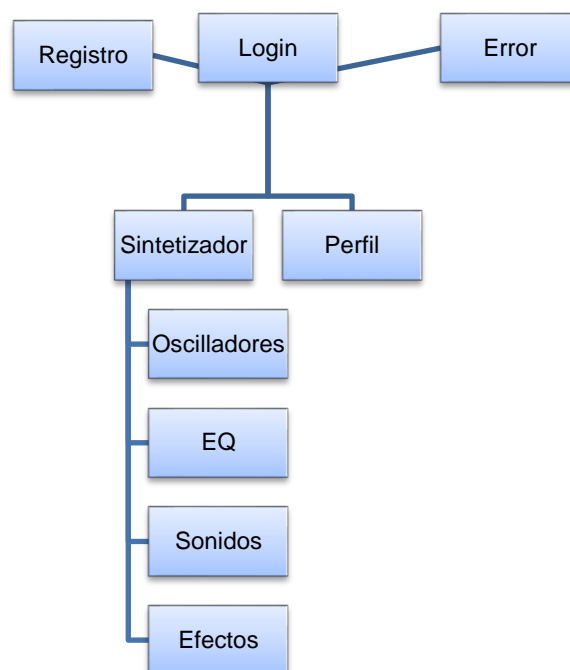


Figura 5.25. Esquema de navegación Interfaz

Capítulo 6: Implementación

Una vez realizado el diseño del sistema software, se procede a explicar con detalle el proceso que comprende la implementación del mismo.

6.1. Estudio de las tecnologías

En esta sección se realizará un análisis y estudio de las tecnologías empleadas en la implementación del sistema expuesto durante esta memoria.

6.1.1. Análisis de las diferentes tecnologías

Para llevar a cabo la implementación del sistema software, se necesita una tecnología que nos ofrezca lo siguiente:

- Soluciones para desarrollar un sistema de tiempo real (multiprocesamiento, temporizadores, semáforos, señales de tiempo real, entrada/salida síncrona y asíncrona etc...).
- Métodos y variables para comunicarnos con el sistema de audio del computador.
- Métodos eficientes para el procesamiento de señales digitales (DSP).
- Documentación para el tratamiento de audio.
- Facilidades para implementar una interfaz de usuario.
- Facilidades para implementar gráficos complejos.

Múltiples lenguajes de programación como Java o Python, ofrecen parcialmente soluciones a los puntos comentados, pero debido a la complejidad del procesamiento de señales digitales es preciso utilizar Frameworks y librerías nativas que simplifiquen esta tarea.

Existen dos lenguajes de programación que nos proporcionan lo expuesto en este punto:

- C++
- JavaScript

A continuación, se describen las ventajas e inconvenientes de estos lenguajes

6.1.1.1. C++

Ventajas

- Dispone de librerías para trabajar con el tiempo, como chrono.
- Ofrece soluciones bastante eficientes para el multiprocesamiento y la sincronización.
- Dispone de librerías para comunicarnos con el sistema de audio del computador.
- Dispone de librerías DSP, como IIPP de Intel.
- La gran mayoría de sistemas software para el procesamiento de audio están implementados en C++ o se basan en este.
- Existen librerías para el procesamiento de gráficos, como Open GL.
- Existen múltiples Frameworks para el desarrollo de interfaces.
- Permite aprovechar al máximo los recursos del ordenador.

Desventajas

- No es portable en nuestro sistema a realizar, necesitaremos realizar una versión para cada SO ya que el sonido se trata de distinta forma en cada uno de estos.
- Los Frameworks existentes no dan soporte a la solución que se busca, ya que están muy limitados a la realización de plugins para DAWs existentes. Con estos Frameworks solo podremos usar nuestro sistema software dentro de un DAW y no podremos realizar una interfaz desde cero.
- Si se usa esta tecnología, el resultado final será de bajo nivel por lo que la dificultad y el tiempo para realizarlo se eleva con creces.
- Debido a la ausencia de Frameworks específicos, se requiere demasiado conocimiento en cuestiones matemáticas y físicas para el tratamiento de las señales de audio digital.

6.1.1.2. JavaScript

Ventajas

- Portable, si se utiliza esta tecnología el software estará disponible para cualquier dispositivo con un navegador.
- Existe una API para este lenguaje llamada Web audio API, destinada a la creación de sonido digital en el ámbito del navegador, la cual se ajusta satisfactoriamente al sistema a desarrollar, ya que ofrece soluciones a alto nivel de síntesis digital que no requerirían de un excesivo conocimiento en cuestiones matemáticas y físicas. Además, da soporte a la comunicación con el sistema de audio del computador.
- Existen Frameworks que proporcionan herramientas para el desarrollo de una aplicación en la nube como React.
- En general, hay una alta disponibilidad de librerías.

Desventajas

- Las capacidades de tiempo real son más limitadas, los callbacks no están sincronizados con precisión. La Web Audio API soluciona parcialmente este problema.
- El ámbito de ejecución, se limita al navegador, por lo que los recursos estarán limitados a este.
- Poco software de síntesis de audio utilizando este lenguaje.
- Da problemas para programas no triviales, al no ser fuertemente tipado.

6.1.2. Conclusión del análisis de tecnologías

Sopesadas las ventajas y desventajas de las tecnologías analizadas en el punto anterior, se llegan a las siguientes conclusiones:

1. Pese a que C++, ofrece más capacidades de tiempo real que JavaScript, la Web Audio API soluciona parcialmente el problema de la sincronización y el tiempo real de manera sencilla por lo que se podría llegar a alcanzar el grado de sincronización requerido.

2. El grado de dificultad en la implementación de la interfaz en JavaScript es más bajo.
3. La ventaja de la total e inmediata disponibilidad de JavaScript aporta un gran valor añadido al producto final.
4. La Web Audio API ofrece muchas soluciones a un nivel más alto que C++, permitiendo que la implementación de este sea más asequible.
5. Debido a la alta disponibilidad de librerías para JavaScript, podremos encontrar soporte y soluciones para gran variedad de los requisitos del programa.

A partir de estas primeras conclusiones, se llega a la conclusión final:

Se elige JavaScript como lenguaje de programación para la implementación del sintetizador, utilizando como entorno de ejecución **NodeJS**. Además, se utilizan la **Web Audio API**, para trabajar con las señales de audio, y los Frameworks ReactJS para la capa Front-end y **Express** para la capa Back-end.

6.2. Tecnologías empleadas

A continuación, se describirán las tecnologías elegidas que se emplearán en la implementación del sistema.

6.2.1. Herramientas

Se han utilizado las siguientes herramientas para el apoyo en la implementación del sistema:

- **Visual Studio Code:** Es un editor de código fuente desarrollado por Microsoft que proporciona una serie de herramientas y extensiones muy útiles para el desarrollo. Como destacado, incluye soporte para depuración, control integrado Git, finalización inteligente de código, fragmentos, resultado de sintaxis, terminal etc...
- **MongoDB Compass:** Es una interfaz para bases de datos MongoDB, que va a permitir conectarse a una base de datos Mongo, visualizar su contenido y realizar modificaciones en esta de una manera visual y sencilla, sin necesidad de introducir comandos por terminal.
- **Postman:** Inicialmente era una extensión para Google Chrome, pero tiene una versión como programa de escritorio, la cual se ha utilizado como apoyo al desarrollo. Este programa permite realizar peticiones HTTP REST a un servidor, mediante una interfaz sencilla e intuitiva, sin la necesidad de desarrollar un cliente. Nos servirá para hacer testeos con la API de la capa Back-end.
- **Google Chrome:** Es un navegador web de Google que se utilizará tanto para visualizar la interfaz del sistema como realizar labores de depuración con el uso de la extensión **DevTools**.

6.2.2. Lenguajes de programación

En este apartado se hablará sobre los lenguajes de programación empleados en la creación del sistema software sobre el que se basa esta memoria, así como de lenguajes de marcado y estilo empleados.

6.2.2.1. JavaScript

El sistema usará JavaScript como lenguaje base de programación, cuyo motivo ya se ha expuesto en la [sección 6.1](#).

Este lenguaje posee las siguientes características: [12]

- Interpretado, es decir, se resuelve en tiempo de ejecución.
- Compilado durante la ejecución del programa.
- Es un lenguaje de scripts, aunque admite estilos de programación orientada a objetos.
- Está basado en prototipos, es decir, los objetos no son creados mediante instanciación de clases, sino mediante la clonación.
- Sintaxis similar a Java y C++.
- Suele usarse como lenguaje de scripting en el navegador, aunque es usado en entornos fuera de este como NodeJS.
- Permite interactuar con el DOM, entre otras capacidades.
- Posee un estándar, ECMAScript, el cuál es soportado por todos los navegadores modernos desde 2012.

6.2.2.2. HTML [13]

HTML son las siglas en inglés para HyperText Markuo Language (Lenguaje de Marcas de Hipertexto), cuyo diseño inicial esta datado en 1991, de la mano de Tim Berners-Lee.

Este lenguaje hace referencia al lenguaje de marcado para la elaboración de páginas web. En 1195 se convirtió en un estándar de la web a cargo de la W3C⁷. Este estándar se ha impuesto en la visualización de páginas web y es usado por todos los navegadores.

Este lenguaje se encarga de definir el significado y la estructura del contenido web mediante el uso de etiquetas rodeadas por corchetes angulares <>. Estas etiquetas aportan una estructura lógica y hacen fácil la interpretación de los códigos tanto por humanos como por máquinas. Además, posee una serie de componentes vitales como los elementos y los atributos.

En el sistema desarrollado servirá tanto de para dotar de una estructura a la interfaz como para que el navegador sea capaz de interpretar la interfaz.

6.2.2.3. CSS [14]

CSS, son las siglas en inglés para Cascading Style Sheet (Hojas de estilo en cascada). Fue propuesto por primera vez por Håkon Wium Lie en 1994 e incluido en la W3C en 1996.

Es un lenguaje orientado para el diseño gráfico, que sirve para dotar de estilos (colores,

⁷ W3C, siglas para World Wide Web Consortium, organización dedicada a la estandarización de la web.

márgenes, bordes) a documentos estructurados escritos con lenguaje de marcado. Comúnmente, es utilizado para dotar de estilo a interfaces escritas en HTML, aunque puede ser aplicado a otro tipo de documentos como XML, SVG, XUL, etc....

En el sistema desarrollado servirá para dotar de estilo a los documentos HTML que componen la interfaz.

6.2.3. Entorno de Ejecución [15]

Se utilizará como entorno de ejecución **NodeJS**. Node es un entorno de ejecución multiplataforma, de código abierto, basado en JavaScript y orientado a eventos asíncronos con E/S de datos, basado en el motor V8 de Google (motor para JavaScript y WebAssembly).

Fue creado en 2009 por Ryan Dahl y su evolución recae en manos de la empresa Joyent, empresa californiana especializada en virtualización de aplicaciones y cloud computing, en la cual Ryan es miembro de la plantilla.

Es un entorno pensado para la capa del servidor, el cual permite levantar un servidor para que se mantenga a la escucha en un puerto concreto. Permite trabajar con un protocolo HTTP y está pensado para trabajar sin hebras.

En nuestro sistema nos proporcionará un servidor para la capa Front-end y otro para la capa Back-end de manera que ambas capas puedan dar servicio, en otras palabras, nos va a permitir ejecutar ambas capas por separado.

6.2.4. Frameworks

Dado que se utiliza NodeJS como entorno de ejecución, a continuación, se detallan los Frameworks utilizados de este entorno.

6.2.4.1. ReactJS [16]

React es un Framework para JavaScript de código abierto, creado por Jordan Walk, un ingeniero software de Facebook, cuya evolución y mantenimiento recae principalmente bajo su responsabilidad, aunque también participa la comunidad.

Este Framework es soportado por NodeJS y permite crear interfaces de usuario a partir de una serie de funcionalidades y objetos que facilitan el proceso de creación. Su característica principal, es la posibilidad de construir aplicaciones que usan datos que cambian todo el tiempo, es decir, permite crear interfaces reactivas de una manera sencilla, que cambian cuando los datos de estas son alterados, de ahí esa reactividad (reaccionan ante los cambios).

Estas interfaces son construidas por medio de componentes, los cuales, encapsulan una funcionalidad relacionada con el componente y renderizan un fragmento de código HTML, aunque no necesariamente, ya que un componente puede estar compuesto a su vez de otros componentes y no incluir código HTML, si no que envía la orden de renderizado a esos componentes hijos. De esta manera la interfaz resultante, será un conjunto de componentes relacionados entre sí.

La construcción de los componentes puede realizarse mediante funciones que devuelvan el componente, o mediante clases. Este último caso será el que se utilizará en la implementación de la capa Front-end, por lo que todos los componentes heredaran

o de **React.Component** o de otro componente que herede de esta clase.

React posee un DOM virtual propio, de manera que compara su DOM con el del navegador para determinar que partes han cambiado, y así actualizar de una manera más eficiente este.

Una de las características de este framework es que incorpora los conceptos de propiedades (props) y estado (state). Las propiedades son los atributos que configuran los componentes, los cuales son recibidos a un nivel superior. El estado representa una instancia del componente, es decir, una representación de este en un momento determinado.

Además, React se caracteriza por seguir un ciclo de vida, una serie de estados por los cuales pasan todos los componentes a lo largo de su existencia, los cuales se muestran en el siguiente esquema:

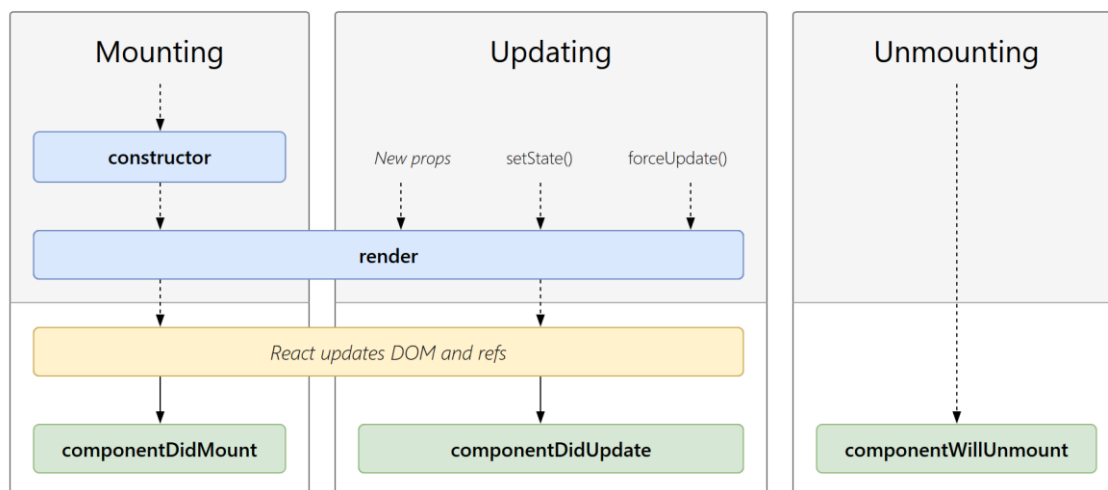


Figura 6.1. Esquema del ciclo de vida de React

Fuente: <https://medium.com/@drewisatlas/react-lifecycle-methods-what-are-they-when-do-we-use-them-edccb5ec860a>

Como se puede observar en el esquema, el ciclo de vida de React pasa por tres estados principales:

1. **Mounting:** Se inicia cuando se invoca el constructor del componente, y el cual va a pasar por una serie de sub-estados hasta que se renderiza (después de `componentDidMount`).
2. **Updating:** Tiene lugar cuando se cambia el estado o el valor de alguna propiedad y provoca que el componente se vuelva a renderizar.
3. **Unmounting:** Tiene lugar cuando el componente se destruye.

Por último, cabe destacar, que ReactJS utiliza una sintaxis similar a la de HTML, llamada **JSX**, la cual constituye una extensión a la sintaxis de JavaScript que permite combinar la sintaxis de JavaScript con la de HTML.

6.2.4.2. Express [17]

Es un Framework diseñado exclusivamente para NodeJS por TJ Holowaychuk en 2010. En 2015 los derechos de administración del proyecto fueron adquiridos por IBM para que posteriormente asignará la dirección del proyecto a la fundación NodeJS.

Este Framework es gratuito y de código abierto. Ofrece una serie de funcionalidades para crear el backend de una aplicación web y APIs,

En la implementación de nuestro sistema software se utilizará para implementar la API de la capa back-end que atenderá las peticiones HTTP procedentes de la capa Front-end.

6.2.4.2. Mongoose [18]

Es un Framework diseñado exclusivamente para NodeJS. Al igual que los anteriores, es gratuito y de software libre cuya evolución y mantenimiento recae en manos de la comunidad y de múltiples empresas colaboradoras en el proyecto.

Mongoose ofrece una serie de funcionalidades para escribir consultas para una base de datos MongoDB de una manera sencilla, implementar validaciones, construcción de middlewares y algunas otras que permiten enriquecer la funcionalidad de la base de datos. Además, permite definir esquemas para modelar la base de datos de manera que se podrá tratar a las colecciones o documentos como objetos.

En nuestro sistema se utilizará para definir los esquemas de la base de datos, enriquecer la funcionalidad, comunicarse con la base de datos, realizar consultas y alterar el estado de esta.

6.2.5. Web Audio API [19]

Aunque no es una tecnología como tal, y tampoco representa un Framework. Es importante hablar de ella en esta sección de tecnologías empleadas, ya que durante el desarrollo de este capítulo se mencionará en varias ocasiones.

La Web Audio API representa un conjunto de clases y métodos para generar, procesar y gestionar el audio en la web. Esta API es nativa de JavaScript por lo que no es necesario ninguna instalación, ya que es propia del lenguaje.

Esta API permite trabajar el audio en la web a partir de nodos con funcionalidades diversas conectados entre sí en forma de grafo. Este grafo, es lo que se denomina el **contexto de audio**. Los nodos de audios tendrán una serie de entradas y salidas, por las cuales pasarán un flujo de datos y se procesarán. Estos flujos de datos son matrices de intensidades de sonido (muestras) correspondientes a segmentos de tiempo muy pequeños (por segundo suelen haber decenas de miles de muestras). Las matrices pueden ser calculadas de manera matemática o extraídas directamente de muestras de audio. En nuestro sistema software se utilizarán en su mayoría las calculadas matemáticamente. Estas matrices son modificadas de una manera u otra a medida que van pasando por los distintos nodos.

Este procesamiento del sonido se denomina muestreo, que consiste en convertir una señal continua, como una onda sinusoidal producida por un oscilador, en una señal discreta, la cual se corresponde con la matriz de muestras.

Cuando el sonido (las muestras) ha sido procesado lo suficiente, este se enviará a un nodo de destino que se encargará de enviarlo a los altavoces para su reproducción, los cuales utilizarán las muestras para reproducir el sonido.

Durante el desarrollo de este capítulo se entrará más a fondo en todas estas cuestiones.

6.2.6. Base de datos [9]

La base de datos que alojará el modelo de datos que utilizará nuestro sistema software será una base de datos MongoDB.

MongoDB surge en 2007 de la mano de MongoDB Inc como una alternativa a las bases de datos SQL. Estas bases de datos son de código abierto, NoSQL⁸, orientado a documentos y de código abierto.

La principal diferencia con las bases de datos SQL tradicionales es que los datos no se guardan en tablas, sino que se guardan en objetos BSON (similares a JSON). Estos objetos organizan la información de manera dinámica sin la necesidad de usar un identificador que sirva para representar la relación entre un conjunto de datos y otros, es decir, los objetos no tienen porque tener una clave primaria (de ahí que se les llame base de datos no relacionales).

MongoDB sustituye las tablas por colecciones, y las filas por documentos que se corresponde con objetos BSON.

6.2.7. Control de versiones

Será necesario controlar los cambios en el código fuente del proyecto y poder tener varias versiones de este ante posibles fallos o cambios en los requerimientos del sistema a lo largo del desarrollo del proyecto. Para ello se utilizará **Git** como tecnología para el control de versiones.

Git fue diseñado y lanzado por Linus Torvalds en 2007 como una solución para asegurar el mantenimiento, compatibilidad y eficiencia de las versiones de los programas cuando estos cuentan con un elevado número de archivos de código fuente. Es un software libre y su mantenimiento está en manos de Junio Hamano. [20]

Para nuestro sistema se utilizará esta tecnología para el control de las versiones de un repositorio con todo el código fuente de este en la plataforma de **Github**, el cual estará actualizado con la última versión del código fuente, pudiendo volver a una versión anterior cuando se quiera o incluso crear ramas con versiones distintas de código.

⁸ NoSQL hace referencia a bases de datos que no utilizan SQL para las consultas

6.3. Instalación y ejecución

6.3.1. Instalación

Dado que el sistema está pensado para ser desplegado en un servidor, se explicará el proceso de instalación para Windows y Linux. Antes de comenzar con la explicación es necesario que el computador en el que se está instalando el sistema software tenga acceso a Internet, ya que se requiere su uso durante este proceso.

Lo pasos de instalación del sistema software serán los siguientes:

1. Instalar la última versión de NodeJS:

- a. Si trabajamos con Windows:
 - i. Descargamos el Instalador para Windows (.zip) desde el sitio web⁹ de NodeJS
 - ii. Descomprimos el .zip, y ejecutamos el archivo .exe
 - iii. Una vez ejecutamos se nos abrirá el instalador de Windows y pulsamos en Siguiente.
 - iv. Abrimos una terminal y ejecutamos `node -v` para comprobar si se ha instalado correctamente
- b. Si trabajamos con Linux ejecutamos por terminal lo siguiente:
 - i. `sudo apt update`
 - ii. `sudo apt install nodejs`
 - iii. `nodejs -v` para comprobar que se ha realizado correctamente

2. Instalar el gestor de paquetes npm de NodeJS:

- a. Si trabajamos con Windows
 - i. El gestor de paquetes se instaló junto con la instalación de Nodejs al ejecutar el .exe
- b. Si trabajamos con Linux ejecutaremos por terminal lo siguiente:
 - i. `sudo apt install npm`

3. Instalar Mongo DB:

- a. Si trabajamos con Windows:
 - i. Descargar la versión para Windows (.zip) desde el sitio web de Mongo DB
 - ii. Descomprimos el .zip y ejecutamos el archivo .exe
 - iii. Una vez ejecutamos se nos abrirá el instalador de Windows y pulsamos en Siguiente.
 - iv. Abrimos una terminal y ejecutamos `db.version()` para ver si se ha realizado correctamente la instalación
- b. Si trabajamos con Linux
 - i. `sudo apt update`
 - ii. `sudo apt install -y mongodb`

4. Clonar el repositorio de Github con el código fuente

- a. Tanto en Windows como en Linux ejecutaremos en una terminal
 - i. `git clone https://github.com/migueg/Sintetizador-Virtual-TFG-UGR.git`

5. Instalar todas las dependencias:

- a. Tanto en Windows como en Linux:

⁹ Web Nodejs: <https://nodejs.org/es/download/>

- i. Nos situamos en la carpeta Frontend del repositorio clonado desde terminal y ejecutamos `npm install`
- ii. Nos situamos en la carpeta Backend del repositorio clonado desde terminal y ejecutamos `npm install`

Nota: El gestor de dependencias npm de Nodejs, instalará automáticamente en el servidor todas las dependencias que aparecen en el package.json que durante la etapa de desarrollo fueron añadidas a este.

6.3.2. Ejecución

Para la ejecución del sistema es necesario ejecutar cada una de las capas que componen a este por separado, para ello hay que realizar los siguientes pasos:

1. Arrancar el servicio MongoDB, en el caso de que este no se encontrara en ejecución, en el puerto 27017.
2. Situar por terminal en el directorio Back-end y ejecutar la siguiente instrucción → `node server.js`.
 - a. Esto ejecutará la capa Back-end en el puerto 8080, asique este puerto debe de estar libre en el momento de su ejecución.
3. Situar por terminal en el directorio Frontend/react-crud y ejecutar la siguiente instrucción → `npm start`
 - a. Esto ejecutará la capa Front-end en el puerto 3000, asique este puerto debe de estar libre en el momento de su ejecución.
4. Una vez realizados los pasos anteriores, desde un navegador web acceder a la dirección → `localhost:3000`, la cual nos dará acceso a la interfaz.

6.4. Implementación

A continuación, se va a proceder a una explicación al detalle del proceso de implementación del sistema en cuestión.

A rasgos generales, la implementación se ha dividido acorde con la arquitectura a capas del sistema, aunque la capa de datos ha estado estrechamente unida al desarrollo de la capa Back-end. De esta manera podemos dividir la implementación en dos partes: Backend, que comprende la capa de datos y la capa Back-end y Frontend, la cual comprende la capa Front-end del sistema, siendo esta última la más completa, laboriosa y de más difícil desarrollo.

Por consiguiente, esta sección se divide en esas dos partes comentadas.

6.4.1. Frontend

Gran parte del procesamiento del sistema se lleva a cabo en la capa Front-end, es decir, gran parte del sistema se ejecutará en el navegador del cliente, siendo la capa Front-end la más compleja y mayor tamaño.

Para cumplir con los objetivos del sistema se necesita buscar la inmediatez, la mayor sincronía posible y evitar los retardos, ya que la experiencia de tocar un instrumento o en este caso un sintetizador virtual se vería afectada con creces. Además, como el sistema se trata de un sistema que genera y reproduce señales de audio se necesita trabajar directamente con el contexto de audio del dispositivo del cliente, es decir, los métodos para reproducir señales de audio tienen que ejecutarse en su dispositivo.

Es por esta razón por la que la mayor parte del procesamiento se realiza en la capa

Front-end.

Al trasladar la mayor parte del procesamiento a la capa Front-end, se reduce al mínimo cualquier retardo que se pudiera producir en una llamada al servidor de la capa Back-end, y se realiza una comunicación directa con el dispositivo del cliente que es requerido para reproducir las señales de audio generadas.

