Escola d'Enginyeria de Telecomunicació i Aeroespacial de Castelldefels

UNIVERSITAT POLITÈCNICA DE CATALUNYA

# TRABAJO FINAL DE GRADO

**Título:** Implementación de un reproductor jukebox o fonola con conexión web

**Autor:** Víctor M. Ramírez Morón

**Director:** Juan Hernández Serrano

**Fecha:** 6 de mayo de 2015

## Resumen

Un jukebox, también conocida en castellano como fonola o rockola, es un dispositivo parcialmente automatizado que reproduce música. Durante décadas estos aparatos aportaron diversión y entretenimiento a multitud de personas, que podían sentirse partícipes de la música que se escuchaba en sus locales preferidos.

El objetivo de este proyecto es volver a permitir la funcionalidad del jukebox con una tecnología actual. Por tanto, se pretende diseñar un jukebox electrónico más acorde a nuestros días, integrado con el equipo de sonido del local y al que los clientes se conectan por la red mediante su smartphone o tablet para participar de la música que sonará en el local.

Sin embargo, la interacción con el jukebox a través de la red presenta nuevos retos que deben ser abordados. En las jukebox clásicas para poder añadir una canción a la lista de reproducción, el usuario introducía monedas en la jukebox; esto venía a ser una especie de control de acceso: sólo los usuarios que pagaban podían poner canciones. En nuestro caso se busca otra solución diferente a un pago directo, ya que podría resultar poco atractivo para el usuario, por ello hemos optado por un sistema basado en tickets, con un ticket el usuario tiene derecho a insertar una canción en el sistema. La manera de obtención de estos tickets pasa por un sistema centralizado en el que un administrador subministrará esos códigos o tickets a cambio de una consumición o un pequeño pago.

**Title:** Implementación de un reproductor jukebox o fonola con conexión web

**Author:** Víctor M. Ramírez Morón

**Director:** Juan Hernández Serrano

**Date:** May 6th 2015

## Overview

A jukebox is a partially automated device that plays music. For decades these devices have entertained many people, which could feel part of the music played at their favourite places.

The objective of this project is to re-enable the functionality of a jukebox making use of current technology. Therefore, the main objective is to design an electronic jukebox perfectly integrated with the premise's sound system and that could be accessed through the network in order to take part of the music played.

However, with the interaction with the jukebox through the network, new challenges arise. In the classic jukebox, in order to add a song to the playlist, users introduced coins in the jukebox; which can be understood as a kind of access control: only users who had payed could put songs. In our case, we want to find an alternative solution to coins and we have opted for a ticket-based system; that is to say that a user with a proper ticket, actually a code, is able to a song to the sound system. The tickets are issued by a centralized server and should be later provided, e.g. in exchange for a drink or a small payment.

# ÍNDICE


**INTRODUCCIÓN ..... 1**

**1 CAPÍTULO 1. DEL JUKEBOX AL NUKEBOX ..... 3**

**1.1 Historia del Jukebox ..... 3**

**1.2 Nukebox ..... 5**

**2 CAPÍTULO 2. ARQUITECTURA ..... 8**

**2.1 Hardware ..... 8**

**2.2 Software ..... 8**

**2.3 Esquema general ..... 11**

**3 CAPÍTULO 3. IMPLEMENTACIÓN DEL BACKEND ..... 12**

**3.1 Diseño de la Base de Datos ..... 12**

**3.2 Sistema de Tickets ..... 14**
3.2.1 Control de Acceso ..... 14
3.2.2 Login ..... 15
3.2.2.1 JSON Web Tokens ..... 15
3.2.3 Análisis de seguridad ..... 17

**3.3 Interfaz de comunicación con el Backend ..... 20**

**3.4 Funciones del administrador ..... 21**
3.4.1 Generar ticket ..... 22
3.4.2 Control del reproductor ..... 23

**3.5 Local ..... 24**
3.5.1 Lista de canciones disponibles ..... 24
3.5.2 Player ..... 25
3.5.3 Playlist ..... 25
3.5.4 Buscar canción ..... 28
3.5.5 Comprobación de ticket ..... 28

**3.6 Spotify ..... 30**
3.6.1 Integración y configuración ..... 33
3.6.2 Funciones ..... 34
3.6.3 Limitaciones ..... 36

**4 CAPÍTULO 4. IMPLEMENTACIÓN DEL FRONTEND ..... 37**

**4.1 Interfaz de usuario ..... 37**

**4.2 Apariencia de la interfaz de usuario ..... 38**

**4.3 Funcionalidades de usuario ..... 40**
4.3.1 Canciones disponibles ..... 40

4.3.2 Búsqueda ..... 40
4.3.3 Introducción de ticket ..... 42
4.3.4 Lista de reproducción ..... 43

**4.4 Interfaz de administrador ..... 44**
4.4.1 Login ..... 44
4.4.2 Página de administrador ..... 45

**5 CONCLUSIONES Y LÍNEAS FUTURAS ..... 47**

**6 BIBLIOGRAFÍA ..... 48**

# ÍNDICE DE FIGURAS

Fig. 0.1 Jukebox original ..... 1
Fig. 1.1 Imagen promocional de Wurlitzer 1100 ..... 3
Fig. 1.2 Primer modelo jukebox de Wurlitzer de 1933 ..... 4
Fig. 1.3 Jukebox QR ..... 6
Fig. 1.4 Esquema de funcionamiento ..... 6
Fig. 1.5 Nukebox desde smartphone y desde navegador de PC ..... 7
Fig. 2.1 Logo oficial de Apache Tomcat ..... 8
Fig. 2.2 Logo de Jersey ..... 8
Fig. 2.3 Esquema de intercambio de datos JSON ..... 9
Fig. 2.4 Logo de JSON ..... 9
Fig. 2.5 Logo de Hibernate ..... 9
Fig. 2.6 MySQL y PHPMyAdmin ..... 10
Fig. 2.7 Logo XAMPP ..... 10
Fig. 2.8 HTML, CSS3 y jQuery ..... 10
Fig. 2.9 Apache Maven ..... 11
Fig. 2.10 Esquema software - hardware ..... 11
Fig. 3.1 Configuración Hibernate ..... 12
Fig. 3.2 Tabla Clientip ..... 13
Fig. 3.3 Tabla Ticket ..... 13
Fig. 3.4 Tabla Song ..... 14
Fig. 3.5 Estructura de un JWT ..... 15
Fig. 3.6 Cabecera JWT ..... 16
Fig. 3.7 Payload JWT ..... 16
Fig. 3.8 Resultado final de JWT ..... 16
Fig. 3.9 Probabilidad de ataque exitoso ..... 19
Fig. 3.10 Gráfica ideal de ataque exitoso con parte visible ..... 19
Fig. 3.11 Esquema en árbol de interfaz e implementaciones ..... 20
Fig. 3.12 Declaración de las clases de implementación ..... 21
Fig. 3.13 Tabla de mapeo binario a carácter ..... 23
Fig. 3.14 Declaración de objeto en java ..... 23
Fig. 3.15 Ejemplo de Singleton ..... 24
Fig. 3.16 Método *lastmodified* ..... 24
Fig. 3.17 Listas de canciones auxiliar y añadidas por usuarios ..... 26
Fig. 3.18 Posibles casos de lista de reproducción ..... 27
Fig. 3.19 Método obtener playlist ..... 27
Fig. 3.20 Variables de respuesta en BBDD ..... 29
Fig. 3.21 Diagrama de introducción de canción en cola ..... 30
Fig. 3.22 Logo Spotify ..... 31
Fig. 3.23 Pantalla principal de Spotify Developers ..... 32
Fig. 3.24 Creación de nueva aplicación ..... 32
Fig. 3.25 Obtención de Client ID y Client Secret ..... 33
Fig. 3.26 Referencia de librería Spotify ..... 34
Fig. 3.27 Creación de objeto Api ..... 34
Fig. 3.28 Parámetros necesarios ..... 34
Fig. 3.29 Obtención de *token* actualizado ..... 35
Fig. 3.30 Esquema lógico de acceso y credenciales ..... 36
Fig. 4.1 Partes del html ..... 37
Fig. 4.2 Función append de jQuery ..... 38

Fig. 4.3 Página principal de Google design ..... 38
Fig. 4.4 Ejemplo de media query CSS3 ..... 39
Fig. 4.5 Página principal ..... 40
Fig. 4.6 Búsqueda ..... 40
Fig. 4.7 Búsqueda II ..... 41
Fig. 4.8 Búsqueda sin coincidencias ..... 41
Fig. 4.9 Introducción de ticket..... 42
Fig. 4.10 Vista móvil y pc de la lista de reproducción ..... 43
Fig. 4.11 Obtención de playlist cada segundo ..... 43
Fig. 4.12 Pantalla de login ..... 44
Fig. 4.13 Guardado y obtención de datos en la sesión cookie ..... 44
Fig. 4.14 Página principal del administrador..... 45
Fig. 4.15 Ticket generado ..... 46

# INTRODUCCIÓN

Un jukebox, también conocido en castellano como fonola o rockola, es un dispositivo parcialmente automatizado que reproduce música. Durante décadas estos aparatos aportaron diversión y entretenimiento a multitud de personas, que podían sentirse partícipes de la música que se escuchaba en sus locales preferidos. Su época dorada se encuentra entre 1940-1970 pero su uso ha ido perdiendo fuerza; siendo, a día de hoy, prácticamente imposible encontrarlos si no es como objetos de decoración. La llegada del CD (desarrollado por Sony y Philips en 1979) y otros formatos digitales fueron dejando de lado este elemento.

**Fig. 0.1 Jukebox original**

El objetivo de este proyecto es volver a permitir esa funcionalidad con una tecnología renovada. Por tanto, se pretende diseñar un jukebox electrónico más acorde a nuestros días, integrado con el equipo de sonido del local y al que los clientes se conectan por la red mediante su smartphone o tablet para participar de la música que sonará en el local.

Sin embargo, la interacción con el jukebox a través de la red presenta nuevos retos que deben ser abordados. En las jukebox clásicas para poder añadir una canción a la lista de reproducción, el usuario introducía monedas en la jukebox; esto venía a ser una especie de control de acceso: sólo los usuarios que pagaban podían poner canciones. En nuestro caso se busca otra solución diferente a un pago directo, ya que podría resultar poco atractivo para el usuario, por ello hemos optado por un sistema basado en tickets, con un ticket el usuario tiene derecho a insertar una canción en el sistema. La manera de obtención de estos tickets pasa por un sistema centralizado en el que un administrador subministrará esos códigos o tickets a cambio de una consumición o un pequeño pago.

El resto de este documento se organiza como se explica a continuación.

En el capítulo 1 se hablará brevemente de la historia de este dispositivo y a continuación se presenta a grandes rasgos qué es y cómo funciona la propuesta de jukebox renovado, por qué se ha diseñado de tal manera y cuál es su estructura.

En el capítulo 2 se hablará de la arquitectura tanto a nivel de hardware como de software, como las tecnologías utilizadas.

En el capítulo 3 se desarrollan los aspectos más técnicos sobre la implementación *backend* de jukebox, incluyendo el uso de librerías y herramientas y la seguridad del sistema.

A continuación, en el capítulo 4 se analizan los aspectos relacionados *frontend* tanto para el cliente como para el administrador y una justificación del diseño elegido.

Este TFG concluye con un capítulo de conclusiones y líneas futuras, donde se tratarán los temas que han resultado más complejos, los problemas encontrados y las posibles líneas futuras.

# 1 CAPÍTULO 1. DEL JUKEBOX AL NUKEBOX

En este primer capítulo hablaremos de cómo funcionaban los jukebox, y cuáles fueron sus inicios. En la segunda sección se explicara de manera más completa en que se basa nuestra modernización del jukebox.

## 1.1 Historia del Jukebox

Un jukebox, también conocida en castellano como fonola o rockola, es un dispositivo que reproduce música cuándo se le introducen monedas. Permite seleccionar canciones para posteriormente reproducirlas. Suelen ser de un metro y medio de altura, con la parte superior redondeada e iluminación de color en el frente y sus lados verticales. La selección de los temas se lleva a cabo a través de una botonera que, mediante una combinación, permite indicar una canción específica entre una lista de discos.

**Fig. 1.1 Imagen promocional de Wurlitzer 1100**

El origen del jukebox se halla en los viejos fonógrafos de monedas. Estos aparatos comenzaron a producirse a finales del siglo XIX. Empleaban cilindros para reproducir las grabaciones y canciones. Éstas quedaban plasmadas en la superficie exterior de los cilindros y luego se reproducían en los fonógrafos mecánicos. A partir de 1910, se fueron imponiendo los discos de gramófono, hasta que en 1929 dejaron de fabricarse los sistemas de cilindros.

La marca referente de estos dispositivos era Rudolph Wurlitzer Company más conocida como Wurlitzer, empresa fundada en Cincinnati, Ohio en los EEUU en el año 1853, por un inmigrante alemán. Actualmente pertenece a la empresa Gibson. Su primer Jukebox es el modelo 'Debutante' de 1933[4] como podemos ver en la imagen inferior.

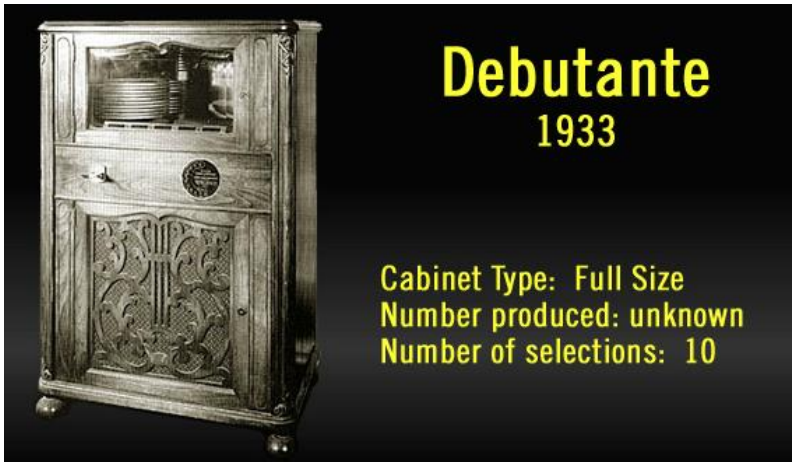


**Fig. 1.2 Primer modelo jukebox de Wurlitzer de 1933**

La palabra "jukebox" (caja de distracción) empezó a utilizarse, en Estados Unidos, en la década de los años 30 del siglo pasado. Se piensa que el término deriva de otro, "jook", que significaba "baile" y era empleado en la jerga afro-americana. Aunque las primeras máquinas a monedas surgieron en Europa, las jukeboxes americanas han sido los aparatos musicales de esas características más populares y que más predicamento han cosechado en todo el mundo.

Las décadas de los 40, 50, 60 y 70 fueron las épocas doradas de las sinfonolas, rockolas o jukeboxes en muchas regiones del mundo.

El principal defecto de los jukebox era que el mismo dispositivo en el que se insertaba el dinero y se gestionaban los discos era también el que reproducía la música. La calidad de la reproducción estaba lejos de ser óptima y además no podía integrarse en los equipos de sonido de los locales, que poco a poco iban tomando presencia.

## 1.2 Nukebox

Como hemos comentado, nuestro proyecto busca recuperar el Jukebox traído a nuestros días, para que la gente que se encuentra en un local pueda intervenir en la música que sonará. De esta forma, la música del establecimiento se adaptará mejor a los gustos o preferencias de los clientes del local. Se trata de una filosofía diferente a la impuesta en la actualidad, en que la música que se escucha está establecida por el tipo de local o por los gustos del propietario/empleado.

El nombre de Nukebox es un pequeño juego de palabras de la unión de New y Jukebox, con esto buscamos darle una mayor personalidad a nuestro proyecto.