A continuación, se explica con detalle todas estas cuestiones.

6.4.1.1. Estructura

Antes de entrar en detalles técnicos sobre la implementación de la capa Front-end, es necesario, entender la estructura de directorios y ficheros que compone la capa. Dicha estructura se muestra en la siguiente imagen:

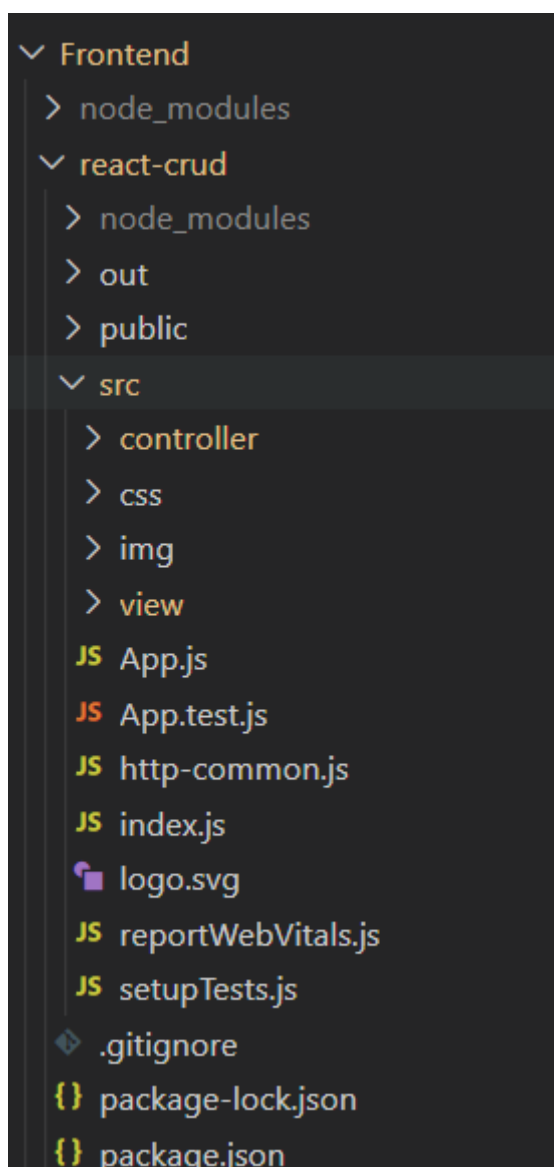


Figura 6.2. Estructura de directorios y ficheros de la capa Front-end

A partir de la imagen, se puede observar lo siguiente:

- **react-crud:** Es el directorio principal que contiene toda la funcionalidad de la capa Front-end, De este directorio cuelgan los siguientes:
 - **node_modules:** alberga los módulos de NodeJS necesarios para que React funcione adecuadamente.
 - **out:** Contiene la API doc de la capa. Dentro de este directorio se encuentra un fichero HTML llamado index.html, el cual, si es abierto en un navegador, se podrá consultar las funciones, clases y variables utilizadas en la capa con sus respectivas documentaciones. Esta API doc ha sido generada automáticamente a partir de los comentarios en el código utilizando un paquete para NodeJS llamado YUIDoc¹⁰.
 - **public:** Se encuentran los elementos públicos de la capa.
 - **src:** Este directorio contiene el código fuente implementado para dotar de funcionalidad a la capa. Dentro de él, destacan los siguientes directorios y ficheros:
 - **Controller:** Se corresponde con la funcionalidad de la subcapa controladora de la capa Front-end.
 - **Css:** Contiene las hojas de estilo, es decir, los ficheros css.
 - **View:** Contiene la funcionalidad de la subcapa vista de la capa Front-end.
 - **App.js:** Este fichero contiene y renderiza los elementos que componen el sintetizador.
 - **Index.js:** Fichero que se encarga de crear y renderizar toda la interfaz.
 - **.gitignore:** Es un fichero de configuración de Git que se encarga de ignorar ciertos cambios en determinados directorios o ficheros cuando se producen, por ejemplo, ignora los cambios producidos en el directorio node_modules ya que los cambios en este se quedan registrados en el package.json.
 - **package.json:** Contiene todas las dependencias de la capa necesarias para el correcto funcionamiento de esta. Este fichero se modifica cada vez que se añade una nueva y es necesario para que NodeJS instale las dependencias necesarias cuando se instala el sistema. Mantiene un registro de que dependencias se necesitan y con que versiones.

6.4.1.2. Sintetizador

Ya comprendida la estructura de directorios que compone a la capa, se procede a exponer los detalles técnicos que envuelven la implementación de la capa.

El objetivo final del sistema en su conjunto es lograr un sintetizador virtual que emule a uno analógico. Como se mencionó en la introducción ([ver Capítulo 1](#)), hay varios tipos de síntesis de audio. En este sistema se va a implementar una **síntesis aditiva**, de manera que la señal de audio resultante será la suma de todas las señales generadas. Este proceso de síntesis será logrado, en gran medida, mediante el uso de la web Audio API, la cual será utilizada exclusivamente en la capa Front-end para el retardo en el sistema.

La síntesis desarrollada en este sistema seguirá el siguiente esquema:

¹⁰ YUIDoc: <https://yui.github.io/yuidoc/>

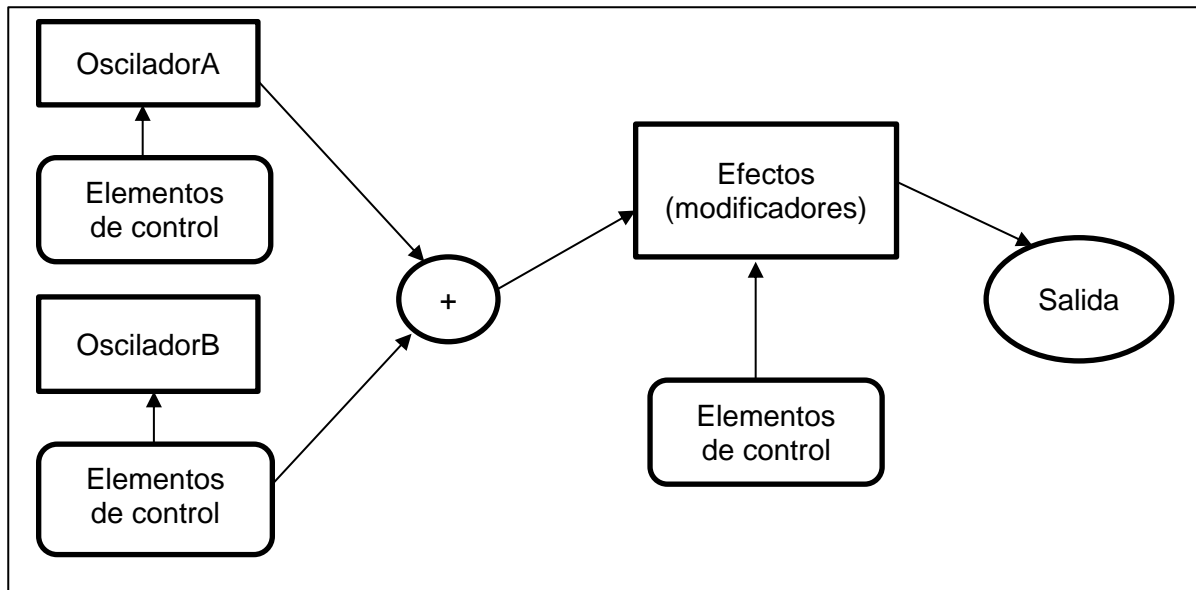


Figura 6.3. Síntesis Aditiva

De acuerdo con el esquema anterior, la señal que el usuario percibirá cuando utilice el sistema será el resultado de sumar:

- Dos osciladores, que contendrán a su vez un número determinado de voces, es decir, de frecuencias de oscilación.
- Efectos, que modificarán la señal producida por los osciladores.

Como se aprecia en el esquema, sobre estos elementos actuarán elementos de control que controlarán los parámetros de estos y que durante el desarrollo de este capítulo serán explicados al detalle.

Esta síntesis aditiva será combinada con una **síntesis sustractiva** mediante el uso de un ecualizador, el cuál será expuesto en la [sección 6.4.1.3.2.3](#).

El objetivo de la capa Front-end será representar el sintetizador de manera gráfica e implementar la lógica necesaria para generar y reproducir sonidos, ya sea interaccionando con la web Audio Api o con la capa Back-end

Dado que la capa Front-end sigue una arquitectura a su vez en capas ([ver sección 5.4.1](#)), para llevar a cabo el sintetizador, es conveniente diferenciar entre las capas de la arquitectura, ya que cada una representa un papel importante a la hora de obtener una señal de audio digital que es el objetivo final del sistema. No obstante, en el proceso de síntesis no solo interviene la capa Front-end, sino que también intervienen las otras capas, aunque gran parte de la lógica se concentra en esta.

En la subcapa **controladora**, se representa el sintetizador mediante la **clase Synth** la clase fachada. Cada vez que el usuario ejecuta interaccione con la vista, está llamará a un método de la clase Synth. Por otro lado, en la subcapa de la vista la clase que representará el sintetizador será la clase App, que contendrá todos los componentes que lo forman.

6.4.1.3. Subcapa controladora

En este apartado se detallará el proceso de implementación de la subcapa controladora. La implementación de esta estará determinada por los diagramas de actividad de la [sección 5.4.3](#), de los cuales se pueden identificar los siguientes cursos de acción:

- **Generación de audio**
- **Reproducción de audio sin MIDI**
- **Reproducción de audio con MIDI**
- **Aplicación de efectos**
- **Gráficos**
- **Comunicación con capa Back-end**

A continuación, se detallan cada uno de ellos.

6.4.1.3.1. Generación de audio

Uno de los objetivos de este proyecto es la generación de señales de audio de manera digital. Para obtener una señal de audio digital es necesario el uso de un oscilador, que generará dicha señal, en el caso del sistema desarrollado, matemáticamente.

El concepto de oscilador, hace referencia a los osciladores de audio analógicos que generaban sonidos mediante circuitos electrónicos. En el mundo digital este concepto es algo abstracto y el oscilador comprenderá un conjunto de métodos que nos proporcionarán el sonido final.

El proceso necesario para generar sonidos de manera digital y por consiguiente implementar un oscilador es algo tedioso que requiere de funciones matemáticas y procesos para comunicarnos con el hardware del ordenador. Sin embargo, podemos aprovechar las capacidades que nos ofrece JavaScript y los navegadores en los que se ejecuta. Parte de estas capacidades nos las proporciona directamente la **web Audio Api** (ver [sección 6.2.5](#)).

Esta generación de audio digital se consigue en la subcapa controladora, aunque intervienen el resto de las capas

Para entender como interviene la subcapa controladora en la generación de audio es necesario aclarar una serie de cuestiones técnicas. Como se está utilizando la Web Audio Api, el elemento principal para generar una señal de audio es un **contexto de audio (AudioContext)**.

6.4.1.3.1.1 AudioContext

El contexto de audio es una interfaz que representa un grafo de procesamiento de audio compuesto por **nodos de audio (AudioNode)** conectados entre sí. Este se va encargar tanto de la creación de los nodos de audio como la de la ejecución del procesamiento de audio. [20]

Los nodos de audio pueden ser de diversos tipos atendiendo a sus entradas y a sus salidas, así que se pueden clasificar de la siguiente manera: [20]

- *Sin entradas o generadores:* Suelen ser usados para generar sonidos. No reciben otros nodos de entrada, pero si pueden contar con múltiples salidas, es decir, su salida puede estar conectada a varios nodos.
- *Procesadores:* Tienen entradas y salidas, pueden estar conectados a otros

nodos por ambos lados. Se encargan de procesar una entrada y generar una salida.

- *De destino:* Cuenta con entradas, pero no tiene salidas, ya que todas las entradas que llegan a este nodo son reproducidas directamente en nuestros altavoces, es decir, manda la señal al hardware del ordenador.

Cada entrada o salida se compone de canales de audio, como puede ser el mono o el estéreo.

Los nodos de audio solo pueden pertenecer a un contexto y se necesita poner especial atención en este último, ya que un contexto con un exceso de nodos en el grafo o con nodos mal situados en él nos pueden generar una latencia alta y por tanto no deseada que afectaría de manera negativa a los objetivos y requisitos del sistema. [20]

Entre los nodos del Audiocontext va a existir un flujo de datos, este flujo de datos se corresponde con las matrices de muestras, las cuales representan los sonidos generados. Estas matrices se corresponden con el **AudioBuffer**. El AudioBuffer es una matriz de la siguiente forma:



Figura 6.4. Representación del AudioBuffer

A partir de la anterior figura, podemos observar lo siguiente: [19]

- Cada fila de la matriz del AudioBuffer se corresponde con un **canal de audio** del sonido, es decir, supongamos que el sonido generado es estéreo, y un sonido en estéreo contiene información en dos canales distintos, por lo que habrá dos filas, ya que los sonidos en estéreo utilizan dos canales el L (izquierdo) y R (derecho) cada uno con información distinta, por lo que el número de filas dependerá de los canales de audio que componen el sonido.
- Cada columna se corresponde con un fotograma, que se corresponden a su vez con unos valores de muestras que se producen en un instante concreto. Hay que tener en cuenta lo siguiente:
 - El **número de fotogramas** dependerá de la frecuencia de muestreo que tengamos
 - La **frecuencia de muestreo** es el número de fotogramas que se reproducirán en un segundo, cuya unidad es el Hz. Cuanto mayor sea este número, mayor calidad tendrá el sonido generado. En nuestro caso se trabajará con una frecuencia de muestreo de **48000 Hz**, la cual aporta gran calidad. Además, la longitud del AudioBuffer se corresponde con esta cifra.
 - En cada fotograma habrá una muestra correspondiente al mismo instante

de tiempo que el resto de las muestras de ese fotograma pertenecientes a los otros canales

- Cada celda se corresponde con una muestra, la cual representa el valor del flujo de audio en un tipo de dato float32.
- La matriz en su totalidad representa un sonido con una duración de X segundos determinada por la longitud de esta, el cual se reproducirá cuando llegue a los altavoces.

Habiendo entendido la base sobre la que se sustenta la Web Audio API, para conseguir emular a un oscilador analógico y por consiguiente generar señales de audio, necesitaremos crear un Audiocontext en el que podamos ir añadiendo nodos de audio para que intercambien, modifiquen y procesen los AudioBuffers que finalmente se reproducirán.

El primer paso será crear el Audiocontext en la clase **Synth**, ya que el deberá ser el mismo para todos los nodos de audio del sintetizador. Concretamente, se crea en el constructor de dicha clase en la siguiente sentencia:

```
constructor(){  
  var AudioContext = window.AudioContext // Default  
  || window.webkitAudioContext;  
  this.#audioCtx = new AudioContext();  
  
  ...  
}
```

Código 6.1: Creación del contexto de audio

Este contexto, será el único contexto del sistema y será utilizado por aquellos elementos o nodos que tengan que enviar alguna señal para su reproducción en los altavoces del dispositivo.

El siguiente paso en la generación de los sonidos será crear dos osciladores para lograr la síntesis deseada en el sistema. Para ello, se creará la clase **oscillator**, que representará un oscilador, de la cual se utilizan dos instancias para simular la existencia de dos osciladores, más tarde se verá con más detalle este aspecto. Estas instancias las contiene la clase Synth:

```
class Synth{  
  #oscillatorA;  
  #oscillatorB;  
  ....  
  constructor(){  
    this.#oscillatorA = new oscillator(this.#masterVolumeNode,this.#audioCtx,this.  
#gainCleanNode);  
    this.#oscillatorB = new oscillator(this.#masterVolumeNode,this.#audioCtx,this.  
#gainCleanNode);  
    ....  
  }  
  ....  
}
```

Código 6.2: Creación del contexto de audio

6.4.1.3.1.1 Polifonía

Como ya se ha mencionado el concepto de oscilador en el mundo digital es algo abstracto y en este punto tenemos un claro ejemplo de ello.

Un solo oscilador analógico es capaz de oscilar varias frecuencias de oscilación de manera simultánea, pudiendo reproducir un acorde, sin embargo, en el mundo digital esto no es siempre posible ya que se necesitan ejecutar funciones de manera sincronizada, encontrándonos de esta manera el primer problema a afrontar, ya que JavaScript no admite la creación de hebras.

El web Audio Api proporciona un nodo oscilador (**OscillatorNode**) que representa una forma de audio periódica, como puede ser una onda sinusoidal. Este nodo es un módulo de otro nodo llamado **AudioScheduledSourceNode** que hace que se cree una frecuencia específica de una onda determinada constante, es decir, crea un tono o nota constante (genera AudioBuffers). [21]

Sin embargo, el OscillatorNode solo es capaz de generar una nota o frecuencia de manera simultánea, siendo imposible la polifonía con un solo oscilador, es decir, no se podría tocar un acorde con un solo oscilador del web audio API. Surge de esta manera la siguiente cuestión, *¿cómo podemos lograr la polifonía usando la Web Audio Api de JavaScript?*

Esta cuestión podría ser abordada de distintas formas. Una manera de abordarla es crear por cada nota un OscillatorNode que va a generar la frecuencia correspondiente a la nota. Fijando el sistema de referencia en las notas de un piano, tendremos 85 OscillatorNode por cada instancia de la clase oscilador, es decir uno por cada tecla del piano. Esto se refleja en el siguiente esquema:

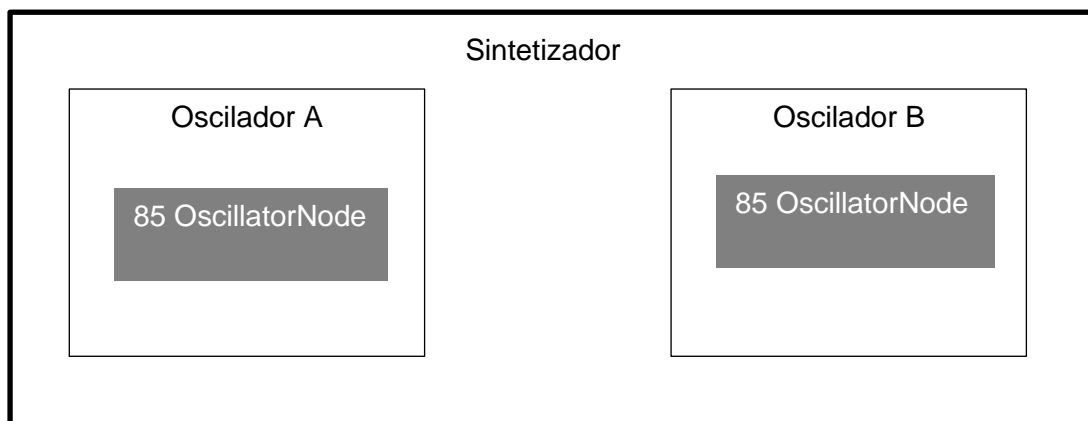


Figura 6.5. Representación de los osciladores

Para representar todo esto en el código, se crea una clase **Voice** que encapsulará el conjunto de voces o OscillatorNode. Esta clase es instanciada en la clase oscilador para poder trabajar con ella:

```

constructor(){
...
    this.voice = new Voice('sine',this.#audioCtx, this.#envelope,this.#gainNode);
...
}

```

Código 6.3: Creación polifonía en la clase oscillator

Los parámetros que recibe en el constructor son los siguientes:

1. Tipo de onda (Más adelante se explicará al detalle).
2. Contexto de audio del sistema.
3. La envolvente (Más adelante se explicará al detalle).
4. Nodo de ganancia del oscilador, donde se mezclarán todas las voces de este produciendo un oscilador final.

6.4.1.3.1.1.2 Creación de voces

Ya que necesitamos un conjunto de OscillatorNode o voces para lograr la polifonía es necesario crearlos uno a uno. La creación de las voces ocurre cuando se crea una instancia de la clase Voice, en donde ocurre lo siguiente:

1. Cuando se llama al constructor se realiza una petición a la API de la capa Back-end, en concreto al **endpoint "notes"** que devolverá un objeto JSON con parejas clave valor de frecuencia y nota, por ejemplo, A3: 440, y las almacena en una variable después de su decodificación. De esta manera se preserva la arquitectura a capas, evitando el exceso de datos en la capa Front-end.
2. Una vez que se tienen las notas, se crean una a una las voces, almacenándolas en un diccionario de la forma {Nota: OscillatorNode}.

Para la creación de las voces se utiliza la función *createOscillator()* del AudioContext que crea un OscillatorNode.

```

createVocies(){
    for(var i in this.notes){
        var voice = this.#context.createOscillator();
        voice.type = this.#type;
        voice.frequency.value = this.notes[i];

        .....
    }
}

```

Código 6.4: Creación de las voces.

Se puede observar en el fragmento de código que al OscillatorNode (variable voice) se le establece un valor para la propiedad **type**, en el que se elige el tipo de onda del oscilador, y a la propiedad **frequency.value** en el que se establece cual es la frecuencia en la que oscilará el oscilador, es decir, que nota generará.

6.4.1.3.2. Reproducción de audio sin MIDI

Llegados a este punto, ya se disponen de todos los mecanismos necesarios para generar una señal de audio digital. Ahora, el objetivo es que el usuario pueda llegar reproducir las notas generadas de manera digital y utilizar los dos “Osciladores¹¹” implementados.

El proceso por el que un usuario llega a reproducir audio utilizando el sistema viene descrito en la [figura 5.11](#). A continuación, se explica cómo se ha logrado tal proceso.

Como ya tenemos las voces creadas necesitamos introducirlas en el contexto para que se reproduzcan, pero no basta con eso, necesitamos establecer algún mecanismo de control sobre ellas, para que el usuario pueda reproducir las notas a su antojo.

Recordemos que el `OscillatorNode` generaba una frecuencia o nota de manera constante, pero esto no interesa, ya que solo se quiere que se reproduzca una nota cuando lo desee el usuario, presionando una tecla del teclado, del teclado MIDI o clicando en la interfaz.

Para controlar esto se utiliza el nodo de ganancia o **GainNode** de la Web Audio Api, que permite controlar la amplitud de las señales de los nodos que recibe como inputs, es decir, permite controlar el volumen de los nodos. Este tipo de nodo se crea llamando a la función **createGain()** del `AudioContext`. [22]

Así que, para controlar las voces el volumen de las voces, se necesita:

1. Conectar cada `OscillatorNode` a un `GainNode`, esto nos permitirá controlar el volumen de los nodos oscilador por separado.
2. Cada `GainNode` de cada `OscillatorNode` o voz, conectarlo a un `GainNode` del oscilador al que pertenezcan, para controlar el volumen de cada oscilador por separado.
3. Los `GainNode` de los osciladores, conectarlo a un `GainNode` maestro que controlará el volumen de todo el sintetizador.

¹¹ Aparece entre comillas ya que se ha visto que no son osciladores como tal.

Por lo tanto, se tiene el siguiente esquema:

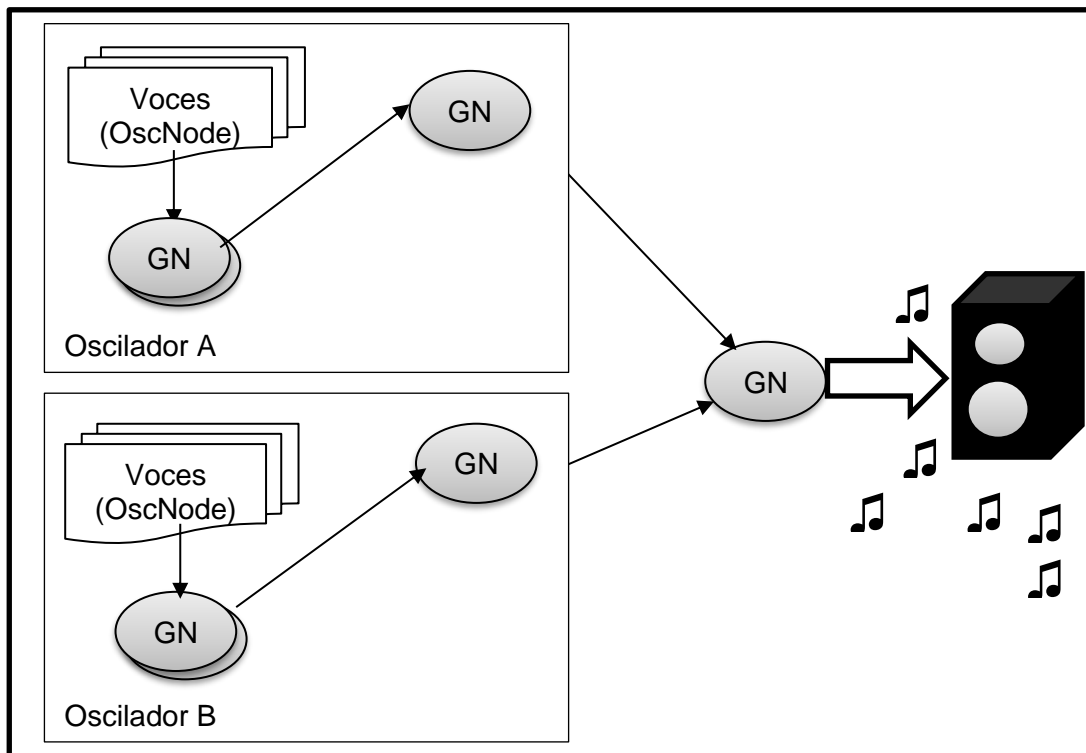


Figura 6.6. Esquema de reproducción de audio

De acuerdo con esta figura tendremos en cada oscilador 85 GainNodes (uno por cada voz) conectados a un GainNode del oscilador que este a su vez se conecta al nodo de ganancia maestro del sintetizador. Además de esto, se necesitará utilizar un GainNode auxiliar, a nivel de sintetizador, que controle la amplitud del audio resultado del proceso de síntesis sin efectos aplicados, ya que los efectos se aplicarán en paralelo (esto se verá más adelante), de manera que por ese nodo pase la señal "limpia". Por lo tanto, en la clase synth necesitaremos dos GainNodes, el maestro y el limpio.

```
class Synth{
    ....
    constructor(){
        ....
        this.#masterVolumeNode = this.#audioCtx.createGain(); //maestro
        this.#gainCleanNode = this.#audioCtx.createGain(); //limpio
        ....
    }
}
```

Código 6.5: Creación del nodo ganancia maestro y el nodo de ganancia limpio

El siguiente paso será crear el nodo de ganancia que contendrán los osciladores y conectarlo al nodo limpio. Para conectar un nodo a otro, se utiliza la función **connect** que poseen todos los AudioNode. Esta función recibe como parámetro el nodo al que se va a conectar el nodo que invoca a la función, de manera que la salida del nodo invocador la recibirá como entrada el nodo pasado como parámetro.

```

class oscillator{
  ....
  constructor(master,context,cleanNode){
    .....
    this.#gainNode = context.createGain();

    ....
    this.#gainNode.connect(cleanNode) //conexión de la ganancia al nodo
                                      limpio

    .....
  }
}

```

Código 6.6 Creación del nodo ganancia del oscilador

A continuación, habrá que conectar las ganancias de las voces al GainNode de la clase oscillator. Además, es necesario iniciar la oscilación en cada uno de los OscillatorsNode para que se empiece a generar sonido, llamando a la función **start**.