Con ese objetivo en mente, el Nukebox ha de resolver algunos problemas que tenían los jukebox, como el que se ha comentado en el apartado anterior sobre la calidad del sonido, ya que en nuestro caso el sistema de audio será el que haya instalado en el local, o la forma de añadir canciones ya que no será necesario estar delante de la máquina para seleccionar una canción.

El Nukebox se basa en una aplicación web a la que los clientes acceden mediante un Smartphone o Tablet a través de su navegador, permitiendo acceder al sistema desde cualquier punto del local en el que se encuentren.

El administrador/camarero subministra una clave a los clientes al realizar una consumición o hacer un pago, esta clave es la que da acceso a poner en cola una canción, para todo lo demás el acceso es libre.

Para obtener una clave el camarero se conectara al servidor mediante un navegador y hará una petición para obtener dicha clave. Esta clave tendrá una vigencia de 30 minutos desde el momento que se genera. Pasado este tiempo la clave queda invalidada.

Este servidor es al mismo tiempo el propio reproductor que estará conectado al sistema de sonido del establecimiento, además de ser el que se encarga de gestionar las listas y el tratamiento y comprobación de claves.

El cliente para acceder al sistema lo que deberá hacer es conectarse a la red WIFI del establecimiento, que será una red abierta y a continuación introducirá la URL en navegador del Smartphone o bien para facilitar esta operación podrá escanear un código QR que le redireccionará directamente. Los códigos QR permiten añadir imágenes o colores para hacerlos más atractivos[3]. Estos códigos QR podrían estar distribuidos por el establecimiento como por ejemplo en las mesas. Podría ser una imagen como la que vemos a continuación:

**Fig. 1.3 Jukebox QR**

En la imagen siguiente podemos ver a modo de esquema como funciona Nukebox, y los elementos que intervienen:

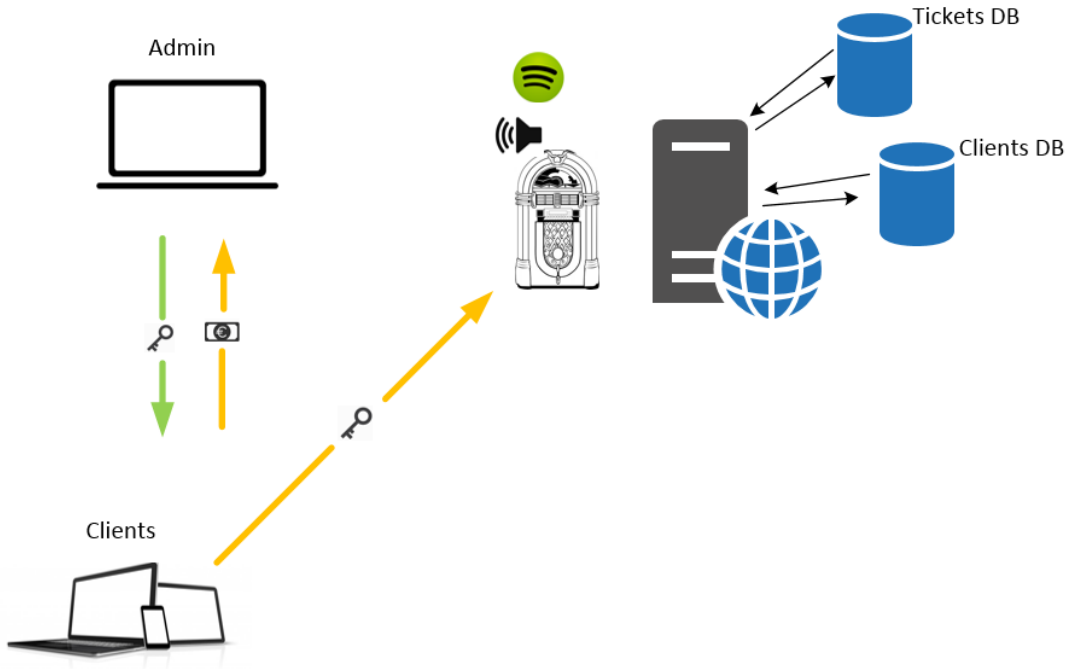


**Fig. 1.4 Esquema de funcionamiento**

Para una mejor experiencia del usuario, la página web se adapta al tamaño de la pantalla, para que sea cómodo para el usuario y el caso de utilizarlo en un smartphone la apariencia será como la de una aplicación, a esta técnica se le conoce como web *responsive*.

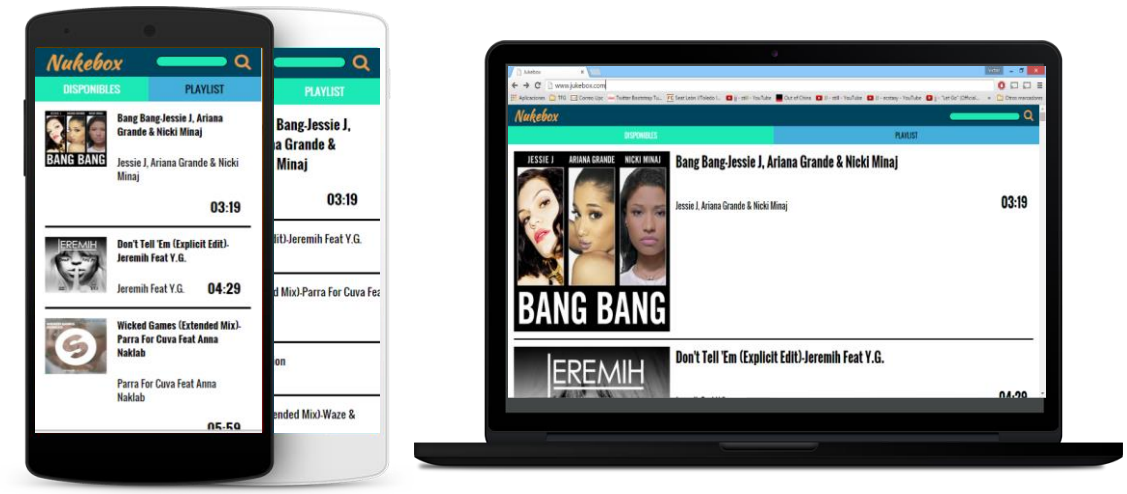


**Fig. 1.5 Nukebox desde smartphone y desde navegador de PC**

El motivo por el que se ha optado por una aplicación web y no una aplicación nativa para Android o iOS es que por una parte esta unificado y con la misma aplicación podemos acceder desde cualquier PC o Smartphone y sólo hay un desarrollo de la aplicación.

El segundo motivo es que garantizamos un mayor uso evitando la descarga de una aplicación ya que algunos clientes posiblemente para poner en cola una sola canción no descarguen la aplicación o incluso puede que dentro del establecimiento la señal de la cobertura no sea óptima.

Una vez que el usuario ha introducido la URL (o capturado el código QR) podrá ver las canciones disponibles (en el caso ser la implementación en local), hacer una búsqueda, ver la lista de reproducción actual y una vez que el cliente decide la canción para ponerla en cola deberá introducir el código anteriormente mencionado. En el caso de ser un código correcto esta canción será reproducida posteriormente. El mismo usuario podrá ver mediante la lista de reproducción el número de canciones se reproducirán antes que la suya, ya que estarán resaltadas en otro color.

Se ha diseñado de tal forma que en el caso de que no se realicen inserciones, el Nukebox reproducirá de manera autónoma canciones disponibles en el sistema y creando una lista de reproducción auxiliar, evitando así que se pare la música.

# 2 CAPÍTULO 2. ARQUITECTURA

En este capítulo veremos de manera más completa qué partes componen el Nukebox y qué hace cada una de ellas, las herramientas que hemos utilizado para llevarlo a cabo y el porqué de esta elección.

## *2.1 Hardware*

Los elementos de hardware que componen el Nukebox son básicamente un PC tanto con GNU/Linux, Windows u OSX ya que el software está disponible para las diferentes plataformas. Este actuará tanto como servidor web, servidor de BBDD, y reproductor multimedia, otro elemento necesario es el equipo de música del establecimiento. También será necesaria una conexión a internet para que el servidor pueda ejecutarse correctamente. Opcionalmente el encargado de subministrar puede tener una Tablet para un acceso más cómodo, o en su defecto un pc. En el apartado 2.3, podemos ver como intervienen estas partes en el total del sistema (ver Fig. 2.10).

## *2.2 Software*

En este apartado vamos a tratar los aspectos relacionados con el software del que se ha hecho uso en nuestro proyecto y una justificación de la elección de estas herramientas.

**Fig. 2.1 Logo oficial de Apache Tomcat**

A nivel lógico lo que encontramos es un servidor web Tomcat[6] que es el que atiende las peticiones de los usuarios, se ha utilizado un servidor Tomcat que es un contenedor de *servlets* ya que todo el lenguaje interno del *backend* está en leguaje Java.

**Fig. 2.2 Logo de Jersey**

Se ha diseñado como un sistema RESTful[8], es decir que sigue la arquitectura REST (Representational State Transfer), a modo de resumen diremos que es sigue un diseño Cliente/Servidor sin estado, es decir las peticiones HTTP no necesitan ninguna información anterior. Las operaciones más importantes son POST, GET, PUT y DELETE. Los recursos son accesibles a través de URIs (host:port//path/path) y se utilizan hipermedios para el intercambio de información (ver Fig. 2.3), como XML o en nuestro caso JSON. Para esto se ha utilizado la herramienta Jersey[7] (ver Fig. 2.2).

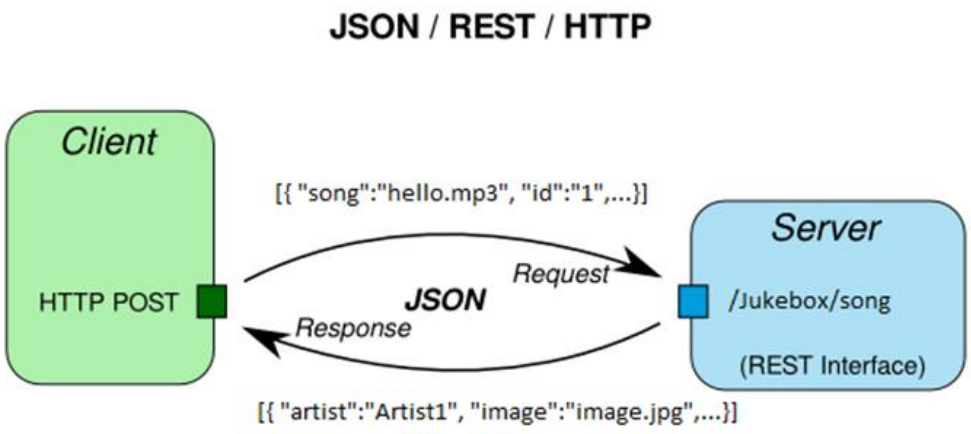


**Fig. 2.3 Esquema de intercambio de datos JSON**

**Fig. 2.4 Logo de JSON**

Para la integración de JSON en el sistema hemos utilizado la librería GSON, que es la que nos permite la conversión de Objetos Java en su representación JSON.

Para la gestión de la base de datos hemos utilizado MySQL, un sistema de gestión de bases de datos relacionales multihilo y multiusuario. La herramienta PHPMyAdmin nos permite acceder y gestionar la base de datos MySQL de manera más sencilla, sin la necesidad de sentencias SQL. Para una mejor interacción de la base de datos hemos utilizado el framework Hibernate[10], que es una herramienta de mapeo objeto-relacional, que permite hacer operaciones de BBDD directamente con objetos java, como salvar o actualizar, y utilizando lo menos posible sentencias SQL.

**Fig. 2.5 Logo de Hibernate**

**Fig. 2.6 MySQL y PHPMyAdmin**

La herramienta utilizada para la creación y gestión de nuestro proyecto ha sido Maven [12] (ver Fig. 2.9) (Apache Software Fundation) que facilita la creación de un proyecto de un determinado tipo gracias a los *archetypes* que en nuestro caso, hemos utilizado el *archetype* webapp y automáticamente crea todos los directorios necesarios y los respectivos ficheros como por ejemplo web.xml. También nos permite una buena gestión de las dependencias gracias a su Project Object Model (POM) para describir el proyecto a construir y sus dependencias de otros módulos y componentes externos, a parir del archivo POM.xml

**Fig. 2.7 Logo XAMPP**

Hemos utilizado el software XAMPP[9], que integra Tomcat, mySQL, PHPMyAdmin además de otros componentes que no hemos utilizado en nuestro proyecto como Apache, PHP o Perl. Requiere una configuración mínima, permitiendo tener los componentes básicos necesarios para un servidor de una manera fácil. Pertenece a Apache Software Fundation y está disponible para Windows, OSX y Linux.

**Fig. 2.8 HTML, CSS3 y jQuery**

Las herramientas y tecnologías que hemos utilizado para la parte de cliente o *front-end* han sido HTML5 y CSS3 para la parte visual, y Javascript para poder interactuar y hacer la página dinámica. Concretamente hemos utilizado el framework jQery[11] que permite agilizar muchas tareas, interactuar con los elementos HTML y agregar la interacción con la técnica AJAX.

**Fig. 2.9 Apache Maven**

Por último para el control de versiones se ha utilizado la plataforma GitHub[14]. La versión utilizada es gratuita, en esta versión el código alojado es público y visible para cualquier usuario. También existe una versión en la que el código es privado pero con un coste. Es la herramienta más utilizada en el mundo del desarrollo de software colaborativo.

## 2.3 Esquema general

En la Fig. 2.10 podemos ver un esquema general del software y hardware que interviene en el sistema. A la izquierda de la figura encontramos usuario y administrador basados en HTML, CSS y jQuery. Éstos se conectan mediante peticiones HTTP e intercambiando datos con el servidor en JSON. El servidor, como hemos explicado, es un servidor Tomcat, que trabaja con una base de datos MySQL. Este servidor también es el encargado de la reproducción de música.

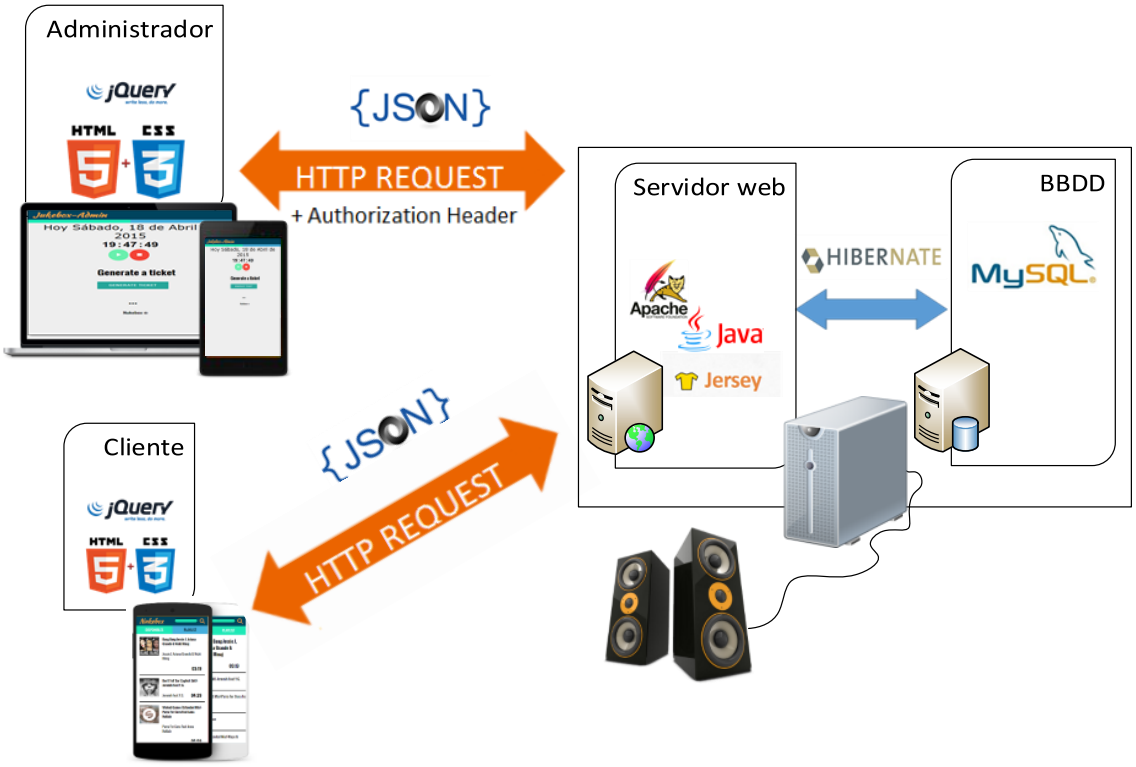


**Fig. 2.10 Esquema software - hardware**

# 3 CAPÍTULO 3. IMPLEMENTACIÓN DEL BACKEND

En este capítulo describiremos cómo hemos llevado a cabo las ideas indicadas anteriormente y qué recursos y herramientas hemos utilizado.