Todo esto se refleja en los siguientes fragmentos de código, donde se puede apreciar que cada elemento también es conectado con el contexto de audio:

```

class Voice{

  createVocies(){

    for(var i in this.notes){
      .....
      var gainVoice = this.#context.createGain();
      gainVoice.gain.value = 0;
      this.#gains[this.notes[i]] = gainVoice;

      voice.connect(gainVoice);

      gainVoice.connect(this.#masterNode);
      voice.start(0);

      this.#voices[this.notes[i]] = voice;

    }
  }
  ....
}

```

Código 6.7: Creación y asignación de las ganancias a cada voz e inicio de oscilación

Como último paso, necesitaremos hacer llegar de alguna manera la información que pasa por los GainNode de los osciladores, ya que contienen el sonido generado, al nodo master del sintetizador, ya que toda la información de los osciladores pasa por él.

6.4.1.3.2.1. Paneo

Uno de los requerimientos del sistema es la posibilidad de poder panear los osciladores por separado. Panear un sonido significa, elegir cuanta cantidad del sonido deseas que se reproduzca por un lado u otro de los altavoces, por ejemplo, si paneamos un sonido a la izquierda se enviaría el sonido exclusivamente por el canal izquierdo, por lo que si tuviéramos dos altavoces se reproduciría por el izquierdo.

El paneo se podrá dar en cada oscilador, y es por esta razón, antes de enviar la información al nodo master, es necesario que pase por un nodo de paneo o **StereoPannerNode** que nos va a permitir realizar tal tarea. Este nodo se crea invocando a la función **createStereoPanner** del Audiocontext.

El nodo de paneo coge un valor entre -1 (izquierda) y 1 (derecha) para realizar el paneo del sonido, inicialmente se establece a 0 para que el paneo este al centro, de manera que salga la misma información por ambos lados de los altavoces. [23]

Finalmente, habrá que conectar el nodo de ganancia al nodo de paneo y el nodo de paneo al nodo master del sintetizador para que se reproduzca en los altavoces. Todo esto, tiene lugar en la clase oscillator:

```
class oscillator{
  ....
  constructor(master,context,cleanNode){
    .....
    this.#panNode = this.#audioCtx.createStereoPanner(); //creación paneo
    this.#panNode.pan.value = 0;
    this.#gainNode.connect(this.#panNode); // conexión ganancia paneo
    this.#panNode.connect(master); //conexión ganancia a master
    .....
  }
}
```

Código 6.8: Paneo y conexión con ganancia master

Hasta ahora, se podrá controlar el volumen de cada una de las señales generadas, pero se necesita controlar aún más estas para cumplir los objetivos del sistema. Otro mecanismo de control es el uso de la envolvente ([ver Sección 1.2](#)).

6.4.1.3.2.2. Envolventes

El valor de ganancia de los GainNode es un valor que cambia con el tiempo. Si este valor se cambia de manera instantánea puede provocar molestos 'clicks' que ensucien la señal. Sobre todo, se aprecia cuando el usuario cambia de una nota a otra, por lo que se necesita algún mecanismo para eliminar estos 'clicks'.

Este mecanismo es el uso de envolventes. Como ya se ha mencionado la envolvente tiene cuatro fases o parámetros, que relacionan el tiempo con la amplitud de una señal, reflejado en la siguiente figura:

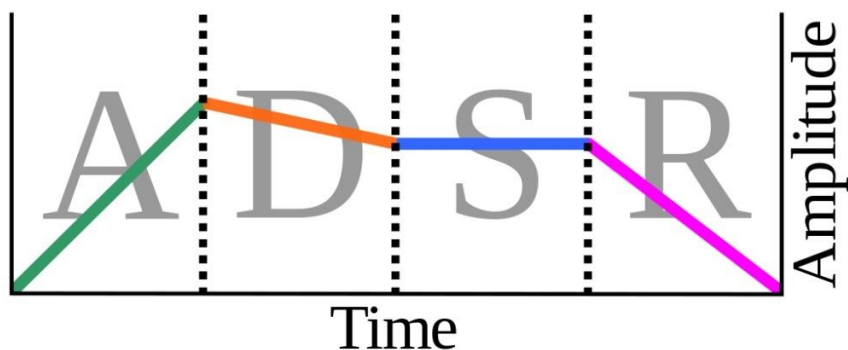


Figura 6.7. Envolvente de una señal

Para evitar estos 'clicks' es necesario que las envolventes sean aplicadas a cada uno de los OscillatorNode o voces, ya que generan las señales de audio. Entonces, cuando un usuario presione una nota ocurrirá lo siguiente:

1. La subcapa de la vista se comunica con la controladora llamando a un método de la clase fachada del controlador (Clase synth), el método **playNote**. Este recibe como parámetro una frecuencia correspondiente a una nota (la nota que se va a reproducir).
2. Este método, llama al método **play** de ambos osciladores indicándole la nota presionada
3. En el método **play**, se comprueba, a partir de un flag de control, si el oscilador está encendido, ya que el usuario puede apagar o encender los osciladores en la interfaz, y se llama al método **start** de la instancia de la clase **Voice** del oscilador, indicándole la nota y el volumen del GainNode del oscilador.
4. En el método **play** de la clase **Voice**, se identifica a que voz corresponde la nota y se llama al método que aplica la envolvente (**envelopeGeneratorOn**).

En el método que aplica la envolvente, se utiliza la función **linearRampToValueAtTime**, invocada por un nodo de ganancia. Esta función recibe como argumento un valor y un tiempo, y permite cambiar el valor de la ganancia del GainNode al valor pasado como argumento de manera gradual en el tiempo pasado como argumento, evitando de esta manera los 'clicks' por el cambio inmediato.

Para aplicar adecuadamente la envolvente, es necesario obtener el tiempo actual en el que se encuentra la señal generada a la que se le aplica, e ir cambiando el valor de la ganancia llamando a dicha función en función de la fase en la que se encuentre. De acuerdo a esto, el proceso de calcular la envolvente será:

1. Obtener el volumen del oscilador.
2. Obtener el tiempo actual en el que se encuentra la señal generada dentro del contexto.
3. Ir modificando la ganancia de acuerdo con la fase de la envolvente en la que se encuentre, determinada por un tiempo de duración.

Es necesario que se obtenga el volumen que el usuario tiene seteado para el oscilador, ya que el mecanismo de reproducción se basa en subir y bajar el volumen de la ganancia de cada voz cuando le corresponda, de manera, que, si el usuario pulsa la nota DO de la primera octava, el volumen de ganancia de la voz correspondiente se establece al del oscilador, mediante el cálculo de la envolvente. Cuando acaba la reproducción de esa voz, el volumen de esta se establece a 0.

Este mecanismo se ha implementado de tal forma, ya que los nodos osciladores producen señales continuas, por lo que, si no fuera controlándolos con los nodos de ganancia, habría que ir creando y destruyendo nodos osciladores o voces cada vez que

el usuario quisiera reproducir notas, ya que la función start solo puede ser invocada una vez desde la creación del nodo, cosa que complicaría la implementación.

Los valores de la envolvente, los cuales son duraciones de intervalos de tiempo de las distintas fases, serán representados por un flotante que indica los segundos que dura la fase, y se almacenarán en un objeto JSON que se irá actualizando a medida que el usuario modifique estos en la interfaz.

Por ejemplo, la fase de Decay se hará de la siguiente manera:

```
//Fase de decay  
current = current / 2 //Current almacena el valor de ganancia en ese momento. Antes  
de la división tiene el valor al que llega después de la fase de ataque  
gainN.linearRampToValueAtTime(current, now + this.#envelope.attack + this.#envelope.decay);
```

Código 6.9: Implementación de la fase de Decay de la envolvente.

Y la envolvente en su totalidad se calculará de la siguiente manera:

```

Class Voice{

    envelopeGeneratorOn(key,gain){
        var gainN = this.#gains[key].gain //obtenemos el nodo de ganancia de la nota
        var current = gainN.value = gain //Representa el volumen en cada una de las
fases
        var now = this.#context.currentTime

        //Se cancelan todos los valores de ganancia del nodo y se establecen a 0
        gainN.cancelScheduledValues(now)
        gainN.setValueAtTime(0 ,now)

        //Fase de ataque
        gainN.linearRampToValueAtTime(current, now + this.#envelope.attack );

        //Fase de decay
        current = current / 2

        gainN.linearRampToValueAtTime(current, now + this.#envelope.attack +
this.#envelope.decay);

        //Fase de sustain
        current = current-0.1

        gainN.linearRampToValueAtTime(current, now + this.#envelope.attack +
this.#envelope.decay +this.#envelope.sustain)
        //Fase de release
        gainN.linearRampToValueAtTime(0, now +this.#envelope.attack +
this.#envelope.decay +this.#envelope.sustain + this.#envelope.release);
    }
}

```

Código 6.10: Implementación de la envolvente.

6.4.1.3.2.3. Ecualización

Recapitulando, hasta ahora tenemos todos los nodos de ganancia de las voces conectados al nodo de ganancia de la clase oscillator, este a su vez conectado al nodo de panning que se conecta al nodo master. Esta cadena todavía no está completa, ya que el flujo de sonido que transcurre por ella se queda en el nodo master y no llega a los altavoces, así que necesitamos hacer llegar la información del sonido a los altavoces de alguna manera.

Otro de los requerimientos del sistema es que se pueda ecualizar el sonido mediante un ecualizador de varias bandas, consiguiendo un tipo de síntesis substractiva. Para que el usuario perciba la ecualización en el sonido final que se reproduce, es necesario que esta se realice antes de que el sonido se envíe a los altavoces y después del nodo master. De esta manera el flujo de sonido será alterado de la siguiente manera:

1. Cuando se crean las voces, es decir, los OscillatorNode, el flujo de sonido que generarán pasará por los nodos de ganancia de cada una.

2. Estos nodos de ganancia dependen del nodo de ganancia del oscilador y cuando el usuario modifique la ganancia del oscilador se modificarán las ganancias de las voces con el mismo valor.
3. Del nodo de ganancia del oscilador el flujo pasará al nodo de paneo y de este al master
4. Al nodo master llegarán los flujos de sonido de los dos osciladores, y de los efectos aplicados (esto se verá más adelante) donde el usuario podrá controlar su intensidad.
5. Del nodo master el flujo pasará al ecualizador, donde el usuario podrá atenuar o intensificar un determinado rango de frecuencias. El flujo circulara por una serie de nodos correspondientes a las bandas del ecualizador y finalmente al destino que representa a los altavoces.

El flujo de sonido contiene información sobre el espectro frecuencial del sonido, es decir, las intensidades de las distintas frecuencias que componen el espectro, de 0hz a 24000hz. La misión del ecualizador será modificar la intensidad de dichas frecuencias, ya sea aumentándola o disminuyéndola. Esto se conseguirá empleando bandas las cuales se corresponderán con BiquadFilterNodes o nodos de filtro que permiten realizar tal tarea. Cada nodo de filtro podrá ser de un tipo distinto, que afectarán de una manera a otra a la intensidad. en el caso del sistema desarrollado se utilizarán los siguientes: [24]



Figura 6.8. High Shelf

Fuente: FabFilter Pro Q3

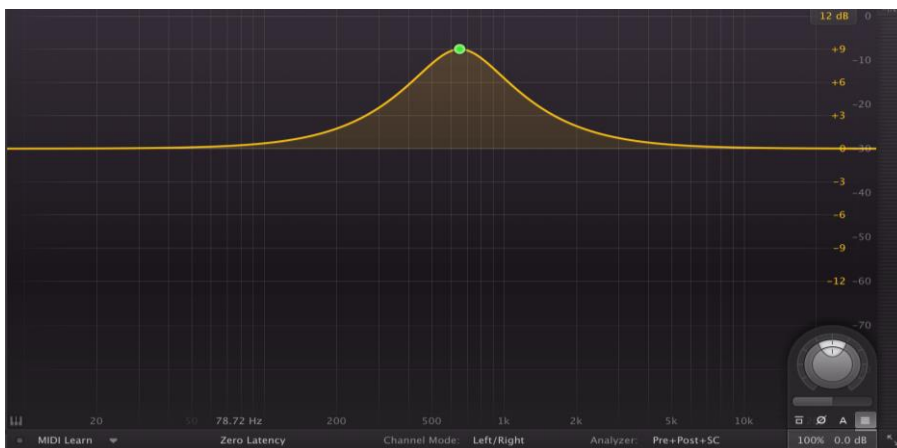


Figura 6.9. Low Shelf

Fuente: FabFilter Pro Q3



Figura 6.10. Low Shelf

Fuente: FabFilter Pro Q3

En total, se tendrán 5 bandas que afectarán a una zona del espectro:

- **Banda de frecuencias bajas:**
 - Afecta a las frecuencias comprendidas entre 0hz-2000hz.
 - Será de tipo Low Shelf.
- **Banda de frecuencias medias-bajas:**
 - Afecta a las frecuencias comprendidas entre 2001hz-8000hz.
 - Será de tipo Peaking.
- **Banda de frecuencias medias:**
 - Afecta a las frecuencias comprendidas entre 8001hz-12000hz.
 - Será de tipo Peaking.
- **Banda de frecuencias medias-altas:**
 - Afecta a las frecuencias comprendidas entre 12001hz-19200hz.
 - Será de tipo Peaking.
- **Banda de frecuencias medias-altas:**
 - Afecta a las frecuencias comprendidas entre 19201hz-24000hs.
 - Será de tipo High shelf.

Nuestro flujo de sonido procedente del nodo master tendrá que pasar por cada una de estas bandas y finalmente será enviado al destino representado por el nodo **destination** del Audiocontext. Para ello, ello en la clase **Equalizer**, la cual representa el ecualizador en la capa controladora, crearemos las bandas llamando a la función *createBiquadFilter()* del Audiocontext, que nos devolverá un nodo de filtro y conectaremos el master a la banda de altos, la de altos a la de medios altos y así sucesivamente hasta llegar al destino:

```

Class Equalizer{
  constructor(master,context,eqGain){
    this.#context = context;

    this.#lw = context.createBiquadFilter();
    this.#lwm = context.createBiquadFilter();
    this.#mid = context.createBiquadFilter();
    this.#hgm = context.createBiquadFilter();
    this.#hg = context.createBiquadFilter();

    master.connect(this.#hg);
    this.#hg.connect(this.#hgm);
    this.#hgm.connect(this.#mid);
    this.#mid.connect(this.#lwm);
    this.#lwm.connect(this.#lw);

    this.#lw.connect(eqGain)
    eqGain.connect(context.destination)
  }
}

```

Código 6.10: Creación y conexión de las bandas.

Nótese, que el filtro de bajos se conecta al nodo **eqGain**, se corresponde con el nodo de ganancia que tendrá el oscilador, ya que para que se consiga un volumen que finalmente sea percibido por el usuario toda esta información tiene que pasar por un nodo de ganancia, debido a que el destino solo acepta nodos de ganancia conectados.

Finalmente, se creará una instancia en la clase Synth para poder trabajar con el ecualizador.

```

Class Synth{
  Constructor(){
    .....
    this.#eq = new Equalizer (this.#masterVolumeNode ,this.#audioCtx,
                              this.#eqGain)
    ....
  }
}

```

Código 6.11: Instanciación del ecualizador

6.4.1.3.3. Reproducción de audio con MIDI

Tenemos los mecanismos necesarios para reproducir notas sin la necesidad de utilizar un dispositivo de MIDI, pero necesitamos funcionalidad para que el usuario pueda reproducir notas con estos dispositivos. En particular, se busca conseguir que el sistema se comuniqué con un teclado MIDI.

Un teclado MIDI es un dispositivo hardware, que comúnmente se conectará al ordenador del usuario vía USB. Los dispositivos MIDI transmiten información musical basada en el tiempo, como puede ser frecuencia, volumen, velocidad, vibrato etc., al dispositivo al

que están conectados. De acuerdo a esto, cuando el usuario conecte el teclado MIDI al ordenador y presione teclas se transmitirán mensajes cada vez que una tecla sea tocada con esta información, así que deberemos interpretar estos mensajes en nuestro sistema y en concreto en la subcapa controladora de la capa Front-end. Aquí es donde entra en juego la **Web MIDI API**, nativa de JavaScript, que proporciona el conjunto de funcionalidad necesaria para poder interpretar la información que llega de los dispositivos MIDI.

Lo primero que se deberá hacer es preguntar a esta API si el dispositivo que está usando el usuario para acceder a la interfaz en el navegador permite comunicación con un dispositivo MIDI. Esto se consigue llamando a la función del navegador **requestMIDIAccess** que devuelve una promesa que cuando se cumple devuelve un objeto que representa el acceso a los dispositivos MIDI, en donde estará contenidos los dispositivos MIDI que se vayan conectando al dispositivo del usuario.

Lo segundo será añadir un **escuchador de eventos** sobre este objeto de acceso, que detecte el evento **statechange**. Este evento se emitirá cada vez que el objeto de acceso MIDI, que se corresponde con el objeto **midi** de la clase **Midi**, cambie de estado, es decir, se añadan o se eliminen dispositivos MIDI de su lista de dispositivos. Cuando se produzca este evento, se llamará a otra función que se encargará de buscar el dispositivo MIDI, si lo hubiera, y elegirlo para recibir mensajes de este.

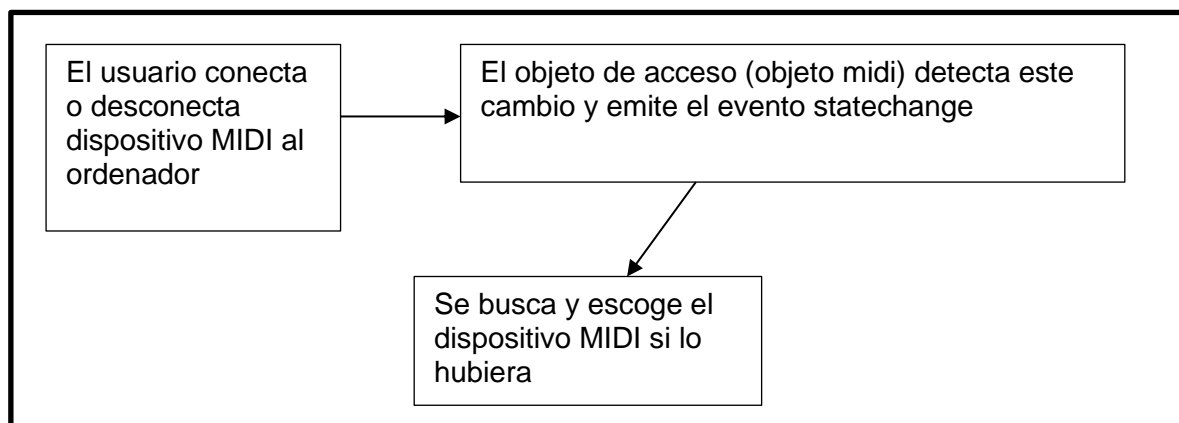


Figura 6.11. Detección de dispositivos MIDI

Todo este proceso tiene lugar en el siguiente método:


```

Class Midi{

requestMidiAccess(){
    var that = this
    navigator.requestMIDIAccess()
    .then((midi)=>{
        that.#support = true;
        that.#midi = midi;
        console.log("Disponible el acceso MIDI")
        midi.addEventListener('statechange',(event)=>this.initDevices(event.target)); //
detecta si se conecta o desconecta el Midi
        this.initDevices()requestAccess(){
            var that = this
            navigator.requestMIDIAccess()
            .then((midi)=>{
                that.#support = true;
                that.#midi = midi;
                console.log("Disponible el acceso MIDI")
                midi.addEventListener('statechange',(event)=>this.initDevices(event.target));
//detecta si se conecta o desconecta el Midi
                this.initDevices()
            },
            (err)=>{
                this.#support=false
                alert("Tu navegador no soporta control por MIDI")
            })
        }
    },
    (err)=>{
        this.#support=false
        alert("Tu navegador no soporta control por MIDI")
    })
}
}

```

Código 6.12: Obtener acceso a MIDI

Una vez que ya tenemos el acceso al MIDI, hay que buscar los dispositivos y seleccionar uno en el caso de que los hubiera para poder empezar a recibir mensajes. El objeto obtenido en la anterior función del código 6.12, contiene una lista de inputs y outputs. La lista de inputs se corresponde con los dispositivos MIDI capaces de enviar información al dispositivo del usuario, y en este caso es lo que nos interesa para nuestro sistema. Así que, para recibir los mensajes que se generan cuando el usuario presiona teclas de su teclado MIDI, necesitamos buscar, seleccionar y almacenar el correspondiente input de esta lista para recibir esos mensajes. Además, necesitaremos mantener el sistema a la escucha ante cualquier mensaje que pueda llegar del dispositivo MIDI. Esto se consigue en la función **initDevices**, la cual realizará lo siguiente:

1. Se comprueba si existen dispositivos conectados
2. Si hay dispositivos conectados, iterará mediante un iterador, ya que son objetos, por los inputs y los outputs y almacenará sus respectivos valores, que no son ni más ni menos que el dispositivo MIDI en sí, en un array auxiliar para poder trabajar con ellos.
3. Se seleccionará el primer dispositivo MIDI disponible, ya que solo nos interesa que el sistema se comunique con uno.
4. Se añadirá un escuchador de eventos sobre el input seleccionado, que escuchará el evento **midimessage** emitido por éste, que se corresponde con un mensaje que viene del dispositivo MIDI. Cuando se de este evento se delegará en otra función que se encargará de tratarlo.

```

Class Midi{

  InitDevices(){
    if(this.#midi.inputs.size > 0){
      var inputArray = []; //Contiene los inputs
      var outputArray = []; //Contiene los outputs

      const inputs = this.#midi.inputs.values()

      //Recorremos los inputs con el iterador
      for (let input = inputs.next(); input && !input.done; input = inputs.next()) {
        inputArray.push(input.value)
      }

      const outputs = this.#midi.outputs.values();

      for (let output = outputs.next(); output && !output.done; output = outputs.next())
      {
        outputArray.push(output.value);
      }

      this.#input = inputArray[0]
      this.#output = outputArray[0]

      //console.log(this.#input)
      this.__showDevice(this.#input.name);
      this.#input.addEventListener('midimessage',
      (event)=>{this.handleMidiMessage(event)}) //Listener para la presion de teclas
      }else{
        this.__showDevice("No hay un MIDI conectado")
      }
    }
  }
}

```

Código 6.13: Detección y selección de dispositivos MIDI

Una vez detectado y seleccionado el dispositivo MIDI, lo único que hace falta interpretar los mensajes que nos vienen de este. Esto se realizará en la función **handleMidiMessage** que se ejecuta cada vez que se pulsa una tecla del teclado MIDI (evento midimessage). El mensaje que llega al sistema es un array con la siguiente estructura: [25]

[CÓDIGO, NOTA, VELOCIDAD]

- **Código:** Entero que indica una acción realizada con el teclado, en este caso, solo interesan los mensajes con código 144, que significa que se ha pulsado una tecla del teclado, y el 128 que significa que se ha levantado una nota.
- **Nota:** Entero que indica un dato del teclado MIDI, que en este caso es la nota presionada.
- **Velocidad:** Entero que indica la intensidad o velocidad con la que se ha pulsado una tecla. Si se ha pulsado muy fuerte la velocidad será mayor, en caso contrario menor. Este valor será despreciado ya que no es necesario para la reproducción de sonidos en nuestro sistema.

A partir de estos mensajes, la función **handleMidiMessage** realizará lo siguiente a partir del mensaje que recibe:

1. Comprueba que el código del mensaje es válido.
2. Comprueba si es un 144 o 148, ya que son acciones distintas.
3. Si es un 144, significa que se tiene que reproducir una nota, por lo que llama a una función auxiliar que se encarga de convertir el dato de nota MIDI en una frecuencia sonora representada por un número flotante.
4. A partir de esa frecuencia, llama a otra función auxiliar, que comprobará que esa frecuencia se encuentra en un rango de frecuencias reproducible por nuestro sintetizador. Como ya se vio en secciones anteriores, en total se pueden reproducir 85 frecuencias que se corresponden con las voces. Después de la comprobación, a partir del diccionario de notas y frecuencias obtenido de la capa Back-end, obtiene la clave de nota (C1, D2, ...) correspondiente a la frecuencia para cerciorarse de que la nota pueda ser reproducida por los osciladores.
5. Si se ha encontrado una clave que se corresponda con una frecuencia, se manda la frecuencia a los osciladores para que la reproduzcan ya que nos hemos asegurado de que la pueden reproducir.