Primero describiremos qué compone nuestra aplicación, los objetos básicos para ello, la manera en que se ha diseñado para hacerlo 'escalable', cómo funciona la parte del administrador, y por último la parte con la que interactúa el cliente y las operaciones de las que dispone, tanto para una implementación local como una implementación en Spotify.

## *3.1 Diseño de la Base de Datos*

En este apartado veremos los objetos que utilizamos para el intercambio de datos que se corresponderán con las tablas de la base de datos, ya que Hibernate hará un mapeo de estos objetos.

Para ello hemos utilizado las anotaciones de Hibernate; en cada objeto que queremos que se realice ese mapeo debemos añadir la anotación `@Entity` y la anotación `@Id` para indicar cuál es la *primary key*, Hibernate también permite hacer las relaciones *one-to-many* y *many-to-many* de la base de datos, pero en nuestro caso no han sido necesarias las relaciones entre tablas.

La base de datos debe ser creada por el usuario, una vez creada deberemos especificar el en fichero hibernate.cfg.xml la base de datos a la que queremos conectarnos, el usuario y contraseña.

Hibernate nos permite la creación automática de las tablas en MySQL, para ello hemos creado una clase `CrearTablas.java`. De tal forma, si están las anotaciones correctas y ejecutamos el código siguiente (ver Fig. 3.1) se crearán las tablas y podremos hacer la primera inserción de prueba ejecutando el método `save` de un objeto.

```
AnnotationConfiguration config = new AnnotationConfiguration();
config.addAnnotatedClass(Ticket.class);
config.addAnnotatedClass(ClientIP.class);
config.addAnnotatedClass(Song.class);
config.configure();
new SchemaExport (config).create(true, true);
```

**Fig. 3.1 Configuración Hibernate**

Una vez hecha esta operación, las tablas que encontramos en nuestra base de datos son:

**Clientip**

Es la tabla que el sistema utiliza para tener control sobre los accesos erróneos de un usuario, intenta prevenir accesos maliciosos que intenten probar claves para acceder al sistema (ver sección3.2.1).

Contiene la IP del cliente que hace intentos erróneos, un contador de intentos, un booleano que marca si ese cliente está excluido del sistema y una marca temporal.

| Columna | Tipo |
|---|---|
| Ip | String |
| ban | booleano |
| counter | Int (Entero) |
| timestamp | String |

**Fig. 3.2 Tabla Clientip**

**Ticket**

Esta tabla contiene información de los tickets generados por el administrador, los campos son el *idticket* que es propiamente el ticket que se le subministra al usuario, una marca temporal para poder hacer comprobaciones de cuando se ha creado este ticket y si está caducado y un *flag* que nos servirá para descartar tickets ya canjeados. Esto se explicará en la sección 3.5.5.

| Columna | Tipo |
|---|---|
| idticket | String |
| date | String |
| used | booleano |

**Fig. 3.3 Tabla Ticket**

**Song**

La tabla song (ver Fig. 3.4) es la que contiene toda la información referente a las canciones, esta información servirá tanto para mostrársela al usuario, como para conocer la ruta donde encontrarlas, o saber si ha sido añadida por un usuario o pertenece a la lista de reproducción auxiliar.

| Columna | Tipo |
|---|---|
| Artist | String |
| Name | String |
| Genre | String |
| Duration | String |
| Image | String |
| Album | String |
| Id | String |

**Fig. 3.4 Tabla Song**

## *3.2 Sistema de Tickets*

Como hemos comentado en el primer capítulo nuestra aplicación se basa en un sistema de tickets que permiten insertar canciones. Estos tickets son básicamente un código que le será pedido al usuario en el momento de la elección de una canción. Por lo tanto, debemos controlar si estos códigos ya se han utilizado, han expirado o directamente no existen. En los siguientes apartados veremos cómo se ha implementado este sistema.

### 3.2.1 Control de Acceso

Una manera a de evitar que alguien que no tenga clave pueda poner una canción en cola es prevenir que pueda probar claves ilimitadamente, ya que depende del número de claves disponibles podría llegar a conseguir alguna. Por lo tanto lo que se ha implementado es un control de la introducción de claves erróneas, básicamente lo que el servidor hace es comprobar las IP de los clientes y sus intentos fallidos y cuando estos intentos llegan a un número fijado que en este caso es 10 ese cliente queda excluido del sistema durante un tiempo establecido de 10 minutos.

Para la implementación de este sistema tenemos la tabla *Clientip* (3.1). En el momento que un usuario introduce una clave errónea se introduce esa IP en la tabla con el contador a 1 y *baneado* que es un booleano tendrá valor 0, hasta el supuesto caso de el contador valer 10, que *baneado* será igual a 1 (true) y se pondrá una marca temporal para comprobar el tiempo que esta *baneado*. Con este método aunque no sea el método más seguro que se podría implementar, para la aplicación que se ha diseñado es suficiente, ya que no hay datos

personales ni información confidencial, lo máximo que podría hacer un atacante es poner una canción en cola, y la probabilidad de conseguirlo, tal y como se detalla en la sección 3.2.3, es muy baja.

### 3.2.2 Login

La creación de los tickets únicamente está reservada para el administrador o los administradores del local. Para prevenir el acceso a personas no autorizadas se ha implementado un sistema de *login*. Este *login* permite el acceso a la página del administrador.

Si el nombre y el *password* introducidos son correctos se hará una redirección directa a la página del administrador y se le devolverá al usuario un token, o clave que le permitirá hacer las peticiones posteriores.

Para la integración en nuestro proyecto de este sistema de *tokens* hemos utilizado la tecnología JWT, hemos tenido que añadir la dependencia a nuestro fichero POM.xml para poder hacer uso de esa librería.

Lo hemos implementado de tal manera que en el momento que se comprueba si el usuario y contraseña sean correctos se crea un objeto tipo `JWT` en el que añadimos los parámetros necesarios como el tiempo de expiración, que en este caso son 3 horas. Una vez generado se le enviará al usuario y en las funciones que estén reservadas para el administrador, el primer paso será comprobar si dispone de un *token* válido y en caso contrario enviar un mensaje de error. En el siguiente subapartado haremos una breve explicación de este sistema, ya que es una herramienta muy utilizada.

#### 3.2.2.1 JSON Web Tokens

JSON Web Token (JWT)[15] es un medio de representación de *claims* (o reivindicaciones) para ser transferidos entre dos partes. Estos *claims* se codifican como un objeto JSON que está firmado digitalmente utilizando *JSON Web Signature (*JWS*).* La pronunciación sugerida de JWT es la misma que la palabra Inglés *jot*.

Un JWT es fácil de identificar, básicamente son tres *strings* separados por un punto (ver Fig. 3.5).

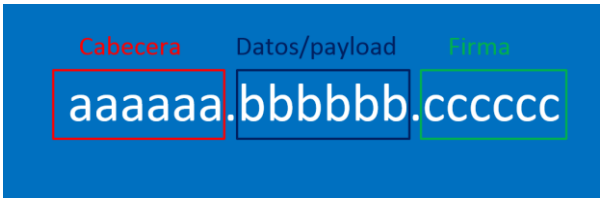


**Fig. 3.5 Estructura de un JWT**

A continuación haremos una breve explicación de las diferentes partes.

### Cabecera

Esta parte contiene dos elementos

- Declaración del tipo (JWT)
- Algoritmo de *hash* (H256 en este caso)

```
{
  "typ": "JWT",
  "alg": "HS256"
}
```

**Fig. 3.6 Cabecera JWT**

### Payload

Aquí se encuentra la información que se quiere transmitir sobre el *token*, también conocida como *JWT Claims.* Hay varios *claims* que se ofrecen, como registrados, privados o públicos.

```
{
  "iss": "jukebox",
  "exp": 1300819380,
  "name": "Victor Ramirez",
  "admin": true
}
```

**Fig. 3.7 Payload JWT**

### Firma

La tercera y última parte del JWT es la firma, esta firma se hace a partir de una función de *hash* (en castellano función resumen) de la cabecera, payload y la clave o *secret.* En este momento tenemos como resultado el *token* que se le subministrará al usuario, una vez unidos todos los componentes. (Ver Fig. 3.8).

```
var encodedString = base64UrlEncode(header) + "." + base64UrlEncode(payload);

HMACSHA256(encodedString, 'secret');
```

**Fig. 3.8 Resultado final de JWT**

### 3.2.3 Análisis de seguridad

Hemos realizado un pequeño análisis sobre la seguridad del sistema y en este apartado queremos justificar algunos parámetros o valores del sistema.

Un posible ataque que podría sufrir nuestro Nukebox, es el intento de inserción de canciones sin disponer de una clave, para ello el atacante iría probando diferentes claves hasta dar con una correcta. Por lo tanto debemos comprobar la probabilidad de que se dé esta circunstancia.

A modo de resumen, recordamos que la probabilidad $p$ de que suceda un evento $S$ es igual a la razón entre el número de casos posibles o favorables $m$ de dicho evento y el número total de casos posibles $n$.

$$p = P\{S\} = \frac{m}{n}$$

El valor que nos interesa determinar es la probabilidad de que un atacante encuentre al menos un código válido después de $t$ intentos.

Lo primero que debemos hacer es determinar las variables que intervienen en nuestro sistema:

- Códigos válidos ($m$): Es el número de claves que se consideran válidas; es decir, que ya han sido creadas por el administrador y que no han expirado. Son las claves con las que busca dar el atacante.
- Códigos disponibles ($n$): Es el número de claves posibles dentro de nuestro sistema; es decir, todas las claves que se pueden generar sin haber repetidas. En nuestro caso se trata de claves de 5 caracteres y cada carácter, tal y como se detalla en la sección 3.4.1, se codifica con 6 bits, por tanto un código tiene 30 bits y $\mathrm{n} = 2^{30}$.

Sabemos que la probabilidad *p* de que un atacante dé al menos con una clave válida después de $t$ intentos se puede expresar como en (1) siendo $q$ la probabilidad de que no encuentre ninguna clave válida en los mismos intentos.

$$p = 1 - q \tag{1}$$

La probabilidad $q$ de que un atacante no encuentre ningún código válido después de $t$ intentos se puede entender si observamos que en el primer intento el atacante tiene una probabilidad de no acertar de $(n - m)/n$ ya que existen $n$ -$m$ códigos inválidos del total de $n$. Si no se hubiese acertado en el primer intento, la probabilidad de acertar en el segundo es ahora de $(n - m - 1)/(n - 1)$, ya que el atacante no repite el código anterior y por la tanto hay un caso menos. Continuando con este razonamiento hasta $t$ intentos se obtiene la fórmula en (2).

$$q = \frac{n-m}{m} \cdot \frac{n-m-1}{n-1} \dots \cdot \frac{n-m-t+1}{n-t+1} = \frac{\frac{(n-m)!}{(n-m-t)!}}{\frac{n!}{(n-t)!}} = \frac{(n-m)!\,(n-t)!}{n!\,(n-m-t)!} \quad (2)$$

Y por tanto la probabilidad $p$ de que un atacante encuentre un código válido después de *t* intentos queda como en (3)

$$p = 1 - q = \frac{(n-m)!\,(n-t)!}{n!\,(n-m-t)!} \quad (3)$$

Sin embargo, cabe la posibilidad de que haya más de un atacante en el sistema. Por lo tanto, a continuación mostramos una generalización de la forma anterior para $k$ atacantes. Vamos a asumir el peor caso, en el que los atacantes colaboran y por tanto prueban códigos diferentes. En este caso, basta con modificar el número de intentos que pasa a ser $kt$ en lugar de *t* y por tanto $q$ se obtendría como en (4), siendo la notación final la más útil para su implementación programática.

$$q = \frac{(n-m)!\,(n-kt)!}{n!\,(n-m-kt)!} = \frac{\prod_{i=m}^{m+kt-1} n-i}{\prod_{i=0}^{kt-1} n-i} = \prod_{i=0}^{kt-1} \frac{n-m-1}{n-1} = \prod_{i=0}^{kt-1} 1 - \frac{m}{n-1} \quad (4)$$

Finalmente, se puede expresar como en (5) la probabilidad p de que $k$ atacantes consigan al menos un código valido después de que cada uno haya realizado $t$ intentos.

$$p = 1 - q = 1 - \frac{(n-m)!\,(n-kt)!}{n!\,(n-m-kt)!} \quad (5)$$

Para fijar un valor de $t$ más o menos realista vamos a estimar que una persona está un promedio de entre una y tres horas en un establecimiento, tomaremos dos como un valor intermedio. El atacante como mucho puede introducir una clave cada 3 segundos, es una limitación que hemos establecido en nuestro servidor y que en cualquier caso es modificable (no deja de ser una política de sistema). Por lo tanto, si hiciese un intento cada 3 segundos ininterrumpidamente serían $\frac{3600_{s/hora}}{3_{s/intento}} = 1200\ intentos/hora$ , es decir aproximadamente $2400$ intentos en 2 horas. Es un valor bastante optimista ya que para ello el atacante debería intentar claves ininterrumpidamente.

También habría que tener en cuenta la restricción de usuarios explicado en la sección 3.2.1, que reduciría significativamente ese valor.

Para este estudio hemos utilizado la herramienta de software matemático Matlab.

Una vez realizado el *plot* de Matblab, en el caso más crítico vemos que con 50 atacantes con 2400 ataques cada uno la probabilidad de que al menos uno consiga introducir una clave valida es de aproximadamente $3.33 \cdot 10^{-3}$ (0.003347). Una cifra más que suficiente para este proyecto, teniendo en cuenta que el efecto de conseguir una clave correcta se reduce a introducir una canción en la lista de reproducción.

Podemos observar que la curva realizada por Matlab es prácticamente recta, pero realmente la probabilidad no aumenta linealmente sino exponencialmente, como se puede ver en la Fig. 3.10. Esto se debe a que la escala en la que estamos trabajando no nos permite poder visualizar la gráfica correctamente. Realmente estamos viendo la parte que se comporta como lineal, como se puede ver en la Fig. 3.10 marcado en rojo.