El método que realiza la conversión del dato de nota MIDI a una frecuencia utiliza la formula **MTS**¹², en la que a partir de una frecuencia se obtiene un dato de nota MIDI. Este formula es la siguiente:

$$d = 69 + 12 \log_2 \left(\frac{f}{440 \text{ Hz}} \right).$$

Donde:

- d es el dato de la nota MIDI

¹² MIDI tuning standard (MTS) : https://en.wikipedia.org/wiki/MIDI_tuning_standard

$$\log_2 \left(\frac{f}{440 \text{ Hz}} \right)$$

- es el número de octavas por encima de la nota LA a 440 Hz, el cual al multiplicarlo por doce del número de semitonos por encima de 440 Hz, ya que una octava tiene 12 semitonos
- Al sumar lo anterior con 69 da el número de notas que hay por encima del DO cinco octavas por debajo del DO central y el DO central¹³

Despejando esta fórmula, sacamos la frecuencia a partir de un dato MIDI:

$$f = 2^{(d-69)/12} \cdot 440 \text{ Hz}$$

En el código, esta fórmula irá acompañada de un parseo del resultado para encontrar la frecuencia en nuestro diccionario de notas:

```
Class Midi{
  __midiNoteToFrequency (){
    var value = Math.pow(2, ((note - 69) / 12)) * 440;
    if(value < 98){
      var parse = parseFloat(Math.ceil(value * 10000) / 10000)
      if(parse === 77.7818){return 77.7817}
      if(parse === 92.4987){return 92.4986}
      if(parse === 46.2494){return 46.2493}
      if(parse === 51.9131){return 51.9130}
      return parse;
    }else if (value < 988){
      return parseFloat(Math.round(value * 1000) / 1000)
    }else{
      return parseFloat(Math.round(value * 100) / 100)
    }
  }
}
```

Código 6.14: Conversión de dato MIDI a frecuencia

6.4.1.3.4. Aplicación de efectos

A parte de generar y reproducir sonidos, el usuario tendrá la posibilidad de aplicar efecto a estos sonidos. Los efectos se encargarán de modificar la señal original, de una manera u otra en función del tipo que se trate, en concreto el usuario podrá aplicar los siguientes efectos:

¹³ Do central es la frecuencia 261,6 Hz

- Reverb
- Delay
- Distorsion
- Filtros

Estos efectos serán aplicados en paralelo. Aplicar un efecto en paralelo significa que a pesar de que estos modifican la señal original, se seguirá reproduciendo la señal original, así que si hubiera efectos aplicados el resultado de la señal que percibe el usuario es la suma de la señal original más las señales modificadas generadas por los efectos, pero la señal original nunca se pierde.

Si en lugar de en paralelo, se aplicasen en cadena, el usuario no percibiría nunca la señal original, sino que percibiría la modificada por los efectos solamente.

Al ser en paralelo, el usuario podrá controlar la cantidad de efecto que quiera percibir, es decir, el volumen de este.

Aunque puedan existir efectos de distintos tipos todos tendrán una parte común ya que compartirán funcionalidad. Estos aspectos comunes son los siguientes:

- Se les conecta un **input**, la señal original.
- Los efectos se conectan a un nodo de ganancia al que denominaremos **wet**.
- Los nodos **wet** se conectan a un **output**, que será un nodo de ganancia de salida.
- La aplicación y desaplicación tiene partes comunes.

Los nodos wet son GainNodes o nodos de ganancia que controlarán el volumen del efecto a través de sus respectivos setters que modificara la propiedad gain.value de estos, correspondiente al volumen.

Todos los efectos recibirán como parámetros en su constructor el AudioContext para crear los nodos que hagan falta para estos, el nodo de ganancia correspondiente a la señal limpia que será el input y el nodo de ganancia master al sintetizador que será el output. Los nodos wet serán creados en el momento de la creación de los objetos llamando a la función ya vista createGain.

Para representar todos estos aspectos comunes, se utiliza la clase **Effect** de la que heredarán todos los efectos. En ella existirán métodos para aplicar y desaplicar un efecto, los cuales siguen los siguientes esquemas:

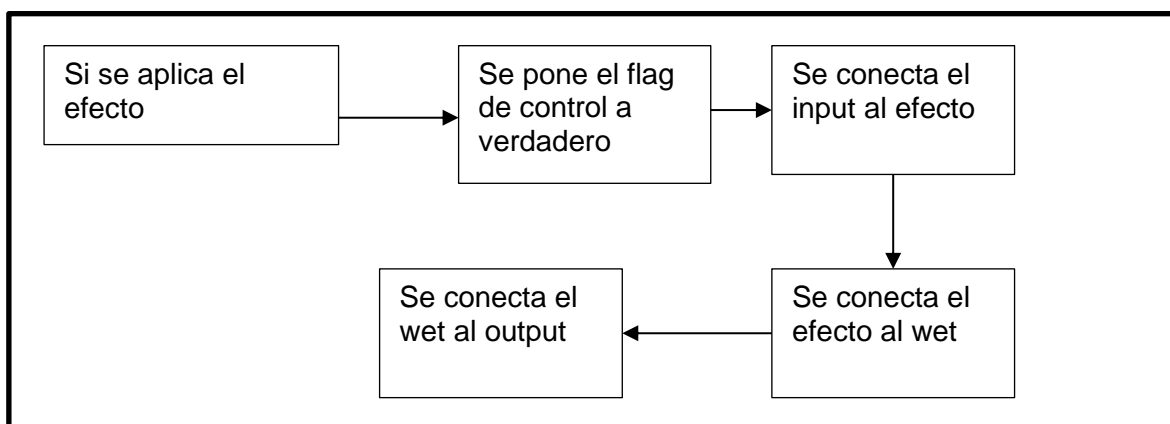


Figura 6.12. Aplicar efecto

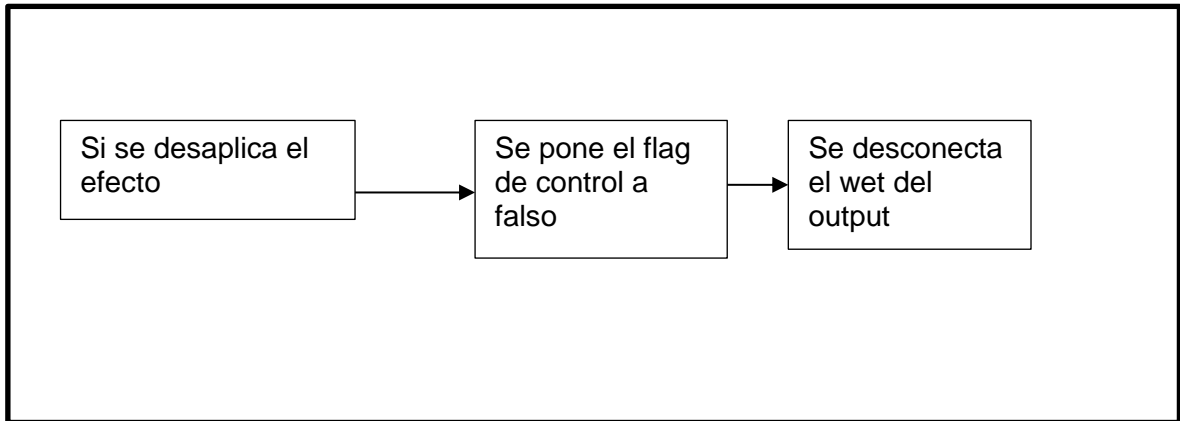


Figura 6.13. Desaplicar efecto

Estos esquemas se reflejan en el código en los métodos **apply** y **disapply** de la clase **Effect**:

```

Class Effect{
    apply(){
        this.#on = true;
        this.input.connect(this.effect)
        this.wet.connect(this.output);
    }

    disapply(){
        this.#on = false;
        this.wet.disconnect(this.output)
    }
}
  
```

Código 6.15: Aplicar y desaplicar efecto

A pesar de que los efectos cuentan con aspectos comunes también poseen diferencias significativas que necesitan de funcionalidades distintas. Cada efecto contará con su respectiva instancia en la clase **Synth** para que se pueda interactuar con ellos.

```

Class Synth{
    constructor(){
        //Efectos
        this.#reverb = new
        Reverb(this.#audioCtx,this.#gainCleanNode,this.#masterVolumeNode);
        this.#delay = new
        Delay(this.#audioCtx,this.#gainCleanNode,this.#masterVolumeNode);
        this.#filter = new
        Filter(this.#audioCtx,this.#gainCleanNode,this.#masterVolumeNode,"highpass"
        );
        this.#distorsion = new
        Distorsion(this.#audioCtx,this.#gainCleanNode,this.#masterVolumeNode);
    }
}
  
```

Código 6.16: Instanciación de Efectos

A continuación, se detalla la implementación de cada uno de los efectos por separado.

6.4.1.3.4.1. *Reverb*

El efecto Reverb modifica las señales de audio de manera que el usuario percibirá una sensación de espacio en los sonidos, consiguiendo simular una reverberación de una sala, el cual estará representado por la clase **Reverb**.

El efecto Reverb viene a simular el comportamiento de un sonido en una sala. Cuando se reproduce un sonido en una sala este colisiona contra las paredes y el mobiliario. En cada colisión las propiedades del sonido se cambian y el sonido que percibimos también. Esta colisión se denomina reverberación, que no es ni más ni menos una muestra modificada por las colisiones del sonido original que nuestro cerebro percibe unos instantes después que el sonido original. Es por ese retardo entre la reverberación y la señal original por la que percibimos esa sensación de espacio.

Sin embargo, cada sala o cada entorno acústico tiene propiedades distintas por lo que simular cada entorno por separado es una tarea excesivamente compleja para un programador.

Para solucionar esto, la Web Audio API nos proporciona un nodo, el **ConvolverNode** del AudioContext, que permite simular esos entornos acústicos. Este nodo proporciona una convolución lineal en un AudioBuffer, es decir, proporciona a partir de funciones matemáticas una señal de audio que se asimilaría a la reverberación producida en un entorno acústico. [24]

Para generar esa reverberación, este nodo necesita un impulso. Un Impulso representa el efecto que tendría un sonido en un entorno acústico, en otras palabras, si grabáramos un sonido en una sala es la diferencia entre el sonido original y el grabado. Estos impulsos son distintos en cada entorno acústico por lo que en el caso de este sistema se optará por un impulso dado en un entorno acústico genérico generado de manera matemática. La función que genera este impulso es la siguiente:

```

Class Reverb extends Effect{
  impulseResponse(duration, decay, reverse){
    var sampleRate = this.audioctx.sampleRate
    var length = sampleRate * duration
    var impulse = this.audioctx.createBuffer(2,length,sampleRate)
    var impulseL = impulse.getChannelData(0);
    var impulseR = impulsimpulseResponse(duration, decay, reverse){

    if (!decay)
      decay = 2.0;
    for (var i = 0; i < length; i++){
      var n = reverse ? length - i : i;
      impulseL[i] = (Math.random() * 2 - 1) * Math.pow(1 - n / length, decay);
      impulseR[i] = (Math.random() * 2 - 1) * Math.pow(1 - n / length, decay);
    }

    return impulse;

  }
}

```

Código 6.17: Generación de impulso

Esta función hace lo siguiente:

1. Obtiene el número de muestras de frames que tendrá el AudioBuffer, el sampleRate del Audiocontext.
2. Calcula la longitud del impulso que se va a generar a partir de la duración pasada como parámetro y el sampleRate.
3. Crea un AudioBuffer (variable impulse) con dos canales o filas, ya que será un impulso en estéreo.
4. Separa ambas filas en dos variables.
5. Rellena todas las filas de manera proporcional al tiempo de caída (decay) y la longitud. Nótese, que se multiplica por un número aleatorio, esto se hace para generar una sensación de espacio al haber información distinta pero parecida en ambos canales del espacio estéreo.

El ConvolverNode que se usa utiliza este impulso como parámetro a la hora de modificar la señal que recibe como input (señal original), concretamente lo recibe en la propiedad buffer del nodo.

La señal producida como salida de este nodo podrá ser filtrada por medio de dos tipos de filtros, un filtro pasa altos, que filtra frecuencias bajas y otro filtro pasa bajos que filtra frecuencias altas. Es por esta razón que a la hora de aplicar el efecto se necesita un tratamiento distinto al resto de efectos ya que los filtros se aplicarán en cadena. La función que aplica el efecto es la siguiente:


```

Class Reverb extends Effect{
  apply(){

    this.effect.buffer = this.impulseResponse(5,5,0);
    this.effect.connect(this.#gainAux);
    super.apply();
    this.#lpf.connect(this.#gainAux,this.#gainFilter);
    this.#hpf.connect(this.#gainFilter,this.wet);

  }
}

```

Código 6.18: Aplicar efecto Reverb

Esta función es llamada cada vez que el usuario quiera aplicar el efecto de Reverb y realiza lo siguiente:

1. Calcula un impulso
2. Conecta el efecto a un primer nodo de ganancia auxiliar del efecto necesario para poder aplicarle los filtros
3. Realiza el tratamiento estándar para los efectos invocando al método del padre.
4. Conecta el primer nodo de ganancia auxiliar con el filtro pasa bajos y el filtro pasa bajos lo conecta con otro nodo de ganancia auxiliar distinto, que representa la señal filtrada por este filtro.
5. Por último, conecta el primer nodo de ganancia auxiliar del filtro pasa bajos con el filtro pasa altos y el filtro pasa bajos lo conecta con el nodo wet.

Cabe destacar, que los filtros empleados son instancias de la clase **Filter**, la cual será explicada más adelante.

Por tanto, los parámetros que el usuario puede cambiar del efecto son los siguientes:

- **Decay:** Tiempo en el que el efecto tarda en desaparecer.
- **Filtro pasa bajos:** Podrá filtrar las frecuencias bajas del efecto.
- **Filtro pasa altos:** Podrá filtrar las frecuencias altas del efecto.

6.4.1.3.4.2. Delay

El efecto Delay es lo que popularmente se conoce como eco, y consiste simplemente en repetir un fragmento de un sonido durante un tiempo determinado de manera retardada cada vez que este se produce.

Para tal tarea, la Web Audio Api nos proporciona el **DelayNode** del Audiocontext, que proporciona ante una señal de audio recibida como entrada un retardo en la salida, por lo que ese retardo temporal en la señal reproducido junto con la señal original da una sensación de “*repetición retardada*”. [24]

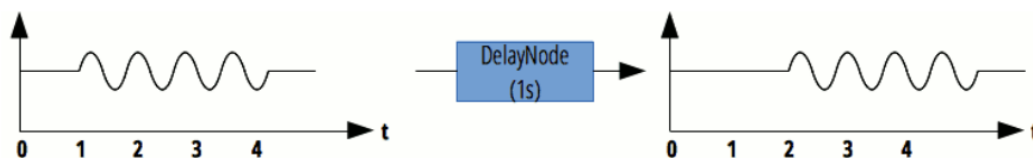


Figura 6.14. Retardo en la señal producida por el DelayNode

Fuente: <https://developer.mozilla.org/en-US/docs/Web/API/DelayNode>

Este DelayNode utiliza la siguiente propiedad que el usuario podrá modificar a través de la interfaz:

- **delayTime:** Es la fracción de tiempo del sonido que se reproducirán del sonido. Es decir si tenemos por ejemplo un delayTime de 1/16 cada sonido retardado se corresponderá con 1/16 segundos de la duración del sonido original.

El delayNode solo nos proporciona una repetición por cada sonido, pero si queremos simular el eco y que el sonido se repita varias veces necesitamos algún mecanismo para ello. Aquí es donde entra en juego el **Feedback**. El feedback nos permitirá tener ese efecto de eco a partir de un nodo de ganancia que estará conectado creando un bucle, esto se puede ver en la función que aplica el efecto:

```
Class Delay extends Effect{
  apply(){

    super.apply()
    this.effect.connect(this.wet);
    this.effect.connect(this.#feedback);
    this.#feedback.connect(this.effect);
  }
}
```

Código 6.19: Aplicar efecto Delay

En el código anterior, podemos observar, que aparte del tratamiento del padre, se crear el bucle del Feedback. Este bucle se consigue conectando el efecto al Feedback y el feedback al efecto, creando una retroalimentación entre ambos y pudiendo controlar la cantidad de feedback existente mediante el nodo de ganancia que representa.

6.4.1.3.4.3. Distorsión

El efecto de Distorsión, produce sobre las señales de audio que recibe como entrada una modificación en estas que provocan que se escuchen saturados asimilándose al efecto de un amplificador de una guitarra eléctrica. El usuario percibirá que los sonidos están sucios y rotos.

La Web Audio API nos proporciona el **WaveShaperNode** del Audiocontext para lograr este efecto. Este nodo aplica una distorsión sobre las señales que recibe como entradas a partir de una curva en forma de onda. Esta curva será necesaria calcularla ya que la requiere en la propiedad del nodo **curve**. De acuerdo con la forma de la curva de distorsión tendremos un resultado u otro produciendo sonidos más cálidos, más suaves o más duros. [26]

En este caso se utilizará una curva de distorsión sigmoidea, es decir, en forma de S ya que es la que mejor se adapta al efecto de distorsión que buscamos

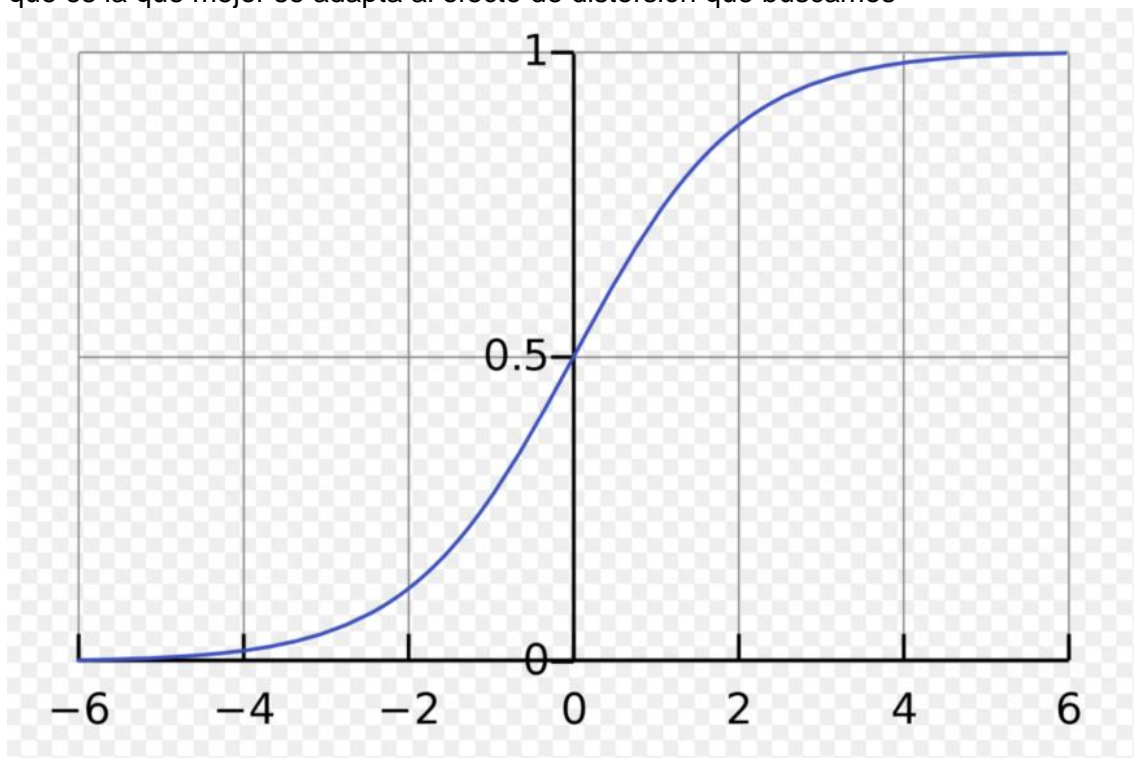


Figura 6.15. Curva de distorsión

Esta curva será calculada de manera matemática en la siguiente función:

```

Class Distorsion extends Effect{
  makeDistortionCurve(amount) {
    var k = typeof amount === 'number' ? amount : 50,
        n_samples = 44100,
        curve = new Float32Array(n_samples),
        deg = Math.PI / 180,
        i = 0,
        x;
    for ( ; i < n_samples; ++i ) {
      x = i * 2 / n_samples - 1;
      curve[i] = ( 3 + k ) * x * 20 * deg / ( Math.PI + k * Math.abs(x) );
    }
    return curve;
  }
};

```

Código 6.20: Obtención de la curva de distorsión

Se puede observar que recibe el parámetro “*amount*”. Este parámetro es un flotante que representa la pendiente de la curva, cuanto más alto más dura será la distorsión provocada.

6.4.1.3.4.4. Filtros

Los filtros son efectos que modifican el espectro frecuencial de una señal de audio. En nuestro código estarán representados por la clase **Filter**. Hay varios tipos de filtros, los cuales modificaran el espectro de una manera u otra. El usuario podrá aplicar los siguientes filtros:

- **Lowpass o pasa bajos:** Establece un límite y las frecuencias por encima de este límite se atenúan, es decir va filtrando desde frecuencias agudas hasta las graves en función de donde se sitúe el límite.
- **Highpass o pasa altos:** Es el inverso al filtro Lowpass.
- **Bandpass:** Se establece un rango de frecuencias y las que quedan fuera de este rango se atenúan.
- **Lowshef:** Se establece un límite y las frecuencias inferiores a este se atenúan o se intensifican. Las frecuencias sobre él no se modifican.
- **Highshelf:** Es el inverso al Lowshef.
- **Peaking:** Se establece un rango de frecuencias y se atenúan o intensifican. Las frecuencias fuera de este rango no se modifican.
- **Notch:** Es el inverso al filtro BandPass.

Estos filtros se consiguen utilizando el ya comentado **BiquadFilterNode** del Audiocontext. Este nodo acepta la propiedad **type** en la que se establece el tipo del filtro, que exclusivamente puede pertenecer a los vistos en esta sección. Así que cada vez que el usuario cambie el filtro se cambia esta propiedad y además, se deberá establecer la frecuencia límite del filtro en el caso de que la tenga. Cada vez que se vaya moviendo este límite se modificará la propiedad **frequency.value** del nodo, que acepta un flotante que representa una frecuencia.

6.4.1.3.5. Gráficos

Otro de los requisitos que el sistema desarrollado debe tener, es la existencia de gráficos que de alguna manera permitan visualizar las señales de audio generadas de manera gráfica. Aunque este requisito pudiera parecer exclusivo de la interfaz y por tanto de la subcapa de la vista, necesitamos analizar las señales de audio en la subcapa controladora para poder enviarle a la subcapa de la interfaz un mensaje que sea fácilmente interpretado a la hora de pintar los gráficos en la interfaz. Es aquí donde entra en juego la clase **Analyser** de la subcapa controladora.

Esta clase se encargará de crear un **AnalyserNode** del Audiocontext, mediante la llamada a la función **createAnalyser** del contexto. Este nodo recibe como input una señal de audio y produce como salida la misma señal de audio sin cambios, solo se encarga de realizar un análisis en el dominio del tiempo y la frecuencia del sonido en tiempo real. Gracias a este nodo podremos extraer datos interpretables por nuestro sistema sobre el sonido generado para procesarlos y posteriormente representarlos en la interfaz. [24]

Los datos obtenidos se basan en la **transformada rápida de Fourier**, conocida por su abreviatura FFT, es un algoritmo que permite obtener una representación del dominio de la frecuencia de un sonido, es decir, la intensidad de cada una de las frecuencias del espectro que posee el sonido, en función del tiempo a partir de una señal discreta, que en nuestro caso son los AudioBuffers. [24]

Este nodo tiene cuenta con dos propiedades que son importantes en el análisis:

- **fftSize:** Es el tamaño del buffer, es decir, del dominio de la frecuencia que se analizará. Debe ser potencia de dos. Los valores más altos dan un análisis más detallado de la señal, aunque acarrea un coste computacional más alto que repercutirá en el rendimiento, por lo que se establecerá un valor intermedio de 512.
- **frequencyBinCount:** Es la mitad del tamaño del fftSize. Se corresponde con la cantidad de valores que se visualizarán. Es decir, tendremos 256 valores a representar.

Para el análisis necesitaremos crear un tipo de dato donde se vaya almacenando el resultado del análisis que haga el nodo. Este tipo de dato será un **Uint8array**, que se corresponde con un array de enteros sin signos de 8 bits. Los datos que se almacenarán son enteros del 0 a 255, y cada dato representará el valor en decibelios de un rango de frecuencias, siendo la última frecuencia representada la mitad de la frecuencia de muestreo, que ya vimos que se eligió 48000Hz, por lo que el último dato es 24000Hz. En este array ira representado el rango de frecuencias de 0 a 24000 Hz y como hay 256 valores cada casilla se corresponde con aproximadamente un conjunto de 94 frecuencias por lo que no tendremos una representación exacta pero sí aproximada.