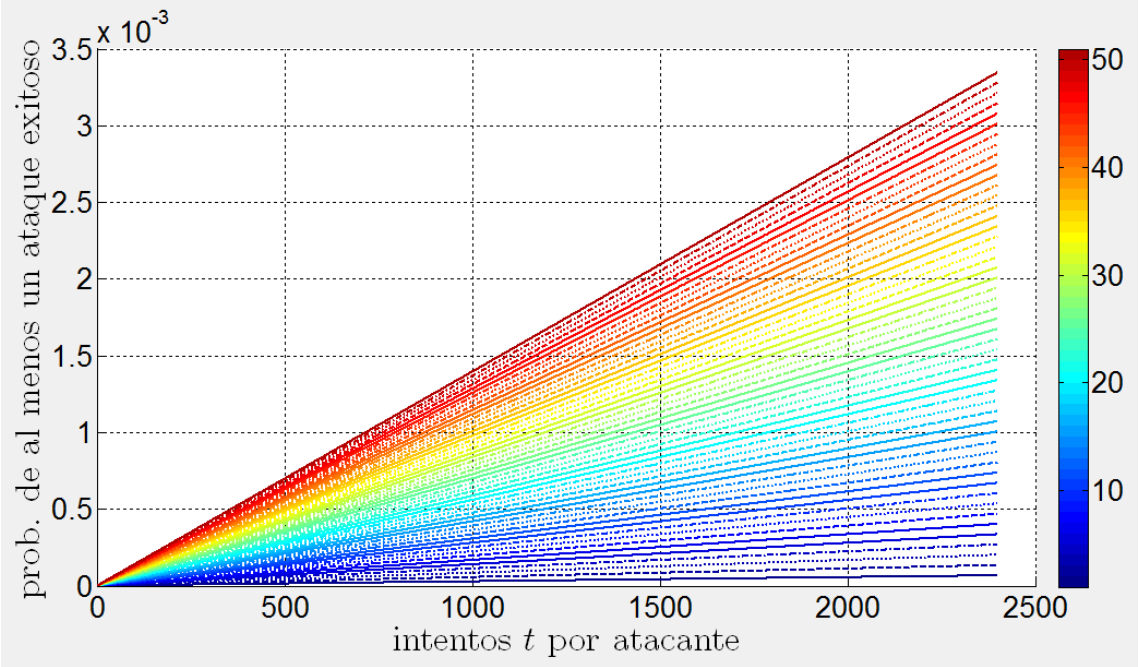


**Fig. 3.9 Probabilidad de ataque exitoso**

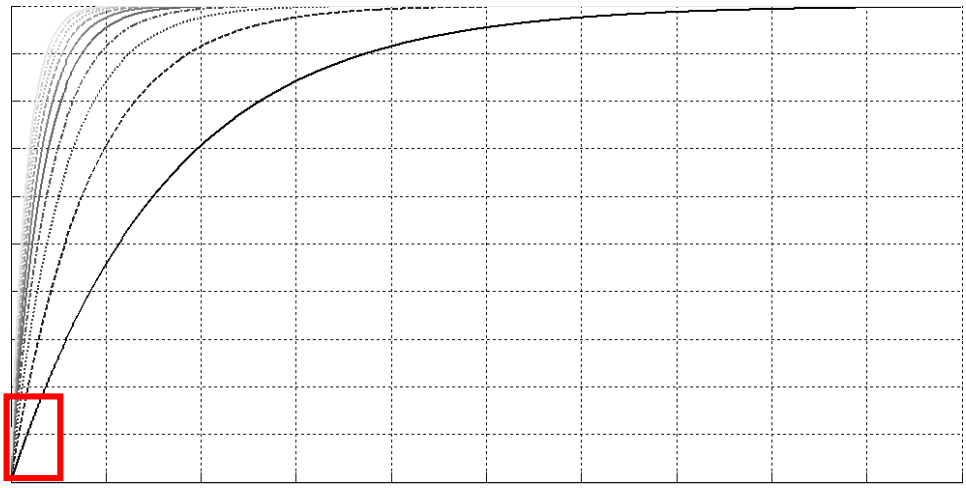

**Fig. 3.10 Gráfica ideal de ataque exitoso con parte visible**

## 3.3 Interfaz de comunicación con el Backend

Una interfaz en Java es una colección de métodos abstractos y propiedades. En ellas se especifica qué se debe hacer pero no su implementación. Serán las clases que implementen estas interfaces las que describan la lógica del comportamiento de los métodos.

En la interfaz establecemos los métodos con lo que reciben y lo que retornaran, por eso solo tendremos una API para las dos implementaciones, y solo deberemos decir si utilizamos la implementación local o la implementación con Spotify. Esto se puede ver de manera gráfica en la Fig. 3.11.

Por lo tanto nuestro proyecto no es sólo una implementación final, sino que también es un esqueleto para otras posibles implementaciones futuras; gracias a cómo se ha diseñado, estas posibles implementaciones podrían ser tanto con música local, como música remota o servicios online de música como p.ej. Grooveshark[18] o SoundCloud[19].

En nuestro caso se han desarrollado dos implementaciones: una con música local, en la que la música se encuentra alojada en el propio servidor, y toda la información se obtiene de estos archivos (explicado en la sección 3.5); otra que se basa en el uso de la API de Spotify, y toda la información se obtiene de sus bases de datos (ver sección 3.6). Con la integración de Spotify se ha querido demostrar la posibilidad de utilizar el mismo código para diferentes implementaciones y mostrar unas directrices para realizar posteriores implementaciones con otros servicios de música en red.

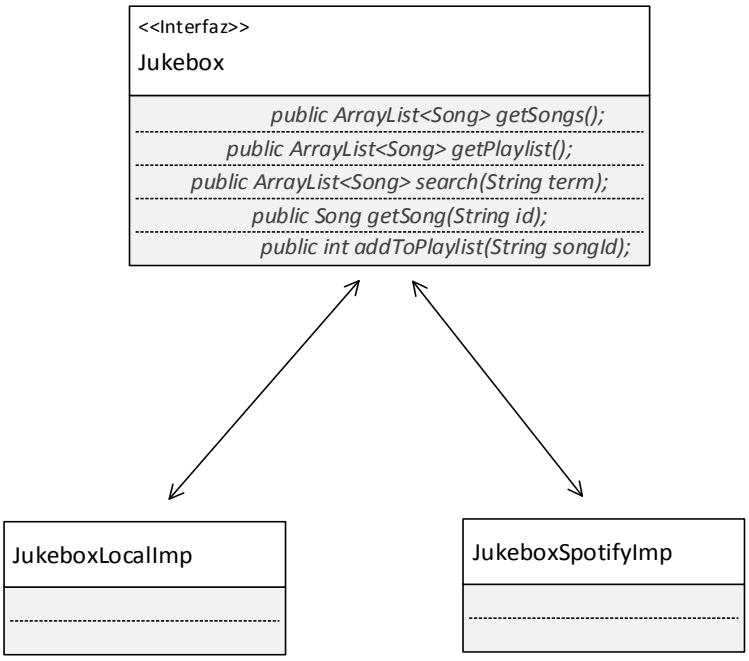


**Fig. 3.11 Esquema en árbol de interfaz e implementaciones**

Los métodos que contiene la interfaz Jukebox son:

- Obtener canciones (`getSongs`): Este método es el encargado de retornar una lista de canciones que serán las que se le mostrarán al usuario. El tipo de variable retornado es un `ArrayList` de objetos `Song`.
- Obtener lista de reproducción (`getPlaylist`): El método también retorna una lista de canciones, pero en este caso son las canciones que serán repreoducidas. El tipo de variable vuelve a ser un `ArrayList` de objetos `Song`.
- Búsqueda (`search`): Permite al usuario realizar una búsqueda al usuario, a partir del texto que se quiere buscar, se retorna una lista de canciones nuevamente.
- Obtener una canción (`getSong`): Este método retorna un objeto Song a partir de su identicador.
- Añadir a la lista de reprucción (`addToPlaylist`): Permite al usuario la introducción de una canción seleccionada a la playlist.

Una vez creada nuestra interfaz lo que debemos hacer es crear nuestras clases para las diferentes implementaciones: local y Spotify. En el momento de declarar la clase debemos añadir `implemements Jukebox`, esto generará los métodos anteriormente indicados, aunque vacíos, y para cada uno de las dos implementaciones lo haremos de manera diferente, pero siempre tendrá la misma estructura, y el objeto que recogerá o devolverá será el mismo (objeto Jukebox).

```
public class JukeboxSpotifyImpl implements Jukebox
public class JukeboxLocalImp implements Jukebox
```

**Fig. 3.12 Declaración de las clases de implementación**

## *3.4 Funciones del administrador*

En este apartado nos centraremos en hacer una descripción de las opciones de las que dispone el administrador del local y la implementación de estas opciones. Nuestro Nukebox se ha diseñado de manera que sea lo más autónomo posible, y una vez configurado el administrador tenga que interactuar lo menos posible y sin ser necesarios conocimientos técnicos. Siendo un usuario más.

### 3.4.1 Generar ticket

La entrega de los tickets o claves que permiten añadir una canción a la playlist que actúan como control de acceso al sistema, son subministrados por el administrador/camarero del establecimiento.

Lo que deberá hacer la persona que se encargue de esta operación será pulsar el botón de generar ticket. En este momento el `.js` hará una petición al servidor para obtener un código.

La generación de códigos se hace de un modo aleatorio, tendremos 30 bits útiles. Para ello generamos un `int` aleatorio, ya que la variable `int` tiene una longitud de 4 bytes que son 32 bits $2^{32}$ $=4294967296$ valores, del que a posteriori se descartarán dos bits.

El aspecto que tienen nuestras claves es un código de cinco caracteres y solo comprenden letras y números, a-z (incluyendo la letra ñ), A-Z y 0-9. Con esto lo que se busca es una mayor comodidad para el usuario, pero sin comprometer la seguridad del sistema.

$$2^6 = 64$$
$$27_{(a-z)} + 27_{(A-Z)} + 10_{(0-9)} = 64$$

Para ello generamos un entero de forma aleatoria con el objeto `Random` de Java, una vez generado hacemos una conversión a binario. Puesto que la conversión a binario descarta los bits a 0 a la izquierda (p.ej. es lo mismo 00010 que 10), iremos comprobando que la longitud sea 32 bits y añadiendo ceros a la izquierda hasta que se cumpla esa condición.

Una vez que tenemos el resultado de la operación de 32 bits, descartamos los 2 últimos quedándonos únicamente con 30. En este momento lo dividimos en cinco elementos de seis bits cada uno, para hacer nuestro propio mapeo. Este mapeo se realiza en la clase `Mapping.java` en la que se hace una llamada al método `mapear` con cada una de las cinco partes y éste retornará un carácter, esta operación se realizará seis veces para posteriormente hacer una concatenación de las respuestas, que será la clave.

En la Fig. 3.13 podemos ver una parte de la tabla de entrada y salida de la conversión que realizamos para generar un ticket, un posible ticket seria:

Entero generado aleatoriamente: `2869652040`
Expresión binaria del número: `10101011000010110110101001001000`
Separación en bloques de 6 bits (+bits descartados): `101010` `110000` `101101` `101010` `010010` `00`

Una vez descartados los dos últimos bits obtenemos;

Resultado final: **`8OUROr`**

| INPUT | OUTPUT |
|---|---|
| 000000 | a |
| 000001 | b |
| 000010 | c |
| 000011 | d |
| ... | |
| 111110 | 8 |
| 111111 | 9 |

**Fig. 3.13 Tabla de mapeo binario a carácter**

La clave se crea en el momento que el cliente la solicita, ya que en momento de creación, se crea una marca temporal o *timestamp* con la hora de creación de este y el código caducará pasados 30 minutos. Nuevamente este valor es una política de sistema y ese valor puede ser modificado para otras implementaciones.

En este momento aparecerá la clave en la página del administrador y le será entregada al cliente.

### 3.4.2 Control del reproductor

Los controles deseables del reproductor son play y stop. Posiblemente algunas implementaciones no permiten el control de estas opciones, como en nuestro caso la implementación Spotify, en la que el reproductor es el propio cliente de Spotify. En el caso de poder utilizar esas opciones, se implementan de la manera descrita a continuación.

Cuándo se pulse el botón de Play llegará una señal al servidor y entonces se inicializará el reproductor. La llamada será a la clase `PlayerController.java` que es la que se encarga de gestionar todas las operaciones relacionadas con el reproductor. Es un *singleton[20]*, es decir una clase de instancia única.

El patrón de diseño *singleton* (instancia única) está diseñado para restringir la creación de objetos pertenecientes a una clase o el valor de un tipo a un único objeto. El constructor de la clase es privado y solo puede ser invocado desde ella misma, a la hora de crearlo no se hace de la manera que estamos acostumbrados como se ve en la imagen inferior:

```
Object newobject = new Object();
```

**Fig. 3.14 Declaración de objeto en java**

Para que otras clases puedan llamarlo lo deben hacer una llamada a un método que se encuentra de manera pública, este método lo que hace es comprobar si el objeto es nulo, de ser así se crea un nuevo objeto, de tipo `PlayerController`, pero esto solo ocurrirá la primera vez, ya que las siguientes veces se retornará el objeto existente.

```
public static synchronized PlayerController getInstance() {
    if (instance == null) {

        instance = new PlayerController();
    }
    return instance;
}
```

**Fig. 3.15 Ejemplo de Singleton**

Estas son las únicas opciones que deberá realizar el administrador además de la generación de claves.

## *3.5 Local*

En esta sección hablaremos de una de las dos implementaciones realizadas, en este caso la implementación local, en la que las canciones están físicamente en el servidor. Esta implementación es la que más opciones nos brinda puesto que todas las partes han sido creadas a nuestra medida. Las partes más relevantes que lo componen son: la lista de canciones disponibles, el reproductor, la lista de reproducción, el buscador y la inserción de canciones a la *playlist*.

### 3.5.1 Lista de canciones disponibles

Para poder mostrar al usuario las canciones que el sistema ofrece en ese momento es necesaria una lista de ellas.
Estas canciones estarán almacenadas en un directorio del equipo que actúa como reproductor. El único requisito es que las canciones estén en formato .mp3 y es muy recomendable que estas canciones contengan la información en los metadatos del fichero mp3, como el nombre de la canción o la imagen del álbum.

Lo primero que el programa hace es una vez que llega la petición REST de obtener la lista de canciones disponibles es comprobar la última modificación del directorio donde estas se encuentran. Para ello utilizamos el método `lastModified()` del objeto `File` de Java.

```
File f = new File(sDirectorio);
long ms=f.lastModified();
```

**Fig. 3.16 Método *lastmodified***

Con esto obtenemos una marca temporal la cual contrastaremos con la última registrada en la base de datos, en el caso de ser iguales, se obtendrá la lista que hay en la base de datos, en el caso de que esos valores no se correspondan se volverán a indexar todos los ficheros mp3, obteniendo los datos de los metadatos y las etiquetas id3v2Tag[21]. Como la imagen del álbum que después se mostrará al usuario.

Una vez obtenidos estos datos se hará una inserción de estos datos en la base de datos. Esto lo que nos permite es que si no ha habido modificaciones es un gasto de tiempo y de recursos volver a indexar, siendo más rápida la obtención desde la base de datos, ya que para el caso de haber una gran cantidad de canciones la operación de indexar puede ser algo lenta.

Para la indexación de los archivos lo que hacemos es ver los ficheros que hay en el directorio y obtener su nombre para crear una ruta. Teniendo una ruta por cada fichero

Tanto si proviene de la base de datos como si proviene de una nueva indexación, se crea un `Arraylist<Song>`, que posteriormente será *parseado* a un *JSONArray*.

### 3.5.2 Player

Ya que queremos un control total de nuestro reproductor y sus listas de reproducción hemos tenido que implementar todo el reproductor.

Nuestro reproductor se basa en dos listas:

La primera lista será una lista auxiliar y su función principal es que aunque ningún usuario ponga canciones en cola no llegue a pararse la música. Esta lista se genera a partir de las canciones disponibles de las que el sistema dispone.

La segunda lista y más importante es la lista que se va generando a partir de las peticiones de los clientes. Estas canciones tienen preferencia respecto a las auxiliares, en el caso de haber canciones pendientes en esta lista sonarán antes de volver a la lista auxiliar. Para que la experiencia sea mejor para el usuario, si la canción que se está reproduciendo es de la lista auxiliar el programa esperará a que esta acabe para reproducirla.

### 3.5.3 Playlist

Una información muy importante para el usuario es ver la lista de reproducción, y además que esta lista sea a tiempo real, la implementación de esta función la encontramos en el método `getPlayList()`. La llamada de este método desde la clase de `JukeboxLocalImp.java` genera un `ArrayList<String>` con el nombre que tiene en el equipo para esa canción, que a su misma vez es su id. La razón por la que es una lista con esos valores es que ese valor es el que pasaremos como parámetro al reproductor ya que se le debe pasar la ruta del fichero.

En la imagen inferior (Fig. 3.17) podemos ver cómo serán las dos listas durante la ejecución del programa; como hemos comentado anteriormente la primera lista es la lista auxiliar que sonará siempre que no se añadan canciones por los clientes, y la segunda serían estas canciones añadidas por usuarios. En color sombreado podemos ver las canciones que ya se han reproducido de ambas listas. Para cada una de las listas tenemos una variable que nos indica la posición que toca reproducir, en la lista auxiliar le hemos llamado *current* y en la principal *inital*. Con esto lo que podemos controlar es si tenemos canciones que reproducir de la lista principal. Cada vez que se reproduce una canción de una lista, aumenta uno ese contador. Es decir, sabremos si hay canciones de la *playlist* por reproducir cuando la resta de ese valor y su longitud sean diferentes, en el caso de ser el mismo valor significará que se ha llegado a la última de esa lista, y hasta que un usuario no añada una nueva sonará únicamente la lista auxiliar.

Encontramos otra variable de tipo *boolean* (con dos posibles valores, true o false), que va cambiando de valor en función de si la canción a reproducir es de la lista auxiliar o la lista principal. Esta variable nos será útil para mostrar la *playlist* al usuario, su uso lo explicaremos a continuación.

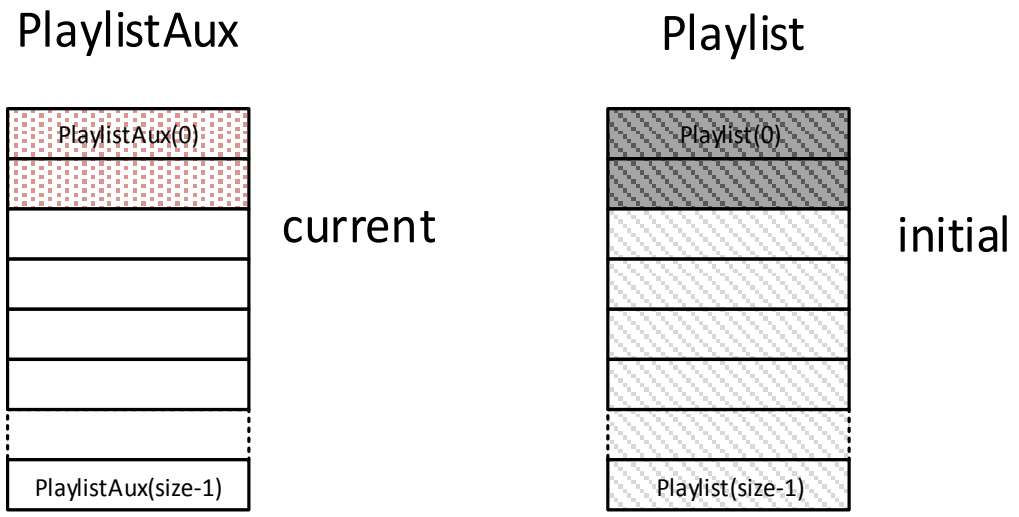


**Fig. 3.17 Listas de canciones auxiliar y añadidas por usuarios**

Para generar la lista primero debemos saber si la canción que se está reproduciendo pertenece a la lista preferente o a la auxiliar. En el caso de ser una reproducción de la preferente la variable *controller* será igual a true. Esto significa que hasta que la longitud y la variable *initial* sean iguales, es decir, que se hayan reproducido todas las añadidas y no haya nuevas, no sonará de la lista auxiliar. Sería el caso de la primera imagen (Fig. 3.18). En caso contrario si la variable *controller* tiene como valor `false` significa que se está reproduciendo una canción de la lista auxiliar, por lo tanto la lista preferente deberá ponerse justo detrás de esta canción, y entonces nos encontraremos en una situación como en el primer caso.

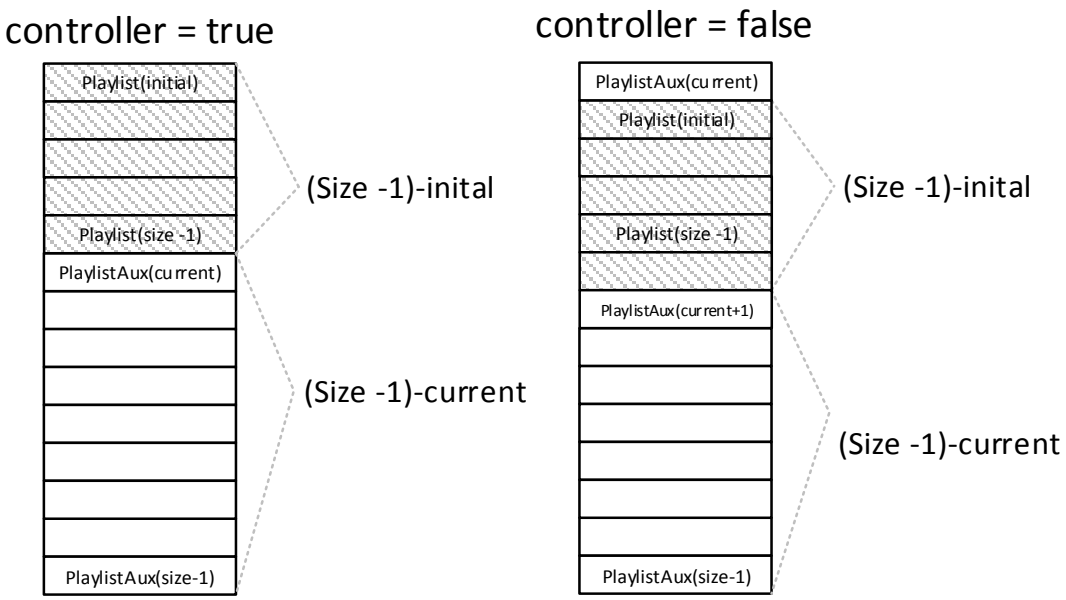


**Fig. 3.18 Posibles casos de lista de reproducción**

Este es el método que contiene toda la lógica que se ha explicado y que se encarga de esta operación:

```
public List<String> getPlayList() {

        List<String> newList = new ArrayList<String>();

        if (controlador == true) {

            if (playlist.size() == 1) {
                newList.add(playlist.get(0));
            } else {
                newList.add(playlist.get(playlist_initial - 1));
            }
            int i = playlist_initial;
            while (i < playlist.size()) {

                if (playlist.size() == 1) {
                    newList.add(playlist.get(0));
                } else
                    newList.add(playlist.get(i));
                i++;
            }
            int i2 = currentsong + 1;
            while (i2 < playlistaux.size()) {
                newList.add(playlistaux.get(i2));
                i2++;
            }

        }

        if (controlador == false) {
            int i = playlist_initial;
            newList.add(playlistaux.get(currentsong));
            while (i < playlist.size()) {
                newList.add(playlist.get(i));
                i++;
            }
            int i2 = currentsong + 1;
            while (i2 < playlistaux.size()) {
                newList.add(playlistaux.get(i2));
                i2++;
            }
            newList.addAll(playlistaux);
        }

        return newList;
```

**Fig. 3.19 Método obtener playlist**

Para que sea más cómodo e intuitivo para el cliente, en el frontend hemos añadido que las canciones que son insertadas por los usuarios tengan un fondo diferente de las canciones que están como auxiliares.

Entonces el usuario sabrá que si inserta una canción esta será insertada justo inmediatamente de la última con fondo coloreado.

Para añadir esta funcionalidad lo que hemos hecho que en el momento que se ejecute la función de obtener la playlist, se inserta un valor delante del nombre, este valor se obtiene en la clase de la implementación local, por defecto cuando se indexan las canciones tienen la variable con el valor “AUX”, entonces en la implementación si el valor nos indica que ha sido introducida por un usuario ese valor se cambia a “MAIN”. En el punto 4.3.4 veremos cómo se ha implementado para que se visualice correctamente.

### 3.5.4 Buscar canción

Otra función que ofrece nuestra aplicación es la de hacer una búsqueda sobre las canciones disponibles. En un entorno real, el administrador del local posiblemente tenga decenas o incluso centenares de canciones, en el caso de querer ver todas las disponibles, se le ofrecen al usuario como hemos visto en un apartado anterior, pero si el usuario quiere buscar un artista, álbum o canción disponible puede resultar incómodo. Esto es por lo que es necesaria esta función.

Una vez que el servidor recibe la petición de búsqueda, se ejecuta el método de nuestra implementación que realiza una petición a la base de datos SQL. Esta petición consulta las diferentes columnas de las canciones como artista, nombre del álbum o nombre de la canción. La función que nos permite hacer esto de una manera eficiente es `LIKE`, esta función lo que hace es devolver entradas que contengan el parámetro dado. Aquí podemos ver un ejemplo que devolvería la canción en la que el nombre de la canción contenga el parámetro.

```
SELECT * FROM song  WHERE Name LIKE '%parametro%';
```

Esto nos devolverá una lista de canciones que el servidor devolverá en forma de `JSONArray`.

### 3.5.5 Comprobación de ticket

Una vez que el usuario introduzca esa clave se realizará una petición al servidor con el identificador de la canción y la clave para que sea comprobada.

Una vez que el servidor recibe la petición y se entra en el método de comprobación de la clave lo primero que se aplica es un retardo de 3 segundos mediante `Thread.sleep(3000)` (el tiempo que se pasa como parámetro está en `ms`). El fin de este retardo es reducir el número de intentos que se pueden efectuar en un tiempo determinado (sección 3.2.3), sin que el usuario pierda interactividad.

Una vez completado el retardo lo primero que hace es comprobar si la IP del usuario esta *baneada*, como hemos comentado anteriormente, significa que ha excedido el número máximo de claves erróneas permitido.

Lo primero que se hace es una consulta a la base de datos con la IP del usuario que hace esta petición, se comprueba el valor 'banned' y en caso de ser `true` el usuario verá un mensaje indicando que ha excedido el número de intentos. Si el resultado obtenido es `false`, se procederá a comprobar la clave. Aquí podemos ver las diferentes respuestas que devolverá el método de comprobación (ver Fig. 3.20). El motivo de ser variables de tipo `static` es porque esto permite acceder desde otras clases a esas variables, en nuestro caso desde la clase `Api.java` para hacer comprobaciones de la respuesta de la BBDD haciendo más comprensible el código.

```
public static int TICKET_OK = 0;
public static int TICKET_ERR_NOT_FOUND = -1;
public static int TICKET_ERR_EXPIRED = -2;
public static int TICKET_ERR_ALREADY_USED = -3;
public static int IPCLIENT_BANNED = -1;
public static int IPCLIENT_OK = 0;
```

**Fig. 3.20 Variables de respuesta en BBDD**

Primero se comprueba si lo que nos devuelve algo la consulta, en caso contrario significa que ese ticket no existe, si el ticket se encuentra en la base de datos comprobamos los campos usado y expirado. Si el usuario introduce una clave errónea por primera vez se inserta su IP en la tabla `clientip`, con el contador de intentos a 1, por cada intento erróneo se incrementará el valor, hasta el valor de intentos erróneos máximos, que cambiara el valor de baneado de false a true. En caso de ser una clave correcta se le notificará al usuario y la canción será añadida.
Desde ese momento el usuario podrá ver donde se ha insertado su canción, y cuando se reproducirá. En la Fig. 3.21 podemos ver un diagrama del funcionamiento.

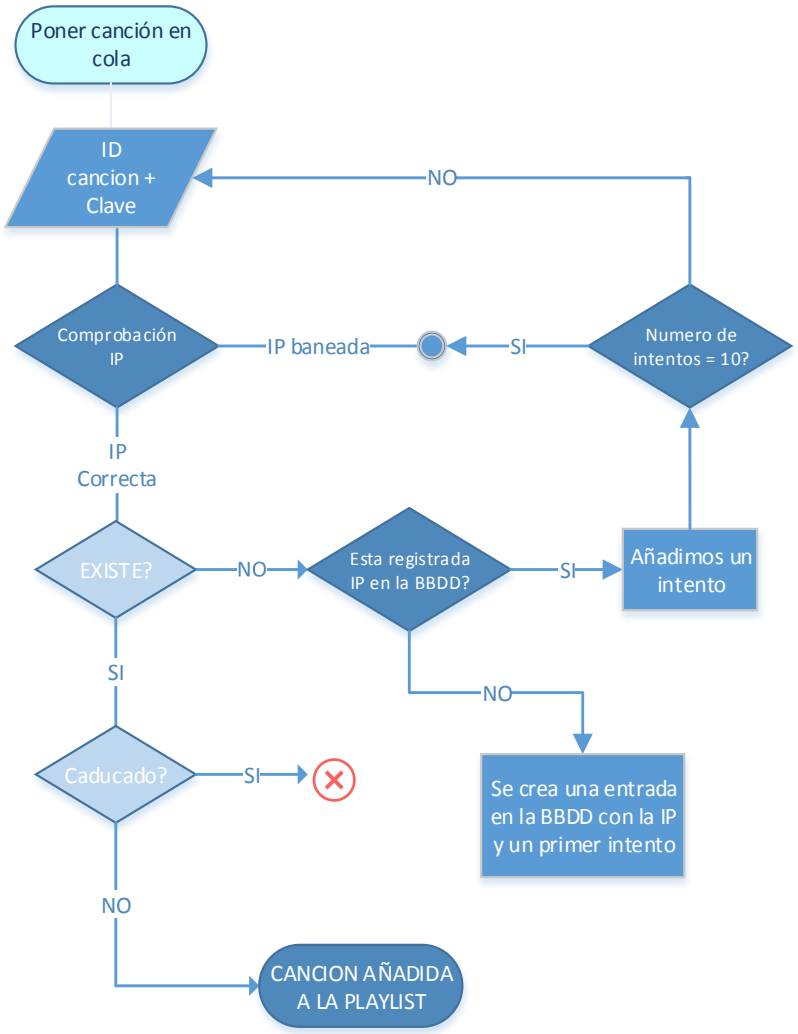


**Fig. 3.21 Diagrama de introducción de canción en cola**

## *3.6 Spotify*

La segunda implementación que hemos realizado ha sido la implementación Spotify, la elección de esta plataforma ha sido motivada por la gran difusión que ha tenido en estos últimos años esta plataforma.

Spotify fue lanzado el 7 de octubre de 2008 al mercado europeo, mientras que su implantación en otros países se realizó a lo largo de 2009. La empresa, que tiene su sede en Estocolmo, Suecia, ha firmado acuerdos con las discográficas Universal Music, Sony BMG, EMI Music, Hollywood Records y Warner Music entre otras.

A enero de 2015 contaba con 60 millones de usuarios activos de los cuales 15 millones eran usuarios de pago. Por nombrar algunas cifras, en diciembre de 2012 proporcionaba aproximadamente 20 millones de canciones y en el año 2014 el número horas de reproducción superó los 7.000 millones.

**Fig. 3.22 Logo Spotify**

Para la implementación con Spotify la manera de interactuar y obtener información la hemos hecho mediante su API, fue publicada en junio de 2014 y ofrece datos sobre listas de reproducción, canciones y álbumes y ofrece esta información en formato JSON.

Como hemos comentado, la manera en la que se ha diseñado nuestra aplicación permite añadir una nueva plataforma únicamente creando la clase de la en la que se especifica cómo se hace, siempre implementando la interfaz de Jukebox. Por lo tanto con el uso de Spotify se ha querido demostrar la facilidad de añadir otros sistemas, al esqueleto del Nukebox.