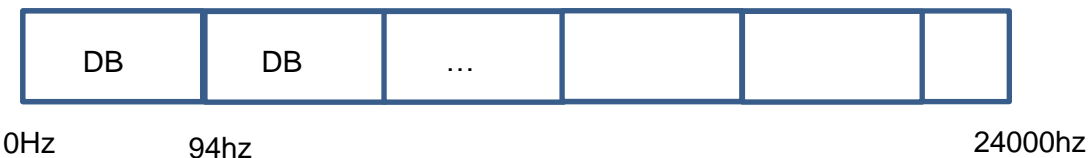


Figura 6.16. Array de frecuencias

Este array será recibido como parámetro por la función **getBytesFrequencyData()** del nodo que se encargará de aplicar la FFT sobre la señal que recibe como input e ir introduciendo los resultados de aplicar el algoritmo en el array. Cada vez que la interfaz requiera datos para pintar se llamará a la siguiente función donde se hace uso de la función del nodo:

```

Class Analyser{
  getData(){
    this.#analyser.getBytesFrequencyData(this.#domain)
    return this.#domain; //Es el Uint8Array
  }
};

```

Código 6.21: Obtención de los resultados del análisis

Por último, cabe destacar, que el análisis se realizará del sonido producido por sintetizador en todo su conjunto, incluyendo los efectos por lo que es necesario que el nodo master esté conectado al analizador. Esto se realiza en el constructor de la clase:

```

Class Analyser{
  constructor(master,context){
    this.#analyser = context.createAnalyser();
    this.#analyser.fftSize = 512;
    this.#sampleRate = context.sampleRate
    this.#size = this.#analyser.frequencyBinCount;
    this.#domain = new Uint8Array(this.#size);

    master.connect(this.#analyser)
  };
}

```

Código 6.22: Constructor del analizador

6.4.1.3.6. Grabación

El usuario tendrá la posibilidad de grabar una secuencia de sonidos generados por nuestro sistema. A nivel de subcapa controladora, la funcionalidad se encapsula en la clase **Recorder**, donde son necesarios los siguientes elementos:

1. Se necesita un **MediaStreamAudioDestinationNode** del Audiocontext que es similar al nodo destination, pero a diferencia de este no envía las señales de audio para reproducirse.
2. Se necesita un objeto **MediaRecorder** de la MediaStream Recording API de JavaScript que proporciona funcionalidad para grabar datos media (audio, video etc..)
3. Se necesita un objeto Blob. Un objeto Blob representa un objeto tipo fichero de datos planos inmutables. Los Blobs representan datos que no necesariamente se encuentran en un formato nativo de JavaScript. Este objeto es necesario para que el elemento HTML que reproducirá la grabación reconozca la información que se ha grabado y pueda reproducirla.

A partir de estos dos elementos el procedimiento es muy simple:

1. Se crea un MediaStreamAudioDestinationNode
2. Se conecta el nodo de ganancia master, ya que contiene toda la información de audio del sintetizador, al MediaStreamAudioDestinationNode.
3. Se crea un objeto MediaRecorder pasándole como parámetro el stream o buffer que contendrá las señales de audio el MediaStreamAudioDestinationNode a medida que las recibe.
4. A medida que el MediaStreamAudioDestinationNode reciba datos el objeto MediaRecorder los recibirá también y se lanzará el evento *"dataavailable"*. Cuando ocurra esto, que se almacenarán en un array
5. Cuando se pare la grabación (se produce el evento *"stop"*) se crea un objeto Blob, pasandole como argumentos los datos almacenados en el array y el objeto *{ 'type': 'audio/mp3; codecs=opus' }* que indica que el array se corresponde con un archivo mp3. Además, se crea un URL para que el elemento HTML que reproduce la grabación pueda reproducirla.

Cuando el usuario inicie la grabación, se llamará al método start del MediaRecorder que empezará a recopilar todo lo que pase por el MediaStreamAudioDestinationNode y cuando se pare se llamará a la función stop del MediaRecorder que parará la grabación.

6.4.1.3.7. Comunicación capa Back-end

Ya hemos visto un conjunto de responsabilidades que tiene la subcapa controladora, pero todas ellas relacionadas con las señales de audio y la Web Audio API. Otras de las responsabilidades de la subcapa es comunicarse con la capa Back-end, en otras palabras, enviará peticiones a la API de esta capa.

Para tal responsabilidad, se ha encapsulado el comportamiento necesario (aunque no todo) para llevarla a cabo en dos clases:

- **Loader:** Que servirá para obtener datos de la capa Back-end, contendrá las peticiones GET.
- **Saver:** Servirá para mandar peticiones que alteren el estado del modelo de datos, es decir, que alteren el estado de la capa de datos, y que se corresponderán con peticiones PUT y DELETE.

Sin embargo, no toda la comunicación con la capa Back-end se realizará en esta capa si no que existirá comunicación también desde la capa de la vista en dos casos excepcionales, el login y el registro, los cuales se explicarán más adelante. Esto se debe a que la instancia de la clase **Synth**, que representa al sintetizador en la subcapa controladora, se crea en el momento en que el usuario hace login en la aplicación a través de un mensaje que envía la subcapa de la vista ya que el sintetizador necesita datos que son únicos a cada usuario logueado.

Otro caso excepcional es el de la obtención de las notas y sus respectivas frecuencias que utilizará el sintetizador ya que serán necesarias requerirlas en las dos subcapas. No obstante, se ha conseguido encapsular el resto de consultas que necesita realizar la capa Front-end.

Las peticiones son realizada mediante la función **fetch()** de JavaScript, la cual sirve para manipular partes del canal HTTP tales como las peticiones y respuestas . Esta función utilizará los siguientes parámetros:

1. URL del endpoint al que va dirigido la petición. En este caso será la URL correspondiente de la API de la capa Back-end.
2. Objeto de opciones. En el sistema desarrollado se utilizarán las siguientes opciones:
 - a. Method: es el método HTTP que se utilice en la petición.
 - b. Headers: Es otro objeto que contiene las cabeceras de la petición HTTP necesarias en la capa Back-end. Se utilizarán las siguientes:
 - i. Content-type: siempre será application/json ya que el cuerpo de las peticiones, si es que lo hay, siempre será un objeto JSON.
 - ii. Authorization: Es un token de autorización, que la capa Back-end envía al hacer Login y que se necesita para solicitar acciones al Back-end. Este token se encuentra en una cookie.
 - iii. User: Es el nombre de usuario que también se encuentra almacenado en una cookie.
 - c. body: Es el cuerpo de la petición que se corresponderá con un objeto JSON.

Esta función devuelve una promesa que cuando se cumple se obtiene la respuesta de la capa back-end y se ejecutará una función que la parseará a un objeto JSON. Después de tener la respuesta en el formato adecuado se formateará la respuesta y se devolverá. Un ejemplo de petición se puede ver en el siguiente código:

```

Class Loader extends DbFetcher{
  async __fetchUrl(url,requestOptions){
    var status;
    var that = this ;

    try{
      await fetch(url,requestOptions)
      .then(function(response){
        status = that.handleStatus(response.status);
        if(status){
          return response.json();
        }else{
          return false;
        }
      })
      .then((data)=>{
        if(data){
          that.data = data.msg;
        }else{
          that.data = data;
        }
      });

    }catch(err){
      that.data = false;
      console.error(err)
    }

    return this.data
  }
}

```

Código 6.23: Función que realiza una petición a la capa Back-end

Notase, que la función en la que está contenida la petición es asíncrona y esto a se debe a que al hacer la petición se obtiene una promesa, esta promesa se mantiene en estado pendiente hasta que recibe la respuesta del servidor y se resuelve, por lo que la función tiene que ser asíncrona para que la promesa permanezca a la espera en segundo plano. Sin embargo, no interesa este comportamiento porque no se desea que se realice otro procesamiento hasta que se reciba una respuesta, de ahí, el uso de la directiva **await** delante de la función `fetch`. Esto indica que hasta que no se resuelva la promesa no se continúe con el procesamiento.

6.4.1.3.7.1. DbFetcher

Todas las peticiones realizadas a la capa Front-end tienen partes comunes que se abstraen en una clase llamada **DbFetcher**, de esta clase heredan tanto la clase Loader, como la clase Saver. Esta clase se encargará de:

- Interpretar los códigos de respuesta HTTP: Todas las respuestas de las peticiones tendrán asociado un código numérico de acuerdo a un estándar¹⁴. Esta clase se encargará de interpretar si es un código de error o no para transformarlo en un booleano que será requerido en la interfaz para mostrar mensajes de éxito o de error
- Modificar y obtener las cookies: Las sesiones de los usuarios serán gestionadas en la capa Front-end mediante el uso de cookies. Estas cookies se necesitarán consultar y en algunos casos modifica. Más adelante se analizará más a fondo lo relativo a las cookies.

6.4.1.3.7.2. Saver

La clase Saver concentrará las peticiones PUT y DELETE que realice la subcapa controladora. Esta clase se encargará de enviar peticiones para:

- Guardar sonidos
- Editar perfil
- Editar contraseña
- Borrar contraseña

Todas estas peticiones siguen el mismo procedimiento:

1. Lanzas la petición al endpoint correspondiente con las opciones pertinentes y los datos que correspondan en el cuerpo.
2. Interpretan el código de la respuesta.
3. Parsean la respuesta.
4. Formatean la respuesta para que la interprete la subcapa de la vista.

Este comportamiento es el mismo en todas las peticiones que se realizan en esta clase, salvo a la hora de mandar la petición para guardar sonidos. Antes de enviar esta petición es necesario de recopilar todo el estado del sintetizador, es decir, el valor de todas las variables que intervienen en la generación y reproducción de sonidos. Para ello se inicia un mensaje en cadena:

1. Se obtienen los datos de los osciladores (tipo de onda, si está activo, paneo, voces, volumen, etc..).
2. Se obtienen los datos correspondientes a las envolventes.
3. Para cada uno de los efectos se obtienen sus datos (si está activo, parámetros, wet, etc..).

Una vez recopilados se construye un JSON que se envía en la petición con la finalidad de que esos datos acaben almacenados en base de datos. Esta recopilación del estado se realizó en el siguiente fragmento de código:

¹⁴ Códigos de respuesta HTTP : <https://developer.mozilla.org/es/docs/Web/HTTP/Status>

```

Class Saver extends DbFetcher{
  __getEnvelopes(){
    this.#envelopes['A'] = this.#oscillatorA.getEnvelope();
    this.#envelopes['B'] = this.#oscillatorB.getEnvelope();
  }

  __getOscillators(){

    this.#oscilators['A'] = this.#oscillatorA.getState();
    this.#oscilators['B'] = this.#oscillatorB.getState();

    this.__getEnvelopes();
    this.#oscilators['A']['envelope'] = this.#envelopes['A'];
    this.#oscilators['B']['envelope'] = this.#envelopes['B'];

  }

  __getEffects(){
    this.#oscilators['effects'] = this.#synth.getEffects();
  }
}

```

Código 6.24: Recopilación de estado

6.4.1.3.7.3. Loader

La clase Saver concentrará las peticiones GET que realice la subcapa controladora. Esta clase se encargará de enviar peticiones para:

- Obtener la capacidad máxima de almacenamiento de sonidos.
- Obtener el perfil del usuario.
- Obtener un sonido o estado completo a partir de su identificador.
- Obtener los metadatos de un sonido (datos que aparecerán en la tabla de sonidos).

El procedimiento que se seguirá para realizar las peticiones es exactamente el mismo al que se sigue en la clase Saver.

6.4.1.4. Subcapa Vista

Vista la implementación de la subcapa controladora, la siguiente que nos encontramos en la capa Front-end es la subcapa de la vista. Esta subcapa es simplemente la interfaz de la aplicación. Esta interfaz cuelga del ReactDOM que se encarga de renderizar todos los componentes contenidos en él. Durante esta sección se detallarán todos los aspectos relacionados con la implementación de esta subcapa.

6.4.1.4.1. Enrutamientos y sesiones

Como se puede ver en el diseño de clases de la subcapa vista, en la [sección 5.4.4](#), la interfaz sigue una estructura jerárquica basada en la virtualización del árbol DOM el ReactDOM, que ReactJS realiza. En este árbol virtualizado se encontrarán los componentes de la interfaz.

El ReactDOM se encargará de renderizar los componentes de la interfaz, para que se visualice en el navegador, a través de la función **render**, que existirá también en todos los componentes, que se corresponden en el código con las clases heredadas de **React.Component**. Esta función de renderizado se encuentra en el fichero index.js y presenta el siguiente código:

```
ReactDOM.render(  
  
  <Router>  
    <Nav />  
    <Switch>  
      <PrivateRoute exact path="/synth" isAuthenticated={isAuthenticated()} component  
={App}/>  
      <PrivateRoute exact path="/profile" isAuthenticated={isAuthenticated()} component  
={Profile}/>  
      <PublicRoute exact path="/signup" component={SignUp}/>  
      <PublicRoute exact path="/" isAuthenticated={isAuthenticated()} component={Login  
}/>  
      <Route component={ErrorPage}/>  
  
    </Switch>  
  
  </Router>,  
  document.getElementById('root')  
);
```

Código 6.25: Renderizado del DOM

A partir del código podemos usar lo siguiente:

1. Los componentes están envueltos en otras etiquetas, PrivateRoute, PublicRoute, Route, Switch y Router
2. Obtiene el elemento HTML con identificador "root". Este elemento es un div de donde colgarán todos los elementos HTML después de la renderización.
3. Se identifican 5 componentes principales:
 - a. App: Será el componente que renderiza el sintetizador.
 - b. Profile: Proporciona una vista para el perfil de usuario.
 - c. SingUp: Proporciona una vista para el registro de nuevos usuarios.
 - d. Login: Proporciona una vista para iniciar sesión en el sistema.
 - e. Errorpage: Proporciona una vista de error para cuando se intenta acceder a una ruta inexistente en la aplicación.
4. Se identifica otro componente adicional, el componente Nav, que se corresponde con la barra de navegación.

Esas etiquetas en las que se encuentran envueltos nuestros 5 componentes, se corresponden con el enrutamiento de la aplicación. La función `render` del `ReactDOM`, renderiza todos los componentes que se encuentran, si los 5 componentes no estuvieran envueltos en las etiquetas, la función renderizaría de manera simultánea esos 5 componentes, teniendo todas las vistas posibles en la misma pantalla. Este comportamiento es indeseado para nuestro sistema, ya que queremos que el usuario pueda ir navegando entre las vistas de manera ordenada, es aquí donde entra en juego el enrutamiento.

El enrutamiento consiste en organizar las vistas posibles de la interfaz en rutas, que se traducen después en una URL, a la que el navegador mandará una petición para obtener el documento HTML de esta. En cada ruta se renderizará un componente. `ReactJS` cuando se encuentra que tiene que renderizar un componente, renderiza en cascada todos los componentes contenidos en otros componentes, por lo que, si por ejemplo se renderiza el componente `App`, se renderizan los componentes que utiliza `App`, si los componentes que utilizan `App` contienen otros componentes, se renderizan y así sucesivamente hasta llegar a un componente hoja.

Sin embargo, no solo basta con el enrutamiento, debemos gestionar de alguna manera que usuarios pueden acceder o no a las distintas rutas mediante el uso de sesiones.

Las sesiones de los usuarios se controlarán mediante el uso de cookies. Las cookies son ficheros que el navegador crea y almacena en el dispositivo del usuario para almacenar información de un sitio web. En el sistema desarrollado, estas cookies serán creadas en el momento del login, ya que es cuando el usuario inicia sesión en el sistema. En concreto se crearán dos:

- **Token:** Es un token de autenticación que la capa de Back-end envía
- **User:** Es el identificador del usuario que la capa de Back-end envía

Estas dos cookies, aparte de ser necesarias para el enrutamiento, serán necesarias para enviar peticiones a la capa Back-end.

Si existen cookies, significa el usuario ha iniciado sesión y por lo tanto tiene permiso a acceder a todas las rutas, siempre y cuando las cookies sean válidas, si no, podrá acceder a un número restringido de ellas.

Todo esto se consigue haciendo uso de un componente de `ReactJS` específico para el enrutamiento, el componente **`BrowseRouter`** nombrado en nuestro sistema como **`Router`**. Este componente mantiene la interfaz sincronizada con la URL, para ello hace uso de otro componente **`Switch`**. Para explicar su funcionamiento es necesario visualizar su código [16]

```

<Switch>
  <PrivateRoute exact path="/synth" isAuthenticated={isAuthenticated()} component
={App}/>
  <PrivateRoute exact path="/profile" isAuthenticated={isAuthenticated()} component
={Profile}/>
  <PublicRoute exact path="/signup" component={SignUp}/>
  <PublicRoute exact path="/" isAuthenticated={isAuthenticated()} component={Login
}/>
  <Route component={ErrorPage}/>

</Switch>

```

Código 6.26: Switch

Si el usuario introduce una URL asociada a la aplicación en el navegador, este código solo renderiza el componente asociado a un path de una ruta, por ejemplo, supongamos que la aplicación es accesible a través de la URL “*http://localhost:3000*”, si el usuario intenta acceder a “*http://localhost:3000/synth*” se renderiza el componente App. Si el usuario ha introducido una URL que no se corresponde con ninguna de las disponibles se renderiza el componente ErrorPage.

Se puede observar que se utilizan tres tipos de rutas: PrivateRoute, PublicRoute y Route.

La ruta de tipo **Route** es el componente que ReactJS nos proporciona para renderizar el componente pasado en la propiedad “*component*” si se utiliza el enrutamiento. Las otras dos tienen un comportamiento especial. Representan componentes de ruta que reciben también la propiedad “*component*” pero además reciben otra segunda propiedad “*isAuthenticated*”. Su valor es un booleano que depende de una función que comprueba si el usuario posee las cookies de sesión o no.

La ruta de tipo **PublicRoute** es para rutas de acceso público, aunque está comprobada que, si el usuario se ha logueado, lo redirige al sintetizador

La ruta de tipo **PrivateRoute** es para rutas de acceso exclusivo a usuarios logueados en la aplicación. En caso de que alguien que no se haya logueado e intente acceder a alguna de estas rutas se le redirecciona al Login.

Este procedimiento se muestra en el siguiente fragmento de código:

```

const PrivateRoute = ({isAuthenticated, ...props}) => {
  return (isAuthenticated) ? <Route {...props} /> : <Redirect to="/" />;
};

const PublicRoute = ({isAuthenticated, ...props}) => {
  return (isAuthenticated) ? <Redirect to="/synth" /> : <Route {...props} /> ;
};

function isAuthenticated(){
  var token = Cookies.get('token')
  if( token && token !== 'undefined'){
    return true;
  }else{
    return false;
  }
}

```

Código 6.27: Rutas privada y públicas

6.4.1.4.2. Login y Registro

Los siguientes elementos de la jerarquía son las vistas de Login y Registro. Estas vistas consisten en dos formularios muy parecidos en el que el usuario introducirá los correspondientes datos y realizará la petición. Ambos formularios contarán con una validación previa antes de enviar los datos a la capa Back-end y evitar una sobrecarga de peticiones o posibles fallos en la capa Back-end al mandar datos de manera incorrecta a esta. A continuación, se detalla aspectos relevantes de la implementación de cada una de las vistas por separado.

6.4.1.4.2.1. Login

La vista del Login tiene el siguiente aspecto:

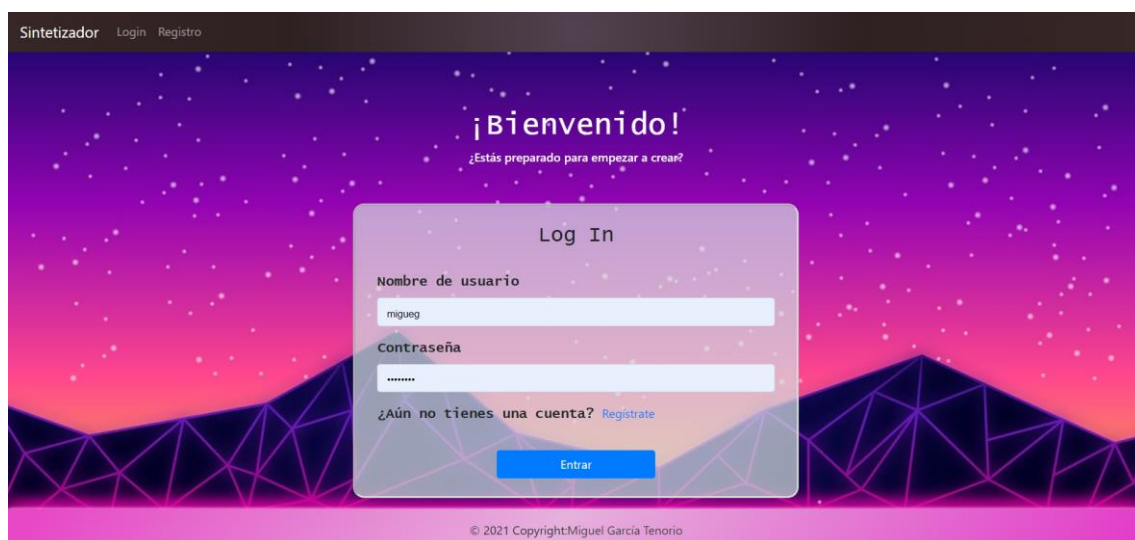


Figura 6.17. Vista Login

Cuando el usuario introduzca los datos y pinche en el botón de entrar ocurrirá lo siguiente:

1. Se previene el comportamiento por defecto del formulario
2. Se produce una validación de los datos:
 - a. Se comprueba que ninguno de los campos del formulario esté vacíos
3. Si están vacíos, se muestra un mensaje indicando que no pueden estar vacíos
4. Si no lo están, se envía la petición al Back-end
5. Si la petición tiene éxito, se crean las cookies de sesión con los datos enviados del Back-end y se redirecciona al usuario a la vista del sintetizador
6. Si la petición no tiene éxito, se muestra un mensaje indicando que el usuario no está registrado en el sistema.

Para el formulario se utiliza la etiqueta `<form>` HTML. Esta etiqueta tendrá una serie de etiquetas `<input>` que serán los datos que el usuario introduzca, por último, tiene un elemento `<button>` que será el botón de enviar. Este botón es de tipo *submit*, que provoca que cuando es pulsado se lance el evento *submit* del formulario, cuyo comportamiento por defecto es recargar la página. Este comportamiento no interesa ya que queremos un comportamiento específico. Esto se evita en la siguiente línea de código.

```
<form style={{width: '90%',  
            marginTop: '4%',  
            marginBottom: '4%',  
            marginLeft: '5%'}}  
onSubmit={(e)=>e.preventDefault()}>
```

Código 6.27: Rutas privada y públicas

Cuando se produce el evento *submit* se llama a la función *preventDefault()* que evita este comportamiento por defecto. Una vez evitado, cuando se clique en el botón se llamará a la función que valida el formulario y no se recargará la página.

Para validar los datos, se cogerán del DOM los elementos HTML a partir de sus identificadores mediante la función **getElementById()** del objeto que representa el DOM, el objeto *document* de JavaScript. Una vez obtenido el elemento, se obtiene el valor que se encuentra en la propiedad *value* de este y se valida. Este será el procedimiento estándar en la validación de los formularios.

Si la validación tiene éxito, se realiza una petición POST a la URL del Back-end correspondiente al login mediante la función *fetch()* de JavaScript.

Si la petición tiene éxito, porque nos devuelve un código HTTP de éxito, en concreto un 200, se establece las cookies de sesión y se redirecciona a la vista del sintetizador. Si no tiene éxito, se obtiene el elemento HTML que contiene el mensaje de error y se modifica su estilo para que se muestre, ya que inicialmente permanece oculto (propiedad *css display* a *none*). Esto se realiza en el siguiente código:

```

await this.__login(user) //Se realiza la petición
if(this.data.state){ //Si tiene éxito
    Cookies.set('token',this.data.msg,{expires:2, path:""})
    Cookies.set('user',this.data.user,{expires:2, path:""})
    window.location.replace('http://localhost:3000/synth')

}else{ //Si no tiene éxito
    document.getElementById('wrong-credentials').style.display = "";
}

```

Código 6.28: Login e interpretación

Nótese, que se hace uso del objeto Cookies. Este objeto es un objeto de un paquete externo llamado “js-cookies” que se encarga de crear y gestionar las cookies en una aplicación web. Las cookies serán establecidas con la propiedad “expires” a 2, significando que las cookies tendrán una validez de 2 días, de manera que el usuario pueda entrar en la aplicación sin hacer login durante 2 días.

6.4.1.4.2.2. Registro

La vista de registro tiene el siguiente aspecto:

Figura 6.18. Vista Registro

El comportamiento del formulario es exactamente el mismo que el del login. Aunque la validación es un poco más compleja, aparte de que los campos vacíos será necesario comprobar que las contraseñas tienen una longitud mínima de 8 caracteres y que ambas coincidan.

También, se realizará una petición POST, pero a la URL correspondiente del registro, pero en esta ocasión ante el éxito no se establecen cookies.

6.4.1.4.3. Sintetizador

Una vez que el usuario ha hecho login en la aplicación se le redirige a la vista del sintetizador. Esta vista tiene el siguiente aspecto:



Figura 6.19. Vista sintetizador

En esta vista podemos identificar tres zonas:

1. Una zona superior que llamaremos cabecera
2. Una zona central que llamaremos zona de síntesis
3. Una zona inferior que llamaremos piano

Esta vista se renderiza en la clase **App**, en el siguiente código:

```

render(){
  return (
    <div className="App" >
      <div className="header">
        <Header ref={this.header} showEQ={this.showEQ} showFX={this.showFX} parent
        Callback={this.updateTable} showOsc={this.showOsc} showLb={this.showLb} />
      </div>
      <div className="oscillators" id="osc">
        <OscComponents ref={this.oscillators}/>
      </div>
      <div className="FX off" id="fx">
        <FX ref={this.fx}/>
      </div>
      <div className="LB off" id="lb">
        <Library parentCallback={()=>this.loadSound()} ref={this.lb}/>
      </div>
      <div className="analyser off" style={{width: '100%'}} id='analyser'>
        <Analyser/>
      </div>
      <div className="piano">
        <Piano />
      </div>
    </div>
  );
}

```

Código 6.29: Renderizado del sintetizador

En el anterior código, se puede observar, que hay más componentes que zonas. Esto es así, porque la zona de síntesis tendrá varias posibles “*layers*” o aspectos que cada uno se corresponde con un componente distinto. Por lo tanto, tendremos:

- Para la zona de la cabecera: el componente Header.
- Para la zona de síntesis:
 - El componente Osccomponents. Son los osciladores.
 - El componente FX. Son los efectos.
 - El componente Library. Es la tabla de sonidos disponibles
 - El componente Analyser. Son los gráficos y el ecualizador
- Para la zona del piano: el componente Piano.

A todos los layers de la zona de síntesis, excepto uno (el que se muestra) se les pasará la clase “*off*” que el fichero css para el componente le incluye la propiedad “*display*” a “*none*” **para** que la etiqueta <div> de la que cuelga todo el componente no se muestre. Esta clase se irá añadiendo o quitando, a medida que se vaya navegando por los layers, pero la gestión de este proceso es responsabilidad del componente Header, por lo que se verá más adelante.

A continuación, se especifican los detalles en la implementación de las distintas zonas

6.4.1.4.3.1. Piano

Es la zona inferior de la vista del sintetizador, la cual se visualiza así:



Figura 6.20. Piano

Uno de los requerimientos del sistema es que el usuario pueda reproducir y controlar señales de audio manera interactiva.

Para que el usuario pueda reproducir notas de manera interactiva e intuitiva se ha optado por la creación de componente gráfico que simule un piano que de la sensación de ser un sintetizador virtual y así poder tocar notas.

Por lo tanto, se necesitan representar las teclas del piano de la manera más aproximada posible para garantizar la experiencia del usuario. Para ello se utiliza la **clase Piano**.

Es aspecto similar al de un piano se ha conseguido mediante una lista HTML (etiqueta ``). Cada elemento de la lista (etiquetas ``) se corresponde con una tecla del piano.

```
<li className="white" id="C" name = "first" onClick={()=>{this.notePlayed()}}>
  <p>C1</p>
</li>
```

Código 6.30: Elemento HTML correspondiente a una nota

Ese aspecto a nota se ha conseguido pasándole como clase a los elementos de la lista, la clase *White* o para las teclas blancas y la clase *black*. Estas clases serán utilizadas en el fichero css que utiliza el componente para dotar de estilo a los elementos mediante propiedades css. Estas propiedades se muestran a continuación:

```

.white{
  height: 12.4vw;
  z-index: 2;
  width: 80vw;
  border-left: 1px solid #bbb;
  border-bottom: 1px solid #bbb;
  border-radius: 0 0 5px 5px;
  box-shadow: -
1px 0 0 rgba(255,255,255,0.8) inset, 0 0 5px #ccc inset, 0 0 3px rgba(0,0,0,0.2);
  background: linear-gradient(to bottom, #eee 0%, #fff 100%);
}

.black{
  height: 7.92vw;
  color: ivory;
  width: 44vw;
  z-index: 3;
  border: 2px solid #000;
  border-radius: 0 0 3px 3px;
  box-shadow: -1px -1px 2px rgba(255,255,255,0.2) inset, 0 -
5px 2px 3px rgba(0,0,0,0.6) inset, 0 2px 4px rgba(0,0,0,0.5);
  background: linear-gradient(45deg, #222 0%, #555 100%)
}

```

Código 6.31: Propiedades css de las teclas

Los elementos de la lista estarán organizados utilizando el valor “*flex*” para la propiedad css “*display*” que permite posicionar elementos en el espacio de una manera sencilla

El usuario podrá accionar una nota del piano y por lo tanto reproducirla, cuando clique en una de las notas o presione una tecla del teclado asignada, de manera que se refleje en la interfaz. Además, es importante mencionar que solo se mostrarán dos octavas por pantalla con la posibilidad de accionar un botón para subir y bajar de octava.

Para que se reproduzca una nota cuando el usuario clique en una de las notas del interfaz, los elementos HTML de la lista detectarán el evento “*click*” en ellos y llamarán a una función que lo gestionara. Esta función identificará que nota se ha pulsado a partir del identificador del elemento HTML que llamo a la función.

Para que el usuario pueda reproducir notas desde el teclado se necesitan dos escuchadores de eventos, uno para que cuando se presione la tecla se reproduzca una nota y el piano emule de manera gráfica que se ha presionado esa nota, y otro para que cuando el usuario suelte la tecla el piano emule que se ha soltado la nota.

```
document.addEventListener('keydown', (event) => {
    const keyName = event.key;
    switch(keyName.toLowerCase()){
        case 'z':
            if(!this.pressedKeys['z'])
                this.handleKeydownEvent('white','C',keyName,'down');
            break;
        ....
    }
})
```

Código 6.32: Escuchador de eventos para cuando se presiona una tecla del teclado

Este código se ejecuta cuando se presiona la tecla y primero comprueba que no está siendo presionada ya la tecla del teclado y luego se llama a una función que maneja el evento, que se encargará de activar la clase del elemento HTML de la nota, para que mediante propiedades CSS se simule que la nota del teclado.

Por consiguiente, cuando el usuario levanta la tecla del teclado, se activará el correspondiente elemento y desactivará la clase del elemento HTML de la nota para que mediante propiedades CSS vuelva a su estado original.

```
document.addEventListener('keyup', (event) =>{
    delete this.pressedKeys[event.key]
    switch(event.key){
        case 'z':
            $('#C').removeClass('white-active');
            break;
    }
})
```

Código 6.33: Escuchador de eventos para cuando se levanta una tecla del teclado presionada

Una vez gestionados los distintos eventos, se llama a una función que se encarga de obtener la frecuencia de la nota a partir de su clave del diccionario de notas, obtenido en el momento de la creación del componente en una petición GET a la capa Back-end, y de enviar llamar al correspondiente método de la clase fachada de la subcapa controladora que delegará para reproducir la nota. La nota reproducida dependerá de la octava que se almacenará en un contador que se aumentará o disminuirá si se pulsa en el botón de subir o bajar octava.

Por último, cabe destacar, que cada vez que se presiona una tecla del teclado o se clicla en una tecla de la interfaz, se simulará gráficamente la presión de la tecla de la interfaz modificando, el color de fondo y el borde del elemento HTML que representa la tecla. El resultado se muestra en la siguiente figura:

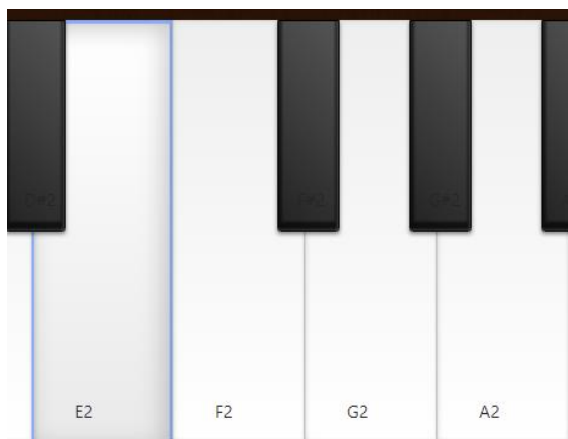


Figura 6.21: Presión de teclas

6.4.1.4.3.2. Cabecera

Se corresponde con la zona superior, tal como se muestra en la siguiente figura.

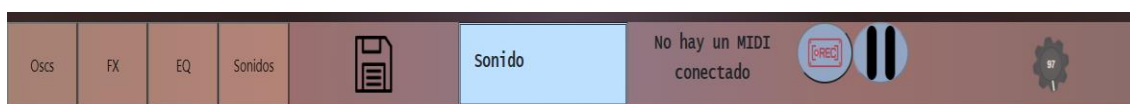


Figura 6.22: Cabecera

Los primeros elementos que podemos identificar son 4 botones. Estos botones son los responsables de alternar entre los posibles “layers” que tendrá la zona de síntesis, ya que estos “layers” no se mostrará de manera simultánea.

En el momento de la instanciación del componente en la función render de la clase App recibe una serie de propiedades fundamentales para alternar estos layers, las cuales se pueden ver en el siguiente fragmento de código:

```
<div className="header">
  <Header ref={this.header} showEQ={this.showEQ} showFX={this.showFX}
    parentCallback={this.updateTable} showOsc={this.showOsc} showLb={this.showLb} />
</div>
```

Código 6.34: Instanciación del componente header y paso de propiedades

Las propiedades que serán utilizadas en este proceso son las que su nombre empieza por show. Estas propiedades son funciones de la clase App, es decir, del padre del componente Header, para mostrar un Layer. Cuando se clique uno de los botones se llamará a la función del padre que le corresponda. Un ejemplo de estas funciones se muestra en el siguiente código:

```

showEQ(){
  var offlb = this.__isOff('lb');
  var offosc = this.__isOff('osc');
  var offfx = this.__isOff('fx');
  var offeq = this.__isOff('analyser')

  if(offeq){
    document.getElementById('analyser').classList.remove('off');
    if(!offosc){
      document.getElementById('osc').classList.toggle('off');
    }

    if(!offlb){
      this.__onOscillators();
      document.getElementById('lb').classList.toggle('off');
    }

    if(!offfx){
      document.getElementById('fx').classList.toggle('off');
    }
  }
}

```

Código 6.35: Función que muestra el componente Analyser

Estas funciones harán lo siguiente:

1. Comprueban si los componentes poseen la clase HTML off
2. Comprueba si el componente que se quiere mostrar está oculto.
3. Si está oculto, le elimina la clase off, para que se muestre
4. Para cada uno de los otros componentes comprueba si no están ocultos y si no lo están les añade la clase off para que se oculten.

Hay un caso especial y es el del componente Libray. Cuando se muestra este componente es necesario apagar los dos osciladores, ya que contará con una barra buscadora donde se podrá escribir y no queremos que se reproduzcan notas mientras escribimos en esta. Para ello cuando se va a mostrar se llama a la función **offoscillators()**, que se encarga de preguntar a la clase fachada si están encendidos los osciladores y si es así se le enviará otro mensaje para que los apague. Cuando el componente se vaya a ocultar se llamará a la función **onoscillators()** que hará lo mismo pero para que se enciendan los osciladores.

El siguiente elemento que nos encontramos en la cabecera es un icono de guardado. Cuando pulse el usuario en él, abrirá un modal que contendrá un formulario para guardar un sonido creado. Más adelante, se explicará el funcionamiento de los modales ya que todos los modales de la aplicación siguen el mismo procedimiento.

Al lado del icono de guardado, hay una zona donde se encontrará el nombre del sonido que hay actualmente cargado, cada vez que se cargue un sonido nuevo este nombre se actualizará. Mas adelante se verá como se cambia este valor ya que intervienen una serie de eventos previos que serán detallados en secciones posteriores. Junto al sonido cargado hay un mensaje que indica el estado del MIDI. En el caso de

que el controlador no detecte MIDI conectado o se desconecte uno, se mostrará el mensaje de “No hay un MIDI conectado”. Cuando se conecta uno, la subcapa controladora lo detectará, y proporcionará el nombre del dispositivo MIDI que reemplazará al mensaje anterior.

A continuación de este mensaje nos encontramos los controles de la grabación, como se muestra en la siguiente figura.



Figura 6.23: Botones de grabación

Inicialmente, se muestra solamente el botón de inicio y el de pausa, aunque hay un tercer botón que permanece oculto en el inicio y un reproductor de la grabación oculto, cuyo código HTML es el siguiente:

```
<Col className='rec' >
    <button className='button' onClick={()=>this.rec('start')} id='start'><img
src={rec}></img> </button>
    <button className='button' id='pause' disabled={true} ><img src={pausa}></img></button>
    <button className='button' onClick={()=>this.rec('stop')} id='restart' style={{display: 'none'}}><img src={reload}></img></button>
    <audio controls id='audio' style={{display: 'none'}}></audio>
</Col>
```

Código 6.36: Controles de grabación

Si se pulsa el botón de grabar la zona de grabación y luego al botón de pause, cambiará el aspecto de la zona como se muestra en la siguiente figura:

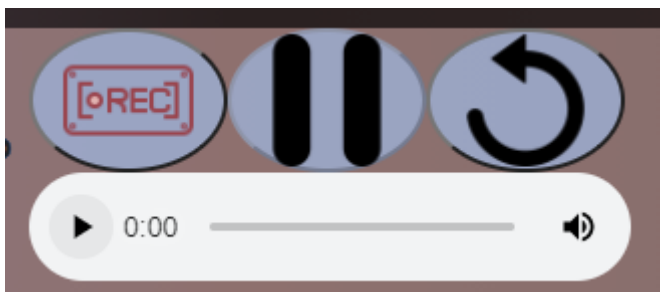


Figura 6.24: Botones de grabación

Se puede observar que aparece un botón de inicio que servirá para comenzar de nuevo la grabación y un control de audio para escuchar y descargar la grabación. Tanto la descarga como la reproducción es ajena a nuestra implementación, ya que la etiqueta HTML *audio* con la opción *controls* incorpora esta funcionalidad. Si en vez del botón de reinicio el usuario pulsa en el botón de rec, la grabación continuará y cuando se pare se

apreciará toda la grabación realizada desde que se pulso por primera vez el botón de rec (si no se ha pulsado el botón de restart), aunque en realidad todo el procesamiento se hace en la subcapa controladora, el componente Header solo se encarga de gestionar los botones y mandar mensajes a la subcapa controladora.

Tanto el botón de pausa como de inicio de grabación llaman a la función **rec()** del componente Header. Su código es el siguiente:

```
rec(state){
  if(state === 'pause'){
    document.getElementById('restart').style.display = "";
    document.querySelector('#restart').addEventListener('click',()=>this.rec('restart')
  )

    document.querySelector('#pause').disabled = true;
    document.querySelector('#pause').removeEventListener('click',()=>this.rec('pau
se'))
    document.getElementById('audio').style.display = "";
  }

  if(state === 'start'){
    document.getElementById('audio').title = 'Record'+ this.state.date.getDate()+ '-'
+ this.state.date.getMonth()+ '-' + this.state.date.getFullYear();
    document.querySelector('#pause').disabled = false;
    document.querySelector('#pause').addEventListener('click',()=>this.rec('pause')
  )

    document.getElementById('restart').style.display = 'none';
  }

  if(state === 'restart'){
    document.getElementById('restart').style.display = 'none';
    document.getElementById('audio').style.display = 'none';
  }

  }
  sinte.rec(state)
}
```

Código 6.37: Función rec

Esta función realiza lo siguiente a partir de un estado de grabación:

1. Si es pause:
 - a. Muestra el botón restar y le añade un escuchador de eventos para el click.
 - b. Desactiva el botón pause.
 - c. Muestra el elemento de control de audio.
2. Si es start:
 - a. Establece la propiedad *title* del elemento HTML del audio a la fecha actual, que se corresponderá con el nombre del archivo correspondiente a la descarga de la grabación.

- b. Activa el botón de pause y le añade un escuchador de eventos para el click.
3. Si es restart:
 - a. Esconde el botón de restart.
 - b. Y esconde el control de audio.
4. Por último, manda el mensaje a la clase fachada (orden: *sinte.rec(state)*) pasándole el estado. La clase fachada delegará en el Recorder que:
 - a. Si el estado es pause, para la grabación
 - b. Si el estado es start, la inicia.
 - c. Si el estado es restart, limpia el buffer de datos, y para la grabación.

El último elemento de Header es el Knob del volumen, que controlará el volumen general del sintetizador, concretamente el nodo master de ganancia. En el apartado de Knobs se entrará en las cuestiones relacionadas con este tipo de componentes.

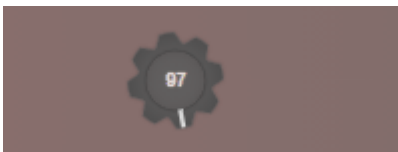


Figura 6.25: Volumen del sintetizador

6.4.1.4.3.3. Zona de síntesis

Como ya se ha mencionado, la zona de síntesis está formada por layers, los cuales son:

- Osciladores
- Efectos
- Zona de ecualización y visualización de gráficos
- Tabla de sonidos almacenados disponibles para cargar

Estos layers nos permitirán provocar acciones sobre el modelo de datos al modificar los parámetros modificables de la interfaz. Por lo general, cada vez que se modifica uno de los parámetros modificables de estos layers ocurrirá lo siguiente:

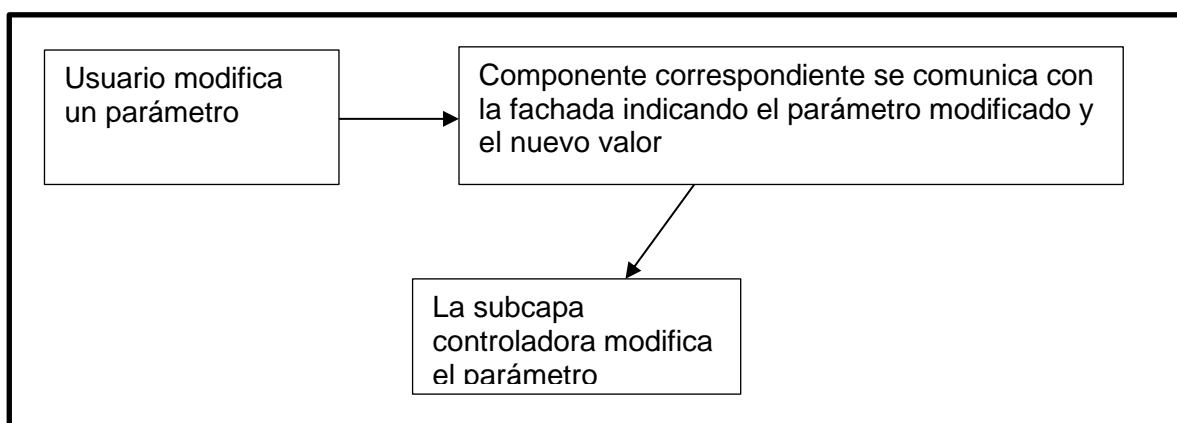


Figura 6.27: Proceso de modificación de un parámetro

A continuación, se detalla cada uno por separado

6.4.1.4.3.3.1 Osciladores

El layer de osciladores se visualiza de la siguiente manera en la interfaz:



Figura 6.26: Osciladores

Para representar los osciladores utilizaremos dos componentes oscilador, es decir, dos instancias de la **clase/componente Oscilador**.

Cada componente dependerá con una instancia de la clase oscillator del controlador. Con el uso de este componente el usuario percibirá la existencia de dos osciladores, de manera que si modifica alguno de los parámetros en realidad se están modificando todas las voces que posee el oscilador.

El componente oscilador, utiliza otro componente que representará la envolvente para conseguir más modularización y encapsulamiento del código. Este componente contendrá contenga un Knob para cada parámetro de esta, de manera que el componente oscilador estará formado por el selector de onda, una barra de paneo, un botón de On/Off, un Knob de volumen y el componente de la envolvente.

El selector de onda se corresponde con el siguiente código HTML:

```
<div className="WaveSelector">
  <select name="selector" className="selector" onClick =
  {(())=>this.checkWave(this.#osc)} id={id}>
    <option value="sine" >Sine</option>
    <option value="triangle" >Triangle</option>
    <option value="square" >Square</option>
    <option value="sawtooth">Sawtooth</option>
  </select>
  <div className="select_arrow"></div>
</div>
```

Código 6.38: Selector de onda

EL código de arriba, es un select con varias opciones, que se corresponde con tipos de onda. Cuando un usuario selecciona una opción, se llama a la función *checkWave* encargada de obtener el valor seleccionado del elemento HTML mediante la función *getElementById* de JavaScript y enviarle la selección a la fachada, para que modifique el tipo de onda de todas las voces.

Todos los parámetros modificables en la interfaz, se corresponderán con un método en la fachada, que se encargará de llamar a los setters correspondientes en las distintas clases de la subcapa controladora para que modifiquen el parámetro.

El botón de On/off se corresponde con *<input>* de tipo checkbox que cuando sea pulsado llamará a la función *checkChecked()* que comprueba que interruptor está

accionado y manda los mensajes a la fachada para que encienda o apague el oscilador, concretamente estos dos:

```
sinte.setChecked('id',true/false);
sinte.onOscillator('id');
```

Código 6.39: Mensajes a la subcapa controladora

Por último, la barra de paneo será un input de tipo range, que establecerá un rango entre -1 (izquierda) y 1(derecha) y modificará los valores de 0.1 en 0.1 unidades. Cuando se interaccione con la barra se llamará a la función setPan que comunicará el cambio a la fachada.

```
<input type="range" id={'range'+this.#osc} onChange={()=>this.setPan(this.#osc)}
      defaultValue='0' step='0.1' min="-1" max="1"
      style={{width: '80%'}}
/>
```

Código 6.40: Barra de paneo

6.4.1.4.3.3.2 Efectos

El layer de efectos estará representado por el componente **FX**, que a su vez estará compuesto de 4 componentes: **Delay**, **Distorsion**, **Reverb**, **Filter**, los cuales heredan del componente padre **Effect**. Se visualizan de la siguiente manera:

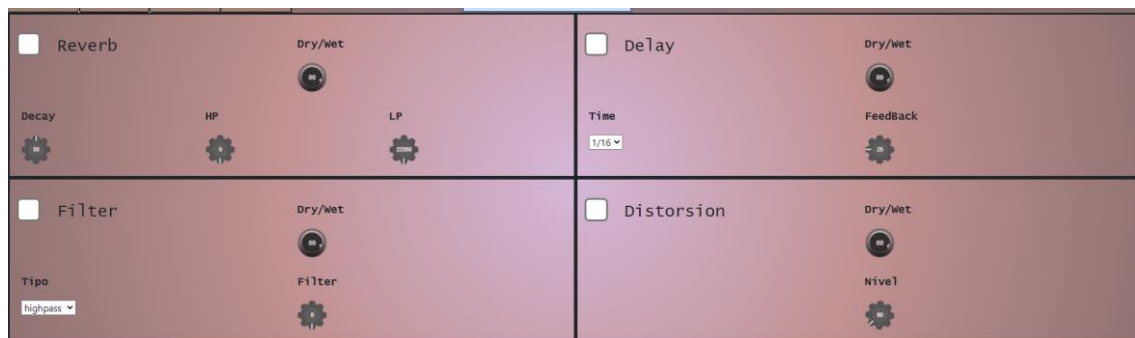


Figura 6.27: Efectos

Estos componentes tendrán contarán con Knobs para modificar parámetros, selectores y checkboxes para apagar o encender el efecto. Estos componentes funcionan igual que el componente oscilador, cuando se mueve un Knob o se pulsa en un checkbox o se elige una opción del selector, se obtiene ese valor modificado y se manda a la fachada para que llame al setter correspondiente para el valor.

6.4.1.4.3.3.2 Zona de ecualización y visualización

Este layer se visualiza de la siguiente manera:



Figura 6.28: Analizador y ecualizador

Este layer se corresponde con el componente **Analyser**. Se pueden observar dos partes bien diferenciadas, la de la izquierda, que será la zona donde se pinten los gráficos y la zona de la derecha, que contiene unas barras para poder aplicar la ecualización.

Para entender cómo se pintan los gráficos en la zona de la izquierda, es necesario visualizar su código:

Código 6.41: Analizador

```
<div className='colum' onMouseLeave={()=>{this.hideIndicator()}} onMouseMove={(e
vent)=>this.detectFreq(event)} id='canva-
cont' style={{width: '50%', height: '100%',order:1 }} >
  <div className='freqs'>
    <p id='0k'>200hz</p>
    <p id='1k'>12khz</p>
    <p id='20k'>24khz</p>
  </div>
  <div id='freq-indicator' style={{
    display: 'none',
    margin: 0,
    padding: 5,
    backgroundColor: 'whitesmoke',
    width: '10vh',
    textAlign: 'center',
    borderRadius: '60%'}}>

  </div>

  <canvas id='canva' onClick={(e)=>this.createCanva(e)} style={{width: '10
0%', height: '100%' }}>
    Alterantive

  </canvas>
</div>
```

Código 6.41: Analizador

Analizando el código de arriba abajo lo primero que nos encontramos es que el `<div>` que contiene la zona va a detectar dos eventos: `"mouseleave"`, `"mousemove"`. El primero hace referencia cuando el cursor del usuario abandona la zona que comprende la etiqueta `<div>` en pantalla, y el segundo, cuando mueve el ratón por el espacio que ocupa la etiqueta `<div>` en pantalla.