La api de Spotify permite obtener y gestionar contenido de Spotify mediante peticiones a su URL, algunas de estas peticiones son abiertas, es decir sin autenticación, mientras que otras necesitan de permisos, como acceso a un usuario o listas de reproducción.

Las operaciones que haremos en Spotify serán añadir una canción a la *playlist*, obtener una canción en concreto y hacer una búsqueda por nombre de artista, álbum o canción.

Lo primero que debemos hacer para poder utilizar la API de Spotify es estar registrado o registrarse en https://www.spotify.com/es/signup/ y crear una aplicación en: https://developer.spotify.com/my-applications.

Una vez en esta página deberemos pulsar en *Create an app* (Fig. 3.23), después de pulsar nos aparecerá un formulario en el que deberemos especificar el nombre de nuestra aplicación, y una pequeña descripción como se puede observar en la Fig. 3.24.

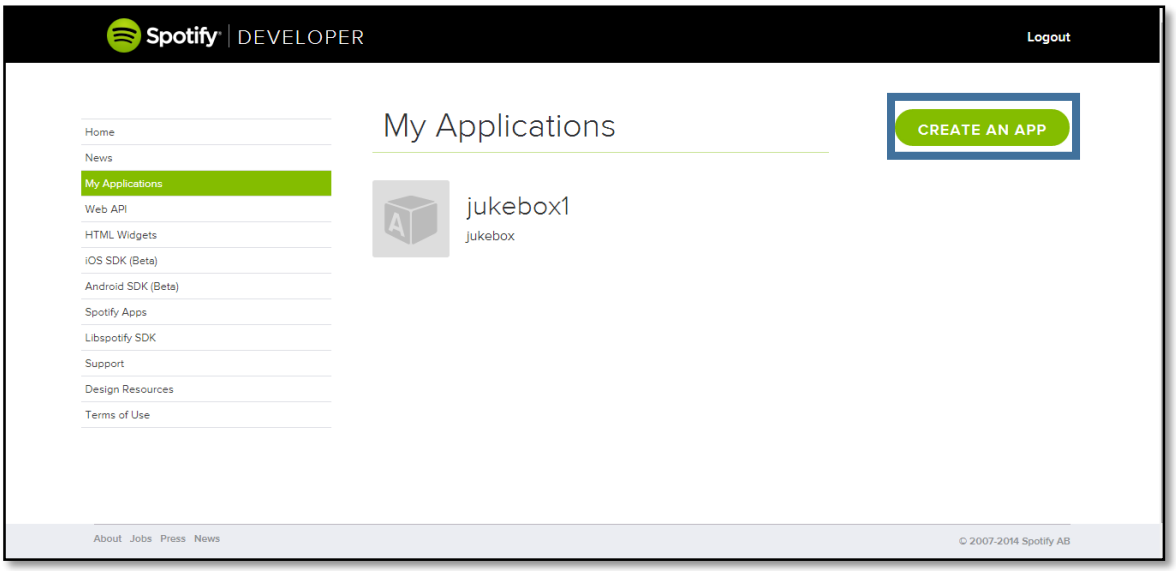


**Fig. 3.23 Pantalla principal de Spotify Developers**

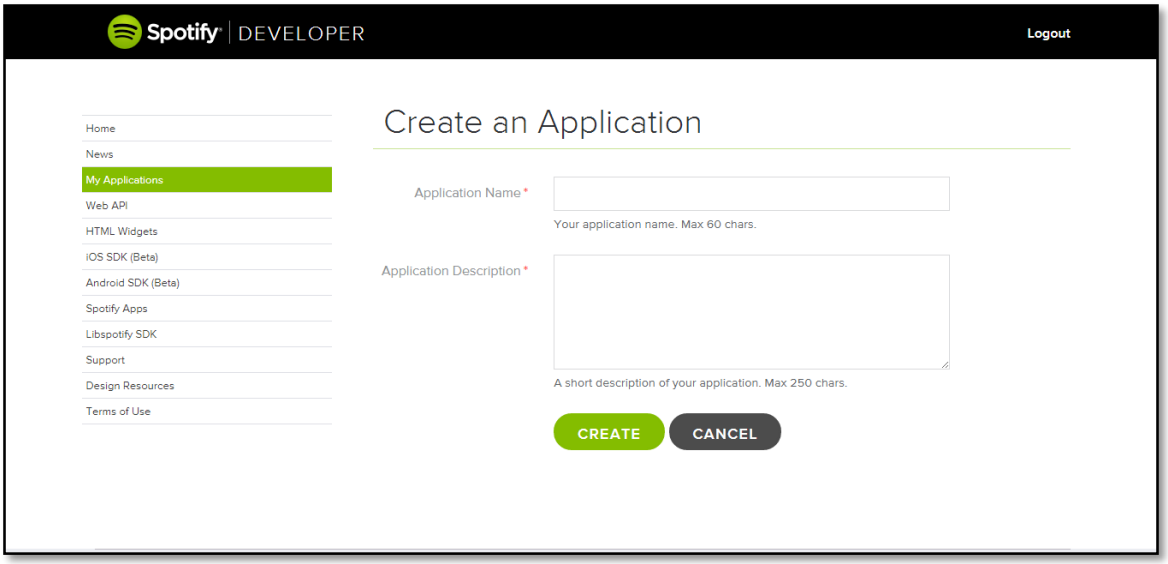


**Fig. 3.24 Creación de nueva aplicación**

En este momento nuestra aplicación ya está creada y podemos obtener los datos que necesitamos para poder acceder a la API, estos datos son `Client ID` y `Client Secret`. Más adelante veremos porque son importantes esos datos.

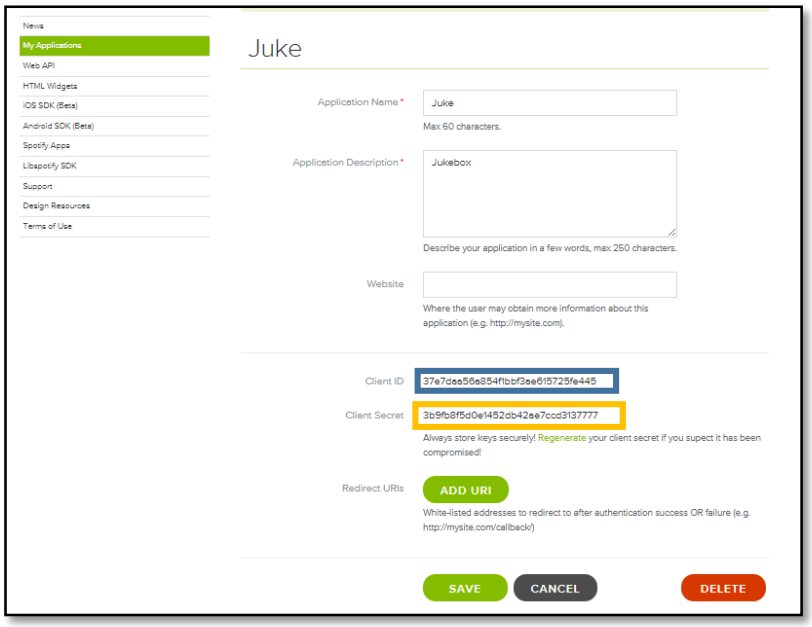


**Fig. 3.25 Obtención de Client ID y Client Secret**

El lenguaje empleado en la API oficial de Spotify es node.js y por lo tanto el tutorial está explicado para ese lenguaje, aunque en la página oficial de Spotify developers, en el apartado que trata sobre la Web API podemos encontrar ejemplos de código y librerías en otros lenguajes como .NET, PHP, Phyton y Perl.

Para nuestra aplicación vamos a utilizar una adaptación o *wrapper* en Java, que nos permite trabajar con objetos directamente. Para utilizarla lo que debemos hacer es descargarla desde el enlace de Github y agregarla a nuestro proyecto como una librería.

Al estar escrito nuestro código en java, utilizar la librería nos permite hacer las operaciones directamente con métodos, evitando hacer las peticiones, y trabajamos directamente con objetos POJO.

Como en el capítulo anterior, a continuación veremos cómo se han implementado las diferentes funciones y veremos las limitaciones que se han tenido con esta implementación, ya que no tenemos tanto control en algunas partes.

### 3.6.1 Integración y configuración

Una vez descargado el proyecto de Github e importado a nuestro *workspace* de Eclipse lo primero que debemos hacer es referenciarlo a nuestro proyecto, en la siguiente imagen. Con esta operación ya podemos hacer llamadas de funciones y crear objetos de la librería.

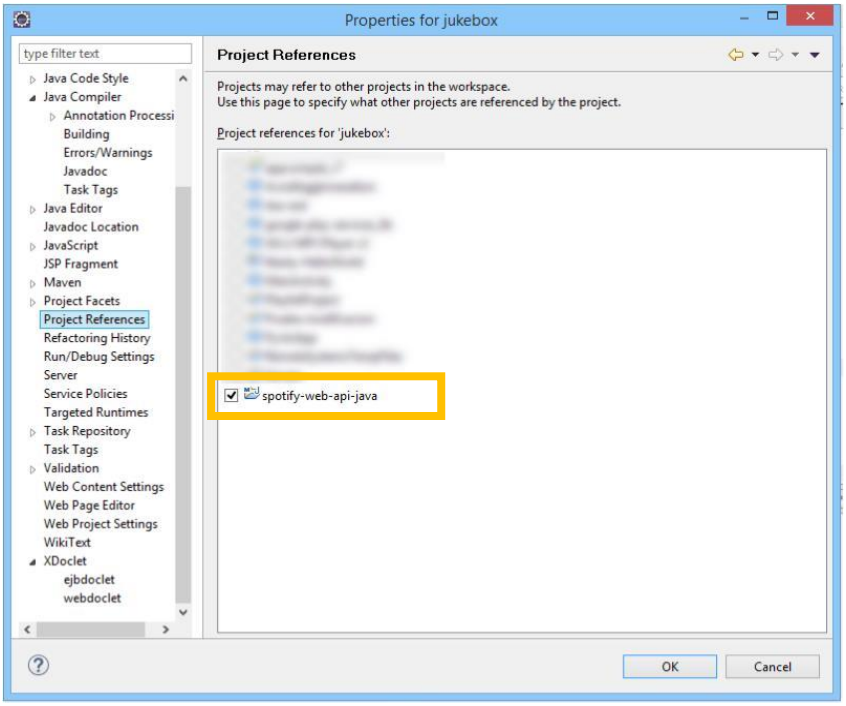


**Fig. 3.26 Referencia de librería Spotify**

### 3.6.2 Funciones

En nuestro proyecto hemos creado la clase `SpotifyOperations.java` que es donde gestionaremos todas las operaciones. Como hemos comentado, la manera de interactuar será con objetos y métodos Java, y la propia librería es la que se encargará de hacer las peticiones y mapearlo a objetos Java.
En la siguiente figura podemos ver el objeto Api, que es el que nos permitirá autenticarnos con la API de Spotify y para ello debemos introducir los valores que hemos obtenido de la página de Spotify developers.

```
final Api api = Api.builder()
            .clientId(clientId)
            .clientSecret(clientSecret)
            .redirectURI(redirectUri)
            .refreshToken(refreshToken)
            .build();
```

**Fig. 3.27 Creación de objeto Api**

```
final static String clientId = "dd3d4e7aa058411e8d236c37b64d0111";
final static String clientSecret = "dcfed4ca09394611afae7d7286984d8a";
final static String redirectUri = "http://localhost:8888";
final static String refreshToken = "AQCQXUjJ_hKt6GEmNpelhYbqzzeBCEDsi5OsjN2gSg8heQntv9
```

**Fig. 3.28 Parámetros necesarios**

Para hacer cualquier operación utilizamos los métodos de la clase Api que previamente hemos construido.

En esta clase encontramos tres métodos:

Encontramos dos tipos de peticiones, las de libre acceso, es decir que no necesitan de un *token*, y por el contrario las que necesitan de ellos.

`searchTrack`: hacer una búsqueda

Para la búsqueda de una canción le pasamos el parámetro de la búsqueda al método, este lo que hace es al objeto api pasarle ese parámetro y la misma librería se ocupa de la petición .Hemos establecido un límite de 20 ítems, para que la petición funcione más rápido y son suficientes respuestas.

`getTrack`: obtener una canción

Este es el método que utilizaremos para obtener la canción seleccionada por el usuario para añadir a la *playlist*. La petición se hace a partir de la id de la canción, la respuesta que obtenemos es nuevamente un objeto `track`.

`addSong`: Añadir una canción a la *playlist*

Para añadir una canción en la *playlist* pasamos como parámetro el id de esa canción, y ejecutamos el método `addTracksToPlaylist()` del objeto api pasando como parámetros el usuario de Spotify, el identificador de la *playlist* donde queremos que se inserte, la canción y por último la posición en la que queremos añadirlo, que en este caso es la última de la *playlist*.

Aquí podemos ver el aspecto que tiene esta sentencia.

```
Final AddTrackToPlaylistRequest
request2=api.addTracksToPlaylist("jukebox", "68a1ualmfcXujcZYkjYMCD",
tracksToAdd).position(-1).build();
```

Para buscar y pedir una canción no necesitamos un *token*, ya que son públicos y no necesitamos permisos del usuario. Estos dos métodos nos permiten hacer una búsqueda como podríamos hacer en la misma aplicación de Spotify, y esta nos devolverá una lista de canciones, y el método `getTrack()` nos permite obtener una canción a partir de su id.

El método más interesante es el método de añadir a la *playlist*, ya que aquí sí que necesitamos los *token*. Previamente el usuario en el cual se añadirán las canciones a la lista de reproducción tiene que aceptar la aplicación que anteriormente hemos creado. En este punto obtendremos el refreshToken. Ya que cuando obtenemos un *token* para tener acceso a información del usuario, este *token* tiene una validez de 3600s (1 hora) y tendremos que ir actualizando ese *token* temporal mediante el refreshToken.

Aquí podemos ver cómo funciona la petición de un *token* actualizado:

```
final RefreshAccessTokenRequest request = api.refreshAccessToken().build();
RefreshAccessTokenCredentials refreshAccessTokenResponse = request.get();
api.setAccessToken(refreshAccessTokenResponse.getAccessToken());
```

**Fig. 3.29 Obtención de *token* actualizado**

En la siguiente imagen podemos verlo a modo de esquema, gracias a una imagen de la documentación oficial de Spotify[23](ver Fig. 3.30).

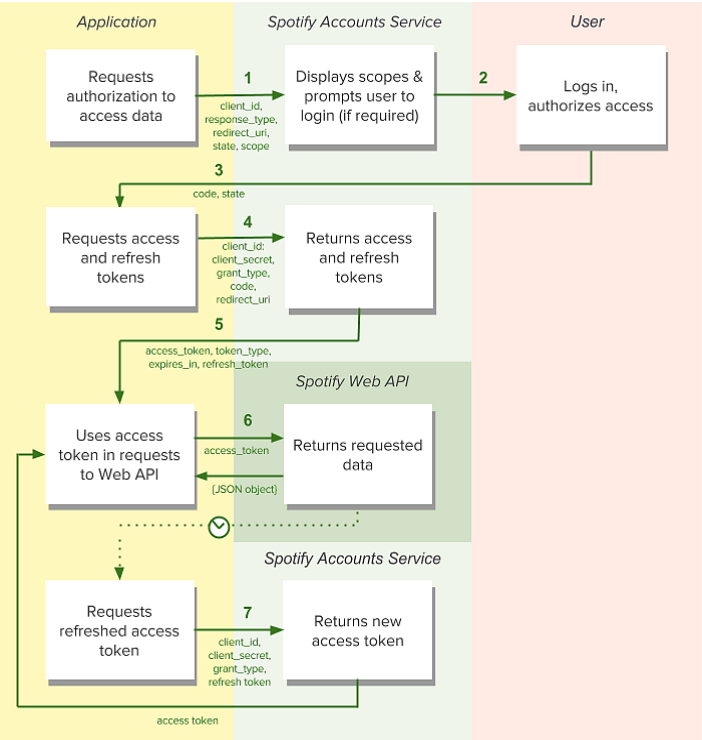


**Fig. 3.30 Esquema lógico de acceso y credenciales**

Como hemos visto en los métodos, la librería de Spotify que hemos obtenido trabaja con unos objetos diferentes a los de nuestra implementación por lo tanto será necesario realizar un mapeo de estos objetos a los POJO de nuestro proyecto como en el caso de `Song` que en caso de la librería el objeto es `Track` y los nombres de las variables son diferentes. Por lo tanto en las operaciones de obtener una canción o una lista de ellas, será necesario este mapeo.

### 3.6.3 Limitaciones

Algunos inconvenientes con los que nos hemos encontrado en esta plataforma, han sido principalmente, la ausencia de control en algunas partes. El factor más relevante ha sido la ausencia de control del reproductor y por lo tanto la gestión de las listas de reproducción. La API de Spotify permite hacer peticiones de listas de reproducción que retornan listas de canciones, y también nos permite eliminar elementos de esa lista de reproducción, el problema viene dado por no poder saber cuándo termina una canción y de esta manera eliminarla. Al no poder eliminar canciones reproducidas, la lista de reproducción que podemos obtener es una lista acumulativa de todas las canciones añadidas y carece de utilidad para el usuario.

Respecto a la lista de canciones disponibles, en el caso de la implementación Spotify, no se muestran todas las canciones de las que dispone el usuario, aparecen las canciones de las cuales el usuario hace una búsqueda.

# 4 CAPÍTULO 4. IMPLEMENTACIÓN DEL FRONTEND

En esta sección trataremos los temas relativos a la interfaz o *frontend* de nuestra aplicación. Encontramos dos partes diferenciadas, la parte que verá el usuario con la que interactuará con las canciones y lista de reproducción, y la parte del administrador donde se gestionan las claves y el reproductor.

## 4.1 Interfaz de usuario

Las partes que componen la interfaz o *frontend* son un fichero HTML, un *js* (Javascript) para hacer nuestra página dinámica y la hoja de estilos CSS.

Encontramos una página HTML sencilla, ya que la información relevante es la que proviene de las peticiones HTTP. A continuación veremos las partes que la componen.

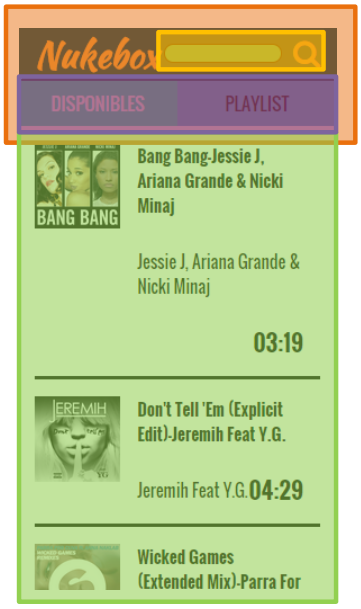


**Fig. 4.1 Partes del html**

En la imagen superior (Fig. 4.1) vemos diferenciado en colores las diferentes partes, en color naranja podemos ver la cabecera, esta parte es estática e igual para todas las vistas, aquí podemos encontrar el título a modo de logo, aunque se trata de una tipografía ya que de este modo nunca se *pixelará*, porque se compone de vectores y no una imagen. La tipografía en este caso es Kaushan Script. En esta cabecera también encontramos, marcado con sombreado amarillo el cuadro de texto para la búsqueda con un icono de una lupa para ser más intuitivo, este icono también se trata de una tipografía por el mismo motivo que el título.(Font-awesome). En color morado encontramos los botones para navegar a las canciones disponibles o la lista de reproducción (en función de donde nos encontremos se resaltará uno u otro).

En la parte sombreada en verde podemos ver la lista de canciones, en nuestro HTML únicamente se trata de un *div* que será rellenado a partir de una función de Javascript. La línea que aparece en nuestro HTML es la siguiente:

```
<div class="contenedorCanciones" id="songs-data">
```

En nuestro Javascript cuando tenemos la lista de canciones en formato JSON rellenamos nuestro *div* con la función de jQuery `.append` con código HTML. Aquí podemos ver un ejemplo de cómo se implementa:

```
$('#songs-data').append("<h1>Aqui generamos código HTML.</h1>");
```

**Fig. 4.2 Función append de jQuery**

## 4.2 Apariencia de la interfaz de usuario

En este aspecto, hemos querido darle a la parte del cliente un diseño sencillo y minimalista pero cuidado. Hemos diseñado la web inspirados en la tendencia que se viene siguiendo este último tiempo para el diseño de webs y aplicaciones, llamado *flat design* o estilo plano. Cuando hablamos de *flat design* aplicado a la web, nos referimos al diseño de una página web que decide usar formas geométricas, colores planos y llamativos e iconos en vez de fotografías o imágenes complejas, al margen de sombras y profundidades.

Un buen ejemplo lo encontramos en la página de Google design[24] donde establece algunos patrones y paletas de colores para el diseño de sus aplicaciones y webs.

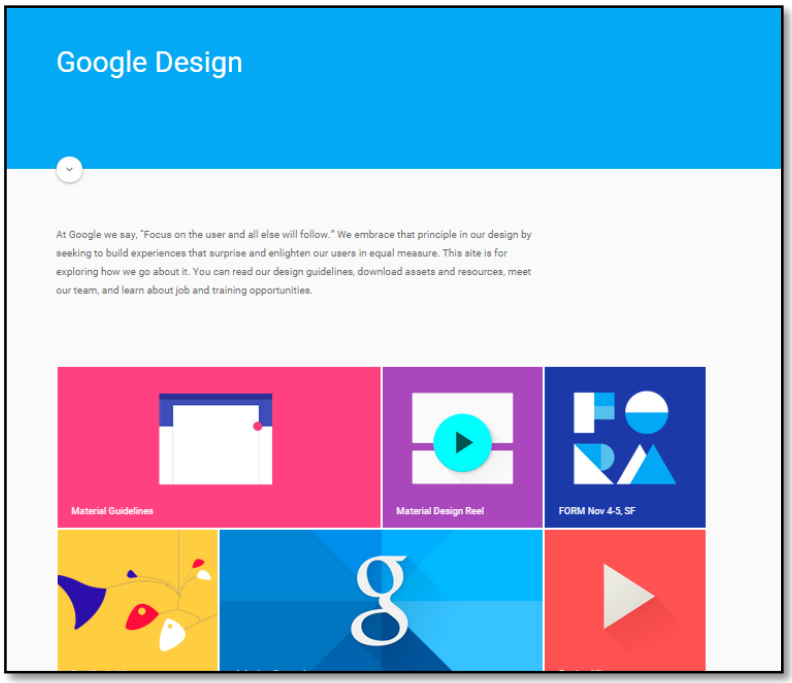


**Fig. 4.3 Página principal de Google design**

En nuestro proyecto la paleta de colores con su valor hexadecimal utilizada es la descrita a continuación:

| | |
|---|---|
| Nombre e icono | #ed9e46 |
| Cabecera | #00445e |
| Seleccionado | #1de9b6 |
| No seleccionado | #45afdc |
| Canción insertada | #B9F6CA |
| Texto | #000000 |
| Retroceder | #ef5350 |

Para la utilización de las tipografías que hemos comentado en el apartado anterior tanto para el titulo como lo el icono hemos tenido que añadir la fuente en nuestro CSS cómo podemos ver en la línea inferior:

```
@font-face{ font-family:’nombre'; src:url(‘url’);
```

Otro aspecto importante que hemos tenido en cuenta es el ajuste de la pantalla para diferentes dimensiones. Principalmente el uso del Nukebox será a través de *smartphones*, por lo tanto es primordial que se pueda interactuar de manera cómoda en esas dimensiones de pantalla.

Para ello hemos utilizado las *media queries* de CSS3. Esto lo que hace es que definir unos estilos para los elementos en función del tamaño de pantalla, orientación incluso color.

Un ejemplo es la adaptación del tamaño de la fuente en función de si se accede desde un navegador de un PC o un teléfono. Para el acceso desde el teléfono se deberá ampliar el tamaño de fuente para que pueda ser leído por el usuario. En la imagen inferior podemos ver un ejemplo de cómo se definen estas operaciones:

```
@media (min-width: 1200px) {
  .container {
    width: 960px;
  }
}
```

**Fig. 4.4 Ejemplo de media query CSS3**

Se ha desactivado el zoom, para garantizar una buena experiencia, ya que no es necesario el uso de zoom, la propia hoja CSS muestra el contendido dependiendo de las dimensiones y se evitan comportamientos no deseados.

Para algunas partes de nuestra aplicación hemos utilizado el framework de Materialize[25], que contiene una hoja de estilos CSS y un archivo .js para las animaciones.

## 4.3 Funcionalidades de usuario

En este apartado veremos las funciones de las que dispone el usuario en el Nukebox, y haremos un recorrido por las diferentes vistas.

### 4.3.1 Canciones disponibles

La primera página a la que accede un usuario es la página de canciones disponibles, donde se muestra una lista de canciones generada por la API como hemos visto en el capítulo de implementación.

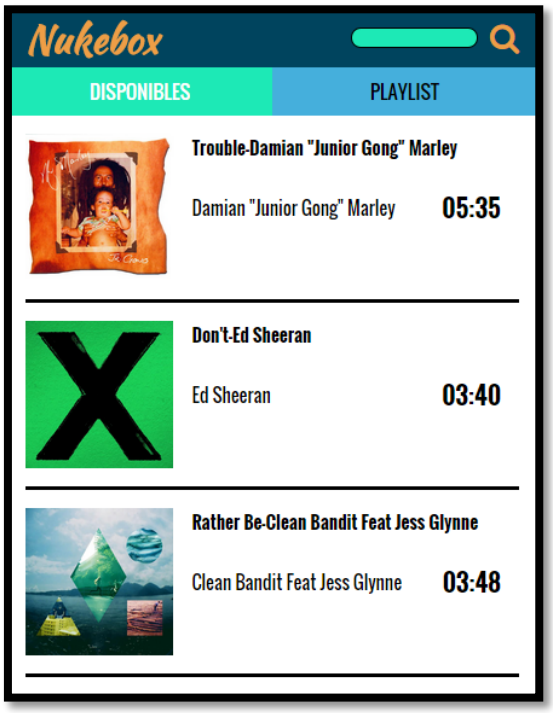


**Fig. 4.5 Página principal**

### 4.3.2 Búsqueda

Desde la misma página el usuario puede realizar búsquedas por nombre de canción, nombre de artista o álbum, a medida que el usuario vaya introduciendo letras irán apareciendo resultados.

**Fig. 4.6 Búsqueda**

Como vemos en la Fig. 4.6 encontramos un cuadro de texto donde el usuario puede introducir su búsqueda.

Para hacerlo más agradable al uso hemos implementado una función de autocompletar, en un primer momento se pensó en mostrar las sugerencias debajo de ese cuadro de texto, como estamos acostumbrados a verlo en la web, pero ya que principalmente se utilizará en un smartphone dónde la pantalla suele ir desde las 3.5” a las 5-5.5” (para los modelos más grandes), puede resultar incómodo. Por lo tanto hemos lo hemos implementado de manera que los resultados que aparecen son en función de esa búsqueda, o un mensaje de error en caso de que la búsqueda no tenga coincidencias.

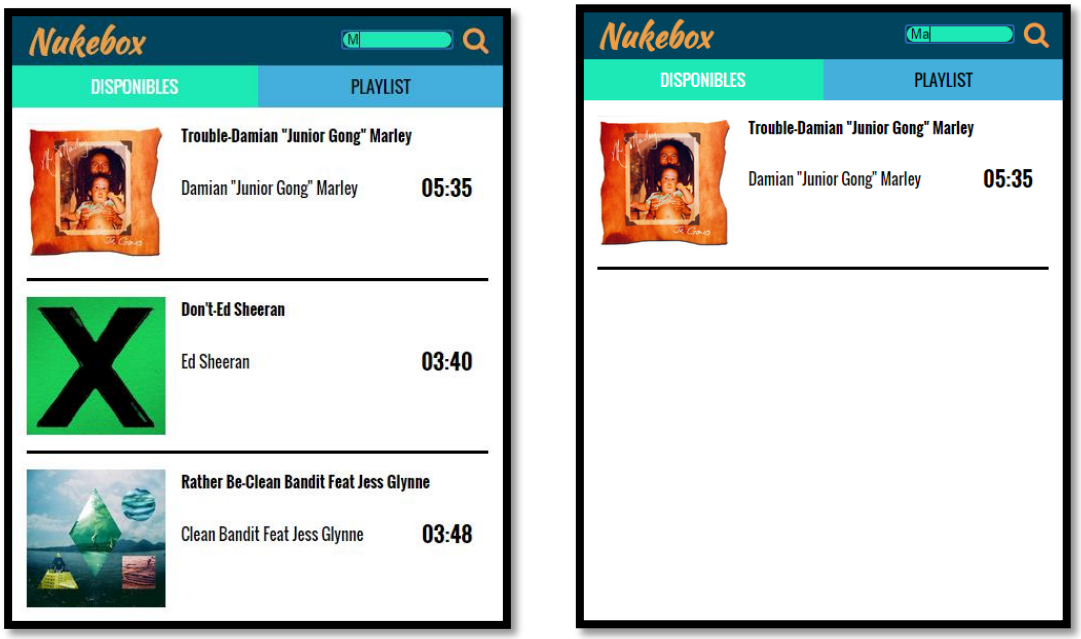


**Fig. 4.7 Búsqueda II**

El ejemplo anterior muestra una búsqueda para el artista Damian Marley. Cuando introducimos ‘M’, encontramos tres coincidencias, mientras que si añadimos la letra a ya solo encontramos la canción que buscábamos, en el caso de introducir una letra más que no coincide, nos muestra un mensaje de que no tenemos coincidencias(Fig. 4.8).

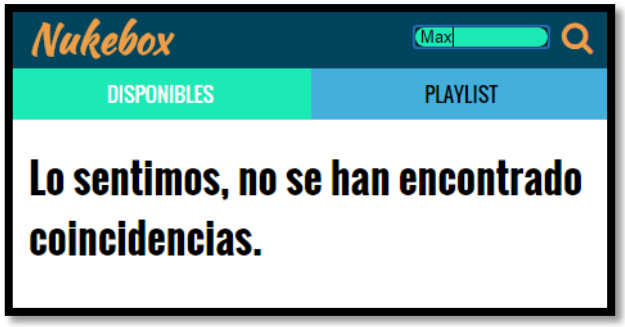


**Fig. 4.8 Búsqueda sin coincidencias**

Para mayor comodidad del usuario la propia aplicación irá haciendo la petición REST al servidor a medida que se vaya escribiendo, esto lo hemos implementado con la función `keyup()`[27] de jQuery sobre el cuadro de texto,

cada vez que el usuario retire el dedo del teclado, o deje de escribir se hará la petición. Esto también previene consultas innecesarias como el ejemplo de pulsar durante un tiempo la misma letra, en la que solo se hará una consulta.

```
$('#track').keyup(function(){
    searchsong()
});
```

En el fragmento de código anterior vemos como está escrito en nuestro js, capturamos el evento `keyup` del campo de texto que tiene como id track y ejecutamos la función de buscar. En este fragmento no solo se ha incluido esa parte, ya que en nuestro `.js` también limpiamos el `div` donde van apareciendo.

### 4.3.3 Introducción de ticket

Una vez que el usuario decide la canción que quiere poner en cola, y pulsa esa canción será redirigido a otra página, en la que se mostrará esa canción con su información. Encontrará un cuadro de texto en el que se le pedirá al usuario que introduzca la clave que previamente se le ha subministrado (ver Fig. 4.9).