Cuando se producen estos eventos se llaman a dos funciones que gestionan los eventos:

→ **detectFreq:** Gestiona el evento `"mousemove"`. Se encarga de mostrar y calcular un elemento que mostrará a que frecuencia del analizador está señalando el usuario con su cursor, tal como se muestra en la siguiente figura:



Figura 6.29: Detector de frecuencia

Mostrar esta frecuencia no es tarea trivial, ya que se necesitar realizar una medición de la zona que ocupa el analizador para poder estimar a que frecuencia está apuntando el usuario. Para ello se realiza lo siguiente

1. Se obtienen la posición (x,y) en píxeles donde se encuentra el cursor del usuario haciendo uso de JQuery .

```
var x = event.pageX - $('#canva-cont').offset().left;  
var y = event.pageY - $('#canva-cont').offset().top;
```

Código 6.42: Posición del cursor

2. Se obtienen los píxeles que ocuparán cada valor del buffer que nos llegue del analizador de la subcapa controladora, dividiendo el ancho del canva entre la longitud del buffer, y se obtiene cuantas frecuencias se pintarán por pixel, dividiendo el número de frecuencias que hay por cada valor del buffer (obtenido de la subcapa controladora) entre los píxeles por valor, tal como se muestra en las siguientes líneas de código:

```
var pixelPerValue = document.querySelector('canvas').width / this.state.bufferSize
var freqPerPixel = this.state.separation / pixelPerValue
```

Código 6.43: Píxeles por valor y frecuencia por píxeles

3. Se obtiene la frecuencia multiplicando la posición x del cursor por las frecuencias por píxeles y dividiendo es producto entre el escalado que tiene el gráfico, y se introduce la frecuencia en el div que la mostrará y se traslada mediante JQuery a la posición del cursor del usuario.

```
var freq = (x * freqPerPixel) / 2.5
var div = document.getElementById('freq-indicator')
div.innerText = freq + 'hz';
$('#freq-indicator').css("transform","translate3d("+x+"px,"+y+"px,0px)");
```

Código 6.44: Caculo de frecuencia y traslación del div

Cuando un sonido se reproduce, el analizador pintará un gráfico, parecido al de la siguiente figura:

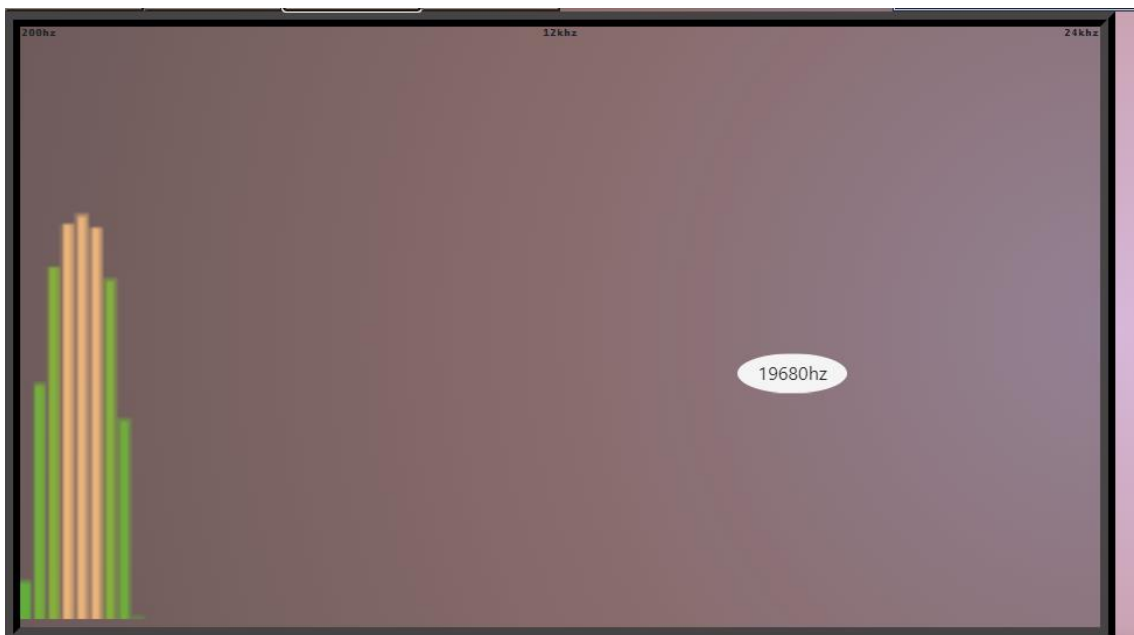


Figura 6.29: Gráficos

Estas barras se irán pintando y cambiando de color de manera sincronizada al sonido. Para ello es necesario hacer uso de la etiqueta `<canva>` destinada a contener los gráficos en el documento HTML. El canva será rellenado gracias a la función **createCanva()** del componente Analyser. En el momento de la creación del componente esta función es llamada y realiza lo siguiente:

1. Se obtiene el elemento canva del DOM.
2. Se obtiene el contexto de dibujo en 2 dimensiones donde se pintarán las barras.
3. Se obtiene la longitud del buffer con la información de pintado mandándole un

- mensaje a la fachada.
4. Se llama a una función *translateCanva()*, que posiciona de manera adecuada el gráfico de barras.
 5. Se llama a la función *draw()* que se encargará de pintar el gráfico de barras en el canva.

```
createCanva(event){
  this.state.canva= document.querySelector('canvas');
  this.state.ctx = this.state.canva.getContext('2d');
  this.state.bufferSize = sinte.getThingsAnalyser('size');
  this.translateCanva()
  this.draw()
}
```

Código 6.45: Creación del canva

Para que el gráfico de barras aparezca de manera adecuada habrá que modificar la posición del canva en la pantalla. Para ello, la función ***translateCanva()*** realizará lo siguiente:

1. Obtiene las dimensiones en pixeles del canva.
2. Calcula el centro tanto en el eje x como en el y.
3. Traslada el canva a las coordenadas de centro calculadas
4. Rota el canva 180°
5. Lo escala a -1 en el eje x, para que se cambie de orientación
6. Lo traslada al punto de centro
7. Limpia el canva de cualquier dibujo que pudiera estar ya pintado

Estas acciones son necesarias para que se visualice correctamente, si no las barras se visualizarían en sentido contrario al de la Figura 6.29 en el eje Y.

```
translateCanva(){
  this.state.width = this.state.canva.width;
  this.state.height = this.state.canva.height;
  var centery = this.state.height/2;
  var centerx = this.state.width/ 2;
  this.state.ctx.translate(centerx, centery);
  this.state.ctx.rotate(180 * Math.PI / 180)
  this.state.ctx.scale(-1, 1)
  this.state.ctx.translate(-centerx, -centery)
  this.state.ctx.fillStyle = 'rgb(0,0,0)';
  this.state.ctx.clearRect(0,0, this.state.width , this.state.height)
}
```

Código 6.46: Posicionamiento del canva

Ya posicionado de manera adecuada el canva, solo falta pintar el gráfico en él. Para ello se hace uso de la función **draw()**, que realiza lo siguiente:

1. Se indica al navegador que se va a realizar una animación y solicita que programe el repintado del canva para el próximo frame. Para ello se utiliza la función `window.requestAnimationFrame()` pasándole como argumento la función `draw`, de manera que continuamente se va a estar pintando el canva ya que en cada frame se llama a esta función. Este repintado puede llegar hasta 60 veces por segundo, por lo que se tendrá una animación bastante sincronizada con el sonido.
2. Se limpia el canva del frame anterior.
3. Se calcula el ancho de la barra que se va a pintar.
4. Se obtiene el buffer con los datos pintados de la fachada, la cual lo obtiene del analizador de la subcapa controladora.
5. Se recorre el buffer y para cada posición de este
 - a. Se calcula la altura de la barra, que será la mitad del valor de la posición.
 - b. Se comprueba el valor de la posición, el cual representa una intensidad de un rango de frecuencias, de manera que a valores altos se establece un color de la barra rojo, a valores intermedios amarillos y a valores bajos verdes.
 - c. Se pinta la barra en la posición que indique un contador para la posición.
 - d. Se incrementa el contador de posición con el ancho de la barra para pintar la siguiente a continuación.

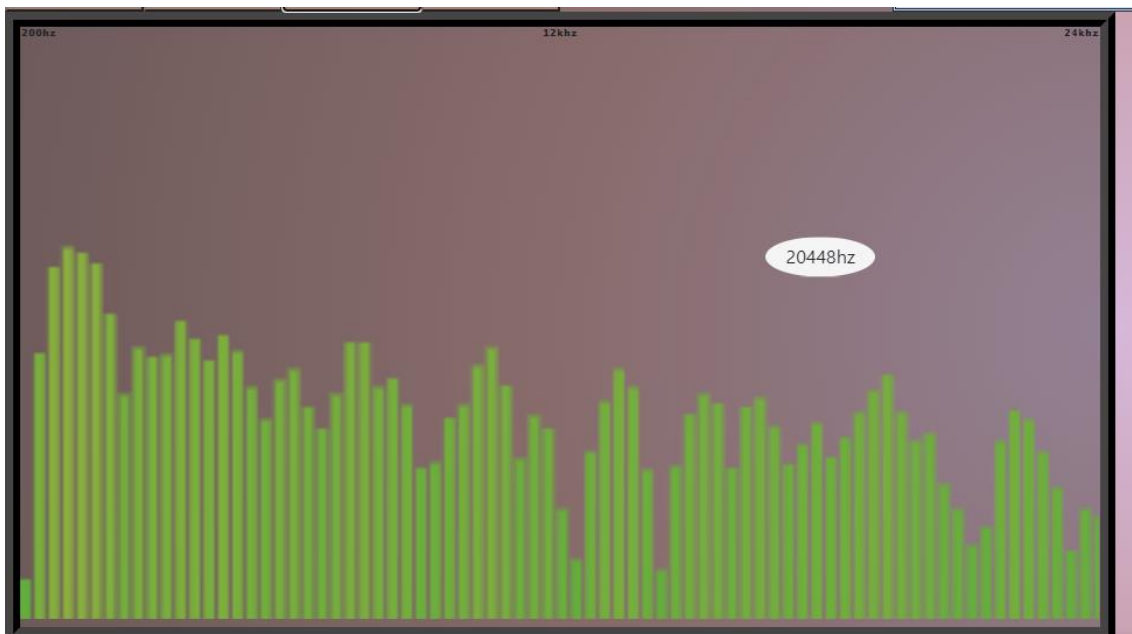


Figura 6.30: Canva dibujado

```

draw(){

    var d = window.requestAnimationFrame(this.draw)

    this.state.ctx.clearRect(0,0, this.state.width, this.state.height)

    var bufferSize = this.state.bufferSize;

    //Calculamos el ancho de la barra que se va a puntar
    var width = ( this.state.width/ bufferSize ) *2.5 ;
    var heigth;
    var x = 0;
    var data = sinte.getThingsAnalyser('data');

    for(var i = 0; i < bufferSize; i++){
        heigth = data[i]/2
        if(data[i] <= 210){
            this.state.ctx.fillStyle = 'rgb(' + (heigth+20) + ',172,46)';
        }else if (data[i] > 85 && data[i] <= 240){
            this.state.ctx.fillStyle = 'rgb(' + (heigth+192) + ',174,114)';
        }else{
            this.state.ctx.fillStyle = 'rgb(' + (heigth+100) + ',50,50)';
        }
        this.state.ctx.fillRect(x,0,width, heigth);
        x += width + 1;
    }
}

```

Código 6.47: Dibujado del canva

A la derecha del canva se encuentra la zona de ecualización. Esta zona se compone de etiquetas *<input>* de tipo *range* rotadas sobre el eje x 90° para que se vean de manera vertical. Su implementación no difiere en nada visto hasta ahora, cuando se mueve el rango, se enviará a la fachada el correspondiente mensaje con la banda que se está modificando y el valor nuevo, que se encargará de delegar para modificar el valor indicado y finalmente ecualizar el sonido de manera adecuada.

6.4.1.4.3.3.2 *Tabla de sonidos*

El último layer que el usuario podrá visualizar será la tabla de sonidos, el cual consiste en una biblioteca de sonidos creados en el sistema previamente guardados. Este layer tiene el siguiente aspecto:

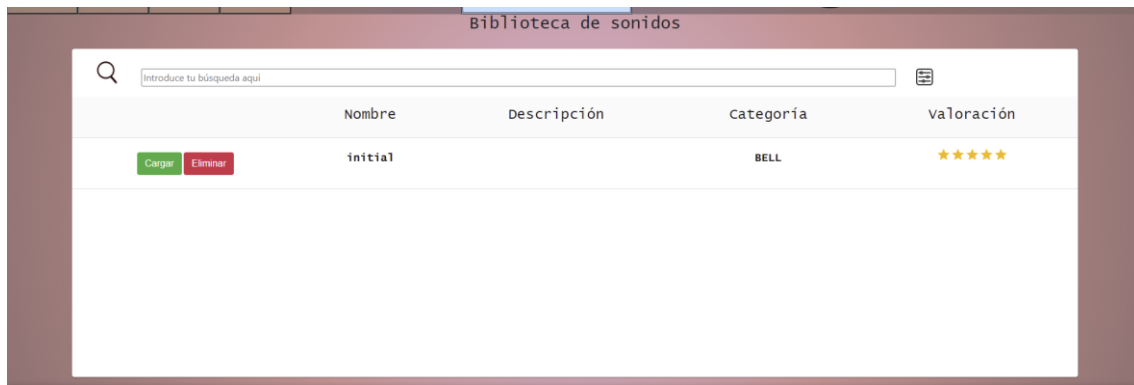


Figura 6.31: Tabla de sonidos

Esta tabla de sonidos está representada en el componente **TableUI**. Además de mostrar los sonidos almacenados, permitirá buscar en la tabla un sonido, filtrar la búsqueda y mandar órdenes a la fachada para eliminar y cargar un sonido.

Al pulsar el botón de cargar o eliminar se abrirá un modal, representado por otro componente que se encargará de comunicarse con la fachada. Más adelante se hablará de la implementación de estos modales.

Las filas de la tabla se pintarán de manera dinámica, es decir, en función de los datos de sonidos que nos lleguen de la capa Back-end pasando previamente por la subcapa controladora. Estos datos son pedidos en el momento de la construcción del componente, tal como se muestra en el siguiente código:

```
async updateTable(){
  const metadata = await sinte.fetchThings('metadata');
  if(metadata){
    this.setState({metadata: metadata , data: true});
  }else{
    this.setState({ data: false});
  }
}

async componentDidMount(){
  await this.updateTable();
  const categories = await sinte.fetchThings('categories');
  this.setState({categories: categories})
}
```

Código 6.48: Obtención de sonidos disponibles

Una vez obtenidos en la función *render()* del componente se comprobará si hay dato, si es así se mapeará el array de datos y para posición se pintará una fila. Si no hay datos se muestra un mensaje de error.

```

render(){
  .....
  <TableBody id='tableBody'>
    {
      this.state.data ? (

        this.state.metadata.map((row) => {

          return (
            <TableRow key={row.name}>
              <TableCell align='center'>
                .....
              </TableCell >
                .....
            )
          )
        }
      ) : (
        .....
      )
    }
  </TableBody>
}

```

Código 6.49: Filas dinámicas de la tabla

Lo verdaderamente interesante de este componente es la búsqueda y el filtrado. Se podrá realizar una búsqueda por nombre y aplicar los filtros de la siguiente figura:

Figura 6.32: Filtros

La búsqueda se hace a nivel de capa Front-end, es decir, se busca entre los elementos del DOM que contienen los datos de los sonidos disponibles, los cuales previamente se obtuvieron de la capa Back-end. La barra de búsqueda se corresponde con una etiqueta `<input>` que detectará el evento `"keyup"` que se produce cuando se escribe en el input, es decir, en la barra de búsqueda. Cuando se detecte este evento, se llamará a la función **`searchInTable()`** que realizará lo siguiente:

1. Obtiene del DOM los siguientes elementos:
 - a. La tabla entera
 - b. Los caracteres que hay escritos actualmente en la barra de búsqueda, es decir, el valor del input.
 - c. El checkbox de aplicar filtros.
2. De la tabla extrae todas las filas, todas las etiquetas `<tr>`, con la función `getElementsByTagName('tr')`.
3. Obtiene el rango de valoración (igual, mayor, menor).
4. Obtiene el valor filtro de categoría
5. Recorre las filas, y para cada fila:
 - a. Obtiene las columnas
 - b. Obtiene los datos de las columnas
 - c. Si hay datos en la fila
 - i. Comprueba si se aplican filtros mediante el valor del checkbox:
 1. Si los hay comprueba, si se introdujo algún carácter en la barra de búsqueda y delega en otra función para que filtre

por los valores de los filtros y los caracteres introducidos si los hubiera.

2. Si no los hay, no hay que filtrar la tabla, así que comprueba si el nombre del sonido de la fila actual coincide con los caracteres introducidos. Si no coincidiesen ocultan la fila de la tabla poniendo la propiedad *display* a *none*.

```
searchInTable(){
    var td,name,category,rating;
    var table = document.getElementById('tableBody')
    var toSearch = document.getElementById('toSearch').value.toUpperCase();
    var tr = table.getElementsByTagName('tr')
    var withFilters = document.getElementById('flexCheckChecked').checked;

    var range = $('#rangeRating').val();
    var categoryFilter = $('#categoryFilter option:selected').text();

    for(var i = 0; i < tr.length; i++){
        td = tr[i].getElementsByTagName('td');
        name = td[1]
        category = td[3]
        rating = td[4].getElementsByTagName('p')[0]
        if(category && rating && name){
            name = name.textContent || name.innerText;
            if(withFilters){
                category = category.textContent || category.innerText;
                rating = rating.textContent || rating.innerText;
                if(toSearch){
                    this.filterTable(name,toSearch,category, categoryFilter,rating,tr[i], rang)
                }else{
                    this.filterTable("",toSearch,category, categoryFilter,rating,tr[i], range)
                }
            }else{
                if(name){
                    if(name.toUpperCase().indexOf(toSearch) > -1 ){
                        tr[i].style.display = "";
                    }else{
                        tr[i].style.display = "none";
                    }
                }else{
                    tr[i].style.display = "";
                }
            }
        }
    }
}
```

Código 6.50: Búsqueda en tabla

La función que se encarga de realizar el filtrado es la función ***filterTable()***. Recibe como parámetros:

- Nombre del sonido de la fila que se está comprobando en la función de búsqueda
- Los caracteres introducidos en la barra de búsqueda
- La categoría del sonido de la fila
- La categoría por la que se quiere filtrar
- La valoración del sonido de la fila
- El elemento HTML de la fila
- El rango de valoración que se va a utilizar en el filtro

A partir de estos parámetros recibe lo siguiente:

1. Obtiene el valor numérico del filtro de valoración.
2. Comprueba si hay nombre que buscar
 - a. Si lo hay, comprueba que el nombre del sonido comprueba con los caracteres introducidos y que la categoría del sonido coincide con la del filtro
 - i. Si se cumple la condición, comprueba, mediante otra función auxiliar que devolverá un booleano, que la valoración del sonido se encuentra dentro del rango de filtrado. El valor devuelto se almacena en una variable de control
 - ii. Si no se cumple, se pone la variable de control a false.
 - b. Si no lo hay, solo es necesario aplicar los filtros, así que solo se comprueba la categoría y se hace lo mismo que en el caso anterior.
3. Si la variable de control es true se muestra la fila y si no, se oculta la fila.

```
filterTable(name,toSearch,category,filter,rating, tr,range){
  var ratingfilter = $('#rating').val();
  var show = false
  if(name){
    if( name.toUpperCase().indexOf(toSearch) > -1
      && category.toUpperCase().indexOf(filter) > -1 ){
      show = this.__checkRange(range,rating,ratingfilter)
    }else{
      show = false;
    }
  }else{
    if( category.toUpperCase().indexOf(filter) > -1 ){
      show = this.__checkRange(range,rating,ratingfilter)
    }else{
      show = false
    }
  }
  if(show){
    tr.style.display = "";
  }else{
    tr.style.display = "none";
  }
}
```

Código 6.51: Filtrado de la tabla

6.4.1.4.4. Knobs

Observando la interfaz, nos encontramos que múltiples parámetros son modificables mediante el uso de Knobs. Estos Knobs tendrán dos tipos de aspectos como se muestran en la siguiente figura:

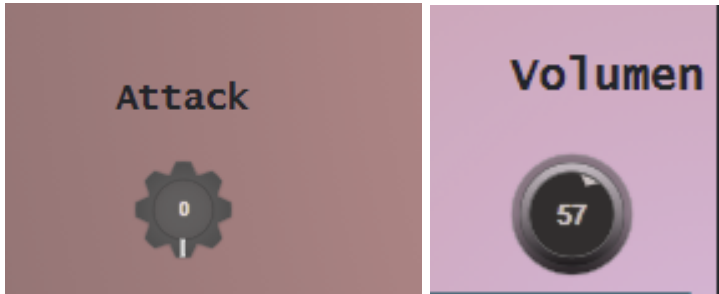


Figura 6.33: Knobs

Estos knobs son representados mediante un componente implementado en la clase **limitedKnob** que limita y controla el comportamiento de un componente externo e importado en el sistema llamado **Knob**. Cada vez que un componente necesite un Knob, incluirá el componente **limitedKnob** y lo renombrará como **Knob**, de manera que en la función de renderizado tendrá un código parecido al siguiente:

```
<Knob
  style={{ display: "inline-block" }}
  min={0}
  max={100}
  width={200}
  height={200}
  type={knobTypes.OSC_VOLUM}
  osc = {this.#osc}
  val={0}
/>
```

Código 6.52: Incluir componente Knob

Se puede apreciar que el componente acepta las siguientes propiedades:

- Style: es el estilo que en línea que tendrá el componente
- Min: Mínimo valor posible del Knob
- Max: Máximo valor posible del Knob
- Width: Ancho en píxeles que ocupará el Knob.
- Height: Alto en píxeles que ocupará el Knob
- Type: tipo de Knob
- Osc: Oscilador al que afectará, solo en el caso de que sea de un oscilador
- Val: Es el valor inicial

El componente renderiza a su vez al componente externo, que es el que aportará la parte gráfica

```

render() {
  let { value, ...rest } = this.props;

  return (

    <Knob value={this.state.value} onChange={this.handleChange}
    rotateDegrees={180} skin={this.#skin} {...rest} />

  );
}

```

Código 6.53: Función render LimitedKnob

Lo interesante del código de arriba es la propiedad `onChange`, que depende de la función `handleChange` del componente `LimitedKnob`. Esta propiedad ejecutará la función siempre que el usuario mueva el Knob y por tanto modifique su valor. La función a la que llama se encarga de limitar el movimiento del Knob, ya que el componente Knob no limitaba el movimiento y el usuario podía rotar indefinidamente el Knob sin un límite inferior ni superior. Este límite, nos interesa ya que no deseamos que el usuario pueda pasar del máximo valor al mínimo, porque produciría cambios bruscos en los valores que repercutirían de mal manera en los sonidos que se estuvieran reproducción, generando una falsa sensación de error en la reproducción.

Para ello cuando un usuario mueve un Knob en la función `handleChange()` ocurrirá lo siguiente:

1. Se establece una máxima distancia que pueda haber entre el valor mínimo y máximo en función de tipo de Knob, ya que no todos los Knobs giran en el mismo rango de valores.
2. Se calcula la distancia entre el valor actual del Knob y el valor anterior.
3. Si la distancia supera a la máxima no se modifica el valor del Knob
4. Si no se supera, se comprueba el tipo del Knob que se ha movido y se delega en la respectiva función que se encargará de, modificar el valor del Knob, que se encuentra en el estado del componente y formateará el valor y lo enviará a la subcapa controladora. Cada tipo de Knob se corresponderá con un setter del parámetro correspondiente en la subcapa controladora.

```

__handleChangeOscillator(val){
  this.setState({ value: val });
  sinte.setVolum(this.osc,this.state.value);

}

```

Código 6.54: Ejemplo de envío de valores del Knob a la subcapa controladora


```

handleOnChange(val) {
  this.#value = val;
  var maxDistance = 10
  if(this.#type === types.REVERBHPF
    || this.#type === types.REVERBLPF
    || this.#type === types.FILTERCONTROL){
    maxDistance = 2300;
  }
  if(this.#type === types.DISTORSIONAMOUNT){
    maxDistance = 60;
  }
  let distance = Math.abs(val - this.state.value);
  if (distance > maxDistance ) {
    return;
  } else {
    switch(this.#type){
      case types.OSC_VOLUM:
        this.__handleOnChangeOscillator(val);
        break;
        .....
    }
  }
}

```

Código 6.55: Función handleOnChange

6.4.1.4.5. Modales

El último aspecto relacionado con la implementación de la subcapa de la vista, es el uso de modales. Los modales son zonas emergentes que permanecen ocultas y se mostrará cuando se pulse algún botón de manera que el usuario no pueda interactuar con nada más, solo con el modal. Toda la funcionalidad esencial de los modales se ha encapsulado en un componente padre llamado **Modal**. De este componente heredarán todos los tipos de componentes posibles. Por consiguiente, existirán los siguientes tipos de modales:

- ModalDelete: Se mostrará cuando se pulsa en el botón de eliminar un sonido.
- ModalLoad: Se mostrará cuando se pulsa en el botón de cargar un sonido.
- ModalPassword: Se mostrará cuando se pulsa en el botón de modificar contraseña en la vista del perfil
- ModalSave: Se mostrará cuando se pulse en el icono de guardar.

El bloque visual de todos los modales presentará en el mismo estilo, aunque el contenido de ellos diferirá en función del tipo de modal, por ejemplo, el ModalSave se visualiza de la siguiente manera:



Figura 6.35: ModalSave

Todos los modales se mostrarán y ocultarán de la misma manera, modificando la propiedad *display* de los elementos del DOM que comprenden el modal en función de su tipo, por ejemplo, la función que muestra el modal se corresponde con el siguiente código:

```
__showLoader(){
  if(this.state.type === 'save'){
    $('#save-button').attr('disabled',true);
    document.getElementById('md-body').style.display = 'none';
    document.getElementById('loader').style.display = 'block';
  }else if(this.state.type === 'password'){
    $('#password-button').attr('disabled',true);

    document.getElementById('md-body-password').style.display = 'none';
    document.getElementById('loader-password').style.display = 'block';
  }else if(this.state.type === 'delete'){
    document.getElementById('md-body-delete').style.display = 'none';
    document.getElementById('loader-delete').style.display = 'block';
  }else{
    document.getElementById('md-body-loader').style.display = 'none';
    document.getElementById('loader-loader').style.display = 'block';
  }
}
```

Código 6.56: Función que muestra el modal

Todos los modales que poseen formularios pasan por los mismos estados al enviar los datos a la clase fachada:

1. Muestran un símbolo de cargado mediante propiedades css
2. Cuando le llegan los datos, los interpretan
3. Si el resultado ha sido el adecuado, muestran un icono de éxito y el mensaje
4. Si no muestran un mensaje de error.

Por lo tanto, se puede deducir, que la subcapa de la vista será capaz de gestionar los errores relacionados con peticiones de una manera adecuada.

Es importante destacar la serie de eventos que se dan en el ModalLoad. Este modal tiene el siguiente aspecto:

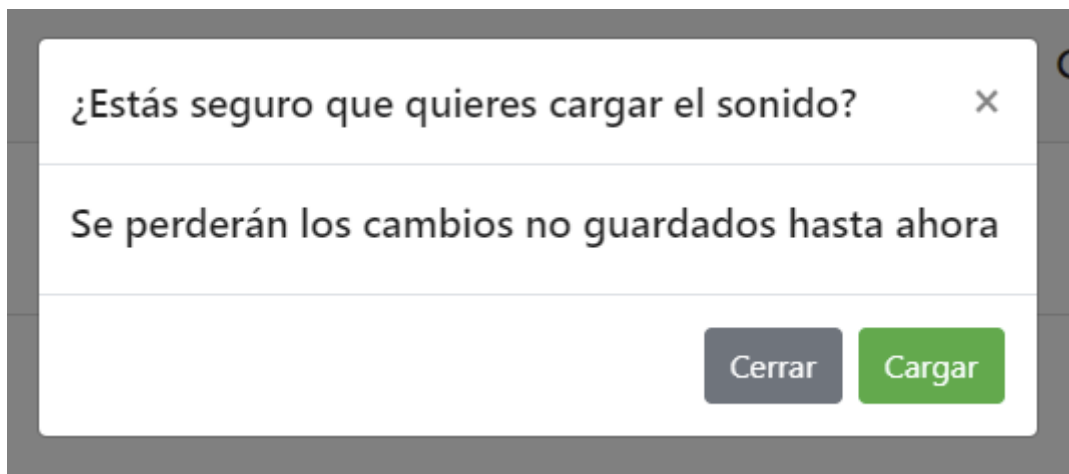


Figura 6.36: ModalLoad

En el caso de que el usuario desee cargar el sonido y pulse en el botón de cargar, se produce una serie de mensaje en cadena entre componentes:

1. El componente llama a la función *loadSound* del componente padre TableUI (el que lo contiene). Este se encarga de comunicarle al controlador que sonido hay que cargar.
2. Cuando la subcapa controladora haya devuelto una respuesta, se va transmitiendo esa respuesta entre los componentes padre hasta llegar al componente App, que es la raíz del sintetizador. Esa transmisión de un componente hoja hasta llegar a la raíz, se consigue pasando como propiedades funciones del componente padre al componente hijo. Si se le pasa una función como propiedad al hijo este podrá ejecutarla.
3. En la clase App, mediante referencias mandará mensaje a los componentes para que modifiquen sus estados con los datos del sonido a cargar, utilizando las referencias de ReactJS, las cuales permiten llamar a métodos de componentes hijos.
4. Cada componente actualizará sus datos y realizará llamadas a la subcapa controladora para que sus objetos también modifique los datos y este toda la capa actualizada.

```

loadSound(){
  var newState = this.lb.current.newState //Llamada a un método de una referencia
  var response = true;
  if(newState.name){
    this.header.current.setName(newState.name);
    if(newState.oscA && newState.oscB ){
      //console.log(newState)//A cargar
      this.oscillators.current.setOscA(newState.oscA);
      this.oscillators.current.setOscB(newState.oscB);

      if(newState.filter && newState.delay && newState.distorsion
        && newState.reverb){
        this.fx.current.setFx('delay',newState.delay);
        this.fx.current.setFx('distorsion',newState.distorsion);
        this.fx.current.setFx('filter', newState.filter);
        this.fx.current.setFx('reverb',newState.reverb);
      }
    }
  }else{
    response = false
  }
}

```

Código 6.57: Actualización del estado de los componentes tras la carga de un sonido

6.4.2. Backend

El Frontend del sistema necesita una serie de datos para funcionar adecuadamente. Estos datos se encuentran en la capa de Datos y son gestionados y extraídos por la capa Back-end. Estas capas son de dimensiones menores a las del Front-end, ya que nuestro sistema no requería tanto procesamiento en el lado del servidor. No obstante, estas capas tendrán responsabilidades vitales en correcto funcionamiento del sistema. A continuación, se detallará todo lo relacionado con la implementación de ambas capas

6.4.2.1. Creación de la capa de datos

Al contrario de la capa Front-end y la capa Back-end, la capa de datos no posee una funcionalidad implementada en código, simplemente se corresponde con una base de datos MongoDB con la funcionalidad propia de la base de datos. Para simplificar las tareas de desarrollo, tanto su creación como gestión son responsabilidad de la capa Back-end. Para poder realizar tal tareas se utiliza el framework **mongoose**.

El primer paso antes de proceder a la creación de la capa de datos es conectar con la base de datos MongoDB, para ello usaremos la función connect de mongoose que nos conectará con ella para realizar comunicaciones entre capa de datos y capa Back-end.

```

const mongoose = require('mongoose');
//Conectamos con la base de datos
mongoose.connect('mongodb://localhost:27017/sintetizador', {
  useNewUrlParser: true,
  useUnifiedTopology: true
})
.then(db => console.log('Conectado a la Base de Datos'))
.catch(error => console.error(error));
return response
}

```

Código 6.58: Conexión con la base de datos

Una vez conectada, se puede proceder a la creación de la estructura de datos que conformará la capa de datos. Al ser una base de datos no relacional, no existen tablas, sino que los elementos se organizan por colecciones. Las colecciones se componen de documentos que no tienen porque tener la misma estructura, esto podría resultar un poco caótico a la hora de organizar nuestros datos. Es aquí donde entran en juego los esquemas o Schemas de mongoose. Estos schemas nos permiten dotar de una estructura a las colecciones.

Un schema se define mediante un objeto en el que las claves se corresponderían con las columnas de la tabla y los valores al tipo de dato. Los schemas, pueden ser instanciados. Una instancia del schema, se correspondería con una fila y en nuestro caso, al tratarse de una base de datos no relacional, se correspondería con un documento dentro de una colección. Un ejemplo de la definición de un schema se muestra en el siguiente código:

```

const filter = new mongoose.Schema({
  effectOn: {type: Boolean},
  wet: {type: Number},
  type: {type: String},
  frequency: {type: Number}
});

```

Código 6.59: Ejemplo de definición de Schemas

Los schemas pueden contener a su vez otros schemas y la estructura de los documentos resultantes será la combinación de la estructura del schema con los schemas contenidos. Cuando se usen los schemas, no será posible salirse de la estructura definida, si no mongoose arrojará un error que se traducirá en un error en el sistema. Sin embargo, los schemas representan documentos, pero no colecciones, así que debemos indicarle a mongoose que schema va a seguir una colección, para que todos los documentos de una colección se ciñan a una determinada estructura. Esto se indica haciendo uso de ***mongoose.model('nombre', Schema)***, que aparte de dotar estructura a una colección proporciona una instancia de la colección necesaria para poder realizar operaciones en ella.

Poniendo como ejemplo a la colección de usuarios, se creará el correspondiente esquema y se le asignará a un modelo o colección:

```
const userSchema = new mongoose.Schema({
  username: {type: String, unique: true},
  password: {type: String, unique: true, length: 80},
  role: {type: String, enum: ["user", "admin"]},
  email: {type: String, unique: true},
  date: {type: Date},
  created: {type: Date}
})
module.exports = mongoose.model('users', userSchema);
```

Código 6.60: Creación del Schema para la colección de Users

De acuerdo al schema, cuando exista algún documento para esa colección, se visualizará en base de datos de la siguiente manera:

```
{
  _id: ObjectId("609934fcd81d224468c5981c"),
  username: "migue",
  password: "$2a$10$hmchRQHLOSMVe9oNmNx3T0g3gzamkD8KM1x062acCKv10aURVfQE6",
  role: "user",
  email: "miguegarciatenorio@gmail.com",
  date: 1999-06-28T00:00:00.000+00:00,
  created: 2021-04-09T22:00:00.000+00:00,
  __v: 0
}
```

Figura 6.37: Visualización del Schema en base de datos

6.4.2.2. Acciones con la capa de datos

Los schemas, aparte de dotar de estructura a las colecciones, proporcionarán también una serie de métodos para realizar acciones en base de datos, como consultas, inserciones, actualizaciones, borrados etc... En concreto los métodos los proporcionan las instancias de las colecciones.

Se utilizarán los siguientes:

- **findOne:** Es el equivalente a una operación SELECT, pero devuelve un solo documento. Puede recibir los siguientes parámetros:
 - Condición de búsqueda, se correspondería con el WHERE.
 - Campos que se quiere que devuelva la consulta
 - Una función, que se ejecutará cuando se realice la consulta.
- **find:** Igual que findOne pero puede devolver muchos documentos.
- **updateOne:** Es el equivalente a una operación UPDATE. Puede recibir los siguientes parámetros:
 - Condición de actualización, se correspondería con el WHERE.
 - Una función, que se ejecutará cuando se realice la consulta.
- **insertOne:** Es el equivalente a una operación INSERT, pero para un solo documento. Puede recibir los siguientes parámetros:
 - Valores a insertar
 - Una función, que se ejecutará cuando se realice la consulta.
- **insertMany:** Igual que insertOne, pero para más de un documento.

- **deleteOne:** Es el equivalente a una operación DELETE. Puede recibir los siguientes parámetros:
 - Condición de actualización, se correspondería con el WHERE.
 - Una función, que se ejecutará cuando se realice la consulta.

Estas operaciones son asíncronas, por lo que irán acompañadas de la sentencia `await` para que la ejecución espere a la finalización de la consulta

```
await userModel.findOne({username: id}).exec(function(err,user){}
```

Código 6.61: Ejemplo de acción con el Schema de usuarios.

6.4.2.3. Manejo de peticiones en la capa Back-end

No será posible provocar acciones sobre los datos definidos por los schemas sin un servidor que escuche las peticiones que llegan desde la capa Front-end. Así que el primer paso para manejar las peticiones, será crear un proceso que se mantenga a la escucha. Para tal tarea se utiliza `express`, permitiéndonos crear un servicio de escucha en el puerto 8080:

```
const app = express(); //Creamos la instancia de express

const PORT = process.env.PORT || 8080;
app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}.`);
});
```

Código 6.62: Escucha de peticiones

Este servicio de `express` no atenderá peticiones a menos que le indiquemos que peticiones y donde atenderlas. Para ello nos proporciona los métodos correspondientes a los métodos HTTP. Estos métodos de `express` reciben como parámetros, la URI donde se atenderá la petición y un callback que manejará la petición. `Express` nos proporciona los siguientes métodos:

- **get:** Solo aceptará peticiones GET en la ruta indicada.
- **post:** Solo aceptará peticiones POST en la ruta indicada.
- **put:** Solo aceptará peticiones PUT en la ruta indicada.
- **delete:** Solo aceptará peticiones DELETE en la ruta indicada.

La función que reciben como parámetro tendrá como argumentos:

- **req:** Es el objeto que representa a la petición y por tanto contendrá toda la información de esta.
- **res:** Es el objeto que representa a la respuesta que dará el servidor a la petición y nos permitirá entre otros establecer el contenido de esta y enviarla.

Un ejemplo de la escucha de una petición se muestra en el siguiente código:

```
app.post("/signup", (req,res) => {  
  if(req.header('Content-Type') === 'application/json'){  
    dbController.registerUser(req,res);  
  }  
  
})
```

Código 6.63: Escucha de petición POST en la URI /signup.

Las peticiones entrantes que sean aceptadas por alguno de esos métodos serán gestionadas todas de la misma manera:

1. Si la petición contiene cuerpo, se comprueba que es un JSON.
2. Se delega el manejo de la petición a otra función controladora
3. En la función controladora
 - a. Si la operación deseada requiere autorización, se comprueba token, más adelante se detallará este proceso.
 - b. Se validan que los datos necesarios para la consulta son correctos, si no son correctos no se realiza la consulta y se envía una respuesta con código de error. De esta manera la validación se da tanto en la capa Front-end como en la Back-end, evitando aquellas peticiones que no fueran generadas por una acción en la interfaz.
 - c. Si son válidos se lanza la consulta a base de datos
 - i. La consulta ejecutará el callback pasado como argumento
 - ii. Dentro del callback, se comprueban si ha habido errores en la consulta
 1. Si los hay se envía un código de error en la respuesta
 2. Si no los hay, se envía una respuesta con código de éxito o se realizan más consultas si se requirieran que siguen el mismo procedimiento.

Un ejemplo ilustrativo de este procedimiento en el código lo podemos ver con la petición para modificar la contraseña:

```
app.post('/password', (req,res) =>{  
  if(req.header('Content-Type') === 'application/json'){  
    dbController.editPassword(req,res);  
  }  
})
```

Código 6.64: Escucha de petición POST en la URI /password.


```

async function editPassword(req,res){
  if(checkJwtToken(req.header('Authorization'))){
    var id = req.header('User');
    var password = req.body.password
    if(id && password){
      await userModel.findOne({username: id}).exec(function(err,user){
        if(err){
          console.log(err)

          sendResponse(res,'500' , err)
        }else{
          var u = user;
          user.changePassword(password,function(response){
            if(response === 'success'){
              const token = jwt.sign({username: u.username, role: u.role},SIGN);
              sendResponse(res,'200',{token: token, msg:'Contraseña modificada c
on éxito'})
            }else{
              sendResponse(res,'500' , 'Erro al modificar la contraseña')
            }
          })
        }
      })
    }
  })
}

}else{

  sendResponse(res,'400','Petición incorrecta')

}
}else{
  console.log('UNAUTHORIZED');
  sendResponse(res,'401','No estas autorizado')
}
}
}

```

Código 6.65: Manejo de petición POST en la URI /password.

Observando el código anterior, podemos darnos cuenta de que las respuestas se envían todas de una misma función, llamada **sendResponse()**. Esta función centraliza el envío de las respuestas. Recibe como parámetros:

- El objeto de la respuesta
- El código HTTP de la respuesta
- El mensaje de la respuesta

Se encargará de construir la respuesta a partir de los parámetros y de enviarla al emisor de la petición.

```
function sendResponse(res,code,msg){
  res.status(code) // server errors
  res.setHeader('Content-type','application/json');
  res.send({msg: msg});
}
```

Código 6.66: Envío de respuesta

6.4.2.4. Seguridad

Se establecen los siguientes mecanismos de seguridad a nivel de Backend:

1. Encriptación
2. Autenticación
3. Autorización
4. Preservación del espacio almacenado

A continuación, se detalla cada uno por separado

6.4.2.4.1. Encriptación

Los usuarios registrados en el sistema accederán a este a través de un nombre de usuario y una contraseña. Si la base de datos del sistema se viera comprometida ante un ataque, las credenciales de los usuarios podrían quedar expuestas y por lo tanto acceder a ellas. Es por esta razón por la que se ha decidido encriptar las contraseñas de los usuarios.

Unas de las ventajas del uso de schemas, es la posibilidad de usar funciones *middleware* que se van a ejecutar previamente a una operación en la base de datos. Esto nos va a servir para encriptar la contraseña del usuario, ya que cuando se vaya a guardar un nuevo usuario o modificar uno existente se ejecutará previamente esta función. La función encargada de esto será la función **pre()**, recibiendo como parámetros la acción que provocará que se ejecute antes este middleware y un callback que contendrá la lógica a realizar.

Para encriptar las contraseñas se hará uso de un paquete externo llamado “*bcryptjs*”. El callback para encriptar la contraseña realizará lo siguiente

1. Se comprueba si no existía contraseña o si se ha modificado una existente
2. Si es así, se genera un string aleatorio con *bcrypt.genSalt*. Esto se hace para que el algoritmo de hash sea menos predecible
3. Si no hay errores al generarlo, se generará una contraseña encriptada con un algoritmo de hash a partir de la contraseña del usuario
4. Por último, se almacena la contraseña encriptada.

```

userSchema.pre("save", function(next){
  const user = this;
  if(this.isModified("password") || this.isNew){
    bcrypt.genSalt(10, function(saltError,salt){
      if(saltError){
        return next(saltError);
      }else{
        bcrypt.hash(user.password,salt,function(hashError,hash){
          if(hashError){
            return next(hashError);
          }

          user.password = hash;
          next();
        })
      }
    })
  }else{
    return next();
  }
}

```

Código 6.67: Middleware para encriptar la contraseña

6.4.2.4.2. Autenticación

La autenticación tendrá lugar cuando un usuario intente hacer login en el sistema. En primer lugar, la capa Back-end comprobará que exista algún usuario con ese nombre de usuario almacenado en base de datos si lo es comprobará su contraseña con la almacenada. Esto sería trivial si las contraseñas no estuvieran encriptadas, pero no es el caso.

Otra de las ventajas del uso de schemas es la posibilidad de crear métodos propios de un schema determinado, es decir, que se ejecutarán schema. Para comprobar que la contraseña del usuario coincide con la almacenada se creará uno de estos métodos. Este método comparará la contraseña con la encriptada mediante el paquete bcrypt, que aplicará una función hash entre ambas contraseñas. Además, ejecutará un callback que interpretará la respuesta.

```

userSchema.methods.checkPassword = function(password,callback){
  bcrypt.compare(password, this.password, function(error,success){
    if(error){
      return callback(error);
    }else{
      return callback(success);
    }
  })
}

```

Código 6.68: Autenticación

6.4.2.4.3. Autorización

Además de la autenticación, el usuario necesita tener autorización para realizar determinadas acciones en el sistema. Esta autorización es la posesión de un Token, representado por una larga cadena de String. Este token será generado y enviado en el momento del login y cada vez que la capa Front-end realice una petición viajará en la cabecera “*Authorization*” de la petición.

La generación del token tiene lugar en la función que se encarga de manejar la petición de login después de autenticar al usuario, concretamente en las siguientes líneas de código:

```
u.checkPassword(user.password,function(isMatch){
    if(isMatch){
        const token = jwt.sign({username: u.username, role: u.role},SIGN); //ge
neramos el token
        sendResponse(res, '200', {token: token, user: u.username}, )

    }else{
        sendResponse(res,'400', 'Contraseña incorrecta')
    }
}
```

Código 6.69: Generación del token

La generación de token se realiza con el uso de un paquete externo llamado “*jsonwebtoken*”, a partir de unos valores y una firma. A partir de un algoritmo de hashing firmará unos valores con la cadena de String de la firma, generando un token asociado directamente a un usuario, ya que se utiliza como uno de sus valores en la firma en nombre de usuario, el cual es único.

Cuando una petición requiera de autenticación, se extraerá el token de la cabecera y se llamará a una función que mediante el uso del paquete externo para los tokens verificará que es un token válido.

```
function checkJwtToken(auth){
    const token = auth;

    return jwt.verify(token, SIGN,function(err,user){
        if(err){
            return false;
        }
        return true;
    })
}
```

Código 6.70: Generación del token

6.4.2.4.4. Preservación del espacio almacenado

Los usuarios tendrán un límite de espacio para almacenar datos, en una perspectiva del que sistema pudiera crecer, en una primera instancia tendrán un total de 100MB disponibles. Es por esta razón la que se comprueba el espacio que tiene disponible antes de una inserción.

La función que comprueba el espacio, primero obtiene cuantos sonidos tiene almacenados el usuario pasado como parámetro y después calcula el tamaño mediante la función *calculateObjectSize* del paquete *bson* ya que los documentos obtenidos en la consulta son datos de tipo BSON. Esta función nos devuelve el tamaño en bytes.

```
async function checkSpace(id){
  var response
  await statesModels.stateModel.find({userID: id}, function(err, docs){
    if(err){
      console.error(err)
    }else{
      response = bson.calculateObjectSize(docs) //Devuelve el tamaño en bytes
    }
  });

  return response
}
```

Código 6.71: Comprobación de espacio

Capítulo 7: Conclusiones y vías futuras

Ya conocidos todos los aspectos que han comprendido el desarrollo de este trabajo se ha llegado a las siguientes conclusiones:

1. JavaScript es una tecnología muy versátil con la que podemos conseguir software de todo tipo.
2. La web Audio API proporciona soluciones bastante adecuadas para la generación y tratamiento de los sonidos en la web, en la que el programador no tiene porque entrar en detalles matemáticos y físicos relativos a las señales generadas.
3. Pese a las ventajas de la web Audio API, está esta limitada a los recursos del navegador, por lo que no es posible ofrecer un software de síntesis de altas prestaciones ya que aumentaría la latencia en la aplicación.
4. ReactJS proporciona muchas facilidades a la hora de desarrollar una interfaz. Sin embargo, a medida que el tamaño aumenta, la interfaz se vuelve difícil de mantener.
5. La generación de gráficos en la web no es una tarea trivial, ya que requiere de esfuerzo, tiempo y dedicación.
6. La síntesis digital en la nube es un campo aún en evolución, que requiere mejoras para igualar la calidad de los sintetizadores analógicos.
7. NodeJS proporciona bastantes ventajas al desarrollador a la hora de implementar un sistema. Es altamente recomendable para desarrollar aplicaciones web ya que ofrece soluciones de todo tipo que ahorran tiempo en las tareas de implementación.

Conocidas las conclusiones a las que se han llegado tras el desarrollo del trabajo, la aplicación web desarrollada tendrá las siguientes vías futuras:

1. Sería interesante que el sistema se desplegara y se le asignara un nombre de dominio.
2. El sistema en un futuro podría incluir distintos tipos de síntesis, más efectos y más osciladores.
3. Mejora de la interfaz.
4. Ya que se ha desarrollado una aplicación web, sería también interesante, crear una comunidad, mediante la existencia de chats o algún mecanismo parecido en los que los usuarios pudieran compartir experiencias, sonidos creados, piezas musicales etc..
5. Integrar la aplicación web en los DAWs existentes.
6. Crear roles de administrador y diseñadores de sonidos.
7. Con el fin de obtener beneficios, crear una tienda en la que los usuarios pudieran adquirir paquetes con sonidos ya creados.

Capítulo 8: Bibliografía final

- [1] IFPI, «Informe IFPI,» 23 3 2021. Available: <https://www.promusicae.es/estaticos/view/24-informes-ifpi>.
- [2] Wikipedia, «Sintetizador analógico,» 11 1 2019. Available: https://es.wikipedia.org/wiki/Sintetizador_anal%C3%B3gico.
- [3] W. C. Pirkle, Designing Software Synthesizer Plug-Ins in C++: For RackAFX, VST3, and Audio Units, Waltham: Focal Press , 2015.
- [4] STRONGSTYLEZ, «Tipos de síntesis musical,» 26 6 2017. Available: <https://strongstylez.wordpress.com/2017/06/26/tipos-de-sintesis-musical/>.
- [5] J. F. Huambachano, «¿Qué es Scrum?,» 25 9 2017. Available: <https://www.scrum.org/resources/blog/que-es-scrum>.
- [6] M. L. R. Almendros, Diapositivas asignatura Metodologías de Desarrollo Ágil, Granada: UGR, 2021.
- [7] Wikipedia, «Arquitectura cliente-servidor,» 03 06 2021. Available: <https://es.wikipedia.org/wiki/Cliente-servidor>.
- [8] Microsoft, «Arquitectura n-capas,» 10 06 2021. Available: <https://docs.microsoft.com/es-es/azure>.
- [9] MongoDB Inc, «MongoDB,» 24 06 2021. Available: <https://www.mongodb.com/>.
- [10] R. H. R. J. J. V. Erich Gamma, Design Patterns: Elements of Reusable Object-Oriented Software (GoF), United States: Addison-Wesley, 1994.
- [11] J. M. R. Moncayo, «¿Que es REST?,» 17 05 2018. Available: <https://openwebinars.net/blog/que-es-rest-conoce-su-potencia/>. [Último acceso: 2 03 2021].
- [12] MDN contributors, «Acerca de JavaScript,» 20 06 2021. Available: https://developer.mozilla.org/es/docs/Web/JavaScript/About_JavaScript.
- [13] Wikipedia, «HTML,» 21 06 18. [En línea]. Available: <https://es.wikipedia.org/wiki/HTML>.
- [14] MDN contributors, «CSS,» 4 7 2021. Available: <https://developer.mozilla.org/es/docs/Web/CSS>.
- [15] OpenJS Foundation, «Node.js,» OpenJS Foundation, 23 06 2021.
- [16] Facebook Open Source, «React,» 23 06 2021. Available: <https://es.reactjs.org/>.
- [17] StrongLoop Inc, «Express,» 20 05 2021. Available: <https://expressjs.com/es/>.
- [18] Mongoose, «Mongoose,» 20 06 2021. Available: <https://mongoosejs.com/>.
- [19] C. d. MDN, «Basic concepts behind Web Audio API,» 06 06 2021. Available: https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API/Basic_concepts_behind_Web_Audio_API.
- [20] MDN contributors, «AudioContext,» 11 6 2021. Available: <https://developer.mozilla.org/en-US/docs/Web/API/AudioContext>.
- [21] MDN contributors, «OscillatorNode,» 15 6 2021. Available: <https://developer.mozilla.org/en-US/docs/Web/API/OscillatorNode>.
- [22] MDN contributors, «<https://developer.mozilla.org/en-US/docs/Web/API/GainNode>,» 22 6 2021. Available: <https://developer.mozilla.org/en-US/docs/Web/API/GainNode>.
- [23] MDN contributors, «StereoPannerNode,» 16 06 2021.
- [24] B. Smus, Web Audio API, O'Reilly Media, 2013.
- [25] W3C Working Draft, «Web MIDI API,» 15 03 2015. Available:

<https://www.w3.org/TR/webmidi/>.

- [26] A. León, «Web Audio Series, Parte 2: Diseño de efectos de distorsión usando Javascript y la API de Web Audio,» 14 12 2017. Available: <https://medium.com/@alexanderleon/web-audio-series-part-2-designing-distortion-using-javascript-and-the-web-audio-api-446301565541>.
- [27] Wikipedia, «Node,» 25 05 2021. Available: <https://es.wikipedia.org/wiki/Node.js>.
- [28] P. Anglea, «Getting Started With The Web MIDI API,» 19 03 2018. Available: <https://www.smashingmagazine.com/2018/03/web-midi-api/>.
- [29] Wikipedia, «Git,» 22 6 2021.. Available: <https://es.wikipedia.org/wiki/Git>.
- [30] J. Zafra, Ingeniería de sonido: conceptos, fundamentos y casos prácticos, RA-MA Editorial.
- [31] SERUM, «SERUM,» 20 02 2021. Available: <https://xferrecords.com/products/serum>.
- [32] Midi city, «Midi city,» 23 02 2021. Available: <https://midi.city/>.
- [33] MDN contributors, «MDN Web Docs,» 22 4 2021. Available: <https://developer.mozilla.org/es/>.