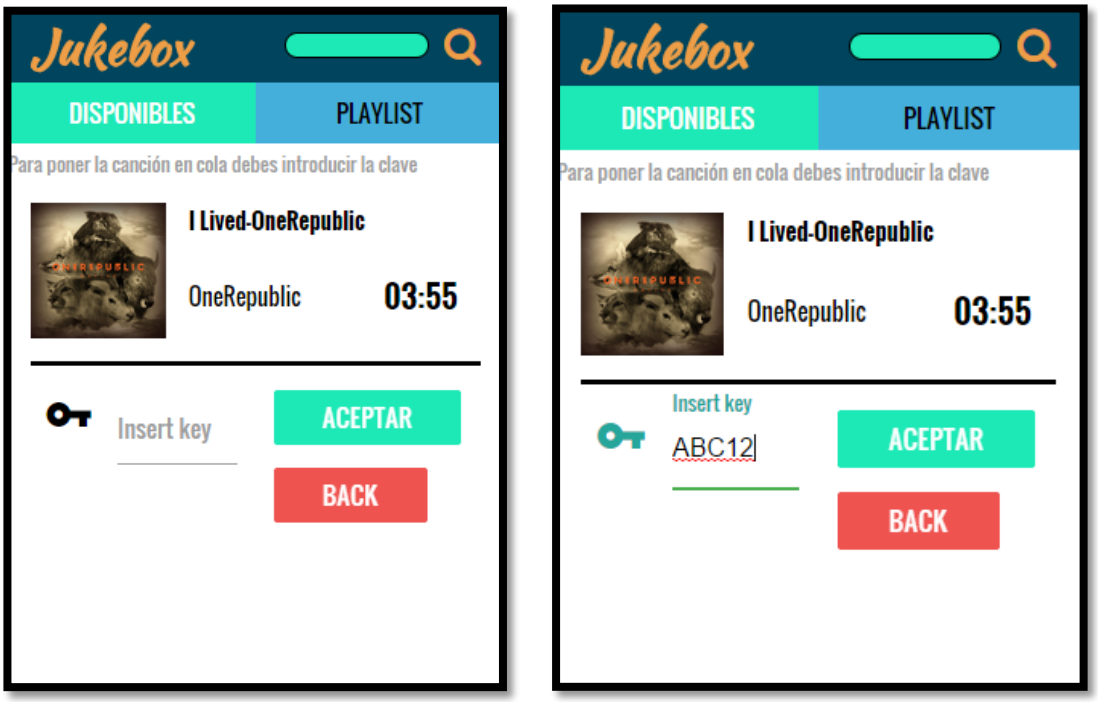


**Fig. 4.9 Introducción de ticket**

Encontramos un cuadro de texto con un icono de una llave, que hace una pequeña animación al pulsar en él. Una vez introducido el código este será canjeado y no podrá volver a ser usado. También encontramos un botón para volver a la página principal.

### 4.3.4 Lista de reproducción

Otra función de la que dispone el usuario es visualizar la lista de reproducción que va actualizándose dinámicamente. Para ello deberá pulsar sobre el botón de playlist.

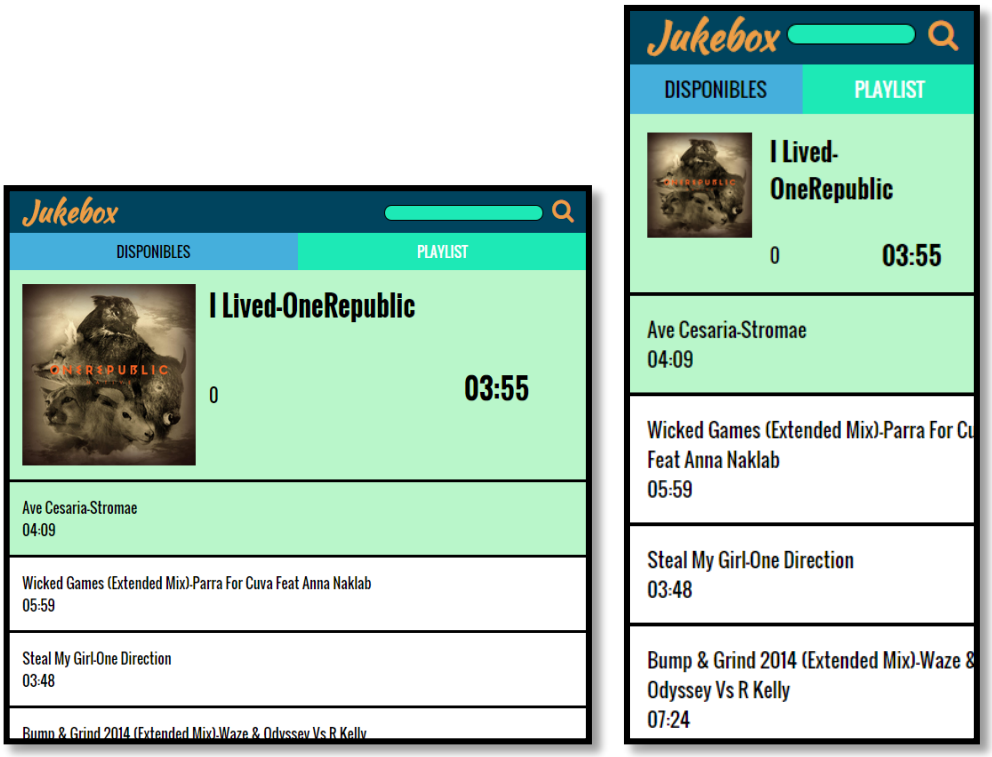


**Fig. 4.10 Vista móvil y pc de la lista de reproducción**

Cuando se genera la página se ejecuta la función para obtener la playlist, en el momento que nuestro `.js` recibe el `JSONArray` de la API comprueba ese valor, en función de si es MAIN o AUX el nombre que se le asigna a la `class` en HTML es diferente. Lo único que debemos hacer es que en el CSS se le asigna un *background* en función del `class`.

El usuario en todo momento verá la lista actualizada con las canciones que otros usuarios hayan añadido, para ello lo que hemos hecho en el `js` de la playlist es ejecutar la función para obtener la lista de reproducción cada segundo. En la imagen siguiente vemos el código necesario. Lo único que debemos indicar es la función que queremos ejecutar y el intervalo de tiempo en ms.

```
window.setInterval(function(){

    getplaylist();
    }, 1000);
```

**Fig. 4.11 Obtención de playlist cada segundo**

## 4.4 Interfaz de administrador

La parte de administrador consta de dos vistas, la primera un login sencillo para asegurar que sólo puede acceder a la segunda página que es la que permite la generación de claves y la gestión del reproductor.

### 4.4.1 Login

En la página de login encontramos dos cuadros de texto para introducir el usuario y contraseña, con la misma animación que encontrábamos en la página de introducir ticket. El cuadro de texto de password, hemos indicado que sea de tipo password para que aparezcan puntos en lugar de texto, lo hemos declarado de la siguiente manera en el HTML: `<input id="password" type="password">`. También encontramos un botón proceder la autenticación. Vemos que la cabecera es igual que en el resto de la aplicación.

**Fig. 4.12 Pantalla de login**

Cómo hemos explicado en el capítulo de la implementación del login (3.2.2) cuando el login es correcto el usuario recibe el *token* que posteriormente se enviará al servidor para las diferentes operaciones, este *token* se almacena en una *cookie* y para el administrador es totalmente transparente.

Para el uso de *cookies* en jQuery primero debemos añadir la referencia al *plugin* de *cookie* de jQuery[26], que previamente hemos descargado. Para añadirlo lo hacemos de la siguiente manera:
`<script src="js/jquery.cookie.js"></script>`. Esto permite almacenar datos de manera muy sencilla en nuestro js, en la imagen siguiente vemos un ejemplo del guardado y la obtención de un valor en una *cookie*.

```
$.cookie('name', 'value');
```

```
$.cookie('name');
```

**Fig. 4.13 Guardado y obtención de datos en la sesión cookie**

### 4.4.2 Página de administrador

Ésta es la página desde donde el administrador puede interactuar con la aplicación. Las operaciones son el control sobre el reproductor (como el play y stop) y la función de generación de las claves. En la siguiente imagen podemos ver el aspecto que tiene (Fig. 4.14 Página principal del administrador):

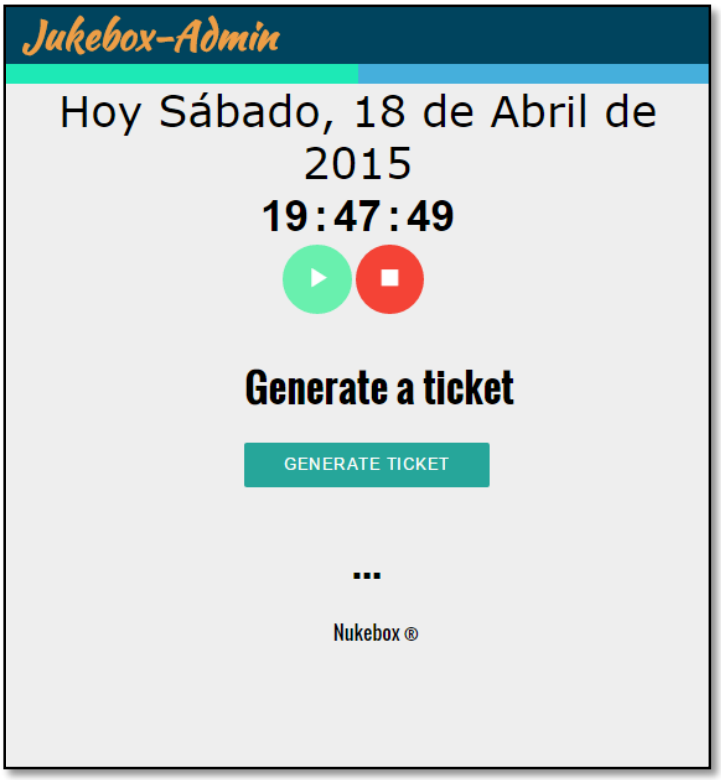


**Fig. 4.14 Página principal del administrador**

Para el uso de estas funciones es necesario disponer de un *token* válido ya que en cada petición se le enviará al servidor, para ello lo obtendremos de la cookie como hemos explicado en el punto anterior. En el caso de no disponer de *token*, o de que éste haya expirado se le redireccionará de nuevo a la página de login y se volverán a pedir los valores de la autenticación.

Antes de inicializar el reproductor el botón de stop estará desactivado y únicamente podrá pulsar la opción de Play, una vez pulsado este botón quedará desactivado y estará activo el botón de stop, y así simultáneamente. Estas operaciones se realizan mediante jQuery.

Si la comprobación del *token* es satisfactoria se mostrara la clave en la misma página. Como se puede ver en la Fig. 4.15.

**Fig. 4.15 Ticket generado**

También podemos ver que aparece un calendario con la hora, para poder notificar el usuario de la hora límite de la que dispone para utilizar la clave, que en este caso es de 30 minutos.

# 5 Conclusiones y líneas futuras

Una vez finalizado nuestro proyecto, vemos que hemos conseguido el principal objetivo que nos habíamos propuesto que era el de recuperar el concepto de jukebox para una época actual. Nuestro Nukebox es finalmente un nuevo jukebox que, con una tecnología totalmente diferente y más moderna, mantiene la esencia y los puntos positivos que aportaba el jukebox original. De hecho, creemos que podría ser posible su implementación en un entorno real.

El Nukebox se basa en una aplicación web y la interacción con éste se realiza a través de un dispositivo que permita una conexión web, principalmente está dirigido a un uso con smartphones. Para que los usuarios puedan añadir una canción al Nukebox, el administrador/camarero subministra una clave al realizar una consumición o hacer un pago. Los usuarios también pueden ver la lista de canciones disponibles, buscar canciones y consultar la lista de reproducción a tiempo real.

Como comentábamos al inicio del documento, la creación de Nukebox nos ha propuesto nuevos retos a nivel de implementación, posiblemente el más determinante haya sido el control de acceso y ofrecer una solución segura, fiable y sin comprometer la usabilidad para el usuario. Una característica que a priori parece poco relevante, pero que realmente es muy importante es la anonimidad del sistema, ya que al no haber usuarios, no se sabe quién utiliza los tickets y posiblemente ofrece más libertad en la elección de la música.

A nivel técnico el haber tenido que implementar todos los componentes, desde la parte de *core,* donde se encuentra la lógica del sistema, hasta la parte de la interfaz y de diseño, se han tenido que utilizar diferentes tecnologías y software y esto ha hecho que hayamos tenido que integrar estos componentes, para crear un producto final y funcional. Muchos de estos elementos ya los habíamos visto durante la carrera; sin embargo, para otros ha sido necesario un proceso de aprendizaje. El punto positivo de haber tenido que desarrollar el sistema desde cero es el haber adquirido una visión más general del desarrollo de una aplicación con todas sus implicaciones.

Como posibles líneas futuras, se podría intentar integrar nuevas funcionalidades, como, por ejemplo, diferentes categorías de tickets. En el caso de poseer un ticket ordinario, si una persona con un ticket *premium* introdujese una canción, se reproduciría antes de la insertada con un ticket ordinario. La obtención de tickets *premium* podría ser a cambio de un precio mayor o más consumuciones. En esta versión, para que unos tickets tuviesen preferencia respecto a otros, sería necesaria una modificación del algoritmo de gestión de tickets y listas de reproducción. También se podrían integrar otras implementaciones como las que se han nombrado durante el documento. Otra idea que surgió durante el desarrollo del proyecto fue la posibilidad de darle un estilo más *retro* ofreciendo las canciones en una especie de libro o carta donde los usuarios pudiesen consultar las opciones de las que disponen. El escenario para esta idea sería un local elegante, en el que los clientes estarían sentados y se les ofrecería el catálogo y podrían seleccionar música desde su mesa.

# 6 Bibliografía


[1] JSON WEB TOKENS *https://bitbucket.org/b_c/jose4j/wiki/Home*

[2] Spotify API Java Wrapper *https://github.com/thelinmichael/spotify-web-api-java/blob/master/README.md*

[3] Creador Códigos QR *http://www.qrcode-monkey.com/es/#custom-image*

[4] Wurlitzer Catálogo *http://www.jukeboxhistory.info/WurliTzer/PDF/WUR%201100-1080A.pdf*

[5] Definición Jukebox *http://es.wikipedia.org/wiki/Jukebox*

[6] Apache Tomcat *http://tomcat.apache.org/*

[7] Documentación Jersey *https://jersey.java.net/*

[8] REST Wikipedia *http://es.wikipedia.org/wiki/Representational_State_Transfer*

[9] XAMPP *https://www.apachefriends.org/es/index.html*

[10] Hibernate *http://hibernate.org/*

[11] jQuery *https://jquery.com/*

[12] Maven *https://maven.apache.org/*

[13] Configuración Maven http://www.chuidiang.com/java/herramientas/maven.php

[14] GitHub *https://github.com/*

[15] Página oficial de JWT *http://jwt.io/*

[16] JWT tutorial *https://scotch.io/tutorials/the-anatomy-of-a-json-web-token*

[17] Spotify *https://www.spotify.com/es/*

[18] Grooveshark *http://grooveshark.com/*

[19] SoundCloud *https://soundcloud.com/*

[20] Singleton Wikipedia *http://es.wikipedia.org/wiki/Singleton*

[21] Manipulación de etiquetas ID3 *https://github.com/mpatric/mp3agic*

[22] Spotify developers apps *https://developer.spotify.com/my-applications*

[23] Autorizaciones Spotify *https://developer.spotify.com/web-api/authorization-guide/*

[24] Google Design *http://www.google.com/design/*

[25] Materialize CSS *http://materializecss.com/*

[26] jQuery cookie *https://plugins.jquery.com/cookie/*

[27] Funcion keyup jQuery *https://api.jquery.com/keyup/